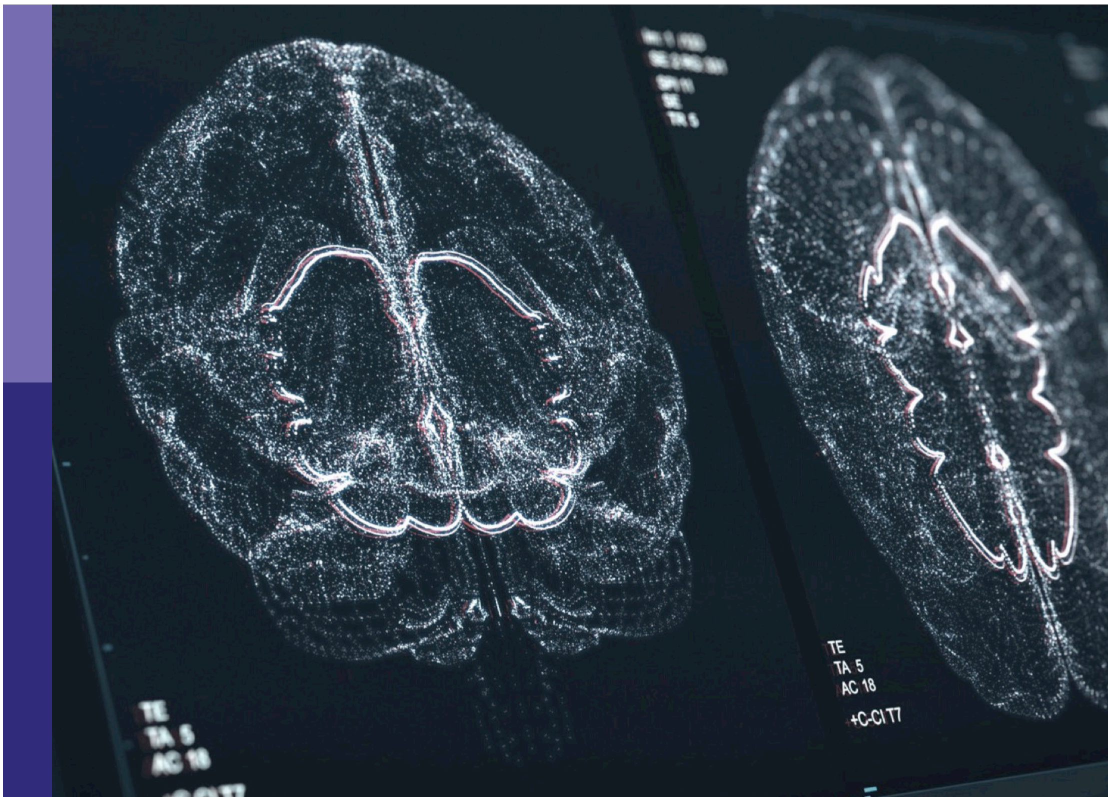
frontiers | Research Topics

Neuroscience, computing, performance, and benchmarks: Why it matters to neuroscience how fast we can compute

Edited by
Felix Schürmann, Omar Awile, James Courtney Knight, Thomas Nowotny,
James B. Aimone and Markus Diesmann

Published in
Frontiers in Neuroinformatics
Frontiers in Neuroscience



ISSN 1664-8714
ISBN 978-2-83252-165-6
DOI 10.3389/978-2-83252-165-6

About Frontiers

Frontiers is more than just an open access publisher of scholarly articles: it is a pioneering approach to the world of academia, radically improving the way scholarly research is managed. The grand vision of Frontiers is a world where all people have an equal opportunity to seek, share and generate knowledge. Frontiers provides immediate and permanent online open access to all its publications, but this alone is not enough to realize our grand goals.

Frontiers journal series

The Frontiers journal series is a multi-tier and interdisciplinary set of open-access, online journals, promising a paradigm shift from the current review, selection and dissemination processes in academic publishing. All Frontiers journals are driven by researchers for researchers; therefore, they constitute a service to the scholarly community. At the same time, the *Frontiers journal series* operates on a revolutionary invention, the tiered publishing system, initially addressing specific communities of scholars, and gradually climbing up to broader public understanding, thus serving the interests of the lay society, too.

Dedication to quality

Each Frontiers article is a landmark of the highest quality, thanks to genuinely collaborative interactions between authors and review editors, who include some of the world's best academicians. Research must be certified by peers before entering a stream of knowledge that may eventually reach the public - and shape society; therefore, Frontiers only applies the most rigorous and unbiased reviews. Frontiers revolutionizes research publishing by freely delivering the most outstanding research, evaluated with no bias from both the academic and social point of view. By applying the most advanced information technologies, Frontiers is catapulting scholarly publishing into a new generation.

What are Frontiers Research Topics?

Frontiers Research Topics are very popular trademarks of the *Frontiers journals series*: they are collections of at least ten articles, all centered on a particular subject. With their unique mix of varied contributions from Original Research to Review Articles, Frontiers Research Topics unify the most influential researchers, the latest key findings and historical advances in a hot research area.

Find out more on how to host your own Frontiers Research Topic or contribute to one as an author by contacting the Frontiers editorial office: frontiersin.org/about/contact

Neuroscience, computing, performance, and benchmarks: Why it matters to neuroscience how fast we can compute

Topic editors

Felix Schürmann — Swiss Federal Institute of Technology Lausanne, Switzerland
Omar Awile — Swiss Federal Institute of Technology Lausanne, Switzerland
James Courtney Knight — University of Sussex, United Kingdom
Thomas Nowotny — University of Sussex, United Kingdom
James B. Aimone — Sandia National Laboratories, United States
Markus Diesmann — Computational and Systems Neuroscience (INM-6), Institute of Neuroscience and Medicine, Julich Research Center, Helmholtz Association of German Research Centres (HZ), Germany

Citation

Schürmann, F., Awile, O., Knight, J. C., Nowotny, T., Aimone, J. B., Diesmann, M., eds. (2023). *Neuroscience, computing, performance, and benchmarks: Why it matters to neuroscience how fast we can compute*. Lausanne: Frontiers Media SA. doi: 10.3389/978-2-83252-165-6

Table of contents

frontiers | Frontiers in Neuroinformatics

TYPE Editorial
PUBLISHED 30 March 2023
DOI 10.3389/fninf.2023.1157418

OPEN ACCESS

EDITED AND REVIEWED BY
Mohammed Fouda,
University of California, Irvine, United States

*CORRESPONDENCE
Felix Schürmann
✉ felix.schuermann@epfl.ch

†These authors have contributed equally to this work

RECEIVED 02 February 2023
ACCEPTED 08 March 2023
PUBLISHED 30 March 2023

CITATION
Aimone JB, Awile O, Diesmann M, Knight JC, Nowotny T and Schürmann F (2023) Editorial: Neuroscience, computing, performance, and benchmarks: Why it matters to neuroscience how fast we can compute. *Front. Neuroinform.* 17:1157418.
doi: 10.3389/fninf.2023.1157418

Editorial: Neuroscience, computing, performance, and benchmarks: Why it matters to neuroscience how fast we can compute

James B. Aimone[1†], Omar Awile[2†], Markus Diesmann[3,4,5†], James C. Knight[6†], Thomas Nowotny[6†] and Felix Schürmann[2*†]

[1]Neural Exploration and Research Laboratory, Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, United States, [2]Blue Brain Project, École Polytechnique Fédérale de Lausanne, Geneva, Switzerland, [3]Institute of Neuroscience and Medicine and Institute for Advanced Simulation and JARA-Institute Brain Structure-Function Relationships, Jülich Research Centre, Jülich, Germany, [4]Department of Physics, Faculty 1, RWTH Aachen University, Aachen, Germany, [5]Department of Psychiatry, Psychotherapy and Psychosomatics, School of Medicine, RWTH Aachen University, Aachen, Germany, [6]School of Engineering and Informatics, University of Sussex, Brighton, United Kingdom

KEYWORDS

performance benchmarking, large-scale simulation, simulation workflows, software sustainability, neuromorphic computing architectures, subcellular neurosimulation, biophysically detailed models

Editorial on the Research Topic
Neuroscience, computing, performance, and benchmarks: Why it matters to neuroscience how fast we can compute

Introduction

At the turn of the millennium the computational neuroscience community realized that neuroscience was in a software crisis: software development was no longer progressing as expected and reproducibility declined. The International Neuroinformatics Coordinating Facility (INCF) was inaugurated in 2007 as an initiative to improve this situation. The INCF has since pursued its mission to help the development of standards and best practices. In a community paper published this very same year, Brette et al. (2007) tried to assess the state of the field and to establish a scientific approach to simulation technology, addressing foundational topics, such as which simulation schemes are best suited for the types of models we see in neuroscience.

In 2015, a Frontiers Research Topic "*Python in neuroscience*" by Muller et al. (2015) triggered and documented a revolution in the neuroscience community, namely in the usage of the scripting language Python as a common language for interfacing with simulation codes and connecting between applications. The review by Einevoll et al. (2019) documented that simulation tools have since further matured and become reliable research instruments used by many scientific groups for their respective questions. Open source and community standard simulators today allow research groups to focus on their scientific questions and leave the details of the computational work to the community of simulator developers.

A parallel development has occurred, which has been barely visible in neuroscientific circles beyond the community of simulator developers: Supercomputers used for large and complex scientific calculations have increased their performance from ~10 TeraFLOPS (10^{13} floating point operations per second) in the early 2000s to above 1 ExaFLOPS (10^{18} floating point operations per second) in the year 2022. This represents a 100,000-fold increase in our computational capabilities, or almost 17 doublings of computational capability in 22 years. Moore's law (the observation that it is economically viable to double the number of transistors in an integrated circuit every other 18–24 months) explains a part of this; our ability and willingness to build and operate physically larger computers, explains another part. It should be clear, however, that such a technological advancement requires software adaptations and under the hood, simulators had to reinvent themselves and change substantially to embrace this technological opportunity. It actually is quite remarkable that—apart from the change in semantics for the parallelization—this has mostly happened without the users knowing.

The current Research Topic was motivated by the wish to assemble an update on the state of neuroscientific software (mostly simulators) in 2022, to assess whether we can see more clearly which scientific questions can (or cannot) be asked due to our increased capability of simulation, and also to anticipate whether and for how long we can expect this increase of computational capabilities to continue.

Larger brain and brain tissue models

A promising advance compared to the state of the field 15 years ago is that we now see an increase in the complexity of network models. Earlier, the balanced random network model composed of a population of excitatory neurons and a population of inhibitory neurons was dominating the literature and few studies reached beyond it. Today, biologically much more realistic network models are in widespread use and have become the new *de facto* standard (Albers et al.; Tiddia et al.; Awile et al.; Borges et al.). These newer models represent the anatomy of the local circuitry of the mammalian cortex at full scale, meaning with all the neurons and synapses. As a consequence, neuron and synapse numbers have increased by an order of magnitude compared to earlier models. The ability to simulate at full scale is decisive because this removes all uncertainties on the scaling of emerging network phenomena with network size which have plagued and occupied theoreticians for a long time (van Albada et al., 2015).

Expansion to the subcellular realm

Most articles in this collection concentrate on describing models developed at the level of neurons and synapses. However, some articles also show how our advances in computing and simulation technology can be used to extend our modeling and simulation capability to the membrane and subcellular biochemical realm. Awile et al. show how subcellular dynamics can be integrated into NEURON simulations. The works of Chen et al. and McDougal et al. enable neuroscientists to study the biophysics of synaptic plasticity and the processes in the spine in detail. As generally accepted models of plastic processes have not yet been established on a phenomenological level, the capability to simulate on the level of subcellular processes is of high relevance.

The role of simulators and workflows

The number of codes targeting the same level of description has decreased somewhat and remaining codes like NEURON (Awile et al.) and NEST (Albers et al.; Pronold et al.) have increasingly embraced and advanced community-based development models and incorporated ideas of the emerging field of research software engineering (RSE). At the same time, it is remarkable that after 15 years of intense research the seemingly fundamental question of whether an event-driven or a clock-driven approach to the simulation of spiking neuronal networks is more efficient, does not seem to have found a consensus (Mo and Tao; Hanuschkin et al., 2010; Krishnan et al., 2018). A reason for this could of course be that there is simply no general answer for any model and hardware, and that in practice simulation codes such as NEURON and NEST employ hybrid approaches.

Furthermore, various variants of language interfaces were developed for the traditional simulation codes (Borges et al.; Herbers et al.). Also new simulation codes were developed expressing network models entirely in Python or implementing code generators for performance critical sections (Dinkelbach et al.; Alevi et al.). Of similar importance to the advances of individual tools is the progress in the digitalization of scientific workflows (Albers et al.; Awile et al.; Feldotto et al.; Herbers et al.) and the observation that not only the source codes but also executable model descriptions of simulation engines are available in publicly curated repositories.

Keeping innovations around—Sustainability of scientific software

Software codes that have been around for 15 years, are still in widespread use by the community today. Neuroscience must therefore acknowledge, as other scientific fields already have, that scientific software can easily have life spans of 40 years or more. Sustainability and portability are consequently of high relevance for software tools that serve a whole community rather than a specific scientific goal as showcased in Chen et al. and Awile et al.. While often new features or increased performance (especially in the case of simulators) are the milestones of such projects, the authors observed that a focus on software sustainability can be an important driver for innovations. Both publications show how the modernization of complex scientific software can be made more tractable by first focusing on putting in place a robust continuous integration, testing, and documentation workflow. As the software developed in the field is becoming more complex to satisfy the scientific needs (e.g., supporting multiple numerical methods, multiphysics simulations, and heterogenous hardware platforms), the implementation of software modularity and composability is concurrently becoming increasingly important.

These methodologies feature prominently in Feldotto et al.. The authors focus here on container technologies to enable complex software setups and workflows for embodied simulations of spiking neural networks.

If simulator engines are on track, how about analysis packages?

Only one paper in this series discusses the performance of a data analytics problem (Porrmann et al.). This may reflect the possibility that the availability of HPC methods is not the most pressing problem in the analysis of neuroscientific data. There is certainly considerable activity in processing pipelines for neuroimaging, but this field finds other forums (Halchenko et al., 2021). Maybe the discrepancy also reflects the fact that in the research field concerned with the spiking activity of neuronal networks, researchers doing simulations have always been somewhat advanced in embracing new hardware and software technologies compared to those involved in analysis.

Embracing the course of computing architecture evolution

A thread running through many of the articles in this collection is how to make the best of the currently available but rapidly changing hardware systems. Since clock frequencies for processors flattened out in the mid-2000s, processor architectures have become progressively more parallel. This applies to latency-optimized CPUs which have become moderately parallel (<100 superscalar cores/CPU) as well as GPUs (>1000s of simple cores/GPU). It is heartening to see that the community is embracing this opportunity and challenge. Alevi et al. present new software for exploiting NVIDIA GPU hardware to accelerate simulation with the popular Brian simulator (Stimberg et al., 2019), complementing the existing Brian2GeNN software (Stimberg et al., 2020). Awile et al. show how code generation can be used to run the NEURON simulator on GPUs. In a similar vein, Tiddia et al. present work on how to efficiently run a large spiking neural network model on a GPU cluster and Dinkelbach et al. describe work on one specific aspect of efficient simulations of spiking neural networks on GPU hardware in their ANNarchy simulation software. Ladd et al. furthermore present an evolutionary algorithm able to run on GPUs that accelerates the building of multi-compartment neuron models. Challenges of how to handle massive parallelism and distributed computing also arise in the context of classical HPC clusters, and Pronold et al. describe how one key bottleneck can be overcome.

Emerging computing architectures

The unsure future of CMOS scaling will present the neural simulation community with an even broader set of architectures beyond CPUs and GPUs. There is an increasing trend toward more specialized components, particularly those that enable artificial intelligence applications such as artificial neural networks (Reed et al., 2022). We hope that such specialization may also enable simulations of biological neural networks without too many adaptations. Looking beyond ANN accelerators, it is also reasonable to expect to see even more diversity through platforms, such as neuromorphic hardware, obtaining widespread use in HPC systems, particularly since they are proving suitable for conventional computing applications (Aimone et al., 2022). Beyond exploiting specific characteristics of biological neural networks, today's neuromorphic computing systems such as SpiNNaker, BrainScales, and Loihi attempt an integration at scale. As a result they enable complex models to be programmed, with biologically fit neurons shown to be realizable on Intel Loihi (Dey and Dimitrov), BrainScaleS-2 (Müller et al.), and SpiNNaker (Peres and Rhodes; Ward and Rhodes).

Rethinking the underlying algorithms

Not only is the computational neuroscience community embracing the challenges of rapidly developing processor architectures but it is also capitalizing on the additional computing power to explore different simulation algorithms and schemes. For instance, Osborne and de Kamps extend the population density technique for neural network simulations to higher-dimensional neuron models and Chen et al. improve on memory efficiency and simulation speed for detailed molecular simulations of neurons. Similarly, McDougal et al. describe the efficient simulation of 3D reaction-diffusion processes in neuronal networks extending on more traditional 1D simulations for dendrites and axons.

Time

While GPUs and large, massively-parallel HPC clusters were not built for the purpose of brain simulations, the inherently parallel nature of how brains operate, makes such systems reasonably well-suited to simulating brain models. However, we must not forget that while computers have become more powerful (i.e., they are able to do more things in parallel), they have not become much faster—ever since frequency scaling (Dennard Scaling) had to stop due to limits in how much heat can be dissipated from an integrated circuit. This puts in question certain scientific problems which require the simulation of long time durations such as needed, for example, in plasticity studies, or extensive training runs in the emerging field of neuro-inspired machine learning. While algorithmic innovations may help us to rethink the supposedly critical sequential paths of computational problems (e.g., AlphaFold applied these to the problem of protein folding), an alternative approach may be the acceleration factors that can be achieved from mapping the computational problem to physical instantiations of the computation such as done by Brainscales-2 (Müller et al.) or as indicated by Trensch and Morrison through spatial computations using SoCs and FPGAs.

Benchmarking as the compass

As the diversity of hardware architectures grows, it will be increasingly important to quantify the suitability of those platforms for actual brain tissue model simulations. It is thus necessary to develop benchmarks (models) and benchmarking (measuring) to objectively quantify the performance of such platforms. While HPC systems have often varied in components and configurations, there have long been standards for linear algebra such as Linpack that allowed rigorous, even if not perfect, comparisons. Herbers et al., Albers et al., and Schmitt et al. make a step toward generic and simulator agnostic frameworks for benchmarking and simulation. However, as we look toward a future with specialized neural network accelerators and general purpose von Neumann systems, the challenge in benchmarking will become more pronounced. This is especially a challenge with neuromorphic hardware, which is both rapidly evolving and exhibits a diversity of approaches with mixed advantages in speed and energy, resulting in a complex basis for evaluation (Trensch and Morrison; Müller et al.). Furthermore, the concept of a FLOP or matrix multiply operation is less meaningful in spiking neural simulations which may be event-driven and sparse. One proposed approach is to develop concrete benchmark spiking networks that can be tested on both neuromorphic systems and conventional processors, which is proving useful in obtaining an early assessment of the relative efficiency of neuromorphic systems compared to both conventional systems and real brains (Ostrau et al.; Kurth et al., 2022).

Author contributions

All authors contributed equally to the editing of the Research Topic. All authors contributed equally to the writing of the article and approved the submitted version.

Funding

JA was supported by the Collaborative Research in Computational Neuroscience (CRCNS) program through the US Department of Energy's Advanced Scientific Computing Research office. The work of OA and FS is supported by funding to the Blue Brain Project, a research center of the École Polytechnique Fédérale de Lausanne (EPFL), from the Swiss government's ETH Board of the Swiss Federal Institutes of Technology. JK was funded by the EPSRC (Grant Number EP/V052241/1). MD and TN are funded by the European Union's Horizon 2020 research and innovation programme under Grant Agreement 945539 (HBP SGA3). Finally, MD's work was supported by HiRSE_PS, the Helmholtz Platform for Research Software Engineering—Preparatory Study, an innovation pool project of the Helmholtz Association.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

Aimone, J. B., Date, P., Fonseca-Guerra, G. A., Hamilton, K. E., Henke, K., Kay, B., et al. (2022). A review of non-cognitive applications for neuromorphic computing. *Neuromorphic Comput. Eng.* 2, 032003. doi: 10.1088/2634-4386/ac889c

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398. doi: 10.1007/s10827-007-0038-6

Einevoll, G. T., Destexhe, A., Diesmann, M., Grün, S., Jirsa, V., de Kamps, M., et al. (2019). The scientific case for brain simulations. *Neuron* 102, 735–744. doi: 10.1016/j.neuron.2019.03.027

Halchenko, Y. O., Meyer, K., Poldrack, B., Solanky, D. S., Wagner, A. S., Gors, J., et al. (2021). DataLad: distributed system for joint management of code, data, and their relationship. *J. Open Source Softw.* 6, 3262. doi: 10.21105/joss.03262

Hanuschkin, A., Kunkel, S., Helias, M., Morrison, A., and Diesmann, M. (2010). A general and efficient method for incorporating precise spike times in globally time-driven simulations. *Front. Neuroinform.* 4, 113. doi: 10.3389/fninf.2010.00113

Krishnan, J., Porta Mana, P., Helias, M., Diesmann, M., and Di Napoli, E. (2018). Perfect detection of spikes in the linear sub-threshold dynamics of point neurons. *Front. Neuroinform.* 11, 75. doi: 10.3389/fninf.2017.00075

Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2022). Sub-realtime simulation of a neuronal network of natural density. *Neuromorphic Comput. Eng.* 2, 021001. doi: 10.1088/2634-4386/ac55fc

Muller, E., Bednar, J. A., Diesmann, M., Gewaltig, M.-O., Hines, M., and Davison, A. P. (2015). Python in neuroscience. *Front. Neuroinform.* 9, 11. doi: 10.3389/fninf.2015.00011

Reed, D., Gannon, D., and Dongarra, J. (2022). Reinventing high performance computing: challenges and opportunities. *arXiv [Preprint] arXiv:2203.02544*. doi: 10.48550/arXiv.2203.02544

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8, e47314. doi: 10.7554/eLife.47314.028

Stimberg, M., Goodman, D. F. M., and Nowotny, T. (2020). Brian2GeNN: accelerating spiking neural network simulations with graphics hardware. *Sci. Rep.* 10, 410. doi: 10.1038/s41598-019-54957-7

van Albada, S. J., Helias, M., and Diesmann, M. (2015). Scalability of asynchronous networks is limited by one-to-one mapping between effective connectivity and correlations. *PLoS Comput. Biol.* 11, e1004490. doi: 10.1371/journal.pcbi.1004490

TECHNOLOGY AND CODE
published: 16 September 2021
doi: 10.3389/fninf.2021.723406

Acceleration of the SPADE Method Using a Custom-Tailored FP-Growth Implementation

Florian Porrmann[1], Sarah Pilz[1], Alessandra Stella[2,3], Alexander Kleinjohann[2,3], Michael Denker[2], Jens Hagemeyer[1] and Ulrich Rückert[1]*

[1] Cognitronics and Sensor Systems, CITEC, Bielefeld University, Bielefeld, Germany, [2] Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6) and JARA-Institute Brain Structure-Function Relationships (INM-10), Jülich Research Center, Jülich, Germany, [3] RWTH Aachen University, Aachen, Germany

The *SPADE* (spatio-temporal **S**pike **PA**ttern **D**etection and **E**valuation) method was developed to find reoccurring spatio-temporal patterns in neuronal spike activity (parallel spike trains). However, depending on the number of spike trains and the length of recording, this method can exhibit long runtimes. Based on a realistic benchmark data set, we identified that the combination of pattern mining (using the *FP-Growth* algorithm) and the result filtering account for 85–90% of the method's total runtime. Therefore, in this paper, we propose a customized *FP-Growth* implementation tailored to the requirements of *SPADE*, which significantly accelerates pattern mining and result filtering. Our version allows for parallel and distributed execution, and due to the improvements made, an execution on heterogeneous and low-power embedded devices is now also possible. The implementation has been evaluated using a traditional workstation based on an Intel Broadwell Xeon E5-1650 v4 as a baseline. Furthermore, the heterogeneous microserver platform RECS|Box has been used for evaluating the implementation on two HiSilicon Hi1616 (Kunpeng 916), an Intel Coffee Lake-ER Xeon E-2276ME, an Intel Broadwell Xeon D-D1577, and three NVIDIA Tegra devices (Jetson AGX Xavier, Jetson Xavier NX, and Jetson TX2). Depending on the platform, our implementation is between 27 and 200 times faster than the original implementation. At the same time, the energy consumption was reduced by up to two orders of magnitude.

Keywords: FP-growth, pattern mining, spike train analysis, embedded devices, performance optimization, low power, parallel and distributed computing, heterogeneous computing

OPEN ACCESS

Edited by:
Felix Schürmann,
École Polytechnique Fédérale de Lausanne, Switzerland

Reviewed by:
Marcel Stimberg,
Sorbonne Université, France
Fernando S. Borges,
Federal University of ABC, Brazil
Bruno Golosio,
University of Cagliari, Italy

****Correspondence:***
Florian Porrmann
fporrmann@techfak.uni-bielefeld.de

Received: *10 June 2021*
Accepted: *16 August 2021*
Published: *16 September 2021*

Citation:
Porrmann F, Pilz S, Stella A, Kleinjohann A, Denker M, Hagemeyer J and Rückert U (2021) Acceleration of the SPADE Method Using a Custom-Tailored FP-Growth Implementation. Front. Neuroinform. 15:723406. doi: 10.3389/fninf.2021.723406

1. INTRODUCTION

Increasing evidence from neuroscience suggests that in order to understand the principles of information processing in the brain, it is important to study not only the activity of isolated neurons in response to the environment and behavior, but also to investigate the concerted dynamics of neuronal networks as a whole. With the rapid advancement of electrophysiological recording techniques in the recent decades, scientists are now able to monitor the spiking activity of individual nerve cells in large neuronal populations, enabling the investigation of the dynamics of hundreds of neurons recorded in parallel (e.g., Jun et al., 2017; Brochier et al., 2018; Steinmetz et al., 2018; Juavinett et al., 2019; Chen et al., 2020). The cell assembly hypothesis (Hebb, 1949) postulates that information is represented by interactions within groups of neurons. Signatures of assemblies in the observed dynamics are groups of synchronously active neurons (e.g., Harris, 2005), or spatio-temporal sequences of neuronal activation. Efficient methods to detect and characterize this coordinated activity are in high demand (Quaglio et al., 2018). Such methods need to deal with challenges related to the highly non-stationary spike time series and the statistical complexity of high-dimensional activity patterns, since the number of possible patterns exponentially increases with the number of observed neurons. Several complementary methods have been developed and calibrated in the past (e.g., Grün et al., 2002a,b; Pipa et al., 2008; Gerstein et al., 2012; Lopes-dos Santos et al., 2013; Torre et al., 2013; Russo and Durstewitz, 2017; Diana et al., 2019; Watanabe et al., 2019; Williams et al., 2020). While the nature and underlying assumptions of these approaches differ, they share the need to scale in runtime performance as the number of observed neurons or the length of the recording increases. This holds true, in particular, with an increasing interest to employ such techniques to analyze and validate simulations of large-scale models of neuronal networks (cf., e.g., Trensch et al., 2018; Gutzen et al., 2018) that easily exceed the volume of available experimental data.

One of the state-of-the-art methods to detect spatio-temporal patterns in large sets of parallel spike trains (Quaglio et al., 2018) is *SPADE*[1], originally proposed by Torre et al. (2013). The method is based on frequent itemset mining (Agrawal et al., 1993). The existing Python implementation of the SPADE method in the Electrophysiology Analysis Toolkit[2] (Elephant; RRID:SCR_003833; Denker et al., 2018) is able to analyze current data sets of moderate size at relatively high computational cost, making the availability of distributed compute resources mandatory and discouraging interactive exploratory analyses. In this work, we put forward an accelerated version of *SPADE* by optimizing the underlying pattern mining flow using a custom-tailored *FP-Growth*[3] (Han et al., 2000) implementation to address the need for enhanced scalability and thereby increase the range of data sets for which the method is practically applicable. Additionally, we show that our optimizations enable the execution of *SPADE* on heterogeneous and low-power embedded devices, which is significantly more energy-efficient than the execution on a modern workstation.

Previously, the focus of development efforts related to *SPADE* concentrated on improving or extending the capabilities of the method, which makes this work the first to address performance and energy efficiency. After Torre et al. (2013) developed the concepts for the statistical evaluation of synchronous spike patterns through *FP-Growth*, Yegenoglu et al. (2016) introduced a technique to identify *spatio-temporal patterns* in *massively parallel spike trains* using formal concept analysis (FCA; Ganter and Wille, 1999), extending the detection of patterns from synchronous to spike patterns with delays. In 2017, these approaches were combined by Quaglio et al. (2017). Since the FCA implementation used by Yegenoglu et al. (2016) required significantly more time and computational power, it was replaced by *FP-Growth*. Stella et al. (2019) introduced an extension to *SPADE*, called *3d-SPADE*, which also accounts for the temporal extent of patterns with delays in the significance estimation. The *SPADE* method is explained in more detail in section 2.3.

On a similar path, the *FP-Growth* algorithm used in *SPADE* (Picado-Muiño et al., 2013) was subject to numerous extensions and modifications from a methodological perspective. Picado Muiño et al. (2012) and Borgelt and Picado-Muiño (2013) introduced a version of FP-Growth in continuous time called CoCoNAD, which avoids the need to discretize the input spike train. CoCoNAD was used for benchmarking of artificial data (Picado-Muiño et al., 2013) and analyses of electrophysiological experiments (Torre et al., 2016). Furthermore, CoCoNAD was extended in Borgelt et al. (2015) to account for patterns with selective neuronal participation, or *fuzzy patterns*. When extending the *SPADE* analysis to delayed patterns, it was necessary to resort back to discretizing data (Quaglio et al., 2017).

In contrast to *SPADE*, where performance improvements were never the main focus, several publications focused primarily on improving and accelerating *FP-Growth* through, e.g., parallel or distributed computing. A detailed explanation of the pattern mining and *FP-Growth* related terms used in this section can be found in sections 2.1, 2.2. The first parallel *FP-Growth* variation, called *MLFPT*, was developed by Zaiane et al. (2001). It divides the input database across all available processors and creates a local FP-tree[4], the data structure used by *FP-Growth*, on each. Afterward, a global header table, a linked list used by *FP-Growth*, is created, linking the different items to their occurrences in local FP-trees. Each processor is assigned an equal portion of the entire itemset on which it performs the pattern mining step.

Chen et al. (2009) developed a parallel *FP-Growth* variant, called *Grided FP-Growth* (*GFP-Growth*), designed to be used on large compute clusters. The main difference to the original *FP-Growth* is that they skip the FP-tree construction by directly mining the conditional pattern bases, sub-databases, created from the FP-tree, using the projection method proposed in Bin and Li (2008). This allows them to split the mining process into independent groups, which can be executed in parallel on any number of compute nodes.

Li et al. (2008) proposed a massively parallel and distributed implementation, called *PFP-Growth*. Their approach is based on *MapReduce* (Dean and Ghemawat, 2004), a programming model for large-scale distributed computing. By dividing the input data into independent *groups*, they can distribute the workload across massive compute clusters without any computational dependencies between the different nodes. In their tests, they achieved nearly linear performance scaling when executing their implementation with a data set consisting of 802,939 web pages on between 100 and 2,500 computers. Zhou et al. (2010) improved *PFP-Growth* by adding load balance features, resulting in a new version they called *BPFP-Growth*. Through proper load balancing during the parallel execution of the pattern mining process, a speedup of 1.5 over the original *PFP-Growth* implementation was achieved. Xia et al. (2018) improved the performance of *PFP-Growth* when processing a massive number

[1] **S**pike **PA**ttern **D**etection and **E**valuation.

[2] http://python-elephant.org

[3] **F**requent **P**attern Growth.

[4] **F**requent **P**attern Tree.

of small files on a Hadoop compute platform, resulting in the creation of *MR-PFP-Growth*. Shi et al. (2017) proposed a distributed *FP-Growth* algorithm, using Apache Spark[5] called *DFPS*, which achieved a significant speedup over *PFP-Growth*.

The previously introduced parallel implementations for *FP-Growth* are designed for use with large data sets, containing a vast number of transactions (1–100 million) and items (more than 10 million), and target large-scale compute clusters with up to several thousand nodes. The algorithms were developed to make pattern mining on these data sets possible in a reasonable time frame. In addition, the use of such massive compute clusters requires good load balancing and fault tolerance so that the computation does not have to be restarted in case a node fails. In contrast, the data sets used with *SPADE* are relatively small, consisting of only a few thousand transactions with, on average, two to three thousand items. Furthermore, while the cited implementations target the parallelization of the baseline *FP-Growth* algorithm, the version developed in this work is custom-tailored for the use in the *SPADE* method. As such, the improved implementation presented here, based around a rather naive approach to parallel and distributed computing of *FP-Growth*, is more suitable for the given problem, as it does not inhibit the portability and can be easily disabled if required. One of the main differences between our implementation and the ones described previously is based on the *filter function*, a part of the SPADE algorithm which significantly reduces the number of patterns mined. It enables us to pursue an implementation approach that would not be possible under normal conditions. Therefore, using code optimizations and minimized overhead, we managed to achieve high performance and high energy efficiency using server- and distributed embedded processors.

The main contributions of this work are as follows.

1. We propose an optimized *FP-Growth* implementation, custom-tailored to the problem presented by the *SPADE* method. A significant performance increase was achieved by incorporating the pattern filtering function used by *SPADE* into the pattern mining. Furthermore, we have implemented parallelization and distributed computing concepts in our customized version of *FP-Growth* to take full advantage of the available hardware.
2. Moving the pattern filtering task into *FP-Growth* resulted in a considerable decrease in memory consumption, to the point where execution on low-power embedded devices is now possible.
3. We evaluated our implementation's performance and showed that a significant performance increase could be achieved with our optimizations compared to the original.

The remainder of this article is structured as follows. In section 2, we first provide an introduction to pattern mining. Subsequently, we introduce the *SPADE* method, in particular, its core algorithm, *FP-Growth*. We identify the bottlenecks of the current implementation and present our optimizations in terms of efficient data handling, memory optimizations, and parallelizations. In section 3, we compare the runtime, energy efficiency, and memory consumption of the original implementation to our optimized solution. For this purpose, we run the optimized version on several different platforms. We demonstrate that our improvements can achieve up to 280 times higher energy efficiency in addition to an acceleration by a factor of up to 200. Finally, in section 4, we discuss the impact of our optimizations on *SPADE's* overall runtime and energy efficiency and present possible future research to improve its performance further.

2. METHOD

In this section, we propose an optimization to significantly accelerate the *SPADE* method used to detect spike patterns in massively parallel spike trains. Therefore, we first discuss the method itself, focusing on the *FP-Growth* algorithm used to identify frequent spike patterns. Afterward, we present our version of *FP-Growth*, optimized for use in the *SPADE* pipeline. By integrating the result filtering step, that had previously been performed separately, directly into the pattern mining process, we achieve a significant performance improvement.

2.1. Introduction to Frequent Pattern Mining

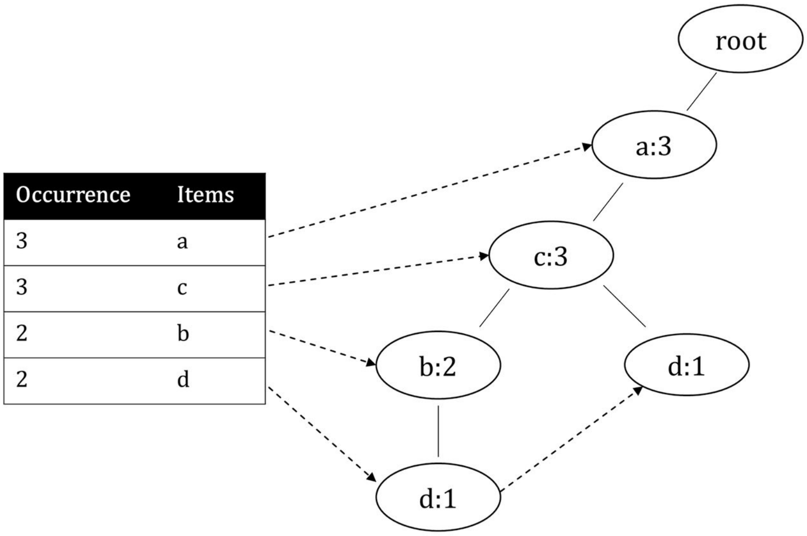
In this paragraph, we first give a short introduction into frequent pattern mining and its terminology. Afterward, these concepts are showcased in a small example. Frequent pattern mining refers to the task of identifying reoccurring patterns within large databases. Agrawal et al. (1993) initially introduced this concept to find patterns in large databases of customer transactions, e.g., from large stores or businesses. Such patterns can, for instance, be used to optimize the product placement in a supermarket, as they provide information about products commonly bought together. In the following, the terms used in conjunction with pattern mining and the concept itself are explained in more detail. Most terms reflect the method's origin in purchase analysis, i.e., *item* and *transaction*. Given an itemset I, a transaction T is defined as a subset of items from I. A transaction database D is defined as a collection of transactions. A frequent pattern (itemset) is a combination of items within a transaction that reoccurs in one or more different transactions of the same database. The occurrence count of a pattern is called *support* S. There are different ways to limit the number of patterns produced, e.g., by setting a minimum pattern length, i.e., that a pattern has to contain at least n-items to be counted or by specifying a minimum occurrence count, i.e., that a pattern has to occur at least m-times to be counted. Additionally, there are two unique categories of frequent patterns: closed frequent patterns and maximal frequent patterns. A pattern P is considered closed when there exists no superset, i.e., a pattern containing P with the same support S as P. Similarly, a pattern P is regarded as a maximal frequent pattern if it has no frequent superset, i.e., there exists no frequent pattern containing P.

The following example showcases the concepts defined above. A pattern P is depicted in the form $P = \{i_1, ..., i_n\}\,(S)$ with $i \in I$. Given the itemset $I = \left\{a, b, c, d\right\}$ and database $D = \{T_1, T_2, T_3\}$

[5] http://spark.apache.org/

$T_1 = \{a, b, c\}$
$T_2 = \{a, c, d\}$
$T_3 = \{a, b, c, d\}$

$n \geq 1$ $s > 0$	$n \geq 2$ $s > 0$	$n \geq 2$ $s \geq 2$
a(3)	a,c(3)	a,c(3)
c(3)	a,b(2)	a,b(2)
b(2)	a,d(2)	a,d(2)
d(2)	b,c(2)	b,c(2)
a,c(3)	c,d(2)	c,d(2)
a,b(2)	b,d(1)	a,b,c(2)
a,d(2)	a,b,c(2)	a,c,d(2)
b,c(2)	a,c,d(2)	
c,d(2)	a,b,d(1)	
b,d(1)	b,c,d(1)	
a,b,c(2)	a,b,c,d(1)	
a,c,d(2)		
a,b,d(1)		
b,c,d(1)		
a,b,c,d(1)		



Occurrence	Items
3	a
3	c
2	b
2	d

FIGURE 1 | Left: All patterns from the pattern mining example presented in section 2.1. **Right**: The header table and FP-tree created from the same example transactions.

where the transactions are $T_1 = \{a, b, c\}$, $T_2 = \{a, c, d\}$ and $T_3 = \{a, b, c, d\}$, without any limitations, 15 frequent patterns can be found in D, as shown in **Figure 1**. Once the minimum pattern length n is increased to 2, only 11 patterns remain. If now also a minimum occurrence s of 2 is specified, the amount of patterns is reduced to 7. Of these patterns, $a, c(3)$, $a, c, d(2)$ and $a, b, c(2)$ are closed and $a, c, d(2)$ and $a, b, c(2)$ are maximal frequent patterns.

2.2. FP-Growth-Based Pattern Mining

The *FP-Growth* algorithm is a highly efficient method to mine frequent patterns from a transaction database. Other well-known algorithms for frequent pattern mining, such as the *Eclat* (Zaki, 2000) or the *Apriori* (Agrawal and Srikant, 1994) algorithm, perform this task through candidate generation, which has the drawback that it can consume a large amount of memory. *FP-Growth* builds a so-called FP-tree, which contains all information about the relations between different items in all transactions. By traversing this tree and recursively creating so-called conditional sub-trees, it is possible to find all frequent patterns without candidate generation, while also requiring significantly less memory. The algorithm operates as follows. First, it iterates over the entire database to store all unique items and their occurrence in a list L, sorted by occurrence. Afterward, all items with an occurrence count below the threshold can directly be discarded. The same applies to transactions that have fewer items than required for the minimum pattern length. Next, the items in each transaction are sorted in descending order based on their occurrence. Subsequently, the actual FP-tree is created by first creating a root-node and sequentially inserting the transactions into the tree. Starting at the root node, for the first item of the current transaction, either a new node is created (if no node for this item exists) or the counter is incremented (if a node exists). This process is repeated for each item in the transaction, always using the newly created node as a base. Once the current transaction has been fully processed, the same process is done for the next transaction, starting once again at the root node. This is repeated until all transactions have been processed and the FP-tree is completed. In parallel to the FP-tree, a header table is built, linking each unique item to its first occurrence in the tree, which then, in turn, links to the second occurrence, and so on. These links are known as *node-links*. The items' order is defined by their occurrence and is equal to the order in the previously created list L. The header table and the FP-tree for the example presented in section 2.1 are depicted in **Figure 1**.

After the FP-tree and the header table are created, the frequent patterns are mined. This is done by iterating over the header table and evaluating the *node-link* for the respective item i. If i only occurs once within the tree, the frequent patterns can be determined by creating all combinations of i with its preceding nodes. Should i occur multiple times in the tree, the preceding nodes form the so-called *conditional pattern base* of i, from which a sub-FP-tree is created, called *conditional FP-tree* of i. The mining process is recursively performed on the conditional tree until all patterns have been mined. Once all patterns for a header table entry have been computed, the same process is repeated for the next entry until the entire header table has been processed, and therefore, all frequent patterns have been mined. It should be noted that there exist no dependencies between the different header table iterations, meaning that they could, in theory, all be performed in parallel. The compute complexity of the *FP-Growth* algorithm depends on the number of items in the header table and the maximum depth of the FP-tree, i.e., again, the number of items. Let n be the number of items. Therefore, the complexity of *FP-Growth* is $O(n^2)$ (Wicaksono et al., 2020).

2.3. Spike Activity Analysis Using the SPADE Method

The *SPADE* method was introduced by Torre et al. (2013) and has since been continuously advanced and improved (Quaglio et al., 2017; Stella et al., 2019). Using *SPADE*, it is possible to detect spatio-temporal spike patterns in parallel spike trains. Spatio-temporal spike patterns are precisely reoccurring delayed sequences of spikes across neurons. They are defined by the times of their occurrences, by the neurons involved, and by the temporal delays between spikes. In order to detect spatio-temporal patterns, *SPADE* employs frequent itemset mining to find reoccurring candidate patterns in the parallel spike train data given as input. The mined patterns are then evaluated for significance by Monte Carlo testing. First, different realizations of surrogate data are generated, which are mined using *FP-Growth* similarly to the original data. Second, patterns detected in surrogates are grouped by shared characteristics, i.e., their number of spikes, duration in time, and number of occurrences, and a p-value is estimated for each group. In a third step, candidate patterns are selected according to their p-value, correcting for multiple testing. Finally, the set of statistically significant patterns is further reduced by conditionally testing each pair of patterns with common spikes. Within this study, we concentrate on the mining of frequent patterns without taking into consideration the statistical tests.

In terms of required computation, between 85 and 90% of *SPADE's* runtime is spent detecting spike patterns within the parallel spike train data fed into the method. For this, first, the spike trains for all N neurons are discretized into *time bins* by segmenting time into small intervals with a bin size b of typically a few milliseconds and mapping each spike onto one bin. If two spikes of the same neuron fall into the same bin, they are considered as one spike. This binning technique accounts for small temporal variability that could prevent patterns from being detected. As a next step, in order to detect delayed spike patterns, a sliding window with a length of w bins (duration equal to $w \cdot b$) is shifted bin by bin over the data (**Figure 2A**). The quantity ω coincides with the maximal allowed duration of a pattern, calculated as the difference in bins between the first and the last spike. Each window is first provided in a matrix representation with the neurons mapped to the rows and the bins to the columns. For further computation, the matrix is converted to a row vector (cf., **Figure 2B**). For each element within the window, its position in the vector is calculated as $n \cdot w + B$, where n is the neuron id (row), w the length of the window, and B the bin id (column). We use ω to denote the index of the window positions (cf. **Figures 2A,C**). This row vector equals a transaction, as described in section 2.1. The vectors of all windows compose the input data for *FP-Growth* (see section 2.2), the pattern mining algorithm employed by *SPADE*. **Figure 2C** shows a highly simplified version of the pattern mining process, and **Figure 2D** depicts the spike trains fed into *SPADE* with the found pattern highlighted in green.

Since typically, a large number of neurons is involved, only closed frequent patterns are kept, while non-closed patterns are rejected (Torre et al., 2013). After the mining is done, the output can still contain repeating patterns caused by the shifting window. A pattern with a duration shorter than the shifting window size will reoccur several times in different windows. Therefore, only those patterns whose first spike occurs in the first bin are kept, and all others are discarded. This can be quickly done, assuming that P is the position of the pattern within the row vector by checking if $P \mod w = 0$ for any of the occurrences of the pattern. Furthermore, a pattern should also contain a minimum number of individual neurons and only occur a maximum number of times to be considered relevant. Patterns with fewer individual neurons or too many occurrences are therefore also ignored. Due to the use of the window and binning, the same neuron can be part of a pattern multiple times, therefore, it is checked, that at least a minimum number of individual neurons are part of the pattern. This entire filtering step is done by applying a custom *filter function* (cf. **Algorithm 1**) to all found patterns, removing a significant portion of them. Of the three filter criteria mentioned, most patterns are discarded when performing the first bin check. Thereby, a large part (typically, between 90 and 100%) of all found patterns are removed. While *SPADE* is in most parts implemented using Python, for the *FP-Growth* algorithm, the highly optimized C-implementation *PyFIM*[6], developed by Christian Borgelt, is used (Borgelt and Picado-Muiño, 2013; Picado-Muiño et al., 2013).

2.4. Identification of Bottlenecks

As mentioned in section 2.3, one of the most time-consuming parts of the *SPADE* method consists of the closed frequent pattern mining, using the *FP-Growth* algorithm, and the result filtering. Therefore, we will first analyze the current implementations of the aforementioned parts and identify their respective bottlenecks. Subsequently, in section 2.5, we will present our optimized version, which achieves a significant speedup compared to the original.

Figure 3 illustrates the current implementation of *SPADE*'s pattern mining flow and its pre-processing steps, on the example of the *movement_PGHF* data set, which is also used during the evaluation (cf., section 3.1). As described in section 2.3, the input spike data is first discretized using binning and the sliding window. Afterward, *FP-Growth* is applied to analyze the resulting row vectors, and all closed patterns are identified. After filtering, only relevant patterns remain and are further processed. For this data set, from 3 MB of spike input data, 200 MB of row vectors are generated and transferred to *FP-Growth*. Depending on the

[6] https://borgelt.net/pyfim.html

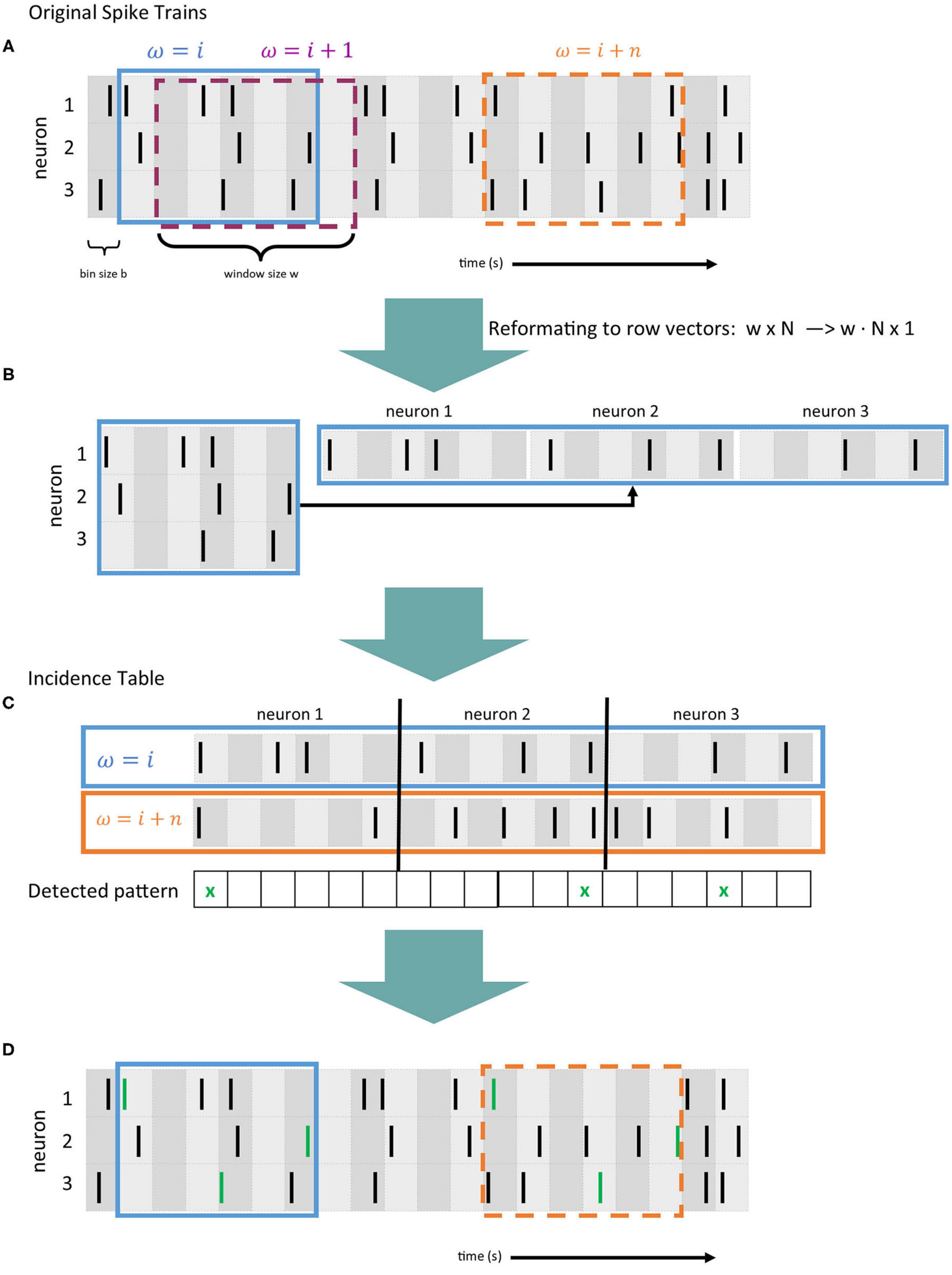


FIGURE 2 | Data preprocessing and evaluation flow of *SPADE* [based on Stella et al. (2019)]. **(A)** Example of 4 spike trains recorded in parallel, where each black line represents a spike. Time is divided into bins (gray vertical areas) of length *b*. A sliding window of size *w* is shifted bin by bin over the data (in blue, purple and orange). **(B)** The window matrix representation is converted to a row vector. **(C)** Simplified visualization of the pattern mining process (also called incidence table), where spikes occurring in the same bin in two window positions ($\omega = i$ and $\omega = i + n$) are detected. Coincident spikes across the two windows are indicated with a green cross. **(D)** Representation of the original spike trains as in panel A, where the spike pattern is detected and indicated with green lines.

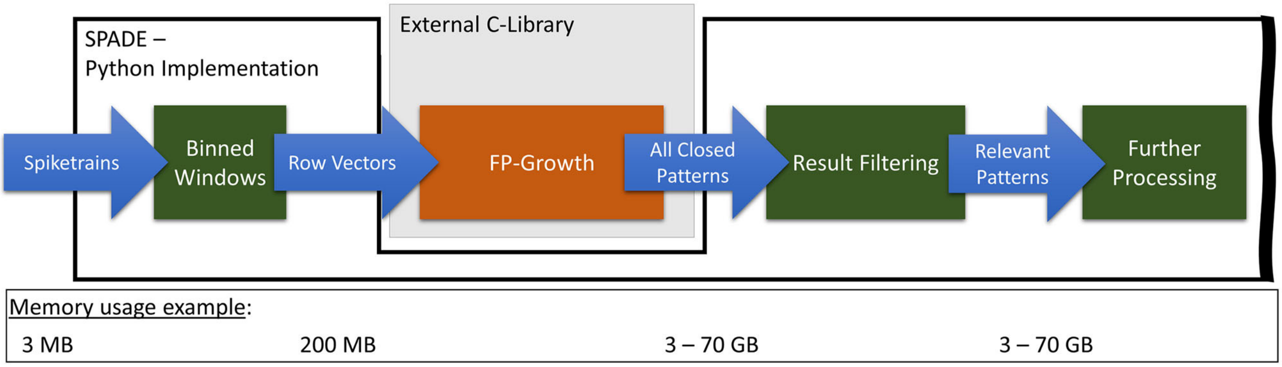


FIGURE 3 | Representation of the original FP-Growth embedding in *SPADE* with special regard to transferred data volumes.

Algorithm 1: Filter function used by *SPADE*

```
Input: The pattern $P$, the support of the pattern $S$, the minimum number of neurons $mn$ and the maximum support $ms$.
Output: Whether to keep the pattern or discard it.
  function FILTER_RESULT($P, S, w, mn, ms$)
      if $S > ms$ then
          return $false$
      end if
      $valid \leftarrow false$
      $neurons \leftarrow []$          ▷ Initialize the list of known neurons
      $cnt \leftarrow 0$
      for each $e \in P$ do
          if $e \mod w = 0$ then       ▷ Check if the spike occurred in the first bin
              $valid \leftarrow true$
          end if
          $n \leftarrow \frac{e}{w}$       ▷ Get the neuron id
          if $n \notin neurons$ then   ▷ Check if the neuron has already been checked
              $neurons[cnt] \leftarrow n$   ▷ Add the neuron to the known list
              $cnt \leftarrow cnt + 1$      ▷ Increment the counter
          end if
      end for
      if $cnt < mn$ then
          $valid \leftarrow false$
      end if
      return $valid$
  end function
```

minimum support and occurrence configurations, *FP-Growth* can consume up to 70 GB of memory.

From our analysis of the current state, we identified three main factors for the long runtime of this part of the algorithm. First, a generic *FP-Growth* implementation is used instead of one that is custom-tailored to the problem at hand. Second, all frequent patterns found by the algorithm are sent back to the Python code. Last, the filtering of the results is performed in Python. As noted in section 2.3, the highly optimized C-implementation of the *FP-Growth* algorithm is used in *SPADE*. However, due to the way *SPADE* operates, it does not need all possible closed patterns; it, in fact, only needs a fraction of them. Therefore, using an implementation that mines all closed patterns, as is currently the case, can significantly impact the performance. Furthermore, due to the data structures used internally by the *FP-Growth* implementation, all items of each found pattern have to be mapped back to their original data elements and inserted into a *numpy*-array to be usable in Python. This process can require a significant amount of time and memory and will be referred to as *conversion to Python*. Depending on the number of patterns, this can take several tens of minutes and consume up to 70 GB of memory. Finally, filtering out the repeating patterns takes a long time, as this is done in pure Python, without the assistance of an optimized C or C++ function, which could considerably speed up the process.

2.5. Optimized Implementation

We resolved the bottlenecks identified in section 2.4, thereby increasing the performance by several orders of magnitude. This was done by developing a custom C++-based *FP-Growth* implementation, which directly includes the result filtering in an external C++-library.

2.5.1. Custom FP-Growth Implementation With Result Filtering

The developed custom C++-based *FP-Growth* implementation is, in parts, based on *PyFIM* by Christian Borgelt. The core implementation of the closed pattern detection, using conditional itemset repositories (Grahne and Zhu, 2003), is entirely adopted from *PyFIM*. There are two significant differences between our version of *FP-Growth* and the general-purpose solution used before. First, the result filter function, applied by *SPADE* to the found closed frequent patterns, is integrated directly into *FP-Growth*. This shifts the filtering from Python to C++, thereby significantly decreasing the runtime and memory consumption, as only a fraction of all patterns needs to be saved. Second, the closed detection is not performed during the pattern mining process but instead afterward. This step was taken because, as

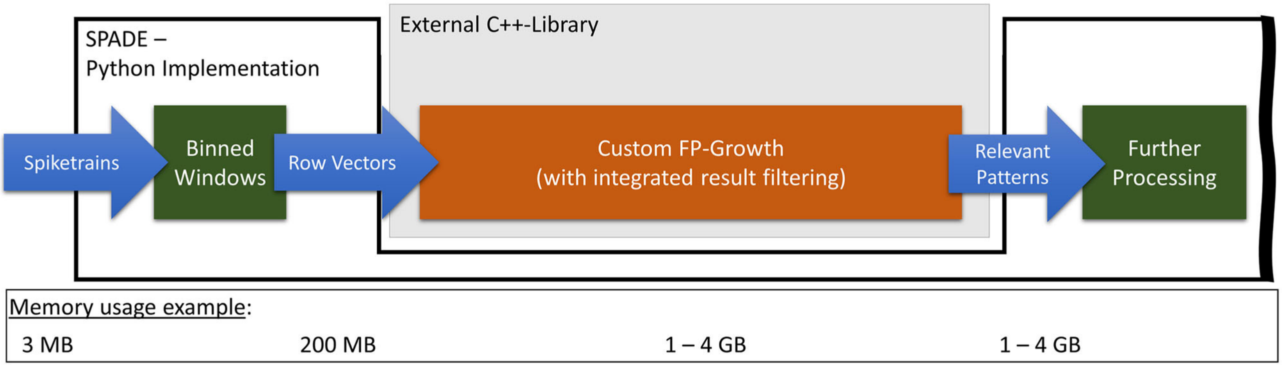


FIGURE 4 | Representation of the optimized FP-Growth embedding in SPADE with special regard to transferred data volumes.

mentioned before, the runtime of the closed frequent pattern detection scales with the number of patterns to check. Therefore, integrating the filter function into *FP-Growth* considerably reduces the number of patterns to check for closure. This decreases the runtime of the closed pattern detection and thus results in pattern mining requiring most of the runtime. Furthermore, the implementation for closed frequent pattern detection used in this work cannot be parallelized, in contrast to the pattern mining, which, as noted in section 2.2, can be reasonably easily performed in parallel. In a situation where the closed pattern detection has to be performed on all patterns, i.e., when there is no filter in place, splitting the mining and detection usually either does not affect the runtime or can even increase it. This is because detecting closed patterns is significantly more complex than pattern mining. **Figure 4** shows how *SPADEs* pattern mining flow changes when using our optimized *FP-Growth* module. Compared to the original flow, the peak memory consumption was reduced from up to 70 GB down to 4 GB.

2.5.2. Pattern Collector

In our custom *FP-Growth* version, we implemented a pattern collector to efficiently and adequately store the found patterns. It stores the pattern, its length, and support directly in memory. The collector allocates a block of memory each time the previous block is full or the new pattern's size exceeds the remaining space. Additionally, access functions have been integrated to allow for fast iteration over all stored patterns. Furthermore, we directly integrated the pattern filter function into the collector. This way, whenever a new pattern is passed to the collector, it first runs through the filter, and if it is invalid, it is discarded. As a result, only valid patterns are stored, and all others are discarded.

2.6. Parallelization and Distributed Computing

As an additional step, we integrated *OpenMP*[7] into our *FP-Growth* implementation, allowing us to parallelize the pattern mining process across all available CPU-cores, thereby significantly increasing the performance. As mentioned in section 2.2, parallelization of the pattern mining is possible because, when iterating over the header table, all iterations are entirely independent of each other, allowing them to be executed in parallel and in any order. Memory conflicts and potential race conditions were evaded by replicating the internal memory structures for each thread, preventing the threads from affecting each other. However, the closed frequent pattern detection algorithm requires its input patterns to be in an orderly fashion, i.e., the results of the first iteration, followed by the results of the second iteration, and so on. Therefore, we further modified the code to instantiate n pattern collector objects, where n is the header table's size. This way, each entry in the header table has its own pattern collector to store all found patterns. This allows the closed detector to operate correctly and removes overhead caused by the threading, as all threads no longer share a single pattern collector. Once the pattern mining process is finished, the closed pattern detector iterates over all n collector objects and identifies the closed frequent patterns. As mentioned in section 2.5.1, our implementation uses the closed pattern detector developed by Christian Borgelt, which cannot be easily parallelized, as mentioned in section 2.2. Therefore, at the moment, the closed pattern detection is performed sequentially on a single core.

The complete independence of the header table iterations allows for the pattern mining to be performed in parallel on all cores of a local processor and computed in parallel on several compute nodes. For this purpose, we integrated *MPI*[8] into our application to distribute the workload across different compute nodes. Through the use of the *MPI* execution environment *mpirun*, it is possible to spawn an arbitrary number of processes for a given application. Furthermore, spawning processes is not limited to the local system but can be done on an arbitrary number of remote nodes, e.g., a compute cluster. However, without integrating *MPI*-specific modifications into the code, execution across multiple nodes will only cause each node to run the entire application. Therefore, the *MPI-API* provides a large selection of functions to allow the processes to communicate, i.e., pass messages between each other. Each process possesses a unique identification number, the so-called *rank*. The *rank* will

[7] Open Multi-Processing - https://www.openmp.org/.

[8] **M**essage **P**assing **I**nterface.

be a number between 0 and the number of processes spawned by *MPI*. In most cases, one process, usually with *rank* 0, collects all results from all processes once they are finished and presents them to the user or continues working with them.

When integrating *MPI* into our code, only a few modifications were necessary. First, the header table loop was modified to start at the *rank* of the current process and stops iterating in steps of one, but instead in steps of size *p*, where *p* equals the total number of processes. This way, each process processes $\frac{n}{p}$ iterations. We equally distributed the workload across all nodes using a round-robin-styled loop to decrease the chance that one process finishes significantly ahead of the others. Finally, after the header table has been processed and all patterns have been mined, all processes except for the root process send their mined patterns, in the correct order, to the root, where they are added to the correct collectors. Afterward, all but the root process terminate, and the root process performs the closed pattern detection and outputs the final results to the user. It should be noted that our distributed approach requires the entire FP-tree to be built on each node, which can take a significant amount of time for large data sets. However, this is not of any concern because due to the nature of the data, the data sets used with *SPADE* are relatively small, causing the FP-tree creation to only take a few seconds.

3. RESULTS

In this section, we evaluate the performance, in terms of runtime, memory consumption, and energy efficiency, of our optimized pattern mining flow on several different devices and compare it to *SPADE's* original program flow. Since in this work, we primarily focused on accelerating the pattern mining and filtering, only the runtimes of the associated steps are examined in the following. Therefore, full runtime refers to the total runtime required by all tasks, i.e., pattern mining, data conversion to Python, and pattern filtering. Since in the original implementation, the pattern mining step also included closed pattern detection and data conversion to Python, for the baseline, these steps are not listed separately. Because we have separated these steps in our optimized version, we include the corresponding runtimes. We show that using our optimizations considerably reduces the runtime and memory consumption and noticeably increases energy efficiency, while producing the same results as the original. Furthermore, due to the memory optimizations, it is now possible to perform the pattern mining on low-power embedded devices.

3.1. Test Setup

We used different platforms for evaluation. The first platform, serving as a baseline, was a workstation equipped with an Intel Xeon E5-1650 v4 (6 cores running at 3.60 GHz) server CPU and 256 GB quad-channel DDR4 memory, running Ubuntu 16.04. For the other evaluations, we used our RECS|Box[9] server (Oleksiak et al., 2019), a modular and scalable microserver platform for resource-efficient heterogeneous high-performance computing.

The RECS|Box is a heterogeneous cluster server that allows the user to choose between several computer architectures, network systems, network topologies, and microserver sizes. In this context, a microserver refers to an independent computer-on-module (CoM) that integrates all components (e.g., CPU, memory, IO, and power subsystem) in a small, compact form factor for integration into a server or embedded environment. In contrast to existing homogeneous microserver platforms that support only a single microserver architecture, RECS|Box seamlessly integrates the full range of microserver technologies in a single chassis, including various CPUs as well as accelerators based on FPGAs[10] and GPUs. Hence, it can be used to easily set up heterogeneous processing platforms optimized for specific application requirements. CoMs are available for all major computing platforms in both low-power and high-performance versions. Like the big-little approach known from mobile processors, this can be used to further increase energy efficiency by dynamically switching, e.g., between 64-bit ARM server processors and 64-bit ARM mobile SoCs[11] or between different FPGA/GPU devices.

Figure 5 gives a high-level overview of the modular approach used for the design of the RECS|Box system architecture. This modularity guarantees flexibility and reusability and thus high maintainability. Microservers are grouped on carrier boards that support hot-swapping and hot-plugging, similar to a blade-style server. Three different carriers are available: one integrating 16 low-power microservers, one for three high-performance microservers, and one integrating PCIe-based hardware accelerators. All microservers are designed based on well-established CoM form factors[12], which facilitates the integration of third-party microserver modules into the RECS|Box. Not only can the platform be individually adapted to the given problem due to its modularity, but it is also able to monitor the power consumption of the individual compute modules very precisely. Furthermore, the installed modules can communicate with each other through high-speed Ethernet over PCI-Express, allowing for fast data exchange, e.g., when performing distributed computing.

For our evaluation, we used high-performance as well as low-power microservers. Firstly, we used a microserver equipped with a HiSilicon *Hi1616* (Kunpeng 916) dotriaconta-core ARM processor (32 cores running at 2.4 GHz) and 64 GB of quad-channel DDR4 memory, running CentOS 7.6, in a dual-socket configuration (resulting in 64 cores/128 GB). In the following, this will be referred to as the *Hi1616* microserver. Next, an *ADLINK Express-BD7*[13] module, equipped with an Intel Xeon D-1577 (16 cores running at 1.30 GHz) and 32 GB dual-channel DDR4 memory running Ubuntu 18.04 was used. Additionally, we used an *ADLINK Express-CFR-E*[14] microserver, equipped with an Intel Xeon E-2276ME (6 cores running at 2.8 GHz)

[9] **R**esource-**E**fficient **C**luster **S**erver – https://embedded.christmann.info/products.

[10] **F**ield **P**rogrammable **G**ate **A**rray.

[11] **S**ystem-**o**n-a-**C**hip.

[12] https://www.picmg.org/openstandards/com-express/

[13] https://www.adlinktech.com/Products/Computer_on_Modules/COMExpressType7/Express-BD7

[14] https://www.adlinktech.com/Products/Computer_on_Modules/COMExpressType6/Express-CFR

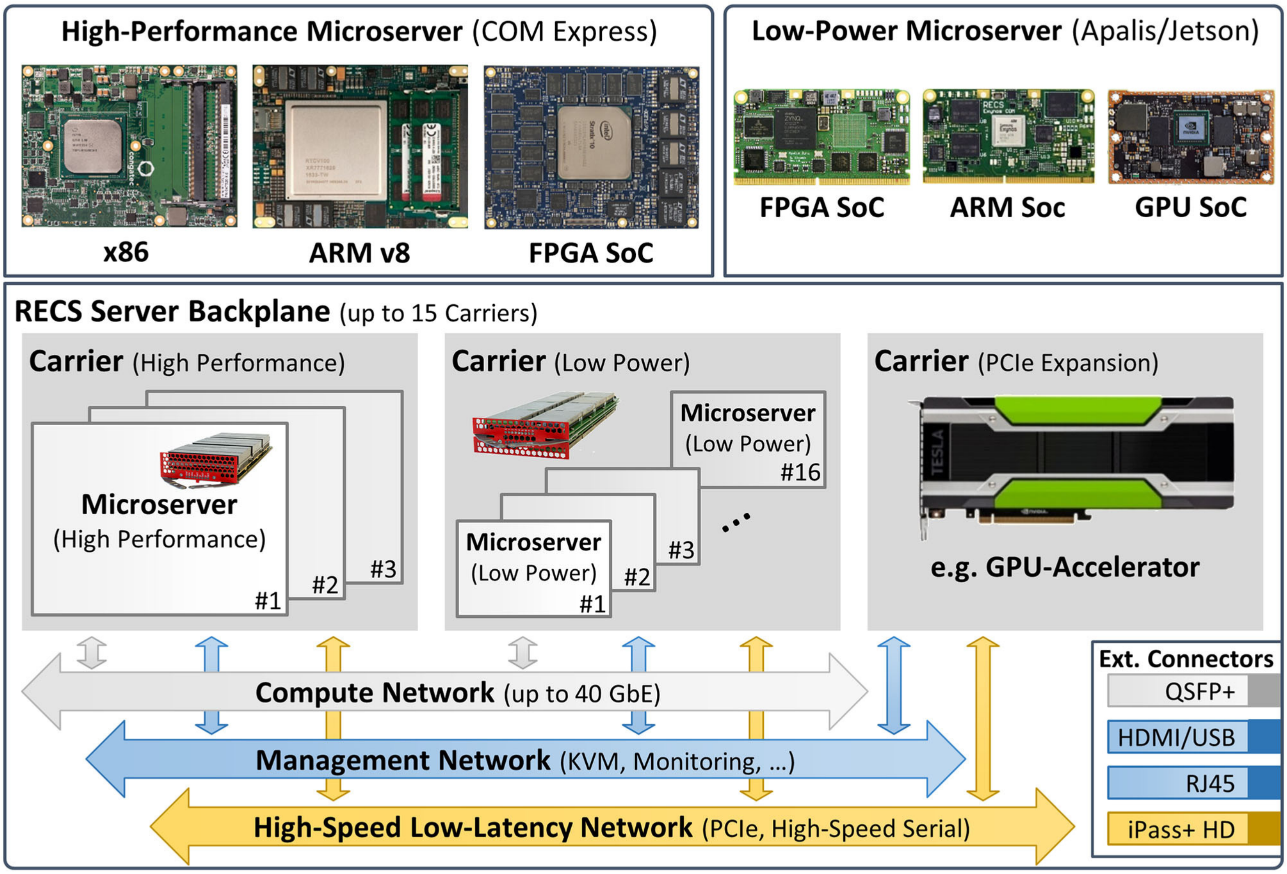

High-Performance Microserver (COM Express)
x86
ARM v8
FPGA SoC
Low-Power Microserver (Apalis/Jetson)
FPGA SoC
ARM Soc
GPU SoC
RECS Server Backplane (up to 15 Carriers)
Carrier (High Performance)
Microserver (High Performance)
#1
#2
#3
Carrier (Low Power)
Microserver (Low Power)
#16
Microserver (Low Power)
#1
#2
#3
Carrier (PCIe Expansion)
e.g. GPU-Accelerator
Compute Network (up to 40 GbE)
Management Network (KVM, Monitoring, ...)
High-Speed Low-Latency Network (PCIe, High-Speed Serial)
Ext. Connectors
QSFP+
HDMI/USB
RJ45
iPass+ HD

FIGURE 5 | Overview of the RECS|Box hardware platform.

and 32 GB of dual-channel DDR4 memory, also running Ubuntu 18.04. Finally, we executed our implementation on three different types of embedded NVIDIA Jetson devices, each running Ubuntu 18.04.

As mentioned above, we also evaluated energy efficiency by measuring each platform's system power consumption during the execution of the test. System power consumption refers to the amount of power consumed by the entire system after the power supply unit (PSU), i.e., CPU, memory, storage, and system accessories. We measure after the PSU because, depending on the unit's quality and overall load, there can be a significant difference between the system's power and the PSU. Using the monitoring features of the RECS|Box, we were able to accurately measure the power consumption of the different devices installed in it. For the workstation, the power consumption was calculated based on continuous voltage and current measurements using a Tektronix MDO4054B-6[15] oscilloscope in combination with a Tektronix TCP0030A[16] current probe. Using the TCP0030A probe, it is possible to continuously measure the electrical current of the 12 V power supply with a sampling rate between 500 and 2,500 samples per second. All tests were performed in an air-conditioned room at about 19°C; therefore, the DC gain accuracy of the probe is $< 1\%$ (cf. Tektronix, 2006).

For the evaluation, we used neural data extracted from *in-vivo* experimental recordings. In the experiment, a macaque monkey performs a delayed reaching and grasping task, while its neural activity is recorded using a 10x10 electrode array chronically inserted in the premotor and motor cortex (Riehle et al., 2013; Brochier et al., 2018). The experimental protocol is as follows: the monkey is trained to self-initiate the trial by pressing a start button, then to wait for a first visual cue, indicating the type of grip that it has to perform (either precision grip -PG- or side grip -SG-). After a delay period of 1 s, the monkey receives the *GO* signal, together with the information of the amount of force to apply on the object (high force -HF- or low force -LF-). After the monkey has successfully grasped and pulled the object with the correct grip, a reward is given. In this study, we consider session *i140703-001* of Monkey *N* which lasts 1003 s, and consists of 141 correct trials with randomized trial type order (i.e., combinations of grip and force conditions: *PGHF, PGLF, SGHF, SGLF*). Detailed descriptions of this published data set are given in Brochier et al. (2018). For this data, the SPADE method can be used to detect behaviorally-locked spatio-temporal spike patterns, mimicking

[15] https://www.tek.com/oscilloscope/mdo4054b-6

[16] https://www.tek.com/datasheet/30-ac-dc-current-probe

the analysis performed in (Torre et al., 2016). In this scenario, it is necessary to segment the data in order to perform a time-resolved analysis: we segment trials into six 500 ms long epochs, each related to a behaviorally relevant event of the trial (*start, cue presentation, early delay, late delay, movement, reward*). Identical epochs belonging to the same trial type are concatenated to form a total of 6x4 = 24 data sets to be analyzed with SPADE. In this example, we consider the segment in which the monkey performs the reaching and grasping movement with precision grip and high force (*movement_PGHF*). The data set has a total duration of 22.32 s, consists of 32 concatenated trials, and has 150 units recorded in parallel after pre-processing. We select specifically only single unit activities (SUA) exhibiting signal to noise ratio (SNR) $>$ 2.5. Furthermore, a buffer time of 200 ms is inserted between successive trials.

This data set is a typical use case for *SPADE* in both length and number of observed neurons, making it a fitting example to benchmark the performance of the method. When transforming the input data, as described in section 2.3, 3602 transactions with 3,000 unique items were created, using a *bin* size of 5 ms and a window length of 100 ms (20 bins). We divide the analysis into eight different *jobs*, each for a fixed pattern size (number of spikes), starting from 2 and ending at 10+ in steps of one. For each pattern size, the minimum number of occurrences is estimated for optimizing the pattern mining: the distribution of number of occurrences of a chance pattern of fixed size is estimated with a Poisson assumption using the average estimated rate of all neurons. By taking the 95% percentile of this distribution, we estimate the number of occurrences that a non-significant pattern would have under independence, giving us a lower bound for the support in the pattern search. The absolute lower bound for pattern occurrences is fixed to 10. In fact, in a classical use case of the method, we would focus on behavior-specific patterns. Thus, patterns occurring in less than 30% of the total number of trials ($\sim$ 30 trials per combination of epoch and trial type) are not considered. The different configurations of pattern sizes and number of pattern occurrences are described in **Table 1**. In addition to the configurations, the table also lists the total number of unfiltered frequent patterns found for each job and how many are left after filtering. With these values, the impact of one of our main optimizations, i.e., filtering the patterns directly when they are mined, can be seen very clearly. This significantly reduces the number of patterns to be stored, thus reducing overall memory consumption and reducing the number of patterns fed to the closed detector to a fraction of the original amount. Through filtering, between 90 and 100% of the mined patterns are discarded.

3.2. Evaluation of the Software Baseline on x86 Server

To determine the runtime, memory consumption, and energy efficiency of the current flow, i.e., create a performance baseline, we executed the latest *SPADE* version (v0.9.0) on the workstation mentioned before, base on an Intel Broadwell Xeon E5-1650 v4. The considerable memory consumption of the baseline flow made execution on the embedded devices impossible. **Table 2** depicts the time, in seconds, required for the entire C-based *FP-Growth* flow, the time, in seconds, to perform the result filtering in Python, and the accumulated runtime, in minutes. The runtime for the *FP-Growth* flow includes the data conversion from Python to C, the pattern detection (including closed pattern detection), and the conversion of the results back to Python. Furthermore, the table also lists the peak memory consumption for each *job*. As can be seen, increasingly complex *jobs* can take from a few minutes up to 2 h and consume more than 70 GB of memory. As mentioned in section 2, these high memory requirements are mainly caused by the need to convert all closed patterns (up to 400 million, depending on the *job*) back to Python, where the filtering is performed. The baseline flow required 6 h and 13 min to complete all eight jobs. Based on the workstations' average power consumption of 64.8 W[17], the entire computation consumed 1.45 MJ[18].

Afterward, we executed our optimized implementation both in single- and in multi-threaded (12-threads) mode. Both runtimes, as well as the peak memory consumption, are depicted in **Table 3**. As only the *FP-Growth* implementation is affected by threading, there was no noticeable difference in the time required for the *closed frequent pattern detection* or the conversion to Python, so only the results from the single-threaded test are listed. By filtering the results directly during the creation

TABLE 1 | Configurations of the eight *jobs* used for the evaluation.

Job	Min. occ.	Min. spikes	Patterns	Filtered patterns
0	88	2	200,971	22,709
1	25	3	16,477,189	1,562,086
2	12	4	246,958,100	8,486,483
3	10	5	424,713,012	398,618
4	10	6	259,915,712	41
5	10	7	109,269,024	0
6	10	8	29,385,509	0
7	10	9	4,637,531	0

TABLE 2 | Workstation runtime and memory consumption of the implementation currently used in *SPADE*.

Job	FP-growth runtime (s)	Filtering runtime (s)	Full runtime (s)	Peak mem. Consumption (GB)
0	0.9	0.9	1.8	0.4
1	41.4	83.6	125.0	3.3
2	2299.0	1386.9	3685.9	44.0
3	6506.7	2351.9	8858.6	77.5
4	3033.1	1451.7	4484.8	45.8
5	1651.5	647.5	2299.0	21.3
6	1369.1	187.7	1556.8	8.0
7	1336.7	30.8	1367.5	3.7
Sum	16238.4	6141.0	22379.4	

[17] Watt.

[18] Joule (watt-second).

TABLE 3 | Workstation runtime of the optimized implementation in single (ST)- and multi (MT)-threaded mode.

Job	FP-growth runtime (s)		Closed det. (s)	Conversion to Python (s)	Full runtime (s)		Peak mem. cons. (GB)	
	ST	MT			ST	MT	ST	MT
0	1.2	0.5	0.0	0.0	1.3	0.6	0.5	0.5
1	14.9	2.4	1.3	1.0	17.2	4.7	1.3	1.3
2	116.4	17.4	13.3	14.3	143.9	45.1	3.5	3.5
3	205.4	32.7	0.5	0.7	206.6	34.0	3.4	3.4
4	195.9	31.3	0.0	0.0	196.0	31.4	1.9	1.9
5	180.9	29.8	0.0	0.0	181.0	29.9	1.8	1.8
6	174.1	25.7	0.0	0.0	174.1	25.8	1.8	1.8
7	171.0	25.2	0.0	0.0	171.1	25.3	1.8	1.8
Sum	1059.8	165.0	15.1	16.0	1091.2	196.8		

TABLE 4 | Runtime of the multi-threaded implementation on the ADLINK Express-BD7, the ADLINK Express-CFR-E and the HiSilicon Hi1616 microserver.

Job	FP-growth runtime (s)			Closed detection (s)			Conv. to Python (s)			Full runtime (s)		
	BD7	CFR	Hi16	BD7	CFR	Hi16	BD7	CFR	Hi16	BD7	CFR	Hi16
0	0.9	0.5	0.8	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.5	0.9
1	2.6	2.2	1.2	2.9	1.1	2.4	1.5	0.6	2.2	7.1	4.0	6.0
2	16.0	16.8	5.0	24.8	10.8	22.9	20.3	8.2	35.6	61.2	35.8	63.7
3	28.0	31.8	8.6	0.8	0.4	0.8	1.1	0.4	1.6	30.0	32.7	11.1
4	27.0	32.0	8.0	0.0	0.0	0.0	0.0	0.0	0.0	27.2	32.1	8.1
5	25.0	30.2	7.5	0.0	0.0	0.0	0.0	0.0	0.0	25.2	30.3	7.6
6	23.8	31.0	7.2	0.0	0.0	0.0	0.0	0.0	0.0	24.0	31.1	7.3
7	23.1	30.4	7.0	0.0	0.0	0.0	0.0	0.0	0.0	23.3	30.5	7.1
Sum	146.4	174.9	45.3	28.5	12.3	26.1	22.9	9.2	39.4	199.0	197.0	111.8

process, it was possible to significantly reduce the peak memory consumption to a maximum of 4 GB. The single-threaded version required 18 min and 11 s to complete all eight jobs, while the multi-threaded version finished all jobs in 3 min and 17 s, making it 114 times as fast as the baseline (see **Table 7**). Regarding energy efficiency, 65 W (70,876 J) and 109.9 W (21,638 J) were consumed in single- and multi-threaded mode, respectively. The multi-threaded implementation achieved an energy efficiency 67 times higher than the current implementation (see **Table 7**).

3.3. Evaluation on RECS|Box for Server Processors

Due to its combined 64 cores running at 2.4 GHz, the *Hi1616* microserver achieved the highest parallel processing speed and overall lowest runtime of all considered platforms (cf. **Table 4**). In terms of overall runtime, compared to the workstation, it finished all jobs in 57% of the time, with an average power consumption of 123.3 W (13,780 J), 64% of the energy the workstation required. Compared to the baseline, a speedup of 200 was achieved while being 105 times as energy efficient (see **Table 7**). The Intel Xeon D-1577 in the *ADLINK Express-BD7*, on the other hand, required just 2 s longer (3 min and 19 s) than the workstation to finish all jobs. However, the average power consumption of the Xeon D was only 51.1 W (10,164 J), meaning only 47% of the energy was required to finish all jobs compared to the workstation. When comparing the results to the baseline, the Xeon D achieved a speedup of 113 while being 143 times more energy efficient. Finally, the Intel Xeon E-2276ME finished all jobs in the same time as the workstation while requiring on average only 60.3 W (11,887 J), i.e., 55% of the energy the workstation required. Compared to the baseline, a speedup by a factor of 114 together with a 122 times higher energy efficiency was achieved (see **Table 7**).

3.4. Evaluation on RECS|Box for Embedded Processors

Over the last decade, energy efficiency has become increasingly important in data centers, especially when focusing on cloud computing (Oleksiak et al., 2017). Therefore, we evaluated our implementation's performance and energy efficiency on several embedded devices, namely the NVIDIA Jetson AGX Xavier[19], the NVIDIA Jetson Xavier NX[20], and up to four NVIDIA Jetson

[19] https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit
[20] https://developer.nvidia.com/embedded/jetson-xavier-nx

TABLE 5 | Runtime of the multi-threaded implementation on all three embedded devices.

Job	FP-growth runtime (s)			Closed detection (s)			Conversion to Python (s)			Full runtime (s)		
	AGX	NX	TX2	AGX	NX	TX2	AGX	NX	TX2	AGX	NX	TX2
0	1.0	1.7	1.9	0.0	0.0	0.0	0.0	0.0	0.0	1.1	1.9	2.1
1	5.9	10.5	8.7	1.6	2.5	1.9	1.8	2.7	2.8	9.3	15.8	13.6
2	39.3	75.5	55.3	15.0	21.7	19.6	19.3	35.9	31.2	73.6	133.3	106.3
3	71.3	143.8	94.1	0.5	0.7	0.6	0.8	2.2	1.7	72.6	146.8	96.5
4	68.6	137.2	90.6	0.0	0.0	0.0	0.0	0.0	0.0	68.6	137.4	90.8
5	69.6	119.9	89.0	0.0	0.0	0.0	0.0	0.0	0.0	69.7	120.0	89.1
6	68.6	132.5	83.7	0.0	0.0	0.0	0.0	0.0	0.0	68.7	132.6	83.8
7	67.0	128.2	88.9	0.0	0.0	0.0	0.0	0.0	0.0	67.1	128.3	89.0
Sum	391.4	749.3	512.2	17.1	24.9	22.1	21.9	40.7	35.7	430.7	816.1	571.2

TX2[21], each running Ubuntu 18.04. These devices feature low power consumption along with a small form factor and are equipped with between four and eight ARM cores. In addition to its quad-core ARM Cortex-A57 CPU, the Jetson TX2 also possesses a dual-core NVIDIA Denver 2 CPU. In contrast to that, the Jetson AGX and NX use hexa- and octa-core NVIDIA Carmel ARMv8.2 CPUs, respectively. With its 32 GB of DDR4 memory, the AGX Xavier possesses four times as much memory as the Xavier NX and the Jetson TX2, which each are equipped with 8 GB. Changing the power mode makes it possible to adjust the CPU and GPU clock frequency and disable all but one core, e.g., disable the Denver cores on the TX2 and only use the ARM cores or only use four of the eight cores on the AGX Xavier. For our tests, we configured each device to use all available CPU cores at their maximum clock frequency. While each core of the Jetson TX2 and the AGX Xavier can achieve a maximum frequency of 2 GHz, the cores on the Xavier NX are limited to 1.4 GHz when all cores are enabled. Because currently, the GPUs integrated in the devices are not used at all, the GPU frequency was limited as much as possible to reduce power consumption. The achieved performance and energy efficiency values were compared to the original flow and the results from the workstation test presented in section 3.2.

3.4.1. Execution on a Single Device

Table 5 summarizes the runtimes of the three different embedded platforms for all eight jobs. The best performance is achieved by the NVIDIA Jetson AGX Xavier, which completed all jobs in 7 min and 11 s, followed by the Jetson TX2 and the Xavier NX with 9 min, 31 s, and 13 min, 36 s, respectively. Although compared to the workstation, the embedded devices' runtime is between 2.2 and 4.4 times longer, they required significantly less power and consumed overall less energy. The most power was required by the AGX Xavier, which consumed an average of 20.4 W, resulting in an energy consumption of 8,786 J, followed by the Xavier NX with 6.7 W (5,468 J, one-fourth of the workstation), and the least amount of energy was required by the Jetson TX2 with 9.1 W (5,181 J, less than one-fourth of the workstation). Compared to the baseline flow, the embedded devices are between 52 and 27 times faster and between 280 and 165 times more energy efficient (see **Table 7**). These results show that even though the runtime is higher than on a workstation, the use of embedded platforms may be more suitable in situations where energy efficiency is of a higher priority than runtime.

3.4.2. Execution on Multiple Devices

In addition to the previous single device execution, we also utilized the *OpenMPI*-based distributed flow described in section 2.6 to run the implementation on up to four NVIDIA Jetson TX2. As mentioned before, only the *FP-Growth* part is accelerated using multi-threading and distributed computing, while everything else is performed sequentially on the root node. Therefore, the runtimes for the closed detection and the conversion to Python are omitted here, as they equal those of the single node execution, depicted in **Table 5**. **Table 6** shows the time required for the *FP-Growth*-based pattern mining, the full runtime of the accelerated section, and the communication overhead. It should be noted that the communication time is part of the *FP-Growth* runtime and is listed separately to show its impact. When looking at the accumulated runtime of the *FP-Growth* part, a noticeable improvement compared to the execution on a single node is visible. For a single TX2, this part took 8:32 min, while, when using two, three, or four TX2, it was reduced to 4:36, 3:15, and 2:32 min, respectively. Two Jetson TX2 significantly outperform the AGX Xavier in terms of runtime and energy efficiency, as the two TX2 only consume 5,986 J 68% of the energy required by the AGX. As only a part of the computation is performed in parallel, an increase in compute nodes will result in a decrease in energy efficiency. However, four TX2 modules are able to finish all jobs in nearly the same amount of time as the workstation (16 s slower) while only consuming 31% (6,670 J) of the energy required by the workstation. Compared to the baseline, the use of between two and four TX2 modules achieved an acceleration by a factor of 66 to 105 and an increase in energy efficiency by a factor of 217 to 242 (see **Table 7**). Ultimately, the decision to make is whether to decrease the runtime by adding more TX2 nodes, resulting in increasing energy consumption or increasing energy efficiency at the cost of an increased runtime.

[21] https://developer.nvidia.com/embedded/jetson-tx2

TABLE 6 | Runtime of the multi-threaded implementation on up to four NVIDIA Jetson TX2.

Job	FP-growth runtime (s)			Communication (s)			Full runtime (s)		
	2 TX2	3 TX2	4 TX2	2 TX2	3 TX2	4 TX2	2 TX2	3 TX2	4 TX2
0	1.5	1.7	1.7	0.1	0.2	0.3	1.7	1.8	1.9
1	5.3	4.1	3.8	0.5	0.6	0.9	9.8	0.9	8.3
2	29.5	21.9	17.4	1.5	2.0	2.3	82.7	71.2	70.4
3	52.6	35.3	27.4	0.3	0.5	0.6	55.1	37.8	30.0
4	50.4	33.9	27.9	0.2	0.4	0.5	50.5	34.0	28.0
5	47.4	34.0	23.7	0.2	0.4	0.5	47.5	34.1	23.9
6	45.2	31.5	23.9	0.2	0.4	0.5	45.3	31.6	24.0
7	44.1	32.3	26.0	0.2	0.4	0.5	44.2	32.4	26.1
Sum	276.0	194.7	151.8	3.2	4.9	6.1	336.8	243.8	212.6

TABLE 7 | Runtime and energy consumption of all platforms.

System	Power (W)	Runtime (s)	Energy		Improvement over baseline	
			Joule	Wh	Energy	Runtime
Workstation (Baseline)	64.8	22,379.4	1,450,182	402.83	1	1
Workstation (ST)	65.0	1091.2	70,879	19.69	20	21
Workstation (MT)	109.9	196.8	21,638	6.01	67	114
Express-BD7	51.1	198.9	10,164	2.82	143	113
Express-CFR-E	60.3	197.0	11,887	3.30	122	114
Hi1616	123.3	111.8	13,780	3.82	105	200
AGX Xavier	20.4	430.7	8,786	2.44	165	52
Xavier NX	6.7	816.1	5,468	1.52	265	27
Jetson TX2	9.1	571.2	5,181	1.44	280	39
2x Jetson TX2	17.8	336.8	5,986	1.66	242	66
3x Jetson TX2	25.0	243.8	6,093	1.69	238	92
4x Jetson TX2	31.4	212.6	6,670	1.85	217	105

TABLE 8 | Full runtime (in seconds) comparison of the original and the optimized flow for different data sets.

Data set	Length (s)	Neurons	Found patterns	Original flow		Optimized flow	
				Runtime	Baseline-%	Runtime	Baseline-%
Baseline	22.32	150	10,214,712	22379.4	100%	196.8	100%
Long	1003.00	150	7,097,875	–	–	3052.6	1551%
Short	5.00	150	73,172	89.4	0.4%	4.0	2%
300 Neurons	22.32	300	28,077,304	28257.7	126%	432.2	220%
450 Neurons	22.32	450	64,933,631	64167.5	287%	1241.1	631%

3.5. Scalability

We analyzed the scalability of our optimized flow in terms of increased compute power, e.g., multiple NVIDIA *Jetson TX2* and with regards to data sets with varying properties, i.e., longer and shorter recordings as well as recordings with up to 450 neurons. All measurements for both the original flow and our optimized version were performed on the workstation system. For this evaluation, we used four different data sets. First, the entire 1,003 s long recording session of 150 neurons mentioned in section 3.1 was used as a baseline data set to analyze how both implementations handle long data sets. Next, to test the opposite, the first 5 s of the *movement_PGHF* data set were used to analyze the performance when working with short inputs. Finally, to test the effect an increase in neurons has on the runtime, we created two data sets, each with a length of 22.32 s and with 300 and 450 neurons, respectively, by stacking spike trains of the original data set. The total runtime, i.e., *FP-Growth*, filtering, closed detection, and conversion to Python, for each data set and both flows, is given in **Table 8**. Furthermore, the table lists the runtime as a percentage of the baseline data set's runtime.

The original flow was unable to process the *long* recording, as we had to stop it after 30 h after it consumed over 200 GB of memory.

When analyzing the table, it can be seen that when the number of neurons is increased, our implementation does not scale as well as the original. This becomes particularly evident when considering that a tripling of the neurons leads to a more than sixfold increase of the runtime in our version. In comparison, the runtime of the original version did not even triple. In contrast to this, when the recording duration increases or decreases, the scaling is comparable to the original. On the one hand, when the recording time is decreased from 22.32 to 5 s, only 2% of the original runtime was required, i.e., a reduction by a factor of 50. On the other hand, when the recording length is increased by a factor of 45, the runtime increases only by a factor of about 15. The main bottleneck and one of the primary factors for the inadequate scaling of the optimized flow are the closed detection and the data conversion to Python. As seen before, in *jobs* where many valid patterns are found by *FP-Growth*, these two steps significantly impact the overall performance, as they are currently executed sequentially on a single CPU core in contrast to the parallel *FP-Growth*. This is also the primary reason for the weaker scaling when the number of neurons increases, which can lead to a significant increase in found patterns. Concluding, it can be said that although our implementation scales not as well as the original, it still scales adequately even when confronted with long data sets or ones containing several hundred neurons. Furthermore, due to the overall significantly lower runtime, our proposed flow is between one and two orders of magnitude faster than the original.

4. DISCUSSION

Finding spike patterns in parallel spike trains using the *FP-Growth* pattern mining algorithm and a custom filter function is one of the most time-intensive parts of the *SPADE* method. In the currently available implementation, pattern mining is performed using a C-based Python module, while the filtering is done directly in Python. There are some significant flaws in the current flow that result in a significantly increased runtime. On the one hand, all found patterns need to be converted from C to Python, which takes a long time and consumes a large amount of memory. On the other hand, performing the pattern filtering in the Python programming language also negatively affects the runtime. Therefore, in this work, we developed a multi-threaded C++-based Python module that, while maintaining the original flow's functionality, performed the task between 27 and 200 times faster, while at the same time being 67 to 280 times as energy efficient depending on the executing hardware. By integrating the pattern filtering function directly into the *FP-Growth* implementation developed in this work, we dramatically reduced the number of produced patterns that need to be converted to Python. This reduced not only the runtime but also the memory consumption. Furthermore, we integrated multi-threading and distributed computing capabilities into our *FP-Growth* implementation to fully utilize the CPU of one or more compute nodes. Additionally, we showed that our implementation scales reasonably when the number of neurons or the length of the recording is changed and is able to finish the processing of a very large data set (1,003 s of neuron activity) in less than an hour, a task that was not possible using the original version. As a result, the improvement of the method enables the analysis of experimental data in a feasible amount of time together with the statistical evaluation of mined patterns, i.e., in the case where *FP-Growth* is applied not only on the original data set, but also on its surrogates, as explained in section 2.3. Our optimized flow opens up the possibility to perform more complex analyses due to the highly reduced amount of time. This makes it possible to handle large state-of-the-art data sets, such as data recorded from multiple Utah arrays (Chen et al., 2020), or Neuropixel probes (Juavinett et al., 2019), and to combine the results of SPADE with other approaches to investigate the correlative structure of neuronal dynamics (Diana et al., 2019; Watanabe et al., 2019; Williams et al., 2020).

4.1. Platform Comparison

Here, we perform a concluding comparison of the results achieved by our optimized implementation on the different platforms. The performance, in terms of runtime, memory consumption, and energy efficiency of our implementation was evaluated on a workstation system, a *Hi1616* microserver equipped with two HiSilicon *Hi1616* CPUs, an *ADLINK Express-BD7* equipped with an Intel Xeon D-1577, an *ADLINK Express-CFR-E* equipped with an Intel Xeon E-2276 and three different embedded computing devices from NVIDIA, namely *Xavier NX*, *AGX Xavier* and *Jetson TX2*. For an easy comparison, some of the most distinctive features of each platform focused on the respective CPU are shown in **Table 9**. These are, among others, the architecture, TDP[22], and ISA[23] of the CPU, as well as the type of memory installed. **Figure 6** shows the performance, in terms of *FP-Growth* runtime only, total execution time, and energy consumption, of the different platforms. The graph is sorted by total execution time. *Total Pattern Mining Flow* refers to the time required for the entire accelerated flow, i.e., *FP-Growth*-based pattern mining, pattern filtering, closed detection, and data conversion to Python, while *FP-Growth Only* exclusively shows the time required for the *FP-Growth*-based pattern mining and the pattern filtering.

Except for the *Hi1616* microserver, *FP-Growth* consumed the largest portion of the runtime on all platforms. Thanks to its 64 cores, the *Hi1616* microserver achieved the highest parallel processing performance and overall fastest execution time (111 s). However, due to the low individual core performance, a significant amount of time was required for the flow's sequential parts, which noticeably increased the full runtime. This, in turn, affected the power consumption, which resulted in the third-highest energy consumption (13,780 J). As can be expected, the longest runtime (1,091 s) and the highest energy consumption (70,876 J) belong to the single-threaded version's execution on the workstation. However, these values are still one order of magnitude lower than the original implementation, whose results are 20 times higher in both aspects (22,379 s and 1,450,185 J).

[22] **T**hermal **D**esign **P**ower.

[23] **I**nstruction **S**et **A**rchitecture.

TABLE 9 | Overview of the different platforms and their distinctive features.

Platform	CPU	Architecture	Memory	Cores	Threads	Clockrate	TDP	ISA
Workstation	E5-1650 v4	Haswell	DDR4	6	12	3.6 GHz	140 W	x86
Exp.-BD7	D-1577	Broadwell	DDR4	16	32	1.3 GHz	45 W	x86
Exp.-CFR-E	E-2276ME	Coffee Lake	DDR4	6	12	2.8 GHz	45 W	x86
Hi1616	Hi1616	Kunpeng	DDR4	32	32	2.4 GHz	85 W	A64
Jetson TX2	Cortex-A57	ARMv8-A	LPDDR4	4	4	2.0 GHz	7.5 W	A64
	Denver	Denver	LPDDR4	2	2	2.0 GHz	7.5 W	A64
AGX Xavier	Carmel	Carmel	LPDDR4X	8	8	2.0 GHz	30 W	A64
Xavier NX	Carmel	Carmel	LPDDR4X	6	6	1.4 GHz	15 W	A64

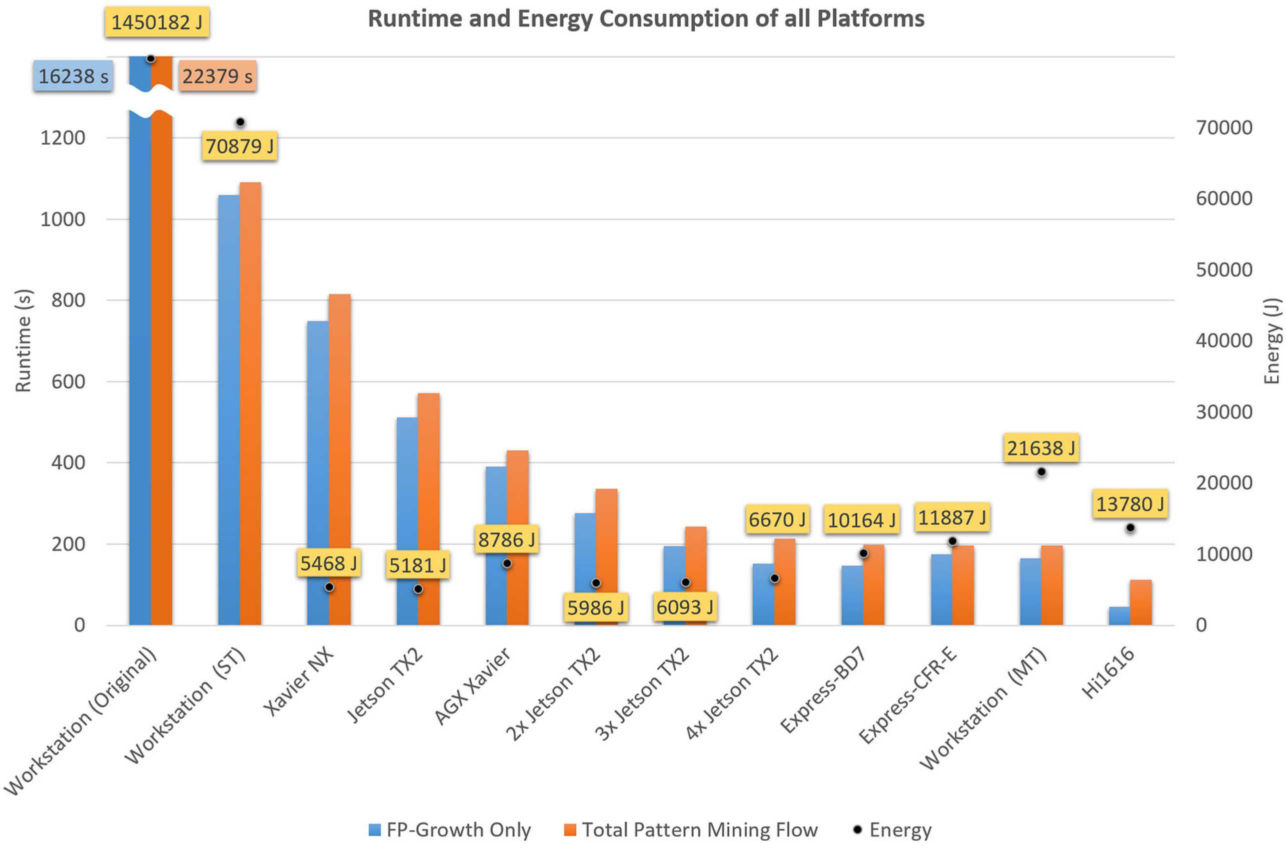


FIGURE 6 | Runtimes of the optimized flow on all considered platforms (cf. **Table 7**). *MT* refers to the multi-threaded, *ST* to the single-threaded and *Original* to the baseline (currently used) version. A detailed overview over the different platforms and their features is presented in **Table 9**.

For identifying the most suitable platform for the given application, both runtime, and energy consumption have to be considered. The lowest energy consumption was obtained using one *Jetson TX2* (5,181 J), while the fastest runtime was achieved on the *Hi1616* microserver (111 s). Comparing the two in consideration of the respective other value, the *TX2* takes five times longer, while the *Hi1616* microserver consumes about 2.7 times more energy. When focusing on only one of these values, it is straightforward to choose the most suitable platform. However, when both factors are of equal importance, the decision becomes significantly more challenging. The most balanced ratio between runtime and energy consumption was achieved on the platforms we looked at when two or three *Jetson TX2* were used in parallel.

4.2. Summary and Future Work

We have presented our optimized version of the *SPADE* method's pattern mining flow in this work, using a custom-tailored *FP-Growth* implementation. Using a data set containing spike trains from experimental data, we performed our evaluation on a typical *SPADE* use case. We showed how our implementation handles different input settings by varying the parameter configuration for the minimum size and occurrence number. Furthermore, using our distributed approach on up to four TX2, a near-linear scaling for the part computed in parallel, i.e., the *FP-Growth*-based pattern mining, was achieved. In this work, our primary focus was the acceleration of the pattern mining and result filtering tasks, as they account for between 85 and 90% of the overall runtime. On the one hand, we showed that our improved version was, depending on the platform used, between 27 and 200 times faster compared to the original implementation. On the other hand, all platforms' energy consumption was up to two orders of magnitude lower than the original *FP-Growth* version currently used in *SPADE*. The highest energy efficiency was achieved by the embedded devices, which, when executing our flow, required only between 41% and 24% of the energy consumed by the workstation, running the multi-threaded version of our optimized implementation. Furthermore, the execution on embedded devices is now possible; previously, this was prevented by the high memory requirements.

In the future, we intend to further improve our flow by looking at ways to accelerate the sections currently executed sequentially, i.e., the closed pattern detection and the data conversion to Python. Depending on the number of patterns found by *FP-Growth*, these parts become the bottleneck, as mentioned in section 3.5. An example of this can be seen in *job 2*, where, depending on the platform, these tasks account for a significant portion of the job's and the overall runtime. For this reason, we will be looking into implementations for parallel closed pattern detection, e.g., the propositions made by Lucchese et al. (2007) and Huynh et al. (2017). Besides the acceleration of these sections, we plan to integrate the filtering even deeper into our *FP-Growth* implementation, e.g., by marking all items that reside in the first bin of their respective windows. This could enable even faster filtering and, in addition, might also reduce the number of header table entries to check. Furthermore, we want to evaluate the usability of GPU-based *FP-Growth* and closed pattern detection implementations, like the ones described in Wang and Yuan (2014), Jiang and Meng (2017), and Wu et al. (2019). At the same time, it will also be of interest to analyze the applicability of a heterogeneous CPU and GPU implementation, i.e., where the workload is shared between the CPU and the GPU. This is something from which especially the embedded devices could significantly benefit, as their GPU is directly connected to the DDR memory allowing for fast data exchange. We also intend to further improve our distributed computing setup performance by exploring different strategies like the ones proposed by Li et al. (2008) and Chen et al. (2009). Additionally, we suggest to investigate different pattern mining algorithms, e.g., LCM[24] (Uno et al., 2004) or DPT[25] (Qu et al., 2020), and evaluate their performance in the given use case. Finally, we want to analyze further the *SPADE* code surrounding *FP-Growth* to find more potential improvement points. At the same time, it might be worthwhile to analyze the *SPADE* code as a whole and identify bottlenecks that can be accelerated using custom C/C++ modules.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data, together with the source code for the Python module presented in this paper can be found at: https://github.com/fporrmann/FPG. The accelerated version of the SPADE method presented in this article is included in the Elephant[26] GitHub project at: https://github.com/NeuralEnsemble/elephant and will be featured starting from the official release 0.11.0. For an interactive demonstration on how to use the SPADE method, please refer to the tutorial page of the Elephant documentation available at: http://tutorials.python-elephant.org.

AUTHOR CONTRIBUTIONS

FP: conceptualization, software implementation and optimization, testing, evaluation, and writing original draft preparation. SP and FP: visualization. AS: data preparation. FP, SP, AS, AK, MD, JH, and UR: writing review and editing. JH and UR: project administration and funding acquisition. All authors have read and agreed to the published version of the manuscript.

FUNDING

This publication was supported by the VEDLIoT and LEGaTO projects, which received funding from the European Union's Horizon 2020 research and innovation program under grant agreements Nos. 957197 and 780681. Besides, the work was supported by the PhD program Design of Flexible Work Environments–Human-Centric Use of Cyber-Physical Systems in Industry 4.0, funded by the North Rhine-Westphalian funding scheme Forschungskolleg and affiliated to the Research Institute for Cognition and Robotics (CoR-Lab), Bielefeld University. This project has also received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreement No. 945539 (Human Brain Project SGA3) and the Helmholtz Association Initiative and Networking Fund under project number ZT-I-0003 and the VSR computation grant JINB33, Jülich.

ACKNOWLEDGMENTS

We thank Prof. Dr.-Ing. Mario Porrmann from Osnabrück University for his comments that greatly improved the manuscript. We also thank Kevin Mika and René Griessl from Bielefeld University for their assistance during the performance and energy efficiency evaluation. We acknowledge the financial support of the German Research Foundation (DFG) and the Open Access Publication Fund of Bielefeld University for the article processing charge.

[24] **L**inear time **C**losed itemset **M**iner.

[25] **D**ynamic **P**refix **T**ree.

[26] RRID:SCR003833; python-elephant.org.

REFERENCES

Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. *ACM SIGMOD Rec*. 22, 207–216. doi: 10.1145/170036.170072

Agrawal, R., and Srikant, R. (1994). "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94* (San Francisco, CA: Morgan Kaufmann Publishers Inc.), 487–499.

Bin, Z., and Li, J. (2008). An improved algorithm based on FP-growth. *J. Pinddingshan* 17, 9–12.

Borgelt, C., Braune, C., Loewe, K., and Kruse, R. (2015). "Mining frequent parallel episodes with selective participation," in *2015 Conference of the International Fuzzy Systems Association and the European Society for Fuzzy Logic and Technology (IFSA-EUSFLAT-15)* (Gijón: Atlantis Press). doi: 10.2991/ifsa-eusflat-15.2015.97

Borgelt, C., and Picado-Muiño, D. (2013). "Finding frequent patterns in parallel point processes," in *Advances in Intelligent Data Analysis XII*, eds A. Tucker, F. Höppner, A. Siebes, and S. Swift (Berlin; Heidelberg: Springer), 116–126. doi: 10.1007/978-3-642-41398-8_11

Brochier, T., Zehl, L., Hao, Y., Duret, M., Sprenger, J., Denker, M., et al. (2018). Massively parallel recordings in macaque motor cortex during an instructed delayed reach-to-grasp task. *Sci. Data* 5:180055. doi: 10.1038/sdata.2018.55

Chen, M., Gao, X., and Li, H. (2009). "An efficient parallel FP-growth algorithm," in *2009 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery* (Piscataway, NJ: IEEE). doi: 10.1109/CYBERC.2009.5342148

Chen, X., Wang, F., Fernandez, E., and Roelfsema, P. R. (2020). Shape perception via a high-channel-count neuroprosthesis in monkey visual cortex. *Science* 370, 1191–1196. doi: 10.1126/science.abd7435

Dean, J., and Ghemawat, S. (2004). "Mapreduce: Simplified data processing on large clusters," in *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, eds E. A. Brewer and P. Chen (San Francisco, CA).

Denker, M., Yegenoglu, A., and Grün, S. (2018). Collaborative HPC-enabled workflows on the HBP Collaboratory using the Elephant framework. *Neuroinformatics*. doi: 10.12751/incf.ni2018.0019

Diana, G., Sainsbury, T. T., and Meyer, M. P. (2019). Bayesian inference of neuronal assemblies. *PLoS Comput. Biol*. 15:e1007481. doi: 10.1371/journal.pcbi.1007481

Ganter, B., and Wille, R. (1999). *Formal Concept Analysis*. Berlin; Heidelberg: Springer. doi: 10.1007/978-3-642-59830-2

Gerstein, G. L., Williams, E. R., Diesmann, M., Grün, S., and Trengove, C. (2012). Detecting synfire chains in parallel spike data. *J. Neurosci. Methods* 206, 54–64. doi: 10.1016/j.jneumeth.2012.02.003

Grahne, G., and Zhu, J. (2003). "Efficiently using prefix-trees in mining frequent itemsets," in *Proceeding of the ICDM'03 International Workshop on Frequent Itemset Mining Implementations (FIMI'03)* (Melbourne, VIC), 123–132.

Grün, S., Diesmann, M., and Aertsen, A. (2002a). Unitary events in multiple single-neuron spiking activity: I. Detection and significance. *Neural Comput*. 14, 43–80. doi: 10.1162/089976602753284455

Grün, S., Diesmann, M., and Aertsen, A. (2002b). Unitary events in multiple single-neuron spiking activity: II. Nonstationary data. *Neural Comput*. 14, 81–119. doi: 10.1162/089976602753284464

Gutzen, R., von Papen, M., Trensch, G., Quaglio, P., Grün, S., and Denker, M. (2018). Reproducible neural network simulations: statistical methods for model validation on the level of network activity data. *Front. Neuroinform*. 12:90. doi: 10.3389/fninf.2018.00090

Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. *ACM SIGMOD Rec*. 29, 1–12. doi: 10.1145/335191.335372

Harris, K. (2005). Neural signatures of cell assembly organization. *Nat. Rev. Neurosci*. 5, 339–407. doi: 10.1038/nrn1669

Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. New York, NY: John Wiley & Sons.

Huynh, B., Vo, B., and Snasel, V. (2017). An efficient parallel method for mining frequent closed sequential patterns. *IEEE Access* 5, 17392–17402. doi: 10.1109/ACCESS.2017.2739749

Jiang, H., and Meng, H. (2017). "A parallel FP-growth algorithm based on GPU," in *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)* (Piscataway, NJ: IEEE). doi: 10.1109/ICEBE.2017.24

Juavinett, A. L., Bekheet, G., and Churchland, A. K. (2019). Chronically implanted neuropixels probes enable high-yield recordings in freely moving mice. *eLife* 8:e47188. doi: 10.14224/1.38304

Jun, J. J., Steinmetz, N. A., Siegle, J. H., Denman, D. J., Bauza, M., Barbarits, B., et al. (2017). Fully integrated silicon probes for high-density recording of neural activity. *Nature* 551, 232–236. doi: 10.1038/nature24636

Li, H., Wang, Y., Zhang, D., Zhang, M., and Chang, E. Y. (2008). "Pfp," in *Proceedings of the 2008 ACM conference on Recommender systems - RecSys 08* (New York, NY: ACM Press). doi: 10.1145/1454008.1454027

Lopes-dos Santos, V., Ribeiro, S., and Tort, A. B. (2013). Detecting cell assemblies in large neuronal populations. *J. Neurosci. Methods* 220, 149–166. doi: 10.1016/j.jneumeth.2013.04.010

Lucchese, C., Orlando, S., and Perego, R. (2007). "Parallel mining of frequent closed patterns: harnessing modern computer architectures," in *Seventh IEEE International Conference on Data Mining (ICDM 2007)* (Omaha, NE: IEEE). doi: 10.1109/ICDM.2007.13

Oleksiak, A., Kierzynka, M., Piatek, W., Agosta, G., Barenghi, A., Brandolese, C., et al. (2017). M2DC–modular microserver DataCentre with heterogeneous hardware. *Microprocess. Microsyst*. 52, 117–130. doi: 10.1016/j.micpro.2017.05.019

Oleksiak, A., Kierzynka, M., Porrmann, M., Hagemeyer, J., Griessl, R., Peykanu, M., et al. (2019). *M2DC-A Novel Heterogeneous Hyperscale Microserver Platform*. Cham: Springer International Publishing AG. doi: 10.1007/978-3-319-92792-3_6

Picado-Muiño, D., Borgelt, C., Berger, D., Gerstein, G. L., and Grün, S. (2013). Finding neural assemblies with frequent item set mining. *Front. Neuroinform*. 7:9. doi: 10.3389/fninf.2013.00009

Picado-Muiño, D., Castro León, I., and Borgelt, C. (2012). "Fuzzy frequent pattern mining in spike trains," in *Advances in Intelligent Data Analysis XI*, eds J. Hollmén, F. Klawonn, and A. Tucker (Berlin; Heidelberg: Springer), 289–300. doi: 10.1007/978-3-642-34156-4_27

Pipa, G., Wheeler, D. W., Singer, W., and Nikolie, D. (2008). NeuroXidence: reliable and efficient analysis of an excess or deficiency of joint-spike events. *J. Comput. Neurosci*. 25, 64–88. doi: 10.1007/s10827-007-0065-3

Qu, J.-F., Hang, B., Wu, Z., Wu, Z., Gu, Q., and Tang, B. (2020). Efficient mining of frequent itemsets using only one dynamic prefix tree. *IEEE Access* 8, 183722–183735. doi: 10.1109/ACCESS.2020.3029302

Quaglio, P., Rostami, V., Torre, E., and Grün, S. (2018). Methods for identification of spike patterns in massively parallel spike trains. *Biological Cybernetics* 112(1-2):57–80. doi: 10.1007/s00422-018-0755-0

Quaglio, P., Yegenoglu, A., Torre, E., Endres, D. M., and Grün, S. (2017). Detection and evaluation of spatio-temporal spike patterns in massively parallel spike train data with SPADE. *Front. Comput. Neurosci*. 11:41. doi: 10.3389/fncom.2017.00041

Riehle, A., Wirtssohn, S., Grün, S., and Brochier, T. (2013). Mapping the spatio-temporal structure of motor cortical LFP and spiking activities during reach-to-grasp movements. *Front. Neural Circ*. 7:48. doi: 10.3389/fncir.2013.00048

Russo, E., and Durstewitz, D. (2017). Cell assemblies at multiple time scales with arbitrary lag constellations. *eLife* 6:19428. doi: 10.7554/eLife.19428

Shi, X., Chen, S., and Yang, H. (2017). "DFPS: distributed FP-growth algorithm based on spark," in *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)* (Chongqing: IEEE). doi: 10.1109/IAEAC.2017.8054308

Steinmetz, N. A., Koch, C., Harris, K. D., and Carandini, M. (2018). Challenges and opportunities for large-scale electrophysiology with neuropixels probes. *Curr. Opin. Neurobiol*. 50, 92–100. doi: 10.1016/j.conb.2018.01.009

Stella, A., Quaglio, P., Torre, E., and Grün, S. (2019). 3d-SPADE: significance evaluation of spatio-temporal patterns of various temporal extents. *Biosystems* 185:104022. doi: 10.1016/j.biosystems.2019.104022

Tektronix (2006). *TCP0030 120 MHz, 30 A AC/DC Current Probe Instruction Manual*. Tektronix.

Torre, E., Picado-Muiño, D., Denker, M., Borgelt, C., and Grün, S. (2013). Statistical evaluation of synchronous spike patterns extracted by frequent item set mining. *Front. Comput. Neurosci*. 7:132. doi: 10.3389/fncom.2013.00132

Torre, E., Quaglio, P., Denker, M., Brochier, T., Riehle, A., and Grün, S. (2016). Synchronous spike patterns in macaque motor cortex during an instructed-delay reach-to-grasp task. *J. Neurosci*. 36, 8329–8340. doi: 10.1523/JNEUROSCI.4375-15.2016

Trensch, G., Gutzen, R., Blundell, I., Denker, M., and Morrison, A. (2018). Rigorous neural network simulations: a model substantiation methodology for increasing the correctness of simulation results in the absence of experimental validation data. *Front. Neuroinform*. 12:81. doi: 10.3389/fninf.2018.00081

Uno, T., Kiyomi, M., and Arimura, H. (2004). "LCM ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets," in *FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, eds B. Goethals and M. J. Zaki (Brighton). doi: 10.1145/1133905.1133916

Wang, F., and Yuan, B. (2014). "Parallel frequent pattern mining without candidate generation on GPUs," in *2014 IEEE International Conference on Data Mining Workshop* (Shenzhen: IEEE). doi: 10.1109/ICDMW.2014.71

Watanabe, K., Haga, T., Tatsuno, M., Euston, D. R., and Fukai, T. (2019). Unsupervised detection of cell-assembly sequences by similarity-based clustering. *Front. Neuroinform*. 13:39. doi: 10.3389/fninf.2019.00039

Wicaksono, D., Jambak, M. I., and Saputra, D. M. (2020). "The comparison of apriori algorithm with preprocessing and FP-growth algorithm for finding frequent data pattern in association rule," in *Proceedings of the Sriwijaya International Conference on Information Technology and Its Applications (SICONIAN 2019)* (Palembang: Atlantis Press). doi: 10.2991/aisr.k.200424.047

Williams, A. H., Poole, B., Maheswaranathan, N., Dhawale, A. K., Fisher, T., Wilson, C. D., et al. (2020). Discovering precise temporal patterns in large-scale neural recordings through robust and interpretable time warping. *Neuron* 105, 246.e8–259.e8. doi: 10.1016/j.neuron.2019.10.020

Wu, Y.-C., Yeh, M.-Y., and Kuo, T.-W. (2019). "Fast frequent pattern mining without candidate generations on GPU by low latency memory allocation," in *2019 IEEE International Conference on Big Data (Big Data)* (Los Angeles, CA: IEEE). doi: 10.1109/BigData47090.2019.9006541

Xia, D., Lu, X., Li, H., Wang, W., Li, Y., and Zhang, Z. (2018). A MapReduce-based parallel frequent pattern growth algorithm for spatiotemporal association analysis of mobile trajectory big data. *Complexity* 2018, 1–16. doi: 10.1155/2018/2818251

Yegenoglu, A., Quaglio, P., Torre, E., Grün, S., and Endres, D. (2016). "Exploring the usefulness of formal concept analysis for robust detection of spatio-temporal spike patterns in massively parallel spike trains," in *Graph-Based Representation and Reasoning, Bd. 9717 (Lecture Notes in Computer Science, 9717)*, eds O. Haemmerlé, G. Stapleton, and C. F. Zucker (Cham: Springer International Publishing), 3–16.

Zaiane, O., El-Hajj, M., and Lu, P. (2001). "Fast parallel association rule mining without candidacy generation," in *Proceedings 2001 IEEE International Conference on Data Mining* (San Jose, CA: IEEE). doi: 10.1109/ICDM.2001.989600

Zaki, M. (2000). Scalable algorithms for association mining. *IEEE Trans. Knowledge Data Eng*. 12, 372–390. doi: 10.1109/69.846291

Zhou, L., Zhong, Z., Chang, J., Li, J., Huang, J. Z., and Feng, S. (2010). "Balanced parallel FP-growth with MapReduce," in *2010 IEEE Youth Conference on Information, Computing and Telecommunications* (Beijing: IEEE). doi: 10.1109/YCICT.2010.5713090

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers in Neuroinformatics

ORIGINAL RESEARCH
published: 07 January 2022
doi: 10.3389/fninf.2021.766697

ConGen—A Simulator-Agnostic Visual Language for Definition and Generation of Connectivity in Large and Multiscale Neural Networks

Patrick Herbers[1†], *Iago Calvo*[2†], *Sandra Diaz-Pier*[1*†], *Oscar D. Robles*[2,3], *Susana Mata*[2,3], *Pablo Toharia*[3,4], *Luis Pastor*[2,3], *Alexander Peyser*[1‡], *Abigail Morrison*[1,5,6] and *Wouter Klijn*[1*]

[1] *Simulation and Data Lab Neuroscience, Jülich Supercomputing Centre, Institute for Advanced Simulation, JARA, Forschungszentrum Jülich GmbH, Jülich, Germany,* [2] *Department of Computer Science and Computer Architecture, Lenguajes y Sistemas Informáticos y Estadística e Investigación Operativa, Rey Juan Carlos University, Madrid, Spain,* [3] *Center for Computational Simulation, Universidad Politécnica de Madrid, Madrid, Spain,* [4] *DATSI, ETSIINF, Universidad Politécnica de Madrid, Madrid, Spain,* [5] *Institute of Neuroscience and Medicine and Institute for Advanced Simulation and JARA BRAIN Institute I, Jülich Research Centre, Jülich, Germany,* [6] *Computer Science 3 - Software Engineering, RWTH Aachen University, Aachen, Germany*

OPEN ACCESS

Edited by:
Omar Awile,
Swiss Federal Institute of Technology
Lausanne, Switzerland

Reviewed by:
Caglar Cakan,
Technical University of Berlin,
Germany
Marcel Stimberg,
Sorbonne Université, France

****Correspondence:***
Wouter Klijn
w.klijn@fz-juelich.de
Sandra Diaz-Pier
s.diaz@fz-juelich.de

† These authors have contributed equally to this work

‡ Present address:
Currently at Google,
Munich, Germany

Received: *29 August 2021*
Accepted: *30 November 2021*
Published: *07 January 2022*

Citation:
Herbers P, Calvo I, Diaz-Pier S, Robles OD, Mata S, Toharia P, Pastor L, Peyser A, Morrison A and Klijn W (2022) ConGen—A Simulator-Agnostic Visual Language for Definition and Generation of Connectivity in Large and Multiscale Neural Networks.
Front. Neuroinform. 15:766697.
doi: 10.3389/fninf.2021.766697

An open challenge on the road to unraveling the brain's multilevel organization is establishing techniques to research connectivity and dynamics at different scales in time and space, as well as the links between them. This work focuses on the design of a framework that facilitates the generation of multiscale connectivity in large neural networks using a symbolic visual language capable of representing the model at different structural levels—ConGen. This symbolic language allows researchers to create and visually analyze the generated networks independently of the simulator to be used, since the visual model is translated into a simulator-independent language. The simplicity of the front end visual representation, together with the simulator independence provided by the back end translation, combine into a framework to enhance collaboration among scientists with expertise at different scales of abstraction and from different fields. On the basis of two use cases, we introduce the features and possibilities of our proposed visual language and associated workflow. We demonstrate that ConGen enables the creation, editing, and visualization of multiscale biological neural networks and provides a whole workflow to produce simulation scripts from the visual representation of the model.

Keywords: multiscale simulation, large scale simulation, visual language, neural networks, connectivity generation, connectome

1. INTRODUCTION

The brain has a multilevel organization, with anatomical and dynamic features spanning orders of magnitudes. Understanding the nature of its components and how they are connected with each other is critical to unraveling this complexity (Evanko and Pastrana, 2013; Morgan and Lichtman, 2013; Peyser et al., 2019), for both healthy and diseased brains (e.g., Chen et al., 2021). Indeed, connectivity is an essential aspect defining the functionality at all organizational scales of the brain (Sporns et al., 2005).

The 21st century has seen multiple interdisciplinary research initiatives initiated to address this important topic (Collins and Prabhakar, 2013; Markram et al., 2015); however, despite advances and efforts toward standardization in this field (Gadde et al., 2012; Gorgolewski et al., 2016), there is no consistent way to represent, visualize, explore, and generate connectivity for simulation or analysis across different scales. Consequently, developing the tools to support investigations of multiscale functional organization remains an open challenge.

A central method in such investigations is numerical simulation of the brain. Existing simulation engines capture the brain behavior at different levels of detail: detailed multi-compartment simulations (e.g., Arbor—Akar et al., 2019; Abi Akar et al., 2021, Neuron—Carnevale and Hines, 2006), point neuron simulations (e.g., NEST—Jordan et al., 2019, Brian—Stimberg et al., 2019) or whole brain level simulations (e.g., The Virtual Brain—TVB Sanz Leon et al., 2013). By exploiting high performance computing we are now able to simulate large scale networks as well as those which represent the brain at different scales simultaneously. However, the absence of methods to create, and explore complex connectivity in these types of networks limits investigations in the relationships between connectivity and function.

Creating a framework that enables simulation of large and multiscale models of heterogeneous neuron populations is only possible by making use of well-defined interfaces, either new or existing. These interfaces must allow weak coupling between software systems and the necessary front ends to interact with them, but at the same time be able to leverage the native functions of the simulation engines and their inherent scaling capabilities. The development of tools and standards which take advantage of these interfaces will additionally enable the comparison of performance and function metrics between different simulation engines. This will allow the formulation of more robust scientific conclusions and move the field of computational neuroscience forward. The implementation of easy to use and flexible tools for benchmarking different simulation tools remains a critical and still unfulfilled requirement by the neuroscience community.

This work focuses on the design and implementation of ConGen, a framework that facilitates the generation of connectivity in large neural networks using a new symbolic visual language capable of representing the model at different structural levels. This symbolic language is specifically designed to provide researchers with a tool to represent and explore the connectivity in models of multiscale and large scale networks. ConGen provides an agnostic way to represent models and later instantiate them with specific simulation frameworks. The connectomes represented in the visual language can be exported to the standardized network representation format NeuroML (Gleeson et al., 2010). These descriptions can be used directly by simulation engines which support the format. ConGen adds functionality with a back end, also interfaced with NeuroML.

The ConGen back end enables interfacing with efficient connection generation approaches (e.g., Djurfeldt et al., 2014) and allows users to launch simulations using the simulation engines' native scaling capabilities. For convenience, the ConGen back end encapsulates a set of basic templates allowing users to generate the connectivity using the standard description. The ConGen back end also includes a number of thin simulator-specific interfaces, enabling basic launching, i.e., using default model parameters, on the target simulator. Users can edit and extend the thin simulator-specific scripts in the ConGen back end to define their own simulation and model parameters. Currently the ConGen back end supports execution of the NEST and TVB simulators, as well as the generation of EBRAINS co-simulation model scripts ready for execution by external tooling. However, due to its modular design it is possible to easily extend the back end to support other simulators.

With this work we provide a bridge between a simple, yet expressive, visual language (see section 2.1.1) embedded into a simulator-agnostic graphical interface, simulation frameworks, and high performance computing infrastructure. By providing a language to describe connectivity in a simulator-agnostic way, ConGen also represents a new platform to assist benchmarking using a generic description of networks based on model description standards. As such, ConGen helps to address the unfulfilled requirement for an easy to use and flexible tool for benchmarking.

This paper is structured as follows: first, we present the state of the art in connectivity visualization and representation techniques, as well as standards for network description. Then, we describe the front end which implements the symbolic visual language: ConGen. Afterwards, we discuss the implementation of the back end which takes the standard output representation from the visual language and translates it into a model instance in one or multiple simulators. In the results section we show two use cases for this framework. The first refers to the well-known model of the cortical mircrocircuit by Potjans and Diesmann (2014) and the second is a multiscale model which combines a simulation in TVB and NEST. Finally, we discuss the use cases and the current limitations of the framework, provide some conclusions, and point toward future directions.

1.1. State of the Art

1.1.1. Visual Representations of Connectivity

Connectivity matrices have long been used as a way to represent connections in the brain. For example, the work of Rubinov and Sporns (2010) presents a Matlab Toolbox intended to generate connectomes at the scale of brain regions using this type of representation. Here, binary entries in the matrix indicate the presence or absence of connections; real-valued entries can be used to represent magnitudes regarding correlational or causal interactions. Although the authors state that the neuroimaging methods available for them were unable to directly detect anatomical or causal directionality, the matrices produced using the Toolbox can incorporate this information if it is available. Mijalkov et al. (2017) created another Matlab Toolbox that allows the user to create visualizations mainly based in connectivity matrices derived from different neuroimaging modalities with the aim to study large scale brain connectivity applying techniques from graph analysis theory.

An alternative representation of connectivity is given by a Connectivity Pattern Table (CPT), a 2D schematic and compact representation intended to shown the spatial structure of connections as well as their strength, proposed by Nordlie and Plesser (2010). The main features of CPTs are a clutter-free presentation of connectivity, the ability to represent connectivity at several levels of aggregation and a high information contents regarding the spatial structure of connectivity.

Other approaches have focused on morphologically detailed connectivity. For example, NeuroLines (Al-Awami et al., 2014) is a multiscale abstract visualization technique for the analysis of neurites and their connections. Here, each neurite is represented as a tree structure based on 3D data of their morphology. Once a synapse is selected, all other synapses linked to the same neurites are visually highlighted for contextual information. In related work, Böttger et al. (2014) developed an edge bundling method which depicts clear and high-resolution pictures of functional brain connectivity data across functional networks in the 3D brain space.

Additional tools address MEG/EEG data import and preprocessing. NeuroPycon (Meunier et al., 2020) is a Python toolbox for the visualization of connectivity analysis in MEG sensors. The visualization is built from the sensor-level connectivity matrix obtained from the computation of the coherence among MEG sensors in alpha band. The colors of the connectivity edges indicate the strength of the connection, and the node size and color represent the number of connections per node. Similarly, Espinoza-Valdez et al. (2021) presented a 3D visualizer of the brain connectivity for EEG data. The selection of electrodes is performed in a dynamic way; graph theory is then applied to characterize brain connectivity in 3D images.

Finally, Fujiwara et al. (2017) introduced a visual analytics system to enable neuroscientists to compare networks. The system provides visual tools for comparison at both individual and population levels. The main visualization techniques they use are based on representations of connectivity and node-linkage matrices (both 2D and 3D).

1.1.2. Abstract Representations of Connectivity

Unfortunately, often the descriptions of network model connectivity do not adhere to any standards (Nordlie et al., 2009). Model definitions rely on a combination of complex text descriptions, pieces of pseudo-code or simulator-specific code, tables, and connectivity patterns without formal definitions. Consequently, ambiguities in the model description make it difficult to independently reproduce the network, or port it from one simulation environment to another (Pauli et al., 2018).

To provide a formal standard that can be used to convey the connectivity of a model, not only in written text and formulas, but also among neuronal simulators, Djurfeldt (2012) developed the Connection Set Algebra (CSA): a mathematical representation of connections between populations of neurons based on set algebra. With this abstract formalism, a connectivity pattern can be defined independently of the implementation by the various simulators. This independence is an important aspect of the modular nature of CSA, allowing it, in principle, to be used in combination with any simulator. The connection with the simulators is formalized in the Connection Generation Interface (CGI; Djurfeldt et al., 2014). The CGI allows the simulator to query connections from the linked connection generator. Both the simulator and the connection generator need to implement the interface.

1.1.3. Standardized Network Model Descriptions

A number of domain languages exist to describe networks at different scales, notably PyNN (Davison et al., 2009), NeuroML (Gleeson et al., 2010), and NineML (Raikov et al., 2011). PyNN is a Python based simulator-independent language. It supports modeling at multiple levels of abstraction. The instruction set of each simulator and PyNN code can be mixed, so models described in PyNN can still access features specific to individual simulation engines. Importantly for our work, PyNN implements the CGI, which allows connection generation using the CSA. During the development of ConGen, PyNN did not support NeuroML, but a NeuroML file export for networks generated with PyNN has since been added. While PyNN enables easier interfacing with various simulators, it has been designed primarily as a scripting tool. No visual tools are available for PyNN, instead network creation follows procedural instructions.

NeuroML is a simulator-independent XML-based formalism that is supported by a variety of neuroscience tools and supports a more biophysically detailed level of modeling than PyNN. The standard consists of three levels, which are built hierarchically and provide a standard for describing morphologically detailed neurons, spiking neurons and populations of neurons. The most recent version of NeuroML (NeuroML2) combined with LEMS (Cannon et al., 2014) has been developed in order to be able to represent both network structure and model dynamics in a standardized and domain specific fashion.

Finally, NineML is an XML-based modeling language similar to NeuroML formalized in an XML Schema Definition (Raikov et al., 2011). Its primary focus is on definitions on the network level, such as populations and connections; as a consequence of this focus it lacks many of the detailed elements present in NeuroML, e.g., biological cell structures of neurons and synapses.

The computational neuroscience community needs to further use and define standards in order to promote reproducibility and robustness of results. With this in mind, efforts like Open Source Brain (Gleeson et al., 2019) try to integrate graphic user interfaces, model description languages and simulation engines into a cohesive effort to simulate the brain.

1.1.4. Simulation Engines

Simulators are an important tool in computational neuroscience. Simulation engines enable the creation and simulation of models at different scales. They typically provide a language, usually a scripting language, for the user to access the simulator's functions.

Some common spiking neuron simulation tools are NEURON (Carnevale and Hines, 2006), NEST (Jordan et al., 2019), and BRIAN (Stimberg et al., 2019). For a detailed comparison on these and other simulators see Tikidji-Hamburyan et al. (2017). Of the three simulators, NEURON is the one with the longest history and largest user community. Arbor (Akar et al., 2019) is a new simulation framework, developed in the context of the HBP, at the morphologically detailed scale and is designed to take full advantage of new computing architectures and reach high scalability. While NEURON and Arbor are used for detailed cell models, NEST is used to simulate primarily point neurons, and multi compartment neurons with up to three compartments. NEST is optimized for simulations of large scale networks—including up to hundreds of millions of neurons and their synapses—on high performance computers while still having great performance on smaller devices. BRIAN supports simulations of both detailed and large scale networks with a focus of separating model definition and simulator implementation details. For ease of use all these simulators have implemented Python interfaces, or can be controlled using a simulator specific language [e.g., PyNEST (Eppler et al., 2009) for NEST].

Simulations of the whole brain are also possible at a coarse resolution. For example, The Virtual Brain is a simulation framework which allows the representation of the brain using neural mass models and simulate them to generate synthetic Electroencephalography (EEG), Magnetoencephalography (MEG), or Blood Oxygen Level Dependent (BOLD) signals. Another emerging simulator at the whole brain scale level is neurolib (Cakan et al., 2021). Similar to TVB, neurolib provides the end user with a variety of neural mass models, the ability to create networks based on empirical connectivity data and generate simulated signals which can be optimized using parameter fitting methods against empirical data.

2. METHODS

This section describes the two components of the proposed framework: the description of the multiscale connectivity using a symbolic visual language, and the translation of the generated models into simulator-specific instructions.

The ConGen front end provides end users with a standardized description of their models in NeuroML. This description can be directly used by simulation engines which support this standard. For convenience, the ConGen back end encapsulates a set of basic functionality allowing users to read the NeuroML file and generate the connectivity. The ConGen back end includes simulator specific thin interfaces, allowing for basic launching on the target simulator. To increase the compatibility of ConGen to multiple simulators, we have extended the NeuroML scheme used to define the models. Illustrating the flexibility of the toolset, one of our use-case expands on the standard NeuroML interface with new functionality. We show that new simulators and functionality can be added by adapting the ConGen back end. Users can also directly interact with the output of the ConGen front end in NeuroML format to create more complex simulations with detailed model specifications which go beyond basic configuration and connectivity definition e.g., cell model specific parameters, pre and post- processing of input and output data, etc.

2.1. Visual Front End for Connectivity Generation

ConGen facilitates the creation, editing and visualization of multiscale neural networks. Connections can be created and visualized at the desired level of abstraction, and mechanisms for the propagation and aggregation of connectivity along the hierarchy are provided. This approach allows the researcher to generate large scale scenarios capturing global behavior and local details at the same time.

The ConGen front end has been integrated into Neuroscheme (Pastor et al., 2015), a visual framework to guide exploration and knowledge extraction from complex neural scenes. Neuroscheme allows the creation of domains that define the set of elements that conform a neuronal scene. For example, Neuroscheme includes the cortex domain, which provides elements corresponding to the organizational levels of column, minicolumn and neuronal cell, as well as defining the properties associated with each element. ConGen has been conceived and developed as a new domain within Neuroscheme, defining a new set of abstract neural elements (i.e., not corresponding to specific brain areas) and connections between them in order to represent models of large scale and multiscale neural networks.

Neuroscheme offers an environment with multiple views where different representations of the data can be visualized in a coordinated manner. In this way, abstract views can be combined with accurate representations of cellular anatomy. The iconic view of a circuit provides a global, simplified view with summarized or aggregated information, while the realistic view provides all details of the neuronal anatomy and spatial distribution. ConGen has been designed to act as a front end for interactive visual definition of neural connectivity, thereby facilitating the creation and manipulation of neural circuit models. Following a top-down approach, ConGen enables the creation of a hierarchy of super-populations and populations and the specification of their connections by establishing the necessary connectivity parameters. Populations constitute the leaves of the hierarchy and grouping them together gives rise to a superpopulation. In turn, superpopulations can be grouped iteratively, also giving rise to hierarchical superpopulations. **Figure 1** shows a hierarchy of superpopulations and populations. Our approach to interaction and visual representation emphasizes simplicity, depicting views using easy symbolic representations. The models so created can be exported using an extended version of NeuroML for further simulator-specific translation. The following subsection details the operations supported by ConGen.

2.1.1. Creation and Parameterization of a Hierarchical Network Structure

ConGen supports the creation of a neural scene and its connectivity by providing an interface that visually displays the created entities and relationships. Each entity will be represented by a circular shape. Entities of the same type will share the same color (superpopulations, populations, inputs and outputs). The number of inner circles will represent the number of descending levels of a superpopulation and the filling of the horizontal bar will be proportional to the number of neurons in each grouping. By simply right-clicking with the mouse, a context menu appears allowing entities to be created and hierarchically structured. To create one or more super-populations, the user simply sets the number of entities to be created, their name and the other configurable parameters. **Figure 2A** shows the super-population

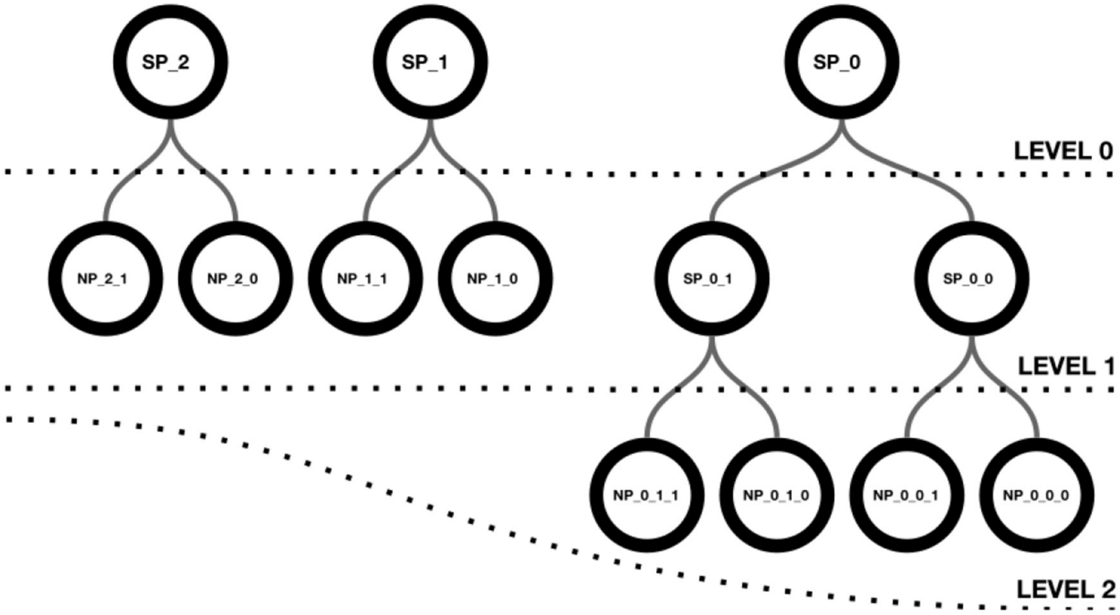


FIGURE 1 | Hierarchical structure of a scene. Level 0 shows three superpopulations that group either populations or descendant superpopulations, as shown in level 1. Superpopulations SP_0_0 and SP_0_1 contain two neuron populations each, as depicted in level 2.

creation panel as well as the visual representation of the three super-populations created. **Figure 2B** illustrates the creation of two populations of neurons within the super-population SP_2. Note that the visual representation of the super-population SP_2, compared to its appearance in **Figure 2A**, now reflects the existence of a hierarchical descendant level (presence of an inner circle) and the number of neurons in the descendant populations (green filling of the horizontal bar).

Continuing the procedure outlined above, the hierarchy initially shown in **Figure 1** can be easily created. **Figure 3A** shows this scene depicted at level 0, composed of three super-populations (SP_0, SP_1 and SP_2). SP_0 in turn contains two child super-populations (SP_0_0 and SP_0_1); each of them, as well as SP_1 and SP_2, containing two populations of neurons. The super-populations can be expanded to show their children, either in the same panel or in a different panel; **Figure 3B** shows the result of expanding all super-populations in a different panel, thus allowing the scene to be visualized at two levels of abstraction simultaneously (level 0 on the left, and level 1 on the right). Similarly, the SP_0_0 and SP_0_1 super-populations can be expanded into a further panel, depicting the scene at the lowest level of abstraction in the right panel of **Figure 3C**.

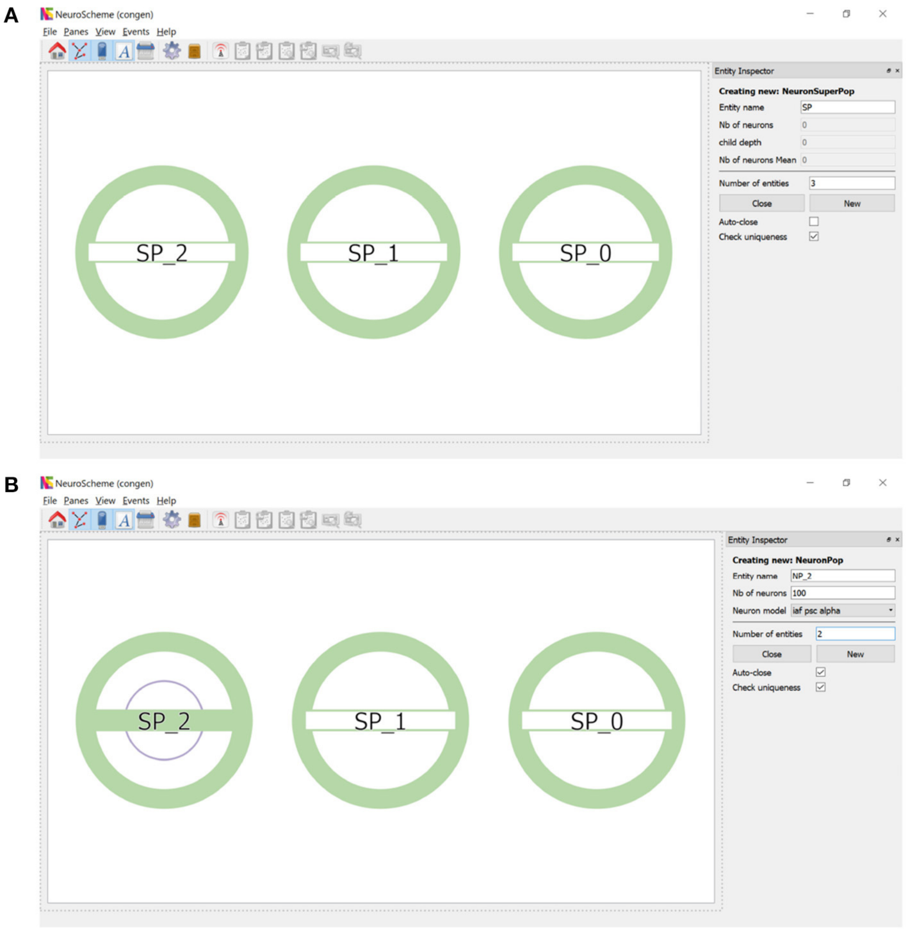
Connections are created by dragging with the mouse from the source population to the target population. **Figure 4A** shows the parameterization options of the connections as well as the context menu that allows auto-connections (i.e., connections of a population to itself) to be added. Each connection is represented by an arrow whose thickness is proportional to the strength of the connection. Since the views shown in the different panels are coordinated, the connections created at the lowest level of abstraction are reflected in an aggregated way in the panels showing the scene at higher levels of abstraction, as shown in **Figure 4B**.

In addition to neuron populations, input and output entities can be created. These entities are external to the hierarchy. Input entities stimulate one or more populations of neurons. An input entity is connected to its target population analogously to the connections between populations. **Figure 4C** shows the result of including an input connected to example populations NP_0_0_1 and NP_0_1_0; note that input entities appear at all levels of abstraction. Output entities, such a measurement devices, can receive a connection from one or more populations of neurons.

2.2. From Visual Representation to Simulation

In this section, we introduce the back end of ConGen, which is used to generate the hierarchical neural network models and interact with the simulation engines (see division of front end/back end in **Figure 5**). The ConGen front end has to serialize the model expressed in ConGen's graphical language by some means. Here, we make use the pre-existing NeuroML standard rather than developing a new declarative language to achieve this goal. Any simulator that supports NeuroML can be considered a potential execution target.

The back end of ConGen consists of a modular translator system. Its purpose is to translate the NeuroML descriptions of the models created in the GUI described in the previous section into simulator-specific code. It is important to highlight that the ConGen back end is not a simulation engine but is able to call functions from different target simulators using the simulator interface. The translation system was designed with the following technical requirements in mind:



Entity Inspector

Creating new: NeuronSuperPop

Entity name: SP
Nb of neurons: 0
child depth: 0
Nb of neurons Mean: 0
Number of entities: 3
Close | New
Auto-close □
Check uniqueness ☑

Entity Inspector

Creating new: NeuronPop

Entity name: NP_2
Nb of neurons: 100
Neuron model: iaf psc alpha
Number of entities: 2
Close | New
Auto-close ☑
Check uniqueness ☑

FIGURE 2 | Creation of super-populations and populations. **(A)** The panel on the right sets the name root of the super-populations (SP in this example) and the number of entities (three in this example) to be created. **(B)** Right clicking on SP_2 allows the creation of descendant neuron populations. The panel on the right sets the name root of the populations (NP_2 in this example) and the number of entities (two in this example) to be created.

1. The translation system should enable the simulation of a model defined in the ConGen GUI by a supported simulator
2. The simulation should be able to be performed in different simulators
3. Adding support for a new simulator should require a low development cost
4. The translation system should be functionally separate from the ConGen GUI
5. The overhead of the system should be low: performance should be close to that of using the simulator directly.

The back end of ConGen is designed to be separate from the GUI (technical requirement 4). This separation allows the front end and back end to be used as independent modules. For example, the front end can run on a desktop computer, while the back end runs and then executes the simulations on a high performance computing system. This modular design also allows individual components to be easily maintained or replaced (technical requirement 3). A modular design requires careful construction of interfaces and data exchange; these are illustrated in **Figure 5**, which shows the overall flow of data from the visual tool to the simulator.

2.2.1. Using NeuroML

The main data exchange between the front end and the back end is via NeuroML: ConGen serializes its visual models to NeuroML files for storage or data exchange. This separation through a common data standard allows the ConGen translator to be entirely independent of the GUI. ConGen uses NeuroML version 1.8.2, which allows networks, layers, and connections to be represented by XML files. At the time that the back end was developed, NeuroML version 2 was still in development. For this reason, the work presented here is based on version 1.8.2, but will be ported to version 2 in the future.

The structure and validity of files is defined by XML schemas, which allows extensions of the described file format. To increase the compatibility of ConGen to multiple simulators, we have extended the NeuroML scheme used for translation. These additions include spatial connectivity and

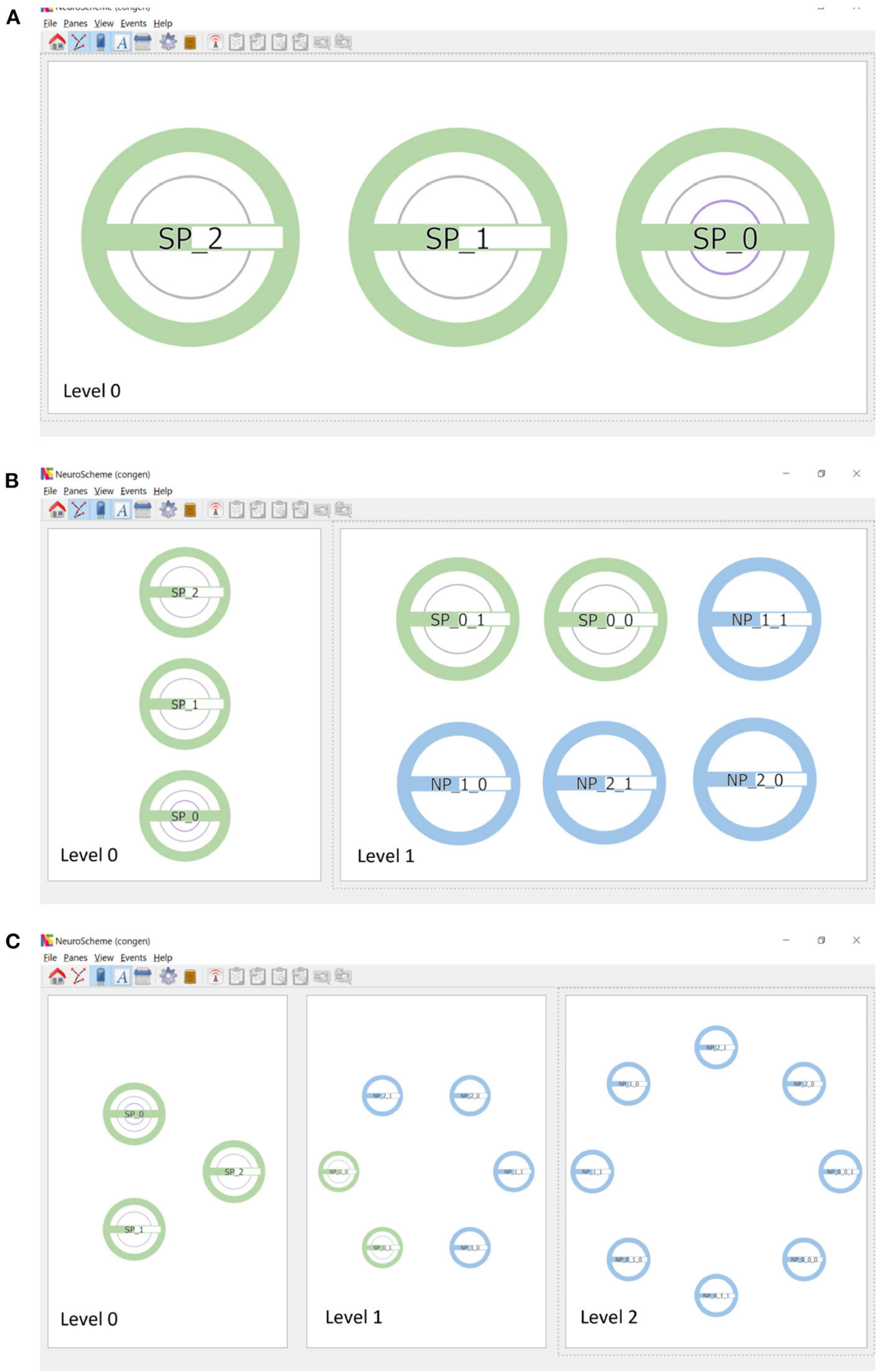
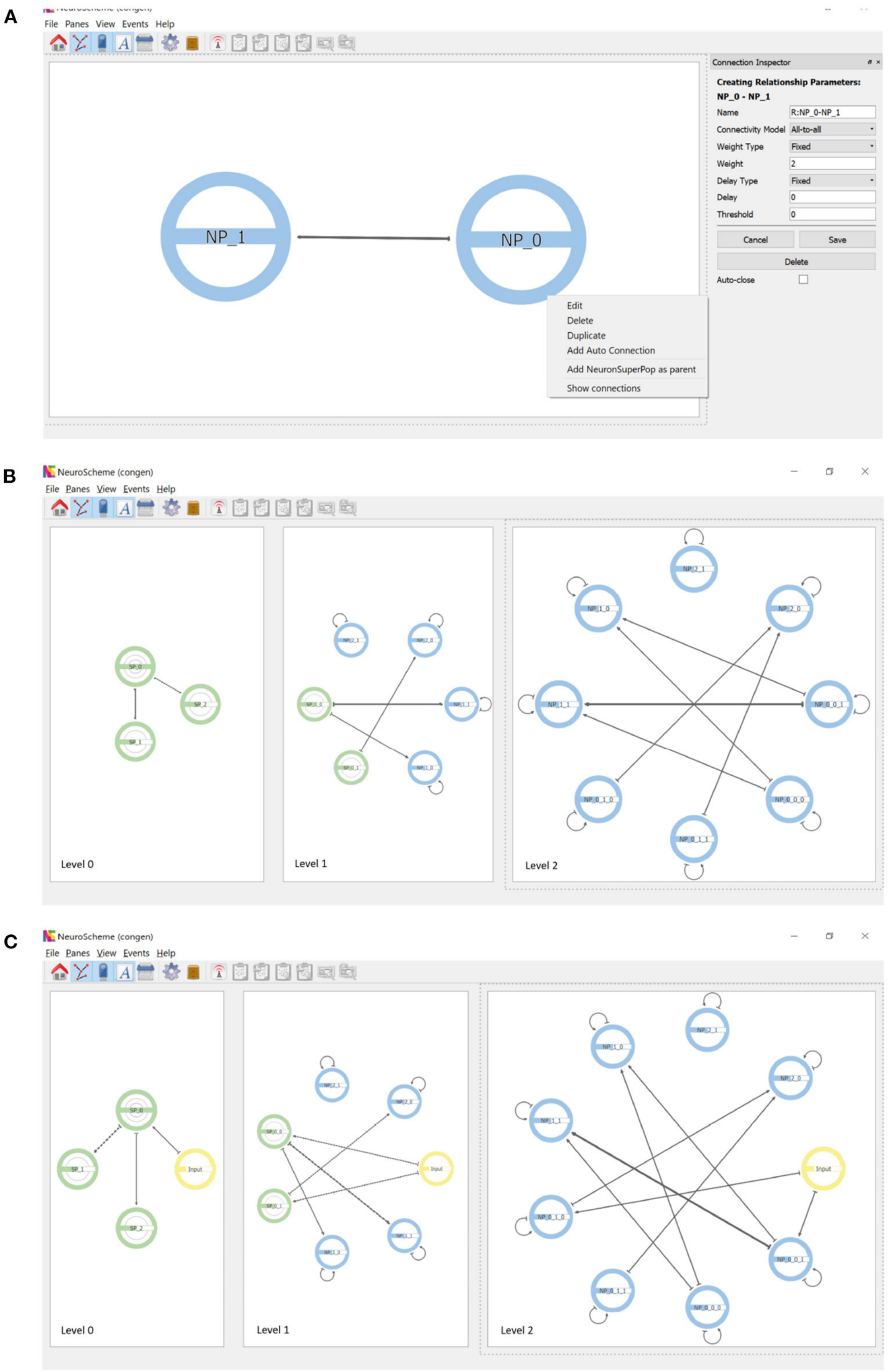


FIGURE 3 | Creation of scenes in ConGen. **(A)** A scene where SP_0 has two hierarchical descendant levels (indicated by the two inner rings) while SP_1 and SP_2 have one hierarchical descendant level each. The green filling of the horizontal bars indicates that SP_1 and SP_2 have the same number of neurons while SP_0 has twice the amount. **(B)** Super-populations can be expanded to visualize the next hierarchical level. Left panel: the three super-populations in a collapsed view. Right panel: The three super-populations have been expanded to show their direct children. **(C)** The panels show the three hierarchical levels of the scene simultaneously. Left: Neuron super-populations at the highest level of abstraction. Middle: The hierarchical entities tree displayed at depth level 2. Right: All entities have been drilled down to show the lowest level of abstraction. Icons have been arranged in a circular layout for convenience for connectivity creation.



Connection Inspector

Creating Relationship Parameters:

NP_0 - NP_1

Name	R:NP_0-NP_1
Connectivity Model	All-to-all
Weight Type	Fixed
Weight	2
Delay Type	Fixed
Delay	0
Threshold	0

FIGURE 4 | Connections and inputs. **(A)** Connections are created by dragging with the mouse from the source to the target population. The panel on the right shows the parameterizable features. Auto-connections can be created through the context menu that appears when right-clicking on a population. **(B)** Connectivity simultaneously displayed at three levels of abstraction. Note the connections of superpopulations represent the aggregation of the connections of their descendant populations. **(C)** Inputs can be created as entities that are external to the hierarchy. Note that inputs appear at every level of abstraction.

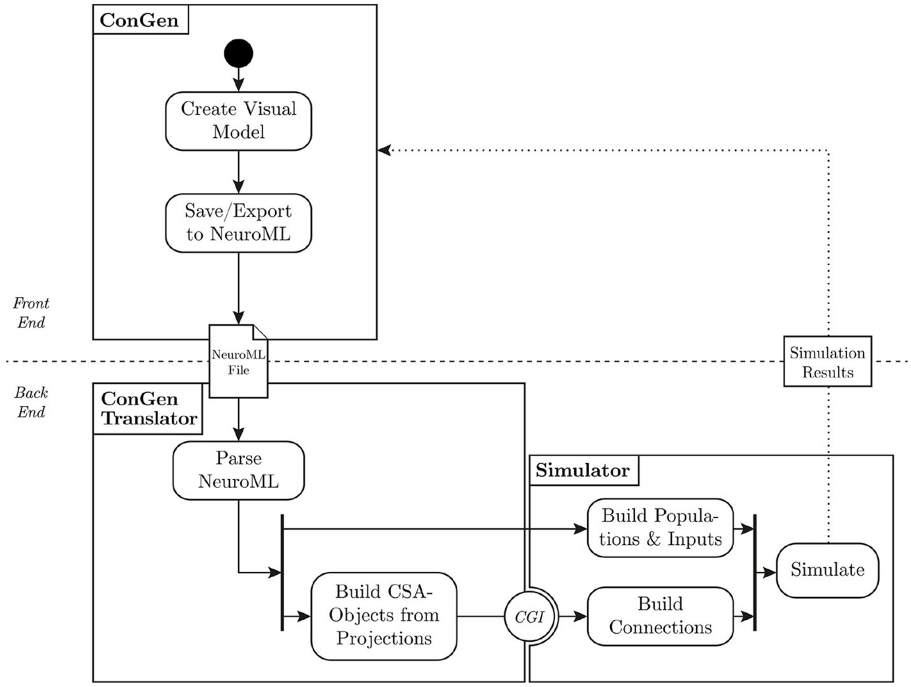


FIGURE 5 | Flow of data from visual representation to simulation. The user creates a model in ConGen and exports it as a NeuroML file. The translator parses the NeuroML file and converts the connectivity into the Connection Set Algebra. The populations and inputs are built directly in the simulator. Using the Connection Generation Interface (CGI), connections are generated from the connection generation library and passed to the simulator, which can then start the simulation. A future extension of this workflow will allow simulation results to be processed and passed back to ConGen (dotted line).

TABLE 1 | XML connectivity pattern tags and their corresponding Python classes and CSA structures.

NetworkML tag	Python class	CSA structure
`<all_to_all/>`	`AllToAll`	`csa.full`
`<one_to_one/>`	`OneToOne`	`csa.oneToOne`
`<fixed_probability/>`	`FixedProbability`	`csa.random(p)`
`<per_cell_connection/>`	`PerCell`	`csa.random(fanIn=n)`
`<gaussian_connectivity_2d/>`	`GaussianSpatialConnectivity`	`gaussian(sigma)`

The Gaussian spatial connectivity has to be combined with CSA's random operator, which samples from the distribution.

synapse parameter distributions. All changes are listed in the **Supplementary Material** (see section Changes to NeuroML).

2.2.2. The ConGen Back End

After a visual model has been saved as a NeuroML file, the file can be used as an input to the ConGen translator. The translator parses the defined network structure, translates the layer and connectivity information, and generates simulator-specific instructions. The translator and the subsequent simulation can be called either independently or invoked directly by ConGen. In the following, we describe the workflow resulting in a set of simulator specific commands which enable the model simulation using specific simulation engines, and how ConGen can be extended to support new simulators.

First, the ConGen back end parses the NeuroML file for translation. XML tags correspond to a Python class, as shown for the example of connectivity patterns in **Table 1**. The parser first reads the populations, then the projections, and lastly the inputs, outputs, and translators. If the model is to be simulated by different simulators at different scales, the back end splits the model into scale-specific sub-models. After the model has been parsed successfully, all string references between objects are replaced with object references. Any errors present in the file (schema mismatch, undefined references) are raised as an exception.

Connectivity patterns are represented by the `ConnectivityPattern` class, which may be subclassed when adding new types of connectivity patterns. When an object of this class is created, the connectivity patterns are transformed to CSA masks, as seen in **Table 1**. Spatial connectivity patterns, based on e.g., 2D euclidean distances, are also supported. To this end, neuron positions can be defined either by neuron instance elements in the NeuroML file or sampled by a template distribution (e.g., Gaussian sampling). Synapse parameters such as weight and delay can be defined in analogously, by explicitly stating neuron instance parameters or by using distributions for whole layer connections. Currently, only Gaussian and Uniform distributions are defined, but additional distributions can be registered by sub-classing the `Distribution` class. In the case of region-to-region connectivity, atlas based connectivity is also supported. Additional details about the implementation of the different connectivity patterns can be found in the **Supplementary Material**.

After parsing, the layers and connections that make up a network are instantiated for the chosen simulator. First, the layers and neuron populations are translated to simulator specific instructions. If the simulator requires neuron positions, any distributions used are sampled at this point. Then, connections between layers are instantiated. Since this connection generation is computationally intensive, we use the Connection Generation Interface (CGI). The CGI calls available internal simulation engine functions to optimally generate connections instead of the high level calls through the Python interface (Djurfeldt et al., 2014). These native calls are typically more efficient (technical requirement 5). We use the `libneurosim` package, which supports the CGI and enables the generation of populations and connections in the simulator (i.e., NEST). Due to the modular nature of the implementation, individual components of ConGen can be easily replaced. For example, the C++ implementation of CSA (`libcsa`) offers increased performance over the Python implementation when generating connectivity, but has limited functionality. Thus, the Python implementation of CSA can be replaced by `libcsa` to accelerate the connectivity generation of large but simple networks.

For convenience to the users and in order to enable the simulation of the generated model in an specific framework, the ConGen back end contains a set of thin layer scripts which can call the target simulation engine (technical requirements 1, 2, and 3). Extending the ConGen back end to support new simulator engines is low effort and consists in the generation of a script which takes the connectivity objects, instantiates the model and, if desired, specifies simulator specific parameters. Pre- and post-processing of input and output data can also be added to this script by the user. At the moment the ConGen back end has a thin layer execution script for NEST and TVB.

The population and cell model parameters have to be defined by the user using simulator specific functions. This can be done either when the user imports the NeuroML file in his or her script or by modifying the thin layer simulator specific file in ConGen in order to add these parameters before model execution. In the current paper we focus specifically on connection generation as this is a complicated task on its own. Setting of model parameters could also be integrated into the ConGen front end, but is left as future work.

ConGen also allows the visual representation and generation of multiscale co-simulation models, and supports the output of multiscale configuration files. The orchestration and deployment of these multiscale simulations is complex (Klijn et al., 2019) and falls outside of the scope of the template based ConGen Translator simulation launching functionality.

3. RESULTS

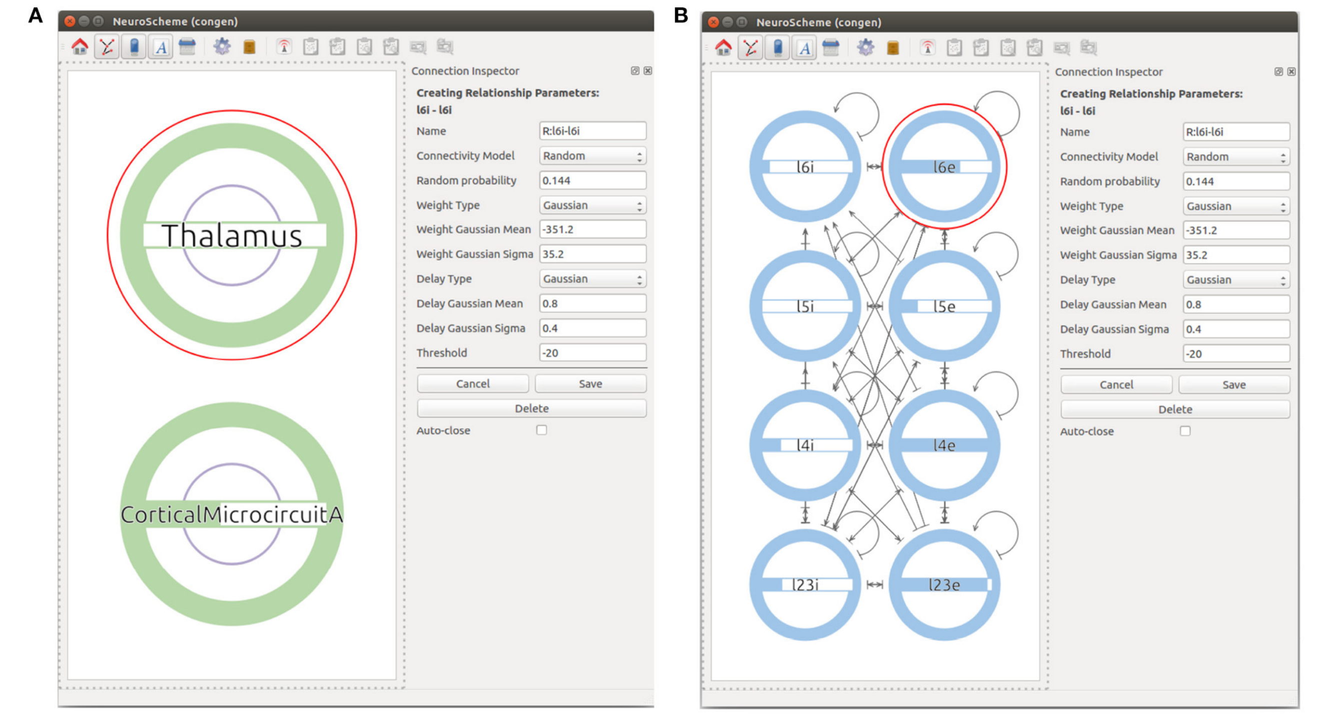
In this section we will describe first two use cases which are used to demonstrate the functionality of ConGen while addressing specific needs from the neuroscience community. Please refer to the **Supplementary Material** section to see where to find and how to execute the example files for these use cases. This section ends with an overview of the supported simulators.

3.1. Use Case 1: The Cortical Microcircuit Model

The Potjans and Diesmann microcircuit model (Potjans and Diesmann, 2014) is an abstraction of 1 mm^3 volume of cortical tissue comprising four layers, each with one excitatory and one inhibitory population. The model has been used to address a variety of scientific questions and is able to show spiking dynamics similar to those observed in real cortical tissue. Due to its importance, we chose this model to test the whole functionality of ConGen, from visual language definition to simulation.

We constructed the model on two levels of abstraction: on the higher level, the representation of the column; on the lower, the representation of the single populations and their connection probabilities. It is important to note that ConGen provides the ability to define how the connectivity should be instantiated by the simulation on a probabilistic or deterministic way. By using CSA below the NeuroML description generated by ConGen, it is possible to create stable, portable and constant instantiations of connectivity patterns which will be the same independently of the target simulator. A step by step description of the model using ConGen is described in the following.

First, the user can start by creating a super-population to represent the cortical microcircuit entity and the Thalamic region. The user can then go one level down in the visualization of the cortical microcircuit super-population to create the eight different populations of the cortical microcircuit using the *Add NeuronPop* option in the menu. After the eight populations have been created, the user can create connections between the populations by clicking and dragging the cursor from the source population to the target population. As the connectivity in the cortical microcircuit model is defined by a set of connection probabilities, the user defines a random connection with Gaussian distributions for the weights and the delays. In order to create an auto-connection, the user right-clicks on the desired population and selects *Add Auto Connection* from the menu. The model at this stage of creation is depicted in **Figure 6**. It is important to highlight that the connectivity in the original manuscript by Potjans and Diesmann (2014) is calculated under a specific set of considerations that are not reflected in the random distribution used in this use case. More specifically, the original model makes use of a fixed number of connections derived from the connection probability: $K_{n,m} = \ln(1 - P_{n,m})/\ln((N_n N_m - 1)/N_n N_m)$, where $K_{n,m}$ is the total number of connections



Connection Inspector

Creating Relationship Parameters:
l6i - l6i

Name: R:l6i-l6i
Connectivity Model: Random
Random probability: 0.144
Weight Type: Gaussian
Weight Gaussian Mean: -351.2
Weight Gaussian Sigma: 35.2
Delay Type: Gaussian
Delay Gaussian Mean: 0.8
Delay Gaussian Sigma: 0.4
Threshold: -20

Cancel Save
Delete
Auto-close □

Connection Inspector

Creating Relationship Parameters:
l6i - l6i

Name: R:l6i-l6i
Connectivity Model: Random
Random probability: 0.144
Weight Type: Gaussian
Weight Gaussian Mean: -351.2
Weight Gaussian Sigma: 35.2
Delay Type: Gaussian
Delay Gaussian Mean: 0.8
Delay Gaussian Sigma: 0.4
Threshold: -20

Cancel Save
Delete
Auto-close □

FIGURE 6 | Views at different levels of cortical microcircuits. Panel **(A)** shows microcircuit super-population and the Thalamic super-population. Panel **(B)** presents the microcircuit super-population where all 8 populations can be seen with their respective connections.

between population n and population m, $P_{n,m}$ is the connection probability between the two populations, and N_x denotes the number of neurons in population x. In our implementation, the connectivity of the model is generated using a pairwise Bernoulli distribution with probability $P_{n,m}$. Therefore, variations in the actual number of connections between the model created with ConGen and the original model are to be expected.

Next, the user can create a set of input devices, in this case Poisson generators, in order to represent input arriving from other regions in the brain. This is done using the *Add Input* option and defining the frequency of the random stimuli produced by the Poisson generator. The input devices can be then connected using the click and drag operation from the source input to the target population.

Finally, in order to create the Thalamic connections, the user goes one level up in the visualization and then selects to expand the children of both the Thalamic and the microcircuit super-populations. This allows a Thalamic super-population to be created that can be then connected with the desired probability to the subpopulations representing the layer 4 and layer 6 of the microcolumn. See **Figure 7** for the final version of the model.

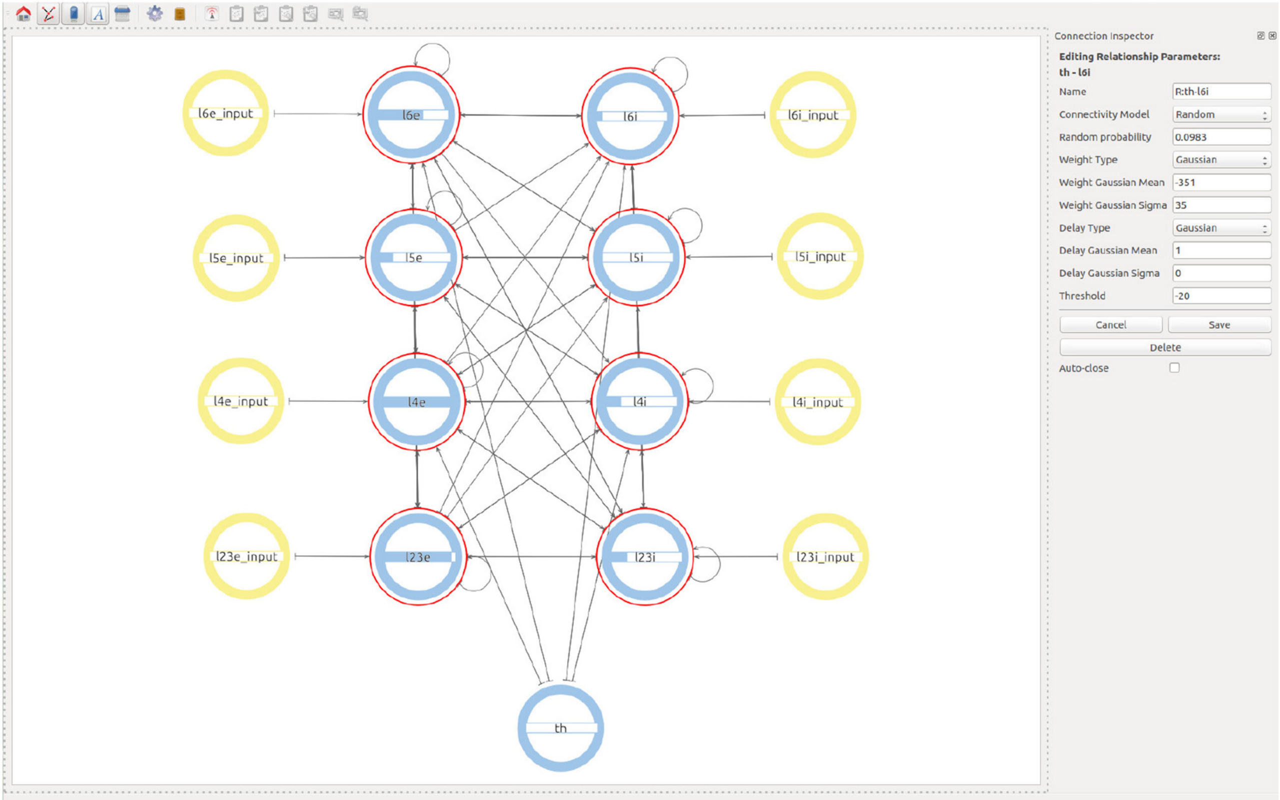
The user can then export the resulting model to JSON or save as NeuroML, producing the file which can be then used by the generation back end to call NEST and execute the simulation. The time required by the ConGen backend to read, generate the model and create the connections using CSA is negligible compared to the actual connectivity generation step using CGI and the following execution of the model in NEST. This goes in agreement with the technical requirement 5 in section 2.2. An expert user is able to define the model in this use case in about 10 min. The resulting NeuroML file is easy to explore and understand by the users.

3.2. Use Case 2: Co-simulation of The Virtual Brain and NEST

The need to simulate the brain at different scales is an emerging requirement of modern computational neuroscience. Researchers may want to simulate the whole brain at a coarse resolution while simultaneously simulating specific areas that are relevant to answer a particular scientific question at a higher resolution. This interaction between simulators is complex (Klijn et al., 2019) and has been addressed in the past by several tools such as MUSIC (Djurfeldt et al., 2010). Having a common language to describe simulations which connects different scales and simulation back ends is essential for providing a usability layer to facilitate this ambitious next step in neuroscience. As ConGen's visual language is agnostic with respect to the target simulation platform, it can be used to define complex multiscale models for co-simulation.

In this second use case of ConGen we generate simulation scripts which are compatible with the co-simulation framework of the EBRAINS infrastructure developed by the Human Brain Project.[1] In particular, we target a whole human brain co-simulation model where different parts are simulated at two

[1] https://ebrains.eu/



Connection Inspector

Editing Relationship Parameters:
th - l6i

Name	R:th-l6i
Connectivity Model	Random
Random probability	0.0983
Weight Type	Gaussian
Weight Gaussian Mean	-351
Weight Gaussian Sigma	35
Delay Type	Gaussian
Delay Gaussian Mean	1
Delay Gaussian Sigma	0
Threshold	-20

Auto-close □

FIGURE 7 | Final view of the complete model considering all populations, connections, and input devices.

scales using two simulators, The Virtual Brain (TVB; Sanz Leon et al., 2013) and NEST (Jordan et al., 2019). The coupling between the simulators is in part performed using the Elephant framework (Denker et al., 2018). For information going from NEST into TVB, the spike activity is translated into firing rates using Elephant. For information flowing from TVB into NEST, firing rates are tunneled to the NEST I/O back ends and defined as firing rates in heterogeneous Poisson generators. It is important to highlight that the coordination and deployment of the simulators is provided by external multiscale simulation infrastructure (Klijn et al., 2019) and not by ConGen itself.

ConGen is used to define the model and simulator at each scale, the connection points between simulators, and the translation modules to be used in order to transform data produced from one simulator and input to the other. In order to make it possible for the ConGen back end to identify which parts of the model belong to each scale, a prefix label is to be used for each component in the multiscale model. In this use case, we use "l" for all model components which should be simulated at the point neuron scale with NEST (in agreement with the model definition in use case 1) and the label "Brain_region" for all model components to be simulated at the whole scale level by TVB.

For the coarse scale, we divide the brain into 68 regions according to the Desikan Killiani cortical atlas (Desikan et al., 2006). Of the 68 regions, 67 are represented using a neural mass model, which in this case is the Kuramoto model (Kuramoto, 1975, 2003), and are to be simulated in TVB. The remaining region is represented as a cortical microcircuit as described in use case 1 and to be simulated in NEST. In this specific use case NEST will simulate a region in the atlas related to the auditory cortex on the right hemisphere, the right Transverse temporal cortex region. The model can be used to study the propagation of audio information from the auditory cortex to the rest of the brain and its interactions using simulations with sound stimuli. Please note that here for simplicity we assume that the phase represented by the state variable in the Kuramoto model can be linked to an indirect measure of the mean neural activity in the region and translated into spikes using the *Rate to Spike translator* available in the co-simulation framework.

The user starts by generating two super-populations, one will represent the brain regions modeled in TVB and the other one the brain region modeled in NEST. Additionally, the user will create one spike to rate translator and a rate to spike translator (see **Figure 8**). These input devices are used to exchange information between scales and will be connected to specific populations within each super-population. Although obviously not existing in the real brain, translator components are nonetheless required to produce a functional multiscale co-simulation model.

Now the user can go one level down in the two super-populations. In the NEST region, the hierarchy, connectivity and inputs of the cortical microcircuit are defined as in use case 1. In

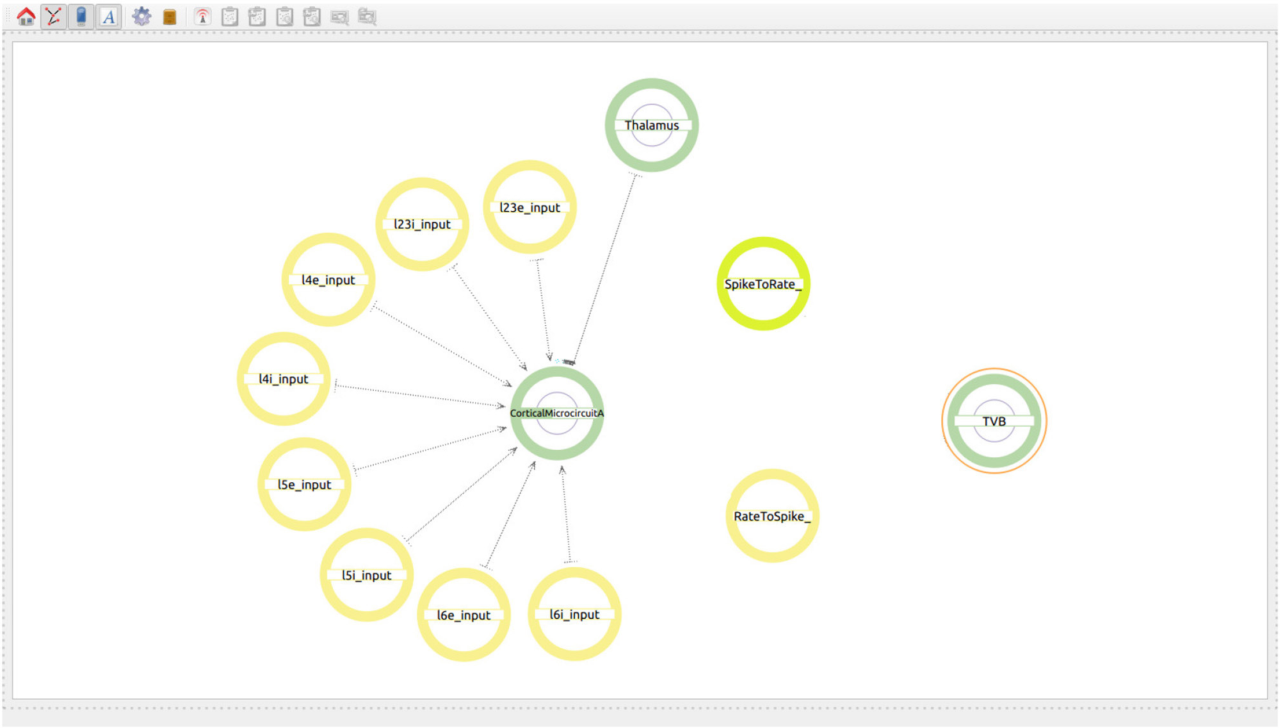


FIGURE 8 | Modeling of the co-simulation use case starting with the super-population for the brain region represented by NEST, the super-population for the brain regions represented by TVB and the translator modules which have the task of translating spikes to rates and vice versa.

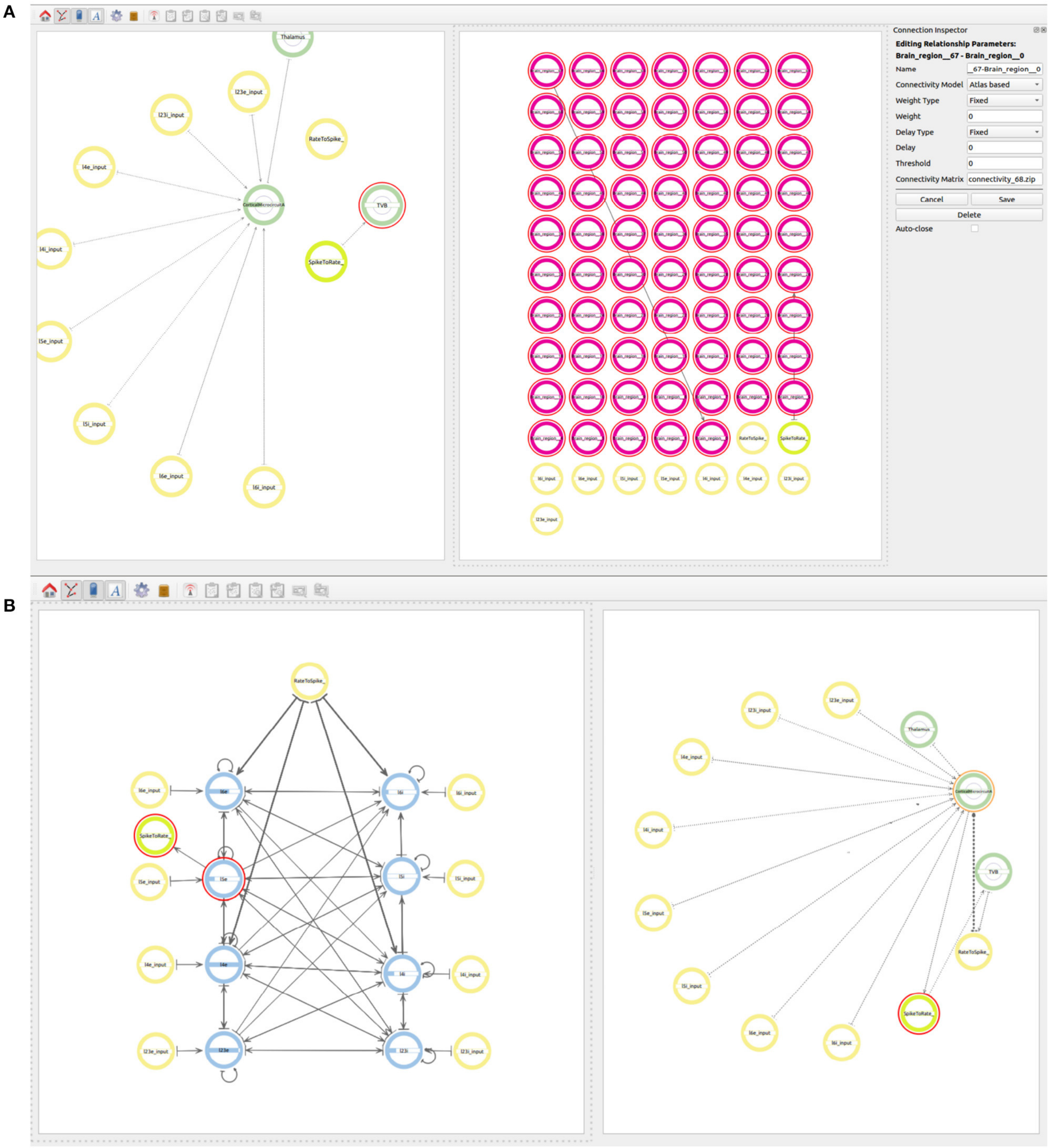
the TVB super-population, 68 population elements are created. The neuron model to be used for 67 of the 68 regions will be a neuron mass model called "nmm kuramoto" corresponding to the Kuramoto model. The last region (correspond to ID 27) is modeled as a proxy for the NEST cortical microcolumn using the model called "proxy." These elements are automatically numbered from 0 to 67 when created by ConGen and are linked to the correct region ID in the Desikan Kiliani brain atlas for simulation by the back end tool.

The next step is to define the connectivity between all the 68 regions. The type of connection to be used in the TVB models is *Atlas based* and the value in the *Connectivity Matrix* field should correspond to the connectivity matrix file to be used at simulation time (see **Figure 9A**). The value indicates a zip file which contains at least two files, one containing the weights matrix and another one containing the tract lengths matrix. The weights matrix is an *NxN* matrix which defines the strength of the connections between brain regions and where *N* is the number of regions in the specific parcellation to be used. The tract lengths matrix has the same dimensions as the weights matrix and specifies the distance between brain regions. These matrices are plain CSV files derived from empirical Diffusion Tensor Imaging (DTI) data [for more information please refer to Sanz Leon et al. (2013), section 1.1]. This ensures that the weight and the delay are loaded from the desired connectivity matrices before simulation. The user only needs to connect the first and the last region which will be connected with the desired atlas. It is important that the regions in the atlas match the range of regions selected in the model and that all regions involved are created with an index e.g., *Brain_region_{index}*.

In order to connect the two simulations on different scales, the output of all 67 regions in the TVB super-population, which are to be simulated in TVB, need to be connected to the *Rate to Spike translator* device created before. The output of the *Spike to rate translator* device must also be connected as input to the 67 regions in the TVB super-population. As mentioned before, region 27 serves as a *proxy* of the NEST super-population and, together with the translator modules, it is also used to simplify the exchange of information between both scales. As all regions in the TVB super-population are connected between each other using the atlas based connectivity, including region 27, it is only necessary to connect the output of region 27 to the *Rate to Spike translator* and the output of the *Spike to rate translator* as input to region 27. This way, the information exchange will be tunneled via the *proxy* region 27 in TVB.

Now the user can also connect the output of the *Rate to Spike translator* device as input to the excitatory and inhibitory populations in layers 4 and 6 of the cortical microcircuit model of the NEST super-population. The output of the excitatory population in layer 5 is then connected to the *Spike to rate translator* device (see **Figure 9B**).

Finally, the user can export the NeuroML file and execute the back end tool in order to generate the simulation files for TVB, NEST and the spike/rate translator modules. Using



Connection Inspector

Editing Relationship Parameters:
Brain_region__67 - Brain_region__0

Name: _67-Brain_region__0
Connectivity Model: Atlas based
Weight Type: Fixed
Weight: 0
Delay Type: Fixed
Delay: 0
Threshold: 0
Connectivity Matrix: connectivity_68.zip

Auto-close □

FIGURE 9 | Views of the multiscale model. **(A)** Establishing connectivity within the brain regions in the TVB super-population using the Atlas based connectivity. A single connection from *Brain_region*_0 to *Brain_region*_67 is used to specify the atlas based connectivity. The connection from region 27 to the rate to spike translator device is also visible at this step. **(B)** The final whole connectivity setup visualized on the right and the inside view of the cortical microcircuit model on the left.

ConGen, and using the cortical circuit model as a starting point, the user takes about five additional minutes to specify the connectivity of this multiscale model. In return, the ConGen back end inputs the specific identifiers, connectivity patterns, and proxy interfaces in TVB, NEST, and the translator module files to enable co-simulation. The resulting files can then be executed using tools from the EBRAINS co-simulation framework (see the **Supplementary Material** section for more

TABLE 2 | Simulator support by ConGen in different modalities: Connectivity setup and generation by ConGen back end, connectivity setup and basic simulator launching via ConGen back end, support for NeuroML file generated by ConGen front end using only standard features (see **Supplementary Material** for details on ConGen's extended connectivity features), and CGI connectivity generation through the ConGen back end.

Type of support	NEST	TVB	EBRAINS multiscale co-simulation	All NeuroML compatible simulators
Connectivity setup	YES	YES	YES	NO
Basic simulation launching	YES	YES	NO	NO
Standard NeuroML	YES	YES	YES	YES
CGI compatibility	YES	NO	NO	NO
Available use cases	YES	YES	YES	NO

details). The performance of the connectivity generation is almost identical between having a single scale model or as part of a multiscale model. The only difference is in the connectivity to and from the translation modules, which depends on the co-simulation infrastructure.

3.3. Supported Simulation Engines

Even though the two use cases presented in this manuscript focus on NEST and TVB, ConGen can be easily extended to work with any simulator that supports the CGI. The only requirement is that a thin interfacing file needs to be generated to deal with importing, accessing the model data with the simulator-specific CGI commands, and launching the simulation. Simulation engines which support NeuroML can directly read the file generated by the ConGen interface and use it as a base for simulation. To summarize we provide an overview of the different simulators supported directly or via the diverse interfaces in **Table 2**.

4. DISCUSSION

ConGen provides an easy way to generate networks at different scales, providing users the ability to visualize the relationships between scales in independent but correlated views and in side by side panels. As a concise but expressive visual language, ConGen provides a new way to define and navigate complex neural network models. The transcription of the defined circuits into NeuroML provides independence with respect to the ultimate choice of simulator.

The interaction offered by the ConGen's visual front end enables a rapid construction of neural network models through simple contextual menus, from defining a hierarchical structure with complex connectivity to parameterizing the neuronal and synaptic properties. The symbolic representations of the language synthesizes the most relevant features while eliminating less important details. The combination of schematic representations together with their arrangement in levels of abstraction yields simplified views of complex models.

The two use cases presented in this work illustrate the visual creation of connectivity in neural network models for their subsequent integration into simulation engines. **Figure 7** provides a good example of how a user can examine the different levels of abstraction and easily identify the relationships between the components in the network. Use case 2 illustrates new possibilities to interact with abstractions that allow definition of multiscale models. At the higher level (**Figure 8**) we see the coarse components that form the model together with the abstract modeling components required to translate information between scales. **Figure 9A** shows an alternative view of the model, with the abstract high-level multiscale components on the left and the whole brain scale region definition on the right. In contrast, **Figure 9B** ConGen provides a more detailed view of the cortical microcircuit region and makes the relationship to translation and other components in the model easy to see and manipulate by the user. One important feature of the ConGen front end is the ability to have multiple panels concurrently showing different hierarchical levels of the model. These panels are connected between each other, so any actions done on one panel are automatically reflected on the others. This is useful for the exploration and design of multiscale models because it allows the user to visualize propagation of model changes from lower scales into higher scales as seen in **Figure 9B**.

Use case 2 provides an initial proof of concept for the definition of multiscale models compatible with the EBRAINS co-simulation infrastructure. The capabilities of the ConGen back end on this area are still limited and need to be extended to support further use cases and more complex interactions between simulators.

When comparing ConGen to existing simulator front ends, such as PyNN, further advantages become apparent. In PyNN, network creation code is inherently sequential, a characteristic that is contrary to the structure of a neuronal network. The visual language introduced in ConGen allows for a holistic view of a network model, which makes it easier to interpret the network or to spot errors. Other front ends like NEST Desktop (Spreizer et al., 2021), the TVB framework, and the NEURON graphic user interface (Carnevale and Hines, 2006) also allow the user to define networks and their connectivity in a visual way but are by definition, in contrast to ConGen, simulator-specific. We hope that ConGen can serve to decouple the way we create network models from the technical aspects of simulation, such as specific execution and deployment definitions, something that projects such as PyNN still require.

5. CONCLUSIONS AND FUTURE WORK

ConGen addresses the complexity inherent to model generation in computational neuroscience from two perspectives. Firstly, it supports the visualization of complex models at different scales, allowing a reduction in the number of elements at higher scales and thus simplifying the visual complexity present in images with a high number of elements. Secondly, it reduces cognitive complexity by structuring the model in hierarchical levels of abstraction that summarize relevant features while eliminating less important details.

Multiscale modeling is a particularly demanding branch of computational neuroscience which exhibits a high degree of abstraction complexity. With the second use case provided in this manuscript we provide a proof of concept for a new visual approach to manage the complexity of constructing such models. The execution of multiscale models, which is not contemplated by the ConGen back end, has high computational demands which can only be efficiently fulfilled by using a dedicated framework such as the EBRAINS co-simulation infrastructure. Further extensions of ConGen may be required to fully facilitate current and future research in this field. With this in mind, ConGen was designed and implemented in a modular fashion; its integration with network definition standards allows developers and users to extend its functionality to include other simulation and emulation platforms (e.g., neuromorphic hardware like SpiNNaker; Furber et al., 2014) in the future.

The current version of ConGen does not fully constrain the modeler with regards to the types of connections it allows between network components scales. NeuroScheme has basic functionality to support this kind of aided connectivity generation but it would need additional metadata for each component to fully enable this functionality. For example, in the case of multiscale use cases, ConGen does not have any metadata which allows it to suggest or prevent possible connections to or from translators or devices. Adding such metadata would be a good extension for the future, and would further increase the usability for beginner users.

The next step for the ConGen back end is to update to NeuroML version 2 with LEMS. The network description of NeuroML version 2 has a different definition of the populations, which makes it necessary to describe cells within populations instead of providing generic cell types for all elements in a population. This is useful for small networks and morphologically detailed networks, but not suitable for the large scale networks targeted by ConGen. As discussed in the section 2, to allow generation of networks using the CSA we had to expand on the NeuroML interface. Although there is currently no direct way to port our work to NeuroML2, our next steps include working with the NeuroML2 development team in order to extend the language with at least a subset of the connectivity patterns available in CSA such as all-to-all or one-to-one. This will probably be implemented with a new population description using LEMS. Another alternative is to move toward NeuroMLite (https://github.com/NeuroML/NeuroMLlite) which is still in development but seems to move toward standardized description of biological and artificial networks with features which are compatible with the ConGen goals and architecture.

Future work also involves extending the back end to incorporate more cases for different simulators and allow more complex models, especially for co-simulation. Plasticity is also an important feature of connectivity that may require new visual language concepts. The direct next extension of ConGen is to allow plastic synapses to be defined and to implement an interactive loop (see **Figure 5**), the dashed arrow indicates the transport of simulation results back to ConGen) where connectivity can be refreshed based on data produced during simulation. This new step will provide a new graphic interface to study dynamic changes in the connectivity of large scale networks.

PyNN has evolved as a strong domain specific language for network representation in the last years. Future work will also involve extending NeuroScheme and the back end in order to support PyNN as a description language. This can be achieved through the porting to NeuroML version 2. The automatic benefit here is that PyNN already incorporates CSA in its description and an extension will increase the range of potential target simulators which can benefit from the visual language proposed by ConGen. Adding models generated by the ConGen front end to Open Source Brain (Gleeson et al., 2019) would also be a step forward to increase integration with current efforts in the direction of standardization. Additionally, the ConGen back end could be later integrated into Open Source Brain, thanks to their usage of common standard like NeuroML and PyNN.

In summary, with this work we propose a novel simulator-agnostic method for the definition and generation of connectivity in multiscale neural network models. ConGen also represents a new way to generate models which can be ported to different simulators using NeuroML or the ConGen back end in order to perform benchmarking and compare functional and execution metrics between simulation engines at different scales. Using the ConGen framework does not require any programming experience; any scientist, regardless of background, can employ a common visual language to express, share, study, and implement connectivity for *in-silico* experimentation, in order to solve complex questions regarding the relationships between structure and function in the brain.

DATA AVAILABILITY STATEMENT

The original contributions presented in the study are included in the article/**Supplementary Material**, further inquiries can be directed to the corresponding author/s.

AUTHOR CONTRIBUTIONS

OR, SM, PT, LP, AP, AM, SD-P, and WK worked on the design of the system. IC, OR, SM, PT, and LP worked on the design of the visual front end. PH, AP, AM, SD-P, and WK worked on the rest of the workflow and designed the use cases. PH, IC, WK, and SD-P worked on the implementation. All authors conceived of the project, reviewed, contributed, and approved the final version of the manuscript.

FUNDING

The research leading to these results has received funding from the Spanish Ministry of Economy and Competitiveness under grants C080020-09 (Cajal Blue Brain Project, Spanish partner of the Blue Brain Project initiative from EPFL), TIN2017-83132, PID2020-113013RB-C21, and PID2020-113013RB-C22, as well as from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements No. 785907 (Human Brain Project SGA2) and 945539 (Human Brain Project SGA3). This research has also been partially funded by the Helmholtz Association through the Helmholtz Portfolio Theme Supercomputing and Modeling for the Human Brain.

ACKNOWLEDGMENTS

We would like to kindly acknowledge the support and discussions with Viktor Jirsa and Lionel Kusch about the generation of co-simulation scripts with TVB and NEST.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2021.766697/full#supplementary-material

REFERENCES

Abi Akar, N., Biddiscombe, J., Cumming, B., Huber, F., Kabic, M., Karakasis, V., et al. (2021). *arbor-sim/arbor: Arbor library v0.5*.

Akar, N. A., Cumming, B., Karakasis, V., Kusters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor–a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia), 274–282. doi: 10.1109/EMPDP.2019.8671560

Al-Awami, A. K., Beyer, J., Strobelt, H., Kasthuri, N., Lichtman, J. W., Pfister, H., et al. (2014). NeuroLines: a subway map metaphor for visualizing nanoscale neuronal connectivity. *IEEE Trans. Visual. Comput. Graph*. 20, 2369–2378. doi: 10.1109/TVCG.2014.2346312

Böttger, J., Schäfer, A., Lohmann, G., Villringer, A., and Margulies, D. S. (2014). Three-dimensional mean-shift edge bundling for the visualization of functional connectivity in the brain. *IEEE Trans. Visual. Comput. Graph*. 20, 471–480. doi: 10.1109/TVCG.2013.114

Cakan, C., Jajcay, N., and Obermayer, K. (2021). neurolib: a simulation framework for whole-brain neural mass modeling. *bioRxiv*. doi: 10.1007/s12559-021-09931-9

Cannon, R. C., Gleeson, P., Crook, S., Ganapathy, G., Marin, B., Piasini, E., et al. (2014). LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Front. Neuroinform*. 8:79. doi: 10.3389/fninf.2014.00079

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge University Press. doi: 10.1017/CBO9780511541612

Chen, X., Wang, Y., Kopetzky, S. J., Butz-Ostendorf, M., and Kaiser, M. (2021). Connectivity within regions characterizes epilepsy duration and treatment outcome. *Hum. Brain Mapp*. 42:3777–91. doi: 10.1002/hbm.25464

Collins, F., and Prabhakar, A. (2013). *BRAIN Initiative Challenges Researchers to Unlock Mysteries of Human Mind*. Available online at: http://www.whitehouse.gov/blog/2013/04/02/brain-initiative-challenges-researchers-unlock-mysteries-human-mind

Davison, A., Bruderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform*. 2:11. doi: 10.3389/neuro.11.011.2008

Denker, M., Alper, Y., and Sonja, G.(2018). *Collaborative HPC-enabled workflows on the HBP Collaboratory using the Elephant framework*. (Germany: INM-ICS Retreat)Available online at: https://juser.fz-juelich.de/record/850028/export/hx?ln=en.

Desikan, R. S., Ségonne, F., Fischl, B., Quinn, B. T., Dickerson, B. C., Blacker, D., et al. (2006). An automated labeling system for subdividing the human cerebral cortex on MRI scans into gyral based regions of interest. *Neuroimage* 31, 968–980. doi: 10.1016/j.neuroimage.2006.01.021

Djurfeldt, M. (2012). The connection-set algebra–a novel formalism for the representation of connectivity structure in neuronal network models. *Neuroinformatics* 10, 287–304. doi: 10.1007/s12021-012-9146-1

Djurfeldt, M., Davison, A. P., and Eppler, J. M. (2014). Efficient generation of connectivity in neuronal networks from simulator-independent descriptions. *Front. Neuroinform*. 8:43. doi: 10.3389/fninf.2014.00043

Djurfeldt, M., Hjorth, J., Eppler, J. M., Dudani, N., Helias, M., Potjans, T. C., et al. (2010). Run-time interoperability between neuronal network simulators based on the MUSIC framework. *Neuroinformatics* 8, 43–60. doi: 10.1007/s12021-010-9064-z

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2009). Pynest: a convenient interface to the nest simulator. *Front. Neuroinform*. 2:12. doi: 10.3389/neuro.11.012.2008

Espinoza-Valdez, A., Negrón, A. P. P., Salido-Ruiz, R. A., and Carranza, D. B. (2021). "EEG data modeling for brain connectivity estimation in 3D graphs," in *New Perspectives in Software Engineering*, eds J. Mejia, M. Munoz, Á. Rocha, and Y. Quinonez (Cham: Springer International Publishing), 280–290. doi: 10.1007/978-3-030-63329-5_19

Evanko, D., and Pastrana, E. (2013). Why mapping the brain matters. *Nat. Methods* 10:447. doi: 10.1038/nmeth.2513

Fujiwara, T., Chou, J., McCullough, A. M., Ranganath, C., and Ma, K. (2017). "A visual analytics system for brain functional connectivity comparison across individuals, groups, and time points," in *2017 IEEE Pacific Visualization Symposium (PacificVis)* (Seoul), 250–259. doi: 10.1109/PACIFICVIS.2017.8031601

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The spinnaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Gadde, S., Aucoin, N., Grethe, J. S., Keator, D. B., Marcus, D. S., Pieper, S., et al. (2012). XCEDE: an extensible schema for biomedical data. *Neuroinformatics* 10, 19–32. doi: 10.1007/s12021-011-9119-9

Gleeson, P., Cantarelli, M., Marin, B., Quintana, A., Earnshaw, M., Sadeh, S., et al. (2019). Open source brain: a collaborative resource for visualizing, analyzing, simulating, and developing standardized models of neurons and circuits. *Neuron* 103, 395–411. doi: 10.1016/j.neuron.2019.05.019

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., et al. (2010). NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comput. Biol*. 6:e1000815. doi: 10.1371/journal.pcbi.1000815

Gorgolewski, K. J., Auer, T., Calhoun, V. D., Craddock, R. C., Das, S., Duff, E. P., et al. (2016). The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments. *Sci. Data* 3, 1–9. doi: 10.1038/sdata.2016.44

Jordan, J., Mork, H., Vennemo, S. B., Terhorst, D.,Peyser, A., Ippen, T., et al.(2019). NEST 2.18.0. Zenodo. doi: 10.5281/zenodo.2605422

Klijn, W., Diaz-Pier, S., Morrison, A., and Peyser, A. (2019). "Staged deployment of interactive multi-application HPC workflows," in *2019 International Conference on High Performance Computing* & *Simulation (HPCS)* (Dublin: IEEE), 305–311. doi: 10.1109/HPCS48598.2019.9188104

Kuramoto, Y. (1975). "Self-entrainment of a population of coupled non-linear oscillators," in *International Symposium on Mathematical Problems in Theoretical Physics* (Berlin; Heidelberg: Springer), 420–422.

Kuramoto, Y. (2003). *Chemical Oscillations, Waves, and Turbulence*. Mineola, NY: Courier Corporation.

Markram, H., Muller, E., Ramaswamy, S., Reimann, M., Abdellah, M., Sanchez, C., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

Meunier, D., Pascarella, A., Altukhov, D., Jas, M., Combrisson, E., Lajnef, T., et al. (2020). NeuroPycon: an open-source python toolbox for fast multi-modal and reproducible brain connectivity pipelines. *Neuroimage* 219:117020. doi: 10.1016/j.neuroimage.2020.117020

Mijalkov, M., Kakaei, E., Pereira, J. B., Westman, E., and Volpe, G. (2017). BRAPH: a graph theory software for the analysis of brain connectivity. *bioRxiv*. 12. doi: 10.1371/journal.pone.0178798

Morgan, J. L., and Lichtman, J. W. (2013). Why not connectomics? *Nat. Methods* 10, 494–500. doi: 10.1038/nmeth.2480

Nordlie, E., Gewaltig, M.-O., and Plesser, H. E. (2009). Towards reproducible descriptions of neural network models. *PLoS Comput. Biol.* 5:e1000456. doi: 10.1371/journal.pcbi.1000456

Nordlie, E., and Plesser, H. E. (2010). Visualizing neuronal network connectivity with connectivity pattern tables. *Front. Neuroinform.* 3:39. doi: 10.3389/neuro.11.039.2009

Pastor, L., Mata, S., Toharia, P., Beriso, S. B., Brito, J. P., and Garcia-Cantero, J. J. (2015). "NeuroScheme: efficient multiscale representations for the visual exploration of morphological data in the human brain neocortex," in *XXV*

Spanish Computer Graphics Conference, CEIG 2015, eds M. Sbert and J. Lopez-Moreno (Benicássim), 117–125.

Pauli, R., Weidel, P., Kunkel, S., and Morrison, A. (2018). Reproducing polychronization: a guide to maximizing the reproducibility of spiking network models. *Front. Neuroinform*. 12:46. doi: 10.3389/fninf.2018.00046

Peyser, A., Diaz Pier, S., Klijn, W., Morrison, A., and Triesch, J. (2019). Linking experimental and computational connectomics. *Netw. Neurosci*. 3, 902–904. doi: 10.1162/netn_e_00108

Potjans, T. C., and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Raikov, I., Cannon, R., Clewley, R., Cornelis, H., Davison, A., De Schutter, E., et al. (2011). NineML: the network interchange for neuroscience modeling language. *BMC Neurosci*. 12:P330. doi: 10.1186/1471-2202-12-S1-P330

Rubinov, M., and Sporns, O. (2010). Complex network measures of brain connectivity: uses and interpretations. *Neuroimage* 52, 1059–1069. doi: 10.1016/j.neuroimage.2009.10.003

Sanz Leon, P., Knock, S. A., Woodman, M. M., Domide, L., Mersmann, J., McIntosh, A. R., et al. (2013). The virtual brain: a simulator of primate brain network dynamics. *Front. Neuroinform*. 7:10. doi: 10.3389/fninf.2013.00010

Sporns, O., Tononi, G., and Kotter, R. (2005). The human connectome: a structural description of the human brain. *PLoS Comput. Biol*. 1:e42. doi: 10.1371/journal.pcbi.0010042

Spreizer, S., Senk, J., Rotter, S., Diesmann, M., and Weyers, B. (2021). NEST Desktop, an Educational Application for Neuroscience. *Soc. Neurosci*. 8, 25–29. doi: 10.1523/ENEURO.0274-21.2021

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8:e47314. doi: 10.7554/eLife.47314

Tikidji-Hamburyan, R. A., Narayana, V., Bozkus, Z., and El-Ghazawi, T. A. (2017). Software for brain network simulations: a comparative study. *Front. Neuroinform*. 11:46. doi: 10.3389/fninf.2017.00046

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

ORIGINAL RESEARCH
published: 01 March 2022
doi: 10.3389/fninf.2021.785068

Routing Brain Traffic Through the Von Neumann Bottleneck: Parallel Sorting and Refactoring

Jari Pronold[1,2]*, *Jakob Jordan*[3], *Brian J. N. Wylie*[4], *Itaru Kitayama*[5], *Markus Diesmann*[1,6,7] *and Susanne Kunkel*[8]

[1] Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6) and JARA-Institute Brain Structure-Function Relationships (INM-10), Jülich Research Centre, Jülich, Germany, [2] RWTH Aachen University, Aachen, Germany, [3] Department of Physiology, University of Bern, Bern, Switzerland, [4] Jülich Supercomputing Centre, Jülich Research Centre, Jülich, Germany, [5] RIKEN Center for Computational Science, Kobe, Japan, [6] Department of Physics, Faculty 1, RWTH Aachen University, Aachen, Germany, [7] Department of Psychiatry, Psychotherapy and Psychosomatics, Medical Faculty, RWTH Aachen University, Aachen, Germany, [8] Faculty of Science and Technology, Norwegian University of Life Sciences, Ås, Norway

Generic simulation code for spiking neuronal networks spends the major part of the time in the phase where spikes have arrived at a compute node and need to be delivered to their target neurons. These spikes were emitted over the last interval between communication steps by source neurons distributed across many compute nodes and are inherently irregular and unsorted with respect to their targets. For finding those targets, the spikes need to be dispatched to a three-dimensional data structure with decisions on target thread and synapse type to be made on the way. With growing network size, a compute node receives spikes from an increasing number of different source neurons until in the limit each synapse on the compute node has a unique source. Here, we show analytically how this sparsity emerges over the practically relevant range of network sizes from a hundred thousand to a billion neurons. By profiling a production code we investigate opportunities for algorithmic changes to avoid indirections and branching. Every thread hosts an equal share of the neurons on a compute node. In the original algorithm, all threads search through all spikes to pick out the relevant ones. With increasing network size, the fraction of hits remains invariant but the absolute number of rejections grows. Our new alternative algorithm equally divides the spikes among the threads and immediately sorts them in parallel according to target thread and synapse type. After this, every thread completes delivery solely of the section of spikes for its own neurons. Independent of the number of threads, all spikes are looked at only two times. The new algorithm halves the number of instructions in spike delivery which leads to a reduction of simulation time of up to 40 %. Thus, spike delivery is a fully parallelizable process with a single synchronization point and thereby well suited for many-core systems. Our analysis indicates that further progress requires a reduction of the latency that the instructions experience in accessing memory. The study provides the foundation for the exploration of methods of latency hiding like software pipelining and software-induced prefetching.

Keywords: spiking neural networks, large-scale simulation, distributed computing, parallel computing, sparsity, irregular access pattern, memory-access bottleneck

OPEN ACCESS

Edited by:
Ludovico Minati,
Tokyo Institute of Technology, Japan

Reviewed by:
Stavros I. Dimitriadis,
Greek Association of Alzheimer's Disease and Related Disorders, Greece
Jiang Wang,
Tianjin University, China
Antonio Marcos Batista,
Universidade Estadual de Ponta Grossa, Brazil

****Correspondence:***
Jari Pronold
j.pronold@fz-juelich.de

Received: *28 September 2021*
Accepted: *24 December 2021*
Published: *01 March 2022*

Citation:
Pronold J, Jordan J, Wylie BJN, Kitayama I, Diesmann M and Kunkel S (2022) Routing Brain Traffic Through the Von Neumann Bottleneck: Parallel Sorting and Refactoring. Front. Neuroinform. 15:785068. doi: 10.3389/fninf.2021.785068

1. INTRODUCTION

Over the last two decades, simulation algorithms for spiking neuronal networks have continuously been improved. The largest supercomputers available can be employed to simulate networks with billions of neurons at their natural density of connections. The respective codes scale well over orders of magnitude of network size and number of compute nodes (Jordan et al., 2018). Still, simulations at the brain scale are an order of magnitude slower than real time, hindering the investigation of processes such as plasticity and learning unfolding over hours and days of biological time. In addition, there is a trend of aggregating more compute power in many-core compute nodes. This further reduces the strain on inter-node communication as one limiting component but increases the urgency to better understand the fundamental operations required for routing spikes within a compute node.

The spiking activity in mammalian neuronal networks is irregular, asynchronous, sparse, and delayed. Irregular refers to the structure of the spike train of an individual neuron. The intervals between spike times are of different lengths and unordered as if drawn from a random process. Consequently, the number of spikes in a certain time interval also appears random. Asynchronous means that the spikes of any two neurons occur at different times and exhibit low correlation. The activity of neurons is sparse in time as compared to the time constants of neuronal dynamics; only few spikes are emitted in any second of biological time. Last, there is a biophysical delay in the interaction between neurons imposed by their anatomy. The delay may be a fraction of a millisecond for neurons within a distance of a few micrometers but span several milliseconds for connections between brain regions (refer to Schmidt et al., 2018a, for an example compilation of parameters).

The existence of a minimal delay in a network model together with the sparsity of spikes has suggested a three-phase cycle for an algorithm directly integrating the differential equations of the interacting model neurons (Morrison et al., 2005). First, communication between compute nodes occurs synchronously in intervals of minimal delay. This communication transmits all the spikes that have occurred on a compute node since the last communication step to the compute nodes harboring target neurons of these spikes. Second, the received spikes are delivered to their target neurons and placed in spike ring buffers representing any remaining individual delay. Finally, the dynamical state of each neuron is propagated for the time span of the minimal delay while the ring buffer is rotating accordingly. Once all neurons are updated, the next communication is due and the cycle begins anew.

Progress in each update phase is shaped by the spiking interaction between neurons and independent of the level of detail of the individual model neurons constituting the network. The choice of the neuron model, however, influences the distribution of computational load across the phases of the simulation. Some studies require neuron models with thousands of electrical compartments (Markram et al., 2015), and efficient simulation codes are available for this purpose (Carnevale and Hines, 2006; Akar et al., 2019; Kumbhar et al., 2019). Here, we focus on simulation code for networks of model neurons described by a handful of differential equations as widely used in the computational neuroscience community. These investigations range from studies with several thousands of neurons on the fundamental interplay between excitation and inhibition (Brunel, 2000) to models attempting to capture the natural density of wiring (Potjans and Diesmann, 2014; Billeh et al., 2020) and the interaction between multiple cortical areas (Joglekar et al., 2018; Schmidt et al., 2018b). Previous measurements on a production code (Jordan et al., 2018) already show that for networks of such simple model neurons the dominating bottleneck for further speed-up is neither the communication between computes nodes nor the update of the dynamical state of the neurons, but the spike-delivery phase. The empirical finding is elegantly confirmed by an analytical performance model encompassing different types of network and neuron models (Cremonesi and Schürmann, 2020; Cremonesi et al., 2020). These authors further identify the latency of memory access as the ultimate constraint of the spike-delivery phase.

Profiling tools like Intel VTune provide measures on where an application spends its time and how processor and memory are used. Two basic measures are the total number of instructions carried out and the number of clock ticks the processor required per instruction (CPI). The former characterizes the amount of computations that need to be done to arrive at the solution. The latter describes how difficult it is on average to carry out an individual instruction due to the complexity of the operation and the waiting for accessing the corresponding part of memory. The product of the two measures is the total number of clockticks and should correlate to the wall clock time required to complete the simulation phase under investigation. Methods of software pipelining and software-induced prefetching attempt to improve the CPI by better vectorization of the code or by indicating to the processor which memory block will soon be required. These optimizations may lead to an increase in the actual number of instructions but as long as the product with the reduced CPI decreases, performance is improving. Nevertheless, a low CPI does not mean that the code is close to optimal performance. If the code is overly complicated, for example by recalculating known results or missing out on regularities in the data it may underutilize data that has been retrieved from memory rendering advanced methods of optimization fruitless. Therefore, in the present study, as a first step, we do not consider pipelining and prefetching but exclusively assess the number of instructions required by the algorithm. It turns out that a better organized algorithm indeed avoids unnecessary tests and indirections. This decrease in the number of instructions also decreases CPI as a side effect until with increasing sparseness of the network CPI climbs up again. The control flow in the code becomes more predictable for the processor until the fragmentation of memory limits the success. The results of our study give us some confidence that further work can now directly address improving the CPI.

In Section 2, we expose spike delivery as the present bottleneck for the simulation of mammalian spiking neuronal networks, characterize analytically the transition to sparsity with growing network size, and present the original algorithm as well as state-of-the-art performance data. Next, we introduce the software environment of our study and the neuronal network model used to obtain quantitative data (Section 3). On the basis of these preparatory sections, Section 4 presents a new algorithm streamlining the routing of spikes to their targets. Subsequently, Section 5 evaluates the success of the redesign and identifies the origin of the improvement by profiling. Finally, Section 6 embeds the findings into the ongoing efforts to develop generic technology for the simulation of spiking neuronal networks.

The conceptual and algorithmic work described here is a module in our long-term collaborative project to provide the technology for neural systems simulations (Gewaltig and Diesmann, 2007). Preliminary results have been presented in abstract form (Kunkel, 2019).

2. SPIKE DELIVERY AS MEMORY-ACCESS BOTTLENECK

The temporally sparse event-based communication between neurons presents a challenging memory-access bottleneck in simulations of spiking neuronal networks for modern architectures optimized for dense data. In the neuronal simulator NEST (Section 3.1), which we use as reference implementation in this study, delivery of spikes to their synaptic and neuronal targets involves frequent access to essentially random memory locations, rendering automatic prediction difficult and leading to long data-access times due to ineffective use of caches. The following subsection provides an analysis of the sparsity of the network representation for increasing numbers of Message Passing Interface (MPI) processes and threads. Based on this, there follows a description of the connection data structures and spike-delivery algorithm in the original implementation. The final subsection provides example benchmarking data for this state-of-the-art simulation code.

2.1. Sparsity of Network Representation

We consider a network of N neurons distributed in a round-robin fashion across M MPI processes and T threads per process. Each neuron receives K incoming synapses, which are represented on the same thread as their target neuron. In a weak scaling scenario, the computational load per process is kept constant. This implies that the number of thread-local synapses

$$S = NK/(MT) \tag{1}$$

does not change. The total network size, in contrast, increases with MT. In the limit of large network sizes, each synapse on a given thread originates from a different source neuron. This scenario was already considered (Kunkel et al., 2014, section 2.4) at the time to analyze the increase in memory overhead observed with increasing sparsity. For completeness, we briefly restate this result in the parameters used in the present work.

The probability that a synapse has a particular neuron j as source neuron is $1/N$ and, conversely, the probability that the synapse has a different source neuron is $1 - 1/N$. The probability that none of the S thread-local synapses has neuron j as a source is $p_\emptyset = (1 - 1/N)^S$. Conversely, $p = 1 - p_\emptyset$ denotes the probability that j is the source to at least one of the thread-local synapses. Therefore, the expected number of unique source neurons of the thread-local synapses are given by $N_\mathrm{u} = pN$ expanding to

$$N_\mathrm{u} = \left(1 - \left[\left(1 - \frac{1}{N}\right)^N\right]^{\frac{K}{MT}}\right) N \tag{2}$$

which is Equation (6) of Kunkel et al. (2014). In weak scaling, MT grows proportionally to N such that

$$N_\mathrm{u} = \left(1 - \left[\left(1 - \frac{1}{N}\right)^N\right]^{\frac{S}{N}}\right) N$$

where they further identified the term $[\cdot]$ in the limit of large N as the definition of the exponential function with argument -1 and therefore

$$\widetilde{N}_\mathrm{u} = \left(1 - \exp\left(-\frac{S}{N}\right)\right) N.$$

They confirm that the limit of N_u is indeed S and that a fraction ζ of S is reached at a network size of

$$N_\zeta = \frac{S}{2\,(1-\zeta)}. \tag{3}$$

Figure 1 illustrates the point where in weak scaling the total network size N equals the number of thread local synapses S. Here, the number of unique source neurons N_u of the thread-local synapses bends. According to the definition (1) of S, here a particular target neuron chooses its K incoming synapses from the same total number of threads $MT = K$ and already half

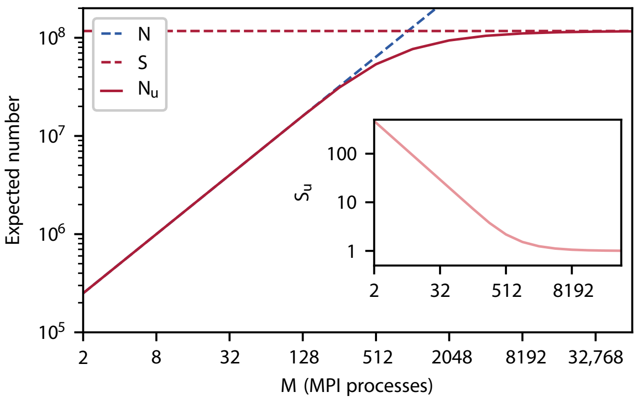


FIGURE 1 | Expected number of unique source neurons N_u (pink curve) of all thread-local synapses as a function of the number of the MPI processes M assuming $T = 12$ threads and 125,000 neurons per MPI process in a weak-scaling scenario; the total number of neurons N (dashed blue curve) and number of thread-local synapses S (dashed pink horizontal line for $K = 11,250$ synapses per neuron). Inset: Expected number of thread-local synapses per unique source neuron $S_\mathrm{u} = S/N_\mathrm{u}$ (light pink curve). All graphs in double logarithmic representation.

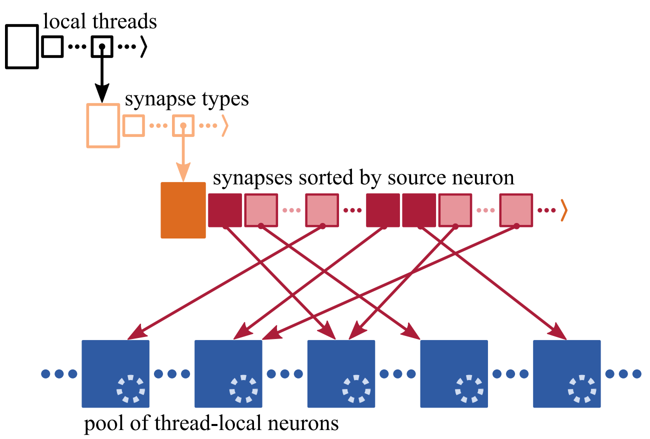

local threads
synapse types
synapses sorted by source neuron
pool of thread-local neurons

FIGURE 2 | Memory layout of synapse and neuron representations on each MPI process. Each process stores the local synapses (pink filled squares) in a three-dimensional resizable array sorted by hosting thread and synapse type. At the innermost level, synapses are arranged in source-specific target segments (dark pink: first synapse; light pink: subsequent targets); only one innermost array is shown for simplicity. Target neurons (blue filled squares) are stored in neuron-type and thread-specific memory pools; only one pool is shown for simplicity. Each neuron maintains a spike ring buffer (dotted light blue circles). Synapses have access to their target neurons through target identifiers (dark pink arrows).

($\zeta = \frac{1}{2}$) of the source neurons of the thread-local synapses are unique. The number of thread-local synapses per unique source neuron S_{u} indicates the sparsity of the network representation on a compute node (inset of **Figure 1**). The measure converges to one exhibiting a bend at the same characteristic point as N_{u}.

2.2. Memory Layout of Synapse and Neuron Representations

A three-dimensional resizable array stores the process-local synapses sorted by hosting thread and synapse type (**Figure 2**), where synapses are small in size, each typically taking up few tens of Bytes. Each synapse has access to its target neuron, which is hosted by the same MPI process and thread (Morrison et al., 2005). The target identifier provides access either through a pointer to the target neuron consuming 8 B or an index of 2 B that is used to retrieve the corresponding pointer. Here, we use the latter implementation of the target identifier reducing per-synapse memory usage at the cost of an additional indirection (refer to Section 3.3.2 in Kunkel et al., 2014).

In the innermost arrays of the data structure, synapses are sorted by source neuron, which is an optimization for small to medium scale systems (see Section 3.3 in Jordan et al., 2018) exploiting that each neuron typically connects to many target neurons (out-degree). Thereby, synapses are arranged in target segments, each consisting of at least one target synapse potentially followed by subsequent targets (Section 2.1). In a weak-scaling experiment, the increasing sparsity of the network in the small to medium scale regime (Section 2.1) influences the composition of the innermost array. As synapses are to an increasing degree distributed across MPI processes and threads, the expected number of source-specific target segments increases while the average segment size decreases (cf. N_{u} and S_{u} in **Figure 1**, respectively). Note that the degree of distribution also depends on the number of synapse types, which is however not considered in this study.

A model neuron easily takes up more than a Kilobyte of memory. Multi-chunk memory pools enable contiguous storage of neurons of the same type hosted by the same thread, where due to the many-to-one relation between target synapses and neurons, the order of memory locations of target neurons is independent of the order of synapses in the target segments.

Synaptic transmission of spikes entails delays, which influence the time when spikes take effect on the dynamics of the target neurons. As typically synapses from many different source neurons converge on the same target neuron (in-degree), it is more efficient to jointly account for their delays in the neuronal target. Therefore, each neuron maintains a spike ring buffer serving as temporary storage and scheduler for the incoming spikes (Morrison et al., 2005).

2.3. Original Spike-Delivery Algorithm

Every time all local neurons have been updated and all recent spikes have been communicated across MPI processes, the spike data needs to be delivered from the process-local MPI receive buffers to the process-local synaptic and neuronal targets. Each spike entry is destined for an entire target segment of synapses (Section 2.2), which is an optimization for the small to medium scale regime introduced in Jordan et al. (2018). Therefore, each entry conveys the location of the target segment within the three-dimensional data structure storing the process-local synapses (**Figure 2**), i.e., identifiers for the hosting thread and the type of the first synapse of the target segment, as well as the synapse's index within the innermost resizable array.

In the original algorithm, each thread reads through all spike entries in the MPI receive buffer but it only proceeds with the delivery of a spike if it actually hosts the spike's targets - all spike entries indicating other hosting threads are skipped. Each thread delivers the relevant spikes to every synapse of the corresponding target segments one by one. On receiving a spike, a synapse transfers synaptic delay and weight to the corresponding target neuron, where the stored target identifier provides access to the neuron. The transmitted synaptic properties, delay and weight, define the time and amplitude of the spike's impact on the neuron, respectively, allowing the neuron to add the weight of the incoming spike to the correct position in the neuronal spike ring buffer.

In a weak-scaling experiment, the increasing sparsity of the network in the small to medium scale regime (Section 2.1) influences algorithmic progression and memory-access patterns. Access to target neurons and their spike ring buffers is always irregular regardless of the degree of distribution of the network across MPI processes, but memory access to synapses become progressively irregular. The number of spike entries communicated via MPI increases to cater to the growing number of target segments (Section 2.2). In consequence, each thread needs to process even more spike entries, delivering relevant spikes to even more but shorter target segments, where successively visited target segments are typically in nonadjacent memory locations. In the sparse limit where each target segment consists of a single synapse, spike delivery to both neuronal and synaptic targets requires accessing essentially random locations in memory. As many synapses of different source neurons converge on the same target neuron, it is impossible to arrange target neurons in memory such that their order corresponds to the order in synaptic target segments. The pseudocode in 2.3.1 summarizes this original spike-delivery algorithm.

2.3.1. Pseudocode

ORI nrn: Original `Receive()` procedure in neuron; **RB** marks access to the spike ring buffer.

```
Data: spike_ring_buffer
Receive(delay, weight)
RB    spike_ring_buffer.AddValue(delay, weight)
```

ORI syn: Original `Send()` function in synapse, which calls the `Receive()` procedure of the target neuron (ORI nrn) passing on synaptic properties.

```
Data: subsq, target_neuron, delay, weight
Send()
    target_neuron.Receive(delay, weight)
    return subsq
```

ORI: Original algorithm delivering spikes to local targets with support for multi-threading, where TID denotes the identifier of the executing thread. **TS** marks iteration over a synaptic target segment. **SYN** marks access to an individual target synapse (ORI syn).

```
Data: recv_buffer, synapses
foreach spike in recv_buffer do
    (tid, syn_id, lcid) ← spike.GetTargetLoc()
    if tid == TID then
        subsq ← true
TS      while subsq do
SYN         subsq ← synapses[tid][syn_id][lcid].Send()
            lcid ← lcid + 1
```

The original algorithm delivers spikes to the neuronal spike ring buffers through the target synapses. Each neuron owns a *spike_ring_buffer*, where the neuron member function `Receive()` triggers the spike delivery by calling the spike ring buffer member function `AddValue()`, which then adds the weight of the spike to the correct position in the buffer (ORI nrn; **RB**). To this end, both `Receive()` and `AddValue()` require the synaptic properties *delay* and *weight*.

Each synapse stores properties such as *delay* and *weight*, an identifier enabling access to the target neuron (*target_neuron*), and an indicator (*subsq*) of whether the target segment continues or not (ORI syn). The synapse member function `Send()` calls the member function `Receive()` of the target neuron passing on the synaptic properties and returns the indicator *subsq*.

The original spike-delivery algorithm has access to the MPI spike-receive buffer (*recv_buffer*) containing all spike entries that need to be delivered and to a three-dimensional resizable array of process-local *synapses* ordered by hosting thread and synapse type (ORI; see **Figure 2**). For each spike entry, the 3D location of the first target synapse is extracted and assigned to the variables *tid*, *syn_id*, and *lcid*, which indicate hosting thread, synapse type, and location in the innermost *synapses* array, respectively. If the executing thread (TID) is the hosting thread of the target synapse, then the variable *lcid* is used in the enclosed while loop to iterate over the spike's entire synaptic target segment within the innermost array *synapses*[*tid*][*syn_id*] (**TS**). To deliver a spike to the target synapse at position *lcid*, the synapse member function `Send()` is called on *synapses*[*tid*][*syn_id*][*lcid*] returning the indicator *subsq* (**SYN**).

2.4. Simulation Time

The work of Jordan et al. (2018) shows that spike delivery is the dominating phase of simulation time from networks with a few hundred thousand neurons to the regime of billions of neurons. In the latter, the number of neurons in the network exceeds the number of synapses represented on an individual compute node; each synapse on a given compute node has a unique source neuron (Section 2.1). Therefore, a neuron finds either a single target neuron on a compute node or none at all. Assuming a random distribution of neurons across MPI processes, the network is fully distributed in terms of its connectivity. From this point on, the computational costs of spike delivery on a compute node do not change with growing network size in a weak scaling scenario; each synapse receives spikes with a certain frequency and all spikes come from different sources. What is still increasing are the costs of communication between the compute nodes. Nevertheless, for smaller networks below the limit of sparsity, Jordan et al. (2018) provide optimizations (their section 3.3) exploiting the fact that a spike finds multiple targets on a compute node. This reduces both communication time and spike-delivery time, but the effect vanishes in the limit (as shown in Figure 7C in Jordan et al., 2018; 5g-sort) where the code continues to scale well with the maximal but invariant costs of spike delivery.

The network model of Jordan et al. (2018) exhibits spike-timing dependent plasticity in its synapses between excitatory neurons. The spike-delivery phase calculates the plastic changes at synapses because synaptic weights only need to be known when a presynaptic spike is delivered to its target (Morrison et al., 2007a). Depending on the specific plasticity rule, these computations may constitute a considerable fraction of the total spike delivery time. Therefore, from the data of Jordan et al. (2018), we cannot learn which part of the spike-delivery time is due to the calculation of synaptic plasticity and which part is due to the actual routing of spikes to their targets. In order to disentangle these contributions, the present study uses the same network model but considers all synapses as static (Section 3.2).

Figure 3 shows a weak scaling of our static neuronal network model across the critical region where sparsity has not yet reached the limit. This confirms that even in the absence of synaptic plasticity spike delivery is the dominant contribution to simulation time independent of the number of MPI processes. The network on a single MPI process roughly corresponds to the smallest cortical network in which the natural number of synapses per neuron and the local connection probability of 0.1 can simultaneously be realized (Potjans and Diesmann, 2014). While our weak scaling conserves the former quantity, the latter drops. In the regime from 2 to 512 MPI processes, the absolute time required for spike delivery almost quadruples (factor of 3.9). Beyond this regime, the relative contribution of spike delivery to simulation time drops below 50% because the time required for communication is increasing. The absolute time for neuronal update remains unchanged throughout as the number of neurons per MPI process is fixed.

3. BENCHMARKING FRAMEWORK

3.1. Simulation Engine

Over the past two decades, simulation tools in computational neuroscience have increasingly embraced a conceptual separation of generic simulation engines and specific models of neuronal networks (Einevoll et al., 2019). Many different models can thus be simulated with the same simulation engine. This enables the community to separate the life cycle of a simulation engine from those of specific individual models and to maintain and further develop simulation engines as an infrastructure. Furthermore, this separation is useful for the cross-validation of different simulation engines.

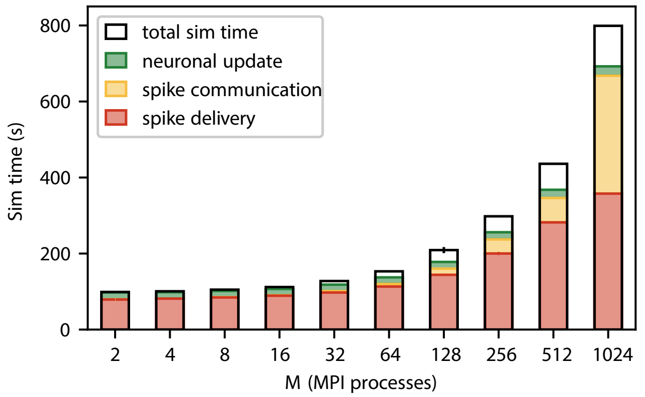


FIGURE 3 | Contributions to the simulation time (sim time) for spiking neural network simulations with NEST (Section 3.1) where the number of MPI processes increases proportionally with the total number of neurons. Weak-scaling experiment running 2 MPI processes per compute node and 12 threads per MPI process, with a workload of 125, 000 neurons per MPI process (network model see Section 3.2). The network dynamics is simulated for 1 s of biological time; spikes are communicated across MPI processes every 1.5 ms. Time is spent on spike delivery (red bars), communication of spike data (yellow bars), neuronal update (green bars), and total sim time (black outline). Error bars (for most numbers of MPI processes hardly visible) indicate the SD over three repetitions. Timings obtained via manual instrumentation of the respective parts of the source code, measured on JURECA CM (Section 3.3).

One such engine is the open-source community code NEST[1] (Gewaltig and Diesmann, 2007). The quantitative analysis of the state-of-the-art in the present study is based on this code and alternative concepts are evaluated in its software framework. This ensures that ideas are immediately exposed to the complications and legacy of real-world code. NEST uses a hybrid between an event-driven and a time-driven simulation scheme to exploit that individual synaptic events are rare whereas the total number of spikes arriving at a neuron is large (Morrison et al., 2005). Neurons are typically updated every 0.1 ms and spike times are constrained to this time grid. For high-precision simulations, spikes can also be processed in continuous time (Morrison et al., 2007b; Hanuschkin et al., 2010). In contrast, synapses are only updated when a spike is arriving from the corresponding presynaptic neuron. The existence of a biophysical delay in the spiking interaction between neurons enables a global exchange of spike data between compute nodes in intervals of minimal delay. The data structures and algorithms for solving the equations of neuronal networks of natural size (Morrison and Diesmann, 2008; Helias et al., 2012; Kunkel et al., 2012, 2014; Jordan et al., 2018) as well as technology for network creation (Ippen et al., 2017) and the language interface (Eppler et al., 2009; Zaytsev and Morrison, 2014; Plotnikov et al., 2016) are documented and discussed in the literature in detail. For the purpose of the present study, it suffices to characterize the main loop of state propagation (Section 1) and concentrate on the details of spike delivery (Section 2).

Besides spikes, NEST supports gap junctions (Hahne et al., 2015; Jordan et al., 2020) as a further biophysical mechanism of neuronal interaction. To allow modeling of mechanisms affecting network structure on longer time scales, NEST implements models of neuromodulated synaptic plasticity (Potjans et al., 2010), voltage-dependent plasticity (Stapmanns et al., 2021), and structural plasticity (Diaz-Pier et al., 2016). For the representation of more abstract network models, NEST, in addition, supports binary neuron models (Grytskyy et al., 2013) and continuous neuronal coupling (Hahne et al., 2017) for rate-based and population models.

The present work is based on commit 059fe89 of the NEST 2.18 release.

3.2. Network Model

As earlier studies on neuronal network simulation technology (latest Jordan et al., 2018), we use a generic model of mammalian neuronal networks (Brunel, 2000) for measuring and comparing proposed algorithmic modifications. The model description and parameters are given in parameter Tables 1–3 of Jordan et al. (2018), and Section 2.4 gives an overview of performance for state-of-the-art code. A figure illustrating the structure of the

[1] https://www.nest-simulator.org

model is part of the NEST user-level documentation[2]. The sole difference of the investigated model with respect to previous studies is the restriction to static synapses for excitatory-excitatory connections. These synapses have a fixed weight whereas in former studies they exhibited spike-timing dependent plasticity (Morrison et al., 2007a).

The network is split into two populations: excitatory (80%) and inhibitory neurons (20%). These are modeled by single-compartment leaky-integrate-and-fire dynamics with alpha-shaped postsynaptic currents. Parameters are homogeneous across all neurons. Each neuron receives a fixed number of excitatory and inhibitory connections with presynaptic partners randomly drawn from the respective population. Thus, every neuron has 11, 250 incoming and, on average, 11, 250 outgoing synapses, independent of the network size. Inhibitory synapses are stronger than excitatory synapses to ensure the stability of the dynamical state of the network. The simulation of 10 ms of biological time, called the init phase, is followed by the further simulation of 1 s of biological time. The former initiates the creation and initialization of data structures that are unchanged in the simulation of subsequent time stretches. The measured wall-clock time of the latter, called the simulation phase, is referred to as "sim time." The mean firing rate across all network sizes considered in this study is 7.56 Hz with a SD of 0.1 Hz.

3.3. Systems

The JURECA Cluster Module (JURECA CM) and the K computer are already specified in Jordan et al. (2018), their characteristics are repeated here in the same words for completeness except the renaming of JURECA to JURECA CM after the addition of a booster module not used here. JURECA CM (Krause and Thörnig, 2018) consists of 1,872 compute nodes, each housing two Intel Xeon E5-2680 v3 Haswell CPUs with 12 cores each at 2.5 GHz for a total of 1.8 PFLOPS. Most of the compute nodes have 128 GB of memory available. In addition, 75 compute nodes are equipped with two NVIDIA K80 GPUs, which, however, are not used in this study. The nodes are connected via Mellanox EDR Infiniband.

Dynamical Exascale Entry Platform-Extreme Scale Technologies (DEEP-EST)[3] is an EU project exploring the usage of modular supercomputing architectures. Among other components, it contains a cluster module (DEEP-EST CM) tuned for applications requiring high single-thread performance and a modest amount of memory. The module consists of one rack containing 50 nodes, each node hosting two Intel Xeon Gold 6146 Skylake CPUs with 12 cores each. The CPUs run at 3.2 GHz and have 192 GB RAM. In total, the system has 45 TFLOPS and aggregates 45 TB of main memory. The system uses Mellanox InfiniBand EDR (100 GBps) with fat tree topology for communication.

Both on JURECA CM and DEEP-EST CM, we compile the application with OpenMP enabled using GCC and link against ParaStationMPI for MPI support. In our benchmarks, to match the node architecture, we launch 2 MPI processes each with 12 threads on every node and bind the MPI processes to sockets using `--cpu_bind=sockets` to ensure that the threads of each process remain on the same socket.

The K computer (Miyazaki et al., 2012) features 82, 944 compute nodes, each equipped with an 8-core Fujitsu SPARC64 VIIIfx processor operating at 2 GHz, with 16 GB RAM per node, leading to a peak performance of about 11.3 PFLOPS and a total of 1,377 TB of main memory. The compute nodes are interconnected via the "Tofu" ("torus connected full connection") network with 5 GBps per link. The K computer supports hybrid parallelism with OpenMP (v3.0) at the single node level and MPI (v2.1) for inter-node communication. Applications are compiled with the Fujitsu C/C++ Compiler and linked with Fujitsu MPI. Each node runs a single MPI process with 8 threads.

3.4. Software for Profiling and Workflow Management

Optimizing software requires the developer to identify critical sections of the code and to guarantee identical initial conditions for each benchmark. This is all the more true in the field of simulation technology for spiking neuronal networks. Despite the advances (Schenck et al., 2014; Cremonesi, 2019; Cremonesi and Schürmann, 2020; Cremonesi et al., 2020) in the categorization of neuronal network applications and the identification of bottlenecks, performance models are not yet sufficiently quantitative and fundamental algorithms and data structures are evolving. Therefore, the field still relies on exploration and quantitative experiments. The present work employs the profiling tool VTune to guide the development as well as the benchmarking environment JUBE for workflow management. In addition, the NEST code contains manual instrumentation to gather the cumulative times spent in the update, communicate, and deliver phases and to determine the total simulation time.

3.4.1. VTune

VTune Profiler[4] is a proprietary performance analysis tool developed by the company Intel providing both a graphical user interface and a command-line interface. It collects performance statistics across threads and MPI processes while the application is running. VTune supports different analysis types instructing the profiling program executing the application to focus on specific characteristics. From the rich set of statistical measures, we select only three basic quantities: Instructions Retired, Clockticks, and Clockticks per Instructions Retired (CPI). The Instructions Retired show the total number of completed instructions, while the CPI is the ratio of unhalted processor cycles (clockticks) relative to the number of instructions retired indicating the impact of latency on the application's execution.

3.4.2. JUBE

Documenting and reproducing benchmarking data requires the specification of metadata on the computer systems addressed and metadata on the configurations for compiling the application,

[2] https://nest-simulator.readthedocs.io

[3] https://www.deep-projects.eu

[4] https://software.intel.com/vtune

for running the simulations, and for evaluating the results. The Jülich Benchmarking Environment (JUBE) [5](Lührs et al., 2016) is a software suite developed by the Jülich Supercomputing Centre. We employ JUBE to represent all metadata of a particular benchmark by a single script.

4. REDESIGN OF SPIKE-DELIVERY ALGORITHM

The algorithmic redesign concentrates on the initial part of spike delivery and access to the spike ring buffers. The initial part of the original algorithm (Section 2.3) does not fully parallelize the sorting of spike events according to the target thread (Section 4.1). Furthermore, access to the spike ring buffers is hidden from the algorithm as the buffer is considered an implementation detail of the object representing a neuron (Section 4.2). Acronyms given in the titles of the subsections label the specific modifications for brevity and serve as references in pseudocode and figures.

4.1. Streamlined Processing of Spike Entries (SRR)

In the original spike-delivery algorithm (Section 2.3), each thread needs to read all spike entries in the MPI receive buffer, even those not relevant for its thread local targets, causing an overhead per spike entry, and hence per process-local synaptic target segment. Moreover, for each relevant spike entry, the thread hosting the targets needs to identify the correct innermost array in the three-dimensional resizable array storing the process-local synapses (**Figure 2**) based on the synapse-type information provided by the spike entry. This entails additional per target-segment overhead.

To address these issues, we adapt the original spike-delivery algorithm such that instead of directly dispatching the data from the receive buffer to the thread-local targets, we introduce a two-step process: First, the threads sort the spike entries by hosting thread and synapse type in parallel, and only then the threads dispatch the spikes, now exclusively reading relevant spike entries. To this end, we introduce a new data structure of nested resizable arrays, called spike-receive register (SRR), where each thread is assigned its own domain for writing. After each spike communication, a multi-threaded transfer of all spike entries from the MPI receive buffer to the spike-receive register takes place: each thread reads a different section of the entire receive buffer and transfers the entries to their SRR domains. The domains are in turn organized into separate resizable arrays, one per hosting thread. Nested resizable arrays enable the further sorting by synapse type. In this way, each element of the MPI receive buffer is read only once and spike entries are immediately sorted. This allows for a subsequent multi-threaded delivery of spikes from the spike-receive register to the corresponding target synapses and neurons such that all spike entries are exclusively read by their hosting thread. At this point, all a hosting thread has to do is to sequentially work through every resizable array exclusively prepared for it in the sorting phase. The additional sorting by synapse type allows the hosting thread to deliver all spikes targeting synapses of the same type in one pass.

4.2. Exposure of Code Dependencies (P2RB)

In the original spike-delivery algorithm (Section 2.3), the target synapse triggers the delivery of a spike to its target neuron, which then adds the spike to its spike ring buffer. For the entire spike-delivery process, this results in alternating access to target synapses and target neurons, or more precisely, the target neurons' spike ring buffers. As synapses store the target identifiers and other relevant information, access to a target synapse is a precondition for access to its target neuron.

In order to expose this code dependency, we separate the two delivery steps: spike delivery to target synapse and corresponding target neuron are now triggered sequentially at the same call-stack level. Moreover, instead of storing a target identifier, each synapse now stores a pointer to the target neuron's spike ring buffer allowing for direct access when delivering a spike. Therefore, the quantitative analysis (Section 5.1) refers to this set of optimizations as P2RB as an acronym for "pointer to ring buffer".

4.3. Pseudocode

SRR+P2RB syn: Adapted `Send()` function in synapse, which returns the pointer to the spike ring buffer of the target neuron (*target_rb*) owned by the synapse and the synaptic properties required for spike delivery to the target neuron.

```
Data: subsq, target_rb, delay, weight
Send()
    return (subsq, target_rb, delay, weight)
```

The pseudocode SRR+P2RB illustrates the changes to the original spike-delivery algorithm (ORI) resulting from the two new algorithms SRR (Section 4.1) and P2RB (Section 4.2).

Instead of a target-neuron identifier, each synapse now owns a pointer (*target_rb*) to the neuronal spike ring buffer. The synapse member function `Send()` returns the pointer and the synaptic properties *delay* and *weight* in addition to the indicator *subsq* (SRR+P2RB syn). This allows the algorithm to directly call `AddValue()` on the spike ring buffer (SRR+P2RB; **RB**) after the call to the synapse member function `Send()` (**SYN**). The `Receive()` member function of the target neuron (ORI nrn) is no longer required. Additionally, the algorithm now makes use of a spike receive register (*spike_reg*) for a preceding thread-parallel sorting of the spike entries from the MPI receive buffer by hosting thread (*tid*) and synapse type (*syn_id*), where each thread writes to its private region of the register (*spike_reg*[TID]). Spikes are then delivered from the spike receive register instead of the MPI receive buffer, where each thread processes only those regions

[5] https://www.fz-juelich.de/jsc/jube

SRR+P2RB: Detailed reference algorithm delivering spikes to local targets with support for multi-threading, where TID denotes the identifier of the executing thread. **TS** marks iteration over a synaptic target segment. **SYN** marks access to an individual target synapse (SRR+P2RB syn); **RB** marks access to the spike ring buffer of the corresponding target neuron. Based on ORI.

```
     Data: recv_buffer, synapses, spike_reg
     parallel foreach spike in recv_buffer do
         (tid, syn_id, lcid) ← spike.GetTargetLoc()
         spike_reg[TID][tid][syn_id].PushBack(spike)
     for syn_id ← 0 to MAX_SYN_ID do
         for tid ← 0 to MAX_TID do
             foreach spike in spike_reg[tid][TID][syn_id] do
                 lcid ← spike.lcid
                 subsq ← true
TS               while subsq do
SYN                  (subsq, target_rb, d, w) ←
                      synapses[TID][syn_id][lcid].Send()
                     lcid ← lcid + 1
RB                   target_rb.AddValue(d, w)
```

of the register that contain spike entries for thread-local targets (*spike_reg*[*tid*][TID] for all possible *tid*).

5. RESULTS

The new data structures and algorithms of Section 4 can be combined because they modify different parts of the code. As the efficiency of the optimizations may depend on the hardware architecture, we assess their performance on three computer systems (Section 5.1). Subsequently, we investigate in Section 5.2 the origin of the performance gain by evaluating the change in the total number of instructions required and the average number of clockticks consumed per instruction.

5.1. Effect of Redesign on Simulation Time

We select three computer systems for their differences in architecture and size (Section 3.3) to measure simulation times for a weak scaling of the benchmark network model (Section 3.2). The number of neurons per MPI process is significantly larger on the DEEP-EST CM and the JURECA CM (125, 000) than on the K computer (18, 000) making use of the respectively available amount of memory per process. On all three systems, we observe a relative reduction in simulation time by more than 30% (**Figure 4**) for the combined optimizations compared to the original code (ORI, Section 2.3). This includes the removal of a call to a function named `set_sender_gid()` from the generic spike delivery code (noSSG). This function attaches identifying information about the source of the corresponding spike which is only required by specific non-neuronal targets such as recorders. However, it causes per-target-segment overhead in all simulations. The functionality can hence be moved to a more specialized part of the code, e.g., the recorder model, and thereby regained if required. The DEEP-EST CM hardly benefits from the removal of the call but the batchwise processing of target segments has an increasing gain reaching 20% at 90 MPI processes. On JURECA CM, the function call does limit the performance and its removal alone improves the performance by 20% for large numbers of MPI processes. Across systems and a number of MPI processes, the combined optimizations lead to a sustained reduction in simulation time.

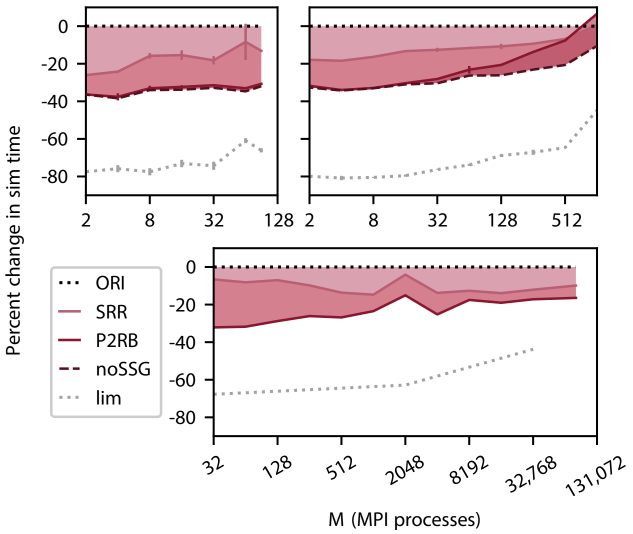


FIGURE 4 | Cumulative relative change in simulation time after a redesign of spike delivery as a function of the number of MPI processes M. Top left panel DEEP-EST CM and top right panel JURECA CM: linear-log representation for a number of MPI processes $M \in \{2;\ 4;\ 8;\ 16;\ 32;\ 64;\ 90\}$ and $M \in \{2;\ 4;\ 8;\ 16;\ 32;\ 64;\ 128;\ 256;\ 512;\ 1024\}$, respectively. Weak scaling of benchmark network model with the same configuration as in **Figure 3**; error bars show SD based on 3 repetitions. Bottom panel K computer: number of MPI processes $M \in \{32;\ 64;\ 128;\ 256;\ 512;\ 1024;\ 2048;\ 4096;\ 8192;\ 16,384;\ 32,768;\ 82,944\}$, gray dotted curve: $M \in \{32;\ 2048;\ 32,768\}$. Weak scaling with different configurations (1 MPI process per compute node; 8 threads per MPI process; 18, 000 neurons per MPI process). The black dotted line at zero indicates the performance of the original code (ORI, Section 2.3). The light carmine red curve indicates a change in sim time (shading fills area to reference) due to sorting of spike entries prior to delivery (SRR, Section 4.1). The dark carmine red curve indicates an additional change in sim time due to providing synapses with direct pointers to neuronal spike ring buffers (P2RB, Section 4.2). The dashed brown curve shows an additional change in sim time after removal of an unrequired generic function call (noSSG). Gray dotted curve indicates a hypothetical limit to the decrease in sim time assuming spike delivery takes no time.

The new data structures and algorithms address the spike-delivery phase only, but an optimization can only reduce simulation time to the extent the component of the code to be optimized contributes to the total time consumed as indicated by the limiting curve in **Figure 4**. In the neuronal network simulations considered here, the delivery of spikes from MPI buffers to their targets consumes the major part of simulation time. Initially, spike delivery takes up 80% of the simulation time for the DEEP-EST CM and the JURECA CM and 70% for the K computer, but on all three systems, the relative contribution decreases with an increasing number of MPI processes. Under weak scaling into regimes beyond 1,024 MPI processes, the absolute time required for spike delivery also initially grows but converges as the expected number of thread-local targets per spike converges to one (cf. Jordan et al., 2018). Although spike-delivery time increases throughout the entire range of MPI processes on DEEP-EST CM and JURECA CM, the relative contribution to simulation time declines because the time required by communication between MPI processes increase more rapidly (for JURECA CM data cf. **Figure 3**).

P2RB increases the size of synapse objects by introducing an 8 B pointer to the neuronal spike ring buffer replacing the 2 B local neuron index of the original algorithm (Section 2.3). We hypothesize that this increase underlies the declining success and ultimately disadvantageous effect observed on JURECA CM. Control simulations using the original code but with an artificially increased object size confirm this hypothesis (data not shown).

5.2. Origin of Improvement

The new data structures and algorithms realize a more fine-grained parallelization and avoid indirections in memory accesses (Section 4). These changes significantly speed up the application (**Figure 4**) across architectures and network sizes. In order to understand the origin of this improvement, we employ the profiling tool VTune (Section 3.4.1) which gives us access to the CPU's microarchitectural behavior. In the analysis, we concentrate on the total number of instructions executed and the clockticks per instruction retired (CPI).

The total number of instructions decreases by close to 50% on all scales. Nevertheless, the contribution of noSSG to the reduction in the number of instructions becomes larger as this algorithm removes code which is called for every target segment. The number of target segments, however, increases with the number of MPI processes until a limit is asymptotically approached (**Figure 1**).

For small problem sizes, the CPI decreases when compared against the baseline (SRR+P2RB), but at around 32 MPI processes, the instructions start to consume more clockticks than in the original algorithm (**Figure 5**). This behavior is apparent on DEEP-EST CM as well as JURECA CM where the additional noSSG optimizations improve performance slightly. We interpret this observation as follows. Initially the more orderly organization of memory enables a shorter latency in memory access. At larger network sizes, CPI is dominated by memory access to the fragmented target segments and this dominance is more pronounced as the new code spends fewer instructions on reading the receive buffer.

Taken individually, the two metrics alone are not sufficient for explaining the decrease in simulation time (**Figure 4**). The product of the number of instructions retired and CPI expresses their interplay and reduces to the total number of clockticks required. Thus, this product is a quantity that directly relates to the separately measured sim time. Indeed, the comparison of this measure, depicted in **Figure 5**, with **Figure 4** shows that the product qualitatively explains the change in sim time. While CPI increases beyond the baseline, the growth in sim time is slowed down by having fewer instructions in total.

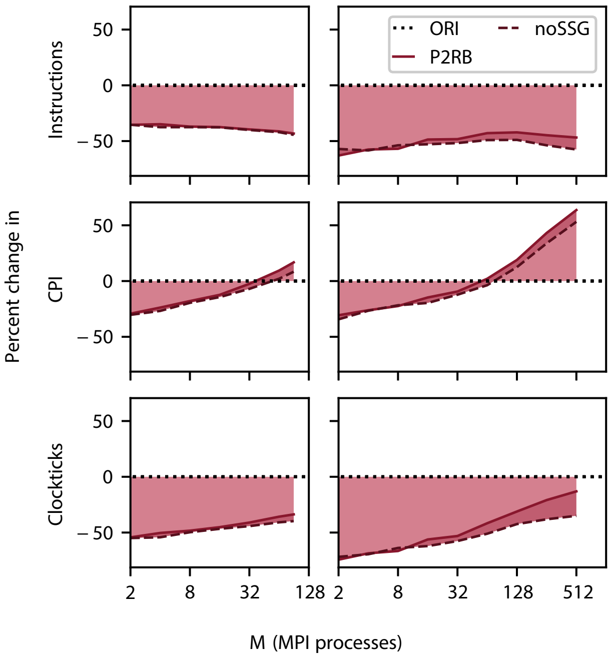


FIGURE 5 | Relative change in instructions retired (top row), clockticks per instruction retired (CPI, middle), and clockticks (bottom) during spike delivery for P2RB (including SRR as in **Figure 4**), and noSSG as a function of the number of MPI processes. Raw data for all three quantities was obtained by VTune (Section 3.4.1). Left column DEEP-EST CM and right column JURECA CM: linear-log representation for number of MPI processes $M \in \{2;\ 4;\ 8;\ 16;\ 32;\ 64;\ 90\}$ and $M \in \{2;\ 4;\ 8;\ 16;\ 32;\ 64;\ 128;\ 256;\ 512\}$, respectively. Black dotted line at zero percent (ORI, Section 2.3) indicates the performance of the original code. Weak scaling of benchmark network model as in **Figure 4**.

6. DISCUSSION

Our investigation characterizes the dominance of the spike-delivery phase in a weak-scaling scenario for a typical random network model (Section 2.4). At small to medium network sizes, spike delivery is the sole major contributor to simulation time. Only if thousands of compute nodes are involved, communication between nodes becomes prominent (**Figure 3**) while spike delivery remains the largest contributor (Jordan et al., 2018). The absolute time spent on spike delivery is dominating for small networks and grows with increasing network size. The reason for this is that the random network under study takes into account that in the mammalian brain a neuron can send spikes to more than ten-thousand targets. With increasing network size, these targets are distributed over more and more compute nodes until in the limit a neuron either finds a single target on a given compute node or more likely none at all. As the number of synapses a compute node represents is invariant under weak scaling, the node needs to process an increasing number of spikes from different source neurons. For the simulation parameters in this study, the expected number of unique source neurons and thereby absolute costs approach the limit when the network is distributed across thousands of compute nodes (Section 2.1) thus also limiting the costs of spike delivery.

In spike delivery, a thread inspects all spikes arriving at the compute node. If the thread hosts at least one target neuron of a spike, the thread needs to access a three-dimensional data structure (**Figure 2**) to activate the corresponding synapses and ultimately under consideration of synapse specific delays place the spike in the ring buffers of the target neurons. The present work investigates whether an alternative algorithm can reduce the number of instructions and decisions when handling an individual spike. The hypothesis is that a more compact code and more predictable control flow allows modern processors a faster execution. On purpose, no attempt is made to apply techniques like hardware prefetching or software pipelining to conceptually separate improvements of the logic of the algorithm from further optimizations that may have a stronger processor dependence. Nevertheless, we hope that the insights of the present work provide the basis for any future exploration of these issues.

The first step in our reconsideration of the spike-delivery algorithm is to look at the initial identification of the relevant spikes for each thread. Originally, each thread inspects all spikes. This means that the algorithms perform many read operations on spikes without further actions and that their proportion increases with an increasing number of threads per compute node. The alternative algorithm which we refer to as SRR (Section 4.1) carries out a partial sorting of the spikes. Each thread is responsible for an equally sized chunk of the incoming spikes and sorts them into a data structure according to the thread on which the target neuron resides and according to the type of the target synapse. Once all threads have completed their work, they find a data structure containing only relevant spikes and complete the spike delivery entirely independently from the other threads. This already leads to a reduction of simulation time between 10 and 20 % on the three computer systems tested while the detailed development of this fraction differs with network size.

As a second step, we remove an indirection originating in the initial object oriented design of the simulation code. Following the concept of describing entities of nature by software objects, neurons became objects receiving and emitting spikes and neuronal spike ring buffers an implementation detail of no relevance for other components. As a consequence when a neuron object receives a spike it needs to decide in which ring buffer to place the spike, for example, to separate excitatory from inhibitory inputs, and delegate this task to the respective buffer. Our alternative algorithm (P2RB, Section 4.2) exposes the corresponding spike ring buffer to the synapse at the time of network construction. The synapse stores the direct pointer and no further decision is required during simulation. This change further reduces simulation time by 10 to 20 %.

One computer system (JURECA CM) shows a pronounced decline in the computational advantage of the combined new algorithm (SRR+P2RB) for large network sizes, which in the case of P2RB, we assume to be due to an increase in synaptic memory footprint. An additional optimization removing a generic function call that enriches spike events by information on the identity of the source neuron mitigates the loss in performance. As this functionality is not required for the interaction between neurons, we moved the function to a more specialized part of the code (noSSG, Section 5.1).

The achievement of the combined algorithm (SRR+P2RB+noSSG) needs to be judged in light of the potential maximum gain. For small networks, spike delivery consumes 70 to 80 % of simulation time, depending on the computer system, while this relative contribution declines with growing network sizes as communication becomes more prominent. Thus, the streamlined processing of spikes reduces spike delivery by 50 % largely independent of network size. In conclusion, with the new algorithm, spike delivery still substantially contributes to simulation time.

In the small to medium scale regime (DEEP-EST CM, JURECA CM), the new code gains its superiority from executing only half of the number of instructions of the original implementation (Section 5.2). The reduction becomes slightly larger with increasing network size. This is plausible as for a given thread, the algorithm avoids processing a growing number of irrelevant spikes (SRR). As the number of synapses per compute node is fixed, but neurons have a decreasing number of targets per compute node, the number of relevant spikes increases. Therefore, decreasing the number of function calls per spike has an increasing benefit.

The picture is less clear for the average number of clockticks required to complete an instruction (Section 5.2). For small networks, the new algorithms exhibit an advantage. However, with increasing network size, eventually more clockticks per instruction are required than by the original algorithm. Nevertheless, these latencies are hard to compare as the new algorithm executes only half of the instructions and may therefore put memory interfaces under larger stress. This result already indicates that methods of latency hiding may now be successful in further reducing spike-delivery time. The product of the number of instructions and the clockticks per instruction gives an estimate of the total number of clockticks required. The observed stable improvement across all network sizes confirms the direct measurements of simulation time.

Faster simulation can trivially be achieved by reducing the generality of the code or by reducing the accuracy of the simulation. While the SRR optimization does not touch the code of individual neuron or synapse models, a critical point in the P2RB optimization with regard to code generality is the replacement of the target identifier in the synapse object (Section 2.3) by a pointer to the corresponding spike ring buffer. Synaptic plasticity is the biological phenomenon by which the strength of a synapse changes in dependence on the spiking activity of the presynaptic and the postsynaptic neuron. This is one of the key mechanisms by which brains implement system-level learning. For a wide class of models of synaptic plasticity, it is sufficient to update the synaptic weight when a presynaptic spike arrives at the synapse (Morrison et al., 2008; Stapmanns et al., 2020). However, at this point in time, the synapse typically needs to inspect a state variable of the postsynaptic neuron or even retrieve the spiking history of the postsynaptic neuron since the last presynaptic spike. This information is only available in the neuron, not in the spike ring buffer. Still, generality is preserved as in the reference simulation engine (Gewaltig and Diesmann, 2007) synapses are not restricted to a single strategy for accessing the target neuron or its spike ring buffer. A static synapse can implement the P2RB idea while a plastic synapse stays with the target identifier from which the state of the neuron, as well as the spike ring buffer, can be reached. But in this way, a plastic synapse does not profit from the advantages of P2RB at all. There are two alternatives. First, the spike ring buffer can be equipped with a pointer to the target neuron. This requires an indirection in the update of the synapse but still avoids the need to select the correct ring buffer during spike delivery. Second, the synapse can store both a target identifier and a pointer to the ring buffer. This removes the indirection for the price of additional per-synapse memory usage. There are no fundamental limitations preventing us from making both solutions available to the neuroscientist *via* different synapse types. In fact, this strategy is currently in use, for example, to provide synapse types with different target identifiers either consuming less memory or requiring fewer indirections (Section 2.2), where template-based solutions prevent the duplication of entire model codes. Users can thus select the optimal synapse-type version depending on the amount of memory available. However, making multiple versions of the same model available reduces the user-friendliness of the application. A domain specific language like NESTML (Plotnikov et al., 2016) may come to the rescue here generating more compact or faster code depending on hints of the neuroscientists to the compiler. This idea could be extended to other parts of the simulation cycle where further information is required to decide on a suitable optimization.

The incoming spike events of a compute node specify the hosting thread as well as the location of the synaptic targets, but they are unsorted with respect to the hosting thread and synapse type. Nevertheless, the present work shows that the processing of spikes can be completely parallelized requiring only a single synchronization between the threads at the point where the spikes are sorted according to target thread and synapse type, which is when all spikes have been transferred from the MPI receive buffer into the novel spike-receive register. This suggests that spike delivery fully profits from a further increase in the number of threads per compute node. Although here we concentrate on compute nodes with an order of ten cores per processor, we expect that the benefits of parallelization extend to at least an order of magnitude more cores, which matches recent hardware developments. The scaling might still be limited by the structure of the spike-receive register having separate domains for each thread writing spike data from the MPI spike receive buffer to the register. If the same number of spikes is handled by more threads, the spikes are distributed to more domains of the register such that during the actual delivery from the register to the thread-local targets each thread needs to collect its spikes from more memory domains.

The local processing of a compute node is now better understood and for large networks, the communication between nodes begins to dominate simulation time already for the machines investigated here. Current chip technology is essentially two-dimensional in contrast to the three-dimensional organization of the brain and parallelization in the brain is more fine grained. Inside, a compute node technology compensates for these advantages by communication over buses. After substantially reducing the number of instructions, we see indications that memory latency is a problem when spikes from many sources need to be processed. Therefore, it remains to be seen whether techniques of latency hiding can further push the limits imposed by the von Neumann bottleneck. Any neuromorphic hardware based on compute nodes communicating by a collective spike exchange in fixed time intervals needs to organize routing of the spikes to target neurons. The ideas presented in the present study on streamlining this process by partial parallel sorting may help in the design of adequate hardware support. However, between compute nodes, the latency of state-of-the-art inter-node communication fabrics is likely to be the next limiting factor for simulation. A possible approach to mitigate this problem is the design of dedicated neuromorphic hardware explicitly optimized for communication. The SpiNNaker project (Furber et al., 2013; Furber and Bogdan, 2020), for example, follows an extreme approach by routing packets with individual spikes to the respective processing units.

Part of the improvements in performance this study achieved come at the price of an increase in the number of lines of code and an increase in code complexity. In general, one needs to weigh the achieved performance improvements against detrimental effects on maintainability. This is particularly relevant for a community code like the one under consideration, in which experienced developers are continuously replaced by new contributors. Highly optimized code may be more difficult to keep up to date and adjust to future compute node architectures. Next to conceptual documentation of optimization to core algorithms, code generation, as explored in the NESTML project (Plotnikov et al., 2016), may be come part of a strategy to reduce this friction between performance and code accessibility. A domain specific language lets a spectrum of users concentrate on the formal description of the problem while experienced developers make sure the generator produces optimized code, possibly even adapted to specific target architectures. Until simulations are fast enough to enable the investigation of plastic networks at natural density we have to find ways to cope with increasing complexity of the algorithms and their respective implementations.

Over the last two decades, studies on simulation technology for spiking neuronal networks regularly report improvements in simulation speed on the order of several percent and improved scaling compared to the state-of-the-art technology. The stream of publications on simulation technology in the field shows that there was and still is room for substantial improvements. Nevertheless, at first sight, it seems implausible that over this time span no canonical algorithm has emerged and progress shows no sign of saturation. The solution to this riddle is that new articles tend to immediately concentrate on the latest available hardware and are interested in their limits in terms of network size. This is driven by the desire of neuroscience to overcome the limitations of extremely downscaled models and arrive at a technology capable of representing relevant parts of the brain. Moreover, investigations of novel models in computational neuroscience have a life-cycle of roughly 5 years, the same time scale at which supercomputers are installed and decommissioned. Thus, both representative network models and the hardware to simulate them are in flux, which makes comprehensive performance studies difficult. The software evolution of spiking network simulation code is largely unknown and the community may profit from a review exposing dead ends and volatile locations of the algorithm. For more systematic monitoring of technological progress, the community needs to learn how to establish and maintain reference models and keep track of benchmarking data and their respective metadata.

The present study streamlines the routing of spikes in a compute node by a fully parallel partial sorting of incoming spikes and refactoring of the code. This halves the number of instructions for this phase of the simulation and leads to a substantial reduction in simulation time. We expect that our work provides the basis for the successful application of techniques of latency hiding and vectorization.

DATA AVAILABILITY STATEMENT

All datasets and code of this study are available in a public repository (Pronold et al., 2021).

AUTHOR CONTRIBUTIONS

All authors listed have made a substantial, direct, and intellectual contribution to the work and approved it for publication.

FUNDING

Partly supported by the European Union's Horizon 2020 (H2020) funding framework under grant agreement no. 785907 (Human Brain Project, HBP SGA2), no. 945539 (HBP SGA3), and no. 754304 (DEEP-EST), the Helmholtz Association Initiative and Networking Fund under project number SO-092 (Advanced Computing Architectures, ACA), and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 368482240/GRK2416. The use of the JURECA supercomputer in Jülich was made possible through VSR computation time grant Brain-Scale Simulations JINB33. This research used resources of K computer at the RIKEN Advanced Institute for Computational Science. Supported by the project Exploratory Challenge on Post-K Computer (Understanding the neural mechanisms of thoughts and its applications to AI) of the Ministry of Education, Culture, Sports, Science and Technology (MEXT).

ACKNOWLEDGMENTS

We are grateful to Mitsuhisa Sato for his guidance, who helped us shape the project, to Johanna Senk and Dennis Terhorst for fruitful discussions and joint efforts during the HPC Optimisation and Scaling Workshop 2019 at the Jülich Supercomputing Centre, Germany, to Sebastian Lührs for his help with JUBE, to our colleagues in the Simulation and Data Laboratory Neuroscience of the Jülich Supercomputing Centre for continuous collaboration, and to the members of the NEST development community for their contributions to the concepts and implementation of the NEST simulator. All network simulations were carried out with NEST (http://www.nest-simulator.org).

REFERENCES

Akar, N. A., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor -a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019, 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia: IEEE). 274–282.

Billeh, Y. N., Cai, B., Gratiy, S. L., Dai, K., Iyer, R., Gouwens, N. W., et al. (2020). Systematic integration of structural and functional data into multiscale models of mouse primary visual cortex. *Neuron* 106, 388.e18–03.e18. doi: 10.1016/j.neuron.2020.01.040

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 183–208. doi: 10.1023/a:1008925309027

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge: Cambridge University Press.

Cremonesi, F. (2019). *Computational characteristics and hardware implications of brain tissue simulations*, Technical Report, EPFL.

Cremonesi, F., Hager, G., Wellein, G., and Schürmann, F. (2020). Analytic performance modeling and analysis of detailed neuron simulations. *Int. J. High Perform. Comput. Appl.* 34, 428–449. doi: 10.1177/1094342020912528

Cremonesi, F., and Schürmann, F. (2020). Understanding computational costs of cellular-level brain tissue simulations through analytical performance models. *Neuroinformatics* 18, 407–428. doi: 10.1007/s12021-019-09451-w

Diaz-Pier, S., Naveau, M., Butz-Ostendorf, M., and Morrison, A. (2016). Automatic generation of connectivity for large-scale neuronal network models through structural plasticity. *Front. Neuroanatomy* 10:57. doi: 10.3389/fnana.2016.00057

Einevoll, G. T., Destexhe, A., Diesmann, M., Grün, S., Jirsa, V., Kamps, M., et al. (2019). The scientific case for brain simulations. *Neuron*. 102, 735–744. doi: 10.1016/j.neuron.2019.03.027

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M. (2009). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2:12. doi: 10.3389/neuro.11.012.2008

Furber, S., and Bogdan, P. (2020). *SpiNNaker: A Spiking Neural Network Architecture*. Boston, MA; Delft: Now Publishers.

Furber, S., Lester, D., Plana, L., Garside, J., Painkras, E., Temple, S., et al. (2013). Overview of the SpiNNaker system architecture. *IEEE Trans. Comp.* 62, 2454–2467. doi: 10.1109/TC.2012.142

Gewaltig, M.-.O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430. doi: 10.4249/scholarpedia.1430

Grytskyy, D., Tetzlaff, T., Diesmann, M., and Helias, M. (2013). A unified view on weakly correlated recurrent networks. *Front. Comput. Neurosci.* 7, 131. doi: 10.3389/fncom.2013.00131

Hahne, J., Dahmen, D., Schuecker, J., Frommer, A., Bolten, M., Helias, M., et al. (2017). Integration of continuous-time dynamics in a spiking neural network simulator. *Front. Neuroinform*. 11:34. doi: 10.3389/fninf.2017.00034

Hahne, J., Helias, M., Kunkel, S., Igarashi, J., Bolten, M., Frommer, A., et al. (2015). A unified framework for spiking and gap-junction interactions in distributed neuronal network simulations. *Front. Neuroinform*. 9:22. doi: 10.3389/fninf.2015.00022

Hanuschkin, A., Kunkel, S., Helias, M., Morrison, A., and Diesmann, M. (2010). A general and efficient method for incorporating precise spike times in globally time-driven simulations. *Front. Neuroinform*. 4, 113. doi: 10.3389/fninf.2010.00113

Helias, M., Kunkel, S., Masumoto, G., Igarashi, J., Eppler, J. M., Ishii, S., et al. (2012). Supercomputers ready for use as discovery machines for neuroscience. *Front. Neuroinform*. 6:26. doi: 10.3389/fninf.2012.00026

Ippen, T., Eppler, J. M., Plesser, H. E., and Diesmann, M. (2017). Constructing neuronal network models in massively parallel environments. *Front. Neuroinform*. 11:30. doi: 10.3389/fninf.2017.00030

Joglekar, M. R., Mejias, J. F., Yang, G. R., and Wang, X.-J. (2018). Inter-areal balanced amplification enhances signal propagation in a large-scale circuit model of the primate cortex. *Neuron* 98, 222–234. doi: 10.1016/j.neuron.2018.02.031

Jordan, J., Helias, M., Diesmann, M., and Kunkel, S. (2020). Efficient communication in distributed simula-tions of spiking neuronal networks with gap junctions. *Front. Neuroinform*. 14:12. doi: 10.3389/fninf.2020.00012

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Ex-tremely scalable spiking neuronal network simulation code: From laptops to exascale computers. *Front. Neuroinform*. 12:2. doi: 10.3389/fninf.2018.00002

Krause, D., and Thörnig, P. (2018). JURECA: modular supercomputer at Jülich Supercomputing Centre. *J. Largescale Res. Facilit*. 4, A132. doi: 10.17815/jlsrf-4-121-1

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., et al. (2019). Coreneuron: an optimized compute engine for the neuron simulator. *Front. Neuroinform*. 13, 63. doi: 10.3389/fninf.2019.00063

Kunkel, S. (2019). "Routing brain traffic through the bottlenecks of general purpose computers: challenges for spiking neural network simulation code, ISC 33 (2019)," in *High Performance Computing* (Frankfurt: ISC High Performance 2019 International).

Kunkel, S., Potjans, T. C., Eppler, J. M., Plesser, H. E., Morrison, A., and Diesmann, M. (2012). Meeting the memory challenges of brain-scale simulation. *Front. Neuroinform*. 5:35. doi: 10.3389/fninf.2011.00035

Kunkel, S., Schmidt, M., Eppler, J. M., Masumoto, G., Igarashi, J., Ishii, S., et al. (2014). Spiking network simulation code for petascale computers. *Front. Neuroinform*. 8:78. doi: 10.3389/fninf.2014.00078

Lührs, S., Rohe, D., Schnurpfeil, A., Thust, K., and Frings, W. (2016). "Flexible and generic workflow management," in: *Parallel Computing: On the Road to Exascale, Volume 27 of Advances in Parallel Computing* (Amsterdam: IOS Press), 431–438.

Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

Miyazaki, H., Kusano, Y., Shinjou, N., Fumiyoshi, S., Yokokawa, M., and Watanabe, T. (2012). Overview of the K computer system. *Fujitsu Scientific Techn. J*. 48, 255–265.

Morrison, A., Aertsen, A., and Diesmann, M. (2007a). Spike-timing dependent plasticity in balanced random networks. *Neural Comput*. 19, 1437–1467. doi: 10.1162/neco.2007.19.6.1437

Morrison, A., and Diesmann, M. (2008). "Maintaining causality in discrete time neuronal network simulations," in *Lectures in Supercomputational Neuro-Sciences: Dynamics in Complex Brain Networks*, eds P. B. Graben, C. Zhou, M. Thiel, and J. Kurths (Berlin; Heidelberg: Springer) 267–278.

Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike-timing. *Biol. Cybern*. 98, 459–478. doi: 10.1007/s00422-008-0233-1

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput*. 17, 1776–1801. doi: 10.1162/0899766054026648

Morrison, A., Straube, S., Plesser, H. E., and Diesmann, M. (2007b). Exact subthreshold integration with continuous spike times in discrete-time neural network simulations. *Neural Comput*. 19, 47–79. doi: 10.1162/neco.2007.19.1.47

Plotnikov, D., Blundell, I., Ippen, T., Eppler, J. M., Rumpe, B., and Morrison, A. (2016). "NESTML: a modeling language for spiking neurons," in *Modellierung 2016. volume P-254 of Lecture Notes in Informatics (LNI)*, eds A. Oberweis and R. Reussner (Gesellschaft für Informatik e.V. (GI)), 93–108. Available online at: http://juser.fz-juelich.de/record/826510.

Potjans, T. C., and Diesmann, M. (2014). The cell-type specific cortical microcircuit: Relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Potjans, W., Morrison, A., and Diesmann, M. (2010). Enabling functional neural circuit simulations with distributed computing of neuromodulated plasticity. *Front. Comput. Neurosci*. 4:141. doi: 10.3389/fncom.2010.00141

Pronold, J., Jordan, J., Wylie, B., Kitayama, I., Diesmann, M., and Kunkel, S. (2021). Code for Routing brain traffic through the von Neumann bottleneck: Parallel sorting and refactoring. doi: 10.5281/zenodo.5148731

Schenck, W., Adinetz, A. V., Zaytsev, Y. V., Pleiter, D., and Morrison, A. (2014). "Performance model for large-scale neural simulations with NEST," in *Extended Poster Abstracts of the SC14 Conference for Supercomputing* (New Orleans, LA).

Schmidt, M., Bakker, R., Hilgetag, C. C., Diesmann, M., and van Albada, S. J. (2018a). Multi-scale account of the network structure of macaque visual cortex. *Brain Struct. Funct*. 223, 1409–1435. doi: 10.1007/s00429-017-1554-4

Schmidt, M., Bakker, R., Shen, K., Bezgin, G., Diesmann, M., and van Albada, S. J. (2018b). A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas. *PLoS Comput. Biol*. 14:e1006359. doi: 10.1371/journal.pcbi.1006359

Stapmanns, J., Hahne, J., Helias, M., Bolten, M., Diesmann, M., and Dahmen, D. (2020). Event-based update of synapses in voltage-based learning rules. *arXiv:2009*, 08667.

Stapmanns, J., Hahne, J., Helias, M., Bolten, M., Diesmann, M., and Dahmen, D. (2021). Event-based update of synapses in voltage-based learning rules. *Front. Neuroinform*. 15:609147. doi: 10.3389/fninf.2021.609147

Zaytsev, Y. V., and Morrison, A. (2014). CyNEST: a maintainable Cython-based interface for the NEST simulator. *Front. Neuroinform*. 8:23. doi: 10.3389/fninf.2014.00023

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Neuroscience

ORIGINAL RESEARCH
published: 10 May 2022
doi: 10.3389/fnins.2022.867027

Parallelization of Neural Processing on Neuromorphic Hardware

Luca Peres and Oliver Rhodes*

Advanced Processor Technologies Group, Department of Computer Science, The University of Manchester, Manchester, United Kingdom

Learning and development in real brains typically happens over long timescales, making long-term exploration of these features a significant research challenge. One way to address this problem is to use computational models to explore the brain, with Spiking Neural Networks a popular choice to capture neuron and synapse dynamics. However, researchers require simulation tools and platforms to execute simulations in real- or sub-realtime, to enable exploration of features such as long-term learning and neural pathologies over meaningful periods. This article presents novel multicore processing strategies on the SpiNNaker Neuromorphic hardware, addressing parallelization of Spiking Neural Network operations through allocation of dedicated computational units to specific tasks (such as neural and synaptic processing) to optimize performance. The work advances previous real-time simulations of a cortical microcircuit model, parameterizing load balancing between computational units in order to explore trade-offs between computational complexity and speed, to provide the best fit for a given application. By exploiting the flexibility of the SpiNNaker Neuromorphic platform, up to 9× throughput of neural operations is demonstrated when running biologically representative Spiking Neural Networks.

Keywords: neuromorphic computing, SpiNNaker, real-time, parallel programming, event-driven simulation, spiking neural networks

OPEN ACCESS

Edited by:
James Courtney Knight,
University of Sussex, United Kingdom

Reviewed by:
Garibaldi Pineda García,
University of Sussex, United Kingdom
Arren Glover,
Italian Institute of Technology (IIT), Italy

****Correspondence:***
Luca Peres
luca.peres-2@manchester.ac.uk

Specialty section:
This article was submitted to Neuromorphic Engineering, a section of the journal Frontiers in Neuroscience

Received: *31 January 2022*
Accepted: *08 April 2022*
Published: *10 May 2022*

Citation:
Peres L and Rhodes O (2022) Parallelization of Neural Processing on Neuromorphic Hardware. Front. Neurosci. 16:867027. doi: 10.3389/fnins.2022.867027

1. INTRODUCTION

The human brain is capable of operating using less energy than a light bulb (Levy and Calvert, 2020). However, simulation of biologically representative Spiking Neural Networks (SNN) is a challenging task on conventional computer hardware. Models from the literature can produce millions of spikes per second, which need to be delivered to hundreds of thousands of neurons (Potjans and Diesmann, 2012; Schmidt et al., 2018; Casali et al., 2019) with very tight timing constraints. A common way to simulate these network dynamics is through CPU-based HPC platforms, using dedicated software such as NEST (Gewaltig and Diesmann, 2007). However, because of the timing constraints and the intrinsic high parallelism of these tasks, they fail to keep energy consumption low when attempting to run these applications, and performance gain is limited by the latency of MPI-based (Ippen et al., 2017) communications. An alternative approach, namely Neuromorphic engineering, inspired by the structure of the brain (Mead, 1990), has proven effective when dealing with this type of simulation (Rhodes et al., 2019), efficiently addressing the sparsity of signals typical of these applications and keeping energy consumption low. This approach is characterized by simple computational units with close access to distributed memory (Mead, 1990; Indiveri et al., 2011). To date, several Neuromorphic platforms have been developed, both in the digital, analog and mixed signal domains (Furber et al., 2014; Akopyan et al., 2015; Schemmel et al., 2017; Davies et al., 2018; Moradi et al., 2018). From a digital perspective, neurons (or neural compartments) are implemented by processors, which usually simulate both the neural dynamics and the synaptic receptors. Analog platforms on the other hand, employ a circuit implementation of models from literature. The efficiency of such systems is usually measured in terms of synaptic events (namely one spike targetting one synapse) per second and neurons they can simulate, with these two measures limited by the on-core memory capacity and computational power in digital neuromorphic platforms and by the physical implementation for analog architectures.

High synaptic fan-in represents one of the biggest challenges in biologically representative SNNs and it usually prevents real-time execution, requiring to slow down the simulations (i.e., resulting in a simulated time longer than the biological time) to process all network activity. Another strong limitation is given by long-range connections between different brain areas (Schmidt et al., 2018), which are typically represented by extremely sparse connectivity patterns. Recent work (Rhodes et al., 2019) demonstrated that, by performing more efficient task-partitioning and by acting on the placement of networks on Neuromorphic hardware, it is possible to improve significantly the throughput of these systems, enabling real-time execution of models that were not possible before.

Real-time simulations of biologically-representative SNNs are a common target in the field. Several solutions have been proposed to address the presented issues, including a procedural generation of the synaptic weights whenever a spike is received, instead of storing these, to reduce the memory footprint and improve performance (Knight and Nowotny, 2021). Some digital simulation platforms managed to achieve remarkable results in terms of real-time simulations, even reaching sub real-time performance for established benchmarks in the field (Knight et al., 2021; Kurth et al., 2021; Heittmann et al., 2022).

This work offers an improved parallelization strategy, namely the Multi-target partitioning, on how to efficiently deploy Spiking Neural Networks on Neuromorphic hardware. This strategy aims at addressing the major bottlenecks of SNN simulations and informing the design of the next generation of Neuromorphic platforms. The use-case platform chosen for this work is SpiNNaker, a many-core digital Neuromorphic platform designed at The University of Manchester (Furber et al., 2014).

Following this introduction, Section 2 provides a background on SNNs simulations in general, together with the critical aspects of real-time simulations and their challenges. Section 3 gives details about the SpiNNaker Neuromorphic platform and how SNNs are mapped on it through the available partitioning strategies. The Multi-target partitioning approach is then presented in Section 3.4. Section 4 demonstrates the advantages of this new strategy through benchmarking on SpiNNaker. Finally, Section 5 contains a discussion about the potentialities of this approach and possible future applications.

2. BACKGROUND

2.1. Neural Processing

SNN simulations are typically performed starting from a high level description of the network characteristics, through high level specification languages such as PyNN (Davison et al., 2009). Groups of neurons sharing the same properties are grouped into ensembles called Populations, and the connections between them are called Projections. Starting from these high level descriptions, Populations and Projections are typically fragmented (or partitioned) such that they can fit the requirements set by the underlying hardware platform. Digital platforms commonly employ a discrete time resolution, using fixed length timesteps, within which all spikes are considered to happen at the same time. Each computational unit involved in the simulation is in charge of handling a subset of a Population, meaning that it needs to update the state of a predefined number of neurons, generate output spikes for those neurons and receive input spikes.

A representation of a neural simulation is shown in **Figure 1**. Two Populations are shown (left), called Pre and Post, respectively, and the neurons are connected with a probability P, meaning that each presynaptic neuron has a probability P to connect to each postsynaptic neuron. An interaction of simulation events is shown on the right, where 3 simulation timesteps are presented for both Populations. In this case each Population is simulated by a separate computational unit. Timesteps are indicated by Δt, and are synchronized among the involved computational units. Both computational units update the neural state for their implemented neurons according to the neuron model equations (light and dark green, respectively, in **Figure 1**). After the state update, neurons will fire, generating spikes that will be delivered to the Post neurons. The remaining fraction of the timestep (t_P in **Figure 1**) is commonly used to process the incoming spikes (light blue).

During a simulation, the length of the green bar (t_{upd}) is constant. Increasing the number of neurons per computational unit extends the green bar. When the input firing activity is high (commonly when the number of input connections is high), or P is increased, the blue bar grows. The length of the blue bar therefore varies according to the amount of synaptic inputs received during the timestep. In order to maintain real-time processing, both the bars need to complete the execution before the beginning of the subsequent timestep (therefore before the next green bar is due to start). **Figure 1** shows an example of a non real-time simulation, where the first timestep for the Post computational unit completes in time, however, during the second timestep the synaptic processing overflows on the third timestep, causing it to start delayed for the Post computational unit. Some platforms allow this case to happen, performing soft real-time simulations, and therefore allowing to overrun timesteps and then recover for the lost time in future timer periods, where the load is reduced. This however violates the hard real-time requirements, which mandate to simulate each individual timestep in the corresponding amount

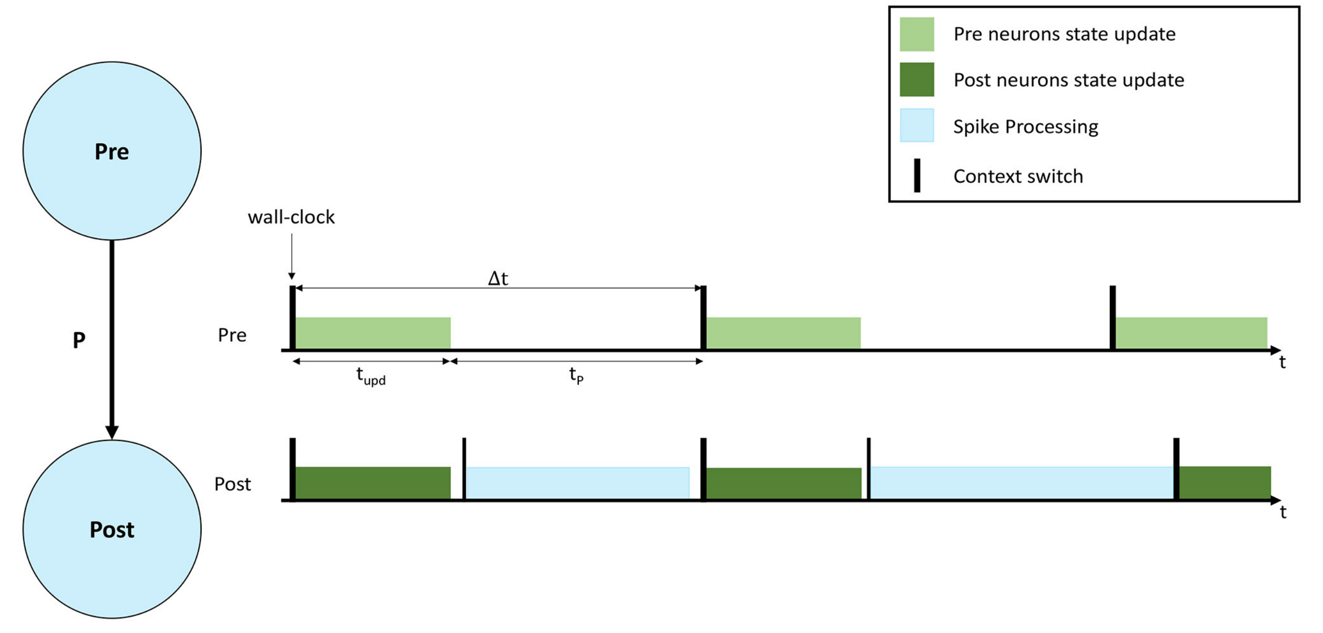


FIGURE 1 | Representation of neural processing. The schematic of a SNN composed of 2 Populations (Pre and Post) with connectivity *P* is shown on the **left**. On the **right** the interaction of simulation events for 3 simulation timesteps, with real-time requirements violation is presented. The green bars show the neural state update, the blue bars the synaptic input processing.

of wall-clock time: i.e., each 0.1 ms of biological time is completed in 0.1 ms.

A reduction of the size of the green bar (neural state update) can be achieved by reducing the number of neurons per computational unit. However, this operation has the effect of requiring additional hardware, since the network becomes more distributed and adds burdens to the communication fabric, increasing the number of destinations for the generated spikes. The time taken to process spikes, as indicated by the length of the blue bar in **Figure 1**, is a function of the number of postsynaptic neurons simulated per unit. When the number of neurons simulated per unit increases, each spike can potentially target more postsynaptic neurons, hence requiring more processing time. While the fan-in to each postsynaptic neuron is independent of the number of neurons simulated, the fan-out of each arriving spike is proportional to the number of available target neurons (defined by the number of neurons simulated per core). Therefore, when this number is reduced, the total available target neurons are reduced, meaning the cost of processing a spike is amortized over fewer individual connections. This reduction in efficiency is a significant problem, as spike processing tends to dominate computation in biologically-representative SNN simulations (Schmidt et al., 2018; Casali et al., 2019).

A more efficient partitioning strategy (Knight and Furber, 2016; Rhodes et al., 2019), demonstrated that it is possible to separate the two phases (neural state update and spike processing) onto separate computational units. This enables simulations with higher numbers of neurons per unit, together with higher efficiency for the synaptic input processing. This approach however still shows some limitations in dealing with very sparse connectivity patterns, as the number of target synapses per spike is still limited by the amount of neurons that can be simulated on a single computational unit. Section 3 presents a novel parallelization approach which overcomes this limitation, maximizing the number of postsynaptic receptors and improving spike processing performance.

3. MATERIALS AND METHODS

3.1. The SpiNNaker System

SpiNNaker is a Globally Asynchronous Locally Synchronous (GALS) many-core digital Neuromorphic platform, specifically designed to simulate SNNs in real time (Plana et al., 2011; Furber et al., 2014). From a hardware perspective, its main building block is the SpiNNaker chip, which contains 18 ARM968 cores (ARM, 2006), each having two separate Tightly Coupled Memories (TCMs), to store local data and simulation code, respectively. Additionally the chip includes a 32 KB shared memory (SysRAM), a 128 MB off-chip shared memory (SDRAM) and a tree-based routing infrastructure which allows direct packet-based communication with 6 other neighboring chips. Each on-chip router can be used as an intermediate hop to forward packets to other chips (Furber et al., 2013; Painkras et al., 2013; Mavaridas et al., 2015). For fault tolerance purposes, the available cores per chip are 17. Access to the shared memories can be performed through bridge or Direct Memory Access (DMA). Bridge access is slow (> 100 ns/word), while a DMA controller provides more efficient bulk transfers (≈ 10 ns/word) up to 64 KB per request, with DMA requests broken down into bursts 128

B wide. Access to the memory controller is however limited to a single channel. Simultaneous attempts to access shared memory give rise to a phenomenon called contention, where a single requesting processor is given access to the memory controller and the others are queued (Painkras et al., 2013; Sharp and Furber, 2013; Rhodes et al., 2018).

SNNs models are simulated through dedicated software (Rhodes et al., 2018; Rowley et al., 2019), with each processor simulating a predefined number of neurons, each implemented through mathematical models governing their neural dynamics. All the available processors (excluding two service cores Rowley et al., 2019, used for system purposes) perform the simulation. This consists in updating the neural state of the implemented neurons in sequential fashion, generating postsynaptic action potentials where necessary, receiving incoming spikes and extracting the synaptic events from incoming packets. SNN simulations on SpiNNaker follow an event-driven approach (Sharp et al., 2011), where cores remain in a low-power state, until an event triggers a processing callback. Periodic timer events are used to advance the simulation time through discrete fixed-length timesteps, while asynchronous events signal the reception of a spike and trigger synaptic processing (Rhodes et al., 2018). Timesteps allow for discretization of continuous time models and, provided the timestep resolution is high enough (commonly 0.1 or 1 ms), allow modeling of neuron state updates *via* exponential integration (Rotter and Diesmann, 1999), calculating the dynamics timestep by timestep.

The spike processing activity spans through most of the simulation timestep and, in case of large networks (Potjans and Diesmann, 2012; Schmidt et al., 2018; Casali et al., 2019), the number of received spike events can cause the neural state update to be preempted and delayed beyond the boundaries of the simulation timesteps (van Albada et al., 2018; Bogdan et al., 2021), resulting in non real-time performance. Real-time performance means that the simulation time of a network matches the modeling time of the network itself, therefore 1 s of activity needs to be simulated in 1 s for it to be in biological real-time.

On SpiNNaker, spikes are delivered through multicast packets in the Address Event Representation (AER) format (Mead, 1989), therefore only containing information about the sender. All synaptic information for a given presynaptic spike (i.e., number of postsynaptic connections, weights and delays) is stored on the postsynaptic side in the SDRAM shared memory. This reduces the amount of information that is transmitted over the communication network, by only specifying the sender. Therefore, upon the reception of a spike packet, each core performs a DMA request to retrieve the associated synaptic data (Rhodes et al., 2018). This information is stored as a sparse synaptic matrix using the compressed-row format, row-indexed by the presynaptic neuron ID. Postsynaptic cores therefore, upon the reception of a spike have a unique identifier of the sender available (given by AER spike packets), and use this as an index to locate the correct synaptic row inside the matrix. By storing the synaptic matrices in the SDRAM memory it is possible to simulate SNNs where neurons have much larger individual fan-ins (a common aspect of biologically-representative SNNs). This overcomes the limitations set by reduced local memory typical of Neuromorphic platforms. This solution also allows simulations of plastic networks, as opposed to the procedural approach (Knight and Nowotny, 2021), and it is more suited to platforms where the memory access is faster than generating pseudo-random values, such as Neuromorphic hardware. This however comes with the penalty of retrieving synaptic rows every time a spike is received, and, in case of plastic networks, a write-back operation for the updated weights is required.

3.2. Homogeneous Parallelization

SNNs on SpiNNaker are commonly partitioned following a Homogeneous parallelization approach (Rhodes et al., 2018; Rowley et al., 2019), where each core simulates a subset of a Population, as described in Section 1. An example of the synaptic matrix representation under the Homogeneous parallelization approach is shown in **Figure 2**. Here, we show a network composed of 2 populations having 12 neurons each, connected with 20% probability (represented on the left). The full synaptic matrix is displayed (top right), where each row corresponds to a presynaptic neuron and each column to a postsynaptic neuron. Where a connection is formed a weight is added to the respective cell. **Figure 2** shows how synaptic matrices are partitioned and mapped to SpiNNaker cores. The right bottom representation shows 3 cores each with its own sparse representation of the synaptic matrix, assuming a limit of 4 neurons per core. This representation reduces the size of the stored matrix, only including the relevant information.

Despite reducing the required memory to store synaptic matrices, this partitioning approach is inefficient; indeed, for sparse connectivity patterns it generates several empty rows, as seen in **Figure 2**. Each core has access to all the presynaptic rows pertaining to the implemented neurons. This limits the number of neurons that can be simulated per core, resulting in an inefficient allocation. Furthermore, for large networks, simply limiting the number of neurons per core is not sufficient, as the amount of incoming synaptic events requires a processing time larger than the timestep itself (van Albada et al., 2018; Bogdan et al., 2021). Also, by reducing the number of neurons per core, the length of the synaptic rows shrinks (as shown in **Figure 2**). This happens because SNNs typically have low connectivity probabilities, especially for long-range connections, therefore having a small number of postsynaptic neurons per core increases the chance of no connections being made, resulting in empty rows in the synaptic matrix. Empty rows are problematic because they cannot be detected until the core has completed the DMA transfer, resulting in wasted processing cycles retrieving meaningless information from SDRAM.

The cost of retrieving a synaptic row from shared memory is however amortized by the number of postsynaptic neurons implemented on each core, as a single transfer per packet is performed. This means that, by simulating more neurons per core it is possible to reduce the number of accesses to memory. A higher number of neurons per core however requires to process additional information, which might not be possible within the boundaries of the timestep.

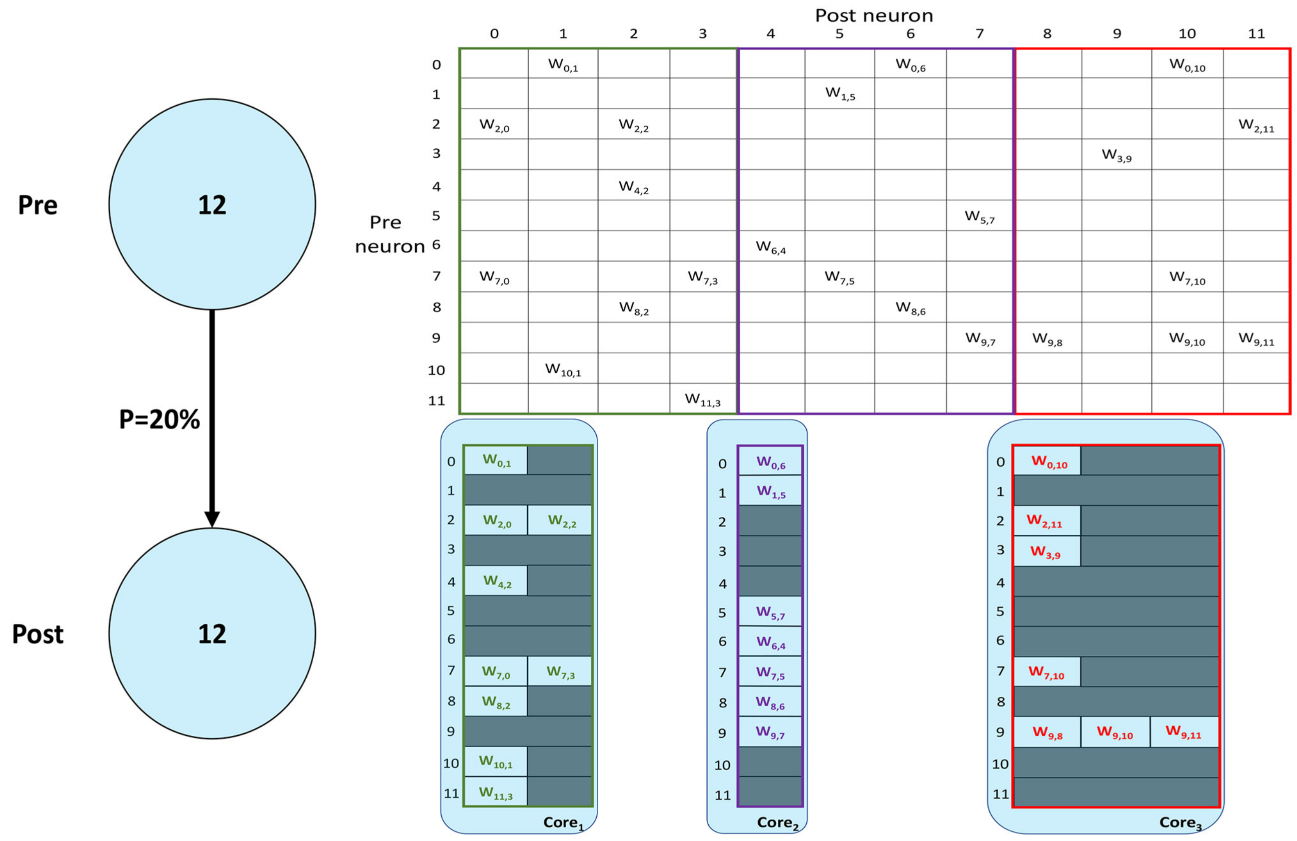

Pre
12
P=20%
Post
12
Post neuron
0 1 2 3 4 5 6 7 8 9 10 11
Pre neuron
0 1 2 3 4 5 6 7 8 9 10 11
$W_{0,1}$ $W_{0,6}$ $W_{0,10}$
$W_{1,5}$
$W_{2,0}$ $W_{2,2}$ $W_{2,11}$
$W_{3,9}$
$W_{4,2}$
$W_{5,7}$
$W_{6,4}$
$W_{7,0}$ $W_{7,3}$ $W_{7,5}$ $W_{7,10}$
$W_{8,2}$ $W_{8,6}$
$W_{9,7}$ $W_{9,8}$ $W_{9,10}$ $W_{9,11}$
$W_{10,1}$
$W_{11,3}$
$W_{0,1}$ $W_{2,0}$ $W_{2,2}$ $W_{4,2}$ $W_{7,0}$ $W_{7,3}$ $W_{8,2}$ $W_{10,1}$ $W_{11,3}$
$Core_1$
$W_{0,6}$ $W_{1,5}$ $W_{5,7}$ $W_{6,4}$ $W_{7,5}$ $W_{8,6}$ $W_{9,7}$
$Core_2$
$W_{0,10}$ $W_{2,11}$ $W_{3,9}$ $W_{7,10}$ $W_{9,8}$ $W_{9,10}$ $W_{9,11}$
$Core_3$

FIGURE 2 | Synaptic matrix partitioning under the homogeneous partitioning. The presented matrix comes from an example network composed of 2 populations having 12 neurons each with 20% connectivity (schematic on the left). The full synaptic matrix is shown on top right. The sparse representation partitioned into 3 different cores is shown on bottom right (with colors matching the full synaptic matrix). The partitioning assumes a limit of 4 neurons per core, therefore 3 cores are required.

Synaptic processing throughput is defined as the maximum number of synaptic events that can be processed per timestep, while maintaining real-time performance (Rhodes et al., 2018).

$$E = \left(\frac{t_P - t_{1^{st}} - t_{last}}{t_{spike}} + 2 \right) Pn \tag{1}$$

$$t_P = \Delta t - t_{upd} \tag{2}$$

$$t_{spike} = m_s Pn + c_s \tag{3}$$

This can be evaluated according to Equations (1)–(3), where E represents the number of synaptic events per timestep, t_P indicates the fraction of the timestep available to process synaptic information, and is obtained by subtracting from the timestep duration (Δt) the time required to update the neural state (t_{upd}) of all the neurons simulated on core. The time required to process a single spike is defined by t_{spike}. This value is expressed by Equation (3) and can be broken in a fixed contribution (c_s), which is paid once per spike packet, corresponding to context switches, synaptic row location in the shared memory and transfer time, and a variable contribution (m_s) which corresponds to the cost of processing a single synaptic event. Spike processing on SpiNNaker is handled through a pipelined approach, therefore the cost of processing the first and the last spike in the pipeline are different due to different API calls (Rhodes et al., 2018). These values are indicated by $t_{1^{st}}$ and t_{last}, respectively, and follow the same rule as t_{spike}, but have different values for fixed and variable costs (Rhodes et al., 2018).

The processing time ($t_P - t_{1^{st}} - t_{last}$) is divided by t_{spike} to obtain the processed spikes per timestep. This number is then incremented by 2, to account for $t_{1^{st}}$ and t_{last} previously subtracted. The number of synaptic events that can be processed in a single timestep is, therefore, given by multiplying the number of spikes by the connectivity probability (P), which indicates the number of postsynaptic connections per spike and then by the number of postsynaptic neurons on core (n).

3.3. Heterogeneous Parallelization

The Heterogeneous Programming Model (Rhodes et al., 2019) is a simulation approach which evolved from a previous study on the partitioning of synaptic matrices on SpiNNaker (Knight and Furber, 2016). This approach aimed at improving the placement

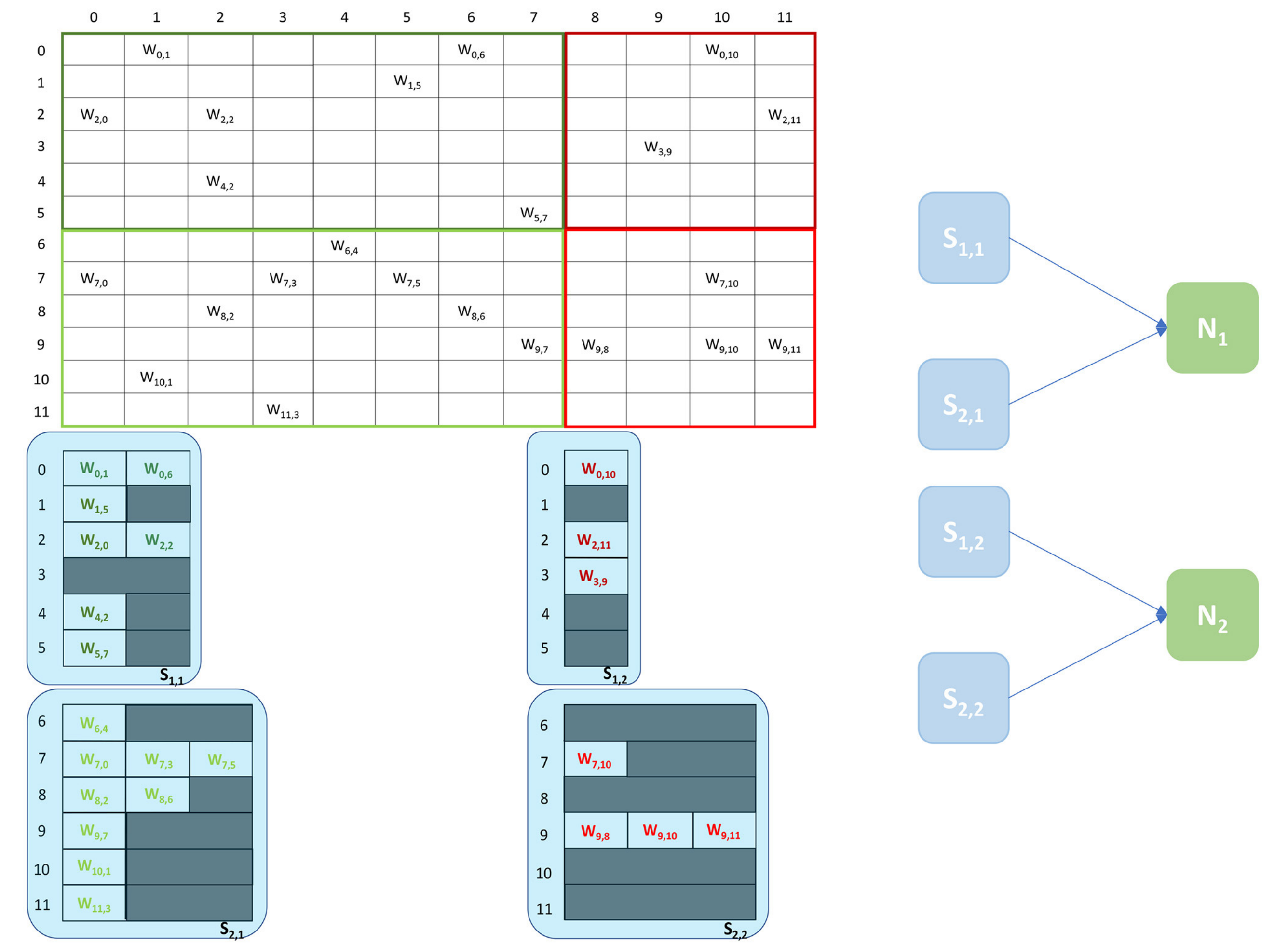


FIGURE 3 | Synaptic matrix partitioning under the Heterogeneous Programming Model. The same matrix presented in **Figure 2** is used. *Synapse* cores allow to partition the matrix by presynaptic index, and to relieve Neuron cores from processing spikes, enabling the possibility of simulating more neurons per core, which in turn allows to increase the length of synaptic rows. A schematic of the ensembles generated by this partitioning is shown on the right, where each *Neuron* core receives inputs from two *Synapse* cores.

of SNNs on SpiNNaker to achieve real-time simulations of complex SNNs (Rhodes et al., 2019). By partitioning the synaptic matrices horizontally (see **Figure 3**), as opposed to the vertical approach (see **Figure 2**), it is possible to maintain longer postsynaptic rows and parallelize processing of incoming spikes. This is achieved by introducing separate cores, called *Synapse* cores, dedicated to the spike processing phase only, each implementing a subset of the synaptic receptors for each postsynaptic neuron (see **Figure 3**). The postsynaptic neurons are simulated on dedicated *Neuron* cores, having the role of advancing the neural state and generating action potentials only. These cores combine the inputs coming from the connected *Synapse* cores, through shared memory. This partitioning strategy allows simulations of higher numbers of neurons per core, therefore increasing the length of the synaptic rows maintained by the connected *Synapse* cores. This enables simulations of sparser connectivity patterns. Through this approach it is furthermore possible to connect multiple *Synapse* cores to each *Neuron* core, increasing the synaptic event throughput of the overall system (Knight and Furber, 2016; Rhodes et al., 2019). The communication between connected *Synapse* and *Neuron* cores happens *via* the chip-local SDRAM shared memory. Each *Synapse* core writes at the end of each timestep the synaptic contributions (representing partial input currents) coming from the receptors simulated by the core. The *Neuron* core reads all the contributions in a single memory block read and computes the total input currents by adding together the values from different *Synapse* cores.

An example of the partitioning of synaptic matrices under this approach is shown in **Figure 3**. In this example the same synaptic matrix addressed in **Figure 2** is used, however it is now split horizontally by presynaptic neurons. Therefore, one *Synapse* core (S_{11}) receives inputs from the lower 6 presynaptic neurons (dark green in **Figure 2**) and the other *Synapse* core (S_{21})

from the higher 6 (light green in **Figure 2**). This increases the number of neurons per core, as the *Neuron* core's sole task is to update the neural state. In this simple example, each *Neuron* core can therefore now simulate 8 neurons, allowing to double the length of the synaptic rows associated to each *Synapse* core. The remaining 4 neurons are simulated by a separate *Neuron* core which replicates the structure of the other ensemble. The two ensembles are shown in **Figure 3** right. N_1 simulates the lower 8 postsynaptic neurons, N_2 the remaining 4 neurons. Each *Neuron* core receives its inputs from 2 *Synapse* cores. The synaptic labels correspond to the cores depicted on the left.

The number of synaptic events that can be processed in a timestep under this approach per *Synapse* core is expressed by Equations (4)–(7), adapted from Equation (1). For this model, t_p represents the spike processing window, which is obtained by subtracting from the duration of the timestep (Δt) the time required by the *Synapse* cores to write the synaptic contributions to shared memory (t_w), minus the time taken by the postsynaptic *Neuron* core to read the contributions from shared memory (t_r). These last two components represent a fraction of the timestep which is wasted, as during t_w no additional spikes can be processed, and during t_r the *Neuron* core has to wait, as it is retrieving the information necessary to update neuron state. The number of neurons is indicated by n. These are simulated by the *Neuron* core of the ensemble. The spike processing times t_{spike}, $t_{1^{st}}$ and t_{last} follow the same rule presented in Equation (3).

$$E = [\frac{t_p - t_{1^{st}} - t_{last}}{t_{spike}} + 2]Pn \tag{4}$$

$$t_p = \Delta t - t_w - t_r \tag{5}$$

$$t_w = aS_c + b \tag{6}$$

$$t_r = cS_c + d \tag{7}$$

A description of the read and write times is given by Equations (6) and (7) and they depend on the number of involved *Synapse* cores (S_c). This dependency can be easily explained by the increase in size of the memory block containing the synaptic contributions (which size is directly proportional to the number of connected *Synapse* cores) to be read by the *Neuron* core every timestep, and by memory access contention, arising when multiple *Synapse* cores try to write to memory at the end of each timestep simultaneously. The lower case coefficients (a, b, c, and d) are hardware specific values. Previously measured quantities, obtained from experimental analysis on SpiNNaker, are shown in **Table 1**. The value described in Equation (4) represents the number of synaptic events per *Synapse* core. The total number of synaptic events per ensemble is calculated by adding together the values for each *Synapse* core belonging to the ensemble. Compared to the Homogeneous partitioning case, with the same number of postsynaptic neurons, this represents a pseudo-linear increase in the processed events per timestep. A demonstration of this can be seen in **Figure 4**, which shows the number of synaptic events processed by SpiNNaker for a 10% connectivity network, with increasing numbers of *Synapse* cores. Here, the blue line shows the 1 ms case and the green line 0.1 ms. For the latter it is not possible to include more than 8 *Synapse* cores per ensemble, as the synaptic contribution reading time from the *Neuron* core's perspective becomes predominant, therefore preventing real-time execution.

TABLE 1 | Reading and writing time coefficients for the Heterogeneous and Multi-target partitioning measured on SpiNNaker.

SpiNNaker reading and writing time coefficients		
Coefficient	**Heterogeneous partitioning value**	**Multi-target partitioning value**
a	0.4	0.9
b	4	0.1
c	0.3	0.6
d	3.9	1.3
e	-	1.2
f	-	0.4
g	-	0.2

Column 2 refers to Equations (6) and (7). Column 3 to Equations (10) and (11).

The Heterogeneous Programming model can achieve impressive performance improvements, however it also presents limitations, as seen in the example shown in **Figure 3**. The length of the synaptic rows is still not optimal, requiring two additional (or more, according to the presynaptic partitioning) *Synapse* cores (S_{12} and S_{22}), to simulate the last four columns, resulting in additional resources being allocated and a sub-optimal partitioning of the matrices. Furthermore the number of synaptic events that can be processed for 0.1 ms timestep simulations is limited to the throughput of 8 *Synapse* cores, which does not allow to fully exploit the available parallelism.

3.4. Multi-Target *Synapse* Cores

Here, we present a novel parallelization approach enabling more efficient use of the available system resources, to address peak synaptic throughput performance and increased sparsity in synaptic connections.

This new approach, termed *Multi-target Partitioning*, extends the concept of *Synapse* cores introduced in Section 3.3, by assigning multiple *Neuron* core targets. Therefore, each neural ensemble will have multiple *Synapse* cores implementing the postsynaptic receptors of multiple *Neuron* cores, instead of matching the neurons of a single *Neuron* core. This technique improves partitioning of the synaptic matrices, by allowing longer rows. This therefore reduces the chance of empty rows for very sparse networks, and, at the same time, allows to amortize the fixed cost of processing a spike (i.e., preprocessing, context switches, and DMA cost) over a larger number of synapses.

An example of the Multi-target partitioning of synaptic matrices is shown in **Figure 5**. The *Synapse* cores now span over a much larger synaptic matrix, covering the entire rows

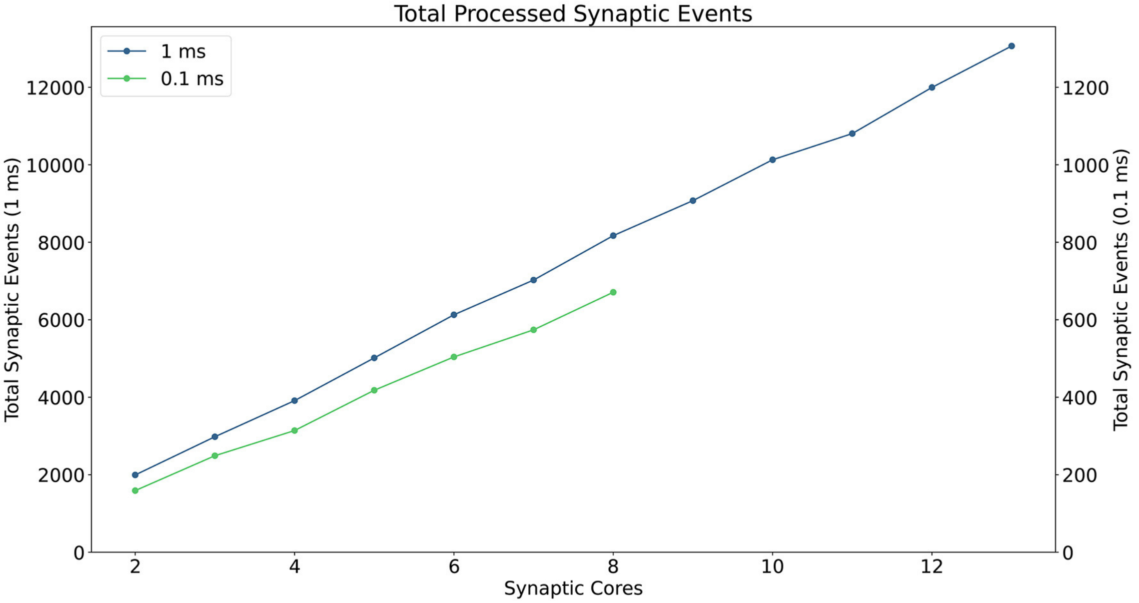


FIGURE 4 | Processed synaptic events per timestep at 10% connectivity with increasing *Synapse* cores per ensemble. The blue line shows the 1 ms case (values reported on the left axis), the green line the 0.1 ms case (values reported on the right axis). The number of *Synapse* cores is limited to 8 for the latter, because of timing constraints due to the synaptic contributions reads.

in the example. The partitioning is performed presynaptically (horizontally), similarly to the Heterogeneous Model. However for the Multi-target partitioning, each *Synapse* core can target multiple postsynaptic *Neuron* cores, implementing all receptors for all target *Neuron* cores (effectively reducing the vertical partitioning). This approach allows to save resources (2 *Synapse* cores in the case of the example in **Figure 5**) and further reduces the chance of having empty rows for a given probability of connection. The number of synaptic events that can be processed per timestep is now modeled by Equations (8)–(13).

$$E = [\frac{t_P - t_{1^{st}} - t_{last}}{t_{spike}} + 2]PN \tag{8}$$

$$t_p = \Delta t - t_w - t_r \tag{9}$$

$$t_w = aS_c - bN_c + cN_cS_c + d \tag{10}$$

$$t_r = eS_c + fN_c - g \tag{11}$$

$$N = nN_c \tag{12}$$

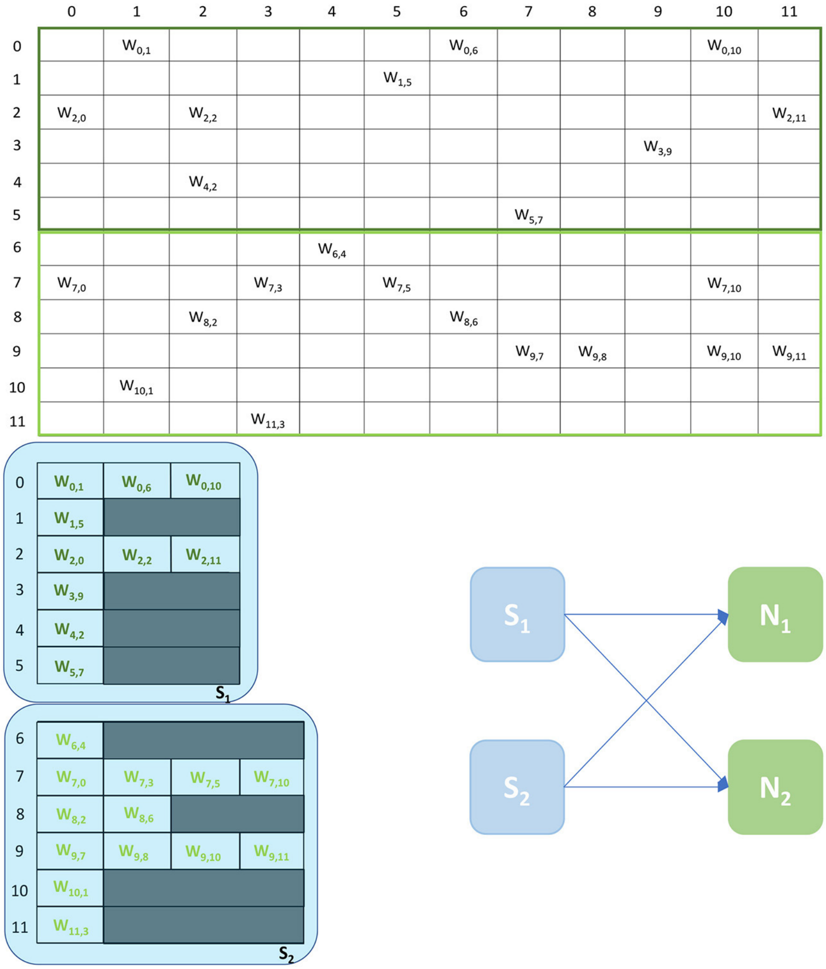
$$t_{spike} = m_s PN + c_s \tag{13}$$

The components are similar to the Heterogeneous model case, however N depends now on the number of *Neuron* cores connected to each *Synapse* core, and is obtained by multiplying the number of neurons per core (n) by the number of connected *Neuron* cores (N_c). This reflects also on the spike processing times, as shown in Equation (13), where the variable cost is now multiplied by the total number of neurons targeted by the spike, therefore by the *Synapse* core. The reading (t_r) and writing (t_w) times now depend on the structure of the ensemble, as both contention and size of the transfer play a key role. The lower case coefficients (a to g) are hardware specific values, which therefore change according to the chosen platform. **Table 1** reports values for the SpiNNaker platform obtained by profiling execution.

3.4.1. Neuromorphic Implementation

A schematic of the core interactions and memory structures for the Multi-target partitioning implementation is shown in **Figure 6**. The ensemble demonstrates 2 *Synapse* cores each targeting 3 *Neuron* cores.

In the Multi-target approach the synaptic matrices are partitioned according to the synaptic view of the ensemble, meaning that the postsynaptic neurons simulated by multiple *Neuron* cores can now be included in a single matrix. Therefore, *Synapse* cores allocate the shared memory region for the current timestep synaptic contributions (blue and green blocks in SysRAM and SDRAM memories in **Figure 6**). This is opposed to the Heterogeneous Model, where the *Neuron* core of the ensemble sets the shared regions. This allows to perform a single block write per *Synapse* core per timestep, instead of fragmenting into multiple regions. This choice is motivated by architectural features, as the read throughput is higher than the write for the SpiNNaker chip (Painkras et al., 2013), therefore it is preferred to have fewer writes per timestep. *Neuron* cores retrieve the address



	0	1	2	3	4	5	6	7	8	9	10	11
0		$W_{0,1}$					$W_{0,6}$				$W_{0,10}$	
1						$W_{1,5}$						
2	$W_{2,0}$		$W_{2,2}$									$W_{2,11}$
3										$W_{3,9}$		
4			$W_{4,2}$									
5								$W_{5,7}$				
6					$W_{6,4}$							
7	$W_{7,0}$			$W_{7,3}$		$W_{7,5}$					$W_{7,10}$	
8			$W_{8,2}$				$W_{8,6}$					
9								$W_{9,7}$	$W_{9,8}$		$W_{9,10}$	$W_{9,11}$
10		$W_{10,1}$										
11				$W_{11,3}$								

0	$W_{0,1}$	$W_{0,6}$	$W_{0,10}$
1	$W_{1,5}$		
2	$W_{2,0}$	$W_{2,2}$	$W_{2,11}$
3	$W_{3,9}$		
4	$W_{4,2}$		
5	$W_{5,7}$		

6	$W_{6,4}$			
7	$W_{7,0}$	$W_{7,3}$	$W_{7,5}$	$W_{7,10}$
8	$W_{8,2}$	$W_{8,6}$		
9	$W_{9,7}$	$W_{9,8}$	$W_{9,10}$	$W_{9,11}$
10	$W_{10,1}$			
11	$W_{11,3}$			

FIGURE 5 | Synaptic matrix partitioning for the Multi-target approach. The used network is the same shown in **Figures 2**, **3**. Here, *Synapse* cores have much longer synaptic rows, further reducing the risk of empty rows, therefore fewer resources are required. The generated ensemble is shown on the right.

of each memory block of each connected *Synapse* core, and compute the offset according to the indices of the implemented neurons (blue and green sub-blocks in **Figure 6**). This results in one write per *Synapse* core and multiple reads per *Neuron* core, according to the number of afferent *Synapse* cores.

During simulation initialization, *Synapse* core 1 allocates the blue region in SDRAM in **Figure 6**, which is large enough to store the contributions to postsynaptic neurons of all 3 *Neuron* cores. *Synapse* core 2 allocates the green region, having the same characteristics. The *Neuron* cores then retrieve the addresses of both memory regions and compute the starting address of their sub-regions according to the implemented postsynaptic neurons. Therefore, *Neuron* core 1 has the N1 sub-region from both the green and blue region, *Neuron* core 2 has the N2 sub-region and *Neuron* core 3 has N3. During a simulation timestep, when a spike is received, *Synapse* cores act the same way as the Heterogeneous Model (Rhodes et al., 2019). They extract the synaptic row address for the received spike, retrieve the correct row from the synaptic matrix and then add the connection weight to the synaptic input buffer (shown as circles in **Figure 6** in blue for *Synapse* core 1 and in green for *Synapse* core 2), according to delay and postsynaptic index. Synaptic input buffers (Morrison et al., 2005; Rhodes et al., 2018) are structures employed to handle synaptic delays, and store the input currents for postsynaptic neurons. These are typically two-dimensional data structures, indexed by postsynaptic neuron ID and delay. When a spike is received on a postsynaptic core, for each postsynaptic neuron, the correct buffer slot is located, according to the delay and destination of the spike. Then, the weight of the connection is added to that buffer slot.At the end of the timestep, the slots of the synaptic input buffers representing the next timestep's synaptic input are written to shared memory (these include all the slots having delay 1 timestep). Therefore, *Synapse* core 1 writes N1, N2 and N3 sub-regions of the blue region, which will contain one slot per postsynaptic neuron having 1 timestep delay, and *Synapse* core 2 does the same for the green region. Different *Synapse* cores write to different memories (either SDRAM or SysRAM), to reduce

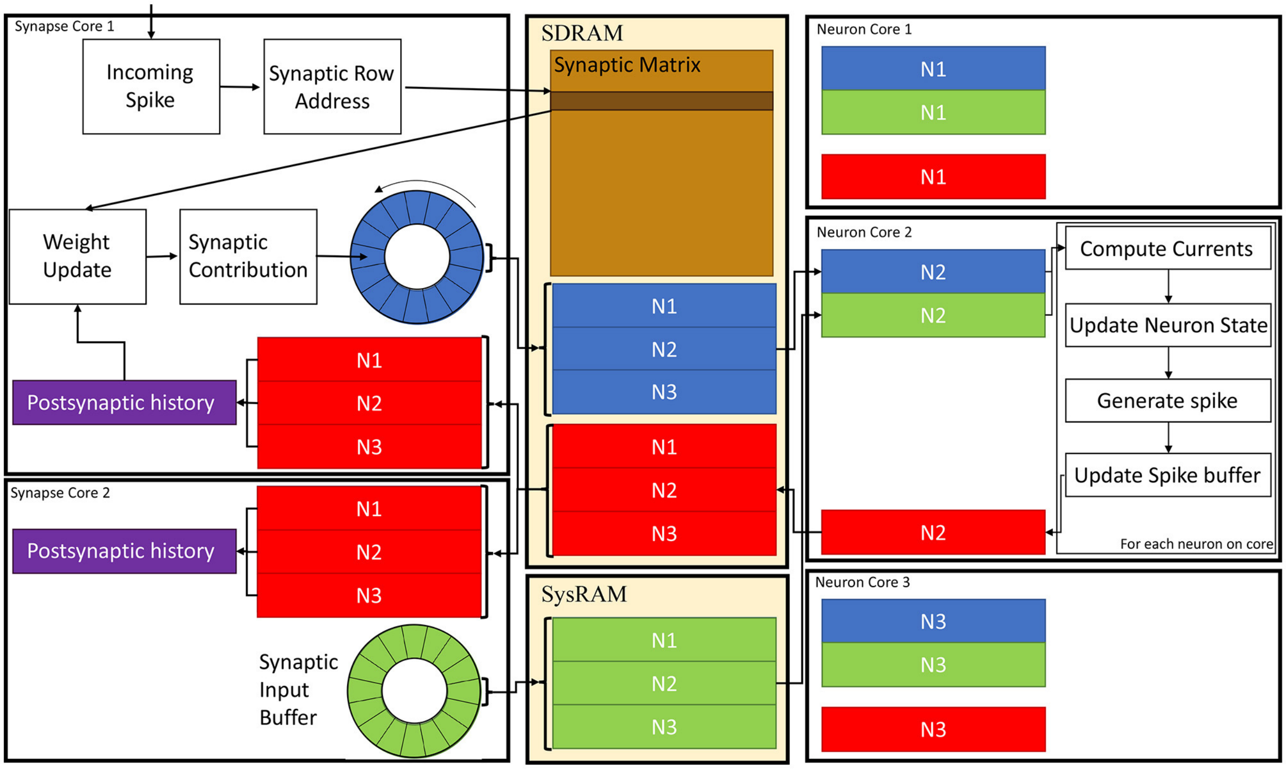


FIGURE 6 | *Synapse* and *Neuron* cores memory interaction for the Multi-target partitioning. 2 *Synapse* cores targeting 3 *Neuron* cores are shown with all the steps from spike reception to neural state update. Communication between cores belonging to the same ensemble happens *via* the two shared memories (SDRAM and SysRAM), through the represented data structures.

contention on the SDRAM memory controller. The destination is decided according to the physical core ID,evenly spreading the contributions between the two memories. Both memories are part of the system memory map, therefore the allocation can be performed simply by specifying the correct memory heap, and the address retrieval is transparent to this operation.

At the beginning of the subsequent timestep, all *Neuron* cores perform reads of the sub-regions. Upon completion, the input currents for each postsynaptic neuron are calculated by adding together all contributions from the *Synapse* cores for the specific neuron. The synaptic currents are then used to update the neuron state, according to the implemented neuron model and, if the model mandates it, a spike is generated.

The time required to read the memory regions is a crucial design parameter, because it sets a boundary on when the *Neuron* core can generate the first spike. In fact until all the contributions are read, the *Neuron* cores cannot start processing the neural state updates. This reflects on when postsynaptic *Synapse* cores can start receiving spikes, effectively reducing the spike processing window. It is therefore of paramount importance to reduce this reading interval as much as possible. In order to address this issue, *Neuron* cores are instructed to perform out-of-order read operations of the sub-regions. This means that, based on the *Neuron* core ID, the first read region will be either from SysRAM or SDRAM. This effectively halves the *Neuron* cores accessing the same memory at the same time, by explicitly instructing half of them to first read from SDRAM and half of them from SysRAM. After each read is completed, each *Neuron* core sends the subsequent request to the other memory.

3.5. Plasticity

The Heterogeneous model and the Multi-target partitioning can be extended to include simulations of plastic SNNs. For plastic networks, the time required to process synaptic events is higher compared to the static case, as a weight update phase is needed. Therefore, simulations of plastic SNNs would also benefit from reduced processing time per synaptic event. Here, we present the steps to extend the plasticity framework, in order to include the Multi-target partitioning. This framework is independent from the implemented plasticity rule, and the synaptic update is fully handled by *Synapse* cores, which implement the chosen rule for a simulation. **Figure 6** also shows the memory structures necessary for the implementation of STDP, as well as the weight update framework.

The plasticity framework adopted by the SpiNNaker toolchain performs synaptic weight updates upon receiving a spike (Galluppi et al., 2015). This minimizes the accesses to shared memory, as synaptic rows are commonly retrieved whenever a spike is received. After a row is stored in local memory, before adding the weight contribution to the correct synaptic input buffer, each weight is updated according to the implemented plasticity rule. STDP rules commonly require information about postsynaptic firing activity (Morrison et al., 2008). This information is stored into a postsynaptic buffer, locally maintained by the *Neuron* cores, which contains one slot per postsynaptic neuron, and is updated every time a neuron fires. The introduction of plasticity into the Multi-target approach adds complexity, since the *Neuron* cores need to communicate back to the *Synapse* cores which neurons have spiked during the timestep, to correctly update the synaptic weights. This operation is again performed through shared memory. All *Synapse* cores share the same postsynaptic region (red region in **Figure 6**), therefore this area is allocated into SDRAM by the *Synapse* core of the ensemble having the lowest index, and the address is retrieved by all the other *Synapse* cores. The *Neuron* cores get the address in the same way as the synaptic contributions, and will use the same offset to get access to their specific sub-regions.

During each timestep, after all the neurons on core have been updated, the postsynaptic buffer (red sub-blocks in the *Neuron* cores), which contains information on whether each neuron has spiked or not, is written to SDRAM by each *Neuron* core. The *Synapse* cores can read this region and update the postsynaptic history (purple buffers in **Figure 6**) for each receptor. In order to keep the memory operations short, the postsynaptic buffers are saved as binary flags, indicating whether each neuron has spiked or not. The update of the postsynaptic history depends on the simulated plasticity rule, which is implemented on the *Synapse* cores (as only these have visibility of the timing of incoming spikes). Only after this read operation is completed is it possible to update the weights and to process the incoming spikes. Therefore, the received spikes before this operation are buffered and ready to be processed when the read is completed. The *Synapse* core read is scheduled to happen after a fixed amount of time (for a given configuration), as the *Neuron* cores require a fixed amount of time to update the neural state and write back the postsynaptic buffers.

4. RESULTS

The performance of the Multi-target partitioning approach presented in Section 3.4 is now evaluated from the perspectives of: system memory (Section 4.1), peak synaptic event throughput (Section 4.2) and the effect of connection sparsity (Section 4.3).

4.1. Memory Access

4.1.1. Experiment Description

This first experiment measures the impact of writing and reading the synaptic contributions between *Synapse* and *Neuron* cores under the new ensembles scheme, showing timings for each possible combination of *Neuron* and *Synapse* cores on a chip. Each *Neuron* core is set to simulate 64 Leaky Integrate-and-Fire (Gerstner and Kistler, 2002) neurons, and afferent *Synapse* cores handle their synaptic receptors. In order to isolate the transfer times, the values are sampled in the context of a neural simulation in absence of spike packets. Therefore, the standard neural state is updated, but the spike processing pipeline and the spike generation phases are turned off. This prevents neural processing from increasing contention, while maintaining the characteristics required by SNN simulations. Each test simulates 100 timesteps, and is repeated 10 times to ensure consistency. For each arrangement timings are presented for both the SysRAM + SDRAM case, and the SDRAM only case. The results are presented in form of heatmaps, where the horizontal axis shows the number of employed *Synapse* cores, while the vertical axis the *Neuron* cores. All the *Synapse* cores for each case are connected to all the *Neuron* cores of the same case. The reported values are the worst case transfer times obtained by this test. These values are fundamental to estimate the impact of memory access time on the approach. Through these measurements it is possible to correctly allocate timings which allow the processors to initiate DMA transfers in time to maintain real-time performance.

4.1.2. Reading Times

Reading time measurements are shown in **Figure 7** (all times measured in μs). The plot on the left presents values using both the shared memories available to the SpiNNaker chips (SysRAM and SDRAM), while the plot on the right contains timings relative to the SDRAM use only. All the purple boxes without a number are combinations of cores not allowed by the machine. The case with a single *Synapse* core has been omitted, since the transfer was completed quickly enough not to impact performance. The timings have been extracted in the context of a neural application simulating 64 neurons per core. Each synaptic weight is stored on a 16 bit (2 B) integer, meaning the contributions of a *Synapse* core targeting a single *Neuron* core amount to 64×2 B $=$ 128 B (each DMA read has this fixed length).

By increasing the number of *Synapse* cores (moving from left to right on the horizontal axis), the number of reads per timestep per *Neuron* core increases. Reads are scheduled by the *Neuron* cores at the beginning of the timestep and performed sequentially, since there is a single DMA engine. As expected, for both the plots, the case with a single *Neuron* core (first line), shows linearly increasing reading times. The use of two separate memories does not influence this aspect, as one read at a time is performed. However it is observed that times in the dual memories plot are slightly lower. This is due to half of the *Synapse* cores contributions being stored into SysRAM which has a lower access time than SDRAM, therefore providing faster access. By increasing the number of *Neuron* cores (from top to bottom on the vertical axis), the contention increases, as multiple *Neuron* cores try to access shared memory to retrieve their synaptic contributions simultaneously. This case demonstrates the benefits of having two different memories in use with separate access. The SysRAM + SDRAM case indeed performs generally better than the single memory case allowing a gain up to 4 μs. There are, however, some isolated allocations where the single memory case performs better. This is probably due to a bad allocation of the cores on the chip, which results in a slower access

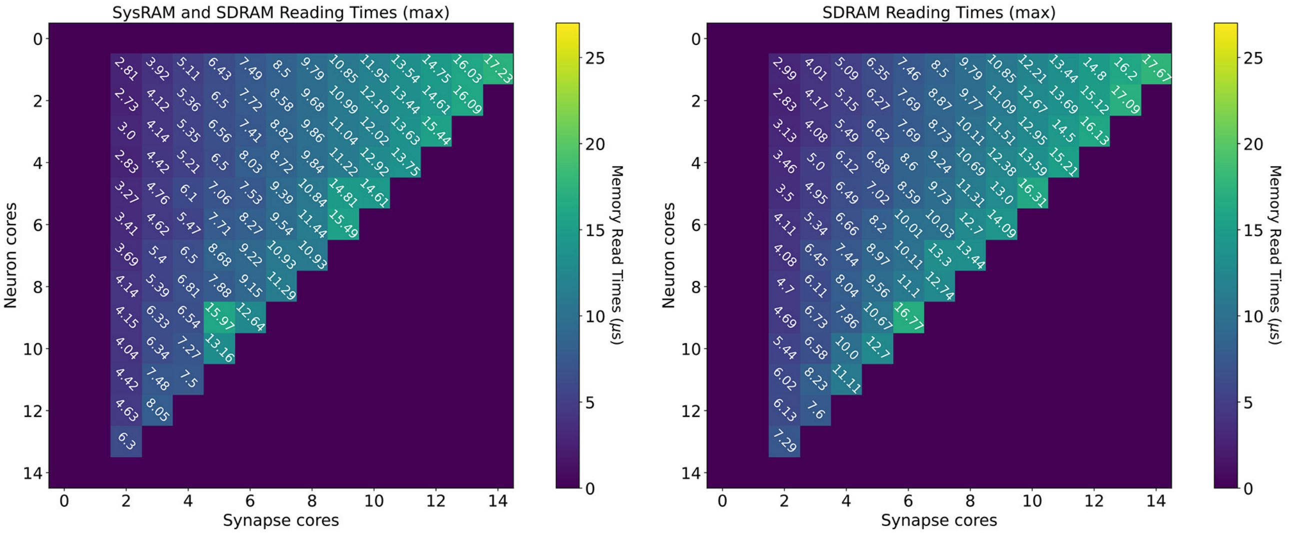


FIGURE 7 | Memory heatmaps showing worst case DMA reading timings for increasing *Synapse* and *Neuron* cores. *Synapse* cores are represented on the horizontal axis, target *Neuron* cores on the vertical. All the measured times are in μs. The two plots represent the dual memory **(left)** and the SDRAM only case **(right)**. Purple blocks represent configurations not allowed by the machine.

to memory. Core allocation affects the memory access time, as to grant fairness, access to memory is regulated by a binary tree with arbiters at every junction point. The cores are on the leaves of the tree. A situation where the allocation of *Synapse* cores is unbalanced can cause higher contention between memory requests, as requests coming from more populated branches of the tree need to be filtered by multiple arbitration steps. This results in additional delays, which increase the total memory transfer time from the cores' perspective. Cores are assigned by the SpiNNaker toolchain during the placement phase. The values reported here represent the measured worst case reading times, therefore they are likely to represent the worst allocation of cores.

The worst case for both the experiments happens with 14 *Synapse* cores, which represents the placement with the highest number of sequential reads, performed by a single *Neuron* core. Furthermore, by keeping the number of *Synapse* cores constant, and increasing the *Neuron* cores, the transfer time becomes higher, as the reading contention increases. This reduces the portion of the timestep available for neural processing. It is therefore of paramount importance to understand the requirement of the SNN to be simulated, in order to determine the appropriate number of *Synapse* cores to allocate per *Neuron* core. It is noted that the values shown here represent the worst case scenario, thus presenting the highest recorded reading times. A more detailed analysis including best and average cases, is provided in the **Supplementary Material**.

The worst case analysis is important from a reading perspective to understand when the *Neuron* cores will start to fire, as the read phase must precede the neural state update and therefore *Neuron* cores must wait until this phase is completed before processing the neuron state update.

4.1.3. Writing Times

The measurements for the writing times are shown in **Figure 8**: the left plot shows the dual-memory case, while the right plot contains the SDRAM only case. Times are measured in μs, and each square represents a single write. Increasing *Synapse* cores are displayed horizontally, while increasing *Neuron* cores on the vertical axis. By increasing the number of *Synapse* cores, the contention grows, as multiple cores attempt to write to shared memory simultaneously. By increasing the number of *Neuron* cores however, the size of each write becomes larger. This is because each *Synapse* core performs one single write per timestep. Therefore, by increasing the number of postsynaptic receptors (connected *Neuron* cores), the number of synaptic contributions to be written grows as well. The size of each write is expressed by Equation (14), where n is the number of neurons per *Neuron* core (64 in this case), w is the size of a contribution (2 B for standard SNNs) and T is the number of target *Neuron* cores for each *Synapse* core. Therefore, in **Figure 8**, T increases vertically from top to bottom.

$$C = nwT \tag{14}$$

Similarly to the read case, the reported times are the worst case measured writing times, and, for some cases, the access time is worse for the dual memory case. This can be due to several factors, as *Synapse* core contributions are partially located in SysRAM and partially in SDRAM. Although SysRAM provides a faster access, it has a slower transfer rate, therefore, for larger transfers, it can result in similar or worse performance compared to SDRAM. This, combined with a bad cores placement, can result in losing the advantages of using SysRAM, negating the faster memory access, due to contention on the memory controller. Average and best case measurements

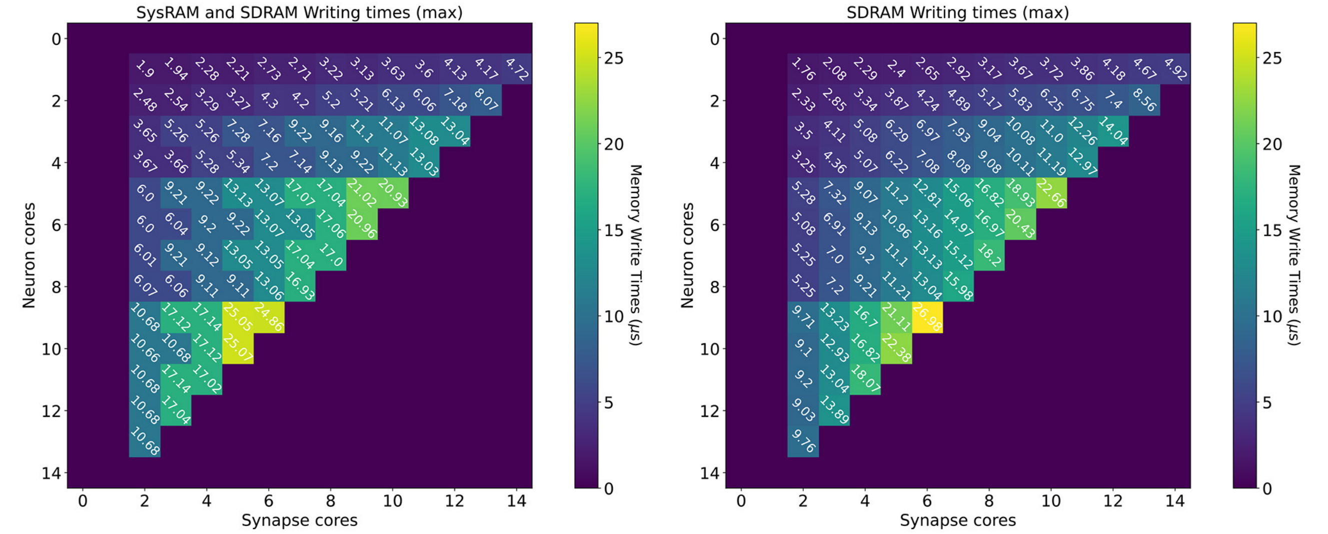


FIGURE 8 | Memory heatmaps showing worst case DMA writing Timings for increasing *Synapse* and *Neuron* cores. *Synapse* cores are represented on the horizontal axis, target *Neuron* cores on the vertical. All the measured times are in μs. The two plots represent the dual memory **(left)** and the SDRAM only case **(right)**. Purple blocks represent configurations not allowed by the machine.

however highlight that this is an isolated case, and show that the dual memory approach is more effective for the arrangements of interest. For more details and analysis, please refer to the **Supplementary Material**.

From a writing perspective, the worst case scenario is useful to instruct *Synapse* cores on when to stop processing incoming spikes and start writing the synaptic contributions to shared memory (in order to meet real-time requirements). The highest recorded writing time is when using SDRAM only with 6 *Synapse* cores targeting 7 *Neuron* cores. This time amounts to 26.98 μs. This does not represent an issue in 1 ms timesteps simulations, but amounts to more than a quarter of the timestep for real-time simulations with 0.1 ms timesteps.

The worst case writing and reading measurements therefore allow to Taylor synaptic contribution writing and reading times to the required number of *Synapse* and *Neuron* cores per ensemble. This avoids overestimations which would further reduce the processing time shown in Equation (9). This analysis shows the importance of balancing the number of *Synapse* and *Neuron* cores according to the application requirements, in order to incur minimal memory access penalties. Network sparsity and firing activity also play a key role in the choice of core allocations, therefore the next sections focus on these aspects.

4.2. Peak Processing Profiling

4.2.1. Experiment Description

The most useful metric when evaluating throughput performance of the Multi-target partitioning is the maximum number of processed synaptic events per timestep. This experiment therefore compares the peak throughput performance for the Multi-target partitioning to previous works. To perform a fair comparison, the same SNN is profiled using the different approaches: Multi-target and Heterogeneous models. The same number of cores is allocated for both configurations, but with different internal connections between *Synapse* cores and target *Neuron* cores. A third configuration is also presented, referred to as *single target expanded*. This consists of a standard Heterogeneous partitioning which maintains the same number of *Neuron* cores as the previous two cases, but allocates the same input *Synapse* cores capacity per *Neuron* core as the Multi-target approach. This last configuration provides a useful comparison, as the number of cores required for the single target Heterogeneous partitioning is adjusted to match the input capability of the Multi-target partitioning. The aim of including these cases is, therefore, twofold: first to compare the Multi-target partitioning to its Heterogeneous counterpart employing the same hardware resources, evaluating the performance difference; second, to show that, to achieve the input processing capability of the Multi-target approach, while using the Heterogeneous partitioning, is necessary to employ a larger number of hardware resources. This is represented by the *single target expanded* case.

A schematic of core allocations for the three approaches is shown in **Figure 9**. The experiments run to evaluate this metric are structured in test cases defined by 2 numbers in the form $[S_c, N_c]$, where S_c is the number of *Synapse* cores and N_c the number of *Neuron* cores – the case shown in **Figure 9** is $[3, 3]$. The Multi-target partitioning is shown on the left, where all the *Synapse* cores are connected to all the *Neuron* cores. The Heterogeneous partitioning is shown on the right, including the two different mappings explored: *single target* and *single target expanded*. The *single target* Heterogeneous partitioning presents 3 *Neuron* cores receiving input from a single *Synapse* core each, showing an input capacity reduced by a third compare to the Multi-target case. The *single target expanded* in the

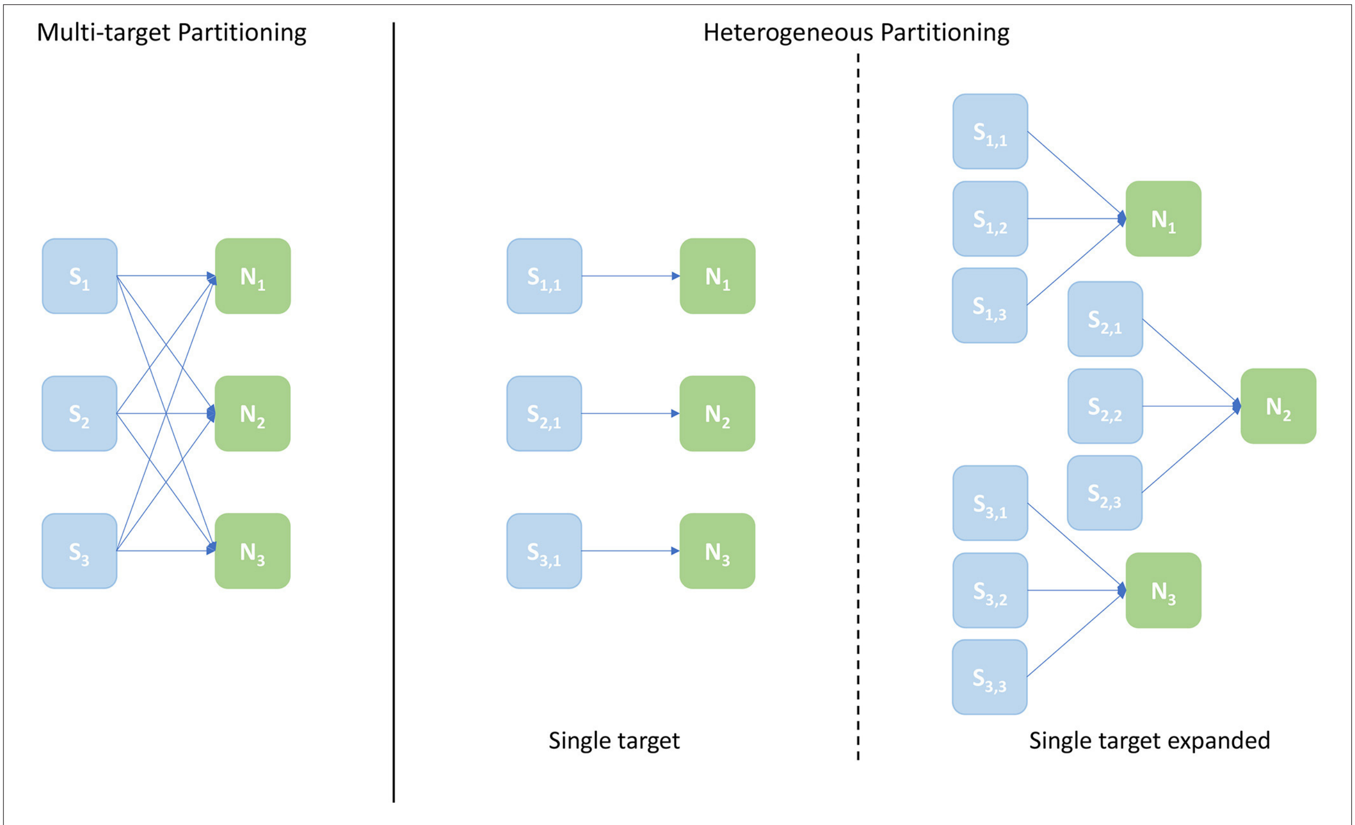


FIGURE 9 | Arrangement of *Synapse* and *Neuron* cores under the explored configurations: Multi-target partitioning **(left)**; Heterogeneous partitioning **(right)**. The example shown demonstrates the [3, 3] test case, with 3 *Synapse* cores and 3 *Neuron* cores. For the Multi-target partitioning configuration, each *Synapse* core targets all *Neuron* cores. Comparison to the Heterogeneous approach is provided by: the Single-target partitioning, where the same overall number of cores are used, but connected one *Synapse* core to each *Neuron* core; and the Single-target expanded partitioning, where the same number of *Neuron* cores is maintained, but each with the same number of *Synapse* cores as implemented in the Multi-target approach.

experiment is therefore comparable with the $[3, 3]$ cases for the two other configurations, however the number of cores allocated is $[9, 3]$. This single-target expanded configuration matches the input capacity per *Neuron* core of the Multi-target partitioning, keeping the same number of neurons and *Neuron* cores (therefore in the presented example each *Neuron* core receives inputs from 3 *Synapse* cores similarly to the Multi-target case, but each *Synapse* core is single target). The intent here is to show that the Multi-target partitioning can reach similar performance compared to this extended configuration, requiring only a fraction of the allocated resources.

The SNN model used for this experiment consists of 2 populations of neurons, configurable with a range of sizes and connectivity (similar to that shown in **Figure 1** left). All the presynaptic neurons are Leaky Integrate-and-Fire (Gerstner and Kistler, 2002) spiking neurons, with current-based exponentially-decaying synapses. Neurons are initialized with the internal voltage above firing threshold to produce spikes in a controlled manner. This approach is adopted to send spikes, instead of using spike sources, as it better represents the interaction between cores when simulating biologically-representative SNNs. This is because spike sources on SpiNNaker generate and send all spike packets together, causing a high firing activity concentrated at the beginning of the timestep, and then they remain silent. Cores implementing Populations (*Neuron* cores in this case) on the other hand, generate spike packets every time a neuron is updated and the model equations require it to spike, therefore distributing the spike packet generation over the timestep.

The size of the presynaptic Population changes according to the number of incoming partitions (number of *Synapse* cores per ensemble) of the postsynaptic Population. These Population sizes have been obtained experimentally, such that the postsynaptic Population receives more spike packets than it can process. This allows saturation of the receivers in order to determine their limits. The number of generated spike packets however needs to be limited, due to limitations set by the SpiNNaker communication infrastructure (Mavaridas et al., 2015). An excessive firing activity would cause higher congestion at the routing level, causing spike packets to be delivered late. This would result in lower processed synaptic events, compared to the real peak throughput, due to late arrivals. More details about Population sizes can be found in the **Supplementary Material**. The postsynaptic Population employs the same type of neurons as the presynaptic Population, and has variable size between 64 to 896 neurons (corresponding to 1–14 *Neuron* cores, respectively). Different connectivity patterns have been tested to demonstrate the robustness of the approach. Here, the 1% connectivity case is shown, as it is commonly found in biologically-representative SNNs (Potjans and Diesmann, 2012; Schmidt et al., 2018). For 0.1, 5, and 10% connectivities, please refer to the **Supplementary Material**.

The same experiment was run both with 1 and 0.1 ms timesteps. The importance of showing results with both timestep resolutions is given by the requirement of biologically-representative SNNs to be modeled using tighter timing resolutions, to better capture their dynamics. Real-time 0.1 ms timestep simulations, indeed, present additional challenges due to tighter timing constraints and a reduced spike processing window (as demonstrated in Section 4.1), which is not amortized by a smaller number of neurons or synaptic receptors.

The simulated network in this experiment was the same for both 1 and 0.1 ms timestep cases, with the exception of the presynaptic Population size, which was scaled down of a factor ≈ 10× (see **Supplementary Material** for exact values). The same experiment was run both for plastic and static networks and the results are presented separately. In order to provide a fair comparison the number of neurons per core is kept fixed at 64. For additional cases, please refer to the **Supplementary Material**.

4.2.2. Static Networks

Figure 10 shows peak synaptic event throughput in the form of barcharts for the experiment with static connections, for both 0.1 ms (left) and 1 ms (right) timesteps. The connectivity between the two Populations is randomly generated with a probability of a connection between a pre- and postsynaptic neuron set to 1%. The *Multi-target* case is represented by the blue bars, while the *single target* with the same amount of cores by the green bars. The purple bars represent the *single target expanded* case. Finally, the yellow bars show the processed synaptic events using the Homogeneous partitioning with the same network and neurons per core.

Both the single target cases (green and purple) make use of the Heterogeneous model. The number of employed cores for each test case is indicated on the horizontal axes. The lower axis refers to the *Multi-target* (blue) and the *single target* (green). The upper axis shows values for the *single target expanded* (purple). The chosen configurations of cores allow direct comparison of the approaches. The left number in each tuple represents the *Synapse* cores of that test case, the right number the *Neuron* cores (as shown by the example presented in **Figure 9**). In the case of the Multi-target partitioning, all the *Synapse* cores of the ensemble target all the *Neuron* cores. For the single target cases the number of *Synapse* cores per *Neuron* core is obtained dividing the first number by the second. The blue and green bars are on the same axis because they employ the same number of cores, the difference between these two cases is in the connections between cores. This demonstrates that it is possible to improve the peak processing by rearranging the available units. The purple cases use the same number of *Synapse* cores per ensemble of the green tests, however, in this case each *Synapse* core has one single target (therefore there is a single *Neuron* core per ensemble). This replicates the input capabilities of the Multi-target partitioning per ensemble, but requires a considerably higher amounts of cores compared to the Multi-target case, resulting in the worst case of 56 total cores compared to 14 (8^{th} test case).

In all the cases the *Multi-target* approach (blue) performs better compared to the *single target* model (green). This is because the Multi-target partitioning performs a more efficient use of the available system resources compared to the Heterogeneous partitioning, allocating a higher input processing capacity to each *Neuron* core.

For the 1 ms timestep experiment the highest synaptic event throughput is given by the [7,7] configuration, where the Multi-target partitioning processes ≈ 9× more synaptic events than the heterogeneous partitioning. The reason why this happens is due to a full exploitation of the source-based partitioning offered by the approach. Each *Synapse* core in the *Multi-target* case receives inputs from one seventh of the presynaptic neurons and targets all the 448 postsynaptic neurons. The *single target* partitioning on the other hand, has each *Synapse* core receiving inputs from all the presynaptic neurons, but targets only 64 neurons. Because the connectivity is very sparse, a reduced input traffic achieves better results.

The *Multi-target* approach performs well also compared to the *single target expanded* (purple), which represents a remarkable result, since the amount of resources in use is much lower, especially in the [7, 7] case. The single target expanded approach employs the same number of *Synapse* cores per ensemble as the Multi-target partitioning, but has a single target per ensemble. Therefore, in the [7, 7] case ([49, 7] for the *single target expanded*), each *Synapse* core receives input from one seventh of the presynaptic neurons and targets 64 postsynaptic neurons only.

The trend is similar for 0.1 ms timesteps, with the *Multi-target* partitioning performing better than the *single target* case. However, with higher numbers of *Synapse* cores targeting higher numbers of *Neuron* cores, performance compared to the *single target expanded* case tends to be lower. This is due to the tight constraints set by the timestep resolution and the fact that memory read and write times for the synaptic contributions do not scale down with the timestep resolution.

This experiment shows that, by efficiently using the Multi-target partitioning, it is possible to achieve comparable results to the *single target expanded* case, but with a fraction of the hardware resources (a quarter in the [7, 7] case). Furthermore, with the same amount of resources it is possible to achieve considerably higher synaptic event throughput.

The general trend for the three approaches, together with the Homogeneous partitioning baseline is compared in **Figure 11**, where the horizontal axis shows the total number of allocated cores, and the vertical axis the processed synaptic events per timestep. The simulations are analogous to those shown in **Figure 10**. Each point in **Figure 11** matches one of the bars (refer to the **Supplementary Material** for a case by case labeled representation of this plot). The *Multi-target* approach shows the best gain, having the steepest increase compared to the other three approaches, performing the best use of the available resources. Additional analysis is performed in the

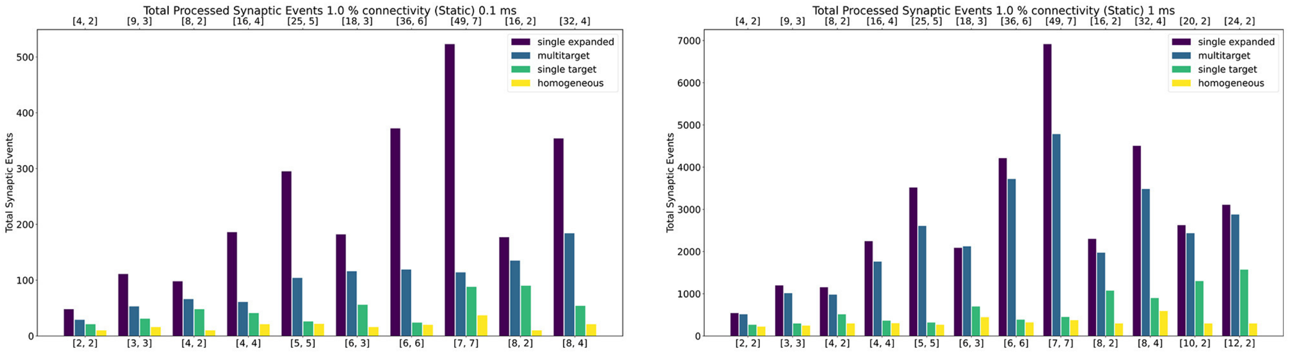


FIGURE 10 | Peak processed synaptic events per timestep. The presented configuration represents a 1% connectivity static network. Both 0.1 ms **(left)** and 1 ms **(right)** cases are shown. Each plot contains the results for the Single target expanded (purple), multi-target (blue), single target (green) and baseline homogeneous partitioning (yellow) cases. The horizontal axes show the number of cores per ensemble in the form of [S_c, N_c], as indicated in Section 4.2.1 and **Figure 9**. The top axis refers to the single target expanded case (purple), the bottom to the other cases.

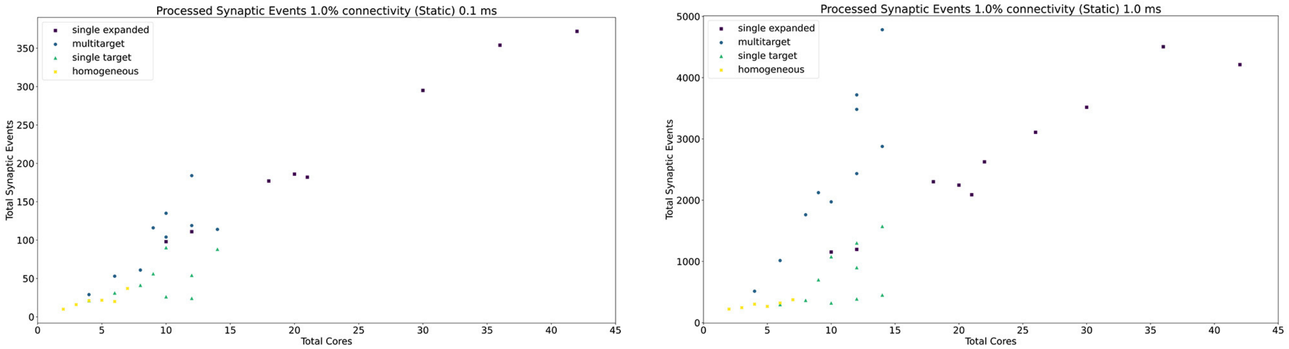


FIGURE 11 | Resource allocation vs. peak performance for the different partitioning strategies (*single target expanded*, *multi-target*, *single target*, and baseline homogeneous). The network is the same used for **Figure 10**, with 1% connectivity and static connections. The color scheme matches that used in **Figure 10**. For a case by case labeled version of this plot, please refer to the **Supplementary Material**.

Supplementary Material including 0.1, 5, and 10% connectivity patterns for both the 0.1 and 1 ms timestep resolutions.

4.2.3. Plastic Networks

Figure 12 shows the results of the experiment with the addition of synaptic plasticity. The color scheme for the bar chart is analogous to the static case and the network is run with 1 ms timestep. Connectivity probability is set at 1%, additional analysis (including 0.1%, 5% and 10% connectivities) can be found in the **Supplementary Material**. The same type of experiment was run for the plastic case, with the exception of the connections being defined through STDP with Spike-Pair rule for timing dependence and additive weight dependence (Morrison et al., 2008). The number of firing neurons has been reduced compared to the static case, as synaptic processing for plastic synapses requires additional steps (as highlighted in Section 3.5). For details regarding population sizes and the employed plasticity rule, please refer to the **Supplementary Material**.

Similarly to the static case, the *Multi-target* approach shows better performance than the *single target* case for all simulated configurations, demonstrating again that the approach makes better use of the available resources. For very sparse networks, with plastic synapses, the *Multi-target* approach achieves peak synaptic event throughput very close to the *single target expanded* simulations. This is due to the differences in processing plastic synapses compared to static synapses. Plasticity, requires the updated weights to be written back to shared memory, therefore doubling the accesses to SDRAM compared to the static case. This operation becomes extremely costly when the number of receptors per row are limited. Therefore, having longer synaptic rows, as in the case of the *Multi-target* approach, allows to further increase the number of synaptic events that can be processed per timestep. **Figure 13** contains a comparison of the general trend for the three approaches (refer to the **Supplementary Material** for a case by case labeled representation of this plot). Similarly to the static case, the Multi-target partitioning shows the steepest increase of processed synaptic events per timestep (vertical axis) with increasing allocated resources (horizontal axis). This further demonstrates that the Multi-target partitioning achieves better performance than previous approaches when the same hardware

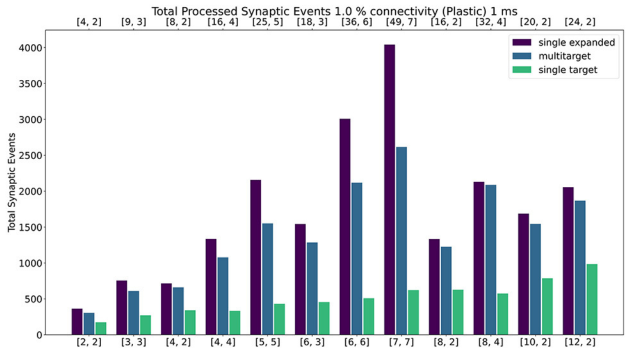


FIGURE 12 | Peak processed synaptic events per timestep. The presented configuration represents a 1% connectivity network with plastic connections. Timestep resolution is set to 1 ms. The plot shows results for the *single target expanded* (purple), *multi-target* (blue) and *single target* (green) cases. The horizontal axes show the number of cores per ensemble in the form of [[S_C, N_C]], as indicated in Section 4.2.1 and **Figure 9**. The **top** axis refers to the single target expanded case (purple), the **bottom** to the other cases.

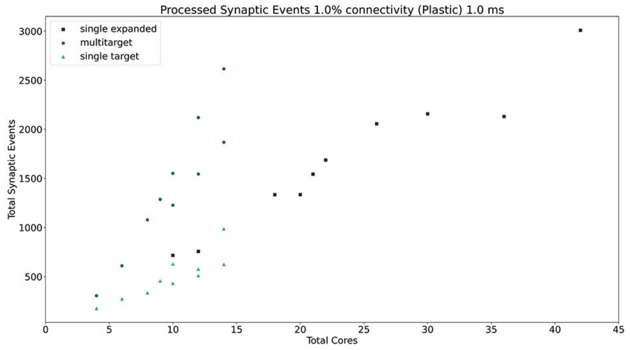


FIGURE 13 | Resource allocation vs. peak performance for the three different approaches (*single target expanded*, *multi-target*, and *single target*). The network is the same used for **Figure 12**, with 1% connectivity and plastic connections. The color scheme matches that used in **Figure 12**. For a case by case labeled version of this plot, please refer to the **Supplementary Material**.

resources are available and comparable results with reduced hardware requirements, also for SNN simulations involving synaptic plasticity.

4.3. Sparsity Profiling

4.3.1. Experiment Description

Profiling of peak synaptic event throughput with a range of connection sparsity levels is now explored. This experiment shows the variation of the processed synaptic events per timestep with increasing numbers of target *Neuron* cores. The number of *Synapse* cores is kept fixed and the target *Neuron* cores are gradually increased. In order to provide a good balance (and according to the peak performance shown in Section 4.2), the chosen number of *Synapse* cores is 7 and the target *Neuron* cores range from 1 to 7, guaranteeing to fit on a single chip. This allocation also allows equal comparison between simulations with 1 ms timestep resolution and 0.1 ms, having set the number of neurons per *Neuron* core in both cases to 64. The connectivity probabilities investigated are: 0.1, 1, 10, and 50%. Connectivity patterns above 50% are beyond the scope of this study, as they are extremely rare in biology (Hagmann et al., 2008), and are handled sufficiently well by traditional hardware (GPUs, CPUs, etc.). The network employed for this experiment has a structure analogous to that described in Section 4.2.1. For this case various sparsity patterns are shown, together with different cores allocations per chip. This experiment is useful to demonstrate the flexibility of the approach in handling multiple sparsity levels, a common feature in biologically-representative SNNs (Schmidt et al., 2018).

4.3.2. Sparsity Results

The results for this experiment are shown in **Figure 14** left for 0.1 ms timestep resolution and in **Figure 14** right for 1 ms timestep resolution. The horizontal axis shows the connectivity probabilities, the vertical axis the processed synaptic events per timestep. Each line represents a different configuration of *Synapse* cores to *Neuron* cores, where each *Synapse* core is connected to all the targets of that configuration. The number of postsynaptic receptors per *Synapse* core therefore can be obtained by multiplying the number of *Neuron* cores by 64 (number of neurons per *Neuron* core).

For the 1 ms case (**Figure 14** right), as expected, simulations with higher number of targets process the highest number of synaptic events per timestep. The most evident jump happens between the configurations with 1 and 2 targets, respectively, where the synaptic rows double in size. This shows that having larger synaptic rows impacts processing times, especially for very sparse networks, by improving the processed synaptic events of ≈ 1 order of magnitude for 0.1% connectivity between worst and best case. This gain reduces when the connectivity probability increases, because of multiple synaptic events are carried per spike. Therefore, the time processing per spike increases as well.

The 0.1 ms case (**Figure 14** left) follows a similar trend to the 1 ms case, however the examples with 6 and 7 targets do not give any improvements. The reason for this is due to the time required to perform the transfers between shared and local memories for the synaptic contributions, which have a higher impact on the timestep relative to the 1 ms case. For the sparse simulations (0.1% and 1% connectivity), having multiple target *Neuron* cores gives advantage similarly to the 1 ms case, however, when the network becomes denser the trend starts to invert, as the cost of processing a single incoming spike dominates over the gain introduced by this approach.

5. DISCUSSION

This work presents a novel parallelization approach for neural processing on Neuromorphic hardware, which improves the performance of SNN simulations by acting on the way synaptic matrices are partitioned and processed. The Multi-target partitioning approach provides additional freedom when designing SNN simulations, by allowing to target applications more specifically, according to their requirements. By allowing parameterization of synaptic and neural processing units, it is possible to allocate the appropriate amount of resources for a

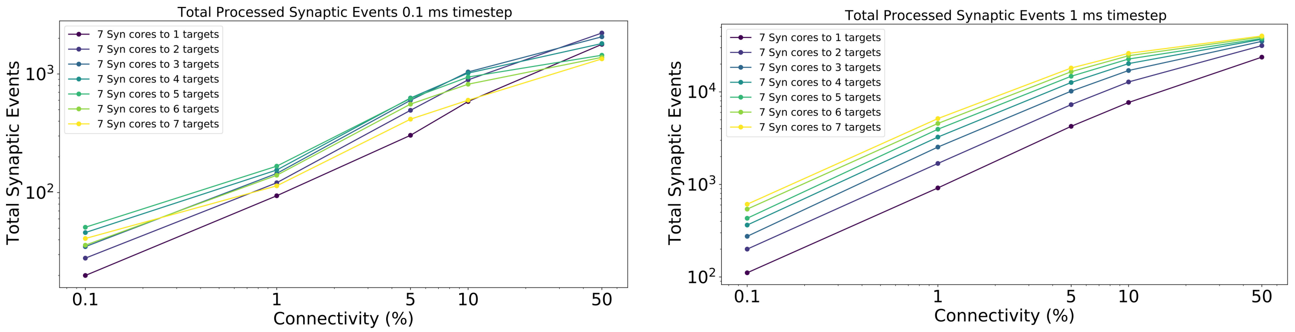


FIGURE 14 | Processed synaptic events for different connectivity configurations. Both the 0.1 ms timestep **(left)** and 1 ms timestep **(right)** cases are shown. The vertical axis shows the total processed synaptic events per timestep, the horizontal axis different connectivity probabilities. Each line represents a different neural ensemble with constant *Synapse* cores, but increasing target *Neuron* cores.

given requirement, prioritizing the number of *Neuron* processing units for sparser applications and increasing the number of *Synapse* processing units when the fan-in dominates. Thanks to these improvements it is possible to maximize the performance, while using minimal hardware resources and therefore reducing power consumption.

Through a SpiNNaker implementation of the Multi-target partitioning approach, it is possible to improve the peak synaptic processing throughput up to 9× compared to previous results for the same hardware resources. Furthermore, it is possible to obtain comparable processed synaptic events per ms, by reducing the hardware resources to a quarter, resulting in a much smaller machine (and energy consumption) dedicated to the simulation (as detailed in Section 4.2).

The Multi-target partitioning approach additionally enables optimal processing of incoming spike packets, providing a larger pool of target neurons for each spike, hence increasing the length of processed synaptic rows for a given connection density. This greatly reduces the required number of accesses to shared memory per timestep, therefore allowing more efficient processing of sparsely connected networks (detailed in Section 4.3). This is shown by Equations (8) and (13), where the number of target neurons of each spike grows according to the number of target *Neuron* cores, expanding the limit beyond a single postsynaptic *Neuron* core. This has the effect of reducing, by a factor N_c the number of destination processors per spike packet, facilitating the routing of spike packets and so reducing the pressure on the communication fabric. Furthermore, this increased number of targets per spike packet allows to amortize the dominating fixed cost of processing a spike (c_s) (Rhodes et al., 2018) over a higher number of postsynaptic receptors, which can now be larger than that of a single *Neuron* core, overcoming this limitation which is still observed for the Heterogeneous partitioning.

The Multi-target partitioning approach is optimal as it comes with minimal additional costs compared to previous approaches. However, the SpiNNaker implementation is limited by the different access patterns to shared memory. The shared memory access time plays a key role in the fraction of the timestep available for spike processing, as shown by Equation (9) and by the recorded values presented in Sections 4.1.2 and 4.1.3. The relatively old technology employed by SpiNNaker represents a bottleneck in this context, resulting in both memory contention and transfer size limiting the total system throughput. This causes the synaptic contributions writing (t_w) and reading (t_r) times (Equations 9–11) to increase with the number of cores in the ensemble, consuming approximately half the timestep duration for high timestep resolution simulations such as 0.1 ms. For this reason the need for faster access to shared memory is proven, by showing that there is a large potential gain in having access to multiple separate shared memories, compared to a single shared memory. This consideration opens up to the possibility of using more advanced memory architectures for Neuromorphic hardware, such as multiport memories, since structures like synaptic matrices and synaptic contributions are non-overlapping and therefore would benefit from the capability of separate independent accesses.

The flexibility of the approach also makes it portable and extendable for the next generation of digital Neuromorphic platforms. SpiNNaker 2, by exploiting its chip organization of cores in quartets, namely QPEs (Höppner et al., 2021; Yan et al., 2021), could map a cluster-based implementation of multiple neural ensembles per chip, where each processor (PE) represents either a *Neuron* core or a *Synapse* core. Since each PE has the capability to efficiently access the local memory of other PEs on the same QPE, it is possible to efficiently share the synaptic contributions within a QPE, overcoming the contention issue. A step further would include a tree-like structure, where QPEs could implement a group of 4 *Synapse* cores, which generate the synaptic contributions as a single block for the 4 cores. Then, a single PE per QPE accesses the chip shared memory to communicate with other QPEs implementing blocks of *Neuron* cores. Following the same strategy, a single *Neuron* core per *Neuron* QPE accesses the shared memory to retrieve the contributions. This would expand the ensemble capabilities to a full chip (up to 160 cores), limiting the memory contention to a quarter of the cores in use, which combined with the much higher memory throughput (6 vs. 1 GB/s for the SpiNNaker SDRAM) would have a large impact on the synaptic contributions reading and writing times.

The Multi-target partitioning approach also has potential benefits in Neuromorphic systems where all synaptic information is stored locally to the computational units. For these systems the approach would allow synaptic compartments to target multiple neural compartments, improving the handling of sparse connections, and overcoming the limitations set by the fixed coupling between synaptic and neural units. Furthermore the added benefits seen when processing plastic connections offers advantages for online learning applications, particularly in sparsely-connected biologically-representative SNNs.

DATA AVAILABILITY STATEMENT

The material and the code generated for this study, as well as the experiments, are available from the SpiNNaker software stack: https://github.com/SpiNNakerManchester, using the branch Multitarget_syn_cores.

AUTHOR CONTRIBUTIONS

LP led the design of the Multi-target partitioning model, built the SpiNNaker implementation, designed and ran the experiments and drafted the manuscript. OR co-designed the Multi-target partitioning model and supervised the research. Both authors read, commented, and approved the final manuscript.

FUNDING

The design and construction of the SpiNNaker machine was supported by EPSRC (the UK Engineering and Physical Sciences Research Council) under grant EP/D07908X/1 and EP/G015740/1, in collaboration with the universities of Southampton, Cambridge, and Sheffield and with industry partners ARM Ltd., Silistix Ltd., and Thales. Ongoing development of the software is supported by the EU ICT Flagship Human Brain Project (H2020 785907 and 945539), in collaboration with many university and industry partners across the EU and beyond. LP was funded by an EPSRC DTA studentship in the Department of Computer Science.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fnins.2022.867027/full#supplementary-material

REFERENCES

Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., et al. (2015). TrueNorth: design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Trans. Comput. Aided Design Integr. Circ. Syst*. 34, 1537–1557. doi: 10.1109/TCAD.2015.2474396

ARM (2006). *ARM968E-S Technical Reference Manual*. ARM. Available online at: https://developer.arm.com/documentation/ddi0311/

Bogdan, P. A., Marcinne, B., Casellato, C., Casali, S., Rowley, A. G., Hopkins, M., et al. (2021). Towards a bio-inspired real-time neuromorphic cerebellum. *Front. Cell. Neurosci*. 15, 622870. doi: 10.3389/fncel.2021.622870

Casali, S., Marenzi, E., Medini, C., Casellato, C., and D'Angelo, E. (2019). Reconstruction and simulation of a scaffold model of the cerebellar network. *Front. Neuroinform*. 13, 37. doi: 10.3389/fninf.2019.00037

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Davison, A., Bruderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform*. 2, 11. doi: 10.3389/neuro.11.011.2008

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The SpiNNaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., et al. (2013). Overview of the SpiNNaker system architecture. *IEEE Trans. Comput*. 62, 2454–2467. doi: 10.1109/TC.2012.142

Galluppi, F., Lagorce, X., Stromatias, E., Pfeiffer, M., Plana, L. A., Furber, S. B., et al. (2015). A framework for plasticity implementation on the SpiNNaker neural architecture. *Front. Neurosci*. 8, 429. doi: 10.3389/fnins.2014.00429

Gerstner, W., and Kistler, W. M. (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. (Cambridge: Cambridge University Press). doi: 10.1017/CBO9780511815706

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430. doi: 10.4249/scholarpedia.1430

Hagmann, P., Cammoun, L., Gigandet, X., Meuli, R., Honey, C. J., Wedeen, V. J., et al. (2008). Mapping the structural core of human cerebral cortex. *PLoS Biol*. 6, e159. doi: 10.1371/journal.pbio.0060159

Heittmann, A., Psychou, G., Trensch, G., Cox, C. E., Wilcke, W. W., Diesmann, M., et al. (2022). Simulating the cortical microcircuit significantly faster than real time on the IBM INC-3000 neural supercomputer. *Front. Neurosci*. 15, 728460. doi: 10.3389/fnins.2021.728460

Höppner, S., Yan, Y., Dixius, A., Scholze, S., Partzsch, J., Stolba, M., et al. (2021). The SpiNNaker 2 processing element architecture for hybrid digital neuromorphic computing. *arXiv preprint arXiv:2103.08392*. doi: 10.48550/arXiv.2103.08392

Indiveri, G., Linares-Barranco, B., Hamilton, T., van Schaik, A., Etienne-Cummings, R., Delbruck, T., et al. (2011). Neuromorphic silicon neuron circuits. *Front. Neurosci*. 5, 73. doi: 10.3389/fnins.2011.00073

Ippen, T., Eppler, J. M., Plesser, H. E., and Diesmann, M. (2017). Constructing neuronal network models in massively parallel environments. *Front. Neuroinform*. 11, 30. doi: 10.3389/fninf.2017.00030

Knight, J. C., and Furber, S. B. (2016). Synapse-centric mapping of cortical models to the SpiNNaker neuromorphic architecture. *Front. Neurosci*. 10, 420. doi: 10.3389/fnins.2016.00420

Knight, J. C., Komissarov, A., and Nowotny, T. (2021). PyGeNN: a Python library for GPU-enhanced neural networks. *Front. Neuroinform*. 15, 659005. doi: 10.3389/fninf.2021.659005

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci*. 1, 136–142. doi: 10.1038/s43588-020-00022-7

Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2021). Sub-realtime simulation of a neuronal network of natural density. *Neuromorph. Comput. Eng*. 2, 021001. doi: 10.1088/2634-4386/ac55fc

Levy, W. B., and Calvert, V. G. (2020). Computation in the human cerebral cortex uses less than 0.2 watts yet this great expense is optimal when considering communication costs. *bioRxiv*. 1, 1–13. doi: 10.1101/2020.04.23.057927

Mavaridas, J., Lujan, M., Plana, L. A., Temple, S., and Furber, S. B. (2015). SpiNNaker: enhanced multicast routing. *Parallel Comput*. 45, 49–66. doi: 10.1016/j.parco.2015.01.002

Mead, C. (1989). *Analog VLSI and Neural Systems*. (Boston, MA: Addison-Wesley Longman Publishing Co., Inc).

Mead, C. (1990). Neuromorphic electronic systems. *Proc. IEEE* 78, 1629–1636. doi: 10.1109/5.58356

Moradi, S., Qiao, N., Stefanini, F., and Indiveri, G. (2018). A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Trans. Biomed. Circ. Syst*. 12, 106–122. doi: 10.1109/TBCAS.2017.2759700

Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biol. Cybern*. 98, 459–478. doi: 10.1007/s00422-008-0233-1

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Comput*. 17, 1776–1801. doi: 10.1162/0899766054026648

Painkras, E., Plana, L. A., Garside, J., Temple, S., Galluppi, F., Patterson, C., et al. (2013). SpiNNaker: a 1-W 18-core system-on-chip for massively-parallel neural network simulation. *IEEE J. Solid State Circ*. 48, 1943–1953. doi: 10.1109/JSSC.2013.2259038

Plana, L. A., Clark, D., Davidson, S., Furber, S. B., Garside, J. D., Painkras, E., et al. (2011). SpiNNaker: design and implementation of a GALS multicore system-on-chip. *ACM J. Emerg. Technol. Comput. Syst*. 4, 1–18. doi: 10.1145/2043643.2043647

Potjans, T. C., and Diesmann, M. (2012). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Rhodes, O., Bogdan, P. A., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., et al. (2018). sPyNNaker: a software package for running PyNN simulations on SpiNNaker. *Front. Neurosci*. 12, 816. doi: 10.3389/fnins.2018.00816

Rhodes, O., Peres, L., Rowley, A. G. D., Gait, A., Plana, L. A., Brenninkmeijer, C., et al. (2019). Real-time cortical simulation on neuromorphic hardware. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci*. 378. doi: 10.1098/rsta.2019.0160

Rotter, S., and Diesmann, M. (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biol. Cybern*. 81, 381–402. doi: 10.1007/s004220050570

Rowley, A. G. D., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., Lester, D. R., et al. (2019). SpiNNTools: the execution engine for the SpiNNaker platform. *Front. Neurosci*. 13, 231. doi: 10.3389/fnins.2019.00231

Schemmel, J., Kriener, L., Muller, P., and Meier, K. (2017). "An accelerated analog neuromorphic hardware system emulating NMDA- and calcium-based non-linear dendrites," in *2017 International Joint Conference on Neural Networks (IJCNN)*. (Anchorage), 2217–2226. doi: 10.1109/IJCNN.2017.7966124

Schmidt, M., Bakker, R., Shen, K., Bezgin, G., Diesmann, M., and van Albada, S. J. (2018). A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas. *PLoS Comput. Biol*. 14, e1006359 doi: 10.1371/journal.pcbi.1006359

Sharp, T., and Furber, S. B. (2013). "Correctness and performance of the SpiNNaker architecture," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*. (Dallas), 1–8. doi: 10.1109/IJCNN.2013.6706988

Sharp, T., Plana, L. A., Galluppi, F., and Furber, S. B. (2011). "Event-driven simulation of arbitrary spiking neural networks on SpiNNaker," in *ICONIP*. (Heidelberg) doi: 10.1007/978-3-642-24965-5_48

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Front. Neurosci*. 12, 291. doi: 10.3389/fnins.2018.00291

Yan, Y., Stewart, T. C., Choo, X., Vogginger, B., Partzsch, J., Hoppner, S., et al. (2021). Comparing Loihi with a SpiNNaker 2 prototype on low-latency keyword spotting and adaptive robotic control. *Neuromorph. Comput. Eng*. 1, 014002. doi: 10.1088/2634-4386/abf150

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

ORIGINAL RESEARCH
published: 11 May 2022
doi: 10.3389/fninf.2022.837549

A Modular Workflow for Performance Benchmarking of Neuronal Network Simulations

Jasper Albers[1,2], Jari Pronold[1,2], Anno Christopher Kurth[1,2], Stine Brekke Vennemo[3], Kaveh Haghighi Mood[4], Alexander Patronis[4], Dennis Terhorst[1], Jakob Jordan[5], Susanne Kunkel[3], Tom Tetzlaff[1], Markus Diesmann[1,6,7] and Johanna Senk[1]*

[1] Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6) and JARA-Institute Brain Structure-Function Relationships (INM-10), Jülich Research Centre, Jülich, Germany, [2] RWTH Aachen University, Aachen, Germany, [3] Faculty of Science and Technology, Norwegian University of Life Sciences, Ås, Norway, [4] Jülich Supercomputing Centre (JSC), Jülich Research Centre, Jülich, Germany, [5] Department of Physiology, University of Bern, Bern, Switzerland, [6] Department of Physics, Faculty 1, RWTH Aachen University, Aachen, Germany, [7] Department of Psychiatry, Psychotherapy and Psychosomatics, School of Medicine, RWTH Aachen University, Aachen, Germany

Modern computational neuroscience strives to develop complex network models to explain dynamics and function of brains in health and disease. This process goes hand in hand with advancements in the theory of neuronal networks and increasing availability of detailed anatomical data on brain connectivity. Large-scale models that study interactions between multiple brain areas with intricate connectivity and investigate phenomena on long time scales such as system-level learning require progress in simulation speed. The corresponding development of state-of-the-art simulation engines relies on information provided by benchmark simulations which assess the time-to-solution for scientifically relevant, complementary network models using various combinations of hardware and software revisions. However, maintaining comparability of benchmark results is difficult due to a lack of standardized specifications for measuring the scaling performance of simulators on high-performance computing (HPC) systems. Motivated by the challenging complexity of benchmarking, we define a generic workflow that decomposes the endeavor into unique segments consisting of separate modules. As a reference implementation for the conceptual workflow, we develop `beNNch`: an open-source software framework for the configuration, execution, and analysis of benchmarks for neuronal network simulations. The framework records benchmarking data and metadata in a unified way to foster reproducibility. For illustration, we measure the performance of various versions of the `NEST` simulator across network models with different levels of complexity on a contemporary HPC system, demonstrating how performance bottlenecks can be identified, ultimately guiding the development toward more efficient simulation technology.

Keywords: spiking neuronal networks, benchmarking, large-scale simulation, high-performance computing, workflow, metadata

Edited by:
Ludovico Minati,
Tokyo Institute of Technology, Japan

Reviewed by:
Justin Wozniak,
Argonne National Laboratory (DOE),
United States
David Phillip Nickerson,
The University of Auckland,
New Zealand

****Correspondence:***
Jasper Albers
j.albers@fz-juelich.de

Received: *16 December 2021*
Accepted: *11 March 2022*
Published: *11 May 2022*

Citation:
Albers J, Pronold J, Kurth AC, Vennemo SB, Haghighi Mood K, Patronis A, Terhorst D, Jordan J, Kunkel S, Tetzlaff T, Diesmann M and Senk J (2022) A Modular Workflow for Performance Benchmarking of Neuronal Network Simulations. Front. Neuroinform. 16:837549. doi: 10.3389/fninf.2022.837549

1. INTRODUCTION

Past decades of computational neuroscience have achieved a separation between mathematical models and generic simulation technology (Einevoll et al., 2019). This enables researchers to simulate different models with the same simulation engine, while the efficiency of the simulator can be incrementally advanced and maintained as a research infrastructure. Increasing computational efficiency does not only decrease the required resources of simulations, but also allows for constructing larger network models with an extended explanatory scope and facilitates studying long-term effects such as learning. Novel simulation technologies are typically published together with verification—evidence that the implementation returns correct results—and validation—evidence that these results are computed efficiently. Verification implies correctness of results with sufficient accuracy for suitable applications as well as a flawless implementation of components confirmed by unit tests. For spiking neuronal network simulators, such applications are simulations of network models which have proven to be of relevance for the field. In a parallel effort, validation aims at demonstrating the added value of the new technology for the community. To this end, the new technology is compared to previous studies on the basis of relevant performance measures.

Efficiency is measured by the resources used to achieve the result. Time-to-solution, energy-to-solution and memory consumption are of particular interest. For the development of neuromorphic computing systems, efficiency in terms of low power consumption and fast execution is an explicit design goal: simulations need to be able to cope with limited resources, for example, due to hardware constraints. Real-time performance, meaning that simulated model time equals wall-clock time, is a prerequisite for simulations interacting with the outer world, such as in robotics. Even faster, sub-real-time simulations enable studies of slow neurobiological processes such as brain development and learning, which take hours, days, or more in nature. High-performance computing (HPC) benchmarking studies usually assess the scaling performance of the simulation architecture by incrementally increasing the amount of employed hardware resources (e.g., compute nodes). In weak-scaling experiments, the size of the simulated network model is increased proportionally to the computational resources, which keeps the workload per compute node fixed if the simulation scales perfectly. Scaling neuronal networks, however, inevitably leads to changes in the network dynamics (van Albada et al., 2015b). Comparisons between benchmarking results obtained at different scales are therefore problematic. For network models of natural size describing the correlation structure of neuronal activity, strong-scaling experiments (in which the model size remains unchanged) are more relevant for the purpose of finding the limiting time-to-solution. For a formal definition of strong and weak scaling refer to page 123 of Hager and Wellein (2010) and for pitfalls in interpreting the scaling of network simulation code see van Albada et al. (2014). When measuring time-to-solution, studies distinguish between different phases of the simulation, in the simplest case between a setup phase of network construction and the actual simulation phase of state propagation. Such benchmark metrics not only depend on the simulation engine and its options for time measurements (see, e.g., Jordan et al., 2018; Golosio et al., 2021), but also on the network model. The simulated activity of a model may not always be stationary over time, and transients with varying firing rates are reflected in the computational load. For an example of transients due to arbitrary initial conditions see Rhodes et al. (2019), and for an example of non-stationary network activity, refer to the meta-stable state of the multi-area model described by Schmidt et al. (2018a). Studies assessing energy-to-solution need to specify whether only the power consumption of the compute nodes is considered or interconnects and required support hardware are also accounted for (van Albada et al., 2018).

The omnipresence of benchmarks in studies on simulation technology demonstrates the relevance of efficiency. The intricacy of the benchmarking endeavor, however, not only complicates the comparison between these studies, but also reproducing them. Neuroscientific simulation studies are already difficult to reproduce (Crook et al., 2013; McDougal et al., 2016; Rougier et al., 2017; Gutzen et al., 2018; Pauli et al., 2018; Gleeson et al., 2019), and benchmarking adds another layer of complexity. Reported benchmarks may differ not only in the structure and dynamics of the employed neuronal network models, but also in the type of scaling experiment, soft- and hardware versions and configurations, as well as in the analysis and presentation of the results. **Figure 1** illustrates the complexity of benchmarking experiments in simulation science and identifies five main dimensions: “Hardware configuration”, “Software configuration”, “Simulators”, “Models and parameters”, and “Researcher communication”. The following presents examples specific to neuronal network simulations, demonstrating the range of each of the five dimensions.

Different simulators, some with decades of development, allow for large-scale neuroscientific simulations (Brette et al., 2007). We distinguish between simulators that run on conventional HPC systems and those that use dedicated neuromorphic hardware. Prominent examples of simulators for networks of spiking point-neurons are NEST (Morrison et al., 2005b; Gewaltig and Diesmann, 2007; Plesser et al., 2007; Helias et al., 2012; Kunkel et al., 2012, 2014; Ippen et al., 2017; Kunkel and Schenck, 2017; Jordan et al., 2018; Pronold et al., 2021, 2022) and Brian (Goodman and Brette, 2008; Stimberg et al., 2019) using CPUs; GeNN (Yavuz et al., 2016; Knight and Nowotny, 2018, 2021; Stimberg et al., 2020; Knight et al., 2021) and NeuronGPU (Golosio et al., 2021) using GPUs; CARLsim (Nageswaran et al., 2009; Richert et al., 2011; Beyeler et al., 2015; Chou et al., 2018) running on heterogeneous clusters; and the neuromorphic hardware SpiNNaker (Furber et al., 2014; Rhodes et al., 2019). NEURON (Carnevale and Hines, 2006; Migliore et al., 2006; Lytton et al., 2016) and Arbor (Akar et al., 2019) aim for simulating morphologically detailed neuronal networks.

The hardware and software configurations used in published benchmark studies are diverse because both underlie updates and frequent releases. In addition, different laboratories may not have access to the same machines. Therefore, HPC benchmarks are performed on different contemporary compute clusters or supercomputers. For example, NEST benchmarks have been

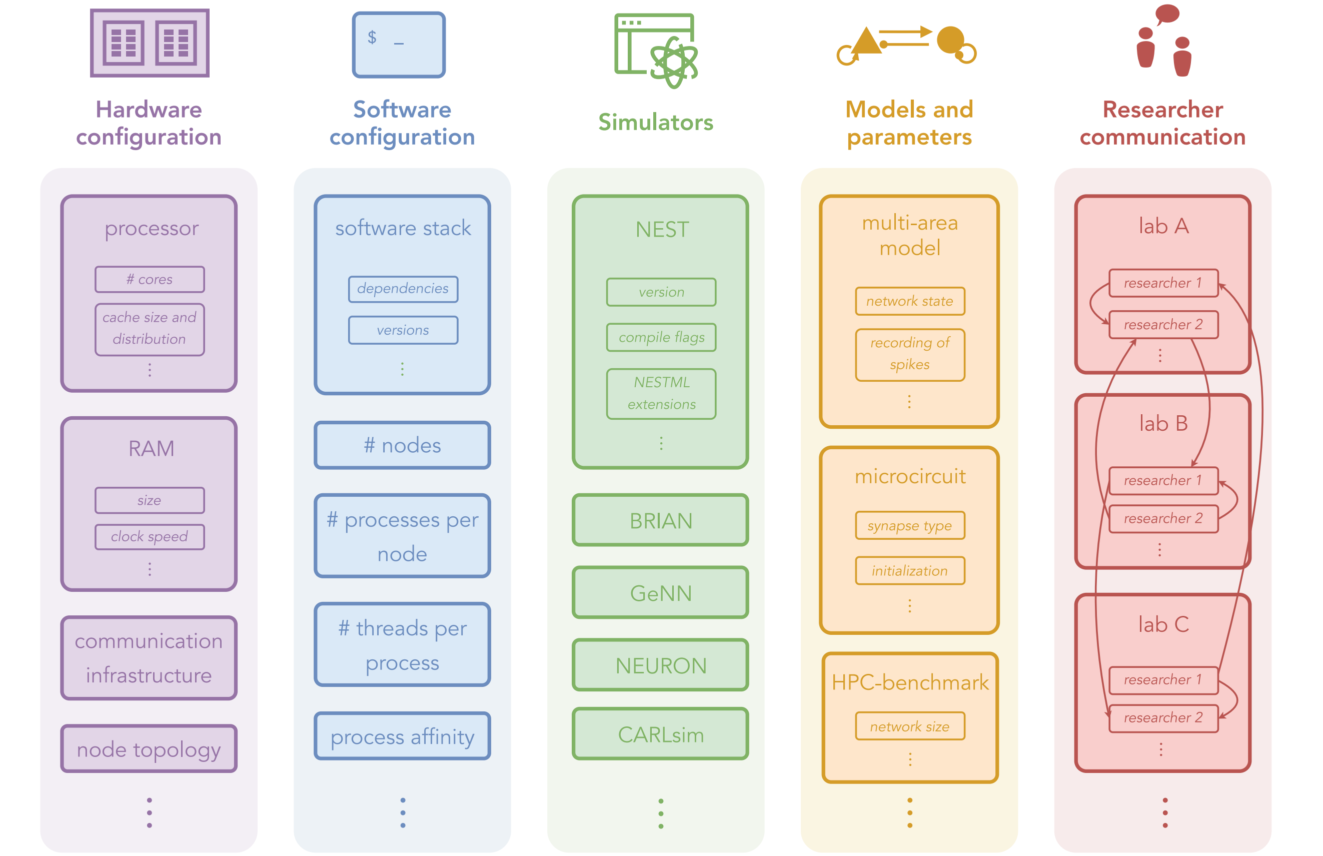


FIGURE 1 | Dimensions of HPC benchmarking experiments with examples from neuronal network simulations. Hardware configuration: computing architectures and machine specifications. Software configuration: general software environments and instructions for using the hardware. Simulators: specific simulation technologies. Models and parameters: different models and their configurations. Researcher communication: knowledge exchange on running benchmarks.

conducted on the systems located at Research Center Jülich in Germany but also on those at the RIKEN Advanced Institute for Computational Science in Japan (e.g., Helias et al., 2012; Jordan et al., 2018). To assess the performance of GPU-based simulators, the same simulation is typically run on different GPU devices; from low-end gaming GPUs to those installed in high-end HPC clusters (Knight and Nowotny, 2018; Golosio et al., 2021). This variety can be beneficial; performing benchmark simulations on only a single system can lead to unwanted optimization toward that type of machine. However, comparing results across different hard- and software is complicated and requires expert knowledge of the compared technologies in order to draw reasonable conclusions.

The modeling community distinguishes between functional models, where the validation is concerned with the questions if and how well a specific task is solved, and non-functional models, where an analysis of the network structure, dynamics, and activity is used for validation. Simulating the same model using different simulation engines often results in activity data which can only be compared on a statistical level. Spiking activity, for example, is typically evaluated based on distributions of quantities such as the average firing rate, rather than on precise spike times (Senk et al., 2017; van Albada et al., 2018). Reasons for that are inevitable differences between simulators such as different algorithms, number resolutions, or random number generators, combined with the fact that neuronal network dynamics is often chaotic, rapidly amplifying minimal deviations (Sompolinsky et al., 1988; van Vreeswijk and Sompolinsky, 1998; Monteforte and Wolf, 2010). The most frequently used models to demonstrate simulator performance are balanced random networks similar to the one proposed by Brunel (2000): generic two-population networks with 80% excitatory and 20% inhibitory neurons, and synaptic weights chosen such that excitation and inhibition are approximately balanced, similar to what is observed in local cortical networks. Variants differ not only in the parameterization but also in the neuron, synapse, and plasticity models, or other details. Progress in NEST development is traditionally shown by upscaling a model of this type, called "HPC-benchmark model", which employs leaky integrate-and-fire (LIF) neurons, alpha-shaped post-synaptic currents, and spike-timing-dependent plasticity (STDP) between excitatory neurons. The detailed model description and parameters can be found in Tables 1–3 of the Supplementary Material of Jordan et al. (2018). Other versions include a network of Izhikevich model neurons and STDP (Izhikevich, 2003) used by Yavuz et al. (2016) and Golosio et al. (2021), the COBAHH model with Hodgkin-Huxley type neurons and conductance-based synapses (Brette et al., 2007) used by Stimberg et al. (2020), and a version with excitatory LIF and inhibitory Izhikevich model neurons where excitatory synapses are updated with STDP and inhibitory-to-inhibitory connections do not exist is used by Chou et al. (2018). Even though balanced random networks are often used for weak-scaling experiments, they describe the anatomical and dynamical features of cortical circuits only at a small spatial scale and the upscaling affects the network dynamics (see van Albada et al., 2015b as indicated above). At larger scales, the natural connectivity becomes more complex than what is captured by this model type. Therefore, models of different complexity need to be benchmarked to guarantee that a simulation engine performs well across use cases in the community. In addition to the HPC-benchmark model, this study employs two more elaborate network models: the "microcircuit model" proposed by Potjans and Diesmann (2014) and the "multi-area model" by Schmidt et al. (2018a). The microcircuit model is an extension of the balanced random network model with an excitatory and an inhibitory neuron population in each of four cortical layers with detailed connectivity derived from experimental studies. The model spans 1 mm^2 of cortical surface, represents the cortical layers at their natural neuron and synapse densities, and has recently been used to compare the performance of different simulation engines; for instance, NEST and SpiNNaker (Senk et al., 2017; van Albada et al., 2018; Rhodes et al., 2019); NEST, SpiNNaker, and GeNN (Knight and Nowotny, 2018); and NEST and NeuronGPU (Golosio et al., 2021). The multi-area model comprises 32 cortical areas of the visual system where each is represented by an adapted version of the microcircuit model; results are available for NEST (van Albada et al., 2021) and GeNN (Knight and Nowotny, 2021). Comparing the performance of the same model across different simulators profits from a common model description. The simulator-independent language PyNN (Davison et al., 2009), for example, enables the use of the same executable model description for different simulator back ends. Testing new technologies only with a single network model is, however, not sufficient for general-purpose simulators and comes with the danger of optimizing the code base for one application, while impairing the performance for others.

Problems to reproduce the simulation outcome or compare results across different studies may not only be technical but also result from a miscommunication between researchers or a lack of documentation. Individual, manual solutions for tracking the hardware and software configuration, the simulator specifics, and the models and parameters used in benchmarking studies have, in our laboratories, proven inefficient when scaling up the number of collaborators. This effect is amplified if multiple laboratories are involved. Similar inter-dependencies are also present between the other four dimensions of **Figure 1**, making it hard to produce long-term comparable results; the exhibited intricacy of benchmarking is susceptible to errors as, for instance, small details in parameterization or configuration may have a large impact on performance.

Standardizing benchmarks can help to control the complexity but represents a challenge for the fast-moving and interdisciplinary field of computational neuroscience. While the field had some early success in the area of compartmental modeling (Bhalla et al., 1992) and Brette et al. (2007) made initial steps for spiking neuronal networks, neither a widely accepted set of benchmark models nor guidelines for performing benchmark simulations exist. In contrast, benchmarks are routinely employed in computer science, and established metrics help to assess the performance of novel hardware and software. The LINPACK benchmarks (Dongarra et al., 2003), for example, were initially released in 1979, and the latest version is used to rank the world's top supercomputers by testing their floating-point computing power (TOP500 list). Although this strategy has been successful for many years, it has also been criticized as misguiding hardware vendors toward solutions with high performance in formalized benchmarks but disappointing performance in real-world applications[1]. For the closely related field of deep learning, Dai and Berleant (2019) summarize seven key properties that benchmarking metrics should fulfill: relevance, representativeness, equity, repeatability, cost-effectiveness, scalability, and transparency. There exist standard benchmarks for machine learning and deep learning applications such as computer vision and natural language processing with standard data sets and a global performance ranking. The most prominent example is MLPerf[2] (Mattson et al., 2020). Another example is the High Performance LINPACK for Accelerator Introspection (HPL-AI) benchmark[3] which is the mixed-precision counterpart to the LINPACK benchmarks. Ostrau et al. (2020) propose a benchmarking framework for deep spiking neural networks and they compare results obtained with the simulators Spikey (Pfeil et al., 2013), BrainScales (Schemmel et al., 2010), SpiNNaker, NEST, and GeNN.

For measuring and comparing the scaling performance of large-scale neuronal network model simulations, there exists, to our knowledge, no unifying approach, yet. Recently, more laboratories make use of established simulators rather than developing their own, and computing resources have become available and interchangeable. The resulting increase in the size of user-communities comes with the demand for even more flexible and efficient simulators with demonstrated performance. To keep up with this progress, we see the need for a common benchmarking framework. We envision a consistently managed array of standard benchmark models together with standard ways for running them. The five dimensions outlined above lend themselves to a modular framework integrating distinct components which can be updated, extended, or replaced independently. The framework needs to cover all steps of the benchmarking process from configuration, to execution, to handling of results. For enabling

[1] https://www.technologyreview.com/2010/11/08/199100/why-chinas-new-supercomputer-is-only-technically-the-worlds-fastest

[2] https://mlcommons.org

[3] https://www.icl.utk.edu/hpl-ai

comparability and reproducibility, all relevant metadata and data need to be tracked. In this work, we develop a conceptual benchmarking workflow that meets these requirements. For a reference implementation named beNNch, we employ the JUBE Benchmarking Environment[4] and the simulator NEST in different versions (Gewaltig and Diesmann, 2007), and we assess the time-to-solution for the HPC-benchmark model, the microcircuit model (Potjans and Diesmann, 2014), and the multi-area model (Schmidt et al., 2018a) on the contemporary supercomputer JURECA-DC (Thörnig and von St. Vieth, 2021). The goal of this study is to set the cornerstone for reliable performance benchmarks facilitating the comparability of results obtained in different settings, and hence, supporting the development of simulators.

The Results section of this manuscript formalizes the general concepts of the benchmarking workflow (Section 2.1), implements these concepts into a reference benchmarking framework for the NEST simulator (Section 2.2), and applies the framework to generate and compare benchmarking data, thereby making a case for the relevance of benchmarking for simulator development (Section 2.3.1). After a discussion of our results in Section 3, Section 4 provides details of specific performance optimizations addressed in this work.

2. RESULTS

2.1. Workflow Concepts

We devise a generic workflow for performance benchmarking applicable to simulations running on conventional HPC architectures. The conceptual workflow depicted in **Figure 2** consists of four segments which depend on each other in a sequential fashion. The segments are subdivided into different modules which are related to the specific realizations used in our reference implementation of the workflow (Section 2.2). We use the term "workflow" to describe abstract concepts that are of general applicability with regard to benchmarking efforts, and "framework" to refer to the concrete software implementation we have developed. Further, we make the distinction between "internal" and "external" modules. Internal modules are considered essential building blocks of the workflow while external modules can be exchanged more readily. The following introduces each of the workflow's conceptual segments and explains how the proposed solution addresses the identified problems (cf. **Figure 1**).

2.1.1. Configuration and Preparation

The first of the four workflow segments consists of five distinct modules that together set up all necessary prerequisites for the simulation. First, the installation of the simulation software and its dependencies is handled by "software deployment", while "machine configuration" specifies parameters that control the simulation experiment conditions, as for example, how many compute nodes to reserve. Together, these two modules target the problem dimensions "hardware configuration", "software configuration", and "simulators". Addressing "models and parameters", the module "model" provides the network model implementation, while "model configuration" allows for passing parameters to the model such as the biological model time to be simulated, thereby separating the model from its parameters. Finally, the "user configuration" module confines user-specific data, such as file paths or compute budgets, to a single location.

2.1.2. Benchmarking

The second segment encompasses all modules related to actually running the benchmark simulation. Compute clusters typically manage the workload of the machine via queuing systems; therefore, compute-intensive calculations are submitted as jobs via scripts which define resource usage and hold instructions for carrying out the simulation. In the workflow, this is handled by the module aptly named "job script generation". Here, the first link between modules comes into play: the workflow channels model, user and machine configuration to create a job script and subsequently submit the script to the job queue via the module "job submission". With the simulator software prepared by the software-deployment module, "job execution" performs the model simulation given the job-submission parameters. While a simulation for neuroscientific research purposes would at this point focus on the output of the simulation, for example, neuronal spike times or voltage traces, benchmarking is concerned with the performance results. These are recorded in the final benchmarking module called "data generation".

2.1.3. Data- and Metadata Handling

A core challenge in conducting performance benchmarks is the handling of all produced data and metadata. While the former type of data here refers to the results of the performance measurements, the latter is an umbrella term describing the circumstances under which the data was recorded according to the dimensions of benchmarking (**Figure 1**). Since executing multiple simulations using different configurations, software, hardware, and models is an integral part of benchmarking, data naturally accumulates. Recording the variations across these dimensions leads to a multitude of metadata that needs to be associated to the measured data. Standardized formats for both types of data make the results comparable for researchers working with the same underlying simulation technology. The workflow segment "Data- and metadata handling" proposes the following solution. First, the raw performance data, typically stemming from different units of the HPC system, are gathered and unified into a standardized format, while the corresponding metadata is automatically recorded. Next, the metadata is associated to the unified data files, alleviating the need for manually keeping track of parameters, experiment choices and software environment conditions. While there are different possible solution for this, attaching the relevant metadata directly to the performance-data files simplifies filtering and sorting of results. Finally, "initial validation" allows for a quick glance at the results such that erroneous benchmarks can be swiftly identified.

[4] https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/_node.html

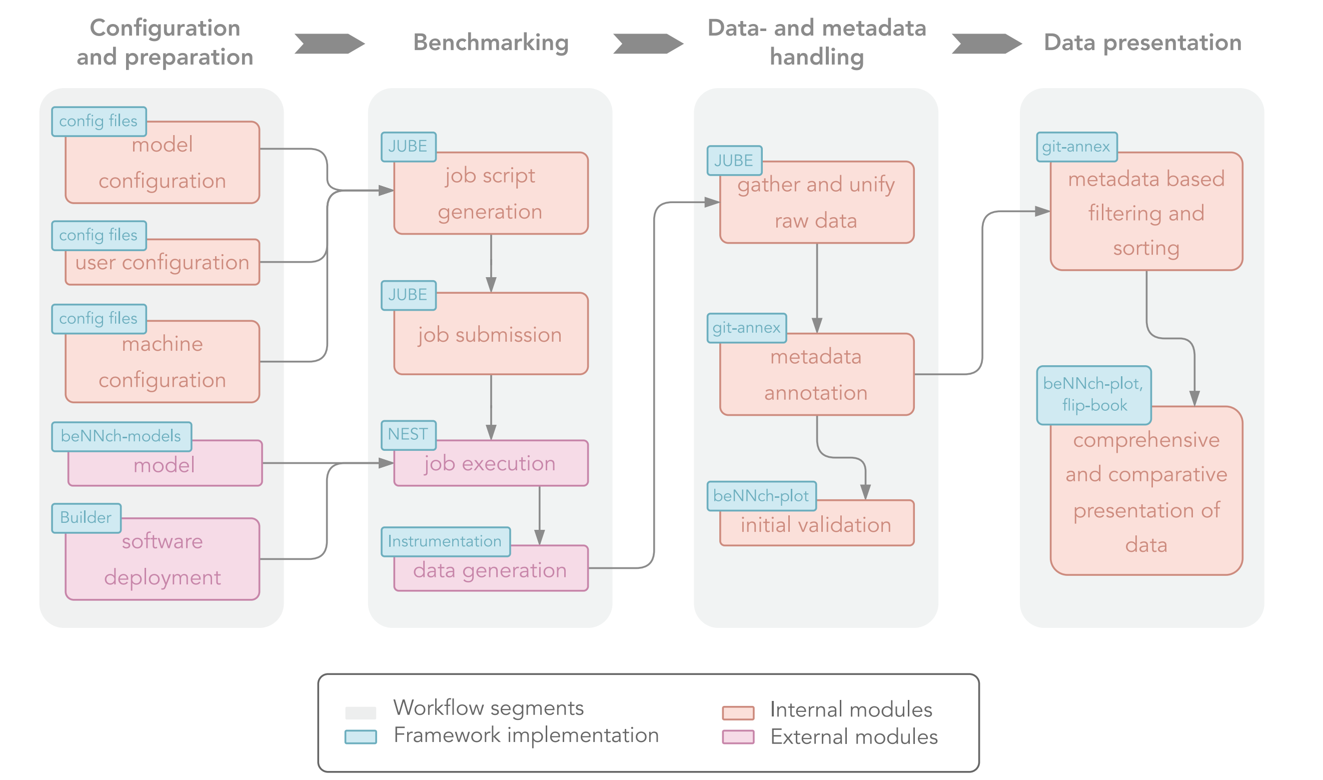


FIGURE 2 | Conceptual overview of the proposed benchmarking workflow. Light gray boxes divide the workflow into four distinct segments, each consisting of multiple modules. Internal modules are shown in orange and external ones in pink. Blue boxes indicate their respective realization in our reference implementation.

2.1.4. Data Presentation

This final workflow segment addresses the challenge of making the benchmarking results accessible and comparable such that meaningful conclusions can be drawn, thereby aiming to cope with the complexity that "Researcher communication" introduces. In a first step, "metadata based filtering and sorting" allows the user to dynamically choose the results to be included in the comparison. Here, dynamic means that arbitrary cuts through the hypercube of metadata dimensions can be selected such that the filtered results only differ in metadata fields of interest. Second, the data is presented in a format for which switching between benchmarks is intuitive, key metadata is given alongside the results, and data representation is standardized. The presentation of data should be comprehensive, consistent, and comparative such that the benchmarking results are usable in the long term. Thereby, the risk of wasting resources through re-generation of results is eliminated, making the corresponding software development more sustainable.

2.2. beNNch: A Reference Implementation

Building on the fundamental workflow concepts developed in Section 2.1, we introduce a reference implementation for modern computational neuroscience: beNNch[5]—a benchmarking framework for neuronal network simulations. The framework serves not only as a proof-of-concept, but also provides a software tool that can be readily used by neuroscientists and simulator developers. While beNNch is built such that plug-ins for any neuronal network simulator can be developed, we specifically implement compatibility with the NEST simulator (Gewaltig and Diesmann, 2007) designed for simulating large-scale spiking neuronal network models. In the following subsections, we detail software tools, templates, technologies, and user specifications needed to apply beNNch for benchmarking NEST simulations. Each of the conceptual modules of **Figure 2** is here associated with a concrete reference.

2.2.1. Builder

Reproducible software deployment is necessary for repeatability and comparability of the benchmarks. In favor of the usability of the benchmarking framework, however, we need to abstract non-relevant information on the hardware architecture and the software tool chain. The tool set is required to install software in a platform independent way and should not depend on a particular flavor of the operating system, the machine architecture or overly specific software dependencies. Additionally, it needs to be able to make use of system-provided tools and libraries, for example, to leverage machine specific MPI implementations. beNNch uses

[5] https://github.com/INM-6/beNNch

the tool Builder[6] for this purpose. Given a fixed software stack and hardware architecture, Builder provides identical executables by deriving the install instructions from "plan files". Integration with other package management systems such as easy_build (Geimer et al., 2014) or Spack (Gamblin et al., 2015) is achieved by using the same environment module systems[7]. Thereby, the required user interaction is minimized and, from a user perspective, installation reduces to the configuration of installation parameters. Given a specified variation of the software to be benchmarked, beNNch calls Builder to deploy the requested software. In doing so, Builder checks whether the software is already available and otherwise installs it according to the specifications in the plan file. The depth to which required dependencies need to be installed and which mechanisms are used depend on the conventions and prerequisites available at each execution site. For any installation, the used software stack—including library versions, compiler versions, compile flags, etc.—are recorded as metadata.

2.2.2. NEST

beNNch implements compatibility with the NEST simulator (Gewaltig and Diesmann, 2007), enabling the performance benchmarking of neuronal network simulations at the resolution of single neurons. The NEST software is complex, and the consequences of code modifications for performance are often hard to predict. NEST has an efficient C++ kernel, but network models and simulation experiments are defined via the user-friendly Python interface PyNEST (Eppler et al., 2009; Zaytsev and Morrison, 2014). To parallelize simulations, NEST provides two methods: for distributed computing, NEST employs the Message Passing Interface (MPI, Message Passing Interface Forum, 2009), and for thread-parallel simulation, NEST uses OpenMP (OpenMP Architecture Review Board, 2008).

2.2.3. Instrumentation

We focus our performance measurements on the time-to-solution. Acquiring accurate data on time consumption is critical for profiling and benchmarking. To this end, we make use of two types of timers to collect this data: the timers are either built-in to NEST on the C++ level, or they are included on the Python level as part of the PyNEST network-model description. The latter type of timers are realized with explicit calls to the function time.time() of the Python Standard Library's time. To achieve consistency throughout the framework, we use standardized variable names for the different phases of the simulation. **Figure 3** shows the simulation flow of a typical NEST simulation. During "network construction", neurons and auxiliary devices for stimulation and recording are created and subsequently connected according to the network-model description. Afterwards, in the course of "state propagation", the network state is propagated in a globally time-driven manner. This comprises four main phases which are repeated until the entire model time has been simulated: update of neuronal states, collocation of spikes in MPI-communication buffers, communication of spikes, and delivery of the received spikes to their respective thread-local targets. NEST's built-in timers provide a detailed look into the contribution of all four phases of state propagation, while timers on the Python level measure network construction and state propagation.

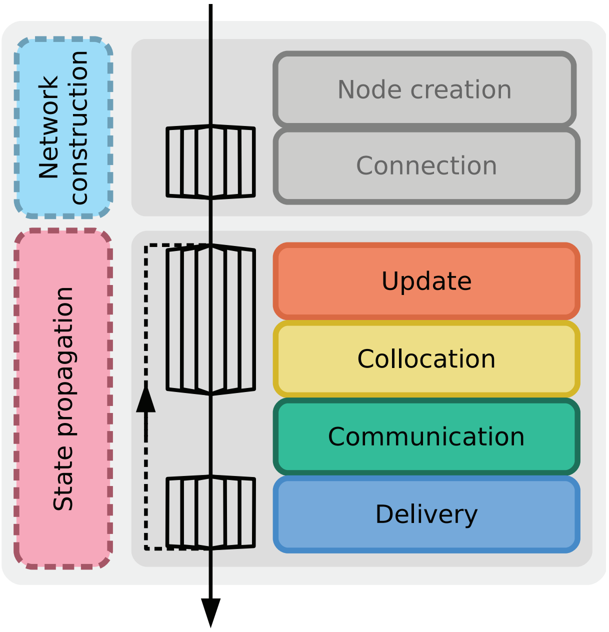


FIGURE 3 | Instrumentation to measure time-to-solution. Successive phases of a NEST simulation; time is indicated by top-down arrow. Fanning arrows denote parallel operation of multiple threads. The main phases network construction (cyan) and state propagation (pink) are captured by external timers on the Python level. Built-in NEST timers on the C++ level measure sub-phases: node creation and connection (both gray, not used in benchmark plots); update (orange), collocation (yellow), communication (green), and delivery (blue). The sub-phases of the state propagation are repeated until the simulation is finished as shown by the dashed arrow connecting delivery and update.

In NEST, the postsynaptic connection infrastructure is established during the "connection" phase. However, the presynaptic counterpart is typically only set up at the beginning of the state propagation phase (see Jordan et al., 2018, for details). In this work, we trigger this step deliberately and include it in our measurement of network-construction time rather than state-propagation time. Besides, it is common practice to introduce a short pre-simulation before the actual simulation to give the network dynamics time to level out; the state propagation phase is only recorded when potential startup transients have decayed (Rhodes et al., 2019). The model time for pre-simulation can be configured via a parameter in beNNch. For simplicity, **Figure 3** does not show this pre-simulation phase.

2.2.4. beNNch-models

We instantiate the "model" module with the repository beNNch-models[8] which contains a collection of PyNEST neuronal

[6] https://github.com/INM-6/Builder

[7] https://modules.readthedocs.io and http://lmod.readthedocs.io

[8] https://github.com/INM-6/beNNch-models

network models, i.e., models that can be simulated using the Python interface of NEST (Eppler et al., 2009). In principle, any such model can be used in conjunction with beNNch; only a few adaptations are required concerning the interfacing. On the input side, the framework needs to be able to set relevant model parameters. For recording the performance data, the required Python timers (Section 2.2.3) must be incorporated. On the output side, the model description is required to include instructions to store the recorded performance data and metadata in a standardized format. Finally, if a network model is benchmarked with different NEST versions that require different syntax, as is the case for switching between NEST 2.X and NEST 3.X, the model description also needs to be adjusted accordingly. Which model version is used in a simulation can thereby be deduced from knowing which simulator version was tested. For fine-grained version tracking, we additionally record the commit hash of beNNch-models and attach it as metadata to the results. Instructions on how to adapt existing models are provided in the documentation of beNNch-models.

The current version of beNNch provides benchmark versions of three widely studied spiking neuronal network models: the two-population HPC-benchmark model[9], the microcircuit model[10] by Potjans and Diesmann (2014) representing 1 mm^2 of cortical surface with realistic neuron and synapse densities, and the multi-area model[11] by Schmidt et al. (2018a,b) consisting of 32 microcircuit-like interconnected networks representing different areas of visual cortex of macaque monkey. The model versions used for this study employ the required modifications described above.

2.2.5. config files

When executing benchmarks, the main user interaction with beNNch consists of defining the characteristic parameters. We separate this from the executable code by providing yaml-based templates for "config files" to be customized by the user. Thereby, the information that defines a benchmark experiment is kept short and well arranged, limiting the number of files a user needs to touch and reducing the risk of user errors on the input side. **Listing 1** presents an excerpt from such a config file which has distinct sections to specify model, machine, and software parameters. While some parameters are model specific, standardized variable names are defined for parameters that are shared between models.

2.2.6. JUBE

At this point, the first segment of the benchmarking workflow (**Figure 2**) is complete and hence all necessary requirements are set up: the software deployment provides the underlying simulator (here: NEST with built-in instrumentation), the models define the simulation, and the configuration specifies the benchmark parameters. This information is now processed by the core element of the framework: generating and submitting simulation jobs and gathering and unifying the obtained performance data. We construct this component of beNNch around the xml-based JUBE[4] software tool using its yaml interface. Built around the idea of benchmarking, JUBE can fulfill the role of creating job scripts from the experiment, user and machine configuration, their subsequent submission, as well as gathering and unifying of the raw data output. Here, we focus on the prevalent scheduling software SLURM (Yoo et al., 2003), but extensions to allow for other workload managers would be straightforward to implement. Our approach aims at high code re-usability. Model specific code is kept to a minimum, and where necessary, written in a similar way across models. Adhering to a common interface between JUBE scripts and models facilitates the integration of new models, starting from existing ones as a reference. Since JUBE can execute arbitrary code, we use it to also record metadata in

```
parameterset:

    - name: model_parameters
      parameter:
       # can be either "metastable" or "ground"
       - {name: network_state, type: string, _: "metastable"}
       # biological model time to be simulated in ms
       - {name: model_time_sim, type: float, _: "10000."}
       # "weak" or "strong" scaling
       - {name: scaling_type, _: "strong"}

    - name: machine_parameters
      parameter:
       # number of compute nodes
       - {name: num_nodes, type: int, _: "4,8,12,16,24,32"}
       # number of MPI tasks per node
       - {name: tasks_per_node, type: int, _: "8"}
       # number of OpenMP threads per task
       - {name: threads_per_task, type: int, _: "16"}

    - name: software_parameters
      parameter:
       # simulator used for executing benchmarks
       - {name: simulator, _: "nest-simulator"}
       # simulator version
       - {name: version, _: "3.0"}
```

Listing 1: Excerpt of a config file in yaml-format for setting model, machine, and software parameters for benchmarking the multi-area model. When giving a list (e.g., for num_nodes), a job for each entry of the list is created. **Model parameters**: network_state describes particular model choices that induce different dynamical fixed points; model_time_sim defines the total model simulation time in ms; scaling_type sets up the simulation for either a weak- or a strong-scaling experiment. The former scales the number of neurons linearly with the used resources which might be ill-defined for anatomically constrained models. **Machine parameters**: num_nodes defines the number of nodes over which the scaling experiment shall be performed; tasks_per_node and threads_per_task specify the number of MPI tasks per node and threads per MPI task respectively. **Software parameters**: simulator and version describe which version of which simulator to use (and to install if not yet available on the machine).

[9] Original repository: https://github.com/nest/nest-simulator/blob/master/pynest/examples/hpc_benchmark.py.

[10] Original repository: https://github.com/nest/nest-simulator/tree/master/examples/nest/Potjans_2014.

[11] Original repository: https://github.com/INM-6/multi-area-model.

conjunction with each simulation. This includes specifications of the hardware architecture as well as parameters detailing the run and model configuration.

2.2.7. git-annex

Without a mature strategy for sharing benchmark results, communication can be a major obstacle. Typically, each researcher has their preferred workflow, thus results are shared over different means of communication, for example, via email attachments, cloud-based storage options, or `git` repositories. This makes it difficult to maintain an overview of all results, especially if researchers from different labs are involved. Ideally, results would be stored in a decentralized fashion that allows for tracking the history of files while allowing on-demand access for collaborators. To this end, we use `git-annex`[12] as a versatile base technology; it synchronizes file information in a standard `git` repository while keeping the content of large files in a separate object store, thereby keeping the repository size at a minimum. `git-annex` is supported by the GIN platform[13] which we employ for organizing our benchmark results. In addition, it allows for metadata annotation: instead of relying on separate files that store the metadata, `git-annex` can directly attach them to the data files, thereby implementing the "metadata annotation" module. Previously this needed to be cataloged by hand, whereas now the framework allows for an automatic annotation, reducing the workload on researchers and thus probability of human mistakes. A downside of following this approach is a limitation to command line-based interaction. Furthermore, `git-annex` is not supported by some of the more widely used `git` repository hosting services such as GitHub or GitLab in favor of `Git LFS`.

A difficult task when scaling up the usage of the framework and, by extension, handling large amounts of results, is providing an efficient way of dealing with user queries for specific benchmark results. Attaching the metadata directly to the performance data not only reduces the visible complexity of the repository, but also provides an efficient solution: `git-annex` implements a native way of selecting values for metadata keys via `git-annex` "views", automatically and flexibly reordering the results in folders and sub-folders accordingly. For example, consider the case of a user specifying the `NEST` version to be `3.0`, the `tasks_per_node` to be either `4` or `8`, and the `network_state` to be either `metastable` or `ground`. First, `git-annex` filters out metadata keys for which only a single value is given; in our example, only benchmarks conducted with `NEST` version `3.0` remain. Second, a hierarchy of directories is constructed with a level for each metadata key for which multiple options are given. Here, the top level contains the folders "4" and "8", each containing sub-folders `metastable` and `ground` where the corresponding results reside. However, it may be difficult to judge exactly what metadata is important to collect; oftentimes, it is only visible in hindsight that certain metadata is relevant for the simulation performance. Therefore, recording as much metadata as possible would be ideal, allowing for retrospective investigations if certain metadata becomes relevant after run time. Importantly, a balance needs to be struck between recording large amounts of metadata and keeping the volume of annotations manageable. In our implementation, we choose to solve this issue by recording detailed metadata about the system, software, and benchmarks, but only attaching what we currently deem relevant for performance to the data. The remaining metadata is archived and stored alongside the data, thereby sacrificing ease of availability for a compact format. This way, if future studies discover that a certain hardware feature or software parameter is indeed relevant for performance, the information remains accessible also for previously simulated benchmarks while staying relatively hidden otherwise. Furthermore, using `git` as a base technology allows to collect data sets provided by different researchers in a curated fashion by using well-established mechanisms like branches and merge-request reviews. This use of `git-annex` thereby implements the "metadata based filtering and sorting" module of **Figure 2**.

2.2.8. beNNch-plot

To enable a comparison between plots of benchmark results across the dimensions illustrated in **Figure 1** it is paramount to use the same plotting style. To this end, we have developed the standalone plotting package `beNNch-plot`[14] based on `matplotlib` (Hunter, 2007). Here, we define a set of tools to create individual plot styles that can be combined flexibly by the user. The standardized definitions of performance measures employed by `beNNch` directly plug into this package. In addition, `beNNch-plot` includes default plot styles that can be readily used, and provides a platform for creating and sharing new ones. `beNNch` utilizes the default plot styles of `beNNch-plot` for both initial validation—a preliminary plot offering a quick glance at the results, thereby enabling a swift judgement whether any problems occurred during simulation—and visualization of the final results.

2.2.9. Flip-Book

When devising a method of presenting benchmark results we found the following aspects to be of crucial relevance for our purposes. First, it should be possible to navigate the results such that plots are always at the same screen position and have the same dimensions, thereby minimizing the effort to visually compare results. To achieve such a format, we decided to create a flip-book in which each slide presents the results of one experiment. Second, relevant metadata should be displayed right next to the plots. This can include similarities across the runs, but more importantly should highlight the differences. As each user might be interested in different comparisons, we let the user decide which kind of metadata should be shown. Third, it should be easy to select only the benchmarks of interest in order to keep the number of plots small. This is already handled by the filtering provided by `git-annex` views as described in Section 2.2.7. As an underlying technology for programmatically creating `HTML` slides we use

[12] https://git-annex.branchable.com

[13] https://gin.g-node.org

[14] https://github.com/INM-6/beNNch-plot

TABLE 1 | Shorthand notation and description of NEST versions used in this work.

Shorthand notation of NEST version	Description
2.20.2	Official 2.20.2 release (Fardet et al., 2021)
3.0rc	Release candidate for 3.0
3.0rc+ShrinkBuff	3.0rc plus shrinking MPI buffers
3.0rc+ShrinkBuff+SpikeComp	3.0rc+ShrinkBuff plus spike compression
3.0	Official 3.0 release (Hahne et al., 2021) = 3.0rc+ShrinkBuff+SpikeComp plus neuronal input buffers with multiple channels

jupyter notebooks[15] in conjunction with the open source HTML presentation framework reveal.js[16]. An exemplary flip-book containing the NEST performance results described in this work is published alongside the beNNch repository[17]. By respecting these considerations, our proposed solution offers a way of sharing benchmarking insights between researchers that is both scalable and flexible.

2.2.10. Exchanging External Modules

beNNch is written in a modular fashion; as such, it is possible to exchange certain modules without compromising the functionality of the framework. In particular, the "external modules" (see **Figure 2**) are implemented such that an exchange is straight-forward to implement. This section presents a recipe to exchange the "job execution" module, i.e., the simulator, along with necessary changes in "data generation" and "model" that follow:

First, the simulator to be substituted instead of NEST needs to be properly installed. Builder—our implementation of the "software deployment" module—provides the flexibility to install any software as well as make it loadable via a module system. Thus, a plan file specifying dependencies as well as source code location and installation flags needs to be created for the new simulator.

Second, models compatible with the new simulator need to be added. On the framework side, the execute commands may need to be adapted. Required adaptations to the models are the same as for PyNEST models and are described in Section 2.2.4.

Third, the instrumentation needs to be changed. As NEST has built-in instrumentation, only superficial timing measurements are necessary on the model level. Depending on the new simulator's existing ability to measure performance, timing measurements might need to be implemented on the simulator or model level. If different measurements than implemented are of interest, a simple addition to an existing list in beNNch suffices to add the recorded data to the csv-format result file.

2.3. Using beNNch for Simulator Development

For the development of simulation software with the goal to optimize its performance, it is vital to focus efforts on those parts of the simulation loop that take the most time to complete. Benchmarking can help in identifying performance bottlenecks and testing the effect of algorithmic adaptations. However, the dimensions of benchmarking identified in **Figure 1** make this difficult: to guarantee that observed differences in performance are caused by changes in the simulator code, many variables need to be controlled for, such as hardware and software configurations as well as simulator versions. General-purpose simulators also need to be tested with respect to different settings and applications to ensure that a performance improvement in one case does not lead to a crucial decline in another case. Neuronal network simulators are one such example as they should exhibit reasonable performance for a variety of different models with different resource demands. A systematic assessment of the scaling performance covering the relevant scenarios is therefore a substantial component of the iterative simulator development.

beNNch, as an implementation of the workflow outlined in Section 2.1, provides a platform to handle the complexity of benchmarking while staying configurable on a low level. The following suggests how beNNch can support the process of detecting and tackling performance issues of a simulator. In a first step, exploration is necessary to identify the performance bottlenecks of the current version of the simulation engine. As many software and model parameters need to be explored, the centralized location of configuration parameters built into beNNch helps in maintaining an overview of conducted experiments. Neuronal network simulations can usually be decomposed into separate stages, such as neuronal update and spike communication. The instrumentation and visualization of these stages is part of beNNch and points the researcher to the respective sections in the code. If a potential bottleneck for a certain model is identified, tests with other models provide the basis for deciding whether these are model- and scale-specific or are present across models, hinting at long-reaching issues of the simulator. beNNch's native support for handling the benchmarking of multiple models alleviates the researchers from operating a different code base for every model. In the process of solving the simulator issue, running further benchmarks and directly comparing new results can assist in deciding which adaptations bear fruit. The standardized visualization tools of beNNch support spotting differences in performance plots. Finally, an ongoing development of a neuronal network simulator should respect the value of insights gained by resource-intensive benchmarks. To this end, beNNch implements a decentralized storage of standardized performance results. In addition to preserving information for the long term, this also helps in communicating between researchers working on the simulator's development.

2.3.1. Use Case: NEST Development

This section illustrates the relevance of performance benchmarks for the development of neuronal network simulators with the

[15] https://jupyter.org

[16] https://github.com/hakimel/reveal.js

[17] https://inm-6.github.io/beNNch

example of recent changes to the NEST code base; for historical context see Section 4.1.1. We use beNNch to outline crucial steps of the development from the release candidate NEST 3.0rc to the final NEST 3.0 and also discuss improvements compared to the latest NEST 2 version (NEST 2.20.2, Fardet et al., 2021). **Table 1** summarizes the NEST versions employed in this study.

Regarding the dimensions of HPC benchmarking in **Figure 1**, this use case primarily addresses the "Simulators" dimension by testing different NEST versions and the "Models and parameters" dimension by testing different network models; the approach can be extended similarly to the other dimensions.

Our starting point is the weak-scaling experiments of the HPC-benchmark model (Jordan et al., 2018); the times for network construction and state propagation as well as the memory usage remain almost constant with the newly introduced 5g kernel (see their **Figures 7**, **8**). **Figure 4** shows similar benchmarks of the same network model conducted with beNNch using the release candidate in **Figure 4A** and the final release in **Figure 4B**. The graph design used here corresponds to the one used in the flip-book format by the framework. A flip-book version of the results shown in this work can be accessed via the GitHub Pages instance of the beNNch repository[17]. While the release candidate in **Figure 4A** exhibits growing state-propagation times when increasing the number of nodes, network-construction times stay constant and are, for $T_{\text{model}} = 1\,\text{s}$, small, making up less than 10% of the total simulation time. The phases "delivery" and "communication" both contribute significantly to the state-propagation time. Jordan et al. (2018) report real-time factors of about 500 (e.g., their **Figure 7C**) in contrast to values smaller than 40 shown here and their simulations are by far dominated by the delivery phase (see their Figure 12). A comparison of our data and the data of Jordan et al. (2018) is not straightforward due to the inherent complexity of benchmarking and we will here emphasize a few concurring aspects: first, Jordan et al. (2018) run their benchmarks on the dedicated supercomputers JUQUEEN (Jülich Supercomputing Centre, 2015) and K Computer (Miyazaki et al., 2012) while our benchmarks use the recent cluster JURECA-DC (Thörnig and von St. Vieth, 2021). Each compute node of the BlueGene/Q system JUQUEEN is equipped with a 16-core IBM PowerPC A2 processor running at 1.6 GHz and each node of the K Computer has an 8-core SPARC64 VIIIfx processor operating at 2 GHz; both systems provide 16 GB RAM per node. In contrast, the JURECA-DC cluster employs compute nodes consisting of two sockets, each housing a 64-core AMD EPYC Rome 7742 processor clocked at 2.2 GHz, that are equipped with 512 GB of DDR4 RAM. Here, nodes are connected via an InfiniBand HDR100/HDR network. Second, Jordan et al. (2018) use 1 MPI process per node and 8 threads per process while our simulations are performed throughout this study with 8 MPI processes per node and 16 threads per process. Third, Jordan et al. (2018) simulate 18,000 neurons per MPI process while we only simulate 11, 250 neurons per process. This list of differences is not complete and only aims to illustrate that potential discrepancies in benchmarking results may be explained by differences in hardware, software, simulation and model configuration, and other aspects exemplified in **Figure 1**.

Having demonstrated that beNNch can perform weak-scaling experiments of the HPC-benchmark model as done in previous publications, we next turn to strong-scaling benchmarks of the multi-area model (Schmidt et al., 2018a). To fulfill the memory requirements of the model, at least three compute nodes of JURECA-DC are needed; here, we choose to demonstrate the scaling on four to 32 nodes. Initially, we compare the latest NEST 2 version (**Figure 5A**) with the release candidate for NEST 3.0 (**Figure 5B**). The improved parameter handling implemented in NEST 3.0rc reduces the network-construction time. However, the communication phase here largely dominates state propagation in both NEST versions shown; both use the original 5g kernel. Previous simulations of the HPC-benchmark model have not identified the communication phase as a bottleneck (Jordan et al., 2018, Figure 12). Communication only becomes an issue when then smallest delay in the network is of the same order as the computation step size because NEST uses the smallest delay as the communication interval for MPI. While the HPC-benchmark model uses 1.5 ms for all connections—which is a good estimate for inter-area connections—the multi-area model and microcircuit use distributed delays with a lower bound of 0.1 ms leading to a fifteen-fold increase in the number MPI communication steps.

The following identifies and eliminates the main cause for the large communication time in case of the multi-area model, thus introducing the first out of three performance-improving developments applied to NEST 3.0rc. Cross-node communication, handled in NEST by MPI, needs to strike a balance between the amount of messages to transfer and the size of each message. The size of the MPI buffer limits the amount of data that fits into a single message, and is therefore the main parameter controlling this balance. Ideally, each buffer would fit exactly the right amount of information by storing all spikes of the process relevant for the respective communication step. Due to overhead attached to operating on additional vectors, a scheme in which the buffer size adapts precisely for each MPI process for each communication step can be highly inefficient. Therefore, in cases where communication is roughly homogeneous, it is advantageous to keep the exchanged buffer between all processes the same size, as is implemented in NEST 3.0rc. While buffer sizes are constant across processes, NEST does adapt them over time to minimize the number of MPI communications. Concretely, whenever the spike information that a process needs to send exceeds what fits into one buffer, the buffer size for the next communication step is increased. However, the original 5g kernel of NEST does not shrink buffer sizes. In networks such as the multi-area model, the firing is not stationary over time; transients of high activity propagate through the network (Schmidt et al., 2018a). In general, startup transients may cause high spike rates only in the beginning of a simulation unless the network is carefully initialized (Rhodes et al., 2019). If the rates decrease, the spiking information becomes much smaller than the available space in the MPI buffer. Consequently, the original 5g kernel preserves unnecessarily large buffer sizes which results in the communication of useless data. To address

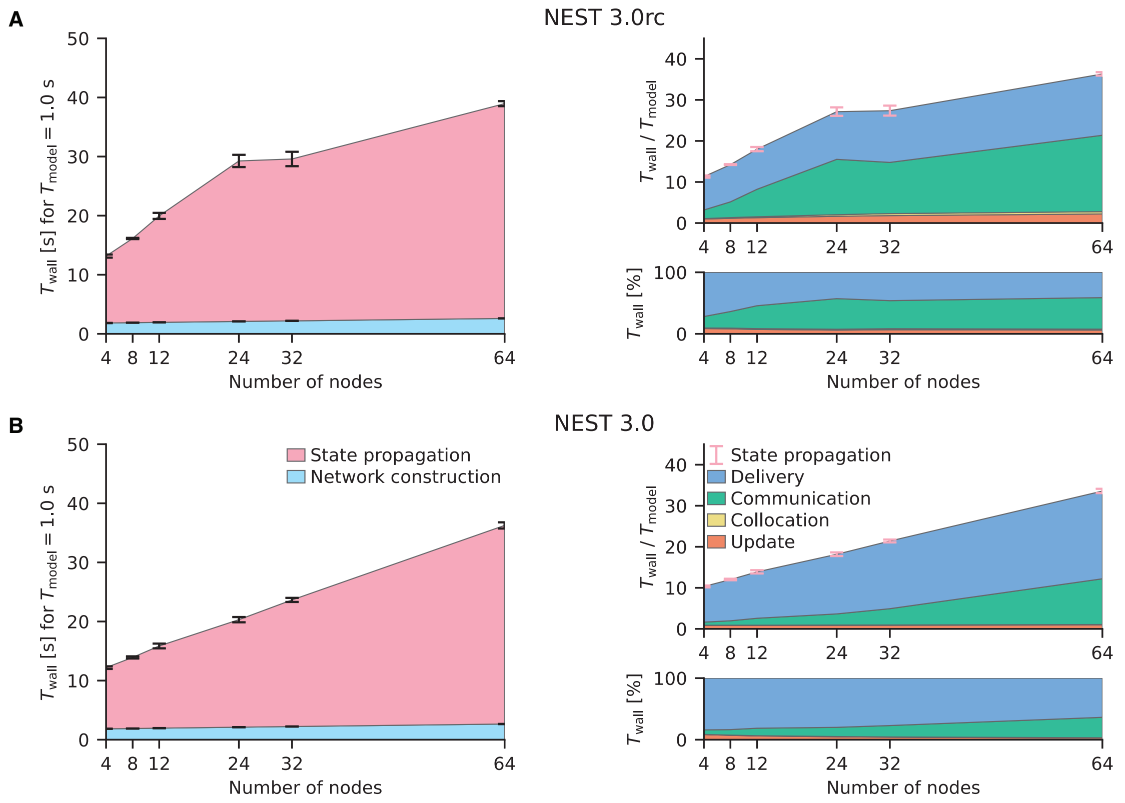


FIGURE 4 | Weak-scaling performance of the HPC-benchmark model on JURECA-DC. **(A)** NEST 3.0rc. The left graph shows the absolute wall-clock time T_{wall} measured with Python-level timers for both network construction and state propagation [legend in **(B)**]; the model time is $T_{\mathrm{model}} = 1$ s. Error bars indicate variability across three simulation repeats with different random seeds. The top right graph displays the real-time factor defined as wall-clock time normalized by the model time. Built-in timers resolve four different phases of the state propagation [legend in **(B)**]: update, collocation, communication, and delivery. Pink error bars show the same variability of state propagation as the left graph. The lower right graph shows the relative contribution of these phases to the state-propagation time. Same colors used for phases as in **Figure 3**. **(B)** NEST 3.0. Same display as **(A)**.

this issue, a mechanism for automatically shrinking the buffer sizes has been introduced. For details see Section 4.1.2. The release candidate with the implementation of shrinking MPI buffers (NEST 3.0rc+ShrinkBuff) approximately halves the time spent in the communication phase compared to the original implementation (compare **Figures 5B,C**).

Next, we assess the strong-scaling performance of the microcircuit model (Potjans and Diesmann, 2014). The model size is similar to the size of one of the 32 areas of the multi-area model. The microcircuit therefore requires fewer resources. We show results of the model run on one to six compute nodes; for a detailed analysis of NEST's thread scaling performance on the example of this model refer to Kurth et al. (2021). Using NEST 3.0rc, the microcircuit is simulated faster than the HPC-benchmark and the multi-area models and achieves approximately real time ($T_{\mathrm{wall}}/T_{\mathrm{model}} \approx 1$, **Figure 6A**). The finer resolution of the vertical axis of the top-right graph reveals a small gap between the state propagation measured with Python timers and the sum of the phases timed on the C++ level which is not visible for the other models. The state-propagation time of the microcircuit is also dominated by the communication phase similarly to the respective benchmarks with the multi-area model (**Figure 5B**) and even increases with the number of nodes used. However, shrinking MPI buffers does not reduce communication significantly (data not shown), indicating that we face a different bottleneck with the microcircuit model. With on the order of 10^3 outgoing connections per neuron, a single neuron of this model has multiple targets on each MPI process and, in particular, on multiple threads of a given process. Since the 5g kernel is designed to send out a separate copy of a neuron's spiking information to each target thread, multiple copies of identical information about the activity of a presynaptic neuron may be sent to the same process, causing unnecessary communication load. To tackle this, we devise a spike compression algorithm which only requires transmitting the spiking information once to each MPI process where it is locally routed to the receiving threads. For details see Section 4.1.3. This algorithm leads to a significant reduction in communication time for the microcircuit model (compare **Figures 6A,B**).

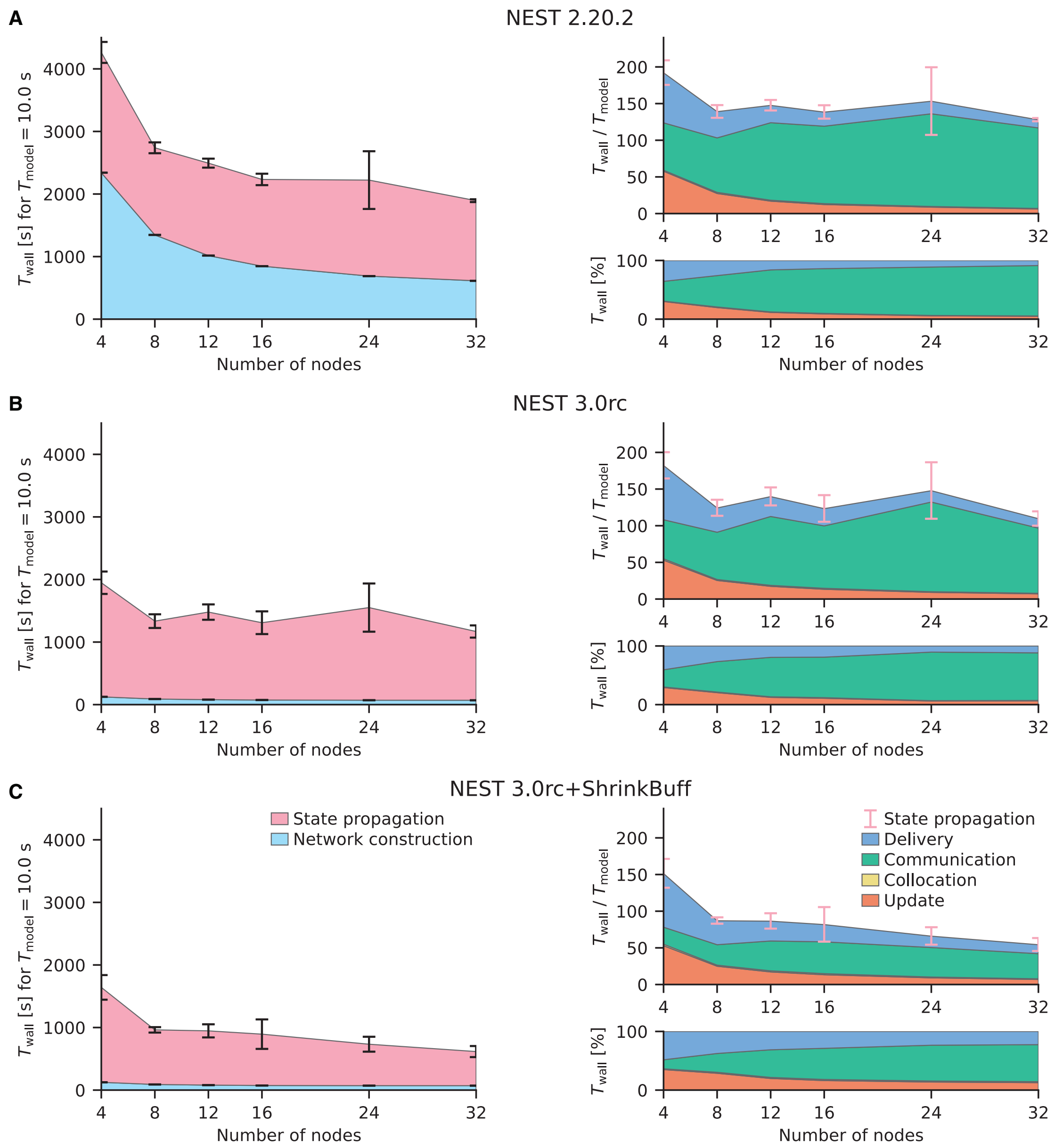


FIGURE 5 | Strong-scaling performance of the multi-area model on JURECA-DC. Same display as in **Figure 4**. The multi-area model is simulated in its meta-stable state leading to a high amount of spikes that are communicated. The model time is $T_{model} = 10$ s. Simulations are repeated for 10 different random seeds. **(A)** NEST 2.20.2 (latest NEST 2 release). **(B)** NEST 3.0 release candidate. **(C)** NEST 3.0 release candidate with shrinking MPI buffers.

The microcircuit model easily fits within the main memory of one compute node of JURECA-DC. Due to the simplicity of the employed model neurons and the absence of synaptic plasticity mechanisms, the network model causes little workload during update and delivery in a strong-scaling experiment—real-time simulation is already possible with a single compute node. Consequently, communication naturally starts to dominate the state-propagation time at a few compute nodes even with the spike-compression optimization described above. While increasing the number of compute nodes from one to two still results in a fair reduction of state-propagation time, scaling is already sublinear, and increasing the number of compute nodes further hardly results in further improvement. Therefore, simulation phases other than the so far discussed communication become important if the objective of the optimizations is, for example, achieving real-time performance with even fewer resources. In the following we highlight an algorithm adaptation that concentrates on the update phase. A redesign of the neuronal input buffers prevents neurons from retrieving the input values for different channels, for example, excitatory and

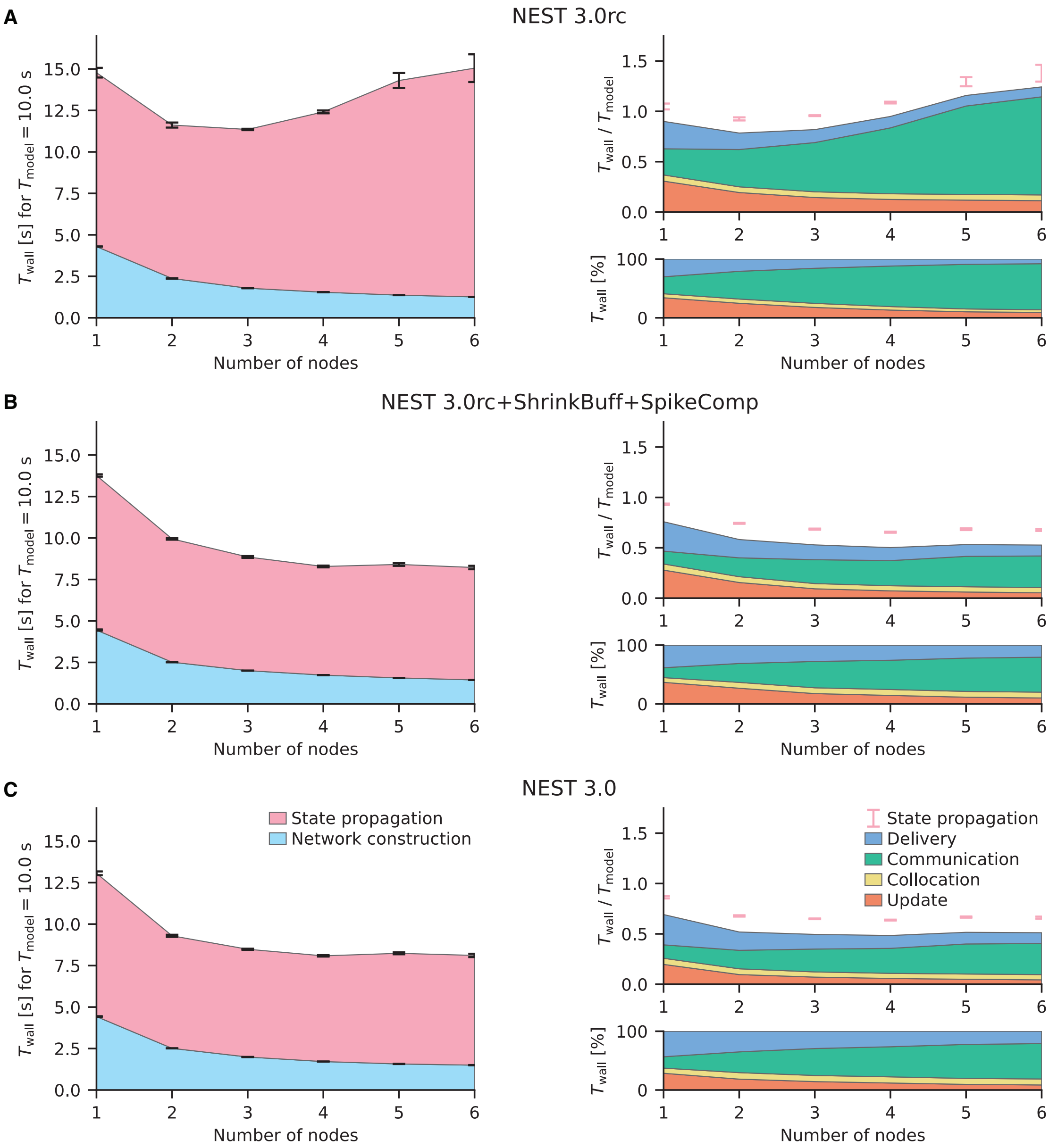


FIGURE 6 | Strong-scaling performance of the microcircuit model on JURECA-DC. Same display as in **Figure 4**. The model time is $T_{\mathrm{model}} = 10$ s. Simulations are repeated for 10 different random seeds. **(A)** NEST 3.0 release candidate. **(B)** NEST 3.0 release candidate with spike compression and shrinking MPI buffers. **(C)** NEST 3.0.

inhibitory, from separate locations in memory. Thereby, the cache can be better utilized during neuronal updates. Instead of maintaining separate buffers for the input channels as in the original 5g kernel, neurons maintain a single buffer with all inputs for a particular simulation time step stored contiguously in memory. For details see Section 4.1.4. This adaptation is most effective for network models with short minimum synaptic delays; both the microcircuit and the multi-area model use 0.1 ms. **Figure 6C** shows the resulting decrease in update time for few compute nodes.

In summary, the analysis with beNNch exposes the communication phase as a major performance bottleneck in microcircuit and multi-area model simulations with the release candidate NEST 3.0rc. The underlying problem is, however, a different one for each of the two models, and they are rectified with different adaptations to the code: the shrinking MPI buffers (Section 4.1.2) improve the performance of the multi-area model while spike compression (Section 4.1.3) increases simulation speed of the microcircuit model. Notably, none of the adaptations introduce performance regressions for the respective other model (data not shown). In addition, the update phase is improved by introducing neuronal input buffers with multiple channels (Section 4.1.4). Returning to the HPC-benchmark model, **Figure 4B** shows that the kernel adaptations are not detrimental to the originally tested model; the overall state-propagation time is preserved with the final NEST 3.0 release. However, the reduced communication and update times here come at the cost of increased delivery times due to an additional indirection introduced with spike compression. Ongoing work targets the delivery phase (Pronold et al., 2021, 2022) and gives a perspective for performance improvements in future NEST releases.

3. DISCUSSION

Benchmarking studies in the field of neuronal network simulations are often hard to reproduce and compare. To overcome this problem, we propose a unified and modular workflow for defining, running, and analyzing benchmark simulations. We identify five dimensions spanning the space of the benchmarking endeavor, and work out their specific challenges: hardware configuration, software configuration, simulators, models and parameters, and researcher communication. The benchmarking concept developed in this study encompasses all five dimensions and proposes solutions for the posed challenges in the form of self-contained and interacting modules. Each module contributes to one of the main workflow segments: configuration and preparation, actual benchmarking, data- and metadata handling, and data presentation. As a proof of concept, we provide a reference implementation of the framework (beNNch), describe the concrete underlying technologies, and apply it to a specific use case: assessing and comparing the performance of different versions of the neuronal network simulator NEST for three different network models. The reference implementation goes beyond existing benchmarking environment software such as JUBE: it adds an interface to models, installs and deploys simulation software, automates data and metadata annotation, and implements storage and presentation of results. The use case illustrates how the framework helps to focus simulator development by detecting performance bottlenecks, and demonstrates the relevance of an accessible and comprehensive benchmarking setup. The software is ready to use, not only for developers of simulation technology, but also for researchers seeking to find optimal performance configurations for their models.

The proposed workflow is generic and not restricted to benchmarking neuronal network simulations with NEST. The reference implementation, however, still faces limitations and open problems. First, it is *a priori* unclear what parameters, configurations or external influences may possibly contribute to differences in the performance of complex software systems such as simulation engines. beNNch seeks to address this problem by employing a metadata archive which—in addition to the selection of metadata directly attached to the performance results—tracks further metadata that are seemingly insignificant at the time of simulation but may become relevant in future investigations. Exhaustiveness, however, can not be claimed. For the exploration and presentation of benchmarking data, the reference implementation uses metadata to filter benchmark results and to highlight differences in a flip-book format. However, even if all relevant metadata were tracked, selecting sensible metadata keys for filtering and highlighting is a hard problem. In the current implementation, this requires knowledge about existing results and, therefore, human input. Future solutions could, for example, categorize and hierarchically structure metadata keys to facilitate and semi-automatize these steps. Second, the network model specifications, expressed in the PyNEST set of commands for the Python language, require adaptations to interface with the benchmarking framework. These include accepting parameters passed by JUBE benchmarking files, adjusting the model specification to work with different versions of the simulation engine, and storing recorded metadata and performance measures such as the duration of simulation phases. At the moment, it is a manual task to keep the benchmarking model version up to date with the original model version, which is error prone. We use rigorous version control of the code, automatic checking for errors (via exceptions), and continuous testing for correct simulation outcome to reduce the risk of errors. This strategy could be automatized further in the future by finding methods to automatically inject respective instrumentation into the executable model descriptions. To mitigate the additional overhead, we keep the necessary changes as minimal as possible, thereby lowering the entry barrier for new models. Third, the reference implementation makes concrete choices on the employed tools. Alternatives, however, may be viable. For example, the required software for the simulations is installed with Builder which can be integrated with other package management systems or replaced by a different solution. Our strategy exploits the native software environment available on a compute cluster which is typically specifically configured for the underlying hardware. An alternative is to use containerized systems such as Docker[18] or Singularity[19]. Replacing NEST by a different simulator requires adapting the model implementations. Expressing the models in the simulator-independent language PyNN (Davison et al., 2009) instead of PyNEST would avoid this. However, additional layers of complexity such as PyNN may have an impact on performance, making it more challenging to pinpoint bottlenecks in the simulator backend. JUBE as an environment to manage jobs on compute clusters could be substituted by tools such as ecFlow[20], AiiDA[21] (Huber et al., 2020), or cylc (Oliver et al., 2021). Further, one could replace git-annex with, e.g., DataLad[22] which is based on the same technology but extends its functionality and provides slightly different metadata handling. The flip-book-style presentation of results could also

[18] https://www.docker.com
[19] https://sylabs.io
[20] https://confluence.ecmwf.int/display/ECFLOW
[21] https://www.aiida.net
[22] https://www.datalad.org

be replaced or supplemented with other approaches, for example an automatically generated overview figure showing results from multiple benchmarking runs together, similar to **Figures 4–6** in this article. Furthermore, tools like Rust Compiler Performance Monitoring and Benchmarking[23] or Sacred[24] cover multiple aspects of the workflow and can be a source of inspiration for further development of beNNch. Fourth, beNNch presently focuses on a single performance measure: the time-to-solution. However, different performance aspects, such as energy-to-solution and memory consumption, may also be of interest. Energy-to-solution, for example, combines power consumption and time-to-solution. Monitoring both power consumption and time-to-solution enables researchers to determine an optimal number of compute nodes balancing speed and energy consumption (van Albada et al., 2018). The memory consumption of the simulation dictates, for instance, the smallest number of nodes required to simulate a network of a given size, or the largest network size possible to simulate on a given machine. Reducing memory requirements was a major driving force behind the improvements to the NEST kernel (Helias et al., 2012; Kunkel et al., 2012, 2014; Jordan et al., 2018) in the past decade. The spike compression introduced here reduces the time-to-solution (communication phase, **Figures 4**, **6**). However, this code change directly affects the memory consumption. Assuming that the number of postsynaptic targets per neuron is fixed, the memory overhead is negligible if the number of MPI processes is small. But in the limit of a large number of MPI processes, i.e., when each neuron has at most one target on each process, the effective size of each synapse is increased by 8 byte. In this limit, users thus are encouraged to actively deactivate the "spike compression" feature. This example illustrates that performance optimizations often have to find a balance between acceptable solutions for different measures. Due to its modular structure, beNNch is ready to include further performance measures.

To achieve long term sustainability, organized and openly available communication on development is essential. Adhering to this guideline, we have developed beNNch as an open source software project from the start, making use of a public issue tracker, suggestions via pull requests, public code reviews, and detailed documentation. This approach facilitates constructive communication between users and developers which enables a targeted progression of the framework by demand. While the concrete application of NEST benchmarks of neuronal network models shaped our specific implementations, the modular structure allows for adaptation to other use cases. In certain domains of software development, it is already common practice to verify each code change on the basis of syntax, results, and other unit tests. The proposed automated approach to execute performance benchmarks creates the opportunity to integrate an aspect of validation directly into the development cycle. This way, performance regressions of algorithm adaptations are immediately exposed, while positive effects can readily be demonstrated. For high-performance software, however, comprehensive checks for scaling performance are particularly costly because they require compute time on state-of-the-art clusters and supercomputers. Therefore, it is important to conduct the performance benchmarks purposefully and with care. By organizing benchmarking results and keeping track of metadata, beNNch helps to avoid redundant benchmark repeats and instead encourages a direct comparison with previous results.

It has long been recognized that software development in science underlies different constraints and needs to fulfill different requirements as compared to industry (Diesmann and Gewaltig, 2002). The software crisis in neuroscience at the beginning of the century led to the foundation of the International Neuroinformatics Coordinating Facility (INCF) in 2005. A first INCF report in 2006 addresses the software challenges of large-scale modeling in neuroscience (INCF Secretariat et al., 2018) and recommends establishing a common set of benchmark models and a corresponding framework for assessing accuracy and efficiency. Furthermore, the report advocates benchmarking neuroscientifically relevant published models rather than network models constructed specifically for the purpose of benchmarking only. In 2007, the community made a first effort in verifying simulation codes by using a number of simple network models (Brette et al., 2007). Executable model descriptions are, in part, already expressed in the simulator independent language PyNN (Davison et al., 2009), but there is no support by a common benchmarking framework, and a focus is set on correctness rather than computational efficiency. The emerging field of Research Software Engineering (RSE) is studying how, in the scientific setting, reliable and sustainable software can be developed, developers can be educated for this purpose, and science organizations and politics can be made aware of its strategic relevance (Manifesto[25] and Akhmerov et al., 2019). Obvious differences to software engineering in the industrial setting include research code being developed by scientists rather than experienced software developers, the time-constrained and thesis-bound nature of scientific projects, and the continuous integration of new contributors into the development process. Our study contributes to RSE conceptually by identifying the dimensions of benchmarking simulation technology and proposing a general workflow capable of coping with the complexity, and practically by developing a reference implementation of a benchmarking framework which can be used to test and refine the concepts. It is too early to tell quantitatively whether the benchmarking framework improves the collaboration in a joint project and the communication between researchers in the community.

The proposed framework enables benchmarking of research software to evolve from one-off tasks of individual researchers to a collaborative routine effort, thereby increasing the benchmarking capacity and reducing its susceptibility to errors. Making beNNch accessible to the community as an open-source software puts the concept to the test. We are looking forward to learn how the current implementation of the framework's components are received and adapted to other applications. Due to the conceptual foundation and modular

[23] https://github.com/rust-lang/rustc-perf

[24] https://github.com/IDSIA/sacred

[25] https://www.software.ac.uk/about/manifesto

structure, we hope that beNNch can adjust to future requirements and ultimately help increase the complexity and explanatory scope of brain models. The benchmarking concepts developed in this work are not limited to neuroscience and can be transferred to other types of simulation research.

4. MATERIALS AND METHODS

4.1. NEST Developments

4.1.1. Brief History of NEST

The series of NEST 2.X releases includes enhancements, bug fixes, and contributions to maintenance with only marginal effects on the PyNEST user interface (Eppler et al., 2009). Performance-related updates to the simulation kernel are accomplished under the hood. The 3g kernel (Helias et al., 2012; Kunkel et al., 2012) is in use from NEST 2.2.0 (van Albada et al., 2015a). NEST 2.12.0 (Kunkel et al., 2017) introduces the 4g kernel (Kunkel et al., 2014) which implements novel data structures allowing for an efficient and flexible representation of sparse network connectivity on highly distributed computing systems such as supercomputers. The 5g kernel (Jordan et al., 2018) in NEST 2.16.0 (Linssen et al., 2018) continues this direction of development toward an optimal usage of HPC systems for large-scale simulations by disentangling the memory usage per compute node from the total network size. The transition from NEST 2 to NEST 3 corresponds to a refurbishment of the simulator code which also breaks the backwards compatibility of the user interface. While improved high-level functionality and parameter handling are the primary goals of this transition, the 5g kernel is supposed to remain. In the past, performance changes due to kernel updates have been predominantly assessed using the HPC-benchmark model. The performance of the NEST 3.0 release candidate ("3.0rc"), however, is in addition evaluated with the microcircuit and multi-area model which exhibit a more complex connectivity structure and a different distribution of synaptic delays. In this way, so far undetected performance bottlenecks are discovered and subsequently resolved, leading to the official release NEST 3.0 (Hahne et al., 2021).

4.1.2. Shrinking MPI Buffers

Motivated by reducing the memory footprint of the postsynaptic infrastructure—necessary to deliver spikes to their process-local targets—the 5g kernel of NEST 3.0rc prepares a separate part of the MPI send buffer for each target process and only includes the relevant spikes. Thus, each process is responsible for sending the spikes of its neurons to all target processes for each communication time step. NEST 3.0rc implements a homogeneous buffer size across processes to avoid overhead introduced by variable buffer sizes; in the latter case, each process would need to complete two rounds of communication, one for transmitting the size, and one for the actual spiking information. Similarly, transmitting a certain amount of information via sending MPI buffers is more efficient when fewer buffers—each carrying more information—are sent. NEST 3.0rc consequently aims to reduce the number of needed MPI buffers to only 1 by dynamically increasing the global buffer size whenever a process cannot fit all spikes into the buffer. Specifically, every time more than a single buffer needs to be sent by a process, NEST increases the buffer size of the following communication step by a factor of 1.5. In this scheme, a reduction of buffer sizes is not implemented, meaning that buffer sizes can only increase or stay constant. The kernel of NEST 3.0rc+ShrinkBuff addresses this by introducing the following algorithm for shrinking the global buffer size. In each communication round in which only a single send buffer is required, the buffer for the following round decreases by a factor of 1.1. Even though this implementation leads to an oscillation of buffer size for constant spiking activity, tests show that this simple mechanism only introduces negligible cost while being robust.

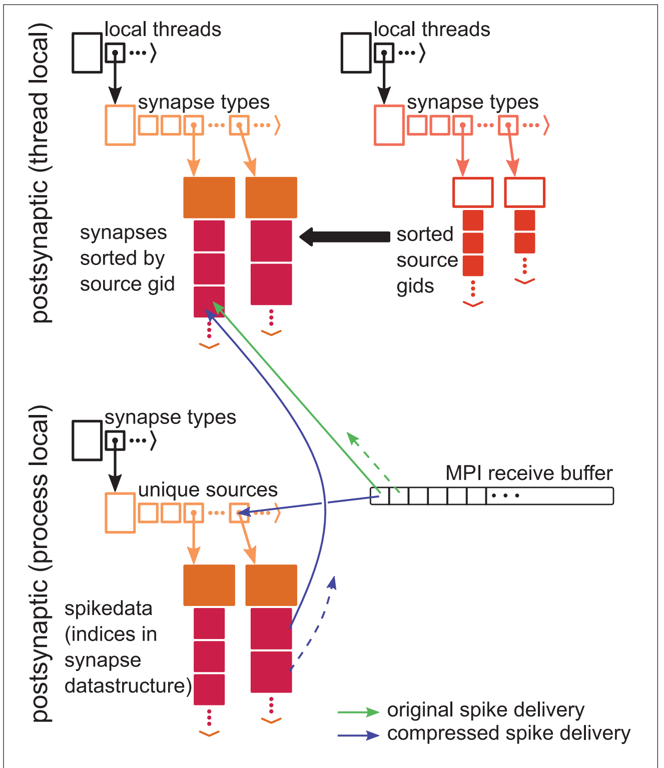

postsynaptic (thread local)
local threads
synapse types
synapses sorted by source gid
local threads
synapse types
sorted source gids
postsynaptic (process local)
synapse types
unique sources
MPI receive buffer
spikedata (indices in synapse datastructure)
original spike delivery
compressed spike delivery

FIGURE 7 | Spike compression adds an additional indirection to post-synaptic spike routing. Green arrow denotes original spike delivery introduced with the 5g kernel (Jordan et al., 2018, same display as their **Figure 4A**). Blue arrow illustrates additional indirection with compressed spike delivery. Dashed arrows indicate spikes from the same source neuron with target on a different thread.

4.1.3. Spike Compression

NEST's 5g kernel (Jordan et al., 2018) introduces a two-tier connection infrastructure for routing spikes. The connection infrastructure consists of data structures on the presynaptic side (the MPI process of the sending neuron) and the postsynaptic side (the MPI process of the receiving neuron), cf. Section 2.2.3. Communication of spikes is organized as follows: when a neuron becomes active, its targets are retrieved from the local presynaptic data structure. These targets represent indices of synapses in the "thread-local" post-synaptic data structure through which spikes are routed to the target neurons. The presynaptic side then creates MPI buffers containing collections of such indices which are subsequently communicated to the postsynaptic side via the MPI `Alltoall` function. To deliver spikes on the postsynaptic side, each thread uses the received spikes to index its local postsynaptic data structure and register a spike in the corresponding synapse (**Figure 7**, "original spike delivery"). If a presynaptic neuron has targets on multiple threads of a process, it hence has to send multiple spikes, i.e., indices in different thread-local data structures, to the target process.

Here, we adapt this infrastructure as follows. We introduce an additional data structure on the post-synaptic side which is shared across threads ("process local"). This data structure contains, arranged by source neuron, the indices of all process-local synapses. While the pre-synaptic part of communicating spikes remains essentially identical, the postsynaptic part incurs an additional indirection: each entry in the MPI receive buffer now represents an index in the new process-local postsynaptic data structure. Using this index, each thread can retrieve the indices of thread-local targets, to which it can then deliver spikes as previously (**Figure 7**, "compressed spike delivery"; note that the origin of the dashed arrow changes). In contrast to the previous implementation, each presynaptic neuron thus sends at most one spike to each process.

In `NEST 3.0`, spike compression is turned on by default, but the previous 5g behavior can be recovered by setting:

```
nest.SetKernelStatus({"use_compressed_spikes": False})
```

4.1.4. Neuronal Input Buffers With Multiple Channels

Simulation technology for spiking neuronal networks requires techniques to handle synaptic transmission delays. The reference simulation code (Section 2.2.2) follows a globally time-driven approach: spikes are constrained to a time grid and regularly exchanged between MPI processes using collective communication. The time grid defines the simulation time step for neuronal updates, whereas the minimum synaptic delay d_{min} in the network model defines the communication interval (Morrison et al., 2005a), which comprises at least one simulation time step. In the microcircuit model and the multi-area model used in this study the minimum delay is 0.1 ms (i.e., $d_{\text{min}} = 1$ simulation time step) and in the HPC-benchmark model it is 1.5 ms (i.e., $d_{\text{min}} = 15$ simulation time steps). While communication and subsequent process-local delivery of spikes define interaction points between neurons, within a communication interval each neuron independently updates its state for all time steps without interruption. Hence, a simulation cycle of neuronal update, spike-communication, and spike-delivery phase propagates the network state by one communication interval, but within each update phase neurons propagate their state in potentially shorter simulation time steps. All spikes emitted by the process-local neurons during such an update are immediately transmitted during the subsequent communication and on the receiver side delivered to their target neurons. Hence, to account for synaptic delays, neurons cannot immediately integrate the incoming spikes into their dynamics, but they need to buffer the inputs until the corresponding delays elapse. To this end, neurons maintain input buffers of $d_{\text{min}}+d_{\text{max}}$ time slots, where d_{max} denotes the maximum synaptic delay in the network (**Figure 8A**). The relative time origin S defining the time slots from which to retrieve inputs during update and the time slots for adding inputs during spike delivery advances by d_{min} time slots at the end of every simulation cycle. In this way, the time slots that were read and reset during the update of the current cycle become available for adding new inputs during the spike delivery in the next cycle. For cases where the communication interval comprises multiple simulation time steps (e.g., HPC-benchmark model), input retrieval is most costly for the first step as the corresponding buffer entry needs to be loaded into cache, but then benefits from the already cached subsequent buffer entries in the subsequent steps of the communication interval. If, however, the communication interval consists of only one simulation step due to a very short minimal synaptic delay (e.g., microcircuit and multi-area model), input retrieval is costly for every simulation step as each step is handled in a separate simulation cycle, and hence caching of relevant input buffer entries is rendered ineffective during the spike communication and delivery that follows each neuronal update phase.

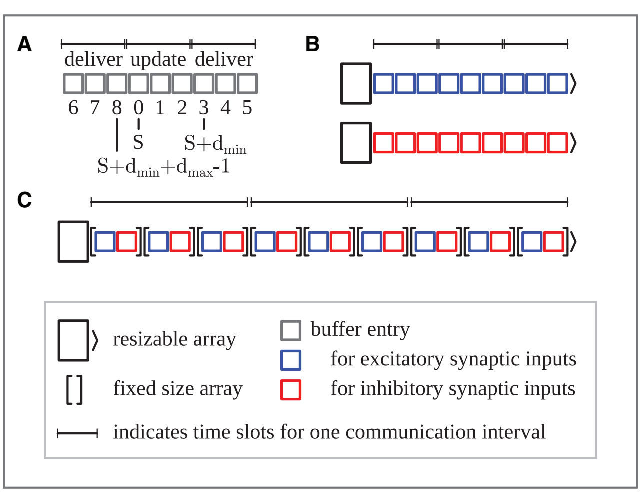

A
deliver update deliver
6 7 8 0 1 2 3 4 5
S
S+d_min
S+d_min+d_max-1
B
C
resizable array
fixed size array
buffer entry
for excitatory synaptic inputs
for inhibitory synaptic inputs
indicates time slots for one communication interval

FIGURE 8 | Neuronal input buffers accounting for synaptic delays in simulations of spiking neuronal networks. **(A)** Structure of neuronal input buffers assuming a minimum synaptic delay d_{min} of three simulation time steps and a maximum delay $d_{\text{max}} = 2d_{\text{min}}$. To buffer upcoming inputs during simulation a total buffer size of $d_{\text{min}} + d_{\text{max}}$ time slots is required, which corresponds to three communication intervals of three simulation time steps each. After every spike communication and subsequent spike delivery to local targets, simulation time is advanced, meaning that the relative time origin S of the neuronal input buffers advances by d_{min} time slots with a wrap-around at the buffer end. A pre-calculated and continuously updated look-up table maps the index relative to S to the actual buffer index. Example: The relative time origin S is located at the fourth time slot. Synaptic delays of the inputs of the middle buffer segment elapse with the upcoming three simulation time steps; the neuron integrates these inputs updating its state. Spikes are then communicated and new inputs delivered to the neuron are added to the time slots in the last or first buffer segment depending on the delay, which is at least d_{min} and at most d_{max}. Relative time origin S then advances to the seventh buffer slot (not shown). **(B)** Original neuronal spike buffers for two input channels (e.g., excitatory and inhibitory synaptic inputs). For each channel a separate resizable array buffers the inputs for the upcoming time slots. **(C)** Multi-channel input buffer for two input channels. A single resizable array stores the inputs for the upcoming time slots, where for each time slot a fixed size array holds the inputs sorted by channel.

Most neuron models need to distinguish between input channels to treat the corresponding inputs dynamically differently, as for example, excitatory and inhibitory synaptic inputs causing different post-synaptic responses. The original input-buffer design required a separate resizable array per channel storing the channel's input values per time slot (**Figure 8B**). This entailed retrieval of the input values for a particular time step from separate locations in memory, which amplifies the cache inefficiency during update for network models with short minimum delays described above. To alleviate this issue, the newly introduced input buffer allows storing the input values for multiple channels per time slot contiguously in fixed size arrays in a single resizable array (**Figure 8C**). Thus, neurons now retrieve all input values for a particular time step by accessing subsequent locations in memory in one pass.

DATA AVAILABILITY STATEMENT

The benchmarking framework is publicly available under https://github.com/INM-6/beNNch. Benchmarks for this study were performed with version `1.0` of `beNNch` which is available as a release on GitHub and on Zenodo (https://doi.org/10.5281/zenodo.6092768, Albers et al., 2022). The data sets generated and analyzed for this study as well as the code to reproduce all figures of this paper are available on Zenodo (https://doi.org/10.5281/zenodo.5784633). An exemplary flip-book containing the results shown in this work can be accessed under https://inm-6.github.io/beNNch.

AUTHOR CONTRIBUTIONS

JA, JP, ACK, SBV, KHM, AP, DT, TT, MD, and JS: study design. JA, JP, ACK, SBV, KHM, DT, and JS: implementation of `beNNch`. JA: execution and analysis of benchmarks. JA, JP, SK, and JJ: figures. AP and JA: implementation of shrinking MPI buffers. JJ and JS: implementation of spike compression. SK: implementation of neuronal input buffers with multiple channels. All authors contributed to the writing of the manuscript and approved it for publication.

FUNDING

This project has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreement No. 945539 (Human Brain Project SGA3) and No. 754304 (DEEP-EST); the Helmholtz Association Initiative and Networking Fund under project number SO-092 (Advanced Computing Architectures, ACA); the Joint Lab Supercomputing and Modeling for the Human Brain; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—368482240/GRK2416 and 491111487; the Helmholtz Metadata Collaboration (HMC), an incubator-platform of the Helmholtz Association within the framework of the Information and Data Science strategic initiative, under the funding ZT-I-PF-3-026.

ACKNOWLEDGMENTS

We thank the members of the NEST development community for their contributions to the concepts and implementation of the NEST simulator, and our colleagues in the Simulation and Data Laboratory Neuroscience of the Jülich Supercomputing Centre for continuous collaboration. We gratefully acknowledge the computing time granted by the JARA Vergabegremium and provided on the JARA Partition part of the supercomputer JURECA at Forschungszentrum Jülich (computation grant JINB33). We acknowledge the use of Fenix Infrastructure resources, which are partially funded from the European Union's Horizon 2020 research and innovation programme through the ICEI project under the grant agreement No. 800858.

REFERENCES

Akar, N. A., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor — a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia: IEEE), 274–282.

Akhmerov, A., Cruz, M., Drost, N., Hof, C., Knapen, T., Kuzak, M., et al. (2019). Raising the profile of research software. doi: 10.5281/zenodo.3378572

Albers, J., Pronold, J., Kurth, A. C., Vennemo, S. B., Haghighi, M. K., Patronis, A., et al. (2022). beNNch. Version 1.0. *Zenedo.* doi: 10.5281/zenodo.6092768

Beyeler, M., Carlson, K. D., Chou, T.-S., Dutt, N., and Krichmar, J. L. (2015). "CARLsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks," in *2015 International Joint Conference on Neural Networks (IJCNN)* (Killarney: IEEE).

Bhalla, U., Bilitch, D., and Bower, J. M. (1992). Rallpacks: A set of benchmarks for neuronal simulators. *Trends Neurosci.* 15, 453–458. doi: 10.1016/0166-2236(92)90009-w

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398. doi: 10.1007/s10827-007-0038-6

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 183–208. doi: 10.1023/a:1008925309027

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge: Cambridge University Press.

Chou, T.-S., Kashyap, H. J., Xing, J., Listopad, S., Rounds, E. L., Beyeler, M., et al. (2018). "CARLsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters," in *2018 International Joint Conference on Neural Networks (IJCNN)*, Rio de Janeiro.

Crook, S. M., Davison, A. P., and Plesser, H. E. (2013). "Learning from the past: approaches for reproducibility in computational neuroscience," in *20 Years of Computational Neuroscience*, Springer Series in Computational Neuroscience, ed J. Bower (New York, NY: Springer), 73–102.

Dai, W., and Berleant, D. (2019). "Benchmarking contemporary deep learning hardware and frameworks: a survey of qualitative metrics," in *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)* (Los Angeles, CA).

Davison, A., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.* 2:10. doi: 10.3389/neuro.11.011.2008

Diesmann, M., and Gewaltig, M.-O. (2002). "NEST: an environment for neural systems simulations," in *Forschung und Wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, eds T. Plesser and V. Macho (Göttingen: Ges. für Wiss. Datenverarbeitung), 43–70.

Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The LINPACK benchmark: past, present and future. *Concurr. Comput.* 15, 803–820. doi: 10.1002/cpe.728

Einevoll, G. T., Destexhe, A., Diesmann, M., Grün, S., Jirsa, V., de Kamps, M., et al. (2019). The scientific case for brain simulations. *Neuron* 102, 735–744. doi: 10.1016/j.neuron.2019.03.027

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M. (2009). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2:12. doi: 10.3389/neuro.11.012.2008

Fardet, T., Vennemo, S. B., Mitchell, J., Mork, H., Graber, S., Hahne, J., et al. (2021). NEST 2.20.2, Version 2.20.2. *Zenedo.* doi: 10.5281/zenodo.5242954

Furber, S., Galluppi, F., Temple, S., and Plana, L. (2014). The SpiNNaker Project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., et al. (2015). "The spack package manager," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (ACM).* doi: 10.1145/2807591.2807623

Geimer, M., Hoste, K., and McLay, R. (2014). "Modern scientific software management using EasyBuild and lmod," in *2014 First International Workshop on HPC User Support Tools* (New Orleans, LA,).

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural simulation tool). *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430

Gleeson, P., Cantarelli, M., Marin, B., Quintana, A., Earnshaw, M., Sadeh, S., et al. (2019). Open source brain: a collaborative resource for visualizing, analyzing, simulating, and developing standardized models of neurons and circuits. *Neuron* 103, 395–411.e5. doi: 10.1016/j.neuron.2019.05.019

Golosio, B., Tiddia, G., Luca, C. D., Pastorelli, E., Simula, F., and Paolucci, P. S. (2021). Fast simulations of highly-connected spiking cortical models using GPUs. *Front. Comput. Neurosci.* 15:627620. doi: 10.3389/fncom.2021.627620

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Front. Neuroinform.* 2:8. doi: 10.3389/neuro.11.005.2008

Gutzen, R., von Papen, M., Trensch, G., Quaglio, P., Grün, S., and Denker, M. (2018). Reproducible neural network simulations: statistical methods for model validation on the level of network activity data. *Front. Neuroinform.* 12:90. doi: 10.3389/fninf.2018.00090

Hager, G., and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers, 1st Edn.* New York, NY: CRC Press.

Hahne, J., Diaz, S., Patronis, A., Schenck, W., Peyser, A., Graber, S., et al. (2021). NEST 3.0. *Zenedo.* doi: 10.5281/zenodo.4739103

Helias, M., Kunkel, S., Masumoto, G., Igarashi, J., Eppler, J. M., Ishii, S., et al. (2012). Supercomputers ready for use as discovery machines for neuroscience. *Front. Neuroinform.* 6:26. doi: 10.3389/fninf.2012.00026

Huber, S. P., Zoupanos, S., Uhrin, M., Talirz, L., Kahle, L., Häuselmann, R., et al. (2020). Aiida 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance. *Sci. Data* 7, 1–18. doi: 10.1038/s41597-020-00638-4

Hunter, J. D. (2007). Matplotlib: a 2D graphics environment. *Comput. Sci. Eng.* 9, 90–95. doi: 10.1109/MCSE.2007.55

INCF Secretariat, Djurfeldt, M., and Lansner, A. (2018). "*1st INCF Workshop on Large-Scale Modeling of the Nervous System.*" Stockholm: F1000 Research Limited. doi: 10.7490/F1000RESEARCH.1116028.1

Ippen, T., Eppler, J. M., Plesser, H. E., and Diesmann, M. (2017). Constructing neuronal network models in massively parallel environments. *Front. Neuroinform.* 11:30. doi: 10.3389/fninf.2017.00030

Izhikevich, E. (2003). Simple model of spiking neurons. *IEEE Trans. Neural Netw.* 14, 1569–1572. doi: 10.1109/TNN.2003.820440

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Front. Neuroinform.* 12:2. doi: 10.3389/fninf.2018.00002

Jülich Supercomputing Centre (2015). *JUQUEEN: IBM Blue Gene/Q® Supercomputer System.* Jülich Supercomputing Centre.

Knight, J. C., Komissarov, A., and Nowotny, T. (2021). PyGeNN: A python library for GPU-enhanced neural networks. *Front. Neuroinform.* 15:5. doi: 10.3389/fninf.2021.659005

Knight, J. C., and Nowotny, T. (2018). GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Front. Neurosci.* 12:941. doi: 10.3389/fnins.2018.00941

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci.* 1, 136–142. doi: 10.1038/s43588-020-00022-7

Kunkel, S., Morrison, A., Weidel, P., Eppler, J. M., Sinha, A., Schenck, W., et al. (2017). NEST 2.12.0. *Zenedo.* doi: 10.5281/zenodo.259534

Kunkel, S., Potjans, T. C., Eppler, J. M., Plesser, H. E., Morrison, A., and Diesmann, M. (2012). Meeting the memory challenges of brain-scale simulation. *Front. Neuroinform.* 5:35. doi: 10.3389/fninf.2011.00035

Kunkel, S., and Schenck, W. (2017). The NEST dry-run mode: efficient dynamic analysis of neuronal network simulation code. *Front. Neuroinform.* 11:40. doi: 10.3389/fninf.2017.00040

Kunkel, S., Schmidt, M., Eppler, J. M., Masumoto, G., Igarashi, J., Ishii, S., et al. (2014). Spiking network simulation code for petascale computers. *Front. Neuroinform.* 8:78. doi: 10.3389/fninf.2014.00078

Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2021). Sub-realtime simulation of a neuronal network of natural density. *Neural. Comput. Eng.* doi: 10.1088/2634-4386/ac55fc

Linssen, C., Lepperod, M. E., Mitchell, J., Pronold, J., Eppler, J. M., Keup, C., et al. (2018). NEST 2.16.0. *Zenedo.* doi: 10.5281/zenodo.1400175

Lytton, W. W., Seidenstein, A. H., Dura-Bernal, S., McDougal, R. A., Schürmann, F., and Hines, M. L. (2016). Simulation neurotechnologies for advancing brain research: parallelizing large networks in NEURON. *Neural Comput.* 28, 2063–2090. doi: 10.1162/neco_a_00876

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., et al. (2020). "MLPerf training benchmark," in *Proceedings of Machine Learning and Systems,* Vol. 2, eds I. Dhillon, D. Papailiopoulos, and V. Sze, 336–349. Available online at: https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf

McDougal, R. A., Bulanova, A. S., and Lytton, W. W. (2016). Reproducibility in computational neuroscience models and simulations. *IEEE Trans. Biomed. Eng.* 63, 2021–2035. doi: 10.1109/TBME.2016.2539602

Message Passing Interface Forum (2009). *MPI: A Message-Passing Interface Standard, Version 2.2.* Technical Report, Knoxville, TN, United States.

Migliore, M., Cannia, C., Lytton, W. W., Markram, H., and Hines, M. (2006). Parallel network simulations with NEURON. *J. Comput. Neurosci.* 21, 119–223. doi: 10.1007/s10827-006-7949-5

Miyazaki, H., Kusano, Y., Shinjou, N., Fumiyoshi, S., Yokokawa, M., and Watanabe, T. (2012). Overview of the K computer System. *Fujitsu Sci. Techn. J.* 48, 255–265.

Monteforte, M., and Wolf, F. (2010). Dynamical entropy production in spiking neuron networks in the balanced state. *Phys. Rev. Lett.* 105:268104. doi: 10.1103/PhysRevLett.105.268104

Morrison, A., Hake, J., Straube, S., Plesser, H. E., and Diesmann, M. (2005a). "Precise spike timing with exact subthreshold integration in discrete time network simulations," in *Proceedings of the 30th Göttingen Neurobiology Conference*, 205B, Göttingen.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005b). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801. doi: 10.1162/0899766054026648

Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., and Veidenbaum, A. V. (2009). A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Netw.* 22, 791–800. doi: 10.1016/j.neunet.2009.06.028

Oliver, H. J., Shin, M., Sanders, O., Fitzpatrick, B., Clark, A., Dutta, R., et al. (2021). cylc/cylc-flow: cylc-flow-8.0b3. *Zenedo.* doi: 10.5281/zenodo.5668823

OpenMP Architecture Review Board (2008). *OpenMP Application Program Interface.* Available online at: http://www.openmp.org/mp-documents/spec30.pdf (accessed September 27, 2016).

Ostrau, C., Klarhorst, C., Thies, M., and Rückert, U. (2020). "Benchmarking of neuromorphic hardware systems," in *NICE '20: Proceedings of the Neuro-inspired Computational Elements Workshop*, Heidelberg. doi: 10.1145/3381755.3381772

Pauli, R., Weidel, P., Kunkel, S., and Morrison, A. (2018). Reproducing polychronization: a guide to maximizing the reproducibility of spiking network models. *Front. Neuroinform.* 12:46. doi: 10.3389/fninf.2018.00046

Pfeil, T., Grübl, A., Jeltsch, S., Müller, E., Müller, P., Petrovici, M. A., et al. (2013). Six networks on a universal neuromorphic computing substrate. *Front. Neurosci.* 7:11. doi: 10.3389/fnins.2013.00011

Plesser, H. E., Eppler, J. M., Morrison, A., Diesmann, M., and Gewaltig, M.-O. (2007). "Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers," in *Euro-Par 2007: Parallel Processing, Vol. 4641 of Lecture Notes in Computer Science*, eds A.-M. Kermarrec, L. Bougé, and T. Priol (Berlin: Springer-Verlag), 672–681.

Potjans, T. C., and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Pronold, J., Jordan, J., Wylie, B. J. N., Kitayama, I., Diesmann, M., and Kunkel, S. (2021). Routing brain traffic through the von Neumann bottleneck: Efficient cache usage in spiking neural network simulation code on general purpose computers. *arXiv [Preprint]*. arXiv: 2109.12855. Available online at: https://arxiv.org/pdf/2109.12855.pdf (accessed March 11, 2022).

Pronold, J., Jordan, J., Wylie, B. J. N., Kitayama, I., Diesmann, M., and Kunkel, S. (2022). Routing brain traffic through the Von Neumann bottleneck: Parallel sorting and refactoring. *Front. Neuroinform.* 15:785068. doi: 10.3389/fninf.2021.785068

Rhodes, O., Peres, L., Rowley, A. G. D., Gait, A., Plana, L. A., Brenninkmeijer, C., et al. (2019). Real-time cortical simulation on neuromorphic hardware. *Philos. Trans. R. Soc. A* 378:20190160. doi: 10.1098/rsta.2019.0160

Richert, M., Nageswaran, J. M., Dutt, N., and Krichmar, J. L. (2011). An efficient simulation environment for modeling large-scale cortical processing. *Front. Neuroinform.* 5:19. doi: 10.3389/fninf.2011.00019

Rougier, N. P., Hinsen, K., Alexandre, F., Arildsen, T., Barba, L. A., Benureau, F. C., et al. (2017). Sustainable computational science: the ReScience initiative. *PeerJ Comput. Sci.* 3:e142. doi: 10.7717/peerj-cs.142

Schemmel, J., Brüderle, D., Grübl, A., Hock, M., Meier, K., and Millner, S. (2010). "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proceedings of the 2010 International Symposium on Circuits and Systems (ISCAS)* (Paris: IEEE Press), 1947–1950.

Schmidt, M., Bakker, R., Hilgetag, C. C., Diesmann, M., and van Albada, S. J. (2018a). Multi-scale account of the network structure of macaque visual cortex. *Brain Struct Funct.* 223, 1409–1435. doi: 10.1007/s00429-017-1554-4

Schmidt, M., Bakker, R., Shen, K., Bezgin, G., Diesmann, M., and van Albada, S. J. (2018b). A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas. *PLOS Comput. Biol.* 14:e1006359. doi: 10.1371/journal.pcbi.1006359

Senk, J., Yegenoglu, A., Amblet, O., Brukau, Y., Davison, A., Lester, D. R., et al. (2017). "A collaborative simulation-analysis workflow for computational neuroscience using HPC," in *High-Performance Scientific Computing, JHPCS 2016, Vol. 10164 of Lecture Notes in Computer Science*, eds E. Di Napoli, M.-A. Hermanns, H. Iliev, A. Lintermann, and A. Peyser (Cham: Springer), 243–256. doi: 10.1007/978-3-319-53862-4_21

Sompolinsky, H., Crisanti, A., and Sommers, H. J. (1988). Chaos in random neural networks. *Phys. Rev. Lett.* 61, 259–262. doi: 10.1103/PhysRevLett.61.259

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8:47314. doi: 10.7554/elife.47314

Stimberg, M., Goodman, D. F. M., and Nowotny, T. (2020). Brian2GeNN: accelerating spiking neural network simulations with graphics hardware. *Sci. Rep.* 10:410. doi: 10.1038/s41598-019-54957-7

Thörnig, P., and von St. Vieth, B. (2021). JURECA: data centric and booster modules implementing the modular supercomputing architecture at Jülich supercomputing centre. *J. Large Scale Res. Facil.* 7:A182. doi: 10.17815/jlsrf-7-182

van Albada, S., Chindemi, G., Deger, M., Diesmann, M., Djurfeldt, M., Enger, H., et al. (2015a). NEST 2.2.0. *Zenedo.* doi: 10.5281/zenodo.5772624

van Albada, S. J., Helias, M., and Diesmann, M. (2015b). Scalability of asynchronous networks is limited by one-to-one mapping between effective connectivity and correlations. *PLOS Comput. Biol.* 11:e1004490. doi: 10.1371/journal.pcbi.1004490

van Albada, S. J., Kunkel, S., Morrison, A., and Diesmann, M. (2014). "Integrating brain structure and dynamics on supercomputers," in *Brain-Inspired Computing*, eds L. Grandinetti, T. Lippert, and N. Petkov (Cham: Springer), 22–32.

van Albada, S. J., Pronold, J., van Meegen, A., and Diesmann, M. (2021). "Usage and scaling of an open-source spiking multi-area model of monkey cortex," in *Brain-Inspired Computing*. Lecture Notes in Computer Science (Cetraro: Springer International Publishing), 47–59.

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Front. Neurosci.* 12:291. doi: 10.3389/fnins.2018.00291

van Vreeswijk, C., and Sompolinsky, H. (1998). Chaotic balanced state in a model of cortical circuits. *Neural Comput.* 10, 1321–1371. doi: 10.1162/089976698300017214

Yavuz, E., Turner, J., and Nowotny, T. (2016). GeNN: a code generation framework for accelerated brain simulations. *Sci. Rep.* 6:18854. doi: 10.1038/srep18854

Yoo, A. B., Jette, M. A., and Grondona, M. (2003). "Slurm: simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, eds D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Berlin; Heidelberg: Springer Berlin Heidelberg), 44–60.

Zaytsev, Y. V., and Morrison, A. (2014). CyNEST: a maintainable Cython-based interface for the NEST simulator. *Front. Neuroinform.* 8:23. doi: 10.3389/fninf.2014.00023

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

Frontiers in Neuroinformatics

ORIGINAL RESEARCH
published: 17 May 2022
doi: 10.3389/fninf.2022.847108

Efficient Simulation of 3D Reaction-Diffusion in Models of Neurons and Networks

Robert A. McDougal[1,2,3]*, *Cameron Conte*[2,4,5], *Lia Eggleston*[6], *Adam J. H. Newton*[1,2,7] and *Hana Galijasevic*[6]

[1] Department of Biostatistics, Yale School of Public Health, New Haven, CT, United States, [2] Center for Medical Informatics, Yale University, New Haven, CT, United States, [3] Program in Computational Biology and Bioinformatics, Yale University, New Haven, CT, United States, [4] Department of Neuroscience, Yale School of Medicine, New Haven, CT, United States, [5] Department of Statistics, The Ohio State University, Columbus, OH, United States, [6] Yale College, Yale University, New Haven, CT, United States, [7] Department of Physiology and Pharmacology, SUNY Downstate Health Sciences University, New York, NY, United States

Neuronal activity is the result of both the electrophysiology and chemophysiology. A neuron can be well-represented for the purposes of electrophysiological simulation as a tree composed of connected cylinders. This representation is also apt for 1D simulations of their chemophysiology, provided the spatial scale is larger than the diameter of the cylinders and there is radial symmetry. Higher dimensional simulation is necessary to accurately capture the dynamics when these criteria are not met, such as with wave curvature, spines, or diffusion near the soma. We have developed a solution to enable efficient finite volume method simulation of reaction-diffusion kinetics in intracellular 3D regions in neuron and network models and provide an implementation within the NEURON simulator. An accelerated version of the CTNG 3D reconstruction algorithm transforms morphologies suitable for ion-channel based simulations into consistent 3D voxelized regions. Kinetics are then solved using a parallel algorithm based on Douglas-Gunn that handles the irregular 3D geometry of a neuron; these kinetics are coupled to NEURON's 1D mechanisms for ion channels, synapses, pumps, and so forth. The 3D domain may cover the entire cell or selected regions of interest. Simulations with dendritic spines and of the soma reveal details of dynamics that would be missed in a pure 1D simulation. We describe and validate the methods and discuss their performance.

Keywords: reaction-diffusion, computer simulation, 3D, multi-scale modeling, reusability

Edited by:
James Courtney Knight,
University of Sussex, United Kingdom

Reviewed by:
Vincenzo Marra,
University of Leicester,
United Kingdom
Dimiter Prodanov,
Interuniversity Microelectronics Centre
(IMEC), Belgium
Iain Hepburn,
Okinawa Institute of Science and
Technology Graduate University,
Japan

***Correspondence:**
Robert A. McDougal
robert.mcdougal@yale.edu

Received: *01 January 2022*
Accepted: *20 April 2022*
Published: *17 May 2022*

Citation:
McDougal RA, Conte C, Eggleston L, Newton AJH and Galijasevic H (2022) Efficient Simulation of 3D Reaction-Diffusion in Models of Neurons and Networks. Front. Neuroinform. 16:847108. doi: 10.3389/fninf.2022.847108

INTRODUCTION

The brain's behavior in health and disease is most naturally observed at the level of functional outcomes, but these outcomes are often indirect consequences of subcellular chemical kinetics (e.g., oxygen and ATP in stroke; amyloid beta and tau in Alzheimer's Disease). The connection between these two scales is non-intuitive due to the many nonlinear-interactions within the brain (e.g., action potentials, networks). Dedicated tools like MCell (RRID:SCR_007307; Stiles et al., 1998) and STEPS (RRID:SCR_008742; Hepburn et al., 2012) enable highly-detailed 3D simulation of parts of neurons to entire cells, enabling the study of microdomains (see e.g., Keller et al., 2008) and other highly localized phenomena but with limited ability to extend to the full cell or a network of neurons to study the implications of these localized dynamics on a broader scale.

The NEURON simulator (RRID:SCR_005393; Hines et al., 2019) has long supported simultaneous simulation of chemical dynamics and networks of neurons, originally through repurposing MOD files—traditionally used for ion channel kinetics—and later through the introduction of a dedicated Python-based reaction-diffusion specification (McDougal R.A. et al., 2013). These early methods were most applicable to phenomena that behave analogously to electrical signaling, such as when a wave of elevated calcium concentration spreads over a large region of the dendritic tree (e.g., Neymotin et al., 2015). Even for these large scale phenomena, a 1D approximation of the tree (so called despite the cell's branching structure because concentration and voltage states are governed by differential equations only on the interior of non-branching sections, with conservation laws governing the branch points) breaks down in regions where the cell is not radially symmetric (e.g., the predicted curvature of a calcium wave front where the dendrite meets the soma) and is inappropriate for smaller scale phenomena on the same spatial scale as the dendrite diameter (e.g., diffusion between neighboring spines); see the examples in the results section.

We have developed a set of approaches with implementations freely available in the development version of NEURON—to efficiently address the need to incorporate 3D intracellular dynamics for subcellular compartments, whole cell and network models. Combining local discretizations and preserving segment mappings accelerates the Constructive Tessellated Neuronal Geometry algorithm (CTNG; McDougal R. et al., 2013) for generating a 3D volume consistent with a neuron point-and-diameter 3D reconstruction, of the sort available *via*, e.g., NeuroMorpho.Org (RRID:SCR_002145; Ascoli et al., 2007). Reaction-diffusion (rxd) kinetics are specified as for 1D simulations, with selected regions of interest simulated selected for 3D simulation *via* a single line function call, while other parts simulated in 1D. The 3D regions of interest are voxelized (meshed into cubic voxels) and any overlapping 3D regions are connected together and with neighboring 1D regions. Threaded, deterministic simulation is enabled using an irregular boundary extension of Newton et al. (2018)'s operator-splitting parallelized Douglas-Gunn Alternating Direction Implicit method (Douglas and Gunn, 1964). Ion channel activity is based on concentrations at the surface of the cell, and ions enter the cell through the surface voxels. Single cell results are validated by comparison to analytic solutions, by comparison of 3D results with other tools, and by comparison of hybrid 1D-3D simulations with pure 3D simulations.

METHODS

Methods and results are described for a development version of NEURON 8.1, although the initial version of most of these methods was introduced in NEURON 7.7. The source code is available at github.com/neuronsimulator/nrn, installers for major platforms are available at neuron.yale.edu, and NEURON can also be installed for linux and macOS *via* `pip install neuron`. The voxelization algorithm is written in a mix of Python, Cython, and C/C++. The interface code is written in Python. For performance reasons, all NEURON reaction-diffusion code used during an active simulation is written in C/C++.

For analyses requiring many simulations, simulation and visualization were split into separate scripts with each simulation's data stored in a SQLite database. To be robust against the possibility of interrupted calculation, simulation scripts checked the database to see if a given set of parameters had already been tested before running the simulation. Graphs were rendered using plotly (for 3D images), plotnine/ggplot, and matplotlib.

Python code for all figures in this manuscript is available on ModelDB (RRID:SCR_007271; McDougal et al., 2017) at modeldb.yale.edu/267018.

Voxelization

3D simulation requires the specification of a 3D domain, typically defined by a mesh (e.g., in VCell; RRID:SCR_007421) or a boundary (e.g., MCell, Smoldyn). Neuron morphologies, by contrast, are typically reconstructed using a series of $(x, y, z; d)$ optical measurements with tree-structured connectivity rooted at the soma, which is sometimes a special case with an outline, typically in 2D. (A neuron's morphology is a graph-theoretic tree in the sense that every non-root section has exactly one parent section, namely the connecting section that is closer to the root.) This information is sufficient for electrophysiology simulation where the space constant is typically on the order of tens of microns, but under-determines the 3D structure for chemical simulation. Several algorithms have been proposed to generate consistent geometries, including our Constructive Tessellated Neuronal Geometries (CTNG) algorithm (McDougal R. et al., 2013) and others (e.g., Lasserre et al., 2011; Mörschel et al., 2017). The full CTNG method is described in our previous paper, but in brief consecutive point-diameter measurements are interpreted as defining the frustum of a right circular cone. Neighboring frusta are joined using clip spheres, with a clipping rule that depends on the taper of frustra and the angle of intersection. Soma outlines are approximated using sheared frusta with dendrites attached to the soma extended to the soma axis to avoid any gaps from the assumption of local radial symmetry given a 2D soma outline. NEURON's Import3D tool stores soma outline points in a Python dictionary; these soma outlines are not used in pure electrophysiology simulations, but the voxelization algorithm checks each section against the dictionary to see if it should be treated as a sequence of frusta or if there is a 2D outline to use.

To accelerate CTNG voxelization and to facilitate its use in simulations incorporating one-dimensional electrophysiology dynamics, we enhanced the original implementation in several ways: (1) additional interpolated points are inserted at electrophysiological compartment ("segment" in NEURON) boundaries so every frusta belongs to exactly one compartment; (2) each electrical compartment is voxelized separately, thus preserving the relationship between voxels and electrical compartments; (3) each frusta and joining sphere is voxelized separately, exploiting convexity to rapidly identify all the relevant voxels; and (4) voxelized meshes are merged together, with voxels being assigned to the segment closest to the root of the electrophysiological tree (typically the soma).

In CTNG, the constructed 3D volume of the neuron is the union of frusta and spheres clipped by planes. These component objects are voxelized using a modified flood-fill algorithm, starting from the center of an end-face for a frustum or the center of the sphere. In the case of a clipped sphere with a small angle, the resulting wedge may be very small, causing all corners of the voxel to be outside. In this case, additional points in the sphere are tested until a voxel is found with at least one interior corner; this voxel is then used as the seed for the flood-fill. Since the spheres serve to smooth the joins between neighboring frusta, in practice they comprise a small percentage of the total voxels, so this search introduces minimal overhead.

The flood fill is propagated through the surface of the shape as much as possible, using the convexity of the objects to automatically fill in interior voxels between two surface points. This is done by traversing rows of voxels perpendicular to the y, z plane. No matter the orientation of the object, any row of voxels that contains part of the object contains one or more surface voxels at each endpoint of the row's intersection with the object. The modified flood fill calculates these endpoints along with any additional surface voxels in the row, and then fills in any non-surface voxels between the endpoints as interior voxels. To find all the rows intersecting the object, the flood fill searches all the rows bordering an intersecting row, using the endpoints of the original row as "guesses" to retain information about the surface and expedite finding endpoints for the surrounding rows. The signed distance to each surface voxel corner is also computed and stored throughout the flood fill; as in the original CTNG implementation, these signed distances are supplied to the marching cubes algorithm (Lorensen and Cline, 1987) to approximate the surface with a triangular mesh.

To approximate the surface, the marching cubes algorithm requires that at least one corner of a voxel containing the surface be outside the object and at least one corner be inside. NEURON generates a warning suggesting a smaller dx (the length of a voxel edge) if any frustum length or diameter is less than the largest distance that fits within a voxel ($\sqrt{3}\,dx$). Some publicly available reconstructions are sampled with very little distance between the 3D points, leading to a very small suggested value of dx and as noted in McDougal R. et al. (2013), sometimes a bumpy 3D reconstruction; in this case, subsampling the 3D points before loading the morphology into NEURON avoids both the bumpiness and the recommendation of a small dx. The extra segment boundaries added by a very high (per unit length) value of `nseg` (the number of electrical compartments in a section) can produce a similar effect; in this case, the solution is to reduce `nseg` to a value appropriate for the electrical space constant. In NEURON, an appropriate choice of `nseg` can be determined for each section based on the so-called `d_lambda` rule (Hines and Carnevale, 2001).

The areas of the triangles in the surface mesh are summed to estimate surface area, and the portion of each surface voxel inside the object is estimated to be the fraction of test points inside the object. As the voxel has been identified as a surface voxel, at least one corner is inside, and thus the volume estimate will never be 0. Test points are sampled on a uniform grid in 1 + `options.ics_partial_volume_resolution` steps in each direction along the voxels edge, starting and ending at a voxel corner. NEURON versions 7.7–8.0 used an alternative rule for estimating partial volumes using dynamic subsampling, however the approach described above and used beginning in NEURON 8.1 is simpler and provides better scaling.

As neurons occupy a small fraction of the volume of their bounding box (1.498 ± 3.406%) for the neurons in Section voxels are stored as a set of locations (i, j, k) within an imaginary grid comprising a padded bounding box. Thus, memory usage to store the discretization is proportional to the volume of the neuron not to the volume of the bounding box. Likewise, NEURON's simulation times scale proportionally to the number of voxels in the cell, not the number of voxels in the bounding box.

Discretization into a 3D grid happens as needed, allowing interactive changes to grid hyperparameters and morphology without the overhead of re-voxelizing the cell. The mesh is typically generated on the first request for a pointer (e.g., for recording concentration at a point), or when the simulation is initialized. NEURON's internal counters for structure or diameter changes are monitored for subsequent changes at each initialization, pointer request, or simulation step, and the morphology is re-discretized if needed; such re-discretization is expected to be rare in practice as NEURON models typically assume cells do not change shape or size during simulation.

Model Specification

Reaction-Diffusion Kinetics

NEURON's basic reaction-diffusion model specification, introduced in McDougal R.A. et al. (2013), is independent of numerical simulation details such as whether the model is to be simulated in 1D or 3D. Readers are directed to the 2013 paper or for a more complete and updated treatment to the relevant section of the online NEURON documentation (nrn.readthedocs.io/en/latest/rxd-tutorials) for full details, but in brief: domains of a cell are specified in Python using `rxd.Region`, chemical species and their properties using `rxd.Species`, and chemical reactions using `rxd.Reaction`, `rxd.Rate`, or `rxd.MultiCompartmentReaction`. The classes `rxd.Parameter` and `rxd.State` allow fixed values that change with location and non-diffusing state variables, respectively. Dynamics at a specific point (e.g., localized pump) are specified using `node.include_flux` where `node` represents the spatial compartment and the flux is measured in changes in *mass*. Using mass changes instead of concentration changes allows the same amount of a substance to enter the cell regardless of the spatial discretization. To specify that all reaction-diffusion kinetics should be simulated in 3D, call `rxd.set_solve_type(dimension=3)`.

NEURON automatically translates the Python kinetics specification into C and compiles them for use during simulation as described in Newton et al. (2018).

Boundary Conditions

Flux across the plasma membrane (the boundary of the 3D intracellular simulation region) is assumed to be fully defined by explicitly modeled mechanisms such as ion channels and pumps; if there are no such mechanisms defined, the boundary is assumed to be fully reflective (no flux). For compatibility with 1D simulation, no special 3D boundary condition syntax is introduced. Instead, all plasma membrane spanning mechanisms may be described in NMODL (Hines and Carnevale, 2000), NeuroML/LEMS and translated to NMODL *via* `jNeuroML` (Cannon et al., 2014), NEURON's ChannelBuilder tool, or a `rxd.MultiCompartmentReaction`. The last option is the most flexible and allows using an `rxd.Parameter` to specify different values in different 3D compartments as well as defining the movement of uncharged particles. The other mechanisms work for charged particles and support variation at the level of a NEURON segment; the currents they generate are distributed proportionally across the segment's surface voxels by the voxel surface area. Faraday's and Avogadro's constants along with the surface voxel volume and the charge of the particle are used to convert the currents into rates of change of concentration at the boundary due to flux across the plasma membrane.

3D Simulation

To efficiently simulate intracellular regions in 3D, we generalized the parallel Douglas-Gunn algorithm of Newton et al. (2018) to support the irregular 3D boundary of a neuron. Unlike extracellular simulation in NEURON— which is simulated coarsely enough that the morphological details can be subsumed into an effective volume fraction— in intracellular simulation the voxels are necessarily much smaller and need to respect the 3D boundary of the cell. As described in the Section 2.1, voxels may have widely varying amounts of surface, and each voxel must be associated with a specific electrical compartment (with a "segment" in NEURON terminology).

The conceptual algorithm for integration is the same as in Newton et al. (2018), however with voxels only existing inside the cell membrane, the number of voxels in any row is no longer necessarily the same. This variation means that although the memory locations for concentration in a particular voxel and in the voxel above it are fixed for a simulation, the offset between voxels and the voxels above them varies throughout the cell and cannot be calculated using a simple arithmetic expression. To work with this irregularity, the indices of every voxel in each line are precomputed at initialization. To find neighbors, NEURON constructs a dictionary (hash array) keyed by the (i, j, k) voxel location with the value of the index of the voxel in memory. Lines in each direction are formed by starting from an arbitrary voxel, backtracking to the beginning of the line (e.g., if forming the lines parallel to the x axis, we successively check for the presence of $(i-1, j, k)$, $(i-2, j, k)$, $\ldots$ until such a voxel is not in the dictionary), and then recording the indices of the voxels in the line until there is no next voxel indexed in the dictionary. Although Python is used to calculate the indices comprising each line, the results are cached and transferred to C++ code that uses them during integration. This process is repeated if and only if the 3D structure is changed (e.g., more segments, different diameters, ...).

All memory indices are relative to a given `rxd.Species`, (or `rxd.Parameter` or `rxd.State`), each of which has its memory allocated independently, allowing for one to be added or removed without requiring memory for the others to be reallocated.

Fixed step integration proceeds using the two-phase operator-splitting approximation as in McDougal R.A. et al. (2013): reactions and fluxes are calculated using an implicit method first and then diffusion is calculated with DG-ADI, also an implicit method. This introduces a source of error that converges to 0 as $dt \rightarrow 0$, and has the advantage of keeping the matrices that need to be inverted (a $\mathcal{O}(n^3)$ task in the general case) small, involving only one location's concentrations for each reaction matrix or one line for each diffusion matrix. No calculations are done for memory associated with `rxd.Parameter` objects, only reactions are calculated for `rxd.State` objects, and both reactions and diffusions are calculated for `rxd.Species` objects. Fluxes from ion channels specified with MOD files are converted into mass changes per segment and then distributed proportionally across the surface voxels assigned to the segment by voxel surface area. Here, it is assumed that every segment has surface voxels. All diffusion calculations explicitly incorporate the effect of voxel by voxel interior volume as voxels with surface do not have their entire volume inside the cell.

Variable step integration uses the CVODE solver from the SUNDIALS suite (Hindmarsh et al., 2005) with all NEURON rates of change (membrane potentials, ion channel states, reactions, and diffusion) represented in one derivative vector. The approximate Jacobian used for the reaction-diffusion part of the problem is a permutation of a block-diagonal matrix, where each block includes the full reaction Jacobian for a given spatial location but only the diagonal part of the diffusion Jacobian. This simplifying approximation allows the approximate Jacobian to be quickly invertible at a tradeoff of a decrease in accuracy, potentially forcing smaller timesteps than CVODE might use with the exact Jacobian.

NEURON concentrations are tied to segments and the surface nodes are assigned the concentration. In general, there are multiple surface nodes per segment. To address this, the segment concentrations are updated at each time step with the weighted average concentration from the segment's surface voxels. In some cases, using only the surface nodes can cause an artificially high concentrations due to relatively few 3D voxels diffusing with a single 1D segment. This effect can be mitigated by using all 3D nodes to calculate concentrations with `options.concentration_nodes_3d = "all"`.

Multiple threads, specified with `rxd.nthread(n)` where `n` is the number of threads, may be used to accelerate intracellular simulation. Load balancing is achieved using a longest-processing-time first greedy algorithm based on the line length for each direction, which is guaranteed to run in no worse than 4/3 the optimal time (Graham, 1969).

Hybrid Simulation

For performance reasons and to better support simulations with narrow dendrites that would otherwise require a small dx, a Python iterable of sections (list, set, ...) may be provided when specifying the simulation dimension to indicate which sections are being set, e.g., `rxd.set_solve_type(apicals, dimension=3)`. Chemical simulations within a given section must either be in 1D or 3D (i.e., this cannot vary by Region), but each section can be set independently. Capturing the 3D nature of the dynamics within a section generally requires that its neighboring sections also be in 3D as otherwise all the incoming diffusive fluxes from neighboring sections are the same regardless of 3D location. As with specification of the full model, multiple dimension specifications for a given section are allowed, with the last one taking effect.

Boundary Identification

At initialization, each `rxd.Region` identifies which of the sections it contains are to be simulated in 1D and which are to be simulated in 3D. If a region does not contain both sections to be simulated in 1D and in 3D, no additional analysis is done and the simulation is purely in the specified dimension. When both dimensions are present, for every given `rxd.Species` instance, every 3D section's parent, if it exists, is checked to see if it is on the 1D section list. Likewise, every 1D section's parent is checked to see if it is on the list of 3D sections. In recommended usage each cell has its own `rxd.Species` instance for a given conceptual molecule (e.g., each cell would have a `self.ca` for its internal calcium concentrations), the search space is constrained to a given cell. Within any given `rxd.Region` containing both 1D and 3D sections, there may be zero (if the 1D and 3D sections are not contiguous), one, or arbitrarily many places where 1D and 3D sections meet.

For each case where 3D and 1D sections meet, boundary voxels are identified by finding the voxels belonging to the 3D section and its spherical endcap that intersect the plane perpendicular to the line segment defined by the two $(x, y, z; d)$ points at the appropriate edge of the 1D section. In particular, this algorithm requires that the boundary or boundaries must occur at either end of a NEURON Section, not in the middle. Each 1D-3D juncture potentially has many boundary voxels, depending on the 3D discretization. Mass diffuses between each 3D boundary voxel and 1D boundary segment at a rate based on the distance between the center of the voxel and the center cross-section of the boundary segment (estimated as the sum of half the voxel dx and half the segment length) and the cross-sectional area of where the voxel meets the 1D region (estimated as its volume raised to the 2/3 power). Boundary voxel identifiers and distances to the 1D boundary nodes are computed at initialization and cached in a data structure passed to the C++ compute engine *via* `ctypes`. Any subsequent changes to the morphology trigger recalculation of the discretization—including identification of boundary voxels—at the next initialization, advance, or node request event.

Simulation

At the beginning of each timestep, fluxes between 1D and 3D boundary compartments are computed according to the finite volume method and Fick's laws: fluxes are proportional to the concentration gradient and inversely proportional to the 1D distance between the centers of the compartments. In particular, we use the common approximation of neglecting the effects of charge on diffusive spread, i.e., we do not consider electrodiffusion (see e.g., Ellingsrud et al., 2020). The 1D and 3D regions are then advanced independently, applying the fluxes as appropriate, thereby weakly coupling them. This weak coupling introduces minimal performance overhead, but at the cost of reduced numerical stability, thereby potentially requiring a smaller timestep (see e.g., Benedikt and Drenth, 2019).

Random Realistic Neuron Morphologies

To assess performance on realistic morphologies, we identified 21 random reconstructions from NeuroMorpho.Org (Ascoli et al., 2007) with metadata indicating realistic diameters and a 3D reconstruction. These were obtained by querying NeuroMorpho.Org's "Browse by Random" tool, once for 50 random cells and once for 10 random cells, and filtering for those meeting the stated criteria. The randomly selected morphologies as identified by their NeuroMorpho.Org name are: `9CL-IVxAnk2-IR_ddaC` (Nanda et al., 2018), `29-1-8` (Martinez-Canabal et al., 2013), `64-8-L-B-JB` (Ehlinger et al., 2017), `243-3-39-AW` (Nguyen et al., 2020), `2017-25-04-slice-2-cell-2-rotated` (Scala et al., 2019), `070601-exp1-zB` (Groh et al., 2010), `160524_7_4` (Kunst et al., 2019), `15892037` (Takagi et al., 2017), `AE5_EEA_Outerthirds_DG-Mol_sec1-cel4-aev5me` (de Oliveira et al., 2020), `AM61-2-1` and `AM81-2-3` (Trevelyan et al., 2006), `B4-CA1-L-D63x1zACR3_1` (Canchi et al., 2017), `Dnmt3bKO-cell-8 and WT-iPS-derived-cell-12MR` (Tarusawa et al., 2016), `Fig5C` (Herget et al., 2017), `glia_4090` (Helmstaedter et al., 2013), `KC-s-4505762` (Takemura et al., 2017), `Mouse_CA2_Ma_Cell_5` (Helton et al., 2019), `RatS1-6-107` (Nogueira-Campos et al., 2012), `RP4_scaled` (Weiss et al., 2020), and `WT-mPFC-A-20X-3-2` (Juan et al., 2014).

Timings

All reported times are based on measurements on Yale's Farnam HPC's general partition, which has a mix of mostly Intel Xeon E5-2660 v3 CPUs with 119 GiB memory per node and some Xeon 6240 CPUs with 181 GiB memory per node.

RESULTS

Validation

Convergence on a Cylinder and the Role of Voxel Refinement

To assess convergence of surface area and volume calculations, we began by considering a cylinder of diameter 2 μm and length 5 μm. Cylinders, unlike neuron morphologies from reconstructions, have analytically known values for surface area and volume; in particular, here the volume is $5\pi\ \mu\text{m}^3$ and the surface area is $12\pi\ \mu\text{m}^2$. Errors were measured for one thousand random orientations (specified as (ϕ, θ) in spherical coordinates) at negative integer powers of $\sqrt{2}$, approximately dx$=$ 0.5, 0.3536, 0.25, 0.1768, 0.125, 0.0884, and 0.0625 μm without using partial volume resolution on the surface voxels. For reasons that are explored later, we did not choose this to be NEURON's default behavior; using the defaults produces volume errors typically around 4x lower.

Volume estimates converged with error scaling at approximately $\mathcal{O}(dx^{2.18})$, with the average absolute error at dx= 0.5 μm being approximately 0.3976 ± 0.4396 μm^3 and the error at dx= 0.0625 μm being approximately $4.271 \times 10^{-3} \pm 8.590 \times 10^{-3}$ μm^3, a 93.10-fold reduction. This convergence compares favorably to the simpler volume estimating approach of counting the full volume of every voxel that is included in the geometry, which converges at approximately $\mathcal{O}(dx)$. In contrast to that approach, which by definition gives overestimates of volumes, our algorithm gave more underestimates than overestimates (582 out of 1,000 test orientations) when dx=0.25 μm, with an average signed error of -4.205×10^{-2} μm^3.

Surface area converged at approximately $\mathcal{O}(dx^{1.21})$: absolute errors at dx = 0.5 μm were 2.963 ± 0.313 μm^2, and at dx = 0.0625 μm were 0.2384 ± 0.0251 μm^2, an ∼12.43-fold reduction. For a convex shape such as a cylinder, surface areas estimated using marching cubes are expected to be under-estimates (and this holds for 999 out of 1,000 of our test cylinder orientations) as the computed surfaces are planes lying strictly inside the shape; as neurons are not convex, surface areas need not be underestimates in those morphologies.

As interior voxels always contribute their full volume—and therefore do not contribute to volume error—and have no surface, we examined if it was advantageous to use a refined mesh on the surface voxels only to reduce error without incurring the full speed penalty from using a finer global mesh. As explained in the methods, we subdivide surface voxels into VR^3 subvoxels, compute the volumes and surface areas for each, and sum the results together for the total values for the voxel. Although this approach is similar to increasing the overall mesh resolution, it still depends on the coarser mesh's determination of which voxels are surface or not, and thus differences may arise in complex morphologies when, for example, small branches pass near each other.

To examine the effect of voxel subdivisions, we held the cylinder orientation constant, parallel to the *x*-axis, and recorded the errors and runtime for various subdivision levels VR. In NEURON, subdivisions used for volume and surface area calculations are independently configurable, using `rxd.options.ics_partial_volume_resolution` and `rxd.options.ics_partial_surface_resolution`, respectively. In general, as shown in **Figure 1**, increasing VR provides volume errors and discretization times comparable to using a higher resolution grid but without introducing additional voxels that would significantly increase simulation overhead. Subdividing for surface area calculation did not improve the error for a given discretization time, likely due to this strategy increasing the number of marching cubes to compute since it must process domains that would otherwise be classified as fully interior or fully exterior. As such, NEURON's default `rxd.options.ics_partial_volume_resolution` of 2 and `rxd.ics_partial_surface_resolution` of 1 are used for all subsequent calculations, i.e., volume calculations use subdivided surface voxels while surface area calculations do not. As discussed below, we found that the accuracy of the surface voxel partial volume calculations affects the error introduced in 1D-3D hybrid models.

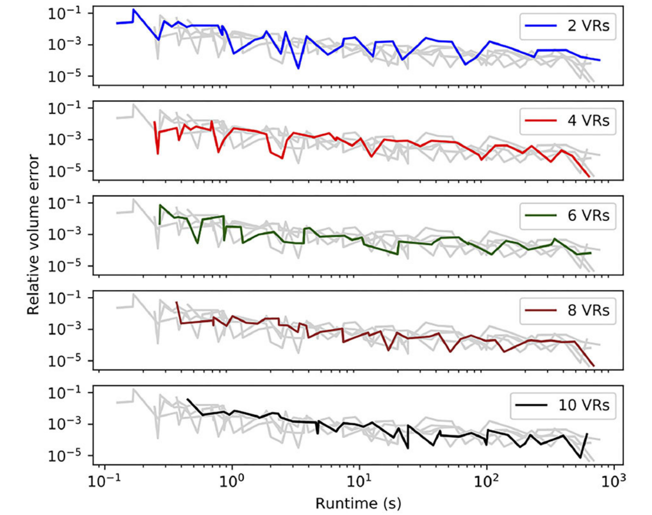


FIGURE 1 | Sub-sampling surface voxels tends to improve the accuracy of the volume estimate for any discretization resolution at the cost of increasing voxelization time. VR is the partial volume resolution, the voxelization times of a cylinder of fixed size and orientation are shown for 50 different values of dx (from 0.01 μm to 0.5 μm) and five values of VR (2, 4, 6, 8, 10). Each subplot highlights one of the VRs, with the others shown in grey. As dx decreases, discretization time increases and relative error tends to decrease, but the error is non-monotonic due to changing alignment of the cylinder with the grids.

Convergence of Discretization on Realistic Geometries

To assess the convergence of volume and surface area estimates on realistic morphologies, we used our voxelization algorithm to estimate these values for 21 randomly chosen neuron reconstructions from NeuroMorpho.Org as described in Section 2.5. For most cells, we tested 12 choices of dx from 0.05 to 0.5 μm, omitting the smaller values for large cells that would require prohibitive setup time or memory at those resolutions. As the true surface area and volumes are unknown, we compared each value for a given morphology to the corresponding value calculated with the smallest dx (**Figure 2**). With dx = 0.5 μm, the majority of whole cell morphologies (13 out of 21) had an estimated relative volume error of <1% with one having error <0.1%. At NEURON's default resolution of dx = 0.25 μm, 15 out of 21 morphologies had a volume error <1% with 5 having a volume error <0.1%. By dx = 0.15 μm, these rates increase to 19 out of 21 and 10 out of 21, respectively. The volume error scaling varies per morphology but scales between $\mathcal{O}(dx^2)$ and $\mathcal{O}(dx^3)$ (**Figure 2B**). With dx=0.15 μm, 10 out of 21 whole cell morphologies had estimated surface area errors <1% and two morphologies had surface area errors <0.1%. For most morphologies, the surface area error scaled between $\mathcal{O}(dx)$ and $\mathcal{O}(dx^2)$ (**Figure 2A**). Thus, the scaling rates for volume and surface area errors with realistic neuron morphologies are

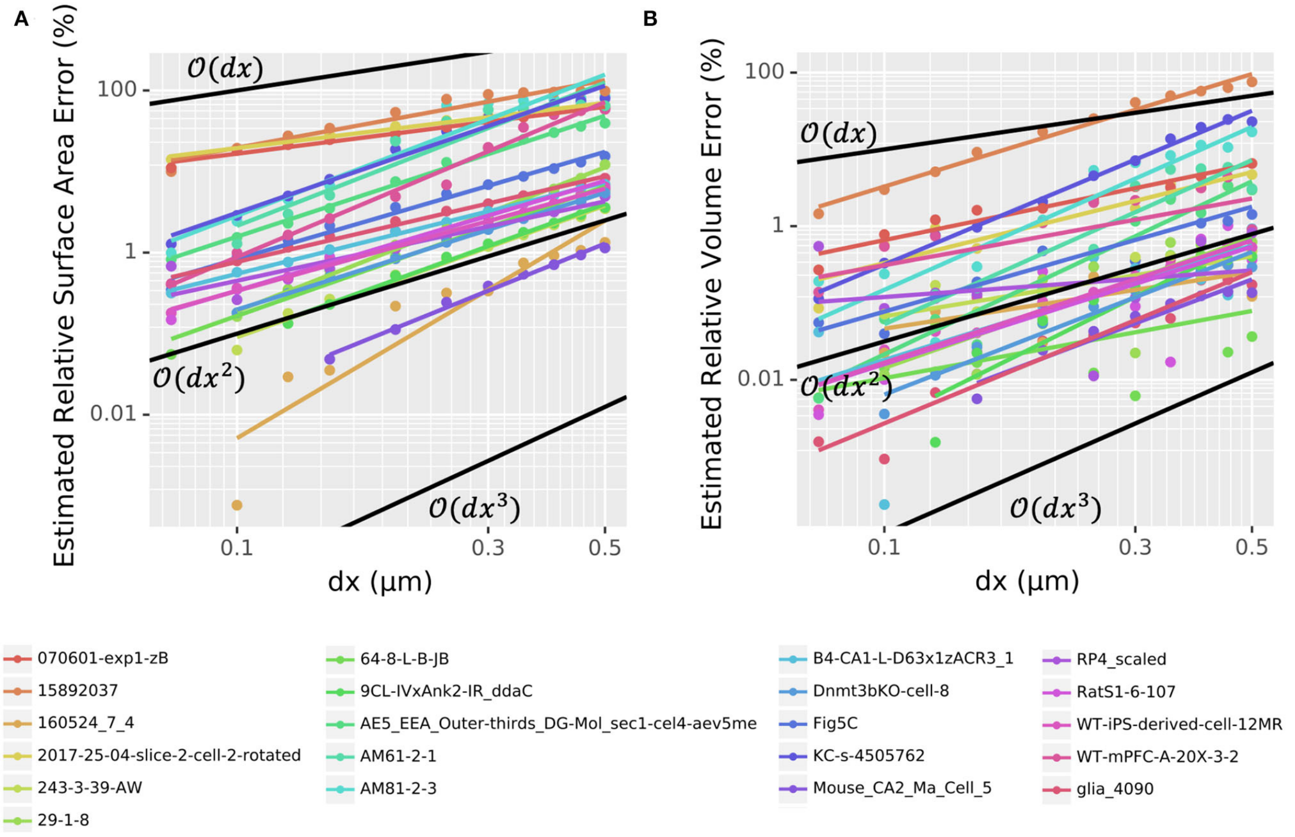


FIGURE 2 | Log-log plots of estimated **(A)** surface area and **(B)** volume relative error as a function of dx for the voxelization of 21 entire morphologies (all sections) chosen randomly from NeuroMorpho.Org. Points indicate measured values; colored lines indicate best-fits. Black lines indicate first-, second-, and third-order convergence, as marked, for reference. Note that the *y*-axis scale is different between **(A)** and **(B)**.

broadly consistent with the rates observed for the cylinder. While the relative errors for the surface area are higher than that for the volume, we note that as described in the methods, for consistency with models not including 3D reaction-diffusion, NEURON always computes ion current influx based on the summed frusta surface areas for the 1D model and only uses the 3D surface areas to distribute the total segment currents into individual voxels.

Voxel-Segment Assignment

For currents through the membrane to correctly alter and be modulated by local near-surface concentrations, each voxel must be assigned to the correct segment and surface voxels must be distinguished from interior voxels. To test these classifications, we constructed a simple geometry, consisting of two parallel connected cylinders, with length 5 µm, diameter 5 µm, and 9 segments, and length 5 µm, diameter 1 µm, 5 segments, respectively. We plotted the quarter of the surface-voxels with $x > 0$, $y > 0$, and $z > 0$ in 3D and color-coded by segment (**Figure 3A**). (The $x > 0$ condition removes the end-face of one of the cylinders.) Visual inspection revealed that our algorithm constructed a continuous surface with no holes and no interior mis-identified voxels. Similar results were found for the other sections of the geometry (not shown), suggesting that the algorithm correctly distinguishes surface and non-surface voxels. To test the voxel-segment assignment, we projected this image into the x, y plane and added markers for the analytically computed segment boundaries (every 5/9 µm for the bigger cylinder and every 1 µm for the smaller cylinder). We additionally added a line segment that passes through the corner of the big cylinder and the midpoint of the first segment of the smaller cylinder which by default in NEURON is assigned a 3D point, and thus this line segment marks the projection of the cone that CTNG adds to join the two cylinders. All segment boundaries aligned with the analytically computed ones and the join cone tapered as expected (**Figure 3**).

Three-Dimensional Simulation

Conservation of Mass

Physically, mass diffusing in any domain should be constant, however the finite limits of computer precision and large numbers of voxels in 3D simulations allow the opportunity for round-off error to accumulate.

To quantify this effect for the serial (1 thread) simulation, we simulated diffusion on a Y-shaped geometry consisting of three sections, each of length 10 µm and diameter 2 µm. One section is positioned from (0, 0, 0) to (10, 0, 0). The other sections

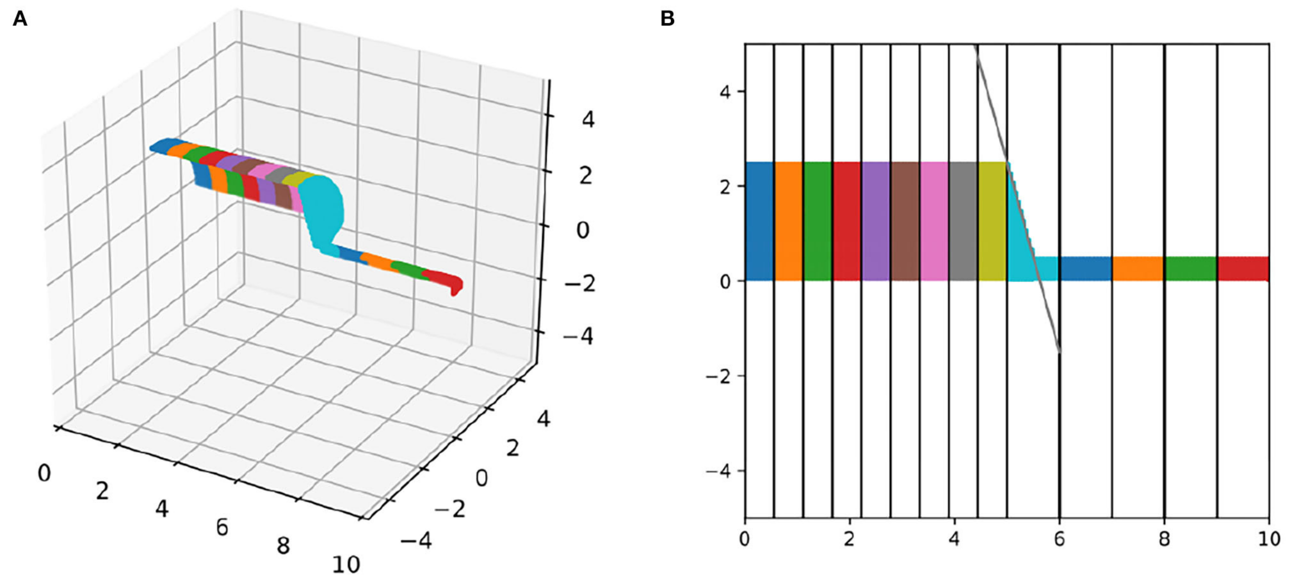


FIGURE 3 | Segment alignment validation. **(A)** 3D plot and **(B)** 2D-projection of surface voxels of a morphology with an abrupt change in diameter and a change in segment length, colored by segment. Vertical lines in **(B)** illustrate the locations of the 1D segment boundaries, which align with the 3D surface nodes. The diagonal black line connects the edge of the last wide segment with the top-middle of the first narrow segment and matches the corresponding 3D cone taper.

continue from there to $(10 + 5\sqrt{3}, \ \pm 5, 0)$, i.e., 30° changes in either direction. While the exact orientation would have no effect on 1D chemical or electrical simulation, we specify these details because they affect the exact number of voxels in the 3D simulation as narrow angles would lead to more overlap of the logical shapes in 3D space and hence less total voxels. We used the default discretization with voxels with dx = 0.25 µm on each side, for a total of 7,904 voxels in our model. Substance diffused with diffusion constant of 1 $\mu m^2/ms$) starting from a concentration of 1 µM on the section parallel to the x-axis and 100 nM on the other two sections and ran for varying lengths of time. For fixed step simulation, we started with a timestep of $dt = 0.025$ ms, NEURON's default. After 4 million timesteps (i.e. by $t = 100,000$ ms) conservation error accumulation led to a relative change of about 8.2219×10^{-10} of the total mass. As timestep reduced to $dt = 0.0125$ ms and $dt = 0.00625$ ms, the relative change in mass after $100,000$ ms reduced to 7.004×10^{-12} and 1.7741×10^{-13}, respectively. For variable step, using NEURON's default tolerance and an `atolscale` of 10^{-6} for the state variable. Without scaling, NEURON's default error tolerance would be 1 µM, small enough for sodium and potassium, but far too large for physiological concentrations of, e.g., calcium which are often about 50–100 nM (Grienberger and Konnerth, 2012). With these settings, by $t = 100,000$ ms, variable step integration accrued a relative change of 2.7389×10^{-13} of total mass over 33,233,603 timesteps. We note that in practice, many NEURON simulations run for orders of magnitude less time, and can expect even smaller error accumulation.

We further examined conservation of mass by running the same simulation with four compute threads. The relative errors in both fixed step and variable step matched the results reported above for the serial case.

Diffusion

To assess the error in our numerical diffusion algorithm, we compared the simulated distribution of concentration on a large finite cylinder to the known analytical solution for the infinite line and infinite space.

In particular, on an infinite 1D line, the Green's function for diffusion with diffusion constant D from initial conditions described by the Dirac delta function $\delta(x)$ is well-known to be:

$$G(x,t) = \frac{1}{\sqrt{4\pi Dt}} \exp\left(\frac{-x^2}{4Dt}\right), \tag{1}$$

see e.g., Balluffi et al. (2005). In particular, this implies that for diffusion on an infinite line with initial concentration 0 everywhere except between A and B where the initial concentration is C, the concentration at position x at time $t > 0$ will be equal to

$$\frac{C}{\sqrt{4\pi Dt}} \int_A^B \exp\left(\frac{-(x-\xi)^2}{4Dt}\right) d\xi. \tag{2}$$

This integral may be evaluated numerically or expressed in terms of the error function (erf). For diffusion on a finite line with reflective boundary conditions (such as an unattached section in NEURON), the exact concentration is an infinite sum of values of that form (with adjusted values of A and B; this is the so-called *method of images*), however this may be numerically neglected as long as the section is sufficiently long and the time sufficiently small.

Likewise, on a right circular cylinder, the solution (Equation 5) applies for all points (r, ϕ, x) (in cylindrical coordinates) provided zero-flux boundary conditions, initial concentrations $u_0(r, \phi, x) = \hat{u}_0(x)$ are independent of r and ϕ (that is, the concentration is uniform on any given cross-section perpendicular to the axis), and a spatially uniform D. This follows immediately from the diffusion equation in cylindrical coordinates as $u_{\phi\phi} = u_r = 0$ (this is immediate at $t = 0$ from the initial conditions and can be shown to hold for $t > 0$) and thus

$$\frac{\partial u}{\partial t} = D\left(\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 u}{\partial \phi^2} + \frac{\partial^2 u}{\partial x^2}\right) \qquad (3)$$

reduces to

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}, \qquad (4)$$

the 1D diffusion equation. A similar reduction applies for any right cylinder regardless of the shape of the base that satisfies the other conditions.

We thus began our diffusion validation by considering a right circular cylindrical section 200 μm long with diameter 1 μm oriented along the x-axis with concentration 0 everywhere except 1 mM between positions 95 and 105 μm with diffusion constant $D = 1\ \mu\text{m}^2/\text{ms}$ with dx values of 0.5, 0.25, and 0.125 μm. The maximum absolute error at time $t = 100$ ms when simulated using NEURON's default 0.025 ms time step when compared to the theoretical values reduced at approximately $\mathcal{O}(\text{dx}^2)$: the maximum absolute error with dx = 0.5 μm was 5.29×10^{-5} mM, with dx = 0.25 μm was 1.28×10^{-5} mM, and with dx = 0.125 μm was 2.79×10^{-6} mM. The distribution of absolute errors and concentration vs position for this problem is shown in **Figure 4**.

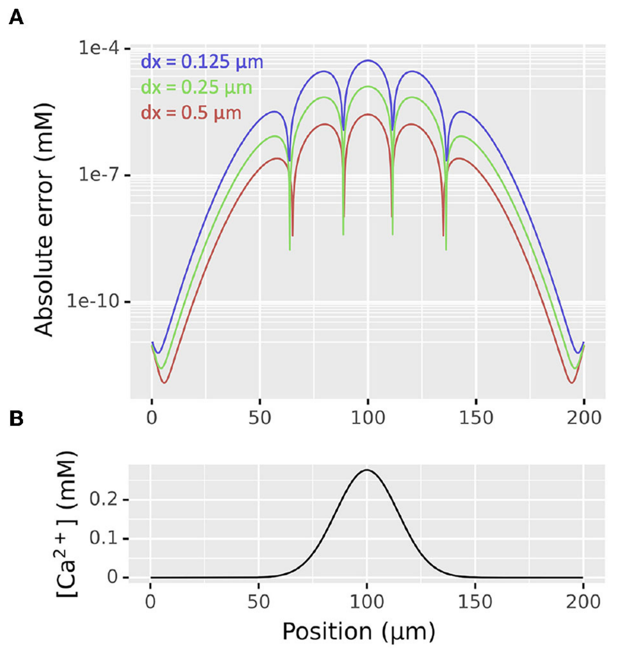


FIGURE 4 | (A) Distribution of absolute errors as functions of spatial discretization dx at $t = 100$ ms from simulation of a diffusion problem on a cylinder of length 200 μm, diameter 1 μm from an initial concentration of 1 mM between positions 95 and 105 μm, 0 elsewhere. The apparent bumpy shape is an artifact of plotting the absolute value on a log scale; at each sudden drop in error the 3D simulated values switch from being an over- to an under-estimate or vice-versa. **(B)** Distribution of concentration at the same time point as determined by the analytic solution.

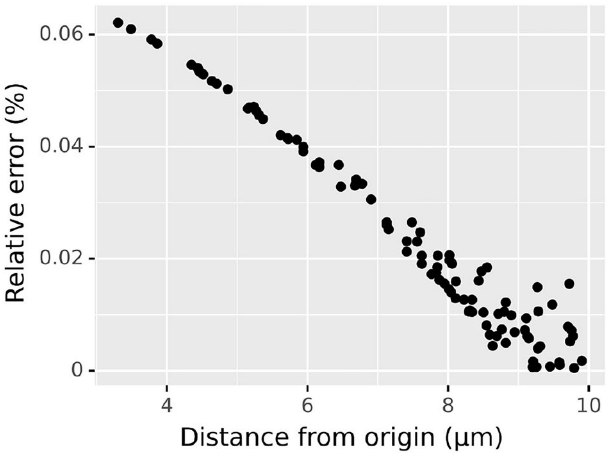


FIGURE 5 | Relative error vs. distance from the origin in 3D diffusion simulation from a cube of elevated concentration centered at the origin, simulated using the default spatial discretization; see text for details.

Analogously, we used the Green's function for diffusion in 3D space from a point source at the origin,

$$G(\vec{x}, t) = \frac{1}{(4\pi Dt)^{3/2}} \exp\left(\frac{-|\vec{x}|^2}{4Dt}\right), \qquad (5)$$

to assess the numerical accuracy of our 3D diffusion algorithm in space as opposed to in a cylinder. Again, we chose a domain sufficiently large and time point sufficiently small to neglect the reflective boundary conditions; in particular, we consider a cylinder centered around the origin of diameter 40 μm and height 40 μm. Within this domain, we take initial concentration of 0 everywhere except in the cube $[-2, 2] \times [-2, 2] \times [-2, 2]$ where we take initial concentration of 1 mM; as before, we suppose the substance diffuses with a diffusion constant of $1\ \mu\text{m}^2/\text{ms}$. Analytic solutions follow from Equation (5) analogously to Equation (2) but with a triple integral over the domain with the non-zero initial conditions. We simulated until $t = 20$ ms using the default spatial discretization of 0.25 μm and plotted the relative error at 100 randomly chosen points within a sphere of radius 10 μm centered around the origin (**Figure 5**). As the initial source was not spherically symmetric, the concentrations themselves are not spherically symmetric, however the relative error (always under 0.1%) exhibits a clear relationship to the distance from the origin, with the relationship becoming weaker as distance (and thus concentration) increases.

Ion Channel Fluxes

To examine the interplay between membrane potential, ion channels, and diffusion, we simulated sodium dynamics at various diffusion rates within a cylindrical "soma" geometry 10 μm in length and 10 μm in diameter with Hodgkin-Huxley channels under a continuous current injection of 0.1 nA. This current injection was sufficient to cause the cell to fire a train of action potentials, each of which admits sodium current into the cell, raising the sodium concentration. (Sodium concentration change is only simulated when sodium dynamics are explicitly modeled, either through NEURON's rxd mechanism as here or through certain MOD file mechanisms.) As shown in **Figure 6**, with a low sodium diffusion rate, the sodium concentration near the surface builds up rapidly. As the diffusion rate increases, the surface concentration approaches that of the corresponding 1D simulation as the sodium is more able to spread across the dendrite's cross-section. By definition, the difference in sodium concentration in the surface voxels leads to a difference in the sodium Nernst potential (which is automatically recalculated by NEURON), which affects subsequent sodium currents and hence spike timing and shape, leading to the separation of spike times for the different diffusion rates shown in the inset to **Figure 6A**.

With larger values of dx, the surface voxels extend deeper into the soma, providing an averaging effect that approaches that of the 1D solution. The resulting numerical difference is most pronounced for small diffusion constants: With dx = 0.5 μm, the surface concentrations at $t = 100$ ms for $D = 10^{-4}$ μm^2/ms and $D = 0.01$ μm^2/ms were 13.35 and 11.24 mM, respectively. With dx = 0.25 μm (NEURON's default), the surface concentrations at the same time point and diffusion constants were 15.59 and 11.16 mM, respectively.

In the case of a single section, the same dynamics would be observed for a 2D model using radial shells to incorporate the difference between near-plasma-membrane concentrations and interior concentrations, however the 3D approach used here avoids the non-physical-realizability of radial shells at branch points (see, e.g., **Figure 1** in Chen and De Schutter, 2017).

3D Simulation on Realistic Geometry

For a more complete test, we compared simulations of scalar bistable dynamics on a realistic cell morphology using our algorithm with using the 3D cell biology simulator VCell (Schaff et al., 1997; Cowan et al., 2012). We used CTNG in NEURON to voxelize the morphology of NeuroMorpho.Org:NMO_02699 (Ascoli et al., 2007; Nikolenko et al., 2007). The voxelized data was exported to a stack of PNG images, where each image represents a z-slice with a value of 0 for voxels not in the cell and a value of 255 for voxels in the cell. These image stacks were then loaded into VCell with each pixel corresponding to one voxel. We note, however, that while this transfer approach correctly transfers information about which voxels are included, it loses the fractional volume calculated for surface voxels within NEURON, so the two tools are not expected to produce identical results as the boundaries vary slightly. In each tool, the initial concentrations were set to be 1 mM in the distal apical and 0 mM elsewhere. Reaction-diffusion was simulated until time $t = 220$ ms, and corresponding z-slices were compared. With both tools, the wavefront was at the same approximate location mid-soma and showed comparable curvature (**Figure 7**).

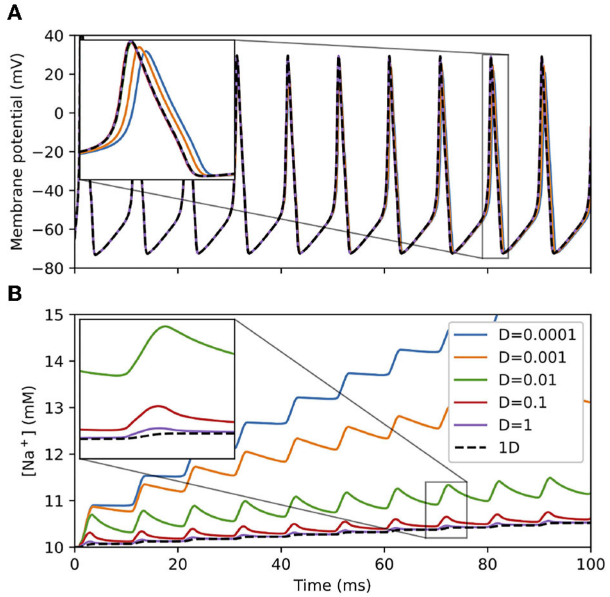


FIGURE 6 | (A) Membrane potential and **(B)** surface voxel sodium concentration of a 3D cylindrical soma with Hodgkin-Huxley channels and sodium accumulation, 10 μm in diameter and 10 μm in length with a constant current injection of 0.1 nA at various diffusion constants D (μm^2/ms). Legend applies to both sub-figures. Insets: magnified views of indicated regions showing differences in 3D results depending on the diffusion constant and convergence to the 1D solution (black dashed line) as the diffusion constant increases.

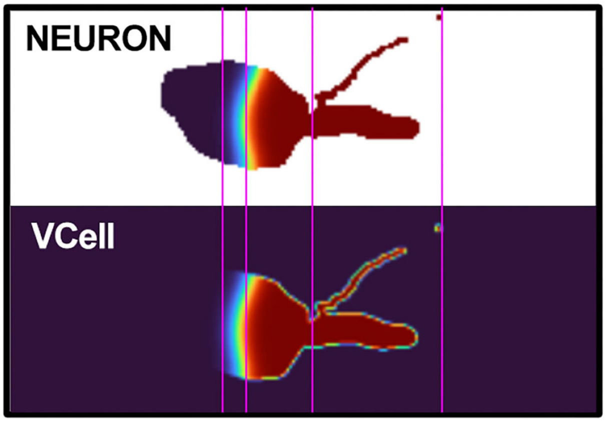


FIGURE 7 | NEURON vs. VCell comparison. Reaction-diffusion NEURON and VCell simulation results of one cell z-slice at $t = 220$ ms. Image cropped to show relevant cell slice areas. Note the similarity between the characteristics of the wave curvature, approximate wave position, and the thickness of the wave front in each simulation.

Orientation Sensitivity With Propagating Wave

The orientation of a section affects how many voxels will be on the boundary and how the surface cuts through them, but the boundary voxel partial volumes and surface areas are inherently only approximations. To assess the impact of these

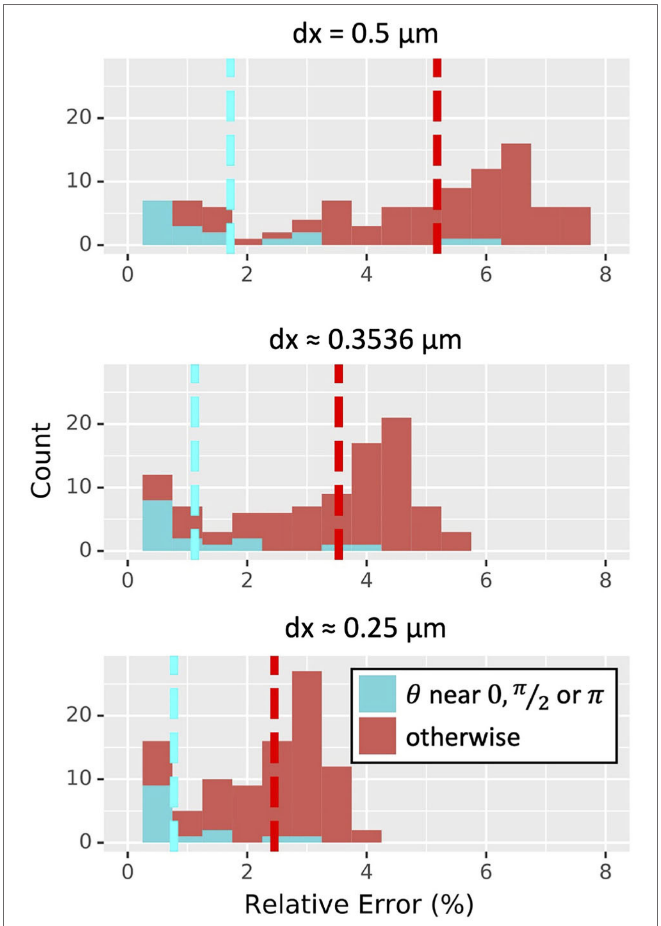


FIGURE 8 | Distribution of relative error in wave speed for the scalar bistable wave with $\alpha = 0.25$ derived from 100 random orientations of a 251 µm long, 2 µm diameter cylinder simulated in 3D at three choices of dx (measured in µm); all graphs are on the same scale. Simulations with θ within 0.1 radians of 0, $\pi/2$, or π corresponding to cylinders nearly parallel to either the x, y plane or the z axis are in cyan and tended to have lower errors than those that were oriented otherwise. Dashed lines indicate mean values.

approximations on models incorporating both 3D reaction and diffusion and components that are sensitive to 1D concentrations (e.g., ion channel kinetics specified using NMODL; Hines and Carnevale, 2000), we considered wave propagation governed by the the scalar bistable equation, $u_t = D\,\Delta u - u(1-u)(\alpha - u)$, and timed wave propagation based on 1D concentrations. Here Δ is the Laplacian operator, D is the diffusion constant, and α is a threshold concentration, above which in the absence of diffusion concentrations will tend to increase and below which concentrations will tend to decrease. As observed in McDougal R.A. et al. (2013), these dynamics exhibit key characteristics of some intracellular signaling processes, like calcium waves. Furthermore, this equation has a known analytic solution in the 1D infinite line case that can be used for validation.

To quantify this effect, we tested 100 random orientations of a 251 µm long dendrite of diameter 2 µm. We initialized the wave with a concentration of 1 on the first 50 µm and 0 elsewhere, then let it diffuse with a diffusion constant of $D = 1$ µm^2/ms. (Concentration by default in NEURON is represented in mM, however the units are omitted here as the dynamics are the same as long as the units are consistent.) For each orientation, we estimated wave speed for three different choices of α (0.15, 0.25, 0.35) and three values of dx (0.5 µm, $2^{-1.5} \approx 0.3536$ µm, 0.25 µm). A plane wave in an infinite cylinder with these dynamics is known to propagate with a wave speed of $c = \sqrt{2}(\frac{1}{2} - \alpha)$ (see, e.g., Fife, 1979). We estimated the wave speed in each simulation by measuring the time it took for the wave front (defined as the farthest point with an average 1D concentration over 0.5) to move from position 100–200 µm. These positions and the total length of the dendrite were chosen as they were found to allow reasonably accurate approximations of the wave speed in 1D simulations—i.e., a large enough distance to be free of boundary effects—while keeping the geometry small enough that the 900 total 3D simulations involved in this study could be run in a reasonable amount of time.

For $\alpha = 0.25$, the average relative error in the estimated wave speed decreased proportionally to dx (4.59 ± 2.24 % for dx = 0.5 µm, 3.12 ± 1.56 % for dx ≈ 0.3536 µm, and 2.17 ± 1.09 % for dx = 0.25 µm). At NEURON's default resolution of dx = 0.25 µm, all orientations led to <4% relative error in estimated wave speed; about three-quarters (74 out of 100) showed <3% relative error, and about one-fifth (21 out of 100) had <1% relative error, with the minimum being 0.13% (**Figure 8**). All three values of α tested showed similar distributions of relative errors of wave speed (not shown). Cylinders whose axis was parallel to the x, y plane or mostly vertical gave less error in wave-time estimates than cylinders whose axes were not aligned with the Cartesian grid.

Hybrid 1D-3D Simulation Validation

Conservation of Mass

Simulations of diffusion should conserve mass for 3D and hybrid 1D-3D models. To test hybrid conservation, simple hybrid models were used, where one section joined to one or two other sections, either aligned or a Y-shaped join. A region of initially elevated concentration was placed in one section away from the join and diffusion to the neighboring sections was simulated. Using different voxel sizes and time-steps showed similar change in total concentrations, on the order of 10^{-11}% of initial amount, consistent with the expected numerical error.

Diffusion

We tested the accuracy of 1D-3D hybrid simulation using a cylindrical dendrite of length 153 µm and diameter 2 µm. For all simulations we used a 1D discretization of two segments per micron. We simulated for 50 ms with a diffusion constant of 1 µm^2/ms from an initial distribution of 0 mM everywhere except for a concentration of 1 mM between positions 70 and 83 µm. We compared four different discretization strategies—pure 1D simulation, pure 3D simulation, 1D on the middle 51 µm and 3D elsewhere, and 3D on the middle 51 µm and 1D elsewhere—at time $t = 50$ ms to the analytical solution calculated using Green's functions as described under "Conservation of mass." The analytical solution's concentration at the midpoint of our cylinder at the end time is ~0.4843 mM. Since there are many 3D

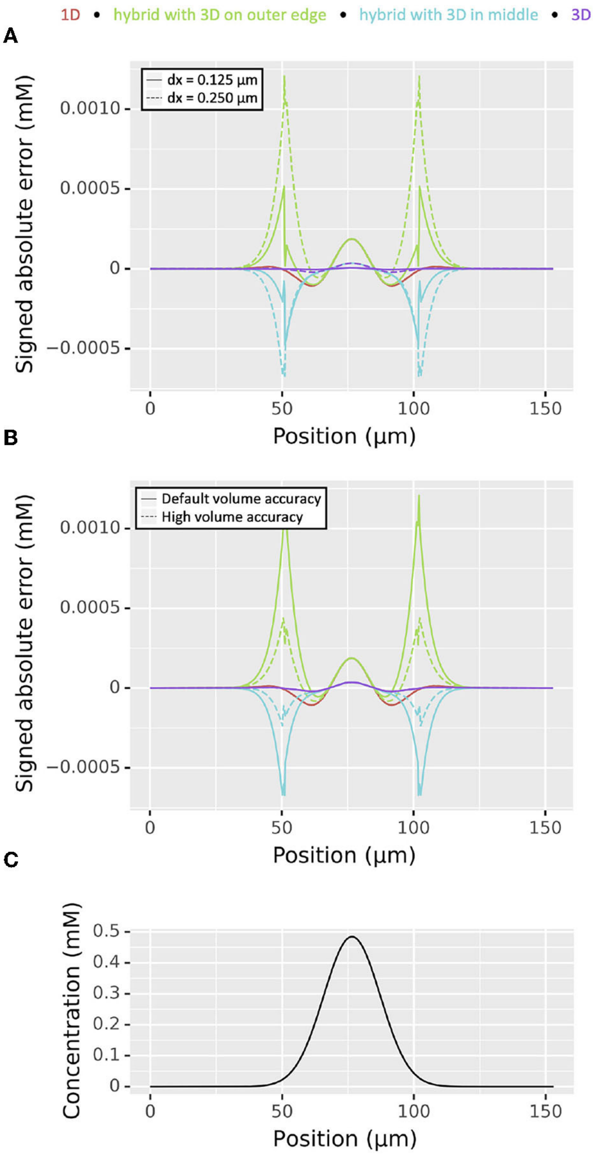


FIGURE 9 | Comparison of the magnitude of signed absolute errors as a function of position for 1D (red), 3D (green), and two hybrid cases (cyan and purple) for diffusion from an area of elevated concentration near the middle of a 153 µm long cylindrical domain ; all plots are for t = 50 ms. **(A)** With different choices of dx. **(B)** With dx = 0.25 µm and different levels of accuracy for estimating the partial volume of voxels that are partly inside and partly outside the cylinder; see text for details. **(C)** Analytically computed concentration distribution.

voxels per x coordinate, we examined the weighted (by volume) concentration as a function of position. In both hybrid cases for both dx = 0.25 µm (the default) and dx = 0.125 µm, numerical absolute error for the weighted concentration at the points where the 1D and 3D domains join exceeded that of the highest absolute error for the pure 3D simulation, however it was generally of the same order (**Figure 9A**). For the 1D in the middle model, when dx was reduced by a factor of two, the maximum error was reduced by a factor >2: the maximum absolute error of the weighted average concentration dropped from 1.21×10^{-3} to 5.17×10^{-4} mM. In surface voxels containing only a small volume of the cell, the error can be larger, reaching up to 1.99×10^{-3} mM and 7.93×10^{-4} mM in the dx = 0.25 µm and dx = 0.125 µm cases on the same hybrid problem, respectively. Maximum errors for the weighted average for the 3D on the middle hybrid problem reduced from 6.72×10^{-4} to 4.76×10^{-4} mM as dx was reduced from 0.25 to 0.125 µm. While all simulations conserve mass, only the fully 1D and fully 3D error curves in **Figure 9A** have an integral of ∼0. This apparent discrepancy is due to inconsistencies in the way 1D and 3D approximate the volume of the cylinder; in this simple geometry, the 3D volumes are consistently under-estimated.

Exploring the volume issue further and motivated by the fact that decreasing dx increases simulation time and the corresponding quantity of generated, we examined the effect of increasing the accuracy of surface voxel partial volume estimates by increasing `rxd.options.ics_partial_volume_resolution`. In particular, for the same setup and holding dx = 0.25 µm constant, we found that the maximum error of the weighted concentrations in the 3D on the outer thirds hybrid case dropped from 1.21×10^{-3} mM when the partial volume resolution was set to 2 (the default) to 4.38×10^{-4} mM when the partial volume resolution was set to 6. Likewise the 3D on the inside case error reduced from 6.72×10^{-4} to 2.37×10^{-4} mM (**Figure 9B**). The analytically computed solution is shown for reference in **Figure 9C**. In both hybrid cases, increasing the accuracy of the partial volume estimates for the surface voxels in this way reduced the absolute error by an amount exceeding that of halving dx. Importantly, after initialization, simulation time is unaffected by the improved partial volume estimates but is greatly affected by dx.

Performance

Defining and simulating a 3D model are logically separate activities: a model only needs to be defined once to be simulated many times (e.g., with different parameters). The most time-consuming part of the definition phase is the voxelization process. Furthermore, in principle, any voxelization that generates the appropriate data structures and maps voxels to segments could be used by the simulation engine. As such, we measure the performance of voxelization (currently single-threaded; described in Section 3.2.1) and the performance of simulation (multi-threaded; described in Section 2.3) separately. To assess the performance using realistic cell shapes, we tested 21 randomly selected morphologies (listed in Section 2.5) with realistic diameters and 3D data from NeuroMorpho.Org (Ascoli et al., 2007).

Voxelization

To assess the voxelization performance, we loaded each of the 21 randomly chosen neuron morphologies one at a time and recorded the initialization time and estimated volume for many choices of spatial resolution `dx`, typically from 0.05 to 0.5 µm. Each timing was run in a separate process, as NEURON caches

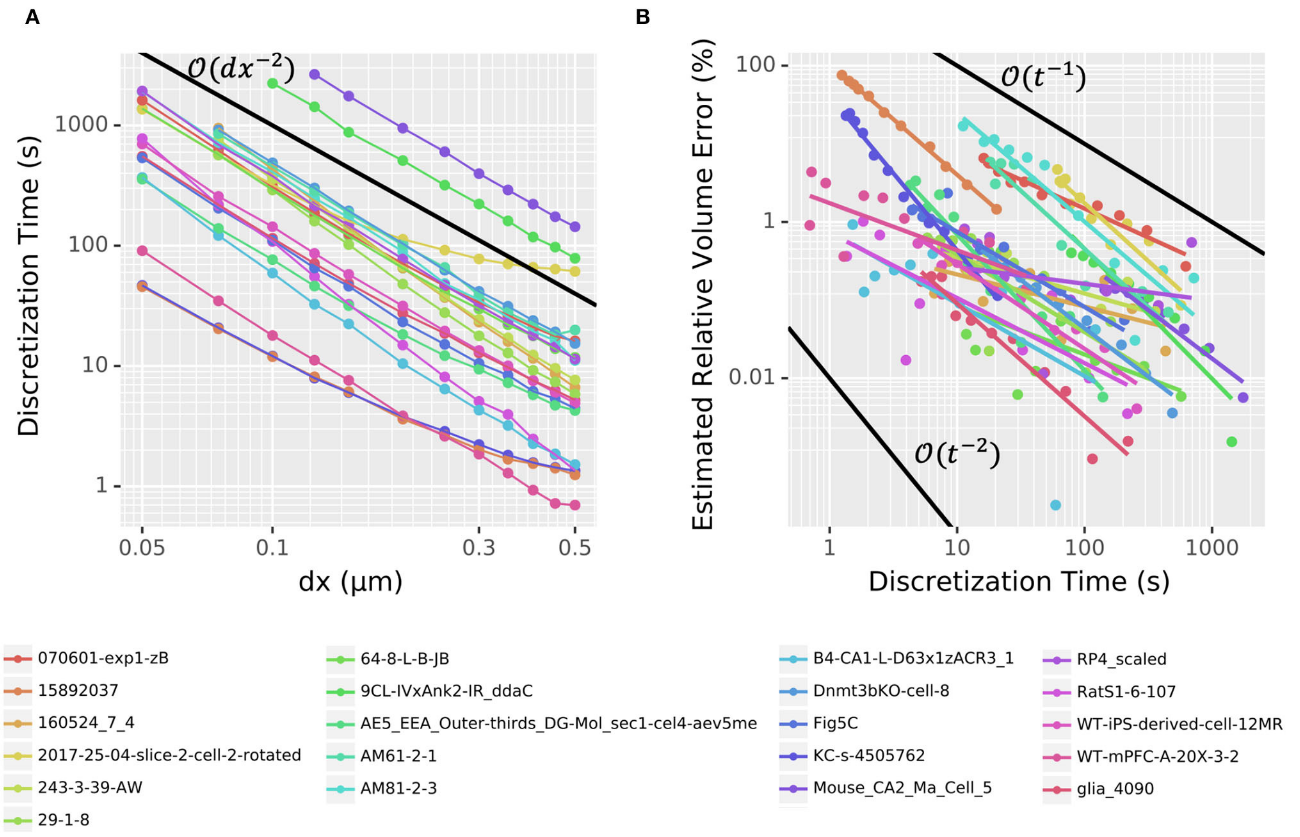


FIGURE 10 | (A) Voxelization time scales as approximately $\mathcal{O}(\mathrm{dx}^{-2})$. Dots denote measured data from 21 different morphologies from NeuroMorpho.Org; colored connecting line segments are illustrative only. **(B)** The scaling of estimated relative volume error varies depending on the morphology, but typically lies between $\mathcal{O}(t^{-2})$ and $\mathcal{O}(t^{-1})$, where t is the time spent computing the voxelization. Points denoted measured values; corresponding colored lines represent a linear (in log-log space) best fit. In both **(A,B)**, black lines give examples of perfect scaling at the rate indicated.

the results to avoid voxelizing the same cell more than once (i.e., subsequent model initializations skip the voxelization step). As shown in **Figure 10A**, the time required to voxelize/discretize the cell scaled nearly consistently at about $\mathcal{O}(\mathrm{dx}^{-2})$ regardless of the morphology, with the main exceptions happening for large `dx`. As before, relative volume error was estimated using the volume calculated for the smallest measured `dx` as the estimated true volume. The relationship between estimated relative volume error as calculated in the convergence on a cylinder section and time spent doing the discretization was noisy and less consistent across morphologies but the error generally scaled between error $= \mathcal{O}(\mathrm{time}^{-2})$ and error $= \mathcal{O}(\mathrm{time}^{-1})$ as shown in **Figure 10B**.

3D Simulation

To assess the scaling of our simulation algorithm with the number of threads used, we simulated three different dynamics (pure diffusion, bistable wave, and calcium wave) across two morphologies (a cylinder of length 50 µm and diameter 1 µm) and a reconstructed cell (NeuroMorpho.Org's NMO_77436 (Canchi et al., 2017)), with three spatial resolutions (dx = 0.25, 0.125, and 0.0625 µm). The pure diffusion dynamics were governed by Fick's laws. The bistable wave modeled here implements the scalar bistable wave equation $u_t = D\Delta u - u(1-u)(\alpha - u)$ analyzed in Fife (1979), and previously used as an example of reaction-diffusion phenomena in McDougal R.A. et al. (2013). The calcium wave model implements Ca^{2+}-induced-Ca^{2+}-release (CICR) driven by the endoplasmic reticulum (ER), and is a simplified version of Neymotin et al. (2015). Waves were initiated by an area of elevated cytosolic concentration (u for the bistable wave and IP3 for the calcium wave) in the first 25 µm in the cylinder case and in section `dend_7[19]` of the apical dendrite which starts approximately 9.35 µm from the soma in the morphologically detailed case.

Excluding the cylinder diffusion and cylinder bistable wave on the coarsest resolution (dx = 0.25 µm), which both initially ran in under 1 second (and for whom threading overhead is thus non-trivial), the rest of the simulations showed speedup as the number of threads increased (**Figure 11**). For the 16 other combinations of morphology, model, and dx: using four threads reduced runtime by up to a factor of 3.63 (2.00±0.65 on average); using eight threads reduced runtime by up to a factor of 6.39 (3.20 ± 1.29 on average); using 16 threads reduced runtime by a factor of up to 9.76 (5.07 ± 2.28 on average). The reported

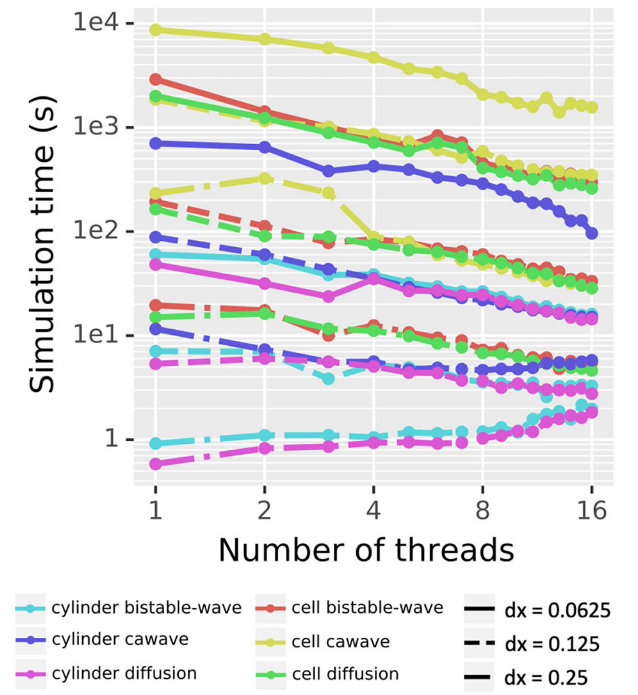


FIGURE 11 | Parallel scaling. Testing all 18 combinations of two choices of morphology, three choices of model, and three choices of dx shows decreased real time for simulation for most combinations as the number of threads increases from 1 to 16. Pure diffusion and a bistable wave on a cylinder at dx = 0.25 μm were the only two test cases that took more real time with 16 threads than with 1 thread. Solid lines indicate dx = 0.0625 μm, dashed lines dx = 0.125 μm, and dash-dot lines dx = 0.25 μm.

runtimes here are based on the best of three runs to limit the contribution from background tasks.

Reported times exclude voxelization time, analyzed above, which is currently single-threaded and is only performed once for a given morphology regardless of the number of simulation studies performed.

To further test performance speed-up, we experimented with different settings of cache prefetch, which in our experiments ultimately did not show significant difference in parallel scaling.

Examples

Three-dimensional simulation offers the ability to explore both the role of a neuron's three-dimensional shape—which is especially relevant wherever the neuron is not approximately conical, such as where the dendrites meet the soma or a spine connects to a dendrite—and the role of precise spatial positioning for, e.g., synapses. In this subsection, we examine examples of each of these, discuss relevant implementation details, and examine different visualization strategies for the resulting volumetric data.

Dendrite-Soma Intersection

Certain cellular phenomena are typically found in one region of the cell and typically not present in neighboring regions even when the neighboring regions are known to be able to support the phenomena. For example, waves of elevated intracellular calcium in pyramidal cells observed in apical dendrites only sometimes invade the soma but when they do are capable of propagating across the soma (Hagenston et al., 2008). There exist mathematical models of wave phenomena where it is known that domain geometry affects wave propagation (e.g., Dronne et al., 2009); 3D simulation allows us to study if the morphology plays a similar role in problems of neuroscientific importance.

For example, we simulated the scalar bistable equation with a threshold $\alpha = 0.1$ mM in the morphology of NeuroMorpho.Org:NMO_53113 (Ascoli et al., 2007; Malik et al., 2016) starting with a concentration of 1 mM on the distal apical and 0 mM elsewhere, and a diffusion constant of 0.25 $\mu m^2/ms$. No other dynamics were included; in particular, no ion channels were simulated and there was no flux across the plasma membrane (Neumann boundary conditions). We chose this cell in part because the soma of this cell was specified using a soma outline in ASC format, allowing NEURON to construct a non-cylindrical approximation to the soma shape. Using a fixed-step simulation (dt = 0.25 ms), we simulated the volume containing the soma and all sections whose center was within a path distance of 70 μm from the soma's center in 3D, with the rest of the cell in 1D. Within this volume of 3D simulation, the smallest diameter was 0.18 μm, and we used a dx = 0.17 μm. Under these conditions, a wave of elevated concentration propagated from the apical toward the soma at approximately uniform speed. Near the soma, the wave front curved and slowed, but propagated into the soma where it eventually straightened and resumed its initial speed (**Figure 12A**) shows the progression of the wave front over time using contours on a 2D projection of the cell.

To assess if the hybrid approach was accurately simulating wave behavior within our region of interest near the soma, we repeated the experiment using all sections whose center was within a path distance of 100 μm from the center of the soma; this expansion added an additional 19 sections to the 3D domain. Simulating in 3D on this expanded domain gave a visually identical contour map of wave propagation (not shown), and an identical prediction for when the wave would cross the center of the soma (t = 188.255 ms), defined as the first time the 1D concentration at the center of the soma exceeded the half-maximal value. This consistency suggests that our original simulation was not losing significant accuracy near the soma despite simulating distal parts of the cell in 1D. By contrast, simulating only the soma and the sections directly connected to the soma in 3D led to a different wave crossing time (t = 194.58 ms), which therefore indicates this smaller region is not a sufficiently large 3D region for studying behavior near the soma.

We note that **Figure 7** presents a similar experiment on a different morphology showing a color-coded 2D slice at a specific time point. The visualization in the latter figure shows more detail on the concentration distribution near the wave front, but cannot show the propagation of the wave front over time.

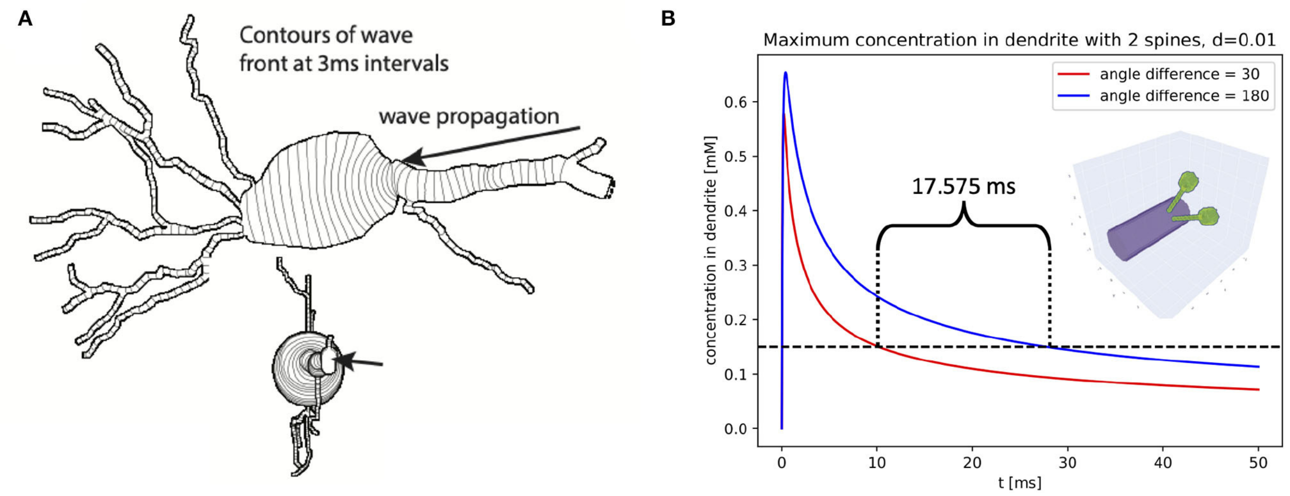


FIGURE 12 | Chemical concentration within neurons is affected by the 3D shape of the cell. **(A)** An intracellular wave entering the soma may curve, slow, and in extreme cases fail to propagate. **(B)** Diffusion of ligand from neighboring spines (red; illustrated as inset) leads to different trajectories of higher peak dendrite concentrations than from spines opposite each other on the dendrite.

Spines

Numerous publications have examined the interactions of spines with each other (e.g., Chiu et al., 2013), how concentrations may be compartmentalized within spines (e.g., Yuste et al., 2000) and the relationship between membrane potential in spine and dendrite (e.g., Jayant et al., 2017). As with dendrites meeting the soma, the exact nature of dynamics at the spine-dendrite juncture depends on the shape of this connection. This is in part dependent on the angle with which the spine attaches to its dendrite, a detail completely lost in 1D simulation, but analagous to the issues arising at the soma.

Spine-dendrite modeling, however, also introduces a new challenge arising from non-physically realizable models where the same volumes is part of two separate sections. Such overlapping sections are common in NEURON, as sections are by default connected to the centroid of the parent section. For most models, the length of most sections is longer than the length of the diameter and the child diameters are generally comparable to the parent diameters, so any discrepancies in local surface area or volume due to the overlaps are typically minimal. Models with spines are a notable exception; spine necks vary in shape and size but for example in layer 6 pyramidal cells of the mouse somatosensory cortex are typically <0.2 μm in diameter and <2 μm long (Ofer et al., 2021), and so attaching the spine at the centroid places much of the neck inside the dendrite. Here our choice of mapping the voxel to the 1D compartment closest to the presumptive soma assures that the spine neck is only that portion extending beyond the dendrite proper, but the 3D volume and surface area calculations will be based only on the part that extends beyond the dendrite, and thus the volumes will disagree, and it is possible that some segments may not have any true surface area. This discrepancy between the 1D and 3D representations can be mitigated by shifting the start of the spine neck to be some distance (almost a radius) away from the centroid of the parent dendrite using the appropriate NEURON `pt3dstyle` while keeping the perimeter inside the parent dendrite. In the case of a cylindrical dendrite with a smaller orthogonal spine, the maximum spine distance from the dendrite centroid can be found by considering the circular cross-section of the dendrite meeting the rectangular cross-section of the spine neck, placed inside the dendrite such that the two lower vertices are on the perimeter of the dendrite. The resulting distance d from the centroid is determined by a right triangle, formed by the center of the dendrite, one of the lower vertices of the spine neck, and the center of the lower edge of the spine neck. This gives a triangle with hypotenuse r_d, adjacent r_n and opposite $r_d - d$, where r_d is the radius of the dendrite and r_n the radius of the spine neck. Then by Pythagoras's theorem $d = r_d - \sqrt{r_d^2 - r_n^2}$.

Using this rule, we constructed a cylindrical dendrite 2.5 μm in diameter and 6 μm long. We attached two spines at position 3 μm orthogonal to the dendrite with necks of length 3 μm, diameter 0.1 μm and cylindrical heads of length 0.5 μm and diameter 0.6 μm. To simulate the spread of a substance from the spines, they were initially filled with substance to a concentration of 2 mM, with 0 mM in the dendrite. When the substance was allowed to diffuse at a rate of 0.01 μm^2/ms, the dynamics of the concentrations within the dendrite varied depending on the angle separating the two spines (**Figure 12B**). We considered two cases: spines 30° apart, and spines 180° apart. We note that in a non-3D simulation these two cases would give identical results. The peak dendritic concentration in each case was reached within the first 0.5 ms, with the closer spines leading to a peak concentration 89% as high as with the spines on the opposite side of the dendrite. All voxels dropped below a concentration of 0.15 mM 17.575 ms earlier when the dendrites were near each other than when they were opposite each other. If the threshold for triggering another reaction was around 0.15 mM, this difference in time above that value could make the difference between whether or not the downstream reaction was triggered. For different choices of parameters (e.g., with a thinner dendrite), the same model could have the peak concentration drop below the threshold in the other order.

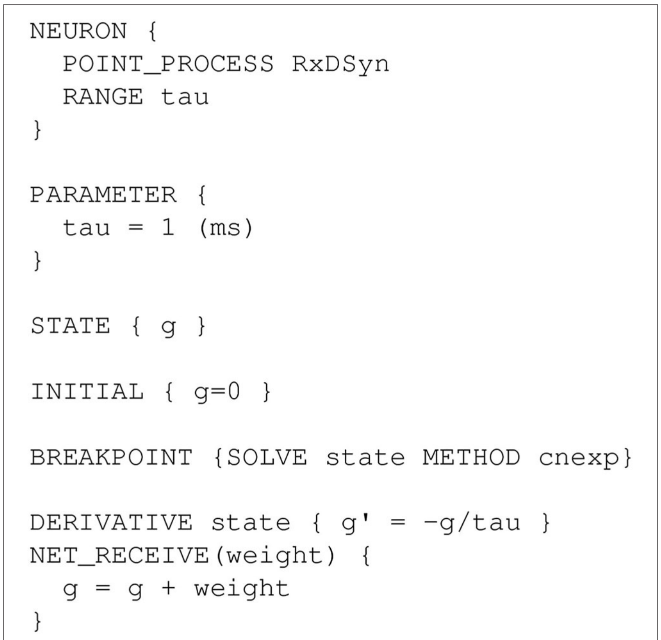
Three-Dimensional Localization of Synapses

Metabotropic receptors and other mechanisms exist at specific points in 3D space. Such dynamics in 1D are often specified using files written in NMODL (Hines and Carnevale, 2000), a domain specific language for ion channel, receptor, and artificial cell kinetics supported by NEURON, Arbor (Akar et al., 2019), and the Python `nmodl` module (github.com/bluebrain/nmodl). We can apply the same approach to synapses located in 3D space, but a few extra considerations are necessary.

First, we begin by defining our post-synaptic response kinetics. In principle, these can be arbitrarily complicated to reproduce experimental observations, however as a first approximation it is not uncommon in modeling to see mechanisms where the rate of production jumps abruptly in response to synaptic and decays exponentially; such an NMODL file is shown in **Figure 13**. In this file, `g` denotes the rate of production of a substance; physically, this corresponds to a change in mass per ms. To support traditional NMODL files, all currents generated by an NMODL mechanism are distributed over the entire segment surface. As a segment is the smallest electrical compartment, that behavior is correct for the electrical aspects of the simulation, however distributing, e.g., sodium currents across the surface would result in sodium changes in all surface voxels. To avoid this issue, a 3D targeted NMODL mechanism must generate only `NONSPECIFIC_CURRENT` with chemical changes driven solely by the rate `g`.

NMODL mechanisms must be compiled before they can be used. This is typically done by running `nrnivmodl`, but we note additional compilation options are sometimes available. NEURON loads compiled NMODL mechanisms from the current directory at startup and can also load them on demand *via* `h.nrn_load_dll`. Once loaded, the `POINT_PROCESS` name (`RxDSyn` in **Figure 13**) is available as a class in NEURON's `h` object. That is, a new instance could be created by `r = h.RxDSyn(seg)`, where `seg` is the segment that contains the mechanism.

Once we have picked the kinetics, the next step is to identify the 3D location to place them. If `ca` is an `rxd.Species` on a 3D region, then `ca.nodes[(x, y, z)]` is an `rxd.NodeList` of `ca` nodes containing the point `(x, y, z)`. As each node covers a volume, there are many points within a `Node` but at most one `Node` that contains the point unless the `rxd.Species` is present on more than one region (e.g., calcium might be present in both the ER and the cytosol, as in Neymotin et al., 2015). The coordinates of the center of a `Node`'s voxel are `(node.x3d, node.y3d, node.z3d)`. Note that if `node` is on the surface, then `node.surface_area` should be strictly positive. If the surface exactly touches a grid corner, it is possible that some voxels with zero surface area will be included in the mesh, but as such, these should not be used for surface-based kinetics. If the segment containing the mechanism was initially unknown, it can be obtained from the selected node *via* `node.segment`.



```
NEURON {
  POINT_PROCESS RxDSyn
  RANGE tau
}

PARAMETER {
  tau = 1 (ms)
}

STATE { g }

INITIAL { g=0 }

BREAKPOINT {SOLVE state METHOD cnexp}

DERIVATIVE state { g' = -g/tau }
NET_RECEIVE(weight) {
  g = g + weight
}
```

FIGURE 13 | Source code for a generic NMODL mechanism called `RxDSyn` that receives synaptic events (`NET_RECEIVE` block) causing the flux *g* to increase abruptly in response to an event by an associated weight, with the flux decaying exponentially with time constant `tau` thereafter (`DERIVATIVE` block).

Mechanisms may be connected to one or more nodes by passing a pointer to the rate to the node's `include_flux` method; in our example, this is `node.include_flux(r._ref_g)`. By default, this method assumes `g` is measured in molecules per second; these units ensure that the same total amount of substance is enters the cell regardless of the discretization. The same flux rate may optionally be applied to other nodes.

Once this is done and the post-synaptic dynamics are connected to a presynaptic event source (e.g., a membrane potential crossing a threshold or a random spike train), then presynaptic events will trigger production of the `node`'s substance at a rate that decays over time (if using the kinetics of **Figure 13**) and that substance is free to diffuse away (**Figure 14**).

DISCUSSION

NEURON 8.1 provides built-in support for parallel, 3D deterministic simulation of intracellular reaction-diffusion dynamics (e.g., protein and ion interactions and diffusion) in whole neurons and in modeler-selected Sections of interest; the remaining Sections, with kinetics expressed identically, continue to use 1D reaction-diffusion simulation, allowing computational resources to be targeted toward locations where the 3D shape is likely to matter such as the relatively large volumes near the soma. Selected cells or Sections are voxelized using an updated

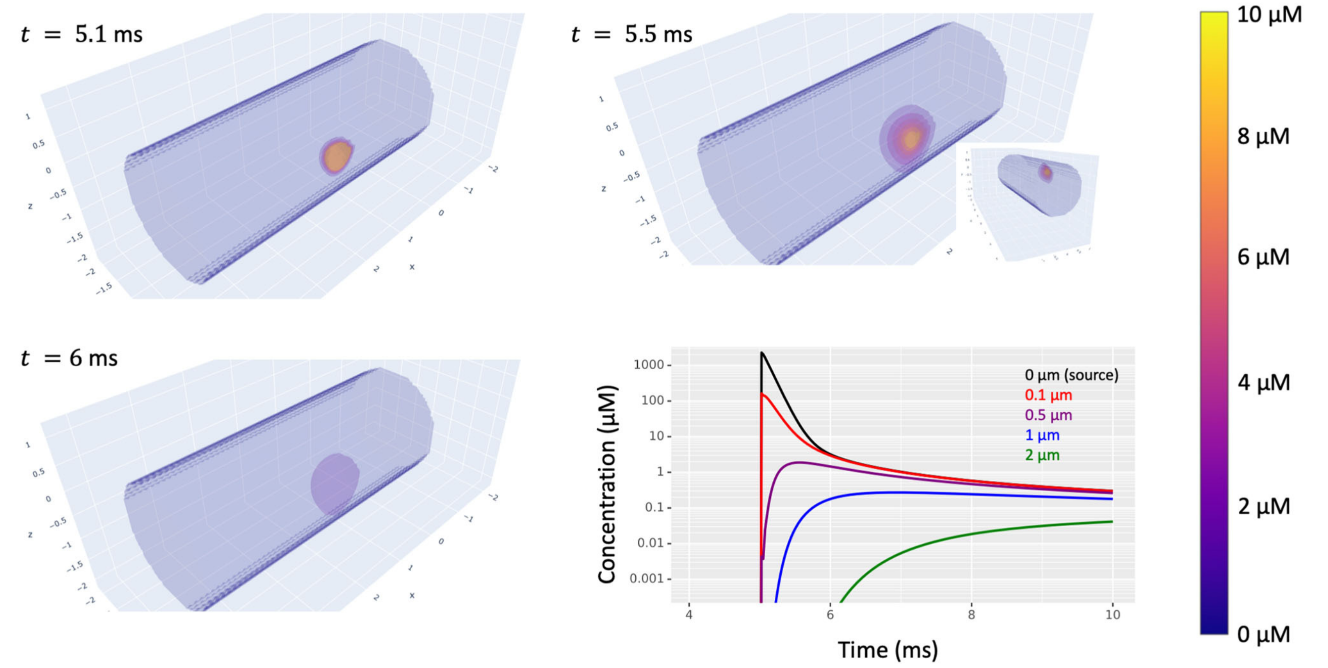


FIGURE 14 | Simulated diffusion in a cylindrical dendrite of a substance produced at (2.5, 0.975, 0.275) in response to synaptic input at time $t = 5$ ms. The volumetric images highlight translucent level sets with all concentrations above 10 μm displayed the same. Bottom-right: concentration (on a log scale) vs. time at five distances from the source on the surface of the dendrite. Inset: a volumetric view from a different angle showing the extent of diffusion into the interior of the dendrite.

version of the CTNG algorithm (McDougal R. et al., 2013) that exploits convexity. Synapses can optionally target their effects to specific 3D compartments (e.g., production of a certain mass of messenger from activation of a metabotropic synapse). Electrophysiology simulations remain simulated as branching 1-dimensional sections as is appropriate given their larger space constants.

NEURON—*via* its `Import3D` library—supports a variety of neuroscience formats for specifying the overall cell morphology, including SWC (Cannon et al., 1998), MorphML (Crook et al., 2007), and Neurolucida ASC (Glaser and Glaser, 1990). Morphologies specified using Neurolucida ASC may include a soma outline; CTNG uses this outline when available to construct a more accurate soma shape than is possible when reading morphologies specified in the other formats. SWC is especially useful as the over 170,000 neuron reconstructions on NeuroMorpho.Org (Ascoli et al., 2007) are all available in SWC format. To study the effects of cell morphology on reaction-diffusion dynamics, modelers may alter a morphology file directly or may modify it using NEURON's standard techniques, such as adding new `Section` objects to insert e.g. spines or using `Section.pt3dchange` and related methods to adjust (x, y, z) or diameter values.

Alternative Strategies

There are two main alternative approaches in the literature for combining 3D reaction-diffusion kinetics with electrophysiology.

The first alternative approach is to have an integrated solver that uses a single mesh. STEPS, for example, simulates ion channel and pump activity on the surface of the 3D mesh (Hepburn et al., 2013); a similar approach was used in the Virtual NEURON study (Brown et al., 2011). Using the same mesh eliminates the possibility of numerical artifacts from coupling, automatically ensures consistent surface areas, and eliminates the possibility of interior surface (e.g., from spines mis-connected at the centroid). We have avoided this approach, instead using 1D electrical with 3D reaction-diffusion as in Grein et al. (2014) to allow the electrical dynamics to be consistent regardless of the dimensionality of the reaction-diffusion simulation, to take advantage of the $\mathcal{O}(n)$ implicit simulation of electrical dynamics on such a 1D-structure (Hines, 1984), and for compatibility with the over 2,000 existing NEURON models (i.e., extending an existing NEURON model with 3D intracellular reaction-diffusion dynamics does not require modifying the existing components, unless a change is desired to their behavior).

The second alternative approach is to use multisimulation; that is, to combine a solver specializing in ion channels and the cable equation like NEURON or MOOSE (RRID:SCR_008031; Dudani et al., 2009) with an external solver specializing in reaction-diffusion simulation. We and our colleagues have used this approach for stochastic 3D model simulation with NEURON Time Warp (Lin et al., 2017). KappaNEURON likewise combines NEURON with the rule-based reaction-diffusion simulator SpatialKappa (Sterratt et al., 2014). Grein et al. (2014) used a similar approach for deterministic 3D simulation coupling NEURON with uG. Additionally, we note that NEURON supports the general multisimulation framework MUSIC (Djurfeldt et al., 2010) which has been used to connect

MOOSE and NeuroRD (Brandi et al., 2011). The multisimulation approach is appealing as it allows each simulator to specialize in its own problem domain, providing a rich set of simulatable features, with each simulator programmed independently. While recognizing the benefits and flexibility of multisimulation, we chose to build our 3D intracellular simulation capability within NEURON to allow for a unified Jacobian matrix, allowing for variable step simulation, to provide consistent coupling semantics, to avoid the need for syncing data between two different potentially parallel tools, and to avoid the need for users to learn two simulator tools.

Special Considerations

Models with exactly zero diffusion of a species that enters or leaves through ion channels pose specific issues in comparing 1D and 3D simulations or 3D simulations with different discretizations. Although mathematically convenient, these models are non-physical as diffusion is necessary to bring a molecule to or through an ion channel. Concentration changes in 1D are based on the active geometry, typically the whole dendrite. Increasing the spatial resolution (e.g., tripling nseg) in a 1D model with no diffusion has no direct effect on concentration dynamics as the volume and total current both scale by the same fraction. In contrast, in a 3D model as ions must enter *via* the surface, with no diffusion, they are trapped there. Reducing voxel edge size by a factor of 2 changes the volume by a factor of 8 but the surface area contained in the voxel by only a factor of 4 on average, leading to a factor of 2 change in the rate at which concentration in surface voxels changes, which could affect ion channel kinetics.

In principle, multigriding could be extended to be used within the chemical dynamics in 1D or 3D; i.e., a species prone to steeper gradients could be simulated on a finer grid, but this risks introducing artifacts, especially in the case of slow or zero diffusion. For example, suppose molecule A on a coarse grid bound with molecule B from a refinement of the grid to form molecule AB. If AB is represented on the same fine grid, then when it dissociates A and B can return to their correct points of origin. If on the other hand, AB is represented on the coarse grid then when it dissociates B could end up in any of the corresponding fine meshes. Thus, even if the diffusion rate was set to zero for all species, molecule B could move by binding to A, entering the coarse grid, and then returning to a different fine grid compartment. A similar problem exists regardless of the relative sizes of the grids if they do not align perfectly. Meshless simulators, like MCell (Stiles et al., 1998), avoid this class of problems entirely at the cost of having to simulate each molecule separately. We note that this problem only pertains to overlapping meshes; separate mesh resolutions on different parts of the cell (e.g., large near the soma, smaller in the distal dendrites) are potentially compatible, although the mesh transition would not in general be expected to align to the boundary between Sections.

We note that the insight gained by a 3D simulation depends on the quality of the 3D mesh. CTNG or any of the alternative rules for converting point-diameter representations into a 3D mesh are inherently approximations as the full shape of the cell is under-determined by the reconstruction data. For a given reconstruction, the mesh quality in NEURON is primarily driven by the choice of `dx` (with smaller values of `dx` giving generally higher quality meshes) as well as the `ics_partial_volume_resolution` and `ics_partial_surface_resolution` options. At least as important is the quality of the reconstruction itself; even when working from the same image stacks, different approaches can lead to logically different reconstructions with branches connected at different points (see e.g., Gillette et al., 2011). Details of the imaging approach can likewise affect the detail present in image stacks of a cell (e.g., dyes may not fill a neuron entirely or a neuron's branches may be amputed by a slice). We recommend that—regardless of metadata annotations—morphologies should be manually reviewed for slice artifacts (e.g., when we randomly selected 21 morphologies, we found that while none were strictly planar several showed minimal z-axis variation), for realistic and non-uniform diameters, for electrical connectivity (no pinch points where the diameter gets very small), and for z-axis errors (some reconstructions show abrupt changes in z values). There is no value in doing a 3D simulation if the 3D morphology is unrealistic.

3D time-series data is in general large and hard to visualize. We deal with the large volume of data by only automatically keeping the current state in memory. The time series of the concentration of a specific species at a specific compartment may be recorded using a `Vector`. For modelers needing to store or visualize all the states at a specific time, simulations may be stopped at a specific time point, where the states are then captured to an appropriate Python data structure. Throughout this paper, we have deliberately illustrated several approaches to visualizing such data: (1) line plots of a single species at a single point as in **Figure 12B**; (2) for traveling waves, plots of the location of the wave front at evenly spaced time points on a 2D projection or slice as in **Figure 12A**; (3) plots of the concentrations at the surface, analagous to the surface segment identities in **Figure 3A**; (4) heatmaps of a slice or projection at a specific time point as in **Figure 7**; and (5) translucent contour maps of concentration level sets at given time points as in **Figure 14**. Example Python code for each type of graph is available in this paper's entry on ModelDB.

Conclusions and Future Directions

From our examples, we make a few observations that apply broadly to other 3D reaction-diffusion simulations: (1) areas that are far from the region of interest do not need to be simulated in 3D; when studying effects at the dendrite-soma intersection, this allows larger dx values than would be possible if the fine distal dendrites also needed to be simulated in 3D. (2) Conversely, such experiments are fundamentally about the role of boundary conditions, therefore other boundaries must be at a far enough distance from the region of interest so as not to affect the results. (3) The accuracy of any results arising from such simulations depends on the accuracy of the voxelized reconstructions. For this reason, we currently recommend using reconstructions in ASC format with a soma outline, as this will provide a non-cylindrical soma. Future versions of NEURON will allow importing a predefined voxelization to more accurately reflect the observed shape, but this will necessarily require matching the surface areas and volumes on the 3D chemical domain with that used for electrophysiology simulation. (4) Regenerative waves have a leading edge that can be plotted on a contour map at regular intervals showing the progression of the wave over time.

NEURON is under continuous development. We intend to improve its support for 3D simulation by streamlining mesh generation: the CTNG algorithm is in-principle embarrassingly parallel and meshes and the data on them could in principle be saved and reused when relaunching NEURON. The first will require reimplementation of CTNG in pure C++ to avoid parallel limitations from Python's GIL, and the second will require an efficient way of validating that the mesh aligns with the 1D skeleton. Both of these enhancements will make it more practical to use the high-quality volume estimates that are necessary to keep 1D-3D coupling errors low but currently require a potentially time-prohibitive initialization. We intend to integrate support for stochastic simulation to study more classes of dynamics and to more faithfully capture phenomena arising from very low concentrations or very small regions (such as spines and boutons). To more accurately capture the dynamics of smaller regions, we intend to add support for optionally including electrodiffusion effects.

We believe that the approach described in this paper provides an intuitive way of incorporating intracellular reaction-diffusion dynamics in computational neuroscience models in a way that more faithfully captures the effects of geometry than is possible in a 1D or 1D + radial simulation. We hope that this allows new insights into the multi-scale processes that underlie our neural activity.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found at: http://modeldb.yale.edu/267018.

AUTHOR CONTRIBUTIONS

CC and RM designed the 3D simulation strategy in consultation with AN. CC implemented it in consultation with AN. LE and RM designed the 3D voxelization strategy and LE implemented it. AN designed and implemented the surface subvoxelization, and improved simulation and voxelization robustness. RM, HG, AN, and CC performed the analyses. All authors drafted, reviewed, edited the manuscript, and contributed to the article and approved the submitted version.

FUNDING

Research reported by this publication was supported by the National Institute of Mental Health (NIMH) and the National Institute of Neurological Disorders and Stroke (NINDS) of the National Institutes of Health under award numbers R01MH086638 and R01NS11613.

ACKNOWLEDGMENTS

We thank Michael L. Hines and William W. Lytton for valuable discussions affecting the design of the interface and its implementation. We thank the Virtual Cell team for various discussions over the years about 3D simulation. We thank the Yale Center for Research Computing for guidance and use of the research computing infrastructure, specifically the Farnam HPC.

REFERENCES

Akar, N. A., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor – a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. (Pavia), 274–282. doi: 10.1109/EMPDP.2019.8671560

Ascoli, G. A., Donohue, D. E., and Halavi, M. (2007). NeuroMorpho.Org: a central resource for neuronal morphologies. *J. Neurosci*. 27, 9247–9251. doi: 10.1523/JNEUROSCI.2055-07.2007

Balluffi, R. W., Allen, S. M., Carter, W. C., and Kemper, R. A. (2005). *Kinetics of Materials, Vol. 1*. Hoboken, NJ: Wiley Online Library. doi: 10.1002/0471749311

Benedikt, M., and Drenth, E. (2019). "Relaxing stiff system integration by smoothing techniques for non-iterative co-simulation," in *IUTAM Symposium on Solver-Coupling and Co-Simulation*, ed B. Schweizer (Cham: Springer International Publishing), 1–25. doi: 10.1007/978-3-030-14883-6_1

Brandi, M., Brocke, E., Talukdar, H. A., Hanke, M., Bhalla, U. S., Kotaleski, J. H., et al. (2011). Connecting MOOSE and NeuroRD through MUSIC: towards a communication framework for multi-scale modeling. *BMC Neurosci*. 12:P77. doi: 10.1186/1471-2202-12-S1-P77

Brown, S.-A., Moraru, I. I., Schaff, J. C., and Loew, L. M. (2011). Virtual NEURON: a strategy for merged biochemical and electrophysiological modeling. *J. Comput. Neurosci*. 31, 385–400. doi: 10.1007/s10827-011-0317-0

Canchi, S., Sarntinoranont, M., Hong, Y., Flint, J. J., Subhash, G., and King, M. A. (2017). Simulated blast overpressure induces specific astrocyte injury in an *ex vivo* brain slice model. *PLoS ONE* 12:e0175396. doi: 10.1371/journal.pone.0175396

Cannon, R. C., Gleeson, P., Crook, S., Ganapathy, G., Marin, B., Piasini, E., et al. (2014). LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Front. Neuroinformatics* 8:79. doi: 10.3389/fninf.2014.00079

Cannon, R. C., Turner, D., Pyapali, G., and Wheal, H. (1998). An on-line archive of reconstructed hippocampal neurons. *J. Neurosci. Methods* 84, 49–54. doi: 10.1016/S0165-0270(98)00091-0

Chen, W., and De Schutter, E. (2017). Time to bring single neuron modeling into 3D. *Neuroinformatics*. 15, 1–3. doi: 10.1007/s12021-016-9321-x

Chiu, C. Q., Lur, G., Morse, T. M., Carnevale, N. T., Ellis-Davies, G. C., and Higley, M. J. (2013). Compartmentalization of GABAergic inhibition by dendritic spines. *Science* 340, 759–762. doi: 10.1126/science.1234274

Cowan, A. E., Moraru, I. I., Schaff, J. C., Slepchenko, B. M., and Loew, L. M. (2012). Spatial modeling of cell signaling networks. *Methods Cell Biol*. 110, 195–221. doi: 10.1016/B978-0-12-388403-9.00008-4

Crook, S., Gleeson, P., Howell, F., Svitak, J., and Silver, R. A. (2007). MorphML: Level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics* 5, 96–104. doi: 10.1007/s12021-007-0003-6

de Oliveira, T. C. G., Carvalho-Paulo, D., de Lima, C. M., de Oliveira, R. B., Santos Filho, C., Diniz, D. G., et al. (2020). Long-term environmental enrichment reduces microglia morphological diversity of the molecular layer of dentate gyrus. *Eur. J. Neurosci*. 52, 4081–4099. doi: 10.1111/ejn.14920

Djurfeldt, M., Hjorth, J., Eppler, J. M., Dudani, N., Helias, M., Potjans, T. C., et al. (2010). Run-time interoperability between neuronal network simulators based on the music framework. *Neuroinformatics* 8, 43–60. doi: 10.1007/s12021-010-9064-z

Douglas, J., and Gunn, J. E. (1964). A general formulation of alternating direction methods. *Numer. Math*. 6, 428–453. doi: 10.1007/BF01386093

Dronne, M.-A., Descombes, S., Grenier, E., and Gilquin, H. (2009). Examples of the influence of the geometry on the propagation of progressive waves. *Math. Comput. Model*. 49, 2138–2144. doi: 10.1016/j.mcm.2008.07.024

Dudani, N., Ray, S., George, S., and Bhalla, U. S. (2009). Multiscale modeling and interoperability in moose. *BMC Neurosci*. 10:P54. doi: 10.1186/1471-2202-10-S1-P54

Ehlinger, D. G., Burke, J. C., McDonald, C. G., Smith, R. F., and Bergstrom, H. C. (2017). Nicotine-induced and d1-receptor-dependent dendritic remodeling in a subset of dorsolateral striatum medium spiny neurons. *Neuroscience* 356, 242–254. doi: 10.1016/j.neuroscience.2017.05.036

Ellingsrud, A. J., Solbrå, A., Einevoll, G. T., Halnes, G., and Rognes, M. E. (2020). Finite element simulation of ionic electrodiffusion in cellular geometries. *Front. Neuroinformatics* 14:11. doi: 10.3389/fninf.2020.00011

Fife, P. (1979). *Mathematical Aspects of Reacting and Diffusing Systems*. Berlin: Springer Verlag. doi: 10.1007/978-3-642-93111-6

Gillette, T. A., Brown, K. M., and Ascoli, G. A. (2011). The DIADEM metric: comparing multiple reconstructions of the same neuron. *Neuroinformatics* 9, 233–245. doi: 10.1007/s12021-011-9117-y

Glaser, J. R., and Glaser, E. M. (1990). Neuron imaging with Neurolucida-a PC-based system for image combining microscopy. *Comput. Med. Imaging Graph*. 14, 307–317. doi: 10.1016/0895-6111(90)90105-K

Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math*. 17, 416–429. doi: 10.1137/0117039

Grein, S., Stepniewski, M., Reiter, S., Knodel, M. M., and Queisser, G. (2014). 1D-3D hybrid modeling-from multi-compartment models to full resolution models in space and time. *Front. Neuroinformatics* 8:68. doi: 10.3389/fninf.2014.00068

Grienberger, C., and Konnerth, A. (2012). Imaging calcium in neurons. *Neuron* 73, 862–885. doi: 10.1016/j.neuron.2012.02.011

Groh, A., Meyer, H. S., Schmidt, E. F., Heintz, N., Sakmann, B., and Krieger, P. (2010). Cell-type specific properties of pyramidal neurons in neocortex underlying a layout that is modifiable depending on the cortical area. *Cereb. Cortex* 20, 826–836. doi: 10.1093/cercor/bhp152

Hagenston, A. M., Fitzpatrick, J. S., and Yeckel, M. F. (2008). MGluR-mediated calcium waves that invade the soma regulate firing in layer V medial prefrontal cortical pyramidal neurons. *Cereb. Cortex* 18, 407–423. doi: 10.1093/cercor/bhm075

Helmstaedter, M., Briggman, K. L., Turaga, S. C., Jain, V., Seung, H. S., and Denk, W. (2013). Connectomic reconstruction of the inner plexiform layer in the mouse retina. *Nature* 500, 168–174. doi: 10.1038/nature12346

Helton, T. D., Zhao, M., Farris, S., and Dudek, S. M. (2019). Diversity of dendritic morphology and entorhinal cortex synaptic effectiveness in mouse Ca2 pyramidal neurons. *Hippocampus* 29, 78–92. doi: 10.1002/hipo.23012

Hepburn, I., Cannon, R., and De Schutter, E. (2013). Efficient calculation of the quasi-static electrical potential on a tetrahedral mesh and its implementation in steps. *Front. Comput. Neurosci*. 7:129. doi: 10.3389/fncom.2013.00129

Hepburn, I., Chen, W., Wils, S., and De Schutter, E. (2012). Steps: efficient simulation of stochastic reaction-diffusion models in realistic morphologies. *BMC Syst. Biol*. 6:36. doi: 10.1186/1752-0509-6-36

Herget, U., Gutierrez-Triana, J. A., Thula, O. S., Knerr, B., and Ryu, S. (2017). Single-cell reconstruction of oxytocinergic neurons reveals separate hypophysiotropic and encephalotropic subtypes in larval zebrafish. *ENeuro* 4. ENEURO.0278-16.2016. doi: 10.1523/ENEURO.0278-16.2016

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. (2005). SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw*. 31, 363–396. doi: 10.1145/1089014.1089020

Hines, M. (1984). Efficient computation of branched nerve equations. *Int. J. Bio-Med. Comput*. 15, 69–76. doi: 10.1016/0020-7101(84)90008-4

Hines, M., Carnevale, T., and McDougal, R. A. (2019). "Chapter: NEURON simulation environment," in *Encyclopedia of Computational Neuroscience*, eds D. Jaeger and R. Jung (New York, NY: Springer New York), 1–7. doi: 10.1007/978-1-4614-7320-6_795-2

Hines, M. L., and Carnevale, N. T. (2000). Expanding neuron's repertoire of mechanisms with nmodl. *Neural Comput*. 12, 995–1007. doi: 10.1162/089976600300015475

Hines, M. L., and Carnevale, N. T. (2001). NEURON: a tool for neuroscientists. *Neuroscientist* 7, 123–135. doi: 10.1177/107385840100700207

Jayant, K., Hirtz, J. J., Jen-La Plante, I., Tsai, D. M., De Boer, W. D., Semonche, A., et al. (2017). Targeted intracellular voltage recordings from dendritic spines using quantum-dot-coated nanopipettes. *Nat. Nanotechnol*. 12, 335–342. doi: 10.1038/nnano.2016.268

Juan, L.-W., Liao, C.-C., Lai, W.-S., Chang, C.-Y., Pei, J.-C., Wong, W.-R., et al. (2014). Phenotypic characterization of C57BL/6J mice carrying the disc1 gene from the 129S6/SvEv strain. *Brain Struct. Funct*. 219, 1417–1431. doi: 10.1007/s00429-013-0577-8

Keller, D. X., Franks, K. M., Bartol Jr, T. M., and Sejnowski, T. J. (2008). Calmodulin activation by calcium transients in the postsynaptic density of dendritic spines. *PLoS ONE* 3:e2045. doi: 10.1371/journal.pone.0002045

Kunst, M., Laurell, E., Mokayes, N., Kramer, A., Kubo, F., Fernandes, A. M., et al. (2019). A cellular-resolution atlas of the larval zebrafish brain. *Neuron* 103, 21–38. doi: 10.1016/j.neuron.2019.04.034

Lasserre, S., Hernando, J., Hill, S., Schuermann, F., de Miguel Anasagasti, P., Abou Jaoudé, G., et al. (2011). A neuron membrane mesh representation for visualization of electrophysiological simulations. *IEEE Trans. Visual. Comput. Graph*. 18, 214–227. doi: 10.1109/TVCG.2011.55

Lin, Z., Tropper, C., McDougal, R. A., Ishlam Patoary, M. N., Lytton, W. W., Yao, Y., et al. (2017). Multithreaded stochastic PDES for reactions and diffusions in neurons. *ACM Trans. Model. Comput. Simul*. 27:7. doi: 10.1145/2987373

Lorensen, W. E., and Cline, H. E. (1987). Marching cubes: a high resolution 3D surface construction algorithm. *ACM Siggraph Comput. Graph*. 21, 163–169. doi: 10.1145/37402.37422

Malik, R., Dougherty, K. A., Parikh, K., Byrne, C., and Johnston, D. (2016). Mapping the electrophysiological and morphological properties of CA 1 pyramidal neurons along the longitudinal hippocampal axis. *Hippocampus* 26, 341–361. doi: 10.1002/hipo.22526

Martinez-Canabal, A., Wheeler, A. L., Sarkis, D., Lerch, J. P., Lu, W.-Y., Buckwalter, M. S., et al. (2013). Chronic over-expression of tgfβ1 alters hippocampal structure and causes learning deficits. *Hippocampus* 23, 1198–1211. doi: 10.1002/hipo.22159

McDougal, R., Hines, M., and Lytton, W. (2013). Water-tight membranes from neuronal morphology files. *J. Neurosci. Methods* 220, 167–178. doi: 10.1016/j.jneumeth.2013.09.011

McDougal, R. A., Hines, M. L., and Lytton, W. W. (2013). Reaction-diffusion in the neuron simulator. *Front. Neuroinformatics* 7:28. doi: 10.3389/fninf.2013.00028

McDougal, R. A., Morse, T. M., Carnevale, T., Marenco, L., Wang, R., Migliore, M., et al. (2017). Twenty years of ModelDB and beyond: building essential modeling tools for the future of neuroscience. *J. Comput. Neurosci*. 42:7. doi: 10.1007/s10827-016-0623-7

Mörschel, K., Breit, M., and Queisser, G. (2017). Generating neuron geometries for detailed three-dimensional simulations using anamorph. *Neuroinformatics* 15, 247–269. doi: 10.1007/s12021-017-9329-x

Nanda, S., Das, R., Bhattacharjee, S., Cox, D. N., and Ascoli, G. A. (2018). Morphological determinants of dendritic arborization neurons in drosophila larva. *Brain Struct. Funct*. 223, 1107–1120. doi: 10.1007/s00429-017-1541-9

Newton, A. J., McDougal, R. A., Hines, M. L., and Lytton, W. W. (2018). Using neuron for reaction-diffusion modeling of extracellular dynamics. *Front. Neuroinformatics* 12:41. doi: 10.3389/fninf.2018.00041

Neymotin, S. A., McDougal, R. A., Sherif, M. A., Fall, C. P., Hines, M. L., and Lytton, W. W. (2015). Neuronal calcium wave propagation varies with changes in endoplasmic reticulum parameters: a computer model. *Neural Comput*. 27, 898–924. doi: 10.1162/NECO_a_00712

Nguyen, V. T., Uchida, R., Warling, A., Sloan, L. J., Saviano, M. S., Wicinski, B., et al. (2020). Comparative neocortical neuromorphology in felids: African lion, African leopard, and cheetah. *J. Compar. Neurol.* 528, 1392–1422. doi: 10.1002/cne.24823

Nikolenko, V., Poskanzer, K. E., and Yuste, R. (2007). Two-photon photostimulation and imaging of neural circuits. *Nat. Methods* 4, 943–950. doi: 10.1038/nmeth1105

Nogueira-Campos, A. A., Finamore, D. M., Imbiriba, L. A., Houzel, J. C., and Franca, J. G. (2012). Distribution and morphology of nitrergic neurons across functional domains of the rat primary somatosensory cortex. *Front. Neural Circuits* 6:57. doi: 10.3389/fncir.2012.00057

Ofer, N., Berger, D. R., Kasthuri, N., Lichtman, J. W., and Yuste, R. (2021). Ultrastructural analysis of dendritic spine necks reveals a continuum of spine morphologies. *Dev. Neurobiol.* 81, 746–757. doi: 10.1002/dneu.22829

Scala, F., Kobak, D., Shan, S., Bernaerts, Y., Laturnus, S., Cadwell, C. R., et al. (2019). Layer 4 of mouse neocortex differs in cell types and circuit organization between sensory areas. *Nat. Commun.* 10, 1–12. doi: 10.1038/s41467-019-12058-z

Schaff, J., Fink, C. C., Slepchenko, B., Carson, J. H., and Loew, L. M. (1997). A general computational framework for modeling cellular structure and function. *Biophys. J.* 73, 1135–1146. doi: 10.1016/S0006-3495(97)78146-3

Sterratt, D. C., Sorokina, O., and Armstrong, J. D. (2014). "Integration of rule-based models and compartmental models of neurons," in *International Workshop on Hybrid Systems Biology* (Vienna: Springer), 143–158. doi: 10.1007/978-3-319-27656-4_9

Stiles, J. R., Bartol, T. M., Salpeter, E. E., and Salpeter, M. M. (1998). "Monte Carlo simulation of neuro-transmitter release using Mcell, a general simulator of cellular physiological processes," in *Computational Neuroscience*, eds J. M. Bower (Boston, MA: Springer), 279–284. doi: 10.1007/978-1-4615-4831-7_47

Takagi, S., Cocanougher, B. T., Niki, S., Miyamoto, D., Kohsaka, H., Kazama, H., et al. (2017). Divergent connectivity of homologous command-like neurons mediates segment-specific touch responses in drosophila. *Neuron* 96, 1373–1387. doi: 10.1016/j.neuron.2017.10.030

Takemura, S.-y., Aso, Y., Hige, T., Wong, A., Lu, Z., Xu, C. S., et al. (2017). A connectome of a learning and memory center in the adult drosophila brain. *Elife* 6:e26975. doi: 10.7554/eLife.26975

Tarusawa, E., Sanbo, M., Okayama, A., Miyashita, T., Kitsukawa, T., Hirayama, T., et al. (2016). Establishment of high reciprocal connectivity between clonal cortical neurons is regulated by the DNMT3B DNA methyltransferase and clustered protocadherins. *BMC Biol.* 14:6. doi: 10.1186/s12915-016-0326-6

Trevelyan, A. J., Sussillo, D., Watson, B. O., and Yuste, R. (2006). Modular propagation of epileptiform activity: evidence for an inhibitory veto in neocortex. *J. Neurosci.* 26, 12447–12455. doi: 10.1523/JNEUROSCI.2787-06.2006

Weiss, L., Jungblut, L. D., Pozzi, A. G., Zielinski, B. S., O'Connell, L. A., Hassenklöver, T., et al. (2020). Multi-glomerular projection of single olfactory receptor neurons is conserved among amphibians. *J. Compar. Neurol.* 528, 2239–2253. doi: 10.1002/cne.24887

Yuste, R., Majewska, A., and Holthoff, K. (2000). From form to function: calcium compartmentalization in dendritic spines. *Nat. Neurosci.* 3, 653–659. doi: 10.1038/76609

Author Disclaimer: The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Neuroscience

ORIGINAL RESEARCH
published: 18 May 2022
doi: 10.3389/fnins.2022.884128

A Scalable Approach to Modeling on Accelerated Neuromorphic Hardware

Eric Müller[1*†], Elias Arnold[1†], Oliver Breitwieser[1†], Milena Czierlinski[1†], Arne Emmel[1†], Jakob Kaiser[1†], Christian Mauch[1†], Sebastian Schmitt[2†], Philipp Spilger[1†], Raphael Stock[1†], Yannik Stradmann[1†], Johannes Weis[1†], Andreas Baumbach[1,3], Sebastian Billaudelle[1], Benjamin Cramer[1], Falk Ebert[1], Julian Göltz[1,3], Joscha Ilmberger[1], Vitali Karasenko[1], Mitja Kleider[1], Aron Leibfried[1], Christian Pehle[1] and Johannes Schemmel[1]

[1] Kirchhoff-Institute for Physics, Heidelberg University, Heidelberg, Germany, [2] Third Institute of Physics, University of Göttingen, Göttingen, Germany, [3] Department of Physiology, University of Bern, Bern, Switzerland

OPEN ACCESS

Edited by:
James B. Aimone,
Sandia National Laboratories (DOE),
United States

Reviewed by:
Srideep Musuvathy,
Sandia National Laboratories (DOE),
United States
Federico Corradi,
Eindhoven University of Technology,
Netherlands

***Correspondence:**
Eric Müller
mueller@kip.uni-heidelberg.de

[†] These authors have contributed equally to this work

Specialty section:
This article was submitted to
Neuromorphic Engineering,
a section of the journal
Frontiers in Neuroscience

Received: 25 February 2022
Accepted: 20 April 2022
Published: 18 May 2022

Citation:
Müller E, Arnold E, Breitwieser O, Czierlinski M, Emmel A, Kaiser J, Mauch C, Schmitt S, Spilger P, Stock R, Stradmann Y, Weis J, Baumbach A, Billaudelle S, Cramer B, Ebert F, Göltz J, Ilmberger J, Karasenko V, Kleider M, Leibfried A, Pehle C and Schemmel J (2022) A Scalable Approach to Modeling on Accelerated Neuromorphic Hardware. Front. Neurosci. 16:884128. doi: 10.3389/fnins.2022.884128

Neuromorphic systems open up opportunities to enlarge the explorative space for computational research. However, it is often challenging to unite efficiency and usability. This work presents the software aspects of this endeavor for the BrainScaleS-2 system, a hybrid accelerated neuromorphic hardware architecture based on physical modeling. We introduce key aspects of the BrainScaleS-2 Operating System: experiment workflow, API layering, software design, and platform operation. We present use cases to discuss and derive requirements for the software and showcase the implementation. The focus lies on novel system and software features such as multi-compartmental neurons, fast re-configuration for hardware-in-the-loop training, applications for the embedded processors, the non-spiking operation mode, interactive platform access, and sustainable hardware/software co-development. Finally, we discuss further developments in terms of hardware scale-up, system usability, and efficiency.

Keywords: hardware abstraction, neuroscientific modeling, accelerator, analog computing, neuromorphic, embedded operation, local learning

1. INTRODUCTION

The feasibility and scope of neuroscientific research projects is often limited due to long simulation runtimes and therefore long wall-clock runtimes, especially for large-scale networks (van Albada et al., 2021). Other areas of neuromorphic research—such as lifelong learning in robotic applications—inherently rely on very long network runtimes to capture physical transformations of their embodiment on the one hand and evolutionary processes on the other. Furthermore, training mechanisms relying on iterative reconfiguration benefit from low execution latencies.

Traditional software-based simulations typically still often rely on general-purpose high-performance computing (HPC) hardware. While some efforts toward GPU-based accelerators provide an intermediate step to improve scalability and runtimes (Yavuz et al., 2016; Abi Akar et al., 2019), domain-specific accelerators—a subset of which are neuromorphic hardware architectures—, have come more and more into the focus of HPC (Dally et al., 2020). Such systems specifically aim to improve on performance and scalability issues—both, in the strong and in the weak scaling cases. Particularly, the possibility to achieve high throughput at low execution latencies can pose a crucial advantage compared to massively parallel simulations.

The BrainScaleS (BSS) neuromorphic architecture is an accelerator for spiking neural networks based on a physical modeling approach. It provides a neuromorphic substrate for neuroscientific modeling as well as neuro-inspired machine learning. Earlier work shows its scalability in wafer-scale applications, emulating up to 200 k neurons and 40 M synapses (Schmitt et al., 2017; Kungl et al., 2019; Müller et al., 2020b; Göltz et al., 2021), as well as its energy-efficient application as standalone system with 512 neurons and 128 k synapses in use cases related to edge computing (Stradmann et al., 2021; Pehle et al., 2022). Compared to the biological time domain, the model dynamics evolve on a 1.000-fold accelerated time scale making the system interesting for iterative and long-running experiments. Constant model emulation speed is attractive for hardware users. However, it often comes with algorithmic challenges. Similar to other neuromorphic systems based on the physical modeling concept, neuroscientific modeling on the BrainScaleS-2 (BSS-2) system requires a translation from a user-defined neural network experiment to a corresponding hardware configuration. BSS-2 operates in continuous time and does not support pausing or resuming of model dynamics. The algorithmic problem statement is global for the user-defined experiment. Therefore, the complexity of the translation process cannot be reduced by partitioning the problem. Many neuromorphic systems have been providing software solutions to solve this problem and enable higher-level experiment descriptions. We developed a software stack for the wafer-scale BrainScaleS-1 (BSS-1) system covering the translation of user-defined experiments from the PyNN high-level domain-specific description language to a hardware configuration (Müller et al., 2020b). While the BSS-2 neuromorphic architecture hasn't been scaled to full wafer size yet, other feature additions such as structured and non-linear neuron models as well as single instruction, multiple data (SIMD) processors make BSS-2 an appealing substrate for modeling of smaller network sizes. In particular, a new challenge is posed by the introduction of SIMD processors in BSS-2 as programmable elements with real-time vectorized access to many observables from the physical modeling substrate. Observables such as correlation sensors are implemented in the synapse circuits, yielding an immense computational power by offloading computational tasks into the analog substrate. Moreover, the configuration space increases significantly: in addition to a static configuration of network topology, the processors allow for flexible handling of dynamic aspects such as structural plasticity, homeostatic behavior, and virtual environments enabling robotic or other closed-loop applications. This "hybrid" approach requires modeling support in the software stack integrating code generation for the processors as well as mechanisms to parameterize plasticity algorithms and other code parts running on the embedded processors.

We present recent modeling advances on the substrate showcasing new features of the system: complex neurons (section 3.1), neuro-inspired machine-learning experiments (section 3.2), closed-loop sensor-motor interaction (section 3.3) and non-spiking operation (section 3.4). We demonstrate network-attached accelerator operation as well as standalone operation. We argue that for successful and sustainable advances in the usage of neuromorphic systems a deep integration between hardware and software is crucial on all layers. The complete system—software together with hardware—needs to be explicitly designed to support access with varying abstraction levels: high-level modelers, expert users and component developers possess different perceptions of the system; in order for a modeling substrate to be successful, it has to deliver on all of these aspects.

1.1. The BrainScaleS-2 Hardware

In this section, we introduce the BSS-2 system and highlight the basic hardware design which is guiding the development of the accompanying software stack. For a more in depth description of the hardware aspects of the BSS-2 system refer to Aamir et al. (2018), Schemmel et al. (2020), and Pehle et al. (2022).

BrainScaleS is a family of mixed-signal neuromorphic accelerators; analog circuits emulate neuron as well as synapse dynamics in continuous time, while communication of spike events and configuration data is handled in the digital domain. In this paper we focus on the single chip BSS-2 system with 512 neurons and 131.072 synapses circuits (see **Figure 1A**). Due to the intrinsic properties of the silicon substrate, the physical emulation of neuron dynamics is 1.000 faster than in biological real time. Currently, the BSS-2 ASIC is integrated in a stationary laboratory setup (**Figure 1C**), as well as in a portable system (**Figure 1B**).

The high configurability of the BSS-2 system facilitates many different applications (see Section 3). For example, the neuron circuits replicate the dynamics of the adaptive exponential integrate-and-fire (AdEx) neuron model (Brette and Gerstner, 2005) and are individually configurable by a number of analog and digital parameters. By connecting several neuron circuits together to form one logical neuron, more complex multi-compartmental neuron models can be formed and the synaptic fan-in of individual neurons can be increased; a single neuron circuit on its own has access to 256 synapses (**Figure 1D**). In addition to the emulation of biologically plausible neural networks, BSS-2 also supports non-spiking artificial neural networks (ANNs). This is facilitated by disabling spiking as well as the exponential, the adaptive and the leak current of the AdEx neuron model, turning the neuron circuits into simple integrators. Furthermore, the high configurability allows countering device-specific deviations between analog circuits which result from imperfections during the manufacturing process (see Section 2.3.6).

The digital handling of spike events enables the implementation of various network topologies. All spikes, including external spikes as well as spikes generated in the neuron circuits, are collected in the "event handling" block and subsequently routed off chip for recording or *via* the synapse drivers and synapses to post-synaptic on-chip partners (cf. **Figure 1D**). One of the key challenges during experiment setup is the translation of neural networks to valid hardware configurations. This includes assigning specific neuron circuits to the different neurons in the network as well as routing events between neurons (cf. Sections 2.3.2, 2.3.3).

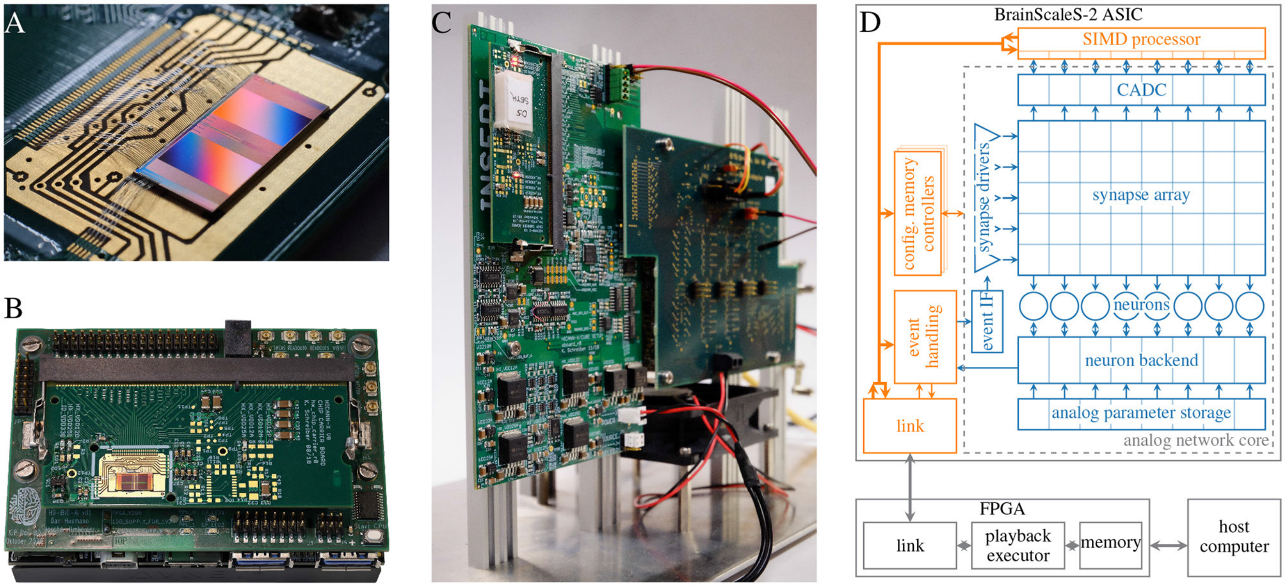


FIGURE 1 | Overview of the BSS-2 system. **(A)** BSS-2 ASIC bonded to a carrier board. The ASIC is organized in two hemispheres each hosting 256 neurons and the accompanying synapse matrix (cf. **D**). **(B)** Portable BSS-2 system. **(C)** Laboratory setup. **(D)** Overview over the signal flow in the BSS-2 system. The depicted analog neural network core and SIMD processor represent one of the two hemispheres visible in **(A)**, which are mirrored vertically below the neurons.

Apart from forwarding spikes, the synapse circuits are also equipped with analog correlation sensors which measure the causal and anti-causal correlation between pre- and post-synaptic spikes. The measured correlation can be accessed by two columnar ADCs (CADCs), which measure correlations row-wise in parallel and can be used in the formulation of plasticity rules (cf. Sections 3.2, 3.3). An additional analog-to-digital converter (ADC), the so-called membrane ADC (MADC), offers the possibility to record single neurons with a higher temporal and value resolution.

Aside the analog neural network core, two embedded SIMD processors, based on the Power$^{\text{TM}}$ architecture (PowerISA, 2010), which allow for arbitrary calculations and reconfigurations of the BSS-2 ASIC during hardware runtime and are the experiment master in standalone operation. They are equipped with 16 KiB static random-access memory (SRAM) memory each and feature a weakly-coupled vector unit (VU), which can access the hemisphere-local synapse matrix as well as the CADC.

Communication to the BSS-2 ASIC as well as real-time runtime control is handled by a field-programmable gate array (FPGA). It provides memory buffers for data received from a host computer or from the chip, with which it orchestrates experiment executions in real time (see Section 2.1). To allow for more complex programs and larger data storage, the on-chip processors can access memory connected to the FPGA.

The software stack covered in this paper handles all the necessary steps to turn high-level experiment descriptions into configuration data, spike stimuli or programs for the on-chip SIMD processor.

In the following we will at first describe the BSS-2 Operating System (BSS-2 OS) in Section 2, before showcasing several applications in Section 3. We conclude the paper with a discussion in Section 4.

2. BRAINSCALES-2 OPERATING SYSTEM

This section introduces key concepts and software components that are essential for the operation of BrainScaleS-2 systems. First, we introduce the workflow of experiments incorporating BSS-2, derive an execution model and specify common modes of operation in Section 2.1. Continuing, we give a structural overview of the complete software stack including the foundation developed in Müller et al. (2020a) in Section 2.2. Following this, we motivate key design decisions and show their incorporation into the development of the software stack in Section 2,3. Finally, we describe advancements in platform operation toward seamless integration of BSS-2 as an accelerator resource in multi-site compute environments in Section 2.4.

Higher abstraction layers scale down the level of required hardware detail knowledge. Naturally, such abstractions impose constraints on and reduce the flexibility of system usage introducing tradeoffs. Therefore, there are tradeoffs between abstraction level and the flexibility to exploit system capabilities. In the following, we explain existing tradeoffs at their occurrence.

2.1. Experiment Workflow

Unlike numerical simulations, which are orchestrated as number-crunching on traditional computers, experiments on BSS-2 are more akin to physical experiments in a traditional lab. Just like for these there is an *initialization* phase, which ensures the correct configuration of the system for this particular experiment and a *real-time* section, where the network dynamics are recorded and

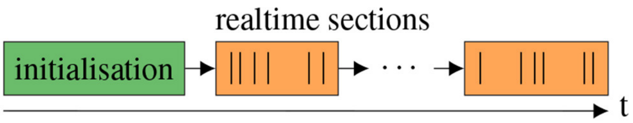


FIGURE 2 | Time evolution of a single execution instance. The initialization is followed by possibly multiple real-time executions with input spike-trains represented by vertical lines.

the actual emulation happens. If multiple emulations share (parts of) the configuration, those experiments can be composited by concatenating the trigger commands for both input and recording (see **Figure 2**).

The fundamental physical nature of the emulation on BSS-2 requires these control commands to be issued with very high temporal precision as the dynamics of the on-chip circuitry can neither be interrupted nor exactly repeated. To achieve this, the accompanying FPGA is used to play-back a sequence of instructions with clock-precise timing, in the order of 10 ns. In order to limit the FPGA firmware complexity, the play-back unit is restricted to sequential execution, which includes blocking instructions (used for times without explicit interaction), but excludes branching instructions. Concurrently to the FPGA-based instruction sequence execution, the embedded single instruction, multiple data central processing units (SIMD CPUs) can be configured to perform readout of observables and arbitrary alterations to the hardware configuration. This means that conditional decisions, e.g., the issuance of rewards, can be performed either *via* the SIMD CPU if they are not computationally too complex or *via* synchronization with the executing host computer which in the current setup has no guaranteed timing.

The initialization phase typically includes time-consuming write operations to provide an initial state of the complete hardware configuration. This is due to both, the amount of data to be transmitted, e.g., for the synapse matrix, and required settling-time for the analog parameters. Since this can take macroscopic amounts of time, at least around 100 µs due to round-trip latency, around 100 ms for a complete reconfiguration, back-to-back concatenation of real-time executions is needed to keep their timeshare high and therefor the configuration overhead low.

Due to the hardware's analog speed-up factor compared to typical biological processes, a single real-time section can be short compared to the initialization phase. Therefore, we concatenate multiple real-time sections after a single initialization phase to increase the real-time executions' timeshare. In the following, this composition is called execution instance and is depicted in **Figure 2**.

Alternatively, instead of this asynchronous high-throughput operation, the low minimal latency allows for fast iterative workflows with partial reconfiguration, e.g., iterative reconfiguration of a small set of synaptic weights.

Based on this we differentiate between three modes of operation. First, in batch-like operation one or multiple execution instances are predefined and run on hardware. Second, in the so-called hardware in-the-loop case hardware runs are executed iteratively where the results of previous runs determine the parameters of successive runs. Last, in closed-loop operation is characterized by tightly coupling the network dynamics of the analog substrate to the experiment controller, either the SIMD CPU or the control host.

2.2. Software Stack Overview

Structuring software into well-defined layers is vital for keeping it maintainable and extendable. The layers are introduced and implemented *via* a bottom-up approach matching the order of requirements in the current stage of the hardware development and commissioning process. This means, that first raw data exchange and transport from and to the hardware *via* the communication layer is established. Subsequently, the hardware abstraction layer implements translation of typed configuration, e.g., enabling a neuron's event output, to and from this raw data. On this level, the calibration layer allows to programmatically configure the analog hardware to a desired working point. Then, hardware-intrinsic relations between configurables and their interplay in experiments (cf. Section 2.1), is encapsulated in a graph structure. Lastly, automated generation of hardware configuration from an abstract network specification enables embedding into modeling frameworks for high-level usage. **Figure 3** gives a graphical overview of this software architecture[1].

2.2.1. Communication

From the software point of view, the first step to utilize hardware systems is the ability to exchange data. With proper abstraction the underlying transport protocol and technology are interchangeable. Communication is therefore structured into a common *connection* interface *hxcomm*[2] that supports various back-ends.

For most hardware setups, we use a custom, reliable regarding data integrity, transport protocol on top of the user datagram protocol (UDP), *Host-ARQ* provided by *sctrltp*[3]. Additionally, we support connection to hardware design simulations *via flange*[4], compare Section 3.6 for both the use during debugging of current and unit testing of future chip generations. Multi-site workflows are transparently enabled already at this level *via* the micro scheduler *quiggeldy*[5].

2.2.2. Hardware Abstraction

A major aspect of any system configuration software is *hardware abstraction*, which encapsulates knowledge about the raw bit configuration, e.g., that bit *i* at address *j* corresponds to enabling neuron *k*'s event output. It therefore decouples hardware usage and detailed knowledge about its memory layout, which is an important step toward providing hardware access beyond the group of developers of the hardware. Responsibility of this layer

[1] All the repositories mentioned in the following are available at https://github.com/electronicvisions under the *GNU Lesser General Public License v2/v3*.
[2] *hxcomm* is available at https://github.com/electronicvisions/hxcomm.
[3] *sctrltp* is available at https://github.com/electronicvisions/sctrltp.
[4] *flange* is available at https://github.com/electronicvisions/flange.
[5] *quiggeldy* is available at https://github.com/electronicvisions/hxcomm.

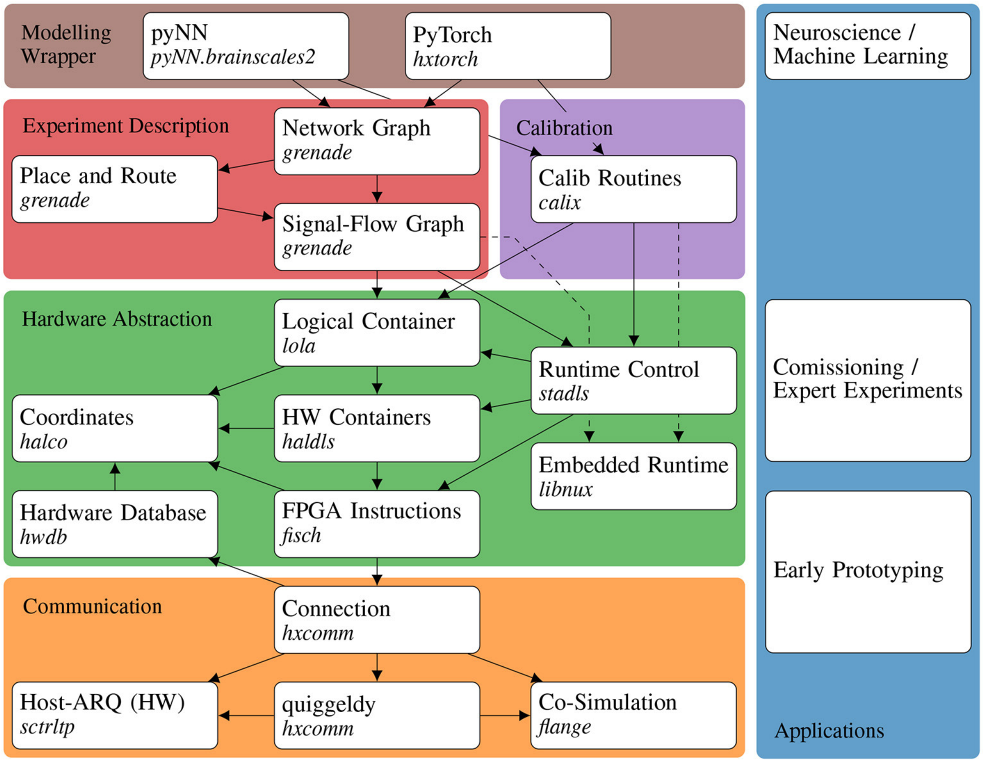


FIGURE 3 | Overview of the BSS-2 software architecture and its applications. Left side: Colored boxes in the background represent the separation of the software into different concerns. White boxes represent individual software APIs or libraries with their specific repositories names and dependencies. Right side: Various applications concerning different system aspects. The arrows represent dependencies in the stack, where the dependent points to its dependencies. For embedded operation additional dependencies on *libnux* are needed (dashed arrows).

can be compared to device drivers. The layers provide an abstract software representation of various hardware components, such as synaptic weights on the chip or values of supply voltages on the periphery board, as well as their control flow.

Within this category the lowest layer is *fisch*[6] (FPGA Instruction Set arCHitecture), the abstraction of FPGA instructions. Combined with communication software this is already sufficient to provide an interface for prototyping in early stages of system development, i.e., the possibility to manually read and write words at memory locations. With knowledge of the hardware's memory layout this allows specifying addresses and word values directly, e.g., bit i (and all other bits in this word with possibly unrelated effects) at address j which then enables the neuron k's event output.

The heterogeneous set of entities on the hardware as well as their memory layout is arranged *via* geometric pattern and contain symmetries, e.g., a row of neurons or a matrix of synapses. An intuitive structure of this fragmented address space is provided by the *coordinate* layer *halco*[7]. It represents hardware components by custom ranged types that can be converted to other corresponding coordinate types, e.g., a `SynapseOnSynapseRow` as a ranged integer $i \in [0, 256)$, that allows conversion to a neuron column (see Müller et al., 2020a).

A software representation of the configuration space of hardware components is implemented by the *container* layer *haldls*[8]. For example a `NeuronConfig` contains a boolean parameter for enabling the spike output. These configuration containers are translatable (e.g., a neuron container represents one, but not a specific one, of the neurons) and also define methods for de- and encoding between their abstract representation and the on-hardware data format given a location *via* a supplied *coordinate*. A logical function- instead of a hardware subsystem-centered container collection is implemented by the *lola*[9] layer. For example the `AtomicNeuron` collects the analog and digital configuration of a single neuron circuit, which is scattered over two digital configurations and a set of elements in the analog parameter array.

The *runtime control* layer *stadls*[10] provides an interface to describe timed sequences of read and write instructions of pairs of coordinates and containers, e.g., changing the synaptic weight of synapse i, j at time t, as well as event-like response data,

[6] *fisch* is available at https://github.com/electronicvisions/fisch.
[7] *halco* is available at https://github.com/electronicvisions/halco.
[8] *haldls* is available at https://github.com/electronicvisions/haldls.
[9] *lola* is available at https://github.com/electronicvisions/haldls.
[10] *stadls* is available at https://github.com/electronicvisions/haldls.

e.g., spikes or ADC samples. These timed sequences, also called playback programs, can then be loaded to and executed on the FPGA which records the response data. Afterwards, the recorded data is transferred-back to the host computer.

We track the constitution of all hardware setups in a database, *hwdb*[11]. It is used for compatibility checks between hardware and software as well as for the automated selection of stored calibration data. We also use it to provide the resource scheduling service with information about all available hardware systems.

This set of layers is feature-complete to formulate arbitrary hardware-compatible experiments and was used as basis for experiments in Schemmel et al. (2020), Göltz et al. (2021), Klassert et al. (2021), Czischek et al. (2022), and Cramer et al. (2022).

2.2.3. Embedded Runtime

In addition to the controlling host system, the two SIMD CPUs on the BSS-2 ASIC require integration into the BSS-2 OS. To enable users to efficiently formulate their programs, we provide a development environment based on C++. It specifically consists of a cross-compilation toolchain based on gcc (GNU Project, 2018) that has been adapted to the custom SIMD extensions of the integrated microprocessors (Müller et al., 2020a). More abstract functionality is encapsulated in the support library *libnux*[12], which provides various auxiliary functionality for experiment design. Moreover, the hardware abstraction layer of the BSS-2 OS (cf. Section 2.2.2) supports the SIMD CPUs as an additional cross-compiled target for configuration containers as well as coordinates.

2.2.4. Calibration

In order to tune all the analog hardware parameters to the requirements given by an experiment, we provide a calibration framework, *calix*[13]. For example, an experiment might require a certain set of synaptic time constants for which analog parameters are to be configured while counteracting circuit inequalities. In Section 2.3.6, this layer's design is explained in detail. The Python module supplies a multitude of algorithms and calibrations for each relevant component of the circuitry: A calibration provides a small experiment based on the hardware abstraction layer (see Section 2.2.2), which is executed on the chip for characterization. An iterative algorithm then decides how configuration parameters should be changed in order to match the measured data with given expectations.

The user-interfacing part provides functions that take a set of target parameters and return a serializable calibration result that can be injected to experiment toplevels (cf. Section 2.2.6). Additionally, we have the option to calibrate the analog circuits locally on chip, using the embedded processors. Aside of enabling arbitrary user-defined calibrations, we provide default calibrations for spiking operation (cf. for example Sections 3.1, 3.2), and non-spiking matrix-vector multiplication (cf. Section 3.4) for convenient entry. They are generated nightly *via* continuous deployment (CD).

2.2.5. Experiment Description

With rising experiment and network topology complexity, a coherent description ensuring topology and data-flow correctness becomes beneficial. Therefore, a signal-flow graph is defined representing the hardware configuration and experiment flow. Compilation and subsequent execution *via* the hardware abstraction layer (cf. Section 2.2.2), of this graph in conjunction with supplied data, e.g., spike events, then forms an experiment execution. The applied execution model follows the experiment workflow described in Section 2.1. It, therefore, restricts flexibility to enable network-topology-based experiment descriptions and the separation of data-flow description and data.

While this aids in construction of complex experiments, detailed knowledge of configuration and its interplay is still required. Solving this, a high-level abstract representation of neural network topology building on top of the signal-flow graph description is developed. An automated translation from this high-level abstraction to a valid hardware configuration is handled by a place-and-route algorithm. This enables hardware usage without detailed knowledge of event routing capabilities and interplay of configuration. While relieving users from providing a valid hardware configuration, this automatism requires tradeoffs to be made between the computational complexity of the algorithms and the size of the explored configuration space to find a matching hardware configuration for a given abstract network representation.

This layer is contained in *grenade*[14], short for GRaph-based Experiment Notation And Data-flow Execution. Its design is explained in detail in Section 2.3.2.

2.2.6. Modeling Wrapper

Various back-end-agnostic modeling languages emerged to provide access to various simulators or neuromorphic hardware systems to a wide range of researchers. The BSS-2 software stack comprises wrappers to two of such modeling frameworks: PyNN (Davison et al., 2009) *via pyNN.brainscales2*[15] and PyTorch (Paszke et al., 2019) *via hxtorch*[16] (Spilger et al., 2020). Their goal is to provide a common user interface and to embed different back-ends into an existing software ecosystem. This allows users to benefit from a consistent and prevalent interface and integration into their established work-flow. The design of these layers' integration with BSS-2 is explained in detail in Section 2.3.4 for PyNN and in Section 2.3.5 for PyTorch.

2.3. Software Design

We base the full-stack software design on the principles laid out in Müller et al. (2020a). We use C++ as the core language to ensure high performance and make use of its compile-time expression evaluation and template metaprogramming

[11] *hwdb* is available at https://github.com/electronicvisions/hwdb.

[12] *libnux* is available at https://github.com/electronicvisions/libnux.

[13] *calix* is available at https://github.com/electronicvisions/calix.

[14] *grenade* is available at https://github.com/electronicvisions/grenade.

[15] *pyNN.brainscales2* is available at https://github.com/electronicvisions/pynn-brainscales.

[16] *hxtorch* is available at https://github.com/electronicvisions/hxtorch.

capabilities. Due to the heterogeneous hardware architecture we employ type safety for logical correctness and compile-time error detection. Serialization support of configuration and control flow enables multi-site workflows as well as archiving of experiments.

In the following, we show enhancements of the hardware abstraction layer (see Section 2.2.2), introduced in Müller et al. (2020a) as well as design decisions for the full software stack with high-level user interfaces. First, support for multiple hardware revisions is shown in (Section 2.3.1). Then, the signal-flow graph-based experiment notation is derived in Section 2.3.2. Following, an abstract network description explained in Section 2.3.3 closes the gap to the modeling wrappers in PyNN (cf. Section 2.3.4) and PyTorch (cf. Section 2.3.5). Closing, the calibration framework is described in Section 2.3.6.

2.3.1. Multi-Revision Hardware Support

As platform development progresses, new hardware revisions require software support. This holds true for both, the ASIC and the surrounding support hardware like the FPGA and system printed circuit boards (PCBs). Additionally, the platform constitution evolves, e.g., by introduction of a mobile system with still one chip but different support hardware or a multi-chip setup.

After a potential development of a second revision, a heterogeneous set of hardware setups may co-exist. For one generation of chips, it is typically possible to combine different revisions with different surrounding hardware configurations, leading to a number of combinations given by the Cartesian product $N = M_{\mathrm{ASIC}} \times M_{\mathrm{Platform}_1} \times \cdots \times M_{\mathrm{Platform}_P}$, where M_{Platform_i} is the number of configurations for a given part of the platform, e.g., the FPGA and M_{ASIC} is the revision of the BSS-2 ASIC.

We provide simultaneous software support by dependency separation and extraction of common code for each affected component across all affected software layers. This way, code duplication is minimized, maintainability of common features is ensured and divergence of software support is prevented. Moreover, phasing-out or retiring hardware revisions is possible without effecting the software infrastructure of other revisions. The to be implemented software reduces to $N' = M_{\mathrm{ASIC}} + M_{\mathrm{Platform}_1} + \cdots + M_{\mathrm{Platform}_P}$ constituents, the combinations are rolled-out automatically. We use `C++` namespaces for separation and `C++` templates for common code, which depends on the individual platform's constituents.

2.3.2. Signal-Flow Graph-Based Experiment Notation

As stated in Section 2.2.2, the hardware abstraction developed in Müller et al. (2020a) is already feature-complete to formulate arbitrary hardware-compatible experiments. However, it lacks a representation of intrinsic relations between different configurable entities. For example, the hard-wired connections between synapse drivers and synapse rows are not represented in their respective configuration but only given implicitly.

Neural networks are predominantly described as graphs. For spiking neural networks single neurons or collections thereof and their connectivity form a graph (Goddard et al., 2001; Gewaltig and Diesmann, 2007; Davison et al., 2009). In machine-learning, the two major frameworks PyTorch (Paszke et al., 2019) and Tensorflow (Abadi et al., 2016) use a graph-based representation of tensor computation or are moving into this direction (PyTorch's JIT intermediate representation Facebook Inc., 2021a and XLA back end Facebook Inc., 2021b; Suhan et al., 2021).

Inspired by this, we implement a signal-flow graph-based experiment abstraction. A signal-flow graph (Mason, 1953) is a directed graph, where vertices receive signals from their in-neighborhood, perform some operation, and transmit an output signal to their out-neighborhood. We integrate this representation at the lowest possible level to fully incorporate all hardware features without premature abstraction.

For BSS-2, the graph-based abstraction is applied at two granularities (see **Figure 4**). First, the initial static network configuration as well as virtualized computation using the on-chip embedded processors is abstracted as a signal-flow graph. Second, data-flow between multiple individual real-time experiments distributed over chips and time are described as a graph.

The signal-flow graph representation yields multiple advantages. Type safety in the graph constituents facilitates experiment correctness regarding on-chip connectivity and helps to avoid inherently dysfunctional experiments already during specification. Debugging benefits from visualization of the graph representation, which directly contains implicit on-chip connectivity. Finally, the signal-flow graph is the ideal source of relationship information for on-chip entity allocation optimization or merging of digital operations.

However, the actual signals are not part of the signal-flow graph representation. They are either provided separately (e.g., external events serving as input), will only be present locally upon execution (e.g., synaptic current pulses) or will be generated by execution (e.g., recorded external events). We implement the experiment workflow described in Section 2.1 consisting of an initial static configuration followed by a collection (batch) of time evolutions (see **Figure 2**).

The signal-flow graph is a recipe for compilation toward the lower-level hardware abstraction layer (cf. Müller et al., 2020a), and eventual execution. The specific implementation of the compilation and execution process is separate from the graph representation in order to allow extensibility and multiple solutions for different requirement profiles. Here, we present a just-in-time (JIT) execution implementation. It supports both, spiking and non-spiking experiments. For every execution instance, the local subgraph is compiled into a sequence of instructions, executed and its results processed in order for them to serve as inputs for the out-neighborhood. While it is feature-complete for the graph representation, it introduces close coupling between the execution on the neuromorphic hardware and the controlling host computer. Host-based compilation can be performed concurrently to hardware execution, increasing parallelism. **Figure 5** shows concurrent execution of multiple execution instances (**Figure 5A**) and the compilation and execution of a single execution instance (**Figure 5B**).

2.3.3. Abstract Network Description

The signal-flow graph-based notation from Section 2.3.2 eases creation of correct experiments while minimizing implicit

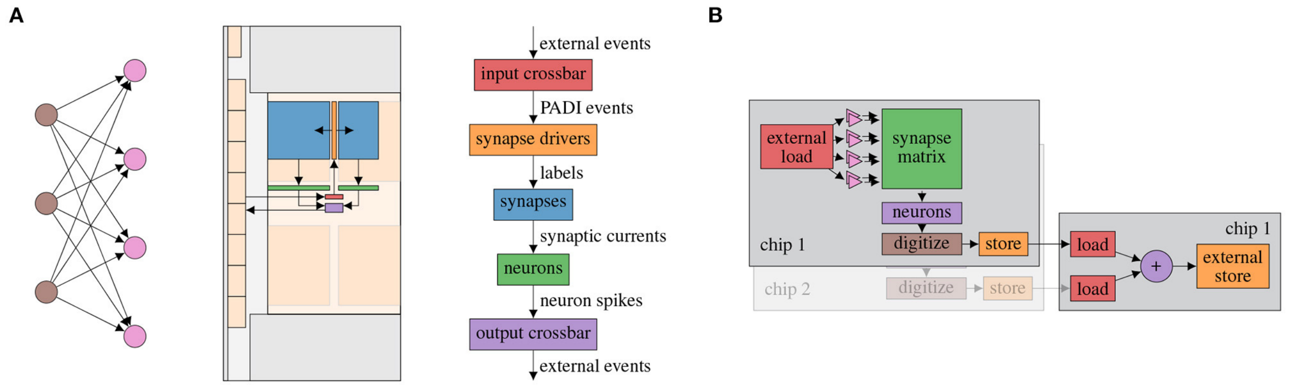


FIGURE 4 | Signal-flow graph-based experiment abstraction on BSS-2. **(A)** Placed feed-forward network represented as signal-flow graph. (Left) Abstract network; (Middle) Actual layout on the chip, the arrows represent the graph edges; (Right) The network graph structure enlarged with signal type annotation on the edges. The color links the same entities in the middle (chip schematic) and right subfigure (vertical data-flow graph). **(B)** Non-spiking network distributed over two physical chips, adapted from Spilger et al. (2020). The result of two matrix multiplications on chips 1 and 2 is added on chip 1. The latter execution instance depends on the output of the two former instances.

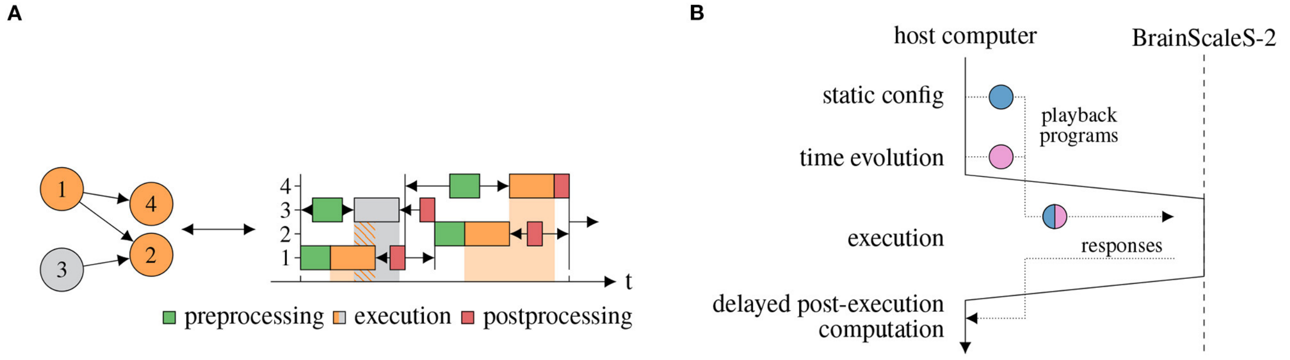


FIGURE 5 | JIT compilation and execution of signal-flow graph of multiple execution instances and within a single execution instance. **(A)** JIT execution of a graph on two physical chips, adapted from Spilger et al. (2020). Left: Execution instance 3 is to be executed on another physical chip than the other execution instances. Right: The execution of instance 3, depicted in gray, can be performed concurrently to execution instance 1. **(B)** JIT compilation and execution of a single execution instance subgraph. First, the static configuration is extracted by a vertex visit and transformed to hardware configuration where applicable. Then, the real-time execution is built by a vertex visit. This built program is executed on the neuromorphic hardware and results are transmitted back to the host computer. Finally, delayed digital operations, which require output data from the execution, are performed on the host computer.

knowledge. However, knowledge of hardware routing capabilities is still required to create a graph-based representation of the hardware configuration which performs as expected. This should not be required to formulate high-level experiments. To close this gap, an abstract representation similar to PyNN (Davison et al., 2009), consisting of populations as collections of neurons and projections as collections of synapses, is developed. Given this description, an algorithm finds an event routing configuration to fulfill the abstract requirements and generates a concrete hardware configuration. This step is called routing. **Figure 6** visualizes an abstract network description and one corresponding hardware configuration.

2.3.4. Integration of PyNN

When it comes to modeling spiking neural networks, a widely used API is PyNN (Davison et al., 2009). It is supported by various neural simulators like NEST (Gewaltig and Diesmann, 2007), NEURON (Hines and Carnevale, 2003), and Brian (Stimberg et al., 2019), as well as by neuromorphic hardware platforms like SpiNNaker (Rhodes et al., 2018) or the predecessor hardware of BSS-2: BSS-1 (Müller et al., 2020b) and Spikey (Brüderle et al., 2009). With the aim of easy access to BSS-2, we expose its hardware configuration *via* the PyNN interface. The module `pyNN.brainscales2` implements the PyNN-API for BSS-2. It offers a custom cell type, `HXNeuron`, which corresponds to a physical neuron circuit on the hardware and replicates the `lola.AtomicNeuron` from the hardware abstraction layer, see section 2.2.2. This allows to set parameters directly in the hardware domain and gives expert users the possibility to precisely control the hardware configuration while at the same time take advantage of high-level features such as neuron populations and projections. **Figure 7** illustrates how

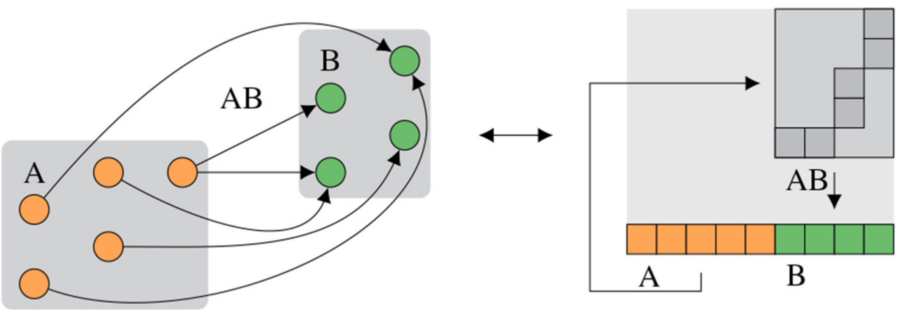
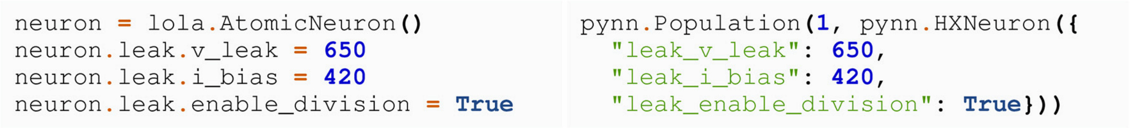


FIGURE 6 | Abstract network notation. Population A consisting of five neurons is connected to population B consisting of four neurons *via* projection AB. (Left) Abstract network. (Right) Placed and routed on the hardware, where the projection AB consists of synapses in the two-dimensional synapse matrix and the populations A and B are located in the neuron row, compare **Figure 1D**.



```
neuron = lola.AtomicNeuron()
neuron.leak.v_leak = 650
neuron.leak.i_bias = 420
neuron.leak.enable_division = True
```

```
pynn.Population(1, pynn.HXNeuron({
  "leak_v_leak": 650,
  "leak_i_bias": 420,
  "leak_enable_division": True}))
```

FIGURE 7 | Comparison between `lola.AtomicNeuron` and `pynn.HXNeuron`.

these parameters are available in the corresponding interfaces. An additional neuron type supporting the translation from neuron model parameters in SI units is currently in the planning. Otherwise, the PyNN program looks the same as for any other back end. Since the PyNN-API is free from hardware placement specifications, they are algorithmically determined by mapping and routing in *grenade* (cf. Section 2.3.3). This step is performed automatically upon invocation of `pynn.run()`, so that the user is not required to have any particular knowledge about event routing on the hardware. Nevertheless, the interface allows that an experimenter can adjust any low-level configuration aside from neuron parameters and synaptic weights.

To exploit the full potential of the accelerated hardware the software implementation's overhead shall be minimal. **Figure 8** presents runtime and memory consumption analysis of the whole PyNN-based stack for a high spike count benchmark experiment. 12 neurons are excited by a regular spike train with 1 MHz frequency and their activity is recorded for one second. These settings are chosen as they roughly equate to the maximum recording rate without loss.

The initial overhead of importing `Python` libraries and setting up the PyNN environment only needs to be performed once for every experiment and is independent of the network topology itself. Run time on hardware is about 1.5 s of which roughly 125 ms are initial configuration and 278 ms are transmission of the input spike train. Post-processing the 1.2×10^7 received spikes (*fisch* and *grenade*) takes about 1.9 s, i.e., in the same order of magnitude as the actual hardware run. Peak memory consumption is reached during post-processing of results obtained after the hardware execution which corresponds to roughly three times the minimum memory footprint of the recorded spike train. With this the stack is well suited to also handle experiments with high spike count without introducing a bottleneck.

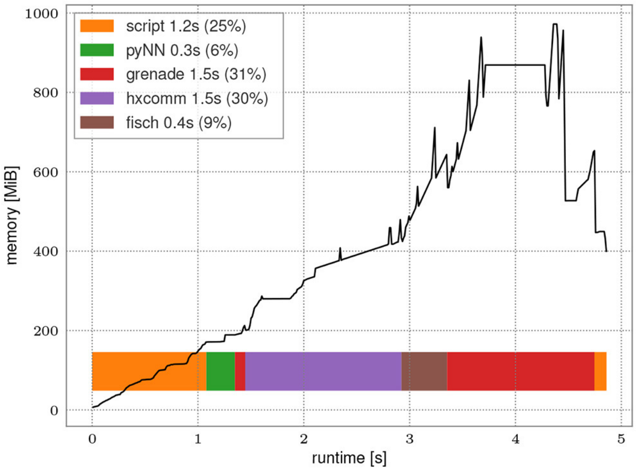


FIGURE 8 | Run time analysis of a PyNN-based experiment with large spike count. Population of 12 neurons is excited by a regular spike train with frequency of 1 MHz. The network is emulated for 1 s on hardware resulting in 1.2×10^7 spike events. The black line represents memory consumption during execution. Horizontal bars represent time consumption in software layers. The annotations in the legend present the individual run time of steps and percentage of the overall run time.

2.3.5. Integration Into PyTorch

To enable access to BSS-2 for machine learning applications, we develop a thin wrapper layer to the PyTorch-API. This extension is called *hxtorch* and was introduced in Spilger et al. (2020) for non-spiking hardware operation emulating analog multiply-accumulate operations and compositions thereof. There, we build on top of the same signal-flow graph experiment description as for the spiking mode of operation (cf. Section 2.3.2). Operations are mapped to the hardware size by using temporal serialization and physical concurrency. The PyTorch extension enhances this by automatic gradient calculation for training. Same as PyTorch, we implement a functional API in `C++` wrapped to `Python` (e.g. `hxtorch.matmul` comparable to `torch.matmul`) and add modules/layers on top in `Python` (e.g., `hxtorch.nn.Linear` comparable to `torch.nn.Linear`). In contrast, our operations are quantized to the hardware-intrinsic digital resolution (5 bit unsigned activations, 6 bit weights plus sign bit and 8 bit signed results). Execution on the hardware is performed individually for each operation using the JIT execution (see Section 2.3.2).

2.3.6. Calibration Framework

On BSS-2, there are a multitude of voltages and currents controlling analog circuit behavior. While some of them can be set to default values, most of them require calibration in order to match experiment-specific target values and to counteract device-specific mismatch. Fundamentally, the calibration can be executed on a host computer or locally on chip, using the embedded processors. We provide the `Python` module *calix* to handle all aspects of the calibration process.

Model parameters are calibrated by iteratively adjusting relevant parts of the hardware configuration. As an example, the membrane time constant is controlled by a bias current: In order to calibrate the membrane time constant of all neurons, the neurons' membrane potentials are recorded while they decay back to their resting potential after an initial perturbation from the resting state. We can perform an exponential fit to the recorded voltage trace to determine the time constant and iteratively tweak the bias current to reach the desired target.

The calibration routine of each parameter is encapsulated using an object-oriented API providing a common interface. Mainly, two methods allow the iterative parameter search: one applies a parameter configuration to the hardware, while the other evaluates an observable to determine circuit behavior. An algorithm calculates parameter updates during the iterative search. In each step, the measurement from the calibration class is compared to the target value and the parameter set is modified accordingly.

A functional API is provided for commonly used sets of calibrations, for example for calibration of a spiking leaky-integrate and fire (LIF) neuron. Technical parameters and multidimensional dependencies are handled automatically as required in this case. This yields a simple interface for experimenters for tweaking high-level parameters, while calibration routines for individual parameters remain accessible for expert users.

The higher-level calibration functions save their results in a typed data structure, which contains the related analog parameters and digital control bits. Further, success flags indicate whether the calibration targets were reached within the available parameter ranges. These result structures can either directly be applied to a hardware setup or serialized to disk. Application of serialized calibration is beneficial compared to repeating the calibration in experiments due to decreased required time and improved digital reproducibility.

Running the calibration on a host computer using `Python` allows for great flexibility in terms of gathering observations from the chip. We can utilize all observables, including a fast ADC, which allows performing fits to measured data—as sketched previously for the calibration of the membrane time constant. While this direct measurement should yield the most accurate results, fitting to a trace for each neuron takes a lot of time. Performing a full LIF neuron calibration takes a few minutes *via* the `Python` module. And importantly, when scaling this approach to many chips, we need to scale the host computing power accordingly.

In order to achieve better scalability, we can control the calibration from the embedded processors, directly on chip, removing the host computer from the loop. However, this approach limits the observables to those easily accessible to the embedded processor, the CADC and spike counters – performing a fit to an MADC trace using the embedded processors would consume lots of runtime and potentially counteract benefits of scaling. As a result, some calibrations have to rely on an indirect measurement of their observable. Again using the neurons' membrane time constant as an example, we can consider the spike rate in a leak-over-threshold setup. However, this introduces a dependency on multiple potentials being calibrated beforehand.

Apart from the need for indirect measurements, on-chip and host-based calibration work similarly: An iterative algorithm selects parameters, we configure them on chip and characterize their effects. Using the embedded processors for configuring parameters and acquiring data from the two on-chip readouts is fully supported and naturally faster than fetching them from a host computer. We use the SIMD CPUs' vector units for parallel access to the synapse array and columnar ADCs. This is enabled by cross-compiler-support (cf. Section 3.3), by which both the scalar unit and vector unit are integrated and accessible from the `C++` language.

We provide routines for on-chip calibration, which allow all LIF neuron parameters to be calibrated in approximately half a minute, with this number staying constant even when considering large systems comprising many individual chips. Similar to the host-based calibration API, *calix* exposes these on-chip routines as conveniently parameterized functions that can be called within any experiment. Their runtime is mostly limited by waiting for configured analog parameters to stabilize before evaluating the effects on the circuits.

2.4. Platform Operation

Over the past decade neuromorphic systems evolved from intricate lab setups toward back ends for the more comfortable execution of spiking neural networks (Indiveri et al., 2011; Furber et al., 2012; Benjamin et al., 2014; Davies et al., 2018; Pehle et al., 2022). One major step along this development path is to provide users with seamless access to the systems.

Small scale prototype hardware is often connected to a single host machine, e.g., *via* USB. This is also a common usage mode for different neuromorphic hardware. To access these devices, users have to have (interactive) access to the particular machine the hardware is connected to. This limits the flexibility of the user and is an operational burden as the combination of neuromorphic hardware and host machine has to be maintained. While this tightly coupled mode of operation is sufficient during commissioning and initial experiments, it is not robust enough for higher work-loads and flexible usage.

An improvement to the situation sketched above is using a scheduler, e.g., SLURM (Yoo et al., 2003), where users can request a resource, e.g., a specific hardware setup, and the jobs get launched on the matching machine with locally attached hardware. This is the typical mode of access also used for other accelerator-type hardware, e.g., GPU clusters. However, this batch driven way is not always ideal as it often requires accounts on the local compute cluster and does not allow for easy interactive usage. In addition, traditional compute load schedulers optimize for throughput and not latency, therefore the scheduling overhead can be significant especially for hardware that is fast and experiments that are short. In the latter case, job execution rates of the order of Hz and faster are required.

Another downside of using a traditional scheduler is that hardware resources are not efficiently utilized when multiple users want to use the same hardware resources at the same time. Therefore, we developed the micro scheduler *quiggeldy* that exposes access to the hardware directly *via* a network connection, but still manages concurrent access from different users. It decouples the hardware utilization from the user's surrounding computations such as experiment preparation, updates in iterative workflows or result evaluation. For this to work runtime control, configuration, input stimulus as well as output data must be serializable which is facilitated *via* cereal (Grant and Voorhies, 2017). The inter-process communication between the user software and the micro scheduler is done with RCF (Delta V Software, 2020). When a user requests multiple hardware runs, it is checked whether certain already performed parts can be omitted, e.g., resets or re-initializations. Experiment interleaving between multiple users is also supported as the initialization state is tracker for each user and is automatically applied when needed.

Having the correct software environment for using neuromorphic hardware is also a major challenge. Nowadays, software vendors often provide a container image that includes the appropriate libraries. However, this approach does not necessarily yield well specified and traceable dependencies, but only a "working" black-box solution. We overcome this downside by using the Spack (Gamblin et al., 2015) package manager with a meta-package that explicitly tracks all software dependencies and their version needed to run experiments on and develop for the neuromorphic hardware. An automatically built container embedding the Spack installation enables encapsulation and eased distribution. This Spack meta-package is also used for the EBRAINS' JupyterLab service and will eventually be deployed to all HPC sites involved in EBRAINS (EBRAINS, 2022). The latter will facilitate multi-site workflows involving neuromorphic hardware and traditional HPC.

3. APPLICATIONS

In this section, we show-case a range of applications of BSS-2. Each application involves use of unique hardware features or modes of operation and motivates parts of the software design.

First, we describe biological multi-compartmental modeling in Section 3.1 concluding in the development of an API for structural neurons. Continuing, functional modeling with spiking neural network (SNN) is demonstrated for a pattern-generation task in Section 3.2, which leads to embedding of spiking BSS-2 usage into the machine learning framework PyTorch and involves host-based training as well as local learning on the SIMD CPUs. Then, embedded operation, where the SIMD CPUs are the experiment orchestrator of BSS-2, is displayed and their implications detailed in Section 3.3. Following, the non-spiking mode of operation implementing ANNs and its PyTorch interface is characterized in Section 3.4. Afterwards, user adoption and platform access to BSS-2 is shown in Section 3.5. Finally, application of the software stack for hardware co-simulation, co-design and verification is portrayed in Section 3.6.

3.1. Biological Modeling Example

BSS-2 aims to emulate biological inspired neuron models. Most neurons are not simple point-like structures but possess intricate dendritic structures. In recent years, the research interest in how dendrites shape the output of neurons has increased (Major et al., 2013; Gidon et al., 2020; Poirazi and Papoutsi, 2020). As a result, BSS-2 incorporates the possibility to emulate multi-compartmental neuron models in addition to the AdEx point-neuron model (Aamir et al., 2018; Kaiser et al., 2022).

In the following, we use a dendritic branch, which splits into two sub-branches, to illustrate how multi-compartmental neuron models are represented in our system (cf. **Figure 9**). At first, we look at a simplified representation of the model (**Figure 9A**). The main branch consists of two compartments, connected *via* a resistance; at the second compartment, the branch splits in two sub-branches, which themselves consist of two compartments each. On hardware this model is replicated by connecting several neuron circuits *via* switches and tunable resistors (cf. **Figure 9B**). Each compartment consists of at least two neuron circuits, directly connected *via* switches, compare colors in **Figures 9A,B**. With the help of a dedicated line at the top of the neuron circuits these compartments can then be connected *via* resistors to form the multi-compartmental neuron model; for more details see Kaiser et al. (2022).

In software, the `AtomicNeuron` class stores the configuration of a single neuron circuit and therefore can be used to configure the switches and resistors as desired. As mentioned in Section 2.3.4, the `HXNeuron` exposes this data structure to the high-level interface PyNN, allowing users to construct multi-compartmental neuron models in a known environment. However, it is cumbersome and error-prone to set individual switches. As a consequence, we implement a dictionary-like hierarchy on top of the `AtomicNeuron`, called `LogicalNeuron` in the logical abstraction layer (cf. Section 2.2).

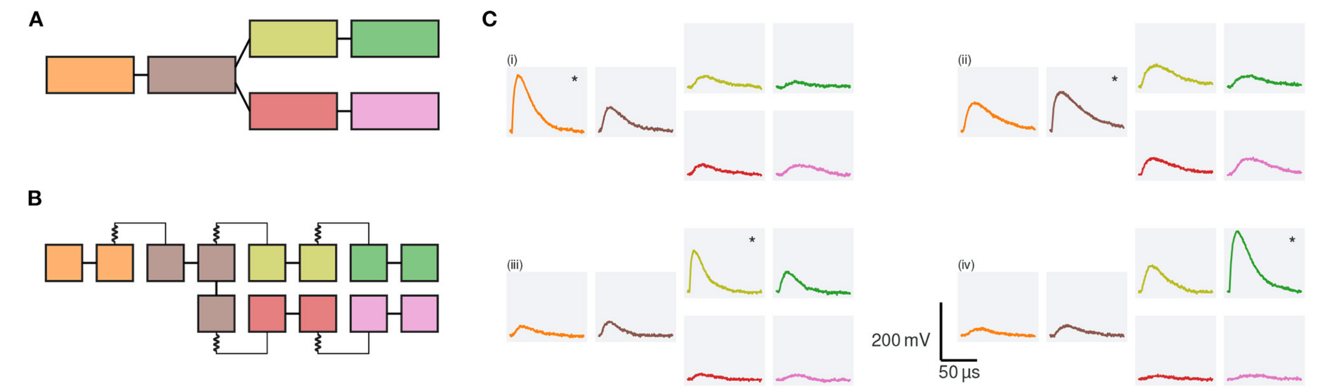


FIGURE 9 | Pulse propagation along a dendrite which branches into two sub-branches. **(A)** Each branch is modeled by two compartments (rectangles). Different compartments are connected *via* resistors (lines). **(B)** Hardware configuration: neuron circuits (squares) are arranged in two rows on BSS-2, compare **Figure 1D**. Each compartment is represented by at least two neuron circuits. Circuits which form a single compartment are directly connected *via* switches (straight lines); compartments are connected *via* resistors. For details see Kaiser et al. (2022). **(C)** Membrane responses to synaptic input: we inject synaptic input at four different compartments; the compartment at which the input is injected is marked by a *. The membrane traces of the different compartments are arranged as in **(A)**. For the top left quadrant (i) the input is injected in the first compartment and decreases in amplitude while it travels along the chain. The response in both branches is symmetric. A similar behavior can be observed when the input is injected in the second compartment, (ii). Due to the symmetry of the model, we only display membrane responses for synaptic input to the upper branch. When injecting the input in the first compartment of the upper branch, (iii), the input causes a noticeable depolarization within the same branch and the main branch but does not cause a strong response in the lower sister branch. All values are given in the hardware domain.

We use a builder pattern approach to construct these logical neurons: the user creates a neuron morphology by defining which neuron circuits constitute a compartment and how these compartments are connected. Upon finalization of the builder, the correctness of the neuron model configuration is checked; if the provided configuration is valid, a `LogicalNeuron` is created. This `LogicalNeuron` stores the morphology of the neuron as well as the configuration of each compartment.

The coordinate system of the BSS-2 software stack (cf. Section 2.2.2), allows to place the final logical neuron at different locations on the chip (Müller et al., 2020a). This is achieved by saving the relation between the different neuron circuits defining the morphology in relative coordinates. Once the neuron is placed at a specific location on the chip, the relative coordinates are translated to absolute coordinates.

Currently, the logical neuron is only exposed in the logical abstraction layer. In future work, it will be integrated in the PyNN API of the BSS-2 system. This will—for instance—allow to easily define populations of multi-compartmental neurons and connections between them.

3.2. Functional Modeling Example

The BSS-2 system enables energy efficient and fast SNN implementations. Moreover, the system's embedded SIMD CPU enables highly parallelized on-chip learning with fast access to observables and thus, promises to benefit the computational neuroscience and machine learning community in terms of speed and energy consumption. We demonstrate functional modeling on the BSS-2 system with a pattern-generation task using recurrent spiking neural networks (RSNNs) with an input layer, a recurrent layer and a single readout neuron. The recurrent layer consists of 70 LIF neurons $\{j\}$ with membrane potential v_j^t, receiving spike trains x_i^t from 30 input neurons $\{i\}$. Neurons in the recurrent layer project spike events z_j^t onto the single leaky-integrate readout neuron with potential y^t.

RSNNs are commonly trained using backpropagation through time (BPTT) by introducing a variety of surrogate gradients taking account of the discontinuity of spiking neurons (Shrestha and Orchard, 2018; Zenke and Ganguli, 2018; Bellec et al., 2020). However, as BPTT requires knowledge of all network states along the time sequence in order to compute weight updates (backwards locking), it is not just considered implausible from a biological perspective, but also unfavorable for on-chip learning, which effectively enables high scalability due to local learning. Therefore, we utilize e-prop learning rules (Bellec et al., 2020), where the gradient for BPTT is factorized into a temporal sum over products of so-called learning signals L_j^t and synapse-local eligibility traces e_{ji}^t. While the latter accumulates all contributions to the gradient that can be computed forward in time, the first depends on the network's error and still requires BPTT. However, Bellec et al. (2020) provide suitable approximations for L_j^t, allowing computing the weight updates online (**Figure 10A**). Such learning rules are favorable for the BSS-2 system, as the SIMD CPU can compute the weight updates locally while the network is emulated in parallel.

E-prop-inspired learning on the BSS-2 system is enabled by adapting Bellec et al. (2020, Equation 28). Here we replace the membrane potentials v_j^t in e_{ji}^t with the post-synaptic recurrent spike train z_j^t,

$$e_{ji}^{t+1} \rightarrow z_j^{t+1} \cdot \mathcal{F}_\alpha\left(z_i^t\right) := \hat{e}_{ji}^{t+1}, \quad \Delta W_{ji}^{\mathrm{hh}} = -\eta \sum_t L_j^t \mathcal{F}_\kappa\left(\hat{e}_{ji}^t\right), \tag{1}$$

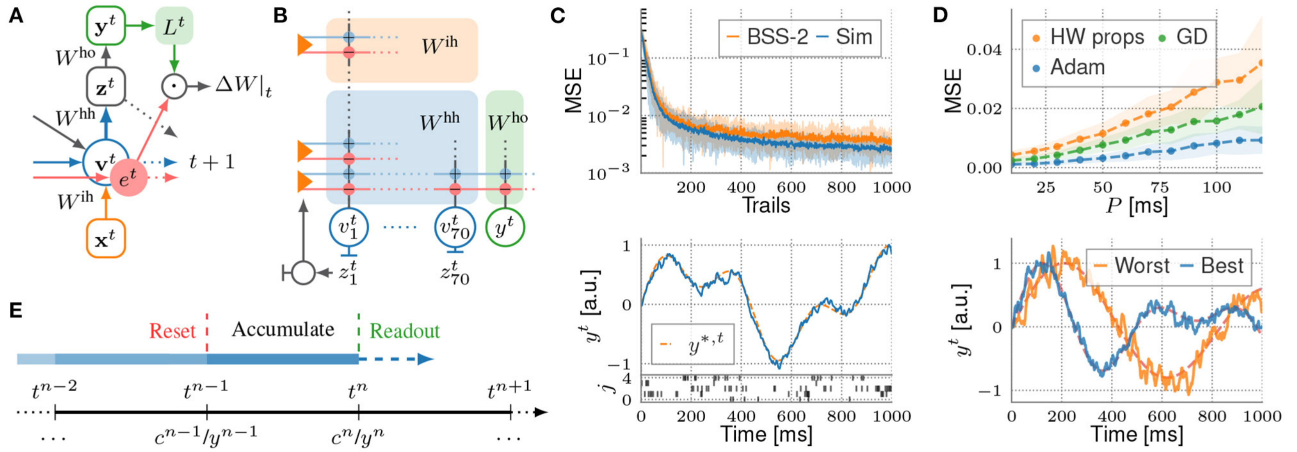


FIGURE 10 | (A) Computational graph of an RSNN for one time step. The contribution to the weight update is computed by merging learning signals L_j^t with eligibility traces e_{ji}^t. **(B)** Representation of the RSNN on the BSS-2 system using signed synapses. Inputs and recurrent spike trains are routed to the corresponding synapse drivers *via* the crossbar. **(C)** s-prop training on hardware. The upper plot depicts the evolution of the MSE while training the BSS-2 system in-the-loop, where the experiment is executed on BSS-2 and weight updates are computed on the host computer, in comparison to training with the network simulated in software, incorporating basic hardware properties (Sim). In both cases the weights are optimized using the Adam optimizer (Kingma and Ba, 2014). The learned analog membrane trace of the readout neuron after training BSS-2 for 1,000 epochs is exemplified in the lower plot, aligned to the spike trains z_j^t of the first five out of 70 recurrent neurons. **(D)** NASProp simulations. The upper plot depicts the MSE over the update period *P* after training with Adam in comparison to a training with GD and a training taking additional hardware properties (noise, weight saturation, etc.) into account (HW props). Optimization with pure GD mimics weight updates computed by the SIMD CPU while on-chip learning. The lower plot shows the worst and best learned readout traces of the target pattern ensemble in simulation. **(E)** Timing of NASProp weight updates. For each update *n* at t^n, the correlation c_{ji}^n are merged with the learning signals L_j^n by incorporating the membrane trace y^n.

where $\mathcal{F}_x$ is an exponential filter with decay constant x. The update rule for input weights, derived in Bellec et al. (2020), is adapted accordingly. The equation for output weights remains untouched. With the readout neuron's membrane trace y^t and an MSE loss measuring the error to a target trace $y^{*,t}$, the learning signals are $L_j^t = W_j^{\mathrm{ho}}\left(y^t - y^{*,t}\right)$. Since this learning rule propagates only spike-based information over time we refer to it as *s-prop*.

Finally, we approach s-prop learning with BSS-2 in the loop (cf. Section 2.1). For this, the network, represented by PyTorch parameters $W^{\mathrm{ih,\,hh,\,ho}}$, is mapped to a hardware representation (see **Figure 10B**) *via hxtorch* (see Section 2.3.5), forwarding a spike tensor on-chip. Inherently, *grenade* (see Section 2.3.2) applies a routing algorithm, finds a graph-based experiment description and executes it on hardware for a given time interval. The routing algorithm allocates two adjacent hardware synapses for one signed synapse weight in software, one excitatory and one inhibitory. Further, *grenade* records the MADC-sampled readout trace y^t and the recurrent spike trains $\mathbf{z}^t$. Both observables are returned as PyTorch tensors for weight optimization on the host side. Experiment results are displayed in **Figure 10C**.

Implementing s-prop on-chip requires the SIMD CPU to know and process explicit spike-times. As this comes with a high computational cost, the correlation sensors are utilized to emulate approximations of the spike-based eligibility traces $\hat{e}_{ji}^t$ in analog circuits, thereby freeing computational resources on the SIMD CPU. The correlation sensors model the eligibility traces under nearest-neighbor approximation (Friedmann et al., 2017) and are accessed by the SIMD CPU as an entity c_{ji}^n, accumulated over a period P. Hence, the time sequence is split into N chunks of size P and weight updates on the SIMD CPU are performed at times $t^n = nP + \tilde{t}$, with $n \in \mathbb{N}_0^{<N}$ (cf. **Figure 10E**) and $\tilde{t} \in [0, P)$ a random offset,

$$\Delta \bar{W}_{ij}^{\mathrm{ih/hh}} = -\eta \sum_n L_j^n \mathcal{F}_{\hat{\kappa}}\left(c_{ji}^{\mathrm{ih/hh},n}\right) \quad \text{and}$$

$$\Delta \bar{W}^{\mathrm{ho}} = -\eta \sum_n \left(y^n - y^{*,n}\right) \mathcal{F}_{\hat{\kappa}}\left(\zeta_j^n\right), \qquad (2)$$

with $\hat{\kappa} = \exp\left(-P/\tau_{\mathrm{m}}\right)$ and ζ_j^n being the recurrent spike count in interval n. Due to the updates rules' accumulative nature, we refer to them as neuromorphic accumulative spike propagation (NASProp). Simulations in **Figure 10D** verify that NASProp endows RSNNs with the ability to solve the pattern-generation task reasonable well.

NASProp's SIMD CPU implementation effectively demonstrates full on-chip learning on the BSS-2 system. In high-level software, on-chip learning is implemented in a PyTorch model, defined in *hxtorch*, holding parameters for the network's projections. Its `forward` method implicitly executes the experiment on the BSS-2 system for a batch of input sequences. Currently, this model learning on-chip serves as a mere black box for the specific network at hand with a static number of layers, as for on-chip spiking networks the network's topology needs to be known upon execution. Therefore, this approach is considered a first step from common PyTorch models to spiking on-chip models.

As for in-the-loop learning, on forwarding a batch of inputs sequences, *grenade* maps the software network to a hardware representation with signed synapses and configures the chip accordingly. Moreover, before executing a batch element, *grenade* starts the plasticity kernel on the SIMD CPU, computing weight updates in parallel to the network's emulation. The plasticity rule implementation relies on *libnux* (cf. Section 2.2.3) and utilizes the VU extension for accessing hardware observables (e.g., c^n_{ji} and y^n) and computing weight updates row-wise in parallel, thereby fully exploiting the system's speed up factor.

In *hxtorch*, learning parameters are configured in a configuration object exposed to `Python`, which is injected to *grenade* and passed to the SIMD CPU before batch execution on hardware begins. As different projections in the network have different update rules, relying on population-specific observables, the network's representation on hardware (cf. **Figure 10B**) is communicated to the SIMD CPU. This allows for identifying signed hardware synapses and neurons with projections and populations on the SIMD CPU. Finally, before each batch element is executed, *grenade* has the ability to write trial-specific details onto the SIMD CPU (e.g. random offset $\tilde{t}$ and the synapse row to perform updates for). Hence, smooth on-chip learning is granted by reliable communication between little-endian host engine and the embedded big-endian SIMD CPU. For serialization of information from and to the SIMD CPU we deploy the `C++` header-only library *bitsery* (Vinkelis, 2020), allowing for seamless transmission of objects between systems of differing endianness.

Due to changing hardware weights during on-chip training, the adjusted weights are reverse mapped to the software representation and stored in the network's parameter tensors. Therewith we utilize PyTorch's native functionality to load and store network parameters. Reverse network mapping is implemented in the *hxtorch* on-chip-learning model by accessing the hardware routing result and is performed implicitly in the model's `forward` method after experiment execution.

Successful implementations of plasticity rules for on-chip learning are facilitated by providing transparency of SIMD CPU programs by means for tracing and recording data. To that end, *libnux* (cf. Section 2.2.3) facilitates logging of any information into a dedicated SIMD CPU memory region, easily accessed from the host engine. Moreover, logging can be redirected to the FPGA-controlled dynamic random-access memory (DRAM), effectively allowing extensive logging of whole learning processes and hardware observables.

3.3. Embedded Operation

Apart from operating BSS-2 tightly coupled to a host computer, the integrated microprocessors can act as system controllers. They can orchestrate the control flow of the experiment and undertake tasks within it. These tasks may include calibration routines, virtual environment simulation or optimizer loops. Embedding them in proximity to the neural network core yields latency and data-locality advantages. In the following, we describe three exemplary experiments that make exhaustive use of the embedded processors as system controllers.

First, Wunderlich et al. (2019) introduce an embedded environment simulation of a simplified version of the Pong video game on the SIMD CPU, see left panel in **Figure 11**. One of the two involved agents plays optimally by design, the other one is represented by a SNN on BSS-2. During the experiment, the latter is trained on-chip using a reward-based spike timing dependent plasticity (STDP) rule. This set-up therefore unites the control flow, virtual environment simulation and learning rule within a single program running on the integrated processors.

Second, Stradmann et al. (2021) describe the application of the BSS-2 system for inference of ANNs that detect atrial fibrillation in medical electrocardiogram (ECG) data. Targeting applications in energy efficient devices, they aim for as little periphery as possible and therefore let the embedded processors orchestrate all classification routines. The resulting tight loop between the analog inference engine and digital data in- and outputs allows for low classification latencies and high throughput of more than 3.600 ECG traces per second.

Third, Schreiber et al. (in preparation) presents the emulation of an insect model with strong biological inspiration on BSS-2. The simplified brain model is embedded into an agent that is fed with stimuli from a simulated environment, see right panel in **Figure 11**. While the neural network is emulated as a SNN within the analog core, the agent itself as well as its virtual environment are both simulated on the SIMD CPU. The authors specifically challenge the virtual insects with a simple path integration task: As depicted in the right panel of **Figure 11**, a simulated swarm-out phase is followed by a period of free flight, where the agent is supposed to return to its nest. The complexity of this task and the comparably low number of involved neurons requires precisely controlled dynamics, which they achieve by integrating experiment specific on-chip calibration routines directly on the SIMD CPUs (cf. Section 2.3.6).

Supporting these complex experiments on the embedded processors and their interaction with the controlling host computer poses specific requirements to the BSS-2 OS. Especially, a cross-compilation toolchain for the SIMD CPU is required.

As described in Section 2.2.3, we therefore provide a cross-compiler based on `gcc` (GNU Project, 2018), which in addition to the processor's scalar unit also integrates its custom vector unit in `C++` (Müller et al., 2020a). Additional hardware specific functionality is encapsulated in the support library *libnux*. It abstracts access to configuration data and observables in the analog neural network core, like synaptic weights or correlation measurements. The exchange of such data with the host is facilitated by integration of the lean, cross-platform binary serialization library *bitsery* (Vinkelis, 2020).

For execution, the compiled programs need to be placed in system memory—in case of BSS-2, each SIMD CPU has direct access to 16 kB SRAM. For a complete calibration routine or complex locally simulated environments, this may not suffice. We therefore utilize the controlling FPGA as memory controller: It allows the on-chip processors to access externally connected DRAM with significantly larger capacity at the cost of higher latency. Programs for the embedded processor can place instructions and data onto both the internal SRAM

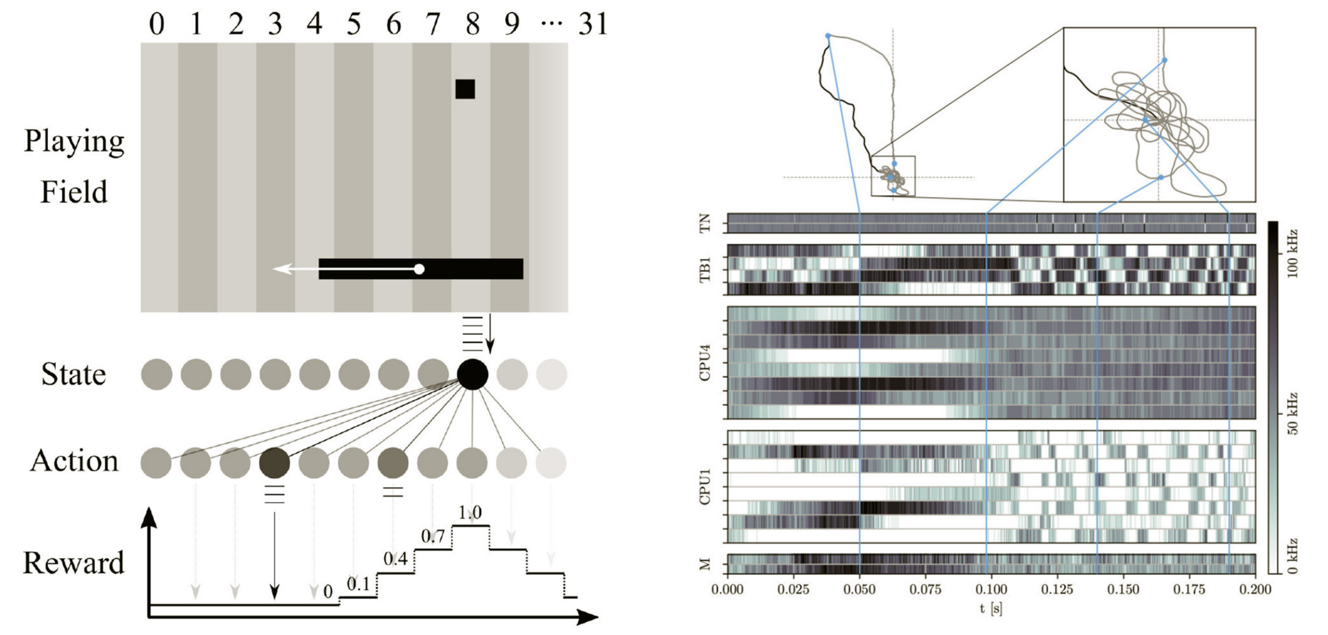


FIGURE 11 | (Left) Reinforcement learning: the chip implements a spiking neural network sensing the current ball position and controlling the game paddle. It is trained *via* a reward-based STDP learning rule to achieve almost optimal performance. The game environment, the motor command and stimulus handling, the reward calculation and the plasticity is performed by a C++ program running on the on-chip processor. Figure taken from Wunderlich et al. (2019). (Right) Recording of a virtual insect navigating a simulated environment. The top panels show the forced swarm-out path in black. During this phase, the SNN emulated by the analog neuron and synapse circuits on BSS-2 perform path integration. Afterwards, the insect flies freely and successfully finds its way back to the starting point and circles around it (gray trajectory). The bottom panel shows the neuronal activity during the experiment. The environment simulation as well as the interaction with the insect is performed by a C++ program running on the on-chip processor. Figure taken from Pehle et al. (2022).

and the external memory *via* compiler attributes. This allows fine-grained decisions about the access-latency requirements of specific instruction and data sections.

Similar to experiments designed for operation from the host system, embedded experiments often require reconfiguration of parts of BSS-2. The hardware abstraction layer introduced in the BSS-2 OS (cf. Section 2.2.2) has therefore been prepared for cross-compilation on the embedded processors. As a result, the described container and coordinate system can be used in experiment programs running on the on-chip SIMD CPUs.

3.4. Artificial Neural Networks

The BSS-2 hardware supports a non-spiking operation mode which supports artificial neural networks (ANNs) implementing multiply-accumulate (MAC) operations (Weis et al., 2020). The operation within the analog core is sketched in **Figure 12A**. Each entry in the vector operand stimulates one or two rows of synapses, when using unsigned or signed weights, respectively. The activations have an input resolution of 5 bit, controlling the duration of synapses' activation. Similar to the spiking operation, synapses emit a current pulse onto the neurons' membranes depending on their weight, which has a resolution of 6 bit. We implement signed weights by combining an excitatory and an inhibitory synapse into one logical synapse. Once all entries in the input vector have been sent to the synapses, the membrane potential resembles the result of the MAC operations. It is digitized for all neurons in parallel using the CADC, yielding an 8 bit result resolution.

As a user interface, we have developed an extension to the PyTorch machine learning framework (Paszke et al., 2019), hxtorch (Spilger et al., 2020). It partitions ANN models into chip-sized MAC operations that are executed on hardware using grenade, see Section 2.2.5. Apart from a special MAC program used for each multiplication, the majority of code is shared between spiking and non-spiking operation. With the leak term disabled, the neurons' membranes represent the integrated synaptic currents, as shown in **Figure 12B**. As the MAC operation lacks any real-time requirements, it is executed as fast as possible to optimize energy efficiency. In terms of circuit parameterization, this means we choose a small synaptic time constant in order for the membrane potential to stabilize quickly. Therefore, a subset of the existing spiking calibration routines can be reused here (cf. Section 2.3.6). There is only one additional circuit—the encoding of input activations to activation times in synapse drivers—that needs to be calibrated.

Defining an ANN model in *hxtorch* works similar to PyTorch: The *hxtorch* module provides linear and convolutional layer classes as a replacement for their PyTorch equivalents. We introduce a few additional parameters controlling the specifics of hardware execution, e.g., the time interval between sending successive entries in the input vector to the synapse matrix, or the option to repeat the vector for efficacy scaling.

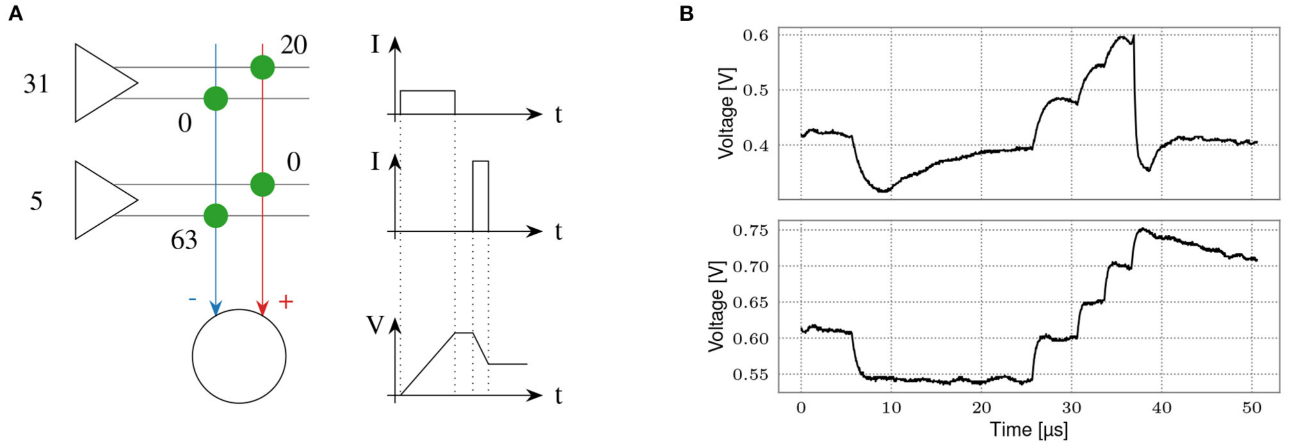


FIGURE 12 | Matrix-vector multiplication for ANN inference. **(A)** Scheme of a multiply-accumulate operation. Vector entries are input *via* synapse drivers (left) in 5 bit resolution. They are multiplied by the weight of an excitatory or inhibitory synapse, yielding 6 bit plus sign weight resolution. The charge is accumulated on neurons (bottom). Figure taken from Weis et al. (2020). **(B)** Comparison between a spiking (top) and an integrator (bottom) neuron. Both neurons receive identical stimuli, one inhibitory and multiple excitatory inputs. While the top neuron shows a synaptic time constant and a membrane time constant, the lower is configured close to a pure integrator. We use this configuration for ANN inference. Please note that for visualization purposes the input timing (bottom) has been slowed to match the SNN configuration (top). The integration phase typically lasts <2 μs.

This enables the user to optimize saturation effects when driving the input currents as well as the gain of the MAC operation for the particular experiment. For both we provide default values as a starting point. The activation function `ConvertingReLU` additionally converts signed 8 bit output activations into unsigned 5 bit input activations for the following layer by a bitwise right shift.

Trained deep neural network models can be transferred to BSS-2 by first quantizing them with PyTorch and subsequently mapping their weights to the hardware domain. For quantization, we need to consider the intrinsic gain factor of the hardware MAC operation.

Figure 13 shows an example application of a deep neural network with BSS-2, using the yin-yang dataset from Kriener et al. (2021). One of the three classes—yin, yang, or dot—are to be determined from four input coordinates $(x, y, 1 - x, 1 - y)$. The network is first trained with 32 bit floating point accuracy using PyTorch, achieving 98.9 % accuracy. After quantizing with PyTorch to the hardware resolution of 5 bit activations and 6 bit plus sign weights, this drops to 94.0 %. Porting the model to BSS-2, after running a few epochs of hardware-in-the-loop training, an accuracy of 95.8 % is finally reached.

In addition to running the ANN on the BSS-2 hardware, a hardware-limitations-aware simulation is available. It can be enabled per layer *via* the `mock` parameter (see **Figure 13B**). For mock mode, we simply assume a linear MAC operation, using a hardware-like gain factor. To investigate possible effects of the analog properties of the BSS-2 hardware on the inference and training, additional Gaussian noise of the accumulators and multiplicative fixed-pattern deviations in the weight matrix can be simulated. The comparison with actual hardware operation shown in **Figure 13D** illustrates how this simple model already captures the most dominant non-linearities of the system. More sophisticated software representations that embrace second-order effects across multiple hardware instances have been proposed by Klein et al. (2021). They have shown how pre-training with faithful software models can significantly decrease hardware allocation time while at the same time increasing classification accuracy compared to plain hardware-in-the-loop training.

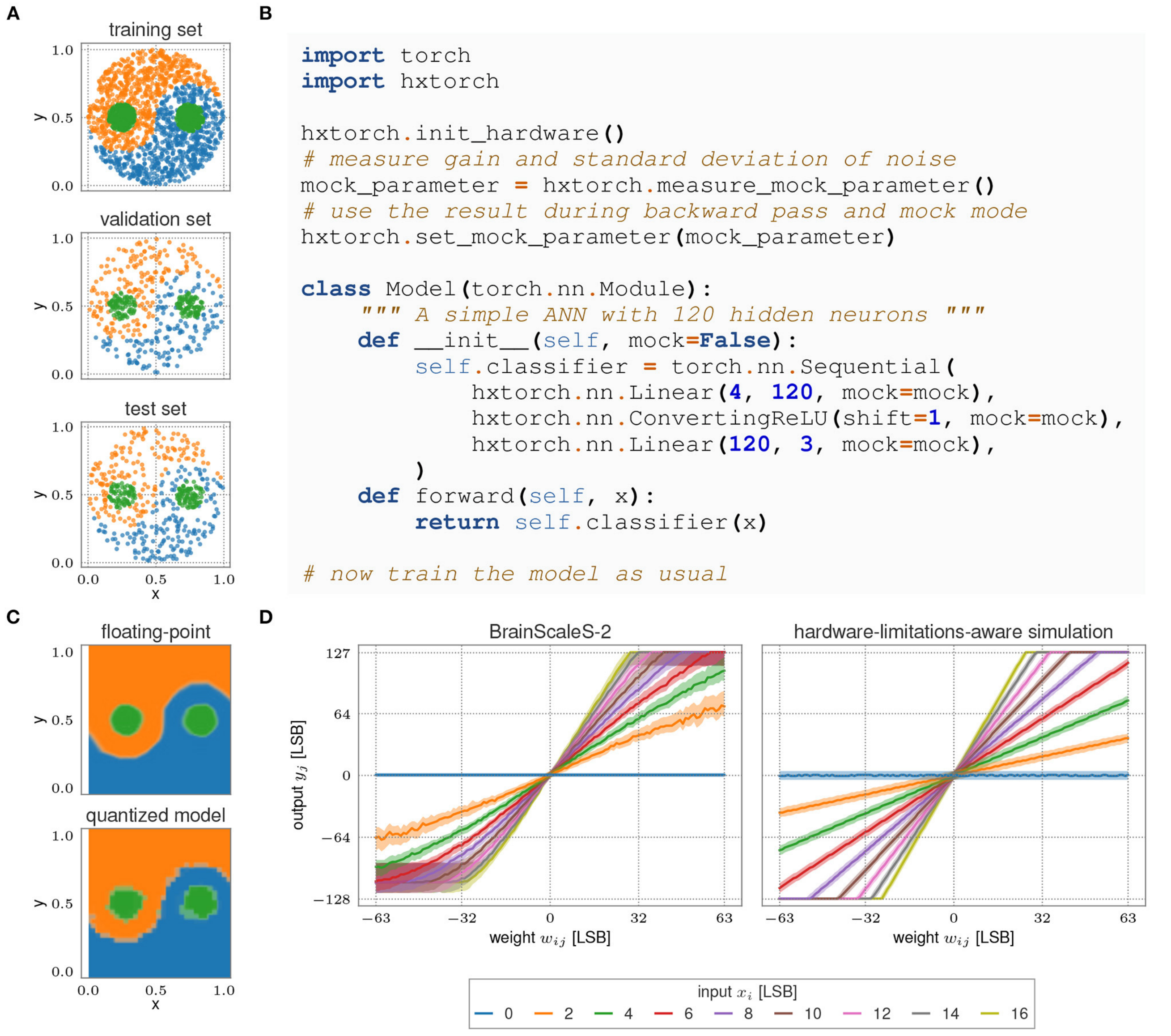
3.5. User Adoption and Platform Access

The BSS-2 software stack aims to enable researchers to exploit the capabilities of the novel neuromorphic substrate. Support for common modeling interfaces like PyNN and PyTorch provides a familiar entry point for a wide range of users. However, not all aspects of the hardware can fully be abstracted away, requiring users to familiarize themselves with unique facets of the system. To flatten the learning curve several tutorials—verified in continuous integration (CI) as "executable" documentation—as well as example experiments are provided[17]. They range from introducing the hardware *via* single neuron dynamics to learning schemes like plasticity rate coding. In addition to the scientific community, they also target students, for example exercises accompanying a lecture about Brain Inspired Computing and hands-on tutorials.

A convenient entry point to explore novel hardware are interactive web-based user interfaces. That is why we integrated the BSS-2 system into the EBRAINS Collaboratory[18] (EBRAINS, 2022). The Collaboratory provides a dynamic VM hosting on multiple HPC sites for Jupyter notebooks running in a comprehensive software environment. An BSS-2-specific

[17] The tutorials and example experiments are available at https://github.com/electronicvisions/brainscales2-demos.

[18] Platform access is available *via* https://ebrains.eu.



```python
import torch
import hxtorch

hxtorch.init_hardware()
# measure gain and standard deviation of noise
mock_parameter = hxtorch.measure_mock_parameter()
# use the result during backward pass and mock mode
hxtorch.set_mock_parameter(mock_parameter)

class Model(torch.nn.Module):
    """ A simple ANN with 120 hidden neurons """
    def __init__(self, mock=False):
        self.classifier = torch.nn.Sequential(
            hxtorch.nn.Linear(4, 120, mock=mock),
            hxtorch.nn.ConvertingReLU(shift=1, mock=mock),
            hxtorch.nn.Linear(120, 3, mock=mock),
        )
    def forward(self, x):
        return self.classifier(x)

# now train the model as usual
```

FIGURE 13 | (A) The Yin-Yang dataset (Kriener et al., 2021) used for the experiment. **(B)** Hardware initialization and model description with *hxtorch*. **(C)** Network response of the trained model depending on the input. (Top) 32 bit floating-point precision; (Bottom) Quantized model on BSS-2 (5 bit activations, 6 bit plus sign weights). **(D)** Output of the MAC operation on BSS-2 (left) compared to the linear approximation (right). The solid line indicates the median, the colored bands contain 95% of each neuron's outputs across 100 identical MAC executions.

experiment service manages multi-user access to the hardware located in Heidelberg utilizing the *quiggeldy* micro scheduler (see Section 2.4). It allows for seamless interactive execution of experiments running on hardware with execution rates of over 10 Hz. This, for example, was utilized during hands-on tutorials at the NICE 2021 conference (NICE, 2021). The execution rates of that demonstration are shown in **Figure 14**.

Furthermore, EBRAINS has begun to provide a comprehensive software distribution that includes typical neuroscientific software libraries next to the BSS-2 client software. As of now, this software distribution has been already deployed at two HPC centers and work is under way to extend this to all sites available in the EBRAINS community. Leaving interactive demos aside, this automatic software deployment will simplify multi-site workflows significantly—including BSS-2 systems—as the scientist is not responsible for software deployment anymore.

3.6. Hardware/Software Co-development

The BSS-2 platform consists of two main hardware components: the ASIC implementing an analog neural network core and digital periphery, as well as an FPGA used for experiment control

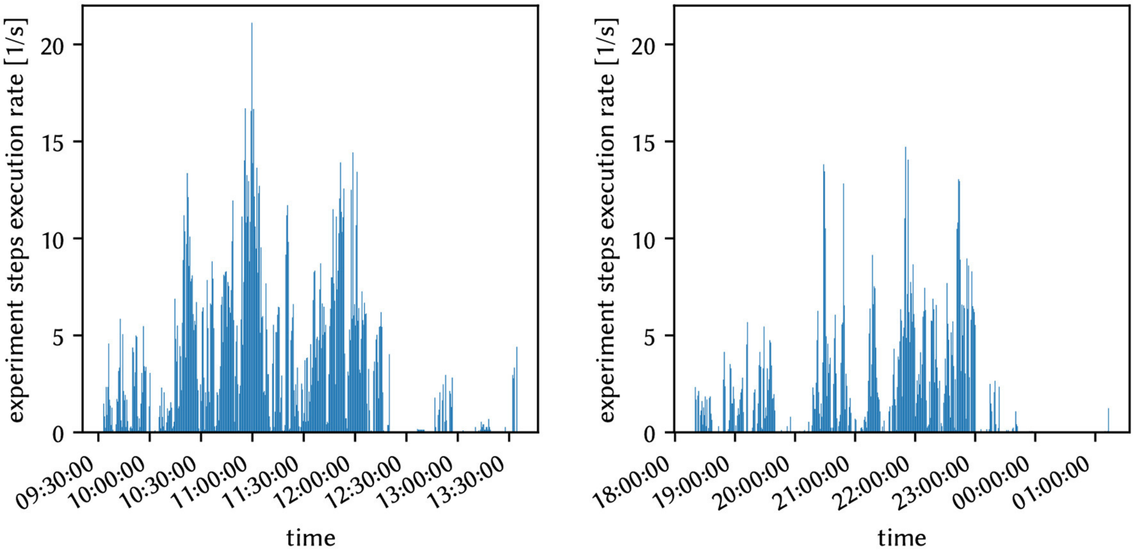


FIGURE 14 | Rate of executed experiment-steps *via quiggeldy* during the two BSS-2 hands-on tutorials at NICE 2021. Experiments were distributed among eight hardware setups. In total there were 86,077 hardware runs executed.

and digital communication. Development of these hardware components is primarily driven by simulations of their analog and digital behavior, where—especially in the case of the ASIC—solid pre-fabrication test strategies need to be employed. Given the complexity of the system, integration tests involving all subsystems are required to ensure correct behavior.

Replicating the actual hardware systems, the setup for these simulated integration tests pose very similar requirements on the configuration and control software. The BSS-2 OS therefore provides a unified interface to both, circuit simulators and hardware systems. For the connection to the simulators, we introduce an adapter library (*flange*) as an optional substitution for the network transport layer. Implementing an additional *hxcomm* back-end, *flange* allows for the transparent execution of hardware experiments in simulation.

This architecture enables various synergies between hardware and software development efforts—specifically, co-design of both components already in early design phases. On system level, this methodology helps to preempt interface mismatch between components of various different subsystems. Positive implications for software developers include the possibility of very early design involvement as well as enhanced debug information throughout the full product life cycle: Having simulation models of the hardware components of the system allows for the inspection of internal signals within the FPGA and ASIC during program runtime. In particular, we have made use of this possibility during the development of a compiler toolchain for the embedded custom SIMD microprocessors, where the recording of internal state helps to understand the system's behavior. Hardware development, on the other hand, strongly profits from software-driven verification strategies and test frameworks. BSS-2 OS especially allows to run the very same test suites on current hardware as well as simulations of future revisions. These shared test suites are re-used across all stages of the platform's life cycle for multiple hardware generations, therefore ever accumulating verification coverage.

4. DISCUSSION

This work describes the software environment for the latest BrainScaleS (BSS) neuromorphic architecture (Pehle et al., 2022): the BrainScaleS-2 (BSS-2) operating system. In Müller et al. (2020b), we introduced the operating system for the BrainScaleS-1 (BSS-1) wafer-scale neuromorphic hardware platform. New basic concepts of the second-generation software architecture were described in Müller et al. (2020a). For example, we introduced a concise representation of "units of configuration" and "experiment runs" supporting asynchronous execution by extensive usage of "future" variables. Key concepts already existing in BSS-1—e.g., the type-safe coordinate system—were extended for BSS-2. In particular, the systematic use of "futures" now allows higher software levels to transparently support experiment pipelining and asynchronous experiment execution in general. Additionally, dividing experiments into a definition and an execution phase also facilitates experiment correctness, software stack flexibility—by decoupling hardware usage from experiment definition—as well as increased platform performance by enabling a separation of hardware access from other aspects of the experiment.

The new software framework is expert-friendly: we designed the software layers to facilitate composition between higher- and lower-level application programming interfaces (APIs). Domain experts can therefore define experiments on a higher abstraction level in certain aspects, and are still able to access low-level functionality. A software package for calibration routines—the process of tuning hardware parameters to the requirements defined by an experiment—provides algorithms and settings for typical parameterizations of the chip, including support for multi-compartmental neurons and non-spiking use cases. An experiment micro scheduler service allows to pipeline experiment runs, and even preempt longer experiment sessions of individual users, to decrease hardware platform latency for other user sessions. Enabling multiple high-level modeling interfaces—such as PyNN and PyTorch—to cover a larger user base was one of the new requirements for BSS-2. To achieve this, we provide a separate high-level representation of user-defined experiments. This signal-graph-based representation is generally suited for high-level configuration validation, optimization, and transformation from higher- to lower-level abstractions. The modeling API wrappers merely provide conversions between data types and call semantics. The embedded microprocessors allow for many new applications: Initially designed to increase flexibility for online learning rules (Friedmann et al., 2017), they have been also used for: environment simulations (Pehle et al., 2022; Schreiber et al., in preparation), online calibration (Section 3.3), general optimization tasks, as well as experiment control (Wunderlich et al., 2019). We ported our low-level chip configuration interface to the embedded processors and thereby allow for code sharing between host and embedded program parts in addition to a software library for embedded use cases. Apart from features directly concerning platform users, we enhanced the support for multiple hardware revisions in parallel facilitating hardware development, commissioning and platform operation. In combination with a dedicated communication layer, this enables not only support for multiple communication backends between host computer and field-programmable gate array (FPGA), such as gigabit ethernet (GbE) or a memory-mapped interface for hybrid FPGA-CPU systems, but also for co-simulation and therefore co-development of software and hardware. Finally, we operate BSS-2 as a research platform. As a result of our contributions to the design and implementation of the EBRAINS (EBRAINS, 2022) software distribution, interactive usage of BSS-2 is now available to a world-wide research community. To summarize, we motivated key design decisions and demonstrated their implementation based on existing use cases: Support for multiple top-level APIs for "biological" and "functional" modeling; support for the embedded microprocessors including structured data exchange with the host, a multi-platform low-level hardware-abstraction layer, and an embedded execution runtime and helper library; support for artificial neural networks in host-based and standalone applications; focus on the user community by providing an integrated platform; sustainable hardware-software co-development.

To build a versatile modeling platform, BSS-2 is a neuromorphic system that improved upon successful properties of predecessors, both, in terms of hardware and software. Simulation speed continues to be an important point in computational neuroscience. The development of new approaches to numerical simulation promising lower execution times and better scalability is an active field of research (Knight and Nowotny, 2018, 2021; Abi Akar et al., 2019), as is improving existing simulation codes (Kunkel et al., 2014; Jordan et al., 2018). Whereas parameter sweeps scale trivially, systematically studying model dynamics over sufficiently long periods as well as iterative approaches to training and plasticity can only benefit from increases in simulation speed. The physical modeling approach of the accelerated neuromorphic architectures allows for a higher emulation speed than state-of-the-art numerical simulations (Zenke and Gerstner, 2014; van Albada et al., 2021). BSS-2 can serve as an accelerator for spiking neural networks and therefore opens up opportunities to work on scientific questions that aren't accessible by numerical simulation. However, to deliver on this promise in reality, both, hardware and software need to be carefully designed, implemented, and applied. The publications building on BSS-2 are evidence of what is possible in terms of modeling on accelerated neuromorphic hardware (Bohnstingl et al., 2019; Cramer et al., 2020, 2022; Wunderlich et al., 2019; Billaudelle et al., 2020, 2021; Müller et al., 2020a; Spilger et al., 2020; Weis et al., 2020; Göltz et al., 2021; Kaiser et al., 2022; Klassert et al., 2021; Klein et al., 2021; Stradmann et al., 2021; Czischek et al., 2022; Schreiber et al., in preparation).

We believe that these publications offer a first glimpse of what will be possible in a scaled-up system. The next step on the roadmap is a multi-chip BSS-2 setup employing EXTOLL (Resch et al., 2014; Neuwirth et al., 2015) for host and inter-chip connectivity. First multi-chip experiments have been performed on a lab setup (Thommes et al., 2022). Additionally, a multi-chip system reusing BSS-1 wafer-scale infrastructure is in the commissioning phase and will provide up to 46 BSS-2 chips. Similar to BSS-1, a true wafer-scale version of BSS-2 will provide an increase in terms of resources by one order of magnitude and thus will enable research that not only looks at dynamics at different temporal scales, but also on larger spatial scales. In terms of software we have been adapting our roadmap continuously to match modelers' expectations. For example, we work on future software abstractions that will allow for flexible descriptions of spiking network models with arbitrary topology in a machine learning framework. PyTorch libraries such as BindsNET (Hazan et al., 2018) or Norse (Pehle and Pedersen, 2021) enable efficient machine-learning-inspired modeling with spiking neural networks and would benefit from neuromorphic hardware support.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found at: Ying-Yang Dataset (https://arxiv.org/abs/2102.08211).

AUTHOR CONTRIBUTIONS

EM initially wrote the extended abstract for the article, devised a first draft of the article, contributed conceptually and in writing to the final manuscript, and lead developer and architect of the BrainScaleS-2 software stack. EA is the main author of the "Functional Modeling" use case and conducted the experiments, contributed to the high-level hxtorch interface for spiking neural networks, and contributed conceptually and in writing to the final manuscript. OB contributed to the hardware abstraction layers and contributed conceptually and in writing to the final manuscript. MC contributed to the high-level PyNN interface and the manuscript. AE co-authored the "Artificial Neural Networks" use case and conducted experiments for this section, contributed to the commissioning of the non-spiking mode of operation, and contributed conceptually and in writing to the final manuscript. JK is the main author of the "Biological Modeling" use case and conducted the experiments, contributed to the high-level PyNN interface, and hardware abstraction layers, and contributed conceptually and in writing to the final manuscript. CM is a main developer of the hardware abstraction layers and the communication layers, contributed to all other software layers and contributed conceptually and in writing to the final manuscript, and maintainer of platform operation. SS contributed to the high-level user experience by testing new features and providing detailed feedback, contributed to the incorporation of calibration data, and the final manuscript. PS is the main developer of the experiment description layers and contributed to all other software layers, contributed conceptually and in writing to the final manuscript. RS contributed to the hardware abstraction layer, calibration procedures, and the final manuscript. YS is the lead designer of the CI/CD workflows used for BrainScaleS-2, contributed to the architecture of BSS-2 OS and contributed conceptually and in writing to the final manuscript. JW is the lead designer of the calibration framework, contributed to the hardware abstraction layer, and contributed conceptually and in writing to the final manuscript. AB contributed to the software methodology, software resources, and the final manuscript. SB contributed to the hardware abstraction layer, commissioning of the hardware platform, and the calibration procedures. BC contributed to the commissioning of the hardware platform and the calibration procedures. FE contributed to the hardware abstraction layer and the commissioning of the hardware platform. JG contributed to the commissioning of the hardware platform, the calibration procedures, and the final manuscript. JI and VK contributed to the transport layer and commissioning of the hardware platform. MK contributed to the methodology and software for co-simulation. AL contributed to the hardware abstraction layer, calibration procedures, and the manuscript. CP contributed to the hardware development, specifically the PPU memory interface, and to a prototype of the "Functional Modeling" use case, contributed to the manuscript. JS is the lead designer and architect of the BrainScaleS-2 neuromorphic system and provided conceptual advice, contributed to the final manuscript. All authors contributed to the article and approved the submitted version.

FUNDING

This work has received funding from the EC Horizon 2020 Framework Programme under grant agreements 785907 (HBP SGA2) and 945539 (HBP SGA3), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2181/1-390900948 (the Heidelberg STRUCTURES Excellence Cluster), the German Federal Ministry of Education and Research under grant number 16ES1127 as part of the *Pilotinnovationswettbewerb Energieeffizientes KI-System*, the Helmholtz Association Initiative and Networking Fund [Advanced Computing Architectures (ACA)] under Project SO-092, as well as from the Manfred Stärk Foundation, and the Lautenschläger-Forschungspreis 2018 for Karlheinz Meier.

ACKNOWLEDGMENTS

The authors wish to thank all present and former members of the Electronic Vision(s) research group contributing to the BrainScaleS-2 neuromorphic platform.

REFERENCES

Aamir, S. A., Müller, P., Kiene, G., Kriener, L., Stradmann, Y., Grübl, A., et al. (2018). A mixed-signal structured AdEx neuron for accelerated neuromorphic cores. *IEEE Trans. Biomed. Circuits Syst*. 12, 1027–1037. doi: 10.1109/TBCAS.2018.2848203

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. (2016). "TensorFlow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA: USENIX Association), 265–283.

Abi Akar, N., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor-a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Los Alamitos, CA: IEEE), 274–282. doi: 10.1109/EMPDP.2019.8671560

Bellec, G., Scherr, F., Subramoney, A., Hajek, E., Salaj, D., Legenstein, R., et al. (2020). A solution to the learning dilemma for recurrent networks of spiking neurons. *Nat. Commun*. 11:3625. doi: 10.1038/s41467-020-17236-y

Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J.-M., et al. (2014). Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* 102, 699–716. doi: 10.1109/JPROC.2014.2313565

Billaudelle, S., Cramer, B., Petrovici, M. A., Schreiber, K., Kappel, D., Schemmel, J., et al. (2021). Structural plasticity on an accelerated analog neuromorphic hardware system. *Neural Netw*. 133, 11–20. doi: 10.1016/j.neunet.2020.09.024

Billaudelle, S., Stradmann, Y., Schreiber, K., Cramer, B., Baumbach, A., Dold, D., et al. (2020). "Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)* (New York, NY). doi: 10.1109/ISCAS45731.2020.9180741

Bohnstingl, T., Scherr, F., Pehle, C., Meier, K., and Maass, W. (2019). Neuromorphic hardware learns to learn. *Front. Neurosci*. 2019:483. doi: 10.3389/fnins.2019.00483

Brette, R. and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol*. 94, 3637–3642. doi: 10.1152/jn.00686.2005

Brüderle, D., Müller, E., Davison, A., Muller, E., Schemmel, J., and Meier, K. (2009). Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system. *Front. Neuroinformatics* 3:17. doi: 10.3389/neuro.11.017.2009

Cramer, B., Billaudelle, S., Kanya, S., Leibfried, A., Grübl, A., Karasenko, V., et al. (2022). Surrogate gradients for analog neuromorphic computing. *Proc. Natl. Acad. Sci. U.S.A*. 119:e2109194119. doi: 10.1073/pnas.2109194119

Cramer, B., Stöckel, D., Kreft, M., Wibral, M., Schemmel, J., Meier, K., et al. (2020). Control of criticality and computation in spiking neuromorphic networks with plasticity. *Nat. Commun.* 11:2853. doi: 10.1038/s41467-020-16548-3

Czischek, S., Baumbach, A., Billaudelle, S., Cramer, B., Kades, L., Pawlowski, J. M., et al. (2022). Spiking neuromorphic chip learns entangled quantum states. *SciPost Phys.* 12:39. doi: 10.21468/SciPostPhys.12.1.039

Dally, W. J., Turakhia, Y., and Han, S. (2020). Domain-specific hardware accelerators. *Commun. ACM* 63, 48–57. doi: 10.1145/3361682

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.* 2:11. doi: 10.3389/neuro.11.011.2008

Delta V Software (2020). *Remote Call Framework*. Delta V Software.

EBRAINS (2022). EBRAINS research infrastructure.

Facebook Inc. (2021a). *PyTorch JIT Overview*. Facebook, Inc.

Facebook Inc. (2021b). *PyTorch on XLA Devices*. Facebook, Inc.

Friedmann, S., Schemmel, J., Grübl, A., Hartel, A., Hock, M., and Meier, K. (2017). Demonstrating hybrid learning in a flexible neuromorphic hardware system. *IEEE Trans. Biomed. Circuits Syst.* 11, 128–142. doi: 10.1109/TBCAS.2016.2579164

Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., et al. (2012). Overview of the SpiNNaker system architecture. *IEEE Trans. Comput.* 99, 2454–2467. doi: 10.1109/TC.2012.142

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., et al. (2015). "The spack package manager: bringing order to HPC software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15* (New York, NY: ACM), 40:1–40:12. doi: 10.1145/2807591.2807623

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430

Gidon, A., Zolnik, T. A., Fidzinski, P., Bolduan, F., Papoutsi, A., Poirazi, P., et al. (2020). Dendritic action potentials and computation in human layer 2/3 cortical neurons. *Science* 367, 83–87. doi: 10.1126/science.aax6239

GNU Project (2018). *The GNU Compiler Collection 8.1*. Free Software Foundation Inc.

Goddard, N. H., Hucka, M., Howell, F., Cornelis, H., Shankar, K., and Beeman, D. (2001). Towards neuroml: model description methods for collaborative modelling in neuroscience. *Philos. Trans. R. Soc. Lond. B Biol. Sci.* 356, 1209–1228. doi: 10.1098/rstb.2001.0910

Göltz, J., Kriener, L., Baumbach, A., Billaudelle, S., Breitwieser, O., Cramer, B., et al. (2021). Fast and energy-efficient neuromorphic deep learning with first-spike times. *Nat. Mach. Intell.* 3, 823–835. doi: 10.1038/s42256-021-00388-x

Grant, W. S., and Voorhies, R. (2017). Cereal - A C++11 library for serialization. Available online at: http://uscilab.github.io/cereal

Hazan, H., Saunders, D. J., Khan, H., Patel, D., Sanghavi, D. T., Siegelmann, H. T., et al. (2018). BindsNET: a machine learning-oriented spiking neural networks library in python. *Front. Neuroinformatics* 12:89. doi: 10.3389/fninf.2018.00089

Hines, M., and Carnevale, N. (2003). The NEURON simulation environment. *Neural Comput.* 9, 1179–1209. doi: 10.1162/neco.1997.9.6.1179

Indiveri, G., Linares-Barranco, B., Hamilton, T. J., van Schaik, A., Etienne-Cummings, R., Delbruck, T., et al. (2011). Neuromorphic silicon neuron circuits. *Front. Neurosci.* 5:73. doi: 10.3389/fnins.2011.00073

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Front. Neuroinformatics* 12:2. doi: 10.3389/fninf.2018.00002

Kaiser, J., Billaudelle, S., Müller, E., Tetzlaff, C., Schemmel, J., and Schmitt, S. (2022). Emulating dendritic computing paradigms on analog neuromorphic hardware. *Neuroscience*. 489, 290–300. doi: 10.1016/j.neuroscience.2021.08.013

Kingma, D. P., and Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint*. doi: 10.48550/arXiv.1412.6980

Klassert, R., Baumbach, A., Petrovici, M. A., and Gärttner, M. (2021). Variational learning of quantum ground states on spiking neuromorphic hardware. doi: 10.2139/ssrn.4012184. [Epub ahead of print].

Klein, B., Kuhn, L., Weis, J., Emmel, A., Stradmann, Y., Schemmel, J., and Fröning, H. (2021). "Towards addressing noise and static variations of analog computations using efficient retraining," in *Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, eds M. Kamp, I. Koprinska, A. Bibal, T. Bouadi, B. Frénay, L. Galárraga, J. Oramas, L. Adilova, Y. Krishnamurthy, B. Kang, C. Largeron, J. Lijffijt, T. Viard, P. Welke, M. Ruocco, E. Aune, C. Gallicchio, G. Schiele, F. Pernkopf, M. Blott, H. Fröning, G. Schindler, R. Guidotti, A. Monreale, S. Rinzivillo, P. Biecek, E. Ntoutsi, M. Pechenizkiy, B. Rosenhahn, C. Buckley, D. Cialfi, P. Lanillos, M. Ramstead, T. Verbelen, P. M. Ferreira, G. Andresini, D. Malerba, I. Medeiros, P. Fournier-Viger, M. S. Nawaz, S. Ventura, M. Sun, M. Zhou, V. Bitetta, I. Bordino, A. Ferretti, F. Gullo, G. Ponti, L. Severini, R. Ribeiro, J. Gama, R. Gavaldà, L. Cooper, N. Ghazaleh, J. Richiardi, D. Roqueiro, D. Saldana Miranda, K. Sechidis, and G. Graça (Cham: Springer International Publishing), 409–420. doi: 10.1007/978-3-030-93736-2_32

Knight, J. C., and Nowotny, T. (2018). GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Front. Neurosci.* 12:941. doi: 10.3389/fnins.2018.00941

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci.* 1, 136–142. doi: 10.1038/s43588-020-00022-7

Kriener, L., Göltz, J., and Petrovici, M. A. (2021). The yin-yang dataset. *arXiv preprint*. doi: 10.48550/arXiv.2102.08211

Kungl, A. F., Schmitt, S., Klähn, J., Müller, P., Baumbach, A., Dold, D., et al. (2019). Accelerated physical emulation of bayesian inference in spiking neural networks. *Front. Neurosci.* 13:1201. doi: 10.3389/fnins.2019.01201

Kunkel, S., Schmidt, M., Eppler, J. M., Plesser, H. E., Masumoto, G., Igarashi, J., et al. (2014). Spiking network simulation code for petascale computers. *Front. Neuroinformatics* 8:78. doi: 10.3389/fninf.2014.00078

Major, G., Larkum, M. E., and Schiller, J. (2013). Active properties of neocortical pyramidal neuron dendrites. *Annu. Rev. Neurosci.* 36, 1–24. doi: 10.1146/annurev-neuro-062111-150343

Mason, S. J. (1953). Feedback theory-some properties of signal flow graphs. *Proc. IRE* 41, 1144–1156. doi: 10.1109/JRPROC.1953.274449

Müller, E., Mauch, C., Spilger, P., Breitwieser, O. J., Klähn, J., Stöckel, D., et al. (2020a). Extending BrainScaleS OS for BrainScaleS-2. *arXiv preprint*. doi: 10.48550/arXiv.2003.13750

Müller, E., Schmitt, S., Mauch, C., Schmidt, H., Montes, J., Ilmberger, J., et al. (2020b). The operating system of the neuromorphic BrainScaleS-1 system. *arXiv preprint*. doi: 10.48550/arXiv.2003.13749

Neuwirth, S., Frey, D., Nuessle, M., and Bruening, U. (2015). "Scalable communication architecture for network-attached accelerators," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)* (New York, NY), 627–638. doi: 10.1109/HPCA.2015.7056068

NICE (2021). *NICE '20: Proceedings of the Neuro-Inspired Computational Elements Workshop*. New York, NY: Association for Computing Machinery.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). "Pytorch: an imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, eds H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Red Hook, NY: Curran Associates, Inc.), 8024–8035.

Pehle, C., Billaudelle, S., Cramer, B., Kaiser, J., Schreiber, K., Stradmann, Y., et al. (2022). The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity. *Front. Neurosci.* 16:795876. doi: 10.3389/fnins.2022.795876

Pehle, C., and Pedersen, J. E. (2021). *Norse – A Deep Learning Library for Spiking Neural Networks*. Available online at: https://norse.ai/docs/ (accessed April 11, 2022).

Poirazi, P., and Papoutsi, A. (2020). Illuminating dendritic function with computational models. *Nat. Rev. Neurosci.* 21, 303–321. doi: 10.1038/s41583-020-0301-7

PowerISA (2010). *PowerISA Version 2.06 Revision b*. Specification, Power.org.

Resch, M. M., Bez, W., Focht, E., Kobayashi, H., and Patel, N., (eds.). (2014). "Sustained simulation performance," in *Proceedings of the Joint Workshop on Sustained Simulation Performance* (Cham: University of Stuttgart (HLRS); Tohoku University; Springer). doi: 10.1007/978-3-319-10626-7

Rhodes, O., Bogdan, P. A., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., et al. (2018). spynnaker: a software package for running PYNN simulations on spinnaker. *Front. Neurosci.* 12:816. doi: 10.3389/fnins.2018.00816

Schemmel, J., Billaudelle, S., Dauer, P., and Weis, J. (2020). Accelerated analog neuromorphic computing. *arXiv preprint*. doi: 10.1007/978-3-030-91741-8_6

Schmitt, S., Klähn, J., Bellec, G., Grübl, A., Güttler, M., Hartel, A., et al. (2017). "Neuromorphic hardware in the loop: training a deep spiking network on the brainscales wafer-scale system," in *Proceedings of the 2017 IEEE International Joint Conference on Neural Networks* (New York, NY). doi: 10.1109/IJCNN.2017.7966125

Shrestha, S. B., and Orchard, G. (2018). "SLAYER: Spike layer error reassignment in time," in *Advances in Neural Information Processing Systems, Vol. 31*, eds S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Red Hook, NY: Curran Associates, Inc.), 1419–1428.

Spilger, P., Müller, E., Emmel, A., Leibfried, A., Mauch, C., Pehle, C., et al. (2020). "hxtorch: PyTorch for BrainScaleS-2 - perceptrons on analog neuromorphic hardware," in *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*, eds J. Gama, S. Pashami, A. Bifet, M. Sayed-Mouchawe, H. Fröning, F. Pernkopf, G. Schiele, and M. Blott (Cham: Springer International Publishing), 189–200. doi: 10.1007/978-3-030-66770-2_14

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8:e28. doi: 10.7554/eLife.47314.028

Stradmann, Y., Billaudelle, S., Breitwieser, O., Ebert, F. L., Emmel, A., Husmann, D., et al. (2021). Demonstrating analog inference on the BrainScaleS-2 mobile system. *arXiv preprint*. doi: 10.48550/arXiv.2103.15960

Suhan, A., Libenzi, D., Zhang, A., Schuh,P., Saeta, B., Sohn, J. Y., et al. (2021). LazyTensor: combining eager execution with domain-specific compilers. *arXiv [Preprint]*. doi: 10.48550/arXiv.2102.13267

Thommes, T., Bordukat, S., Grübl, A., Karasenko, V., Müller, E., and Schemmel, J. (2022). Demonstrating BrainScaleS-2 Inter-Chip Pulse Communication using EXTOLL. *arXiv [Preprint]*. doi: 10.48550/arXiv.2202.12122

van Albada, S. J., Pronold, J., van Meegen, A., and Diesmann, M. (2021). "Usage and scaling of an open-source spiking multi-area model of monkey cortex," in *Brain-Inspired Computing*, eds K. Amunts, L. Grandinetti, T. Lippert, and N. Petkov (Cham: Springer International Publishing), 47–59. doi: 10.1007/978-3-030-82427-3_4

Vinkelis, M. (2020). *Bitsery*. Available online at: https://github.com/fraillt/bitsery (accessed April 11, 2022).

Weis, J., Spilger, P., Billaudelle, S., Stradmann, Y., Emmel, A., Müller, E., et al. (2020). "Inference with artificial neural networks on analog neuromorphic hardware," in *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*, eds J. Gama, S. Pashami, A. Bifet, M. Sayed-Mouchawe, H. Fröning, F. Pernkopf, G. Schiele, and M. Blott (Cham: Springer International Publishing), 201–212. doi: 10.1007/978-3-030-66770-2_15

Wunderlich, T., Kungl, A. F., Müller, E., Hartel, A., Stradmann, Y., Aamir, S. A., et al. (2019). Demonstrating advantages of neuromorphic computation: a pilot study. *Front. Neurosci*. 13:260. doi: 10.3389/fnins.2019.00260

Yavuz, E., Turner, J., and Nowotny, T. (2016). GeNN: a code generation framework for accelerated brain simulations. *Sci. Rep*. 6, 1–14. doi: 10.1038/srep18854

Yoo, A. B., Jette, M. A., and Grondona, M. (2003). "SLURM: simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing* (Berlin; Heidelberg: Springer), 44–60. doi: 10.1007/10968987_3

Zenke, F., and Ganguli, S. (2018). SuperSpike: supervised learning in multilayer spiking neural networks. *Neural Comput*. 30, 1514–1541. doi: 10.1162/neco_a_01086

Zenke, F., and Gerstner, W. (2014). Limits to high-speed simulations of spiking neural networks using general-purpose computers. *Front. Neuroinformatics* 8:76. doi: 10.3389/fninf.2014.00076

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Neuroinformatics

ORIGINAL RESEARCH
published: 19 May 2022
doi: 10.3389/fninf.2022.884180

Deploying and Optimizing Embodied Simulations of Large-Scale Spiking Neural Networks on HPC Infrastructure

Benedikt Feldotto[1], Jochen Martin Eppler[2], Cristian Jimenez-Romero[2], Christopher Bignamini[3], Carlos Enrique Gutierrez[4], Ugo Albanese[5], Eloy Retamino[6], Viktor Vorobev[1], Vahid Zolfaghari[1], Alex Upton[3], Zhe Sun[7,8], Hiroshi Yamaura[9], Morteza Heidarinejad[8], Wouter Klijn[2], Abigail Morrison[2,10,11], Felipe Cruz[3], Colin McMurtrie[3], Alois C. Knoll[1], Jun Igarashi[8,12], Tadashi Yamazaki[9], Kenji Doya[4] and Fabrice O. Morin[1]*

[1] Robotics, Artificial Intelligence and Real-Time Systems, Faculty of Informatics, Technical University of Munich, Munich, Germany, [2] Simulation and Data Lab Neuroscience, Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation, JARA, Forschungszentrum Jülich GmbH, Jülich, Germany, [3] Swiss National Supercomputing Centre (CSCS), ETH Zurich, Lugano, Switzerland, [4] Neural Computation Unit, Okinawa Institute of Science and Technology Graduate University, Okinawa, Japan, [5] Department of Excellence in Robotics and AI, The BioRobotics Institute, Scuola Superiore Sant'Anna, Pontedera, Italy, [6] Department of Computer Architecture and Technology, Research Centre for Information and Communication Technologies, University of Granada, Granada, Spain, [7] Image Processing Research Team, Center for Advanced Photonics, RIKEN, Wako, Japan, [8] Computational Engineering Applications Unit, Head Office for Information Systems and Cybersecurity, RIKEN, Wako, Japan, [9] Graduate School of Informatics and Engineering, The University of Electro-Communications, Tokyo, Japan, [10] Jülich Research Centre, Institute of Neuroscience and Medicine (INM-6), Institute for Advanced Simulation (IAS-6), JARA BRAIN Institute I, Jülich, Germany, [11] Computer Science 3-Software Engineering, RWTH Aachen University, Aachen, Germany, [12] Center for Computational Science, RIKEN, Kobe, Japan

OPEN ACCESS

Edited by:
Omar Awile,
Swiss Federal Institute of Technology Lausanne, Switzerland

Reviewed by:
Mikael Djurfeldt,
Royal Institute of Technology, Sweden
Andrey Palyanov,
Siberian Branch of the Russian Academy of Sciences (RAS), Russia

****Correspondence:***
Benedikt Feldotto
feldotto@in.tum.de

Received: *25 January 2022*
Accepted: *19 April 2022*
Published: *19 May 2022*

Citation:
Feldotto B, Eppler JM, Jimenez-Romero C, Bignamini C, Gutierrez CE, Albanese U, Retamino E, Vorobev V, Zolfaghari V, Upton A, Sun Z, Yamaura H, Heidarinejad M, Klijn W, Morrison A, Cruz F, McMurtrie C, Knoll AC, Igarashi J, Yamazaki T, Doya K and Morin FO (2022) Deploying and Optimizing Embodied Simulations of Large-Scale Spiking Neural Networks on HPC Infrastructure. Front. Neuroinform. 16:884180. doi: 10.3389/fninf.2022.884180

Simulating the brain-body-environment trinity in closed loop is an attractive proposal to investigate how perception, motor activity and interactions with the environment shape brain activity, and vice versa. The relevance of this embodied approach, however, hinges entirely on the modeled complexity of the various simulated phenomena. In this article, we introduce a software framework that is capable of simulating large-scale, biologically realistic networks of spiking neurons embodied in a biomechanically accurate musculoskeletal system that interacts with a physically realistic virtual environment. We deploy this framework on the high performance computing resources of the EBRAINS research infrastructure and we investigate the scaling performance by distributing computation across an increasing number of interconnected compute nodes. Our architecture is based on requested compute nodes as well as persistent virtual machines; this provides a high-performance simulation environment that is accessible to multi-domain users without expert knowledge, with a view to enable users to instantiate and control simulations at custom scale via a web-based graphical user interface. Our simulation environment, entirely open source, is based on the Neurorobotics Platform developed in the context of the Human Brain Project, and the NEST simulator. We characterize the capabilities of our parallelized architecture for large-scale embodied brain simulations through two benchmark experiments, by investigating the effects of scaling compute resources on performance defined in terms of experiment runtime, brain instantiation and simulation time. The first benchmark is based on a large-scale balanced network, while the second one is a multi-region embodied brain simulation consisting of more than a million neurons and a billion synapses. Both benchmarks clearly show how scaling compute resources improves the aforementioned performance metrics in a near-linear fashion. The second benchmark in particular is indicative of both the potential and limitations of a highly distributed simulation in terms of a trade-off between computation speed and resource cost. Our simulation architecture is being prepared to be accessible for everyone as an EBRAINS service, thereby offering a community-wide tool with a unique workflow that should provide momentum to the investigation of closed-loop embodiment within the computational neuroscience community.

Keywords: spiking neural networks, embodiment, Neurorobotics Platform, high performance computing (HPC), NEST, musculoskeletal modeling, large-scale brain simulation, parallel computing

1. INTRODUCTION

While theories exist that describe how brain architecture and neuronal activity support human-specific, higher-level cognitive abilities such as common sense, capacity for generalization and self-awareness, their experimental validation *in vivo* is usually impossible for both technical (e.g., lack of reproducibility, observability and perturbability) and ethical reasons. As such, simulating the human brain becomes necessary in order to test data-driven hypotheses coming from theoretical neuroscience regarding the structure-function-activity trifecta, and thus establish the link between these in an ethical, reproducible and fully observable manner.

In particular, it is only through simulation that the functional capacity of a given brain model can be consistently evaluated at multiple scales and under various operating conditions, or that the individual contribution of its sub-components to the emergence of advanced cognitive functions can be teased apart. In short, as Nobel physicist Richard Feynman concluded, "what I cannot create I do not understand." Not just any isolated simulation will do, though. To have any relevance to data collected from living beings, the simulated brain must be afforded with the possibility to interact with a dynamic, physically realistic and sensory-rich environment. This is what we refer to as embodiment. Only then can the simulated neuronal activity be expected to somewhat match, even to a limited extent, that of an actual brain in natural settings. This aspect is therefore essential when studying cognitive mechanisms that involve sensorimotor integration or motor control.

Such an embodied simulation framework must be able to simulate the brain at scale in order to capture the contributions of multiple brain regions involved in goal-directed actions, and to account for the effects of various learning mechanisms, from single synapses up to network effects of different neural populations. It requires significant computing capabilities and a distributed architecture to cope with the highly parallel, resource-intensive nature of large-scale neuronal network simulations, as well as features that allow interactive experimentation while keeping brain and body simulation in sync.

We demonstrate a prototype for a simulation service on the EBRAINS research infrastructure; this prototype enables users to run custom embodied large-scale brain simulations through the Neurorobotics Platform (NRP), the component of EBRAINS dedicated to closed-loop neuroscience. Implemented with the NEST simulator for large-scale spiking neural networks, these brain simulations are run in a distributed manner on a variable allocation of high-performance computing (HPC) nodes of the supercomputer Piz Daint. Within this framework, large-scale biologically plausible neuronal networks with multiple regions are simulated in NEST and interconnected with a physics simulation of a musculoskeletal system in Gazebo. A dedicated graphical user interface in the NRP frontend enables anyone entitled to adequate compute resources on EBRAINS to schedule jobs on the Piz Daint supercomputer at the Swiss National Supercomputing Center (CSCS) and to launch new NRP instances. This process enables users to run, control and interact with embodied simulation experiments online as intuitively as is possible using local installations of the NRP, but backed by the considerable computing power of Piz Daint.

2. STATE OF THE ART

2.1. Large-Scale Neuronal Simulations on HPC Infrastructure

Several tools for the simulation of spiking neurons and networks thereof have been developed. They allow to model a high degree of biological plausibility but differ in their focus on different aspects of the biological models or the technology they use. Highlighting a few, NEURON (Hines and Carnevale, 1997; Awile et al., 2022) GENESIS (Bower and Beeman, 2007) and Arbor (Abi Akar et al., 2019) allow the modeling of complex compartmental neurons and are tailored to the simulation from the sub-cellular level to networks, while the *Open Source Brain* (Gleeson et al., 2019) provides functionalities for visualization and focuses on user collaboration and accessibility of neuron models and networks.

Due to the improved availability of compute resources for neuroscience research through programs like the Human

Brain Project's Fenix/ICEI[1], or the Neuroscience Gateway[2] and advances in simulation technology (e.g., Jordan et al., 2018; Kumbhar et al., 2019), it became routinely possible for computational neuroscientists to run large-scale simulations of spiking neuronal networks with great efficiency. Most modern neuronal network simulators achieve linear scaling for a large range of simulations of neuroscientific models and have thus opened the way to increased model sizes and more complex learning paradigms.

Meanwhile, a large number of projects are making use of these technological developments, which also resulted in a number of large-scale modeling publications (Markram et al., 2015; Senk et al., 2018; Igarashi et al., 2019; Billeh et al., 2020). Many of the studies are scaling to considerable portions of the world's largest supercomputers and reach far beyond the simple random balanced network that has been the norm in the field for many years. By integrating data from multiple neuroanatomical and electrophysiological sources, they enable the study of biological phenomena with an unprecedented level of detail. At the same time, the developers of the simulation tools are facing new challenges when it comes to coupling simulators amongst each other to increase the realism of the simulated models and to allow for an integration of physics simulators in scenarios such as those described in the present work.

2.2. Simulations of Spiking Neural Networks Controlling Virtual Embodied Agents

Previous works involving simulations of spiking neural networks connected to an embodied agent (either a robot or a musculoskeletal system) have mostly aimed at understanding motor control in the brain in relation to sensorimotor integration. Many of them focused on functional performance and were often carried out in a robotic context (e.g., Gilra and Gerstner, 2018; Bahuguna et al., 2019; Angelidis et al., 2021). Others more specifically investigated the robustness, versatility and capacity for adaptation of biological motor systems, for which there is still no satisfactory mechanistic explanatory framework. As an example, DeWolf et al. (2016) used the Neural Engineering Framework (NEF; Eliasmith and Anderson, 2004) to implement a multi-area brain model capable of controlling a three-link arm which also successfully exhibited adaptation to changes in arm dynamics and kinematic structure.

Other research efforts found in the literature involving spiking neural networks controlling a body were about replicating specific features of biological motor systems, with a focus usually placed more on simple movement generation rather than complex, behaviorally-relevant interactions with the environment (e.g., Allegra Mascaro et al., 2020; Fernándes et al., 2021; Kalidindi et al., 2021). In order to achieve task completion, these often involved some network training/optimization process, be it biologically realistic (e.g., STDP in Fernándes et al., 2021) or derived from AI approaches (back-propagation through time in Kalidindi et al., 2021). As for the simulation of musculoskeletal systems, it usually attempted to remain as biologically realistic as possible (e.g., through the use of Hill muscle models), but the experimental implementation did not provide straightforward means for reuse and reproducibility testing. Very few efforts reported in the literature besides the Neurorobotics Platform (see Section 2.3 below) actually focused on this aspect, which makes them all the more remarkable (e.g., Jordan et al., 2019). The latter introduces a toolchain to connect NEST with OpenAIGym making use of the MUSIC interface (Djurfeldt et al., 2010; Brocke, 2020). In Bahuguna et al. (2019), MUSIC is used to connect NEST with Gazebo. The Neurorobotics Platform connects physics and neural simulations directly using Nengo (Angelidis et al., 2021) or NEST (Allegra Mascaro et al., 2020).

The brain-body-environment trinity for different species at different levels of complexity from single body limbs to full body simulations has been simulated in multiple frameworks. The most popular example for invertebrates can be found in the OpenWorm platform (Szigeti et al., 2014; Sarma et al., 2018), which is made for the complete simulation of the Caenorhabditis elegans modeled with both its full body using fluid-simulation dynamics and the full neural network consisting of 302 neurons. A whole body simulation model including environment interaction of a vertebrate is found in Ferrario et al. (2021) with the simulation of a tadpole and serves as an experiment platform for research questions ranging from decision-making to movement generation. While both of the aforementioned simulation platforms are specialized for the given species, AnimatLab (Cofer et al., 2010) is a more generic simulation platform, which allows simulations of a wide range of vertebrates and invertebrates. Cofer et al. (2010) described a human arm flexion as an example.

The most complex work connecting a spiking model of the brain to a body can be found in Yamada et al. (2016). It describes a system encompassing a musculoskeletal model of human fetus at 32 weeks of gestation, a brain (2.6 million leaky integrate-and-fire spiking neurons and 5.3 billion synaptic connections) and some limited environmental modeling, which was used to comparatively study touch-driven cortical learning via limited embodied interactions under intrauterine and extrauterine environmental conditions.

2.3. Neurorobotics Platform

The HBP Neurorobotics Platform (NRP) is the backbone of the EBRAINS Closed-Loop Neuroscience service (Knoll et al., 2016). It provides access to a physically realistic simulated environment within which users can simulate and use all kinds of neural models (including spiking neural networks running on neuromorphic chips) composed into functional architectures, and connected to physical incarnations (musculoskeletal models or robotic systems). The simulation of the environment is carried out in Gazebo, an open-source robotic simulator. The neural models can be implemented using one of several frameworks, such as the software simulators NEST (Gewaltig and Diesmann, 2007) or Nengo (Bekolay et al., 2014), or the neuromorphic system SpiNNaker (Furber et al., 2014). The execution of the various simulators involved in a given NRP

[1] https://fenix-ri.eu/

[2] https://www.nsgportal.org/

simulation is orchestrated by a dedicated component referred to as the Closed-Loop Engine (CLE). The connections between the simulated agents' bodies and brains are entirely user-configurable, within the limitations imposed by the application programming interfaces (APIs) of the various simulators. The details of the connections are established through a dedicated framework of so-called Transfer Functions, which are responsible for the conversion and processing of data in transit for seamless recurrent communication between simulators. The NRP can be downloaded and installed locally for maximum experimental convenience, or accessed online in order to leverage the EBRAINS HPC infrastructure for large-scale experiments, as in the present case.

The functional connection of neural models to embodied agents allows neuroscientists to explore how the brain performs a number of tasks in closed loop, from lower-level sensorimotor tasks to higher cognitive functions (e.g., contextual awareness, decision making, etc.). The NRP thus enables cognitive and computational neuroscientists to explore the relationships that exist between the architectural characteristics of neural circuitry (usually constrained by anatomical and connectome data), neuronal dynamics (activity at either population or single-cell level), and their function expressed as the overt behavior of an embodied agent. Furthermore, *in silico* simulation provides a level of control over experimental parameters that enables studies that would be either technically impossible or ethically unacceptable. For example, only in simulation one can fully observe the effect of knocking out a particular ion channel in a specific neuronal sub-population with perfect efficiency, or carry out lesioning studies with perfect reproducibility. As such, the NRP provides a unique enabling platform to probe the functional consequences of e.g., stroke (Allegra Mascaro et al., 2020) or pharmacological tampering on the central nervous system. It is therefore a valuable tool to elucidate outstanding questions around motor control in both health and disease. However, until the work reported in the present paper, the NRP was run either locally or as a cloud service (i.e., on virtual machines). As such, the size of simulations that could be run on the NRP was limited by the typical computing resources of standard computers or virtual machines.

2.4. The Neural Simulation Tool NEST

NEST is a simulator for large networks of spiking neurons connected by phenomenological synapse models. It supports hybrid parallel simulations using threading within CPUs and the message passing interface (MPI) across multiple CPUs and computing nodes. In previous studies, NEST has shown excellent scaling over a large number of architectures even on the world's largest supercomputers (Kunkel et al., 2014; Jordan et al., 2018). The details of the parallelization are entirely transparent to the users, who do not need to handle or care about node placement onto processes or inter-process communication. Neurons in NEST can be anything from simple point neuron models like the integrate-and-fire neuron to complex compartmental neurons, as long as they can be expressed as a single C++ class. Synapses can be either static or change their weight over time according to a plasticity algorithm. Examples of such algorithms are spike-timing-dependent plasticity (STDP), short-term plasticity (STP), or third-factor neuromodulated weight dynamics. Many different neuron and synapse models have been developed over time and are included in any distribution of the NEST source code.

NEST can be used from Python by means of a module called PyNEST that wraps the NEST simulation kernel, which itself is written in C++. Simulation scripts can then use functions like `Create()` and `Connect()` to create neurons and devices for stimulation and recording, and to connect these elements using different connection rules, respectively. A web-based graphical frontend called NEST Desktop simplifies the task of network creation by offering graphical metaphors and a point-and-click interface and has been especially useful in classroom scenarios. To keep the actual simulation of the neuronal network separate from the graphical frontend, NEST was extended by the NEST Server, which allows steering NEST via a RESTful API that listens on a specific TCP/IP port and maps incoming requests of the form http://localhost:5000/api/Create to calls of the PyNEST API (the function `Create()` in the example).

When NEST is run in an MPI-distributed fashion, each process (or task, in MPI terminology) executes the same simulation script, but only creates its share of neurons, devices, and connections. The individual tasks also apply configuration changes only to local elements of the simulation and record data only from the entities they are responsible for. This is not a problem in many simulation experiments, where simulation scripts are run for the full simulation time and data is analyzed only after the simulation has finished and data from the different result files of the different processes has been manually combined. Due to the distributed nature of data collection in NEST, NEST Server originally only supported non-distributed simulation runs. To support the framework described in this work, NEST Server has now been extended to also support distributed scenarios by using a master-worker paradigm: The first MPI process (MPI rank 0, master) is responsible for both providing the RESTful API to clients, and forwarding all incoming commands to the workers (i.e., all MPI ranks but 0) and collecting their result data. In addition, the master process also participates in the neuronal simulation and thus also serves as a worker itself. A set of heuristics is used to combine and present the worker's response data to an outside caller as a consistent view that does not differ from one that the caller would see when only a single MPI process is used.

3. SOFTWARE ARCHITECTURE

The following provides the implementation details of a software architecture that integrates the Neurorobotics Platform and NEST Server for embodied simulations, supports browser-based online control of and interaction with experiments, and is highly scalable. This setup leverages a cloud computing infrastructure and HPC computing resources, both provided by EBRAINS. Despite the complexity of the architecture, automated deployment and online interactivity are provided through a dedicated graphical user interface available in the NRP frontend.

3.1. Infrastructure

The software service presented in this article is deployed across multiple compute systems at the Swiss National Supercomputing Center[3] (CSCS). Therein, persistent virtual machines are used in order to let users interact with the NRP continuously and request HPC resources in the form of compute nodes. The overall architecture is illustrated in **Figure 1**. The NRP frontend and a proxy responsible for assigning NRP backends and handling REST calls are deployed on a virtual machine on the *Castor* cluster, while the actual embodied simulations (NRP backend and NEST brain simulation) are run on requested compute nodes of Piz Daint. For every NRP job, at least two compute nodes are requested, the first running the NRP backend, the second and any additional nodes running NEST Server. As such, the setup is fully scalable in terms of computing capabilities and is able to support custom large-scale embodied simulations.

The software interface between NRP frontend on Castor and NRP backend on Piz Daint compute nodes is instantiated on demand. We implement a UNICORE[4] (Uniform Interface to Computing Resources) interface in the NRP proxy to interact with the SLURM workload manager (Yoo et al., 2003). UNICORE's REST API is used to request compute jobs, transmit configuration files and launch the NRP with NEST via startup scripts. We set up an SSH tunnel between the NRP frontend virtual machine and the cluster compute node running the NRP backend to enable bidirectional communication during runtime.

To facilitate fast and automated update cycles in a cloud infrastructure with our multi-component software architecture, we integrate all software components in Docker containers, in particular the NRP frontend, NRP backend and NEST Server. We use Jenkins with Ansible for Continuous Integration and Continuous Deployment (CI/CD); installation and Docker image instantiation on Castor is fully automated; new Piz Daint NRP backend images can be pushed to the Docker registry and then pulled to the Piz Daint login nodes using the Sarus container engine (Benedičič et al., 2019). The main advantage of this approach is the fast deployment of software improvements and new releases of the NRP and its components. This ensures forward compatibility of the platform during the ongoing NRP development. The architecture also allows multiple versions to be made available in the Docker registry so that custom software versions can be instantiated on demand.

3.2. Graphical User Interface

The setup is intended to enable future community access to an EBRAINS service that lets users experiment with large-scale embodied simulations without in-depth knowledge of the required underlying supercomputing infrastructure and architecture deployment. For this purpose, we implement a new section in the NRP frontend as shown in **Figure 2**, which can be accessed through a web browser. With it, users can request and launch the NRP on Piz Daint as compute jobs with customized resources, as well as manage instantiated jobs with running NRP instances. The job duration, compute node number and memory allocation can be customized depending on the duration and complexity of the experiments to be simulated. The frontend section also includes a list of past and running compute jobs, and lets users abort and inspect these during and after runtime. After starting the NRP backend distributed on requested Piz Daint compute nodes, it is accessible and can be selected just like any other backend running on virtual machines. Launching an experiment lets users interact with the rendered virtual environment and control the experiment interfaces and procedure runtime either graphically or programmatically via Python scripts in the NRP Virtual Coach. The Virtual Coach includes a Python REST interface to the NRP so that users can control simulations and observe its status via callback functions from a Python script. Additionally, experiment scripts can be modified programmatically and finally recorded data can be requested for postprocessing of experimental data.

3.3. NRP-NEST Coupling Architecture

Since the beginning of the development of the NRP, NEST has been a first-class citizen in the NRP platform. It was initially integrated through a direct import of the Python module for NEST into the NRP CLE, which entailed a number of drawbacks in terms of code maintenance and distribution on multiple compute nodes of Piz Daint. To overcome the main drawbacks of the previous coupling, we started from the existing solutions and devised a new architecture based on the idea of separating NEST from the NRP by channeling all communication through the NEST Server and its RESTful API: instead of importing PyNEST directly, the NRP would only talk to NEST via HTTP requests and responses. The change to this new architecture constitutes a minimally invasive change to the NRP itself, as all code can be encapsulated in a new module that implements the NRP interface specification for integrating brain simulators on the one side, and a client for the NEST Server on the other. By having NEST run in its own process space, the issues related to code maintenance are eliminated, because NEST can run on any suitable Python version independently, and the version of NEST does not have to be taken into account by the NRP as long as the RESTful API of the NEST Server remains unchanged. The requirement for running distributed simulations of the brain simulation is naturally fulfilled in the new architecture as long as the MPI-enabled version of the NEST Server is used. The complete new coupling architecture is depicted in **Figure 3**.

Within the new client-server based architecture of the NRP-NEST coupling, the NEST Client Python module exposes a number of API functions in the client-side user API (③ in **Figure 3**) that allow to configure a neuronal network and the needed devices, run a network simulation for a given amount of time, and retrieve the recorded data. No actual computation takes place within the client. Rather, the latter forwards all operations to the NEST Server, which is based on a master/worker paradigm in which the *master* (MPI rank 0) provides a RESTful API to the NEST client and coordinates the workers by exchanging data and control commands with them. All MPI ranks (including rank 0) together execute the neuronal simulation. By virtue of this split in responsibilities, the actual details of the distribution remain completely transparent to the NEST Client.

[3] https://www.cscs.ch/

[4] https://sourceforge.net/projects/unicore/

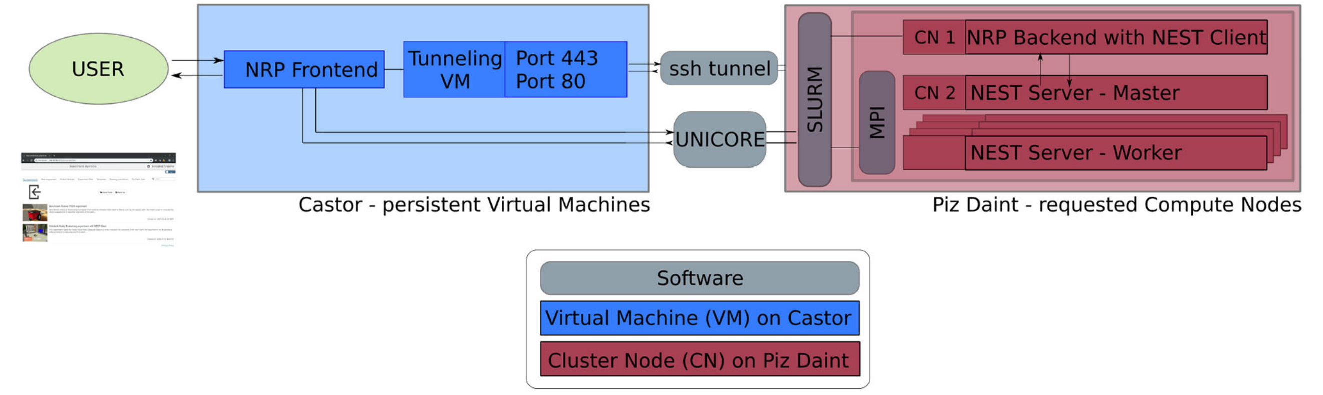
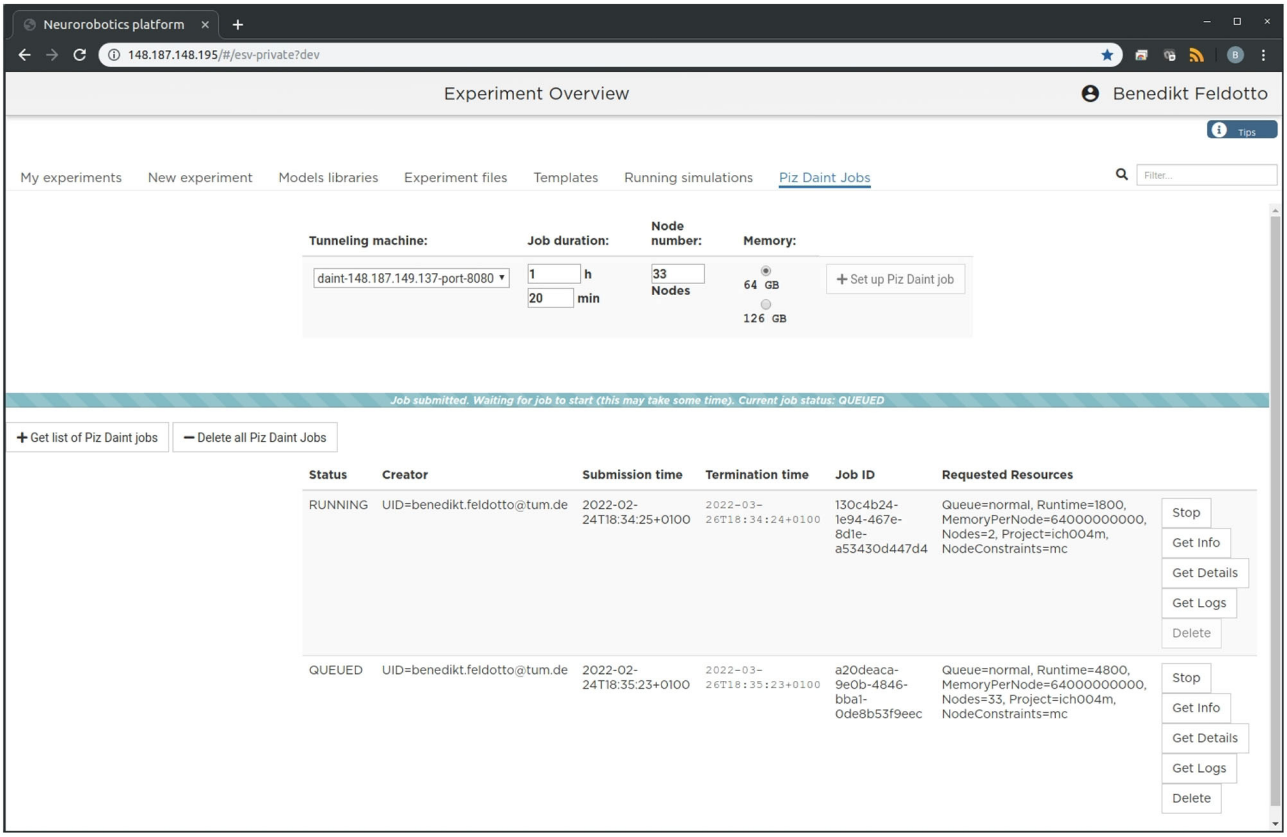

USER
NRP Frontend
Tunneling VM
Port 443
Port 80
ssh tunnel
UNICORE
Castor - persistent Virtual Machines
SLURM
MPI
CN 1
NRP Backend with NEST Client
CN 2
NEST Server - Master
NEST Server - Worker
Piz Daint - requested Compute Nodes
Software
Virtual Machine (VM) on Castor
Cluster Node (CN) on Piz Daint

FIGURE 1 | Software architecture. Persistent virtual machines are interfaced with requested compute resources in order to offer a flexible user interface with HPC resources. We use UNICORE as an interface to schedule compute jobs, and a SSH tunnel on demand establishes the bidirectional connection between NRP frontend and backend running on the two distinct computing systems.



Tunneling machine: daint-148.187.149.137-port-8080

Job duration: 1 h 20 min

Node number: 33 Nodes

Memory: ● 64 GB ○ 126 GB

Job submitted. Waiting for job to start (this may take some time). Current job status: QUEUED

Status	Creator	Submission time	Termination time	Job ID	Requested Resources	
RUNNING	UID=benedikt.feldotto@tum.de	2022-02-24T18:34:25+0100	2022-03-26T18:34:24+0100	130c4b24-1e94-467e-8d1e-a53430d447d4	Queue=normal, Runtime=1800, MemoryPerNode=64000000000, Nodes=2, Project=ich004m, NodeConstraints=mc	Stop Get Info Get Details Get Logs Delete
QUEUED	UID=benedikt.feldotto@tum.de	2022-02-24T18:35:23+0100	2022-03-26T18:35:23+0100	a20deaca-9e0b-4846-bba1-0de8b53f9eec	Queue=normal, Runtime=4800, MemoryPerNode=64000000000, Nodes=33, Project=ich004m, NodeConstraints=mc	Stop Get Info Get Details Get Logs Delete

FIGURE 2 | Graphical user interface. The large-scale simulation setup on HPC resources can be managed, accessed and controlled via a standard browser. We implemented a dedicated tab in the Neurorobotics Platform frontend that lets users parametrize supercomputing jobs and manage allocated resources. A new compute job running the NRP backend instance with distributed NEST can be requested and started with a single click from this frontend.

In terms of deployment, the described separation between client and server allows execution of the components on different computing units. In particular, it is now possible to execute the NRP (including the NEST Client) and the NEST Server codes on different nodes of a given supercomputer or compute cluster, or even on completely independent machines. All NEST-related operations such as loading the network, stepping the simulation, creating devices and connectivity, are managed by the

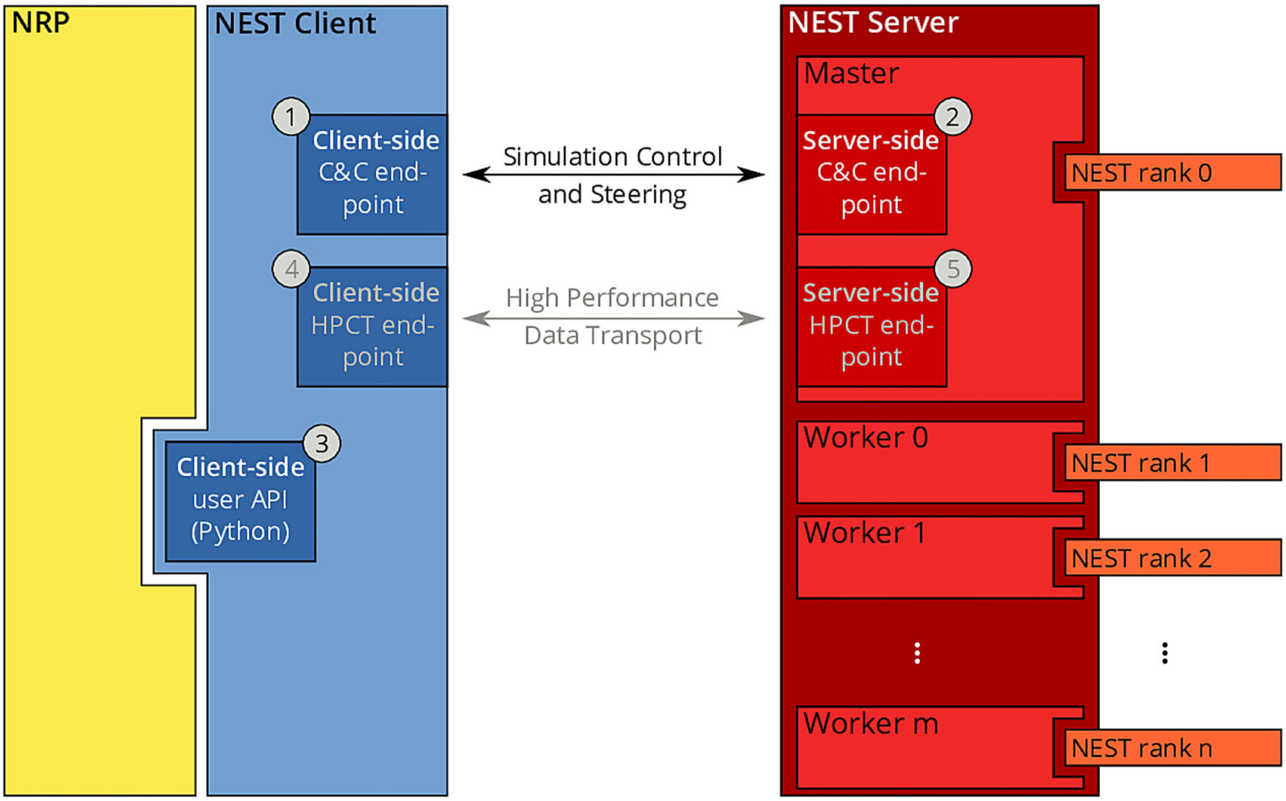

NRP
NEST Client
1
Client-side C&C end-point
4
Client-side HPCT end-point
3
Client-side user API (Python)
Simulation Control and Steering
High Performance Data Transport
NEST Server
Master
2
Server-side C&C end-point
5
Server-side HPCT end-point
NEST rank 0
Worker 0
NEST rank 1
Worker 1
NEST rank 2
Worker m
NEST rank n

FIGURE 3 | NRP-NEST client-server architecture. The NRP (yellow) connects to the NEST Client via its client-side user API for Python ③. The NEST Client (blue) provides a channel for talking to the NEST Server (between ① and ②) for simulation control and steering. A high-performance transport layer for data (between ④ and ⑤) is possible, but not yet implemented. The NEST Server (red) encompasses all NEST MPI processes (rank 0 to n), but only rank 0 (Master) offers the RESTful API visible to the outside world.

NEST Client that provides a set of user-friendly methods for all relevant operations.

The methods of the client-side API are (roughly speaking) just wrappers of the corresponding PyNEST functions. An example of such a method is `get_kernel_status()`, a call to which translates to an HTTP request for the URL `host:port/api/GetKernelStatus`, where *host* and *port* are the IP address of the machine on which NEST Server is running and the port it is listening on, respectively. Such a request results in a call of the PyNEST function `GetKernelStatus()` by the NEST Server. The return value of that function will be included in the response in the form of a JSON-encoded dictionary.

Below is a non-exhaustive list of the methods provided by the NEST Client client-side API to control and configure the simulation of the neuronal network in NEST:

- `get_kernel_status()`: access to NEST simulation parameters
- `startup()`: reset the kernel and set the number of threads and the simulation resolution
- `load_network()`: load a network in the form of a simulation script
- `run_simulation()`: drive a network simulation for a given amount of time
- `create_device()`: create a given number of network devices of a given type
- `connect_device()`: connect a device to a neuron using the provided connection parameters
- `set_device_params()`: set the given parameters on a device
- `get_population_parameters()`: retrieve parameters from a neuronal population

The second component of the coupling architecture, NEST Server, can be considered a language-independent interface to NEST that can be deployed either locally or on a remote machine as outlined above and in Section 2.4. Prior to any simulation, an instance of the NEST Server has to be started independently from the NRP and the NEST client with a degree of MPI parallelization that is suitable for the neuronal simulation at hand. As of writing, the NEST Server is fully integrated into the current release of NEST (Hahne et al., 2021) and can be either used after compiling the source from scratch, or from the NEST Docker image. All benchmark simulations presented in Section 6 have been realized using the containerized version of the NEST Server.

3.4. HPC Parallelization

Since the inception and widespread use of multi-socket/multi-core architectures several years ago, it has become more and more evident that a purely threaded application or one that purely relies on message passing for distributing the workload onto multiple processes is not sufficient for achieving optimal performance. Since then, the use of hybrid parallelization strategies that use threads within a CPU socket and message passing via MPI across CPU sockets and compute nodes has become the *de facto* standard for neuronal simulators. As this new paradigm has a high implementation complexity, many of the modern simulator codes shield the user from the details of the parallelization and provide suitable abstractions that also allow scientists not trained in computer science to use large-scale HPC machines efficiently.

For the NRP-NEST use case, we make use of the UNICORE REST API for requesting an individual number of compute nodes and running embodied simulations on compute resources customized to the user experiment. In contrast to a usual supercomputing job, in our case, not all compute nodes execute the same software but in fact run different sub-components of the overall architecture. After job approval, the execution of the architecture startup script is initiated via UNICORE, which launches the NRP backend, SSH tunneling service and NEST Server with workers on specific compute nodes. The general allocation layout is such that the user always requests $N + 1$ compute nodes, with the NRP and its NEST Client as well as the tunneling services started on the first node, and NEST Server on the remaining N nodes. This specific allocation of software components to compute nodes is done via the Slurm Workload Manager, assigning individual component execution scripts to the corresponding subsets of the overall allocated compute node list. Appropriate settings of thread pinning and process affinity are used to achieve good performance. For the deployment of NEST on Piz Daint for example, one MPI process is launched per physical CPU socket and set to use all of the 36 virtual and real cores by means of one OpenMP thread per core. NEST itself will then take care of the distribution of neurons and synapses onto the processes and threads by assigning neurons to threads in a round-robin fashion and allocating synapses on the process that is responsible for the post-synaptic neuron. We run two NEST workers on every individual compute node automatically (each assigned 36 CPU cores, see Section 5 for more details) to optimally use allocated compute resources. The described allocation scheme allows for fully customized scaling of computing capacity, which is only limited by the physical number of available nodes in the given HPC system.

4. MODELS AND SETUP

We implemented two different benchmark experiments in the Neurorobotics Platform to evaluate our software architecture: the first one is a rather synthetic balanced random network without any body connection; the second one is based on a biologically derived multi-region brain model connected to a virtual musculoskeletal rodent model.

4.1. HPC Benchmark With Balanced Networks

The random balanced network introduced by Brunel (2000) has been adopted by the NEST development community as a benchmark for large-scale simulations of spiking neural networks on HPC supercomputers (Morrison et al., 2007; Helias et al., 2012; Kunkel et al., 2014). This benchmark simulates a network with a large number of spiking neurons split into excitatory and inhibitory populations and random connectivity. The excitatory—excitatory synapses exhibit the multiplicative depression and power law potentiation model of Spike Timing Dependent Plasticity (STDP) described in the work of Morrison et al. (2007), while all connections targeting or originating from inhibitory neurons are static.

The number of neurons in the network corresponds to 11,250 multiplied by a scale parameter. The indegree of each neuron is fixed to 11,250 synapses regardless of the scale parameter. In this work, we use a scale factor of 20 yielding a network with 225,000 neurons and roughly 2.5 billion synapses. The network is simulated with a computational resolution of 0.1 ms for a duration of 1s. A four-wheeled Husky robot is loaded in a static virtual room but is left unconnected from the neural network and merely serves as a base workload for the NRP. In this benchmark setup, physics are simulated with Gazebo and the *ODE* engine.

4.2. Embodied Multi-Region Rodent Brain Experiment

The embodied multi-region rodent brain experiment aims to examine the dynamic mechanism of the cortico-basal ganglia-cerebellar-thalamic (CBCT) circuit in motor control through combined simulation of the brain model and the physical musculoskeletal model of a mouse. The embodied simulation includes 1,005,905 spiking neurons with 1,588,469,795 synapses in NEST. Neuronal output from the brain simulation controls the physical simulation of a mouse musculoskeletal model with 8 muscles in Gazebo and the *Simbody* physics engine. The NRP experiment view lets the user inspect, adapt and interact with the simulation online, **Figure 6** (bottom) shows the 3D rendering of the moving musculoskeletal body and the brain activity as a spike raster plot in the NRP frontend.

4.2.1. The Multi-Region Rodent Brain Model

The CBCT model is based on the biologically constrained spiking network models of the cerebral cortex (Ctx), basal ganglia (BG), cerebellum (CB), and thalamus (TH) (Gutierrez et al., 2020). The numbers of neurons in the CBCT loop add up to more than 90% of the number of all neurons in rodents, primates, and humans (Azevedo et al., 2009; Herculano-Houzel, 2009).

The model consists of a reference cortical patch of $1 \times 1\text{mm}^2$ and connected BG, CB and TH models with proportional number of neurons. In total, the model incorporates 1,005,905 neurons (**Table 1**). Simulations of such a large network combined with the musculoskeletal model requires efficient use of high-performance computing (HPC), especially for model optimization by repeated evaluations of the generated dynamics against experimental data. The NRP infrastructure provides the framework for managing access and execution by HPC.

TABLE 1 | Summary statistics of the 1×1 mm^2 unit of the rodent brain model.

Model	#Neurons	#Layers	#Neuron types
M1 (Ctx)	58,805	5	19
S1 (Ctx)	94,396	7	22
VL (TH)	6,144	2	3
VM (TH)	6,144	2	3
BG	10,976	5	5
CB (M1)	414,720	6	6
CB (S1)	414,720	6	6
Total	1,005,905	33	65

Parrot neurons and neurons instantiated by NRP devices as interface are not included.

Moreover, it allows easy and efficient integration of the brain model with physical models with realistic behavioral constraints, which facilitates better validation and improves predictive power of the simulated models.

The CBCT model of the multi-region rodent brain (**Figure 4**) is composed of the following regional models:

Cerebral Cortex: The model incorporates the primary motor cortex (M1) and the primary somatosensory cortex (S1), based on previous works of Igarashi et al. (2019), Sun Zhe (2019), and Sun and Morteza Heidarinejad (2019). A unit model has the size of 1,000 x 1,000 x 1,400 (height x width x length) μm^3 and contains six 2D sheets for the arrangement of neural populations in layers 1, 2/3, 4, 5A, 5B, and 6 based on reported cortical organization and experimental data (Lev and White, 1997; Weiler et al., 2008). Main neuron types are single bouquet (SBC) and elongated neurogliaform (ENGC) cells in layer 1; intratelencephalic (IT), parvalbumin-expressing (PV), and somatostatin-expressing (SST) neurons at layers 2/3, 5A, 6; and IT, pyramidal-tract (PT), PV and SST neurons in layer 5B (Jiang et al., 2013; Shepherd, 2013; Tremblay et al., 2016). The model also incorporates vasoactive intestinal peptide-expressing (VIP) neurons in layer 2/3 and connections based on Jiang et al. (2015). The S1 model Sun Zhe (2019) includes additional neurons in layer (L4).

For each layer (except layer 1), the numbers of excitatory and inhibitory neurons follow a ratio of 4:1, with a total number of about 58,000 and 94,000 neurons in 1×1mm^2 for M1 and S1, respectively. The spatial organization is based on pseudo-randomly generated neuronal positions uniformly distributed within layer boundaries. Connections are generated using several 2D Gaussian probability functions describing distance-based connectivity between excitatory and inhibitory neurons, including recurrent connections, in different cortical layers. The relative magnitude of the connections, as well as the parameters of the Gaussian functions, are taken from reported laser-scanning photo-stimulation and patch-clamp experimental recordings (Song et al., 2005; Weiler et al., 2008; Lefort et al., 2009; Xu and Callaway, 2009; Kätzel et al., 2011; Apicella et al., 2012; Avermann et al., 2012; Jiang et al., 2013; Pfeffer et al., 2013; Xue et al., 2014; Lee et al., 2015; Pala and Petersen, 2015). Leaky-integrate-and-fire models with conductance-based synapses from the standard NEST model library are used. To achieve resting and functional states, neurons are stimulated by bias currents drawn from normal distributions with optimized mean and standard deviation parameters.

Basal Ganglia: The BG model is a topologically organized version (Gutierrez et al., unpublished) of previous works from Liénard and Girard (2014) and Girard et al. (2020). Fixed parameters were defined based on biological constraints, while free parameters were optimized against electrophysiological recordings. The total number of neurons sum up to around 10,000 for rodents following a reference 1×1mm^2 cortical surface (**Table 1**), with most of them being medium spiny neurons (MSN). Neurons were spatially and uniformly organized in 2D space. Main inputs are from cortico-striatal neurons (CSN) and pyramidal tract neurons (PTN) in the cortex (M1 and S1) and the centromedian/parafascicular neurons (CMPf) in the thalamus (TH). The model considers glutamatergic excitatory inputs with AMPA and NMDA receptors and inhibitory inputs by GABA receptors. The model uses multi-synapse LIF neuron models from NEST. Connections follow the same architecture as in Girard et al. (2020), with specifications based on optimized bouton counts, and focused or diffused axonal domains. Simulation tests reproduced the firing rate of previous models in the resting state.

Cerebellum: The CB model consists of two regions connected with S1 and M1. Each cerebellar region is a corticonuclear microcomplex model developed in NEST based on the previous work of Yamaura et al. (2020). The cerebellum is modeled as seven stacked layers corresponding to 1×1mm^2: upper and lower molecular layers, Purkinje cells, granular layer, deep cerebellar nucleus, and Pons (Eccles, 1967). The upper molecular layer was modeled as a group of four 2D layers of stellate cells, while the lower one as a single sheet of basket cells. Similarly, the granular layer was composed of eight sheets of granular cells and one sheet of Golgi cells. All other nuclei were modeled within single sheets. Number of neurons (**Table 1**) for each population were defined from previous data (Lange, 1975; Ito and Itō, 1984; Harvey and Napper, 1991; Heckroth, 1994). The cerebellum contains around 80% of the neurons (around 820,000 neurons) of our full brain model. Neurons were modeled as conductance-based leaky integrate-and-fire units, with parameters defined based on previous studies by Yamaura et al. (2020). Excitatory synapses were modeled as AMPA or NMDA, and inhibitory as GABA-A or GABA-B alpha-shaped synapses. Connections were settled according to known anatomical structures (Eccles, 1967; Apps and Garwicz, 2005; Barmack and Yakhnitsa, 2008), using 2D Gaussian functions for defining the spatial scope and connection probability between neurons. Most internal parameters such as capacitances, conductances, and synaptic weights were tuned and tested to reproduce electrophysiological and behavioral results on optokinetic responses, a cerebellum-dependent eye movement task based on the previous work by Yamaura et al. (2020). On the other hand, firing rates and synaptic weights for neurons in Pons were adjusted to obtain the mean firing rate of mossy fibers at 8 Hz, which resulted in reproducing plausible resting activity patterns. At that regime, granular cells revealed different temporal activity patterns, with random repetition of transitions between burst and silent states.

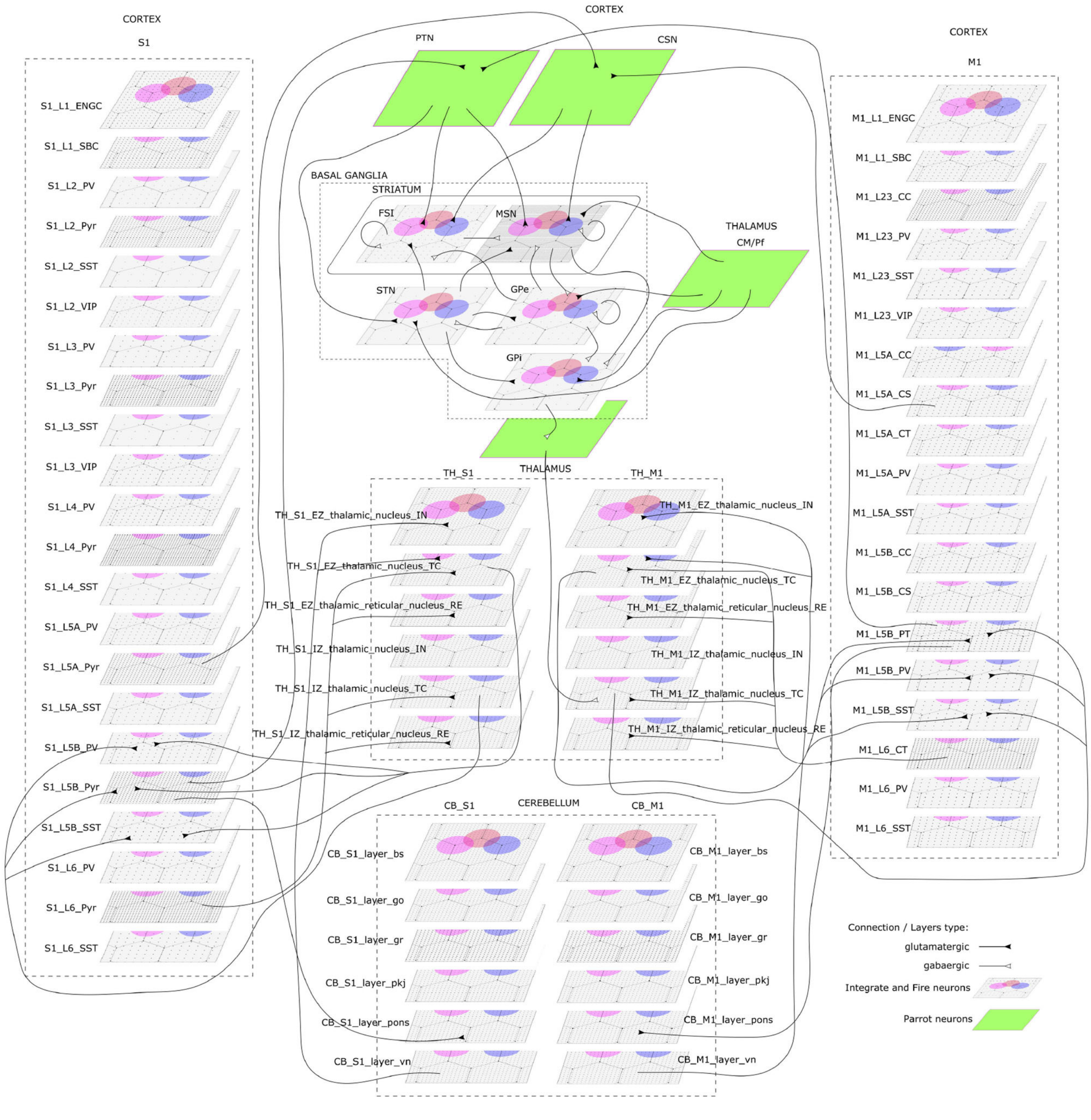


FIGURE 4 | The cortico-basal ganglia-cerebellar-thalamic (CBCT) model of the rodent brain. The model includes the cortex (S1, M1), the basal ganglia (BG), the cerebellum (CB), and the thalamus (TH). Within each region, neural populations are topologically organized in 2D-layers of 1×1 mm^2, with dots on their surface indicating the spatial allocation and density for each neuron type. Only main inter-regional connections are displayed for clarity. Layers in green correspond to the interface between different regions or simulated input, which are implemented using NEST's *parrot* neurons that just relay incoming spikes to multiple targets.

Thalamus: The TH model (Igarashi et al., unpublished) consists of two regions, ventral lateral nucleus and ventral medial nucleus, connected with M1 and S1, respectively. The individual thalamic nucleus is composed of excitatory and inhibitory zones receiving inputs from the cerebellum and basal ganglia. Each region-zone contains 1024 excitatory thalamocortical cells, 1024 inhibitory interneurons, and 1024 inhibitory thalamic reticular cells, arranged in a unit size corresponding to 1×1mm^2 of the cerebral cortex. Thalamocortical cells and two types of inhibitory neurons are mutually connected, with no excitatory recurrent connections among thalamocortical cells.

Inter-regional connections: Inter-regional connections are set as topographic connections between two neural sheets. Major inter-regional pathways include: M1 L5A to BG Striatum, S1

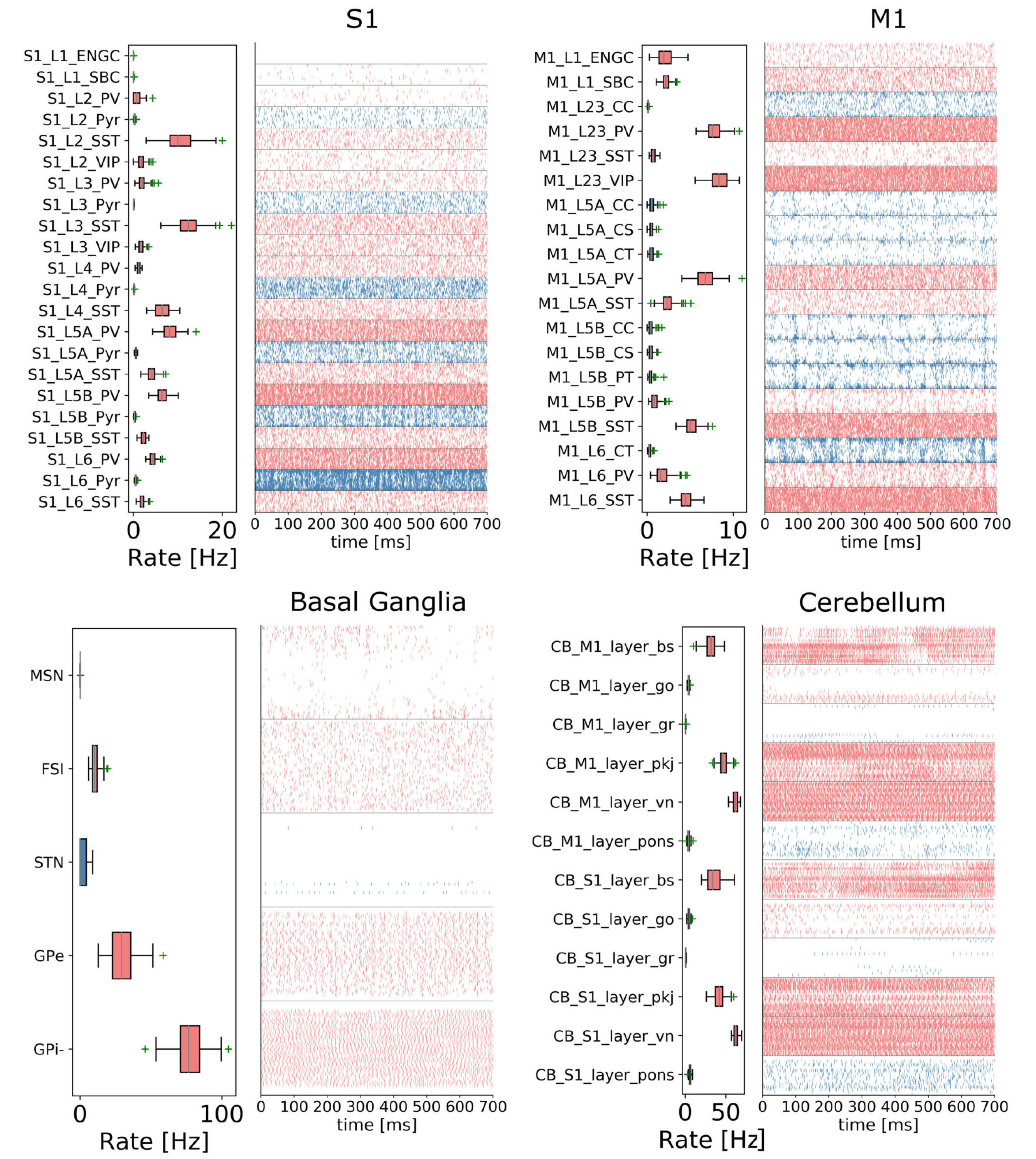


FIGURE 5 | Resting-state of the CBCT circuit (Gutierrez et al., 2020). Spike rasters (right) and mean firing rate (left) per neuron type (thalamus activity is not displayed).

L5A to BG Striatum, BG GPi/SNr to TH, M1 L5B to CB Pons, S1 L5B to CB Pons, CB deep cerebellar nucleus to TH, M1 L6 to TH, S1 L6 to TH, TH to L2/3 M1, and TH to L4 S1. A major challenge when integrating different models is to guarantee their optimized activities are maintained after combination. For instance, in the basal ganglia model, inputs from cortical models

(M1 and S1, layers L5A and L5B) were adjusted to match the firing activity of PTN (pyramidal track neurons) and CSN (cortico-striatal neurons) inputs from Poisson spike trains used on the optimization of the isolated model. NEST's *parrot* neurons (models that just relay incoming spikes to their targets) were used to gradually replace Poison-based neurons by M1 and S1 based neurons. Thus, inputs involved in inter-regional connections were adjusted to those used on individual optimizations, using or not using parrot neurons on the connections.

4.2.2. Resting-State Activity

In order to reproduce resting-state neural activity that is simulated in this benchmark experiments, Poisson noise generators and constant current inputs were optimized to reproduce the average firing rates of individual neural populations based on physiological data (**Figure 5**). In S1, M1, and TH, neurons showed low-rate and irregular firing. Layers 5 and 6 in S1 generated gamma oscillation of around 40 Hz. Similarly, M1 bottom layers displayed oscillatory behavior. GPe and GPi/SNr in BG showed high-rate firing while others were kept low. In CB, Purkinje cells exhibited regular firing patterns, whereas granule cells emitted spikes sparsely. We acknowledge that the spiking activities of few neural populations could be slightly higher. While this model is a first version of the CBCT model used for benchmarking of the architecture presented here, a future release of our model aims to improve firing activities as well as other metrics.

4.2.3. Embodied Simulation

The multi-region brain model is embodied into a simulated rodent musculoskeletal model and a virtual environment in the NRP. We replicate the physical experiment platform introduced in Mathis et al. (2017) as a simulation model *in silico* in the Neurorobotics Platform. In this model, the animal is held in place, rewarded by a Lickometer, and is able to manipulate a joystick to which additional forces can be applied via a linear solenoid magnet. We modeled a rodent housing, Lickometer and joystick in the Neurorobotics Platform using the Robot Designer[5] plugin for Blender as part of the Neurorobotics Platform design tools. The joystick was connected to the world with two revolute joints representing two degrees of freedom. The mouse manipulated the joystick with its left forelimb, while a small effort of −0.001 Nm is applied to the joystick joint.

For the musculoskeletal system, we adapted a rodent simulation model that has been used in a stroke rehabilitation study in the Neurorobotics Platform recently (Vannucci et al., 2019; Allegra Mascaro et al., 2020). The skeleton thereof was modeled according to anatomical data and scans, and is an early version of the fully parameterized rodent model presented in Ramalingasetty et al. (2021). We anchored the rodent body model to the experimental apparatus, leaving only three moving segments of the left forelimb capable of movement: humerus, ulna/radius and the foot. Body and humerus were connected via two revolute joints, humerus and ulna/radius via one revolute joint and the foot was attached to the joystick via a ball joint with 3 degrees of freedom. With this configuration, the mouse was able to move the joystick in the forward/backward and lateral/medial directions. The skeleton was simulated as a rigid-body simulation with the *Simbody* multibody physics engine in Gazebo. We added 8 muscles to the forelimb joints, 2 for every rotation axis, with 2–5 muscle pathpoints each. Muscles were simulated with the OpenSim muscle implementation (Delp et al., 2007) and modeled with type "Millard2012EquilibriumMuscle" as described in Millard et al. (2013). Every muscle was actuated in normalized range [0,1]. **Figure 6** illustrates the overall setup rendered in the Neurorobotics Platform frontend.

For the benchmark experiments in this study, we set up a naive representative brain-to-body connection. We connected one layer, the elongated neuroglia form cells of the motor cortex, to three muscles of the rodent model. For this we made use of *spike sinks* that read out the membrane potential of a leaky integrate-and-fire neuron with infinite threshold and connected to all neurons of the given population, and apply it as muscle activation signal. We also instantiated spike sinks for all layers in M1 and logged the corresponding voltages in the NRP frontend console for inspection. Additionally, we created spike sources consisting of Poisson neurons connected to all neurons of the given population for three layers of M1 including the elongated neurogliaform cells.

A base activation was sent to all muscles for the first 5 simulation steps (corresponding to 0.1s of simulation time) to stabilize the biomechanical model. Additionally, all spike sources including the source to the elongated neurogliaform cells of the motor cortex were set with a spike rate of 0 in every iteration. After 5 simulation steps, the clipped voltage readout of the elongated neurogliaform cells was applied as activation value to three muscles continuously (muscle activation in range [0,1]). Reaching 25 simulations steps (0.5s simulation time), we feed a rate of 5000.0 into the Poisson generators representing the spike source of the elongated neurogliaform cells. As a result, spike activity in this layer rose and the brain layer readout devices transmitted an increased muscle activation to the aforementioned three muscles. The overall experimental procedure resulted in a loose stabilization of the joystick in the first 25 simulation steps (0.5s of simulation time), followed by a forward motion of the rodent leg pushing the joystick forward as a consequence. Starting after 5 simulation steps a status message was shown in the frontend to indicate the current state of network input activation repeatedly to the user.

5. BENCHMARK EXPERIMENT PROCEDURE

We ran the benchmark experiments on the XC40 multicore compute nodes of Piz Daint[6]. Each node of this partition is equipped with two Intel® Xeon® E5-2695 v4 18-core CPUs running at 2.10 GHz (2 x 18 cores, each having 2 virtual cores) and 120 GB of RAM. All experiments were executed with

[5] https://github.com/HBPNeurorobotics/BlenderRobotDesigner

[6] https://www.cscs.ch/computers/piz-daint/

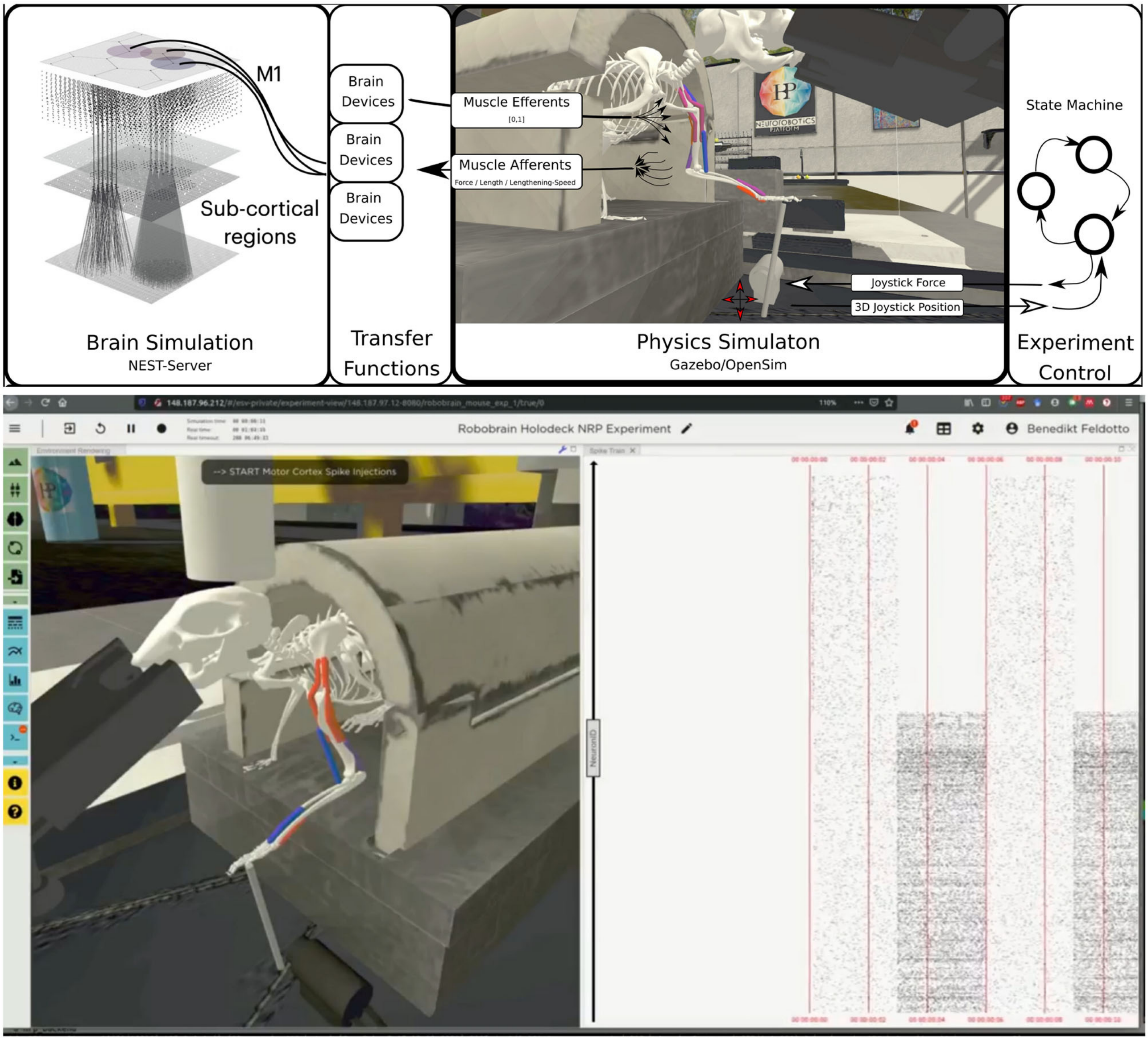


FIGURE 6 | Embodiment of the multi-region rodent brain. (Top) The multi-region brain model is interconnected with the rodent musculoskeletal simulation via Transfer Functions. The readout rate of motor cortex populations actuates the rodent muscles moving the joystick forward. (Bottom) The user can interact with the simulation during runtime via the NRP frontend. Here we show the rendering of the simulated experiment on the left (muscle color coding: red–active, blue–not active) and spike trains of the motor cortex population on the right.

the Neurorobotics Platform version 3.2 and NEST version 3.0 (Hahne et al., 2021).

We executed one NEST process per CPU using all 36 (virtual) cores, hence at most two NEST processes per compute node. Every experiment ran for 1s of simulation time (assigned as a timeout to every experiment), consisting of 50 CLE step times of 20ms each (meaning that data between body and brain was exchanged every 20ms in simulation time). We carried out a series of benchmark experiments, starting with a single NEST process and scaling up to 64 processes, doubling the process number at each run. At the beginning of every benchmark series, we requested 33 compute nodes (1 NRP node, 32 NEST nodes) to ensure all runs in the same series that included a variable number of NEST processes were executed on the exact same node allocation. Every benchmark series was repeated multiple times with a new node allocation every time. Hereafter, we report the first 8 successful repetitions of every benchmark experiment. For reproducible experiment execution, we instantiated the NRP NEST setup with scripts directly on a Piz Daint login node. This experiment procedure allows us to access and collect all recorded performance data from different sources directly. After launching the framework, the NRP experiment was controlled from the main script via the NRP Virtual Coach. The experiment procedure is presented in the pseudocode below:

```
for number of benchmark repetitions do
    Salloc - Request Piz Daint job with 33 nodes
    NRP - Launch backend container on compute node #1
    SSH - Set up SSH tunnel from NRP backend to frontend
    for number of NEST tasks n = $2^i$ do
        NEST - Launch NEST with n processes on n/2 cluster nodes
        Virtual Coach - Import benchmark experiment into NRP storage
        Start experiment runtime timer
        Virtual Coach - Launch NRP benchmark experiment
        Virtual Coach - Start NRP benchmark experiment
        while Experiment is running do
            Virtual Coach - Wait for experiment to be finished
        end while
        Stop experiment runtime timer
        Virtual Coach - Save CLE profiler performance data
        NEST - Save network performance data
        Sacct - Save job performance data
        Virtual Coach - Delete benchmark experiment from NRP storage
    end for
end for
```

After every experiment we collected performance data from the NEST Server (neural network creation time, connection time and duration of last simulation step), the NRP CLE profiler (brain/robot/transfer function step times) and Slurm workload manager (memory and energy consumption). The total experiment runtime was tracked manually as shown in the pseudocode and the real-time factor was calculated as the quotient of CLE step simulation time to real time, whereas the CLE real time was taken to be the mean value of all CLE step times except the first one (indeed, the first CLE step executes initialization procedures, hence is significantly larger and does not reflect the CLE step time of the overall experiment). The code to run the benchmark and the results presented in this paper can be found in the GitHub repository https://github.com/HBPNeurorobotics/nestserver_benchmarks.

6. RESULTS

We first executed and evaluated the HPC Benchmark experiment based on random networks without brain to body connection, and afterwards ran the multi-region rodent brain model with connection to the musculoskeletal rodent model. For a succeeding comparison between the benchmarks we added a third configuration that is a subset of the embodied multi-region rodent brain experiment with only the motor cortex as the brain model. This configuration shall not represent a biological simulation, but instead serves purely as a benchmark since a midsize brain in connection with the musculoskeletal model provides additional insights for the distribution of computation required for the simulation of brain and body.

Diagrams showcasing the compute node scaling use logarithmic scaling on the *x*-axis. We also present a linear expectation starting from the first point as the mean of all first data points without outliers that are not in range mean ±12%. The CLE profiler times represent a random benchmark repetition (here, the 4th), the *y*-axis is clipped as the initialization step takes significantly longer than usual runtime executions.

6.1. HPC Random Balanced Network Benchmark

The HPC Benchmark showed good repeatability with only a small variance between the benchmark runs. The runtime (**Figure 7**) could be reduced exponentially close to the linear expectation from about 500 to about 40 s, more than 12 times faster, when increasing the number of NEST processes from 1 to 64. The real-time factor increased exponentially first, but only up to about 8 processes; with more than 16 NEST processes a partial saturation appeared that resulted in an increase of the real-time factor up to 64 processes, albeit with a smaller slope. Overall the real-time factor could be increased from around 0.0036 to 0.150, a factor of more than 40. The NEST procedures scaled very well generally. The time required to build the network in NEST (i.e., creating and connecting nodes, **Figure 8A**) scaled nearly linearly. Simulation time (**Figure 8B**) scaled supra-linearly, but reached the same time performance as a linear scaling would have with 64 processes. Scaling up from 1 to 64 processes, the network building time could be reduced by a factor of about 61 and the time to simulate a brain step by about 60. The maximal memory required by a single HPC node (**Figure 8C**) could be reduced nearly linearly, from a maximum resident set size of about 87GB down to approximately 2.5GB. Along this scaling the amount of consumed energy (**Figure 8D**) did not increase linearly, but only by a factor of about 12 from 104kJ up to 1,200kJ. We observed that the three procedures of brain, robot and Transfer Function execution (**Figure 9**) all have an initial simulation step that takes significantly longer than the usual step time and hence is not considered in our analysis. In line with expectations, robot and Transfer Function

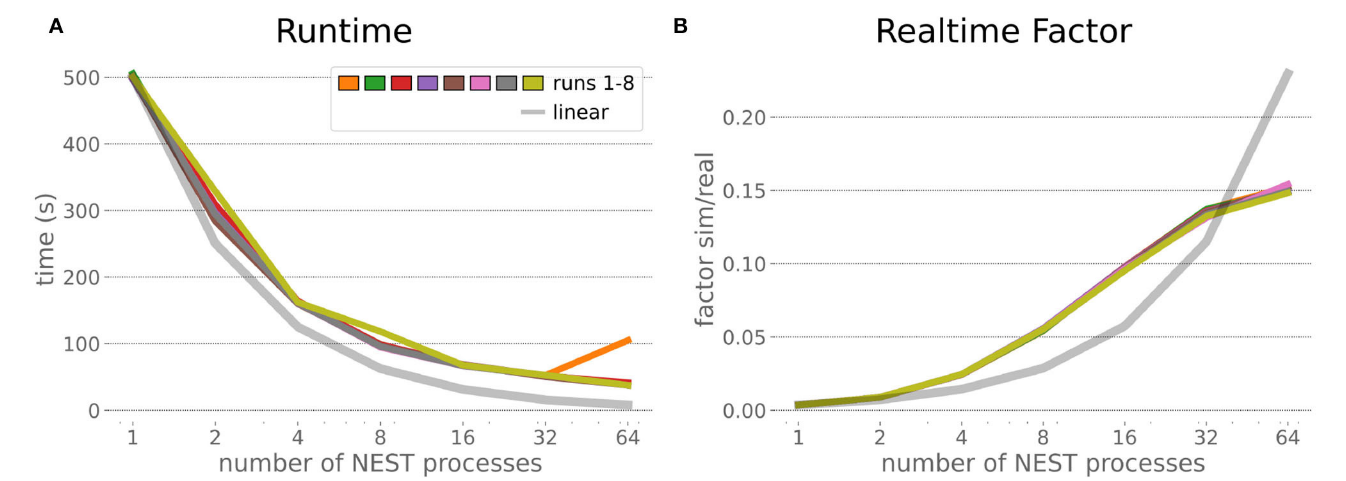


FIGURE 7 | HPC Benchmark runtime and real-time factor. The runtime **(A)** can be reduced by a factor larger than 12 from 500s to about less than 40s by exploiting 64 NEST processes on 32 compute nodes compared to a single process. Simultaneously, the real-time factor **(B)** improves supra-linearly, but performance increases less significantly when using more than 32 NEST processes.

execution step time varied over time but were not affected by the experiment parallelization. We observed a decrease of brain simulation time with increased numbers of NEST processes, with a slight overshoot in the first simulation steps and then stabilization at a mean value. As the robot and transfer function step times are relatively low in contrast to the brain execution, the latter one prominently defines the experiment runtime speed. Overall, the HPC Benchmark scales well, in most aspects nearly or supra-linearly, the one (beneficial) exception being the less-than-linear increase of consumed energy.

6.2. Embodied Multi-Region Rodent Brain

The increased network size in the embodied multi-region rodent brain experiment compared to the balanced network benchmark (only 0.6 times the number of connections (1,588,456,283 vs. 2,531,475,000), but 4.8 times more network nodes compared to the balanced network benchmark (1,089,147 vs. 225,001) resulted in a larger overall experiment runtime as well as individual step execution times. The total runtime of the experiments showed a higher variance in repetitions (**Figure 10**) compared to the HPC Benchmark, which can be partly attributed to the larger execution times in general, and shows less than linear duration decrease but still a big improvement in time. The execution runtime could be reduced from about 600s down to about 100s, with two runs decreasing the runtime only down to about 200s during scale-up. Experiment runs that lasted longer usually took longer runtime in all node configurations in comparison to mean runtimes. The real-time factor of the experiment increased however only slightly and saturated using about 32 NEST processes, with a small decrease with 64 processes. This real-time factor was improved from about 0.0069 to 0.0480 (for 32 processes) during the scale-up, a factor of around 7. NEST procedures scaled exponentially (**Figure 11**), the simulation time (**Figure 11B**) close to linear, building time (**Figure 11A**) with a somewhat flatter decrease. The network building time could be sped up by a factor of more than 17 and the time for the last simulation step by a factor of about 30. The required memory did scale close to linear (factor of 17) to the number of nodes, and the consumed energy again increased far less than linearly, from about 120 kJ up to 1,450 kJ, by a factor of about 12 (**Figures 11C,D**). In contrast to the HPC Benchmark with balanced networks, execution times for robot and transfer functions changed over time (**Figure 12**), along with the scripted experiment procedure. We could clearly see an increase of computation time required by the Transfer Functions when a layer of the motor cortex was addressed with even a fixed spike rate of 0 after 5 execution steps. Changing the input rate to a higher value at 25 CLE steps did not have an impact on the execution time. We observed that the Transfer Function execution time increased slightly when scaling the experiment up to 64 NEST processes running on 32 different compute nodes. However, the execution time of Transfer Functions was still low compared to the brain execution time.

The robot execution time increased up to about 0.08s, reaching the highest values at around 25 simulation steps. It decreased afterwards at about 27 simulation steps and remained at a relatively low value of around 0.02s until the end of the benchmark time being 50 simulations steps. Brain step execution times showed less variability compared to the balanced network benchmark experiment, and were much higher in general than robot and Transfer Function execution times. Overall, the embodied multi-region rodent brain benchmark did not scale as well as the HPC Benchmark and showed more variability in terms of execution times. However, regardless of the network size, nearly all inspected timings still scaled close to linearly in relation to the number of nodes, which thus can be taken to speed up the experiment execution and decrease its runtime significantly.

6.3. Comparison

In order to optimize the experiments at scale, it is important to examine where the largest potential for improvements is,

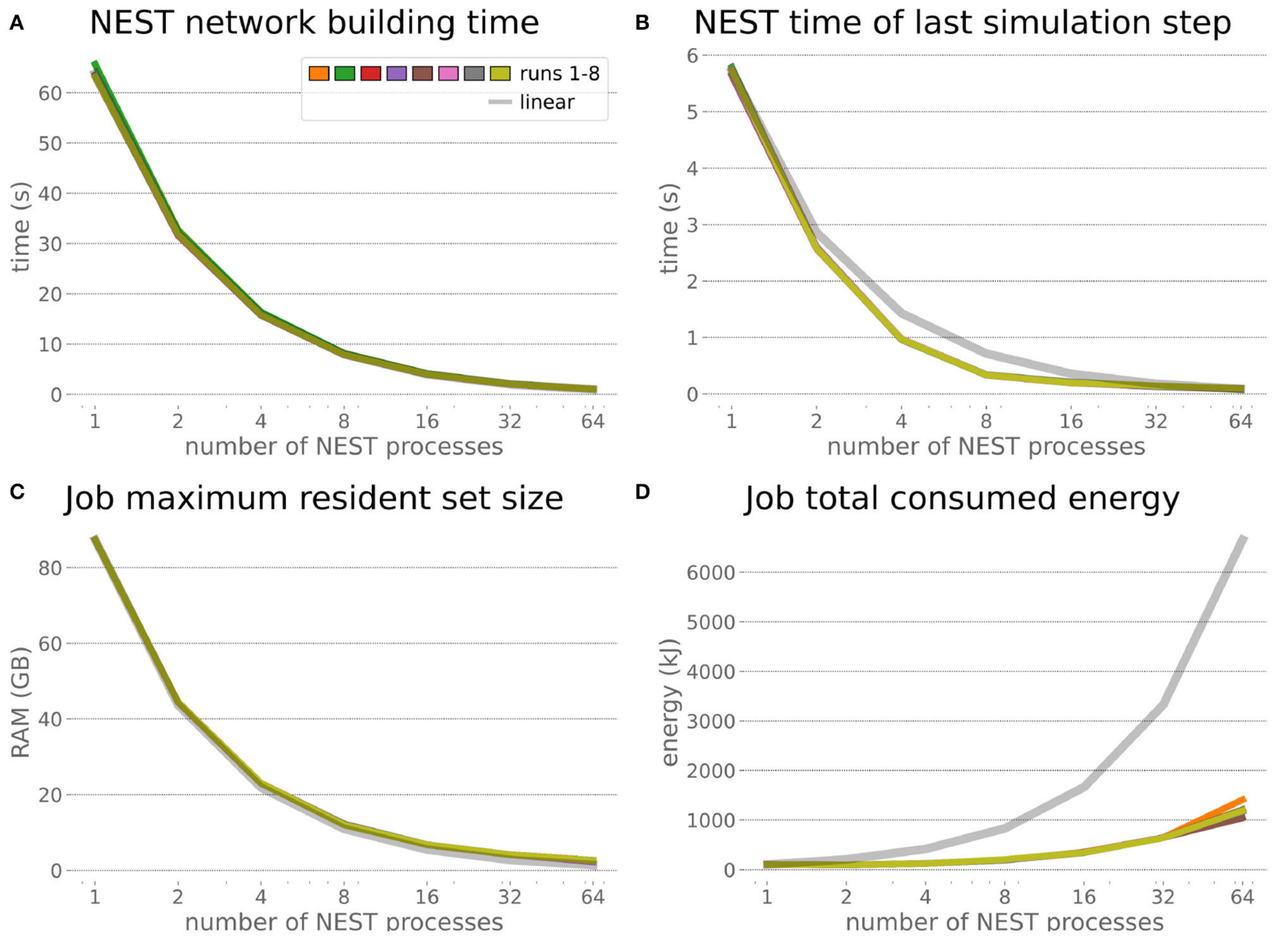


FIGURE 8 | HPC Benchmark NEST times and workload manager characteristics. **(A)** Network building time (i.e., creating and interconnecting nodes) scales nearly linearly. The time to simulate the last brain step **(B)** scales even supra-linearly with 2–32 NEST processes. Similarly, the required memory per compute node **(C)** reduces close to linear, but the total energy consumed **(D)** by all tasks is only 12 times more for 64 NEST processes compared to 1 process.

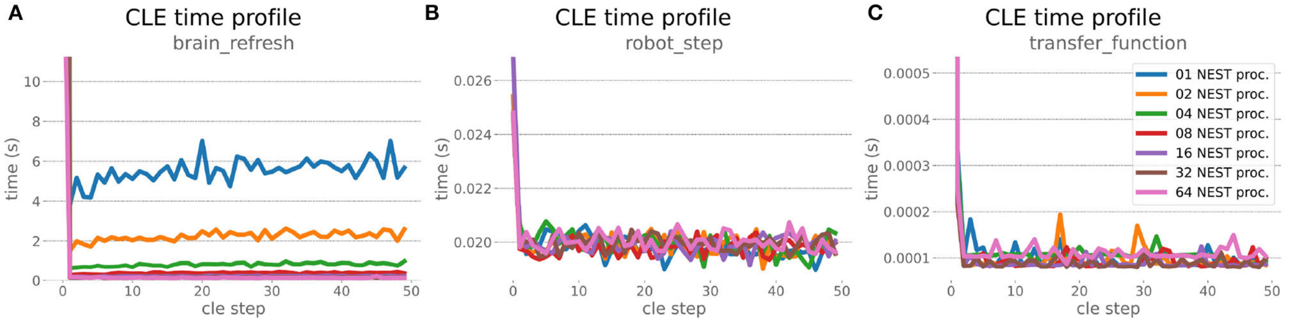


FIGURE 9 | HPC Benchmark CLE profiler times. Robot **(B)** and transfer function **(C)** execution times do not change during the scaleup, as they are not parallelized and just run on the first compute node in the allocation. Both are neglectable compared to the brain step time **(A)** that runs faster with additional compute nodes. The first timestep includes additional initialization procedures and hence takes significantly longer than the usual runtime step time, in the diagrams we clip the y-axis for better visibility of the relevant runtime data.

and what the costs related to scaling up execution will be. Therefore, we inspected the brain-to-robot compute time ratio as well as consumed node hours for all executed experiments and executed a third benchmark run that consisted of the embodied

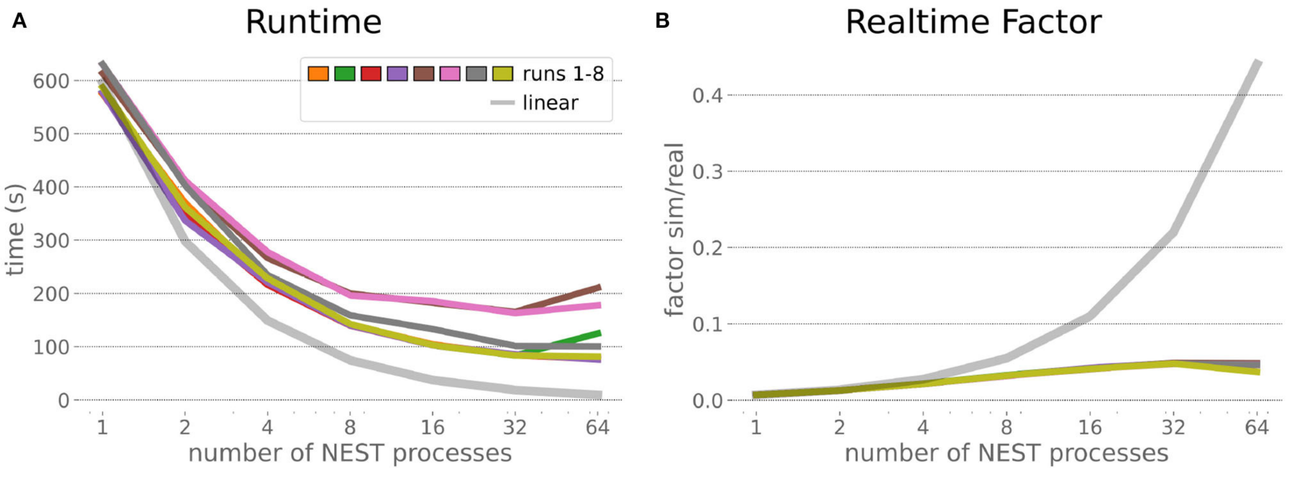


FIGURE 10 | Embodied multi-region rodent brain benchmark runtime and realtime factor. Experiment runtime **(A)** shows a variability in repetitions, but can be improved exponentially by a factor of about 6 when scaling up to 64 NEST processes. The realtime factor **(B)** can be improved up to about 0.048, but starts saturating from 32 NEST processes onwards.

multi-region rodent brain setup, but with only the motor cortex as an active brain region.

In the NRP, at every CLE simulation step both robot and brain simulations are executed in parallel, and only after completion of these steps are Transfer Functions executed to process information to be communicated between both simulations. Obviously, when either one of the robot or brain simulation takes consistently longer to execute than its counterpart, that component becomes the target for optimizing the overall NRP simulation. In the top part of **Figure 13**, the ratio between brain and body simulation step time is visualized; times are mean values over all 8 benchmark repetitions. As both simulations are executed in parallel, the most efficient performance is achieved with both having the same execution time. For all experiments reported herein, the brain simulation step took longer than the robot simulation step. With our distributed architecture the ratio between the parallelized brain simulation step time and the (shorter) robot simulation step time improved as the brain simulation step time was reduced by distribution. This effect was less significant for small neural networks such as the embodied rodent brain experiment with a motor cortex only (B), but was very relevant for the full embodied multi-region rodent brain experiment (C) and balanced networks benchmark experiments (A). For both these large neural networks, the ratio between the two simulation time steps improved with the number of NEST processes, with the best result obtained for the 64 NEST processes we tested for these benchmark experiments. In the bottom part of **Figure 13**, the required node hours for every benchmark were calculated as the product of the pure experiment runtime, including experiment launch and execution but excluding the overall architecture setup and initialization, and the utilized number of nodes. As can be seen, the number of required node hours scaled less than linearly, i.e., exponentially but with small increments when scaling up the utilized node number. For the node hours of the HPC Benchmark (D) with balanced networks, the increase was by a factor of less than 14 from about 0.14 to 0.67/1.87 (best case/worst case), whereas for the embodied rodent brain experiment with Motor cortex only (E) it was by a factor of less than 35 from around 0.04 to 0.71/1.39, and for the full embodied multi-region rodent brain experiment (F) (which consumes the most resources), the consumption increased from 0.17 to 1.35/3.75 by a factor of less than 23. We also observed a higher variability of number of required node hours with increasing experiment complexity, and the embodied rodent brain experiment with only motor cortex showing the steepest increment.

7. CONCLUSION

In this paper, we presented a distributed architecture for large-scale embodied simulations of spiking neural networks, together with the results of benchmark experiments run on our setup. We sought to develop the software components of a future simulation service on the EBRAINS research infrastructure, while at the same time understanding the benefits and drawbacks of distributing simulations across nodes of the Piz Daint supercomputer.

For this purpose, we connected the Neurorobotics Platform for physics simulation via a REST interface to NEST for simulation of spiking neural networks used as brain models. We distributed this brain simulation across multiple HPC compute nodes via MPI parallelization, and thereby sped up both experiment loading and execution times. The proposed software architecture can be controlled via a browser-based graphical user interface integrated into the NRP frontend, and it extends across both persistent virtual machines and HPC compute nodes. To facilitate the technical implementation, we utilized standard tools such as Docker for containerization, Jenkins for automated deployment, and UNICORE for HPC

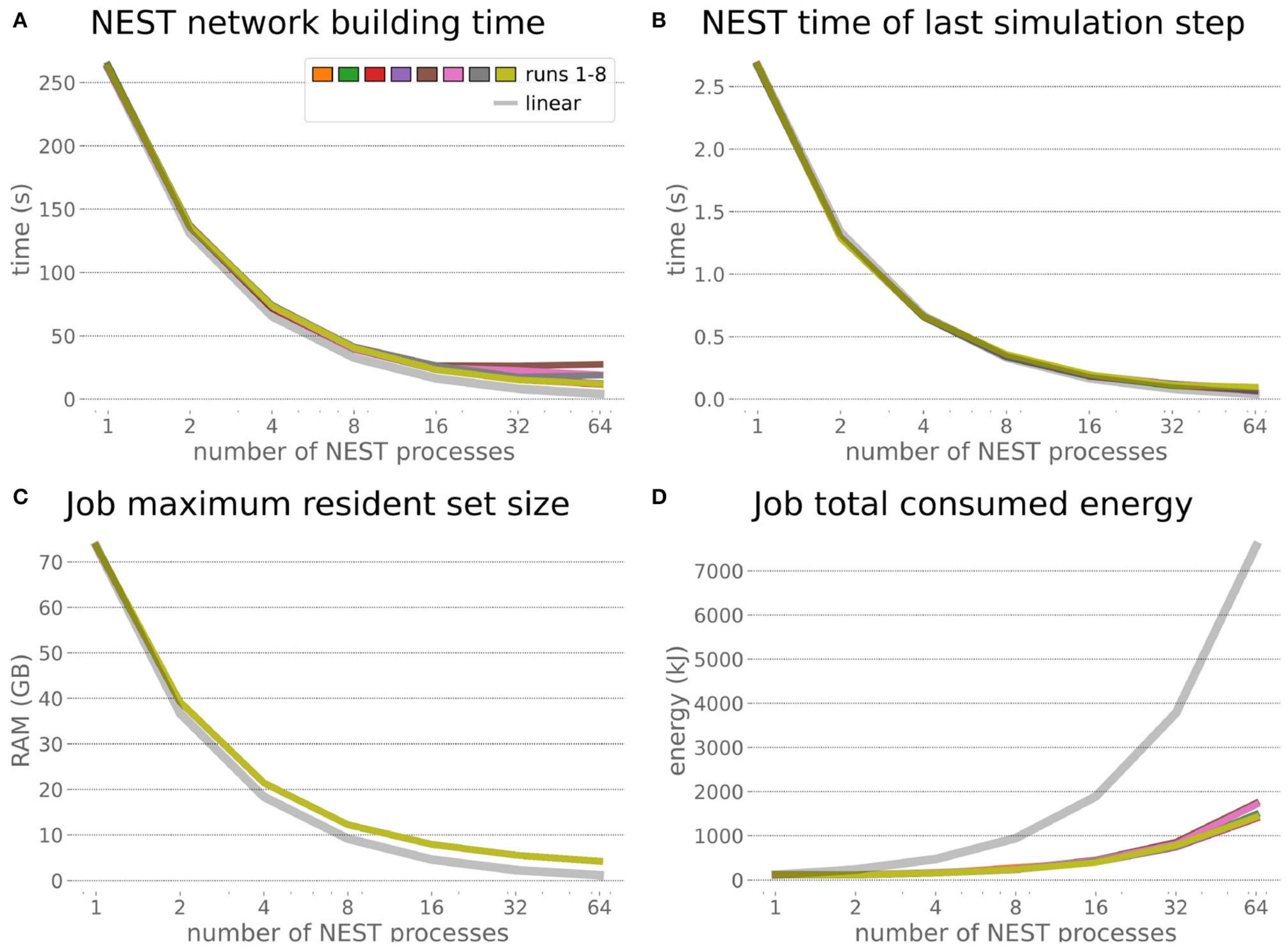


FIGURE 11 | Embodied multi-region rodent brain benchmark NEST times and workload manager characteristics. NEST network building time **(A)** scales up close to linearly, NEST simulation time **(B)** nearly optimally linearly. Building time and simulation time shorten by factors of about 17 and 30, respectively. The required memory per node **(C)** for running the experiment scales close to linearly, but the total amount of consumed energy **(D)** only increases by a factor of about 12.

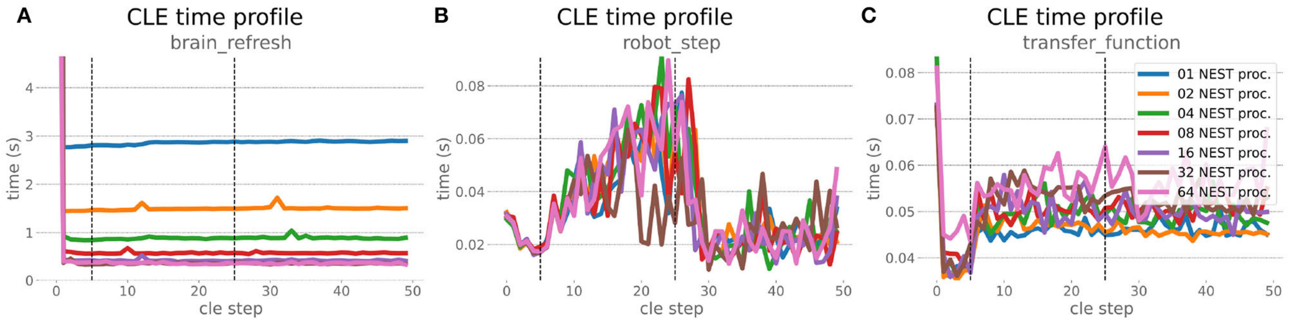


FIGURE 12 | Embodied multi-region rodent brain benchmark CLE profiler times. Experiment execution has an impact on step times, setting the spike ratio of a motor cortex layer at step 5, and feeding the brain at 25 steps has a visible effect in Transfer Function **(C)** and robot **(B)** execution. Transfer Functions execute slightly slower with scaling up the experiment, but is still small compared to the large improvement in the brain execution times **(A)**. The first timestep includes additional initialization procedures and hence takes significantly longer than the usual runtime step time, in the diagrams we clip the y-axis for better visibility of the relevant runtime data.

job handling. This should enable easy transfer of the proposed architecture to other computing sites, in particular those that are part of the FENIX research infrastructure and/or EBRAINS.

The presented setup is fully scalable, as the number of compute nodes involved in the simulation can be user-defined, and as multiple experiments executed on different job allocations can

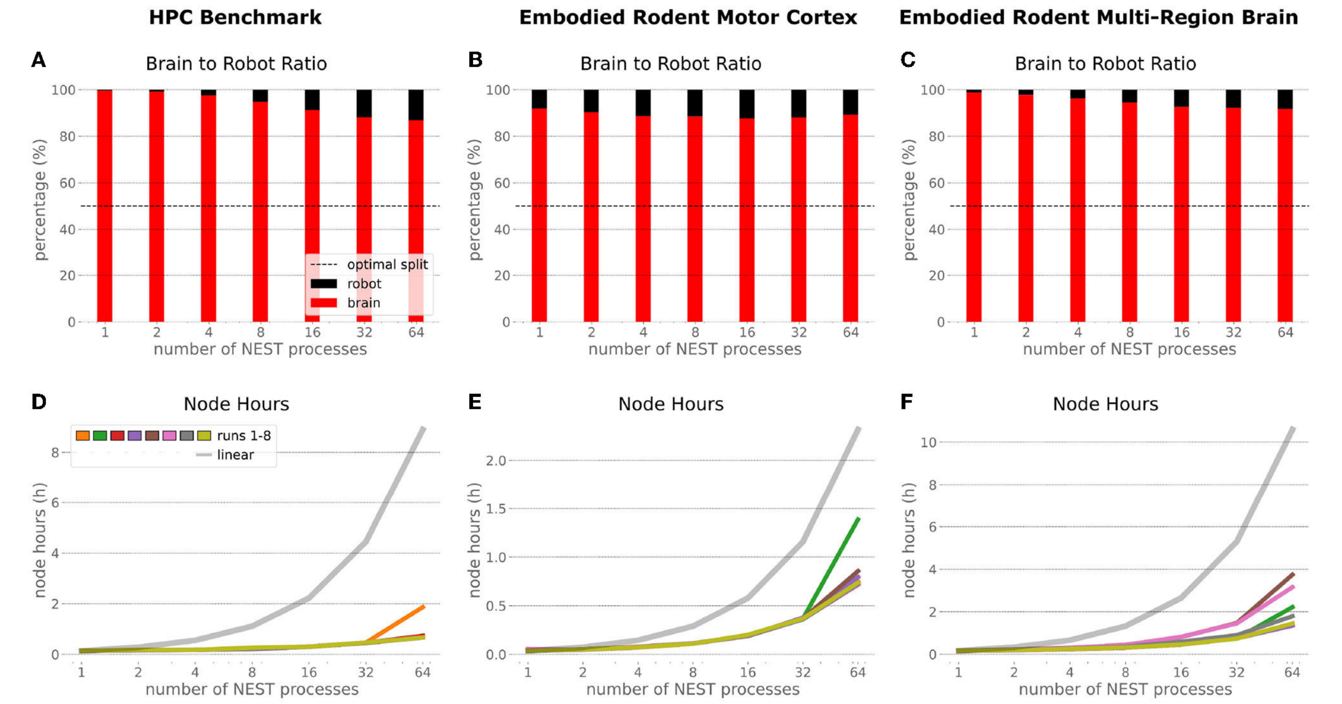


FIGURE 13 | Comparison of benchmark experiments. Brain to Body step time ratio (mean values over all 8 benchmark repetitions) and node hours required to run the simulation for the HPC Benchmark, embodied rodent brain experiment with Motor cortex only and full embodied multi-region rodent brain experiment. For all experiments the brain execution takes longer than the robot simulation step **(A–C)**. This imbalance can be improved with our distributed architecture in particular for large neural networks as with the HPC Benchmark **(A)** and full embodied multi-region rodent brain experiment **(C)**. The required amount of node hours to run the experiments does not scale up linearly, it increases exponentially with small slope only. The embodied multi-region rodent brain experiment **(F)** requires the most node hours, but the required node hours increase by factors about less than 14, 35 and 23 for the HPC Benchmark **(D)**, embodied rodent brain experiment with only Motor cortex **(E)** and full embodied multi-region rodent brain experiment **(F)**, respectively.

be launched simultaneously via the same front-end. Experiments run interactively, meaning that the user can join the simulations at any time via the web-based front-end, interact with the virtual agent and environment, or change the configuration of e.g., brain parameters, transfer functions and robot control.

We demonstrated the potential of our setup with two benchmark experiments scaled up from 2 to 33 compute nodes (1 to 64 NEST processes) using a balanced brain benchmark simulation and a multi-region embodied rodent brain model. We were able to speed up the total experiment execution time for the HPC Benchmark with balanced networks by a factor of up to 12, and for the RoboBrain experiment by a factor of about 6, thus demonstrating the potential benefits of distributing a brain simulation over multiple nodes, especially as it gets larger. Furthermore, the real-time factor could be improved, particularly for the benchmark based on balanced networks. It saturated with more than 32 nodes, however, potentially indicating that scaling-up is not always beneficial in cases where the overhead required for communication with all compute nodes at every simulation step becomes significant in relation to the compute load on each individual node. Nevertheless, the improvements we could demonstrate with distribution in terms of real-time factor lay the foundation for large-scale experiments that could otherwise not be carried out interactively due to their slow execution.

With both benchmark experiments we also demonstrated that NEST scales linearly, or near-linearly when parallelizing across 1–64 processes in terms of network building and simulation time. Regarding the cost of distribution for the benchmark experiments, we found that both energy consumed and compute node hours required scale sub-linearly and hence provide a strong argument for distributed simulations. The parallelization of the brain simulation accounts for better usage of computation time in our examples, as both brain and robot simulation are executed in parallel at every simulation step in the NRP, and thus can be better aligned with each other since the brain simulation is consistently the limiting factor. This ratio may even improve for a more complex rodent model physics simulation with more muscle actuators.

When NEST is run in a stand-alone fashion, it shows excellent scaling (Kunkel et al., 2014; Jordan et al., 2018) and is even able to achieve sub-realtime performance for certain models (Kurth et al., 2022). There are several reasons why the scaling is not at this level for the use-case presented in this article. First, due to the synchronization between network and physics simulation, NEST is executed in steps of 20 ms in the NRP and such stepped simulations are inherently more expensive due to the increased function call overhead and the fact that data-structures have to be paged in and out much more frequently rather than operating on them in a more continuous way. Second, both simulators are executed in parallel, but the data exchange still needs to be executed sequentially, which adds to the raw neural network simulation times. Third, we chose a REST-based communication interface between the NRP and NEST Server for the first version of the interface presented here, since it is functionally complete and has successfully been used in other contexts. This communication via text-based data representations (JSON over HTTP) is obviously inefficient compared to lower level protocols such as Google's Protocol Buffers[7] or Cap'n Proto[8]. We are aware of this restrictions and already working on moving to more optimized communication methods with higher bandwidth and lower latency (e.g., Insite framework). It is worth noting here, that the current setup will support any future NEST improvements transparently, as long as these do not change the NEST Server API.

Overall, we approached saturation when scaling up to about 64 NEST processes. For the larger embodied multi-region rodent brain experiment, this saturation was visible with 32 processes in terms of both runtime and real-time factors. With both benchmark experiments we demonstrated that a scale-up to about 8 nodes could bring a significant performance improvement in terms of initialization and runtime of experiments, at the cost of only few additional node hours and concomitant energy consumption. With more compute nodes, additional improvements were possible, albeit less significantly and at a slightly higher cost. We proved the repeatability of our results by executing every benchmark experiment 8 times. Even though we ran our benchmark experiments with only 1s simulation time in order to save energy, we think it is safe to assume that our results will scale, as we showed relatively stable simulation execution step sizes in the CLE profiler data.

The setup we presented here is intrinsically highly scalable, insofar as the number of compute nodes can be passed as a parameter and can be much larger than the 33 compute nodes used for the presented benchmark experiments. While we simulated a multi-region brain model consisting of about one million neurons in these benchmark experiments, a biological mouse brain is assumed to have around 70 million neurons, and therefore another scale-up by a factor of 70 would be needed to simulate such a brain at the naturalistic scale. We are currently not aware of any embodied brain simulation model with larger scale that is implemented with the given software tools and that we could have used for our benchmarks, but such models are clearly part of future work. While the benchmarks presented saturate in terms of performance with about 32 or 64 compute nodes, it has been demonstrated that NEST scales well above that with a larger number of CPU cores (Kunkel et al., 2014; Kurth et al., 2022). Knowing that there are 1813 available multicore compute nodes on the Piz Daint supercomputer, we could approach this simulation scale with our current setup with just a parameter change—and a good budget. The Piz Daint supercomputer also provides GPU compute nodes that are well known for efficient parallel computing. However, Kurth et al. (2022) show that NEST distributed on CPU cores is faster and more energy efficient than any neuromorphic and GPU based simulation known to us.

A wide variety of experiments are supported with our setup, as it easily enables one to add additional muscles for the rodent model (e.g., a freely running mouse with additional muscles), use a different musculoskeletal model altogether (Human, monkey) or use NEST-based spiking neural networks to control a robotic system. In particular, we posit that integration of a detailed model of spinal cord circuitry with the whole-brain model presented herein would be highly relevant in order to investigate *in silico* experiments related to motor control, neurotechnology and neurorehabilitation. The proposed setup is therefore extremely versatile and can support research efforts in multiple high-impact fields, such as neuroscience, robotics and neuromorphic computing.

More generally, the present work lays the foundation to address the scientific dimension of large-scale brain simulation in addition to its technical one. The scientific investigation and validation of the dynamics emerging from the interaction of several types of neurons is indeed critical, as well as the optimizations of the high-degree-of-freedom parameter space of network models. Biological constraints were incorporated in the different regions of the CBCT model; however, once interconnected, the model as a whole requires a proper framework for systematic simulation with additional naturalistic constraints or boundary conditions, i.e., a body, for relevant experimentation on cognitive and motor functions. The reference model size defined herein in relation to the $1 \times 1\text{mm}^2$ cortical patch provides an initial setup for starting such validation process. However, the ultimate goal is the simulation of the full-brain network. Previously, large-scale simulations of the CBCT model were performed on the decommissioned K computer (Miyazaki et al., 2012) using NEST 2, reaching a network size of $7 \times 7\text{mm}^2$; thus, 51 million neurons, more than a single hemisphere of the mouse brain (Gutierrez et al., 2020). The new NEST 3 (de Schepper et al., 2022), NRP, EBRAINS HPC infrastructure, as well as the Fugaku supercomputer (Sato et al., 2020), provide a promising new horizon for 1:1 scale simulations.

In summary, we introduced a versatile NRP-based setup that supports embodied large-scale brain simulations. It can accommodate spiking neural networks implemented in NEST and connected to customizable musculoskeletal systems or robotic agents. We tested it with several models of spiking neural networks, including a highly complex multi-area brain model, thus demonstrating the capacity of this setup for *in silico* closed-loop neuroscience at scale. Importantly, it leverages the HPC capabilities of a supercomputer while supporting online interactivity with the ongoing simulations. With this setup, we thus lay the foundations toward the democratization of *in silico* behavioral experiments with large-scale multi-area brain models. Indeed, the *raison d'être* of this work is to remove some of the main entry barriers that prevent computational neuroscientists or neuromorphic engineers from testing the functional capabilities of their models through embodied simulations, and make it as easy as possible for them to leverage HPC infrastructures without being a power

[7] https://developers.google.com/protocol-buffers

[8] https://capnproto.org/

user thereof. In order to achieve this vision, the upcoming development efforts will focus on integrating the setup fully into the EBRAINS research infrastructure, especially in terms of federated user resource management and the creation of a dedicated service account. With this, it is our hope that this work will not be yet another attempt at simulating the brain, but a blueprint that can be reused by many, and an enabling technology for the concept of embodiment to gain traction in the neuroscience community.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found below: https://github.com/HBPNeurorobotics/nestserver_benchmarks.

AUTHOR CONTRIBUTIONS

BF and FM conceptualized the benchmark study and orchestrated the implementation. BF, CB, UA, ER, VV, VZ, AK, and FM implemented necessary features in the Neurorobotics Platform. CB, AU, FC, and CM supported the implementation and execution on HPC nodes of Piz Daint and virtual machines on Castor. JE, CJ-R, WK, and AM implemented the parallelization of NEST on multiple cluster nodes. JE and CB implemented the client-server interface between the NRP and NEST. CG, ZS, HY, MH, JI, TY, and KD implemented the multi-region rodent brain model. BF integrated the multi-region rodent brain model into the NRP, adapted rodent and environment model and implemented the brain to body interconnection. BF implemented the NRP frontend GUI to launch the NRP on Piz Daint compute nodes. BF, JE, and CJ-R executed the benchmark experiments. Everyone contributed to analyzing the benchmark data and writing the paper. All authors contributed to the article and approved the submitted version.

FUNDING

This work received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements No. 785907 (Human Brain Project SGA2) and no. 945539 (Human Brain Project SGA3). We acknowledge the use of Fenix Infrastructure Resources, which are partially funded from the European Union's Horizon 2020 research and innovation programme through the ICEI project under the grant agreement no. 800858. This project was supported by MEXT as Program for Promoting Researches on the Supercomputer Fugaku (hp200139, hp210169). Part of this study was supported by MEXT KAKENHI grant no. 17H06310.

ACKNOWLEDGMENTS

The authors like to thank Samuel Omlin (NRP), Massimo Benini (OpenStack), and Alberto Madonna (Sarus) for their support implementing the architecture and running experiments for this study.

REFERENCES

Abi Akar, N., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor–a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia), 274–282.

Allegra Mascaro, A. L., Falotico, E., Petkoski, S., Pasquini, M., Vannucci, L., Tort-Colet, N., et al. (2020) Experimental and computational study on motor control and recovery after stroke: toward a constructive loop between experimental and virtual embodied neuroscience. *Front. Syst. Neurosci.* 14:31. doi: 10.3389/fnsys.2020.00031

Angelidis, E., Buchholz, E., Arreguit, J., Rougé, A., Stewart, T., von Arnim, A., et al. (2021). A spiking central pattern generator for the control of a simulated lamprey robot running on SpiNNaker and Loihi neuromorphic boards. *Neuromorphic Comput. Eng.* 1, 014005. doi: 10.1088/2634-4386/ac1b76

Apicella, A. J., Wickersham, I. R., Seung, H. S., and Shepherd, G. M. (2012). Laminarly orthogonal excitation of fast-spiking and low-threshold-spiking interneurons in mouse motor cortex. *J. Neurosci.* 32, 7021–7033. doi: 10.1523/JNEUROSCI.0011-12.2012

Apps, R., and Garwicz, M. (2005). Anatomical and physiological foundations of cerebellar information processing. *Nat. Rev. Neurosci.* 6, 297–311. doi: 10.1038/nrn1646

Avermann, M., Tomm, C., Mateo, C., Gerstner, W., and Petersen, C. C. (2012). Microcircuits of excitatory and inhibitory neurons in layer 2/3 of mouse barrel cortex. *J. Neurophysiol.* 107, 3116–3134. doi: 10.1152/jn.00917.2011

Awile, O., Kumbhar, P., Cornu, N., Dura-Bernal, S., King, J. G., Lupton, O., et al. (2022). Modernizing the neuron simulator for sustainability, portability, and performance. *bioRxiv [preprint]*. doi: 10.1101/2022.03.03.482816

Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., et al. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *J. Compar. Neurol.* 513, 532–541. doi: 10.1002/cne.21974

Bahuguna, J., Weidel, P., and Morrison, A. (2019). Exploring the role of striatal D1 and D2 medium spiny neurons in action selection using a virtual robotic framework. *Eur. J. Neurosci.* 49, 737–753. doi: 10.1111/ejn.14021

Barmack, N. H., and Yakhnitsa, V. (2008). Functions of interneurons in mouse cerebellum. *J. Neurosci.* 28, 1140–1152. doi: 10.1523/JNEUROSCI.3942-07.2008

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T., Rasmussen, D., et al. (2014). Nengo: a Python tool for building large-scale functional brain models. *Front. Neuroinform.* 7, 48. doi: 10.3389/fninf.2013.00048

Benedičič, L., Cruz, F., Madonna, A., and Mariotti, K. (2019). "Sarus: highly scalable docker containers for hpc systems," in *International Conference on High Performance Computing* (Cham), 46–68.

Billeh, Y. N., Cai, B., Gratiy, S. L., Dai, K., Iyer, R., Gouwens, N. W., et al. (2020). Systematic integration of structural and functional data into multi-scale models of mouse primary visual cortex. *Neuron* 106, 388–403. doi: 10.1016/j.neuron.2020.01.040

Bower, J. M., and Beeman, D. (2007). Constructing realistic neural simulations with genesis. *Neuroinformatics* 1401, 03–125. doi: 10.1007/978-1-59745-520-6_7

Brocke, E. (2020). *Method Development for Co-Simulation of Electrical-Chemical Systems in Neuroscience* (Ph.D. thesis). KTH Royal Institute of Technology.

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comp. Neurosci.* 8, 183–208. doi: 10.1023/A:1008925309027

Cofer, D., Cymbalyuk, G., Reid, J., Zhu, Y., Heitler, W. J., and Edwards, D. H. (2010). Animatlab: a 3d graphics environment for neuromechanical simulations. *J. Neurosci. Methods* 187, 280–288. doi: 10.1016/j.jneumeth.2010.01.005

de Schepper, R., Eppler, J. M., Kurth, A., Nagendra Babu, P., Deepu, R., Spreizer, S., et al. (2022). *NEST 3.2*. Zenodo.

Delp, S. L., Anderson, F. C., Arnold, A. S., Loan, P., Habib, A., John, C. T., et al. (2007). Opensim: open-source software to create and analyze dynamic simulations of movement. *IEEE Trans. Biomed. Eng*. 54, 1940–1950. doi: 10.1109/TBME.2007.901024

DeWolf, T., Stewart, T. C., Slotine, J.-J., and Eliasmith, C. (2016). A spiking neural model of adaptive arm control. *Proc. R. Soc. B Biol. Sci*. 283, 20162134. doi: 10.1098/rspb.2016.2134

Djurfeldt, M., Hjorth, J., Eppler, J. M., Dudani, N., Helias, M., Potjans, T. C., et al. (2010). Run-time interoperability between neuronal network simulators based on the music framework. *Neuroinformatics* 8, 43–60. doi: 10.1007/s12021-010-9064-z

Eccles, J. C. (1967). Circuits in the cerebellar control of movement. *Proc. Natl. Acad. Sci. U.S.A*. 58, 336. doi: 10.1073/pnas.58.1.336

Eliasmith, C., and Anderson, C. (2004). *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. Cambridge, MA: MIT Press.

Fernándes, J. P., Vargas, M. A., García, J. M., Carrillo, J. A., and Aguilar, J. J. (2021). A biological-like controller using improved spiking neural networks. *Neurocomputing* 463, 237–250. doi: 10.1016/j.neucom.2021.08.005

Ferrario, A., Palyanov, A., Koutsikou, S., Li, W., Soffe, S., Roberts, A., et al. (2021). From decision to action: Detailed modelling of frog tadpoles reveals neuronal mechanisms of decision-making and reproduces unpredictable swimming movements in response to sensory signals. *PLoS Comput. Biol*. 17, e1009654. doi: 10.1371/journal.pcbi.1009654

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The SpiNNaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Gewaltig, M.-O., and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia* 2, 1430. doi: 10.4249/scholarpedia.1430

Gilra, A., and Gerstner, W. (2018). "Non-linear motor control by local learning in spiking neural networks," in *Proceedings of the 35th International Conference on Machine Learning-PMLR* (Stockholm), 1773–1782.

Girard, B., Lienard, J., Gutierrez, C. E., Delord, B., and Doya, K. (2020). A biologically constrained spiking neural network model of the primate basal ganglia with overlapping pathways exhibits action selection. *Eur. J. Neurosci*. 53, 2254–2277. doi: 10.1111/ejn.14869

Gleeson, P., Cantarelli, M., Marin, B., Quintana, A., Earnshaw, M., Sadeh, S., et al. (2019). Open source brain: a collaborative resource for visualizing, analyzing, simulating, and developing standardized models of neurons and circuits. *Neuron* 103, 395–411. doi: 10.1016/j.neuron.2019.05.019

Gutierrez, C. E., Zhe, S., Yamaura, H., Heidarinejad, M., Igarashi, J., Yamazaki, T., et al. (2020). "Simulation of resting-state neural activity in a loop circuit of the cerebral cortex, basal ganglia, cerebellum, and thalamus using NEST simulator," in *Proceedings of the Annual Conference of the Japanese Neural Network Society, Vol*. 30, 63–65.

Hahne, J., Diaz, S., Patronis, A., Schenck, W., Peyser, A., Graber, S., et al. (2021). *NEST 3.0*. Zenodo.

Harvey, R., and Napper, R. (1991). Quantitatives studies on the mammalian cerebellum. *Progress Neurobiol*. 36, 437–463. doi: 10.1016/0301-0082(91)90012-P

Heckroth, J. A. (1994). Quantitative morphological analysis of the cerebellar nuclei in normal and lurcher mutant mice. i. morphology and cell number. *J. Compar. Neurol*. 343, 173–182. doi: 10.1002/cne.903430113

Helias, M., Kunkel, S., Masumoto, G., Igarashi, J., Eppler, J., Ishii, S., et al. (2012). Supercomputers ready for use as discovery machines for neuroscience. *Front. Neuroinform*. 6, 26. doi: 10.3389/fninf.2012.00026

Herculano-Houzel, S. (2009). The human brain in numbers: a linearly scaled-up primate brain. *Front. Hum. Neurosc*. 31, 2009. doi: 10.3389/neuro.09.031.2009

Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput*. 9, 1179–1209. doi: 10.1162/neco.1997.9.6.1179

Igarashi, J., Yamaura, H., and Yamazaki, T. (2019). Large-scale simulation of a layered cortical sheet of spiking network model using a tile partitioning method. *Front. Neuroinform*. 13, 71. doi: 10.3389/fninf.2019.00071

Ito, M., and Itō, M. (1984). *The Cerebellum and Neural Control*. Raven Press.

Jiang, X., Shen, S., Cadwell, C. R., Berens, P., Sinz, F., Ecker, A. S., et al. (2015). Principles of connectivity among morphologically defined cell types in adult neocortex. *Science* 350, aac9462. doi: 10.1126/science.aac9462

Jiang, X., Wang, G., Lee, A. J., Stornetta, R. L., and Zhu, J. J. (2013). The organization of two new cortical interneuronal circuits. *Nat. Neurosci*. 16, 210–218. doi: 10.1038/nn.3305

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Front. Neuroinform*. 12, 2. doi: 10.3389/fninf.2018.00002

Jordan, J., Weidel, P., and Morrison, A. (2019). A closed-loop toolchain for neural network simulations of learning autonomous agents. *Front. Comput. Neurosci*. 13, 46. doi: 10.3389/fncom.2019.00046

Kalidindi, H. T., Cross, K. P., Lillicrap, T. P., Omrani, M., Falotico, E., Sabes, P. N., et al. (2021). Rotational dynamics in motor cortex are consistent with a feedback controller. *Elife* 10, e67256. doi: 10.7554/eLife.67256

Kätzel, D., Zemelman, B. V., Buetfering, C., Wölfel, M., and Miesenböck, G. (2011). The columnar and laminar organization of inhibitory connections to neocortical excitatory cells. *Nat. Neurosci*. 14, 100–107. doi: 10.1038/nn.2687

Knoll, A., Gewaltig, M.-O., Sanders, J., and Oberst, J. (2016). Neurorobotics: a strategic pillar of the human brain project. *Sci. Robot*. 25–35. Available online at: http://archive.www6.in.tum.de/www6/Main/Publications/knollNeuro2016.pdf

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., et al. (2019). Coreneuron: an optimized compute engine for the neuron simulator. *Front. Neuroinform*. 13, 63. doi: 10.3389/fninf.2019.00063

Kunkel, S., Schmidt, M., Eppler, J. M., Plesser, H. E., Masumoto, G., Igarashi, J., et al. (2014). Spiking network simulation code for petascale computers. *Front. Neuroinform*. 8, 78. doi: 10.3389/fninf.2014.00078

Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2022). Sub-realtime simulation of a neuronal network of natural density. *Neuromorph. Comput. Eng*. 2, 021001. doi: 10.1088/2634-4386/ac55fc

Lange, W. (1975). Cell number and cell density in the cerebellar cortex of man and some other mammals. *Cell Tissue Res*. 157, 115–124. doi: 10.1007/BF00223234

Lee, A. J., Wang, G., Jiang, X., Johnson, S. M., Hoang, E. T., Lanté, F., et al. (2015). Canonical organization of layer 1 neuron-led cortical inhibitory and disinhibitory interneuronal circuits. *Cerebral cortex* 25, 2114–2126. doi: 10.1093/cercor/bhu020

Lefort, S., Tomm, C., Sarria, J.-C. F., and Petersen, C. C. (2009). The excitatory neuronal network of the c2 barrel column in mouse primary somatosensory cortex. *Neuron* 61, 301–316. doi: 10.1016/j.neuron.2008.12.020

Lev, D. L., and White, E. L. (1997). Organization of pyramidal cell apical dendrites and composition of dendritic clusters in the mouse: emphasis on primary motor cortex. *Eur. J. Neurosci*. 9, 280–290. doi: 10.1111/j.1460-9568.1997.tb01398.x

Liénard, J., and Girard, B. (2014). A biologically constrained model of the whole basal ganglia addressing the paradoxes of connections and selection. *J. Comput. Neurosci*. 36, 445–468. doi: 10.1007/s10827-013-0476-2

Markram, H., Muller, E., Ramaswamy, S., Reimann, M., Abdellah, M., Sanchez, C., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

Mathis, M. W., Mathis, A., and Uchida, N. (2017). Somatosensory cortex plays an essential role in forelimb motor adaptation in mice. *Neuron* 93, 1493–1503. doi: 10.1016/j.neuron.2017.02.049

Millard, M., Uchida, T., Seth, A., and Delp, S. L. (2013). Flexing computational muscle: modeling and simulation of musculotendon dynamics. *J. Biomech. Eng*. 135, 021005. doi: 10.1115/1.4023390

Miyazaki, H., Kusano, Y., Shinjou, N., Shoji, F., Yokokawa, M., and Watanabe, T. (2012). Overview of the k computer system. *Fujitsu Sci. Tech. J*. 48, 255–265. doi: 10.1016/j.procs.2014.05.052

Morrison, A., Aertsen, A., and Diesmann, M. (2007). Spike-timing-dependent plasticity in balanced random networks. *Neural Comput*. 19, 1437–1467. doi: 10.1162/neco.2007.19.6.1437

Morteza, H., and Sun Zhe, J. I. (2019). "Hierarchy of inhibitory circuit acts as a switch key for network function in a model of the primary motor cortex," in *28th Annual Computational Neuroscience Meeting: CNS*2019 Meeting Abstracts* (Barcelona), 132.

Pala, A., and Petersen, C. C. (2015). In vivo measurement of cell-type-specific synaptic connectivity and synaptic transmission in layer 2/3 mouse barrel cortex. *Neuron* 85, 68–75. doi: 10.1016/j.neuron.2014.11.025

Pfeffer, C. K., Xue, M., He, M., Huang, Z. J., and Scanziani, M. (2013). Inhibition of inhibition in visual cortex: the logic of connections between molecularly distinct interneurons. *Nat. Neurosci*. 16, 1068–1076. doi: 10.1038/nn.3446

Ramalingasetty, S. T., Danner, S. M., Arreguit, J., Markin, S. N., Rodarie, D., Kathe, C., et al. (2021). A whole-body musculoskeletal model of the mouse. *IEEE Access*. 9, 163861–163881. doi: 10.1109/ACCESS.2021.3133078

Sarma, G. P., Lee, C. W., Portegys, T., Ghayoomie, V., Jacobs, T., Alicea, B., et al. (2018). Openworm: overview and recent advances in integrative biological simulation of caenorhabditis elegans. *Philos. Trans. R. Soc. B* 373, 20170382. doi: 10.1098/rstb.2017.0382

Sato, M., Ishikawa, Y., Tomita, H., Kodama, Y., Odajima, T., Tsuji, M., et al. (2020). "Co-design for a64fx manycore processor and "fugaku," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, GA: IEEE), 1–15.

Senk, J., Hagen, E., van Albada, S. J., and Diesmann, M. (2018). Reconciliation of weak pairwise spike-train correlations and highly coherent local field potentials across space. *arXiv[Preprint].arXiv:1805.10235*. doi: 10.48550/arXiv.1805.10235

Shepherd, G. M. (2013). Corticostriatal connectivity and its role in disease. *Nat. Rev. Neurosci*. 14, 278–291. doi: 10.1038/nrn3469

Song, S., Sjöström, P. J., Reigl, M., Nelson, S., and Chklovskii, D. B. (2005). Highly nonrandom features of synaptic connectivity in local cortical circuits. *PLoS Biol*. 3, e68. doi: 10.1371/journal.pbio.0030068

Sun, Z., and Morteza Heidarinejad, J. I. (2019). "Spatially organized connectivity for signal processing in a model of the rodent primary somatosensory cortex," in *28th Annual Computational Neuroscience Meeting: CNS*2019 Meeting Abstracts* (Barcelona), 133.

Szigeti, B., Gleeson, P., Vella, M., Khayrulin, S., Palyanov, A., Hokanson, J., et al. (2014). Openworm: an open-science approach to modeling caenorhabditis elegans. *Front. Comput. Neurosci*. 8, 137. doi: 10.3389/fncom.2014.00137

Tremblay, R., Lee, S., and Rudy, B. (2016). Gabaergic interneurons in the neocortex: from cellular properties to circuits. *Neuron* 91, 260–292. doi: 10.1016/j.neuron.2016.06.033

Vannucci, L., Pasquini, M., Spalletti, C., Caleo, M., Micera, S., Laschi, C., et al. (2019). "Towards in-silico robotic post-stroke rehabilitation for mice,," in *2019 IEEE International Conference on Cyborg and Bionic Systems (CBS)* (Munich: IEEE), 123–128.

Weiler, N., Wood, L., Yu, J., Solla, S. A., and Shepherd, G. M. (2008). Top-down laminar organization of the excitatory network in motor cortex. *Nat. Neurosci*. 11, 360–366. doi: 10.1038/nn2049

Xu, X., and Callaway, E. M. (2009). Laminar specificity of functional input to distinct types of inhibitory cortical neurons. *J. Neurosci*. 29, 70–85. doi: 10.1523/JNEUROSCI.4104-08.2009

Xue, M., Atallah, B. V., and Scanziani, M. (2014). Equalizing excitation-inhibition ratios across visual cortical neurons. *Nature* 511, 596–600. doi: 10.1038/nature13321

Yamada, Y., Kanazawa, H., Iwasaki, S., Tsukahara, Y., Iwata, O., Yamada, S., et al. (2016). An embodied brain model of the human foetus. *Sci. Rep*. 6, 27893. doi: 10.1038/srep27893

Yamaura, H., Igarashi, J., and Yamazaki, T. (2020). Simulation of a human-scale cerebellar network model on the k computer. *Front. Neuroinform*. 14, 16. doi: 10.3389/fninf.2020.00016

Yoo, A., Jette, M., and Grondona, M. (2003). Slurm: simple linux utility for resource management, job scheduling strategies for parallel processing, volume 2862 of lecture notes in computer science (Seattle, WA).

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

ORIGINAL RESEARCH
published: 23 May 2022
doi: 10.3389/fninf.2022.877945

Auto-Selection of an Optimal Sparse Matrix Format in the Neuro-Simulator ANNarchy

*Helge Ülo Dinkelbach, Badr-Eddine Bouhlal, Julien Vitay and Fred H. Hamker**

Department of Computer Science, Chemnitz University of Technology, Chemnitz, Germany

Modern neuro-simulators provide efficient implementations of simulation kernels on various parallel hardware (multi-core CPUs, distributed CPUs, GPUs), thereby supporting the simulation of increasingly large and complex biologically realistic networks. However, the optimal configuration of the parallel hardware and computational kernels depends on the exact structure of the network to be simulated. For example, the computation time of rate-coded neural networks is generally limited by the available memory bandwidth, and consequently, the organization of the data in memory will strongly influence the performance for different connectivity matrices. We pinpoint the role of sparse matrix formats implemented in the neuro-simulator ANNarchy with respect to computation time. Rather than asking the user to identify the best data structures required for a given network and platform, such a decision could also be carried out by the neuro-simulator. However, it requires heuristics that need to be adapted over time for the available hardware. The present study investigates how machine learning methods can be used to identify appropriate implementations for a specific network. We employ an artificial neural network to develop a predictive model to help the developer select the optimal sparse matrix format. The model is first trained offline using a set of training examples on a particular hardware platform. The learned model can then predict the execution time of different matrix formats and decide on the best option for a specific network. Our experimental results show that using up to 3,000 examples of random network configurations (i.e., different population sizes as well as variable connectivity), our approach effectively selects the appropriate configuration, providing over 93% accuracy in predicting the suitable format on three different NVIDIA devices.

Keywords: neural simulator, rate-coded networks, auto-tuning, code generation, CUDA

OPEN ACCESS

Edited by:
*Thomas Nowotny,
University of Sussex, United Kingdom*

Reviewed by:
*Ankur Sinha,
University College London,
United Kingdom
Mantas Mikaitis,
The University of Manchester,
United Kingdom*

****Correspondence:***
*Fred H. Hamker
fred.hamker@informatik.tu-chemnitz.de*

Received: *17 February 2022*
Accepted: *28 April 2022*
Published: *23 May 2022*

Citation:
Dinkelbach HÜ, Bouhlal B-E, Vitay J and Hamker FH (2022) Auto-Selection of an Optimal Sparse Matrix Format in the Neuro-Simulator ANNarchy. Front. Neuroinform. 16:877945. doi: 10.3389/fninf.2022.877945

1. INTRODUCTION

Models in computational neuroscience are implemented with different degrees of biological detail. Particularly at the systems-level, a significant subset of models incorporate dynamic rate-coded neurons to explain emergent functions of such networks and link them to experimental data. In such networks, neurons are connected to other neurons by axons and synapses, whose joint effect is captured by so-called weights w_{ij} and describes in how far the firing rate x_i of a presynaptic neuron i affects the firing of a post-synaptic neuron j. As outlined by Dinkelbach et al. (2012), the sum of weighted inputs $w_{ij} \cdot x_i$, required to be computed at each time step, is the dominating operation in large-scale rate-coded neural networks, well before other operations such as the numerical integration of ordinary differential equations (ODE). It was shown using a simplified network model that the choice of either a multi-core CPU or a GPU (Graphical Processing Unit) as the computing backend depends on the network's structure. GPU implementations were more efficient on mid- and large-scale networks in comparison to a multi-core CPU implementation. Dinkelbach et al. (2019) observed for a linear rate-coded model that the network had to consist of thousands of neurons in order to utilize a GPU effectively.

When applied on populations of neurons, the weighted sum of synaptic inputs can be computed by a sparse matrix-vector multiplication (SpMV) between a (sparse) matrix **W** and a dense vector $\vec{x}$ which results in a dense vector $\vec{y}$:

$$\vec{y} = \mathbf{W} \times \vec{x}. \tag{1}$$

The SpMV operation, which is a central kernel in many scientific applications, is considered to be memory-bound and is impaired by irregular access patterns to the dense vector $\vec{x}$ (e.g., Temam and Jalby, 1992; Goumas et al., 2008; Williams et al., 2009; Greathouse and Daga, 2014; Langr and Tvrdik, 2016; Filippone et al., 2017). While each non-zero element of **W** is only accessed once in the SpMV operation, there is frequent access to $\vec{x}$ at different positions (e.g., Williams et al., 2009). Depending on the distribution of the non-zeros within a row of the matrix, this can lead to cache misses or re-loads, leading to noticeable performance decreases on CPUs and especially on GPUs (e.g., shown in Dinkelbach et al., 2012). For optimal performance, the number of these scattered accesses should be reduced, for example through a reuse, efficient caching (CPU-oriented architectures) or pre-loading into shared memory (GPU) of the dense vector (e.g., Goumas et al., 2008; Williams et al., 2009; Greathouse and Daga, 2014). To overcome this issue, many different formats were proposed to perform the SpMV operation efficiently on single-core, multi-core CPUs or GPUs (see Langr and Tvrdik, 2016; Filippone et al., 2017 for more details). Nevertheless, understanding the efficiency of applied optimizations can be difficult as the interaction of optimizations with each other or the underlying hardware is hard to predict (see Goumas et al., 2008; Balaprakash et al., 2018 for a detailed discussion). The efficiency of a single optimization may depend on the matrix as well as on the specific platform as demonstrated in the work of Williams et al. (2009). However, the efficiency of an implementation can also change by advancements made by compilers and hardware as pointed out by Steinberger et al. (2016).

Due to the generally unknown sparsity of a matrix, choosing an efficient parallel implementation of the SpMV operation for a given matrix is therefore an important and hard problem (e.g., Liu and Vinter, 2015b; Lehnert et al., 2016; Hou et al., 2017). However, there exists some knowledge about which given format is more suitable for a given matrix. For example, Vázquez et al. (2011) and Sedaghati et al. (2015) suggest that the density of a matrix is a guiding factor for the selection of a particular data structure. Furthermore, as shown by Vázquez et al. (2011), the variability of row lengths can be a relevant criterion in the selection of data formats.

Machine learning methods received increasing attention for the tuning of implementations at various levels, including the selection of code variants, parallelization strategies, or even complete algorithms (see Balaprakash et al., 2018 for a recent review). Modern multi-core CPUs and GPUs in combination with compilers offer a rich possibility for programmers to adapt their code to increase performance. Therefore, the possible search space even for relatively simple operations can reach millions of configurations (e.g., as shown by Datta et al., 2008; Ganapathi et al., 2009 for the stencil operation). Auto-tuning methods considering the SpMV operation were investigated for single-thread, multi-core as well as GPU configurations either using hand-tuning (e.g., Choi et al., 2010), heuristics (e.g., Whaley et al., 2001; Sedaghati et al., 2015), or machine learning methods (e.g., Ganapathi et al., 2009; Benatia et al., 2018; Pichel and Pateiro-Lopez, 2018; Chen et al., 2019). As hardware and algorithms steadily evolve, it is important to integrate auto-tuning principles inside the specific application. Such an integration allows to adjust the build process considering the target platform (Balaprakash et al., 2018).

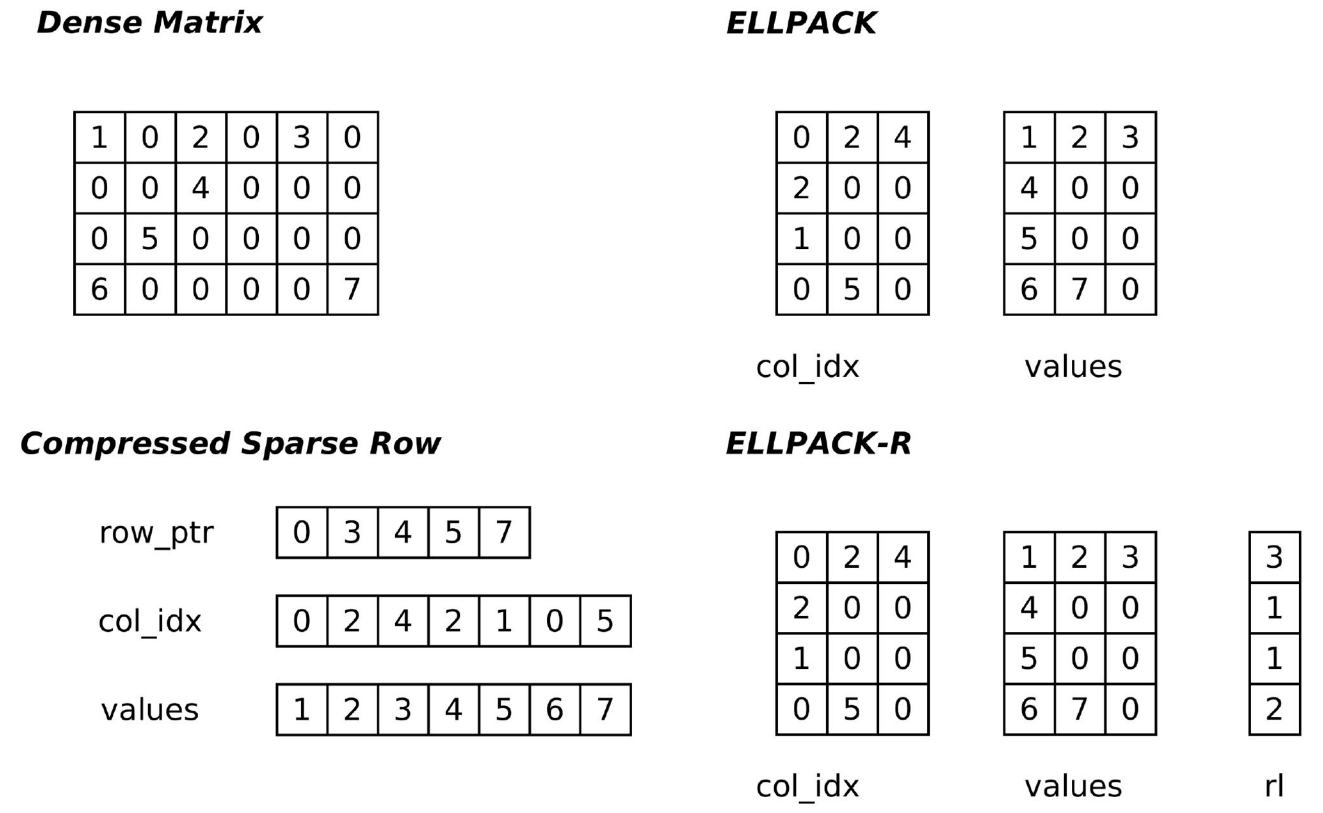
The present article shows that implementing different sparse matrix formats in a neural simulator can improve the overall performance of rate-coded neural networks. We present a two-stage heuristic already embedded in our neural simulation framework ANNarchy (Artificial Neural Networks architect, Vitay et al., 2015). We also demonstrate that the performance can be improved by integrating machine learning methods. This should help developers of neural network models selecting a suitable data structure representation for their specific network.

2. RELATED WORK

2.1. Sparse Matrix Formats for SpMV

As outlined in the introduction, the SpMV operation has been thoroughly investigated and several sparse matrix formats have been proposed. The following collection of formats is just a short overview and by no means exhaustive. For more details, refer to the reviews of Langr and Tvrdik (2016) and Filippone et al. (2017).

Probably the most common and well-known format is the compressed sparse row (or Yale) format (CSR). The non-zeros of each row are stored in two arrays (one for the column indices and the other one for the values). The start and stop indices of a row are stored in a row pointer array. The ELLPACK/ITPACK format (Kincaid et al., 1989) was intended to be efficient for vector processors. This format decomposes the non-zeros into two dense matrices whose dimensions are number of rows times the maximum number of non-zeros within a row, one matrix representing the column indices, the other the values. If the matrix has heterogeneous row lengths, non-existing entries need to be marked by a neutral element, which likely creates a large



Dense Matrix

1	0	2	0	3	0
0	0	4	0	0	0
0	5	0	0	0	0
6	0	0	0	0	7

ELLPACK

col_idx			values		
0	2	4	1	2	3
2	0	0	4	0	0
1	0	0	5	0	0
0	5	0	6	7	0

Compressed Sparse Row

row_ptr	0	3	4	5	7		
col_idx	0	2	4	2	1	0	5
values	1	2	3	4	5	6	7

ELLPACK-R

col_idx			values			rl
0	2	4	1	2	3	3
2	0	0	4	0	0	1
1	0	0	5	0	0	1
0	5	0	6	7	0	2

FIGURE 1 | Schematic representation of the compressed sparse row (CSR), ELLPACK, and ELLPACK-R formats derived from a dense matrix. The CSR format comprises three dense vectors: a *row_ptr* array where the begin and end of subsequent rows are encoded. These indices are needed to select the correct values from the column index and value array. Contrary to CSR, in ELLPACK/ELLPACK-R the column indices and the values are encoded in dense matrices. The ELLPACK-R has an additional row-length (rl) array to encode the row lengths to spare the index checking.

memory overhead. This format is considered as GPU-friendly if the dense matrices are stored in column-major[1] order (Bell and Garland, 2009; Vázquez et al., 2011). Vázquez et al. (2011) proposed an extended version, ELLPACK-R, which introduces an additional row-length array to encode varying row lengths instead of checking each matrix entry with an if-clause. An overview of the different sparse matrix formats is depicted in **Figure 1**.

2.2. ANNarchy

The ANNarchy neural simulator is written in Python and intended for the simulation of biologically detailed neural networks. The equation-based interface of ANNarchy allows a flexible and easy definition of the neuron and synapse models (Vitay et al., 2015). Using an automatic code generation approach, the model description is transformed into C++ code allowing the use of parallel programming frameworks such as OpenMP for multi-core CPUs or CUDA for GPUs for the efficient implementation of rate-coded and spiking models (Vitay et al., 2015; Dinkelbach et al., 2019).

The current version 4.7.1.1 of ANNarchy provides several sparse matrix formats for the computation of rate-coded neural network models. In addition to the already existing list-in-list/compressed sparse row implementation (as described in Dinkelbach et al., 2012), an ELLPACK/ITPACK (Kincaid et al., 1989; Vázquez et al., 2011) and a dense matrix format have been added, which will be evaluated in Section 4. ANNarchy also implements a Hybrid format as described by Bell and Garland (2009) and a blocked sparse row (BSR) format as described by Verschoor and Jalba (2012) and Eberhardt and Hoemmen (2016), but preliminary tests have shown that those formats are not performing well in comparison to the others on the dataset used in this work, so they are omitted for the present article. We hypothesize that the structure of the matrices in our dataset, i.e., a relatively homogeneous row length (for Hybrid) and a high scattering across the matrix (for BSR), are limiting factors for these data formats.

Further, we extended our code generation approach to allow auto-vectorization (using compiler hints e.g., `#pragma simd`) for the continuous neural and synaptic state updates by

[1] This means that the data of a column is stored continuously in memory instead of storing a row continuously (which is referred to as row-major).

reordering the code to reduce the number of branches. We introduce for continuous transmission an implementation using AVX-512, AVX and SSE4.2 instructions[2] to address most of the currently available CPU architectures.

2.3. Auto-Tuning Methods

As outlined by Balaprakash et al. (2018), auto-tuning in high-performance computing is utilized at various levels within an application. Many of these works/ideas are conjuncted with highly optimized libraries like ATLAS[3] (Whaley et al., 2001), SPARSITY (Im et al., 2004), or OSKI (Vuduc et al., 2005). These frameworks are often not limited to the SpMV operation but implement a set of operations from the basic linear algebra (BLAS) routines. This is in contrast to optimized libraries such as clSpMV (Su and Keutzer, 2012) or SMAT (Li et al., 2013) which only focus on the SpMV operation. From our perspective, there are two types of approaches that are of special interest.

First, hand-tuning of a specific format is probably the most common approach, where data structures are adapted to the algorithm or processed data. Some examples are the CSR-like (Hou et al., 2017), ELLR-T (Vázquez et al., 2012), BCSR (Choi et al., 2010), BELLPACK (Choi et al., 2010), and sliced ELLPACK Kreutzer et al. (2014) data structures. Especially for GPUs arise the question of load balancing, i.e., how many threads should be used and how many blocks should be used for computation at the same time. The effect of the block size can already vary noticeably on a single example as demonstrated by Eberhardt and Hoemmen (2016). The performance was most consistent on a Sandy Bridge CPU in comparison to a GPU and a Xeon Phi. Guo and Wang (2010) proposed a model-driven approach for the fine-tuning of the blocked CSR and blocked ELLPACK format to tackle this issue.

The second class of approaches is the selection of a suitable format for a given matrix, as investigated by Li et al. (2013), Greathouse and Daga (2014), Sedaghati et al. (2015), or Benatia et al. (2018). The main idea is to derive the decision based on a set of features. The mapping of features into a decision can be based on either heuristics or machine learning methods. For instance, Lehnert et al. (2016) have shown that performance prediction using machine learning methods can outperform explicit performance models. The predicted computation time is then used to derive the matrix format decision. In the present manuscript we will follow the second class of approaches, more precisely the work of Lehnert et al. (2016) and Benatia et al. (2018), using regression techniques to predict the performance of a sparse matrix format applied on a given matrix.

3. METHODS

Our focus is to develop an efficient tool that can predict with high accuracy the suitable format for each connectivity matrix of a specific neural network. In the following, we propose two methods for matrix format selection: The first is based on a simple heuristic (Section 3.1) and the second uses a machine learning model (Section 3.2) for predicting the appropriate format.

3.1. Two-Stage Heuristic for Format Selection on GPUs

We followed the idea of Sedaghati et al. (2015), who analyzed the obtained GFLOPS (floating operations per second, see Section 4 for a more detailed description) on several matrices for potential correlations. In their work, they showed that a quite good heuristic can be based on the fraction of non-zeros. We are going to compare three available implementations: the CSR format using an updated version of the algorithm presented in Dinkelbach et al. (2012), the ELLPACK-R presented in Vázquez et al. (2011) as well as a dense matrix representation.

There are several factors influencing the performance achieved with a given implementation on GPUs. One crucial fact is to ensure coalesced memory access toward accessed data (e.g., Bell and Garland, 2009; Dinkelbach et al., 2012; Yavuz et al., 2016). A memory access is considered as coalesced if all threads within a half-warp[4] can use the data loaded from a 32-, 64-, or 128-byte segment (Bell and Garland, 2009). One key difference between the implementations of the SpMV using CSR and ELLPACK-R is that they are parallelized over different dimensions: while our CSR implementation computes one row per warp, a warp in ELLPACK-R computes a set of rows at the same time.

Considering these different computation patterns and the fact that a dense matrix is efficient for densely packed matrices, one can obtain a simple decision tree as depicted in **Figure 2**. The decision is two-fold: first we decide based on the matrix density, i.e., the ratio of nonzeros to the total number of elements in the matrix, whether the density is greater than a threshold. The matrix is considered as dense in this case. Otherwise the average number of non-zeros in a row (avgnzr) is considered. If this value is lower or equal to 128, the ELLPACK-R format is selected, otherwise CSR is chosen. The threshold for the first decision stage is derived from observations made on the experiments shown in Section 4.1. However, these observations should be verified if they generalize, therefore we also analyzed the 3,000 data points generated for the machine learning model (as shown in **Supplementary Material**, Section 3) and confirmed that the threshold of 60% is appropriate for this decision stage. The threshold for the second decision stage is based on theoretical knowledge about the computation patterns. The threshold should be chosen as a multiple of the warp size to ensure a full utilization of the computation blocks. We analyzed the performance as a function of the average number of non-zeros in a row (see **Supplementary Material**, Section 3) and derived the value of 128 as suitable decision threshold for our dataset. However, the analysis also suggests that this threshold could be fine-tuned to

[2] We use SIMD intrinsics which should not be confused with actual inline assembly (for more details, see: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html).

[3] Project homepage: http://math-atlas.sourceforge.net/.

[4] A warp is a group of 32 CUDA threads which process a given set of instructions at the same time. Even though they can proceed in the code concurrently, the efficiency rises if their execution does not diverge (Bell and Garland, 2009).

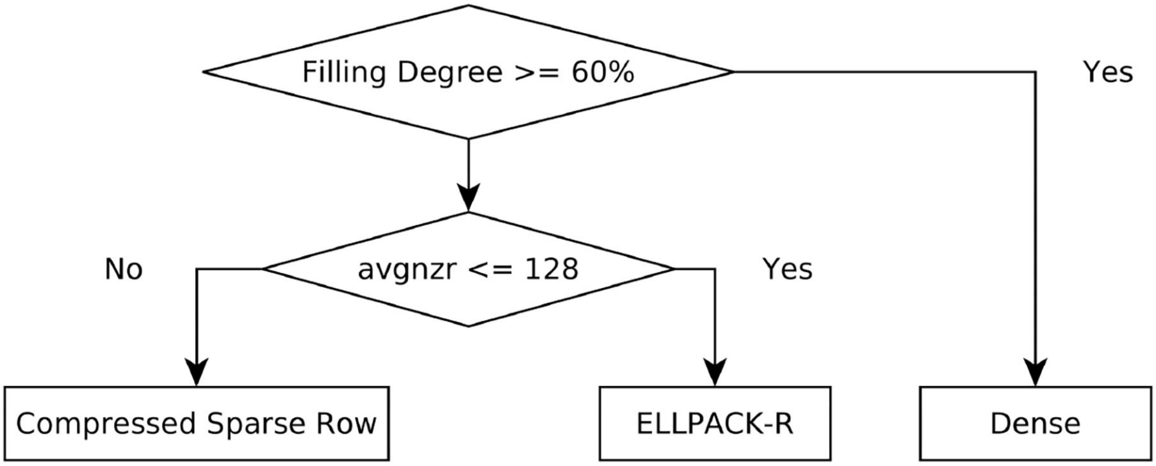


FIGURE 2 | Two-stage heuristic for the matrix format selection on GPUs. The threshold values for both decision points were selected based on the analysis of our datasets (see **Supplementary Material**, Section 3 for more details).

a specific CUDA device to achieve an optimal performance of the heuristic.

3.2. Format Selection Using Machine Learning

The heuristic approach is limited, as it is difficult to identify differences arising from the execution of a given implementation on different devices (see Section 4.2). To be efficient on diverse devices, one would need to fine-tune the decision parameters for each device. Therefore, it would be useful to have an automatic selection which can be adapted through machine learning to data obtained from each device.

The implementation of the prediction model requires three general steps. The first step is made offline and consists in generating the dataset necessary for the training and testing of the model. The second step is also offline and consists in training the model and testing it. The last one is online and consists of using the model and performing predictions that help in selecting the most suitable format.

3.2.1. Creation of the Dataset

For our benchmark, we follow a scheme similar to Dinkelbach et al. (2012). We create two populations in ANNarchy. The population sizes were randomly chosen from a fixed set of sizes within the range of 1,000–20,000 neurons. We create a projection between those two populations, which will be referred to as the connectivity matrix in this section. For the creation of this matrix, we either use a random probability (in the range of 1–100%) or a fixed number of entries per row (ranging from 128–4,096 entries). Using this scheme, we create 3,000 different network configurations. Each network is then generated, compiled and simulated for 1,000 steps using each data structure (in this case the CSR, ELLPACK-R and dense matrix formats). At the end of this procedure, we obtained 3,000 data points which consist of a list of features (described in the next section), the achieved computational time for each of the three formats and the format which would be chosen by the heuristic.

3.2.2. Feature Selection

The computation time of a rate-coded network heavily depends on the number of connections between the different populations. Since these various connections are structured in the format of sparse matrices, we focus on the properties of this particular type of matrix to define the relevant input features to the auto-tuning network. We derive for the matrices the features depicted in **Table 1**.

This set of features is a subset of features which are typically used in the SpMV auto-tuning literature (e.g., Li et al., 2013; Lehnert et al., 2016; Benatia et al., 2018; Chen et al., 2019). In particular, the work of Lehnert et al. (2016) and Benatia et al. (2018) suggests that the set of features used to detect a format depends on the format itself. For instance, we left out the difference between the maximum number of nonzeros (MAXNZR) and the average nonzeros per row (AVGNZR) as our preliminary experiments indicated that this feature is not helpful on our dataset. Considering the work of Vázquez et al. (2011) and Benatia et al. (2018), we believe this feature is a helpful indicator for the Hybrid format which is not used in the present work (see Section 5 for more details) and thus we omit this criterion. Li et al. (2013) proposed two additional values to characterize diagonals in matrices which might indicate the usage of diagonal formats. It is worth noting that not all approaches use such features. Pichel and Pateiro-Lopez (2018) use for example, an image-like tensor to represent the features of the connectivity matrix which is scaled down to be used as input to a convolutional neural network (AlexNet, Krizhevsky et al., 2012) to derive the optimal matrix format.

TABLE 1 | A set of features used to characterize the sparse matrices.

Features	Description
N	Number of rows in the matrix
M	Number of columns in the matrix
NNZ	Number of nonzeros in the matrix
DES	Density of the matrix
AVGNZR	Average number of nonzeros per row
MINNZR	Minimum number of nonzeros per row
MAXNZR	Maximum number of nonzeros per row

TABLE 2 | Best network configurations found by the Optuna library within 150 trials for each dataset.

	NVIDIA K20m	NVIDIA RTX 2060	NVIDIA RTX 3080
Normalization	7	7	7
Dense	119	124	155
Dense	187	195	86
Dense	199	105	85
Dense	96	127	150
Dense	/	/	66
Output	3	3	3

3.2.3. Machine Learning Model

The machine learning model is implemented using the TensorFlow (Abadi et al., 2016) library version 2.6.2[5]. The fully-connected feedforward neural network consists of an input layer with seven neurons representing the features (as discussed in Section 3.2.2), a feature normalization layer, a number of hidden layers and one output layer with three neurons. Each of these neurons represents a possible data structure: CSR, ELLPACK-R, and dense. The output of these neurons, i.e., the predicted performance for a given network in GFLOPs, is then read out to determine the fastest configuration. The hidden layers consist of rectified linear units (ReLu) and the number of layers as well as the number of units in each layer is determined by Optuna (Akiba et al., 2019), a Bayesian optimization library for hyper-parameter optimization used in many machine learning workflows. The search space is here the set of possible configurations, in our case the number of layers from 2 to 5 (motivated by the work of Benatia et al. (2018) who identified four layers as optimal for ELLPACK and five as optimal for CSR on their dataset), the number of neurons in each layer (64–256) and the learning rate (1e-7 to 1e-2). The objective function provided to Optuna is the test accuracy, an average resulting from a 5-fold cross-validation (see Section 4.4.1 for more details) without repetitions. We configured Optuna to perform 150 trials for each of the three datasets (i.e., the three CUDA devices considered in this work) and the obtained best configurations are depicted in **Table 2**.

The optimizer is Adam with the default parameters and the learning rate is determined by Optuna. The loss function is the mean squared error (mse), as this is a regression problem.

4. RESULTS

All the experiments were performed using the ANNarchy 4.7.1.1 release[6]. The measured computation times are recorded with the Python `time` package. When we analyze the performance in this section, we evaluate the execution of 1,000 steps within the ANNarchy neural simulator. As the populations are not defined by means of equations, the simulation time is almost equal to the execution time of the SpMV. We use in this article FLOPS (floating operations per second) as a metric to evaluate the performance, which is used commonly across the SpMV literature. This value is computed for a given data structure based on the measured computation time t in seconds for the 1,000 iterations (as mentioned in Section 3.2.1) and the number of nonzeros (nnz) in the matrix:

$$\text{FLOPS} = \frac{2 \times 1{,}000 \times nnz}{t} \tag{2}$$

The factor 2 comes from the fact that the SpMV requires one multiplication and one addition for each non-zero value. For an easier handling of the values, we transform then FLOPs to GFLOPs (giga-FLOPs). Langr and Tvrdik (2016) suggest to choose compiler flags for performance comparisons in order to achieve the best possible performance. The ANNarchy framework was therefore configured to use the optimization flags *-march=native*[7] *-O3*[8] *-ffast-math*[9] for the g++ compiler to enable typical optimizations. The CUDA compiler is configured without further compiler flags as *-O3* is automatically enabled for device codes[10]. For a more detailed discussion on the effect of *-ffast-math* and the CUDA compiler counterpart *–use_fast_math* we would like to refer to **Supplementary Material**, Section 4. The following sections will compare the performance achieved on three NVIDIA devices: a K20m, a RTX 2060, and a RTX 3080. Some hardware characteristics are provided in the **Supplementary Material**, Section 1.

4.1. Dense vs. Sparse Matrix Formats

Sparse matrix representations require a memory overhead to index the elements of a matrix (e.g., row pointers). When the matrix becomes denser, it may become inefficient to use a sparse matrix representation instead of a dense one (see **Supplementary Material**, Section 2 for more details). To illustrate this, we define a 2,000 × 2,000 matrix with varying sparsity levels ranging from 10% to fully-connected. We compare the achieved throughput in GFLOPs averaged across 15 runs for a single thread on a AMD Ryzen 7 2700X CPU (**Figure 3**) and three different NVIDIA devices (**Figure 4**). The CSR data structure (blue), the dense format (orange) and a format selected by the heuristic (green) are compared.

[5] https://doi.org/10.5281/zenodo.5645375

[6] https://doi.org/10.5281/zenodo.6417924

[7] The march flag let the compiler generate the code for a specific CPU architecture. Providing *native* let the compiler determine the CPU automatically. For more details, see https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html.

[8] For more details, see https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[9] For more details, see https://gcc.gnu.org/wiki/FloatingPointMath.

[10] https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#ptxas-options-opt-level

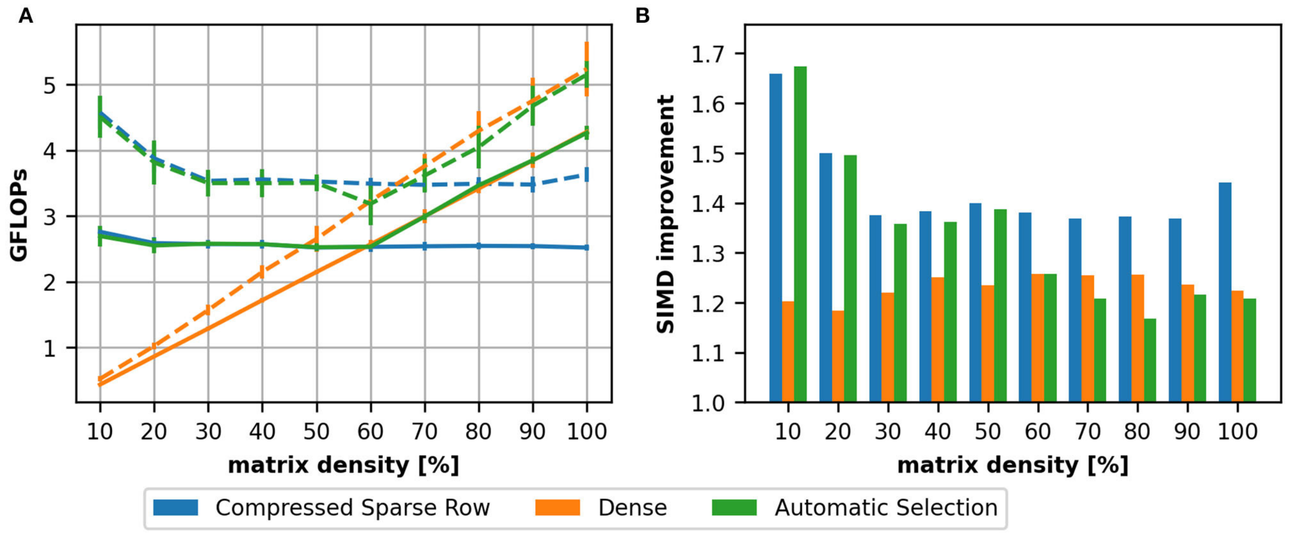


FIGURE 3 | Comparison between a dense matrix representation and the compressed sparse row format on a AMD Ryzen 7 2700X using a single thread. We depict the achieved performance in GFLOPs as a function of matrix density **(A)**. In this setup we compare a 2,000 × 2,000 matrix with varying density levels and compare a CSR (blue) and dense (orange) implementation. We compared additionally the improvement by a hand-written AVX implementation (dashed line). The gained improvement by this implementation is depicted in **(B)**.

For the CPU (**Figure 3A**), we can see that the GFLOPs are almost constant for the CSR format, i.e., the computation time increases linearly with the number of non-zeros in the matrix, while the contrary applies for the dense matrix as the computation time is not dependent on the number of non-zeros: the achieved GFLOPs are low for sparse matrices and increase with the matrix density. As outlined in Section 2.2, we added also hand-written vectorization using AVX on the AMD Ryzen7 CPU. The results for the vectorized implementations are depicted in **Figure 3A** as dashed lines. The relative improvement provided by the vectorization is also depicted as a bar graph in **Figure 3B**. We can see that the improvement is below the theoretical maximum which would be four for double precision on an AVX-capable CPU. The reduced efficiency, especially for the dense matrix format, should be linked to the fact that the SpMV is a memory-bound problem. We also see that the improvement is almost the same for a density around 20% while the improvement achieved on the CSR depends on the density: for small densities, the implementation benefits mostly for small row lengths and the reduced memory consumption.

To evaluate the performance on GPUs we compare the K20m (**Figure 4A**), the RTX 2060 (**Figure 4B**) and the RTX 3080 (**Figure 4C**). On all three devices, we can see that for small densities the achieved throughput of the CSR (blue line) implementation is lower than for higher densities. This is a consequence of the implementation [as discussed in Section 3.1; more details can be found in Dinkelbach et al. (2012) for our version of the CSR and in Vázquez et al. (2011) for the ELLPACK-R format] as the thread groups processes rows together: there must be a sufficient number of elements in a row to achieve a high throughput.

In both experiments, we can see that, for higher matrix densities, the CSR format is outperformed by the dense matrix format (orange line). This motivated the first stage of our heuristic (green line). The value 60% was originally obtained on the K20m GPU. A comparison to the newer devices would suggest 70%. We have analyzed this for all examples in our dataset and determined 60% as a suitable value (see **Supplementary Material**, Section 3).

4.2. Different Sparse Matrix Formats

This section illustrates the necessity for different sparse matrix formats. We investigate the performance improvement of an ELLPACK-R and dense implementation against the CSR on three GPUs which is a criterion suggested by Langr and Tvrdik (2016). To compare the formats, we compute the ratio between the GFLOPS required by CSR and the GFLOPS of the other format. A more detailed analysis of these values is depicted in the **Supplementary Material**, Section 3.

Figure 5 depicts the average performance on the 3,000 data points in our dataset. The orange line represents the median of the obtained values and the green triangle represents the mean. The CSR format outperforms the other two formats in most cases on the K20m (**Figure 5A**) and the RTX 3080 (**Figure 5C**), as the average performance of ELLPACK-R and dense is lower than 1.0. However, there is a noticeable number of values >1.0, indicating that some matrices benefit from another format than CSR. We also found that the results on the RTX 2060 (**Figure 5B**) are different in the sense that the ELLPACK-R outperforms in many cases the CSR format which is represented by the average >1.0.

Comparing the results obtained on the three investigated CUDA devices supports the claim of Balaprakash et al.

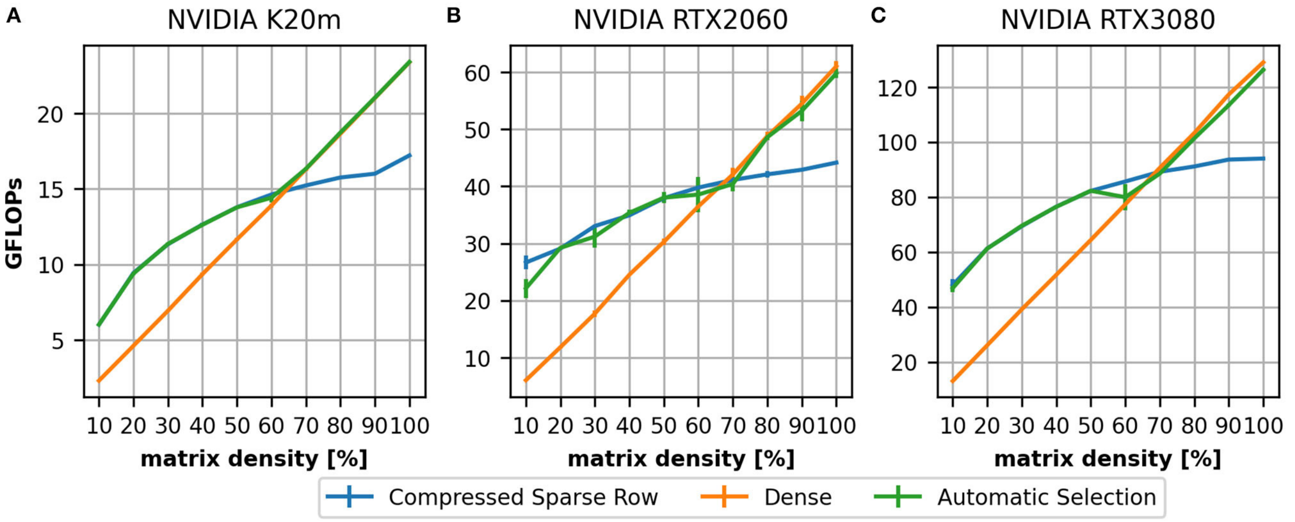


FIGURE 4 | Achieved performance in GFLOPs on three devices: a NVIDIA K20m **(A)**, a NVIDIA RTX 2060 **(B)**, and a NVIDIA RTX 3080 **(C)**. As for the single thread CPU (**Figure 3**) we compare a CSR (blue) and dense (orange) implementation on a 2,000 times 2,000 matrix with varying filling degree. In the range of 60–70% the dense matrix representation outperforms the CSR which motivated the first stage of our heuristic.

(2018). The performance behavior of a given implementation can drastically change with evolving hardware. The relative performance of our ELLPACK-R and dense implementations toward the CSR implementation indeed shrinks noticeably.

4.3. Automatic Format Selection

In this section, we report on the results of the two strategies for automatic format selection: the heuristic and the predictive machine learning approach. We compare the results on the K20m (**Figure 6A**), the RTX 2060 (**Figure 6B**), and the RTX 3080 (**Figure 6C**). Considering the distribution of the selected formats, we generally notice that there is no significant difference between the K20m and the RTX 3080 but the results of RTX 2060 appears to deviate. Furthermore, the machine learning model delivers more accurate results than the heuristic, especially on the RTX 2060. The heuristic tends to select on all three devices the CSR (blue bars) in too many cases, in particular on the RTX 2060. As noted earlier, this might be improved by device-specific thresholds used in the second stage of the heuristic. The machine learning model was able to select in 95.67% (K20m), 93.0% (RTX 2060), and 94.83% (RTX 3080) of the cases the correct format resulting in the fastest computation time. The selection of the heuristic was in 87.67% (K20m), 71.67% (RTX 2060), and 77.83% (RTX 3080) of the cases correct. We hypothesize that device-specific decision thresholds could improve the performance achieved on the RTX 2060 and RTX 3080, but it would be difficult to derive these thresholds on all possible hardware. It might be interesting to note that CSR format was in 63.83% (K20m), 44.67% (RTX 2060), and 60.17% (RTX 3080) of the cases the correct format.

4.4. Validation and Stability of the Machine Learning Approach

The performance of the ML approach depends on the correct selection of features and the size of the dataset dedicated to training and testing. However, the choice of a basic cross-validation method (random split of the data into 80% for training and 20% for testing) is not sufficient to estimate the appropriateness of the trained model, since it may have by coincidence excellent results only on the part selected for testing (20%). To avoid this issue, we have opted for the repetitive cross-validation method (Section 4.4.1). To define the proper size of the data required to obtain a stable model (a high accuracy with the lowest standard deviation), we also perform tests (using the repetitive cross-validation method) on different dataset sizes (Section 4.4.2).

4.4.1. Cross-Validation

The five-fold cross-validation procedure divides the data set into five non-overlapping folds. During each iteration of the process, a fold is retained as a test set, while all others are used for the training. In the end, a total of five models are fitted and evaluated on the five retained test sets, and the average performance accuracy is calculated. This procedure is repeated ten times, and the mean performance across all folds and all repetitions is reported.

Figure 7 shows the variation of the performance of the 10 repetitive five-fold cross-validations applied on the dataset of the NVIDIA K20m. We can see that for the dataset with 3,000 data points, the optimal performance selection rate slightly varies depending on the fraction of data selected as training set but retains a high level of correctness over 93% and therefore still outperforms the heuristic.

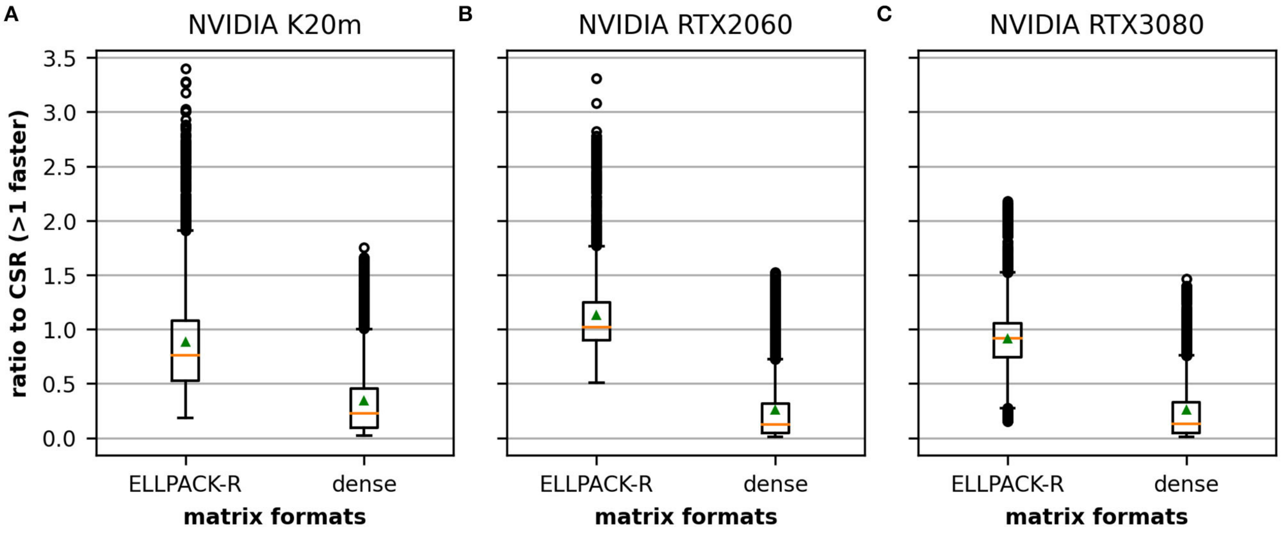


FIGURE 5 | Relative performance of ELLPACK-R and dense matrices in comparison to a CSR averaged across the 3,000 matrices in our dataset. We compare the results obtained on the NVIDIA K20m **(A)**, the NVIDIA RTX 2060 **(B)**, and the NVIDIA RTX 3080 **(C)**. Although CSR is the fastest data structure in many cases, there is a noticeable number of cases where the other formats appear to be superior. The performance differences between the matrix formats are higher on the Tesla K20m **(A)** and the NVIDIA RTX 2060 **(B)** than on the RTX 3080 **(C)** especially for the ELLPACK-R matrix format. The orange line depicts the median, the green triangle the mean and the circle denote outliers.

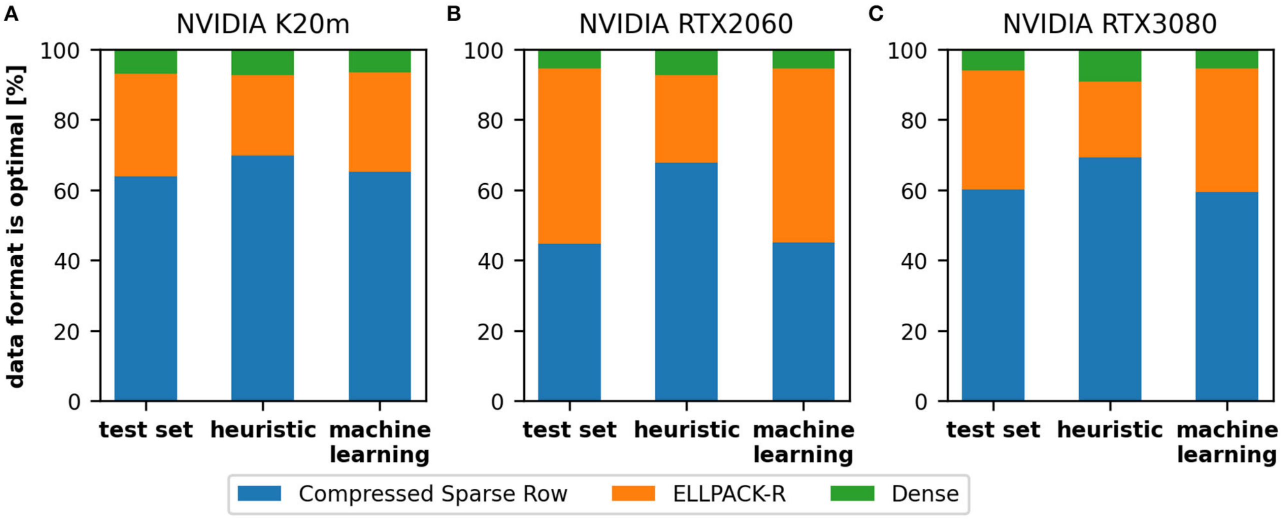


FIGURE 6 | The distribution of selected formats on three GPUs: NVIDIA K20m **(A)**, NVIDIA RTX 2060 **(B)**, and NVIDIA RTX 3080 **(C)**. We compare the data (left), the heuristic (middle), and the machine learning model (right) for each GPU. We can see that our heuristic tends to select the compressed sparse row (blue bars) in too many cases, which leads to lower performance, in particular on the NVIDIA RTX 2060.

4.4.2. Influence of the Size of the Dataset

Generating the dataset can be quite time-consuming: the generation of the 3,000 data points required 2–3 days in this case. We therefore performed experiments (multiple repetitive five-fold cross-validations with varying each time the size of the dataset) to define the smallest dataset size enabling us to achieve a good accuracy of the selection of the correct matrix format. Bayesian optimization using Optuna for 150 trials is used to select the best architecture in each case.

Figure 8 shows the accuracy variation of the optimal format selection with respect to the number of samples used for training. As one would expect, the performance increases with the size

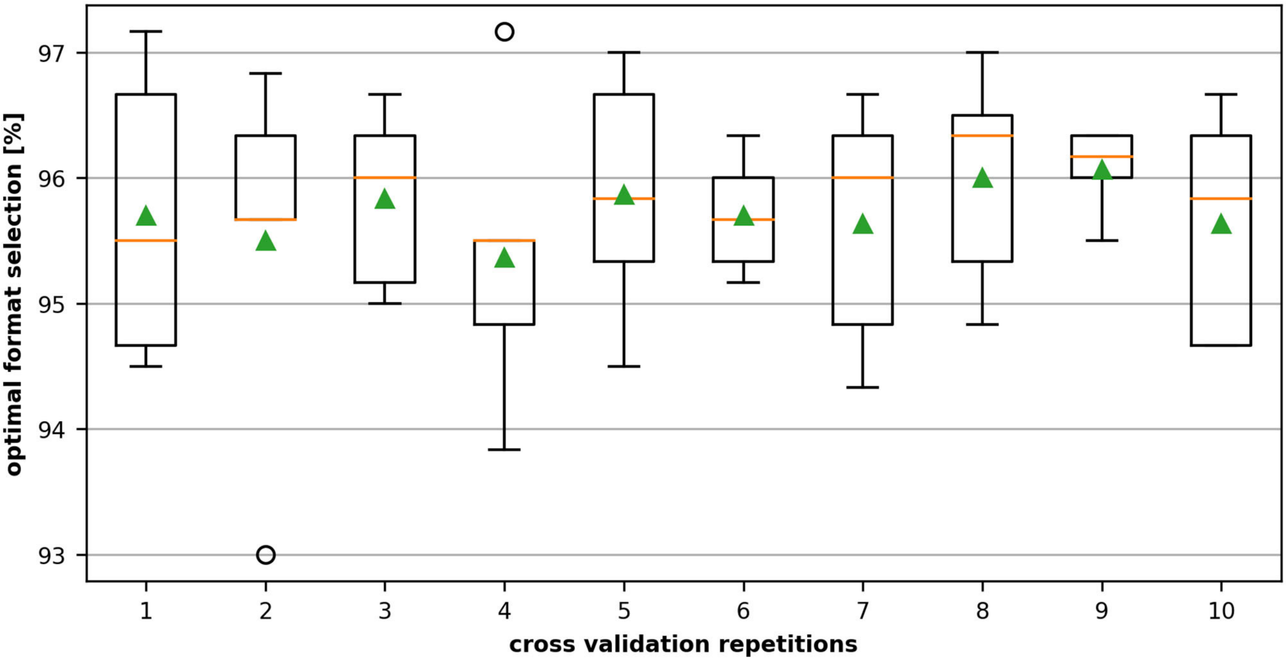


FIGURE 7 | Ten repeated cross-validations on 3,000 data samples recorded on the NVIDIA Tesla K20m. The dataset is divided into five non-overlapping folds. During each validation stage, four folds containing 2,400 samples are used for training, and the remaining fold with 600 units is used for testing the accuracy of the best format selection. The middle (orange) line of the box is the median, the green triangle the mean and the circles denote outliers.

of the dataset. However, already with one-third of the dataset we could achieve an accuracy of 92.94% for the selection of the optimal format.

5. DISCUSSION

Continuous transmission is a dominating computation kernel for rate-coded neural networks (Dinkelbach et al., 2012) that corresponds to the sparse matrix vector multiplication, a well-investigated topic by many researchers over decades on various hardware platforms. In this article, we investigated the application of the ELLPACK-R and dense format derived from the literature and study their performance within the neural simulation framework ANNarchy.

As stated in the literature, there is no “one-size-fits-all” solution, although the CSR format achieves a good performance in many cases, which was also shown, e.g., by Benatia et al. (2018) or Chen et al. (2019). Using a larger set of connection matrices, we have shown that the usage of different matrix formats can help to improve the performance on CPUs as well as GPUs by distinguishing between sparse and dense matrices (Section 4.1). For GPUs, we further studied the ELLPACK-R format proposed by Vázquez et al. (2011) in addition to our CSR implementation (Dinkelbach et al., 2012). In Section 4.2, we have shown that CSR is in many cases the best format, but it can be outperformed by a noticeable factor by the ELLPACK-R and the dense matrix format. In summary, the availability of different sparse matrix formats can be used to improve the performance but the selection is not trivial, as expected from the literature (e.g., Liu and Vinter, 2015a).

In the case of heavy simulations, a user-friendly simulation environment should measure and select the right sparse matrix format for a specific network. We presented a first automatic selection based on some simple rules which we derived from experiments and which is implemented in ANNarchy 4.7.1.1. We have also shown that this heuristic-based selection can be improved by the help of machine learning techniques. Our approach using machine learning techniques is comparable to the work of Lehnert et al. (2016) and Benatia et al. (2018). Based on a set of features, we build up a neural network which predicts the performance of the format. Lehnert et al. (2016) used computational time for the performance evaluation while we used GFLOPs as a metric. Both our work and that of Lehnert et al. (2016) uses regression for the prediction of the performance of the data format. Contrary to the previously discussed works, we do not use a fixed network but use the hyperparameter optimization framework Optuna to find a suitable network configuration for a given dataset. There is an important caveat: Comparing matrix formats using FLOPS as a metric generates a hardware dependency (Langr and Tvrdik, 2016), which we also observed in our recorded data (see Section 4.2). This means that the users need to generate the dataset on their own machine, which requires several hours up to a few days for the data

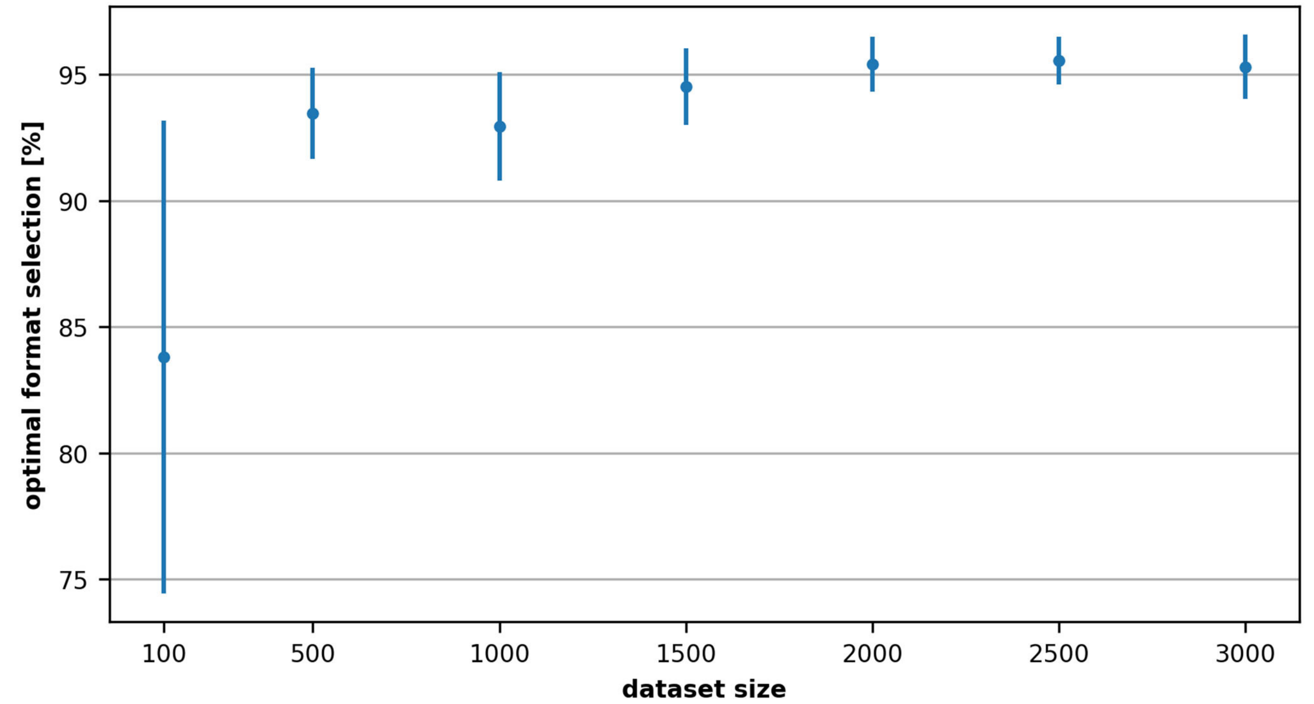


FIGURE 8 | Variation of the accuracy of the optimal format selection with respect to the number of samples used for training (NVIDIA Tesla K20m). Each measurement and its corresponding standard deviation represents the average of 10 repeated cross-validations.

generation, although the results in Section 4.4.2 suggest that the number of required data points can be reduced.

The present work demonstrates the performance improvements that can be reached by using the ELLPACK-R format in ANNarchy. However, the ELLPACK/ELLPACK-R formats require more memory caused by padding zeros for strongly varying row lengths and therefore, Bell and Garland (2009) proposed a Hybrid format, which combines an ELLPACK format for most entries, and those elements which are in the long rows are stored in a separate coordinate format. This was not the case in our dataset, and its not clear to us how relevant this is for neurocomputational models, as this would mean that the number of synapses per neuron vary strongly within one projection. The present CSR implementation could be further optimized for short rows using the CSR-stream implementation proposed by Greathouse and Daga (2014), although this introduces another hyper parameter: the number of nonzeros processed by one warp. The CSR5 storage format (Liu and Vinter, 2015a) introduces additional two hyperparameters but should be efficient for SIMD-capable CPUs, GPUs, or other accelerators like the Xeon Phi, while introducing a memory overhead around 2% of the original CSR (Liu and Vinter, 2015b).

Other works focus on the grouping of rows into computation blocks, i.e., by slicing the matrix into pieces, as done for the CSR (e.g., Oberhuber et al., 2011) or the ELLPACK format (e.g., Monakov et al., 2010; Kreutzer et al., 2014). Kreutzer et al. (2014) highlight that their modified sliced ELLPACK format is applicable to GPUs as well as SIMD-capable CPUs. Another class of formats proposed in the literature are blocked formats such as the blocked compressed sparse row (BSR or BCSR, e.g., Choi et al., 2010; Verschoor and Jalba, 2012; Eberhardt and Hoemmen, 2016; Benatia et al., 2018) or the blocked ELLPACK format (Choi et al., 2010). The idea is that matrix is split into several small dense matrices. As these sub-matrices are dense, a coalesced and fully cacheable access to the dense vector is possible, which is desirable for performance (Temam and Jalby, 1992; Im and Yelick, 2001; Im et al., 2004; Goumas et al., 2008; Williams et al., 2009). These formats appear to be efficient if the nonzeros in a matrix are clustered, although the selection of the correct block size can be challenging (Im and Yelick, 2001). For matrices where the nonzeros are widely spread, the memory overhead will be too large and no performance benefit can be expected in comparison to other formats.

The present work focuses on the performance prediction for sparse matrix formats on GPUs. Nonetheless, the same procedure can be applied for CPUs. Preliminary tests with the current ANNarchy 4.7.1 release has shown that the performance differences between formats are small in comparison to the differences observed on GPU. This hardens the correct performance prediction and opens the question of whether the approach is necessary at all. It is important to note that the recent implementations of our CPU formats are not comparable to highly optimized libraries like OSKI, SPARSITY, or ATLAS, as low-level optimization like padding, local store blocking or register blocking (e.g., presented in Im and Yelick, 2001; Im et al., 2004; Williams et al., 2009) are still missing. We started to apply such optimizations, e.g., hand-written SpMV which improve the performance (see Section 4.1), but this increases the complexity of the code generation noticeably. Nonetheless, we have implemented in the ANNarchy 4.7.1.1 the heuristic selection of dense matrices instead of sparse matrices.

Brian2 (Stimberg et al., 2019), GeNN (Yavuz et al., 2016) as well as ANNarchy do not switch the floating precision from double to single precision automatically. As highlighted by Hopkins et al. (2020), this could lead to numerical errors whose importance need to be evaluated by the modeler. However, the performance improvement on GPUs and CPUs (especially using SIMD extension) could be noticeable. The reduction of precision can improve the performance of the SpMV, e.g., shown by Bell and Garland (2009) or Greathouse and Daga (2014) and is therefore beneficial for the simulation of rate-coded models (Dinkelbach et al., 2012). Yavuz et al. (2016) have shown that the choice of single precision in context of two spiking models at different scales can improve the performance.

The presented findings may also be of interest for the implementation of spiking networks. The currently available spiking simulators use either CSR-like (e.g., Brian2, GeNN, coreNeuron; Kumbhar et al., 2019), dense (e.g., GeNN) or object-oriented (NEST) representation of synapses, while also using code generation approaches (see Blundell et al., 2018 for a recent review). At the very least, the differentiation between sparse and dense matrices could be helpful for some models as shown by Yavuz et al. (2016), as the usage of dense matrices does not break coalescence as CSR does (e.g., Dinkelbach et al., 2012; Yavuz et al., 2016). The computational load induced by the spike propagation can be quite low in comparison to the update of neural equations (Plesser and Diesmann, 2009), so there is a chance that the overhead induced by the sparse matrix format can have a negative impact on performance.

Ongoing work will target the application of other sparse matrix formats for the simulation of rate-coded and spiking models in ANNarchy. For rate-coded models, this could be formats which use structural properties, such as the diagonal format. Some neuro-computational models developed in our lab (e.g., Jamalian et al., 2017) contain matrices which have a banded matrix structure. A promising direction may be the implementation of sliced matrix formats (e.g., Kreutzer et al., 2014). For spiking models, the compressed sparse blocks format (CSB, Buluç et al., 2009, 2011) could be beneficial for the implementation of spiking models with plasticity rules. The CSB format is proposed to be suitable for the SpMV as well as the transposed SpMV, an uncommon property for SpMV formats (Buluç et al., 2009; Steinberger et al., 2016). With respect to the machine learning model, reducing the number of required data points is critical, as users will likely not be patient enough to gather the necessary data. Active learning methods (Cohn et al., 1996) may be used to allow the ML network to ask for additional samples where its uncertainty is maximal, focusing data generation to the most interesting regions.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found at: Neural simulator ANNarchy: https://github.com/ANNarchy/ANNarchy (zenodo doi: 10.5281/zenodo.6417924); Scripts for simulation/analysis: https://github.com/hamkerlab/Dinkelbach2022_ANNarchyAutoTuning (zenodo doi: 10.5281/zenodo.6534573).

AUTHOR CONTRIBUTIONS

HD and B-EB designed and performed the research, programming, and data analysis. JV and FH guided the research. FH acquired the funding. HD writing first draft. HD, B-EB, JV, and FH writing, reviewing, and editing. All authors contributed to the article and approved the submitted version.

FUNDING

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) with the project Auto-tuning for neural simulations on different parallel hardware (DFG HA2630/9-1). The publication of this article was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) project number 491193532 and the Chemnitz University of Technology.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2022.877945/full#supplementary-material

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. (2016). "Tensorflow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16* (Savanna, GA: USENIX Association), 265–283.

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. (2019). "Optuna: a next-generation hyperparameter optimization framework," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Anchorage, AL: ACM), 2623–2631. doi: 10.1145/3292500.3330701

Balaprakash, P., Dongarra, J., Gamblin, T., Hall, M., Hollingsworth, J. K., Norris, B., et al. (2018). Autotuning in high-performance computing applications. *Proc. IEEE* 106, 2068–2083. doi: 10.1109/JPROC.2018.2841200

Bell, N., and Garland, M. (2009). "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09* (New York, NY: ACM Press). doi: 10.1145/1654059.1654078

Benatia, A., Ji, W., Wang, Y., and Shi, F. (2018). BestSF: a sparse meta-format for optimizing SpMV on GPU. *ACM Trans. Architect. Code Optim*. 15, 1–27. doi: 10.1145/3226228

Blundell, I., Brette, R., Cleland, T. A., Close, T. G., Coca, D., Davison, A. P., et al. (2018). Code generation in computational neuroscience: a review of tools and techniques. *Front. Neuroinform*. 12, 68. doi: 10.3389/fninf.2018.00068

Buluç, A., Fineman, J. T., Frigo, M., Gilbert, J. R., and Leiserson, C. E. (2009). "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures - SPAA '09* (Calgary, AB), 233. doi: 10.1145/1583991.1584053

Buluç, A., Williams, S., Oliker, L., and Demmel, J. (2011). "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011* (Anchorage, AL: IEEE), 721–733. doi: 10.1109/IPDPS.2011.73

Chen, S., Fang, J., Chen, D., Xu, C., and Wang, Z. (2019). "Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures," in *Proceedings - 20th International Conference on High Performance Computing and Communications, 16th International Conference on Smart City and 4th International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2018* (Exeter), 649–658. doi: 10.1109/HPCC/SmartCity/DSS.2018.00116

Choi, J. W., Singh, A., and Vuduc, R. W. (2010). Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM Sigplan Not*. 45, 115. doi: 10.1145/1837853.1693471

Cohn, D. A., Ghahramani, Z., and Jordan, M. I. (1996). Active learning with statistical models. *J. Artif. Intell. Res*. 4, 129–145. doi: 10.1613/jair.295

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., et al. (2008). "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008* (Austin, TX). doi: 10.1109/SC.2008.5222004

Dinkelbach, H. Ü., Vitay, J., Beuth, F., and Hamker, F. H. (2012). Comparison of GPU-and CPU-implementations of mean-firing rate neural networks on parallel hardware. *Network* 23, 212–236. doi: 10.3109/0954898X.2012.739292

Dinkelbach, H. Ü., Vitay, J., and Hamker, F. H. (2019). "Scalable simulation of rate-coded and spiking neural networks on shared memory systems," in *2019 Conference on Cognitive Computational Neuroscience* (Berlin), 526–529. doi: 10.32470/CCN.2019.1109-0

Eberhardt, R., and Hoemmen, M. (2016). "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures," in *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016* (Chicago, IL: IEEE), 663–672. doi: 10.1109/IPDPSW.2016.42

Filippone, S., Cardellini, V., Barbieri, D., and Fanfarillo, A. (2017). Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Softw*. 43, 1–49. doi: 10.1145/3017994

Ganapathi, A., Datta, K., Fox, A., and Patterson, D. (2009). "A case for machine learning to optimize multicore performance," in *1st USENIX Workshop on Hot Topics in Parallelism, HotPar 2009* 2009 (Berkeley, CA).

Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., and Koziris, N. (2008). "Understanding the performance of sparse matrix-vector multiplication," in *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2008* (Toulouse), 283–292. doi: 10.1109/PDP.2008.41

Greathouse, J. L., and Daga, M. (2014). "Efficient sparse matrix-vector multiplication on gpus using the CSR storage format," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (New Orleans, LA), 769–780. doi: 10.1109/SC.2014.68

Guo, P., and Wang, L. (2010). "Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs," in *Proceedings - 2010 International Conference on Computational and Information Sciences, ICCIS 2010* (Chengdu), 1154–1157. doi: 10.1109/ICCIS.2010.285

Hopkins, M., Mikaitis, M., Lester, D. R., and Furber, S. (2020). Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. *Philos. Trans. R. Soc. A* 378, 20190052. doi: 10.1098/rsta.2019.0052

Hou, K., Feng, W. C., and Che, S. (2017). "Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi- and many-core processors," in *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017*, 713–722. doi: 10.1109/IPDPSW.2017.155

Im, E.-J., and Yelick, K. (2001). Optimizing sparse matrix computations for register reuse in Sparsity. *Lect. Notes Comput. Sci*. 2073/2001, 127–136. doi: 10.1007/3-540-45545-0_22

Im, E. J., Yelick, K., and Vuduc, R. (2004). Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perf. Comput. Appl*. 18, 135–158. doi: 10.1177/1094342004041296

Jamalian, A., Bergelt, J., Dinkelbach, H. Ü., and Hamker, F. H. (2017). "Spatial attention improves object localization: a biologically plausible neuro-computational model for use in virtual reality," in *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)* (Venice), *Vol. 2018*, 2724–2729. doi: 10.1109/ICCVW.2017.320

Kincaid, D. R., Oppe, T. C., and Young, D. M. (1989). *Itpackv 2d User's Guide, Technical Report CNA-232*. Technical report, Center for Numerical Analysis. University of Texas at Austin.

Kreutzer, M., Hager, G., Wellein, G., Fehske, H., and Bishop, A. R. (2014). A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM J. Sci. Comput*. 36, C401–C423. doi: 10.1137/130930352

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). "ImageNet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems* (Lake Tahoe, NV), 1097–1105.

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., et al. (2019). Coreneuron: an optimized compute engine for the neuron simulator. *Front. Neuroinform*. 13, 63. doi: 10.3389/fninf.2019.00063

Langr, D., and Tvrdik, P. (2016). Evaluation criteria for sparse matrix storage formats. *IEEE Trans. Parallel Distrib. Syst*. 27, 428–440. doi: 10.1109/TPDS.2015.2401575

Lehnert, C., Berrendorf, R., Ecker, J. P., and Mannuss, F. (2016). "Performance prediction and ranking of SpMV kernels on GPU architectures," in *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Grenoble), 9833. doi: 10.1007/978-3-319-43659-3_7

Li, J., Tan, G., Chen, M., and Sun, N. (2013). "SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Seattle, WA: ACM), 117–126. doi: 10.1145/2491956.2462181

Liu, W., and Vinter, B. (2015a). "CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing* (New York, NY: ACM), 339–350. doi: 10.1145/2751205.2751209

Liu, W., and Vinter, B. (2015b). Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Comput*. 49, 179–193. doi: 10.1016/j.parco.2015.04.004

Monakov, A., Lokhmotov, A., and Avetisyan, A. (2010). "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *International Conference on High-Performance Embedded Architectures and Compilers*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Pisa), 5952. doi: 10.1007/978-3-642-11515-8_10

Oberhuber, T., Suzuki, A., and Vacata, J. (2011). New row-grouped CSR format for storing sparse matrices on gpu with implementation in CUDA. *Acta Techn. CSAV* 56, 447–466. Available online at: http://journal.it.cas.cz/56(11)4-Contents/56(11)4c.pdf

Pichel, J. C., and Pateiro-Lopez, B. (2018). "A new approach for sparse matrix classification based on deep learning techniques," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC* (Belfast: IEEE), 46–54. doi: 10.1109/CLUSTER.2018.00017

Plesser, H. E., and Diesmann, M. (2009). Simplicity and efficiency of integrate-and-fire neuron models. *Neural Comput*. 21, 353–359. doi: 10.1162/neco.2008.03-08-731

Sedaghati, N., Ashari, A., Pouchet, L.-N., Parthasarathy, S., and Sadayappan, P. (2015). "Characterizing dataset dependence for sparse matrix-vector multiplication on GPUs," in *Proceedings of the 2nd Workshop on Parallel Programming for Analytics Applications - PPAA 2015* (San Francisco, CA), 17–24. doi: 10.1145/2726935.2726941

Steinberger, M., Derlery, A., Zayer, R., and Seidel, H. P. (2016). "How naive is naive SPMV on the GPU?," in *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016* (Waltham, MA). doi: 10.1109/HPEC.2016.7761634

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8, e47314. doi: 10.7554/eLife.47314.028

Su, B.-Y., and Keutzer, K. (2012). "clSpMV: A cross-platform openCL SpMV framework on GPUs," in *Proceedings of the 26th ACM international conference on Supercomputing - ICS '12* (Venice: ACM), 353. doi: 10.1145/2304576.2304624

Temam, O., and Jalby, W. (1992). "Characterizing the behavior of sparse algorithms on caches," in *Proceedings Supercomputing '92* (Minneapolis, MN), 578–587. doi: 10.1109/SUPERC.1992.236646

Vázquez, F., Fernández, J. J., and Garzón, E. M. (2011). A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurr. Comput*. 23, 815–826. doi: 10.1002/cpe.1658

Vázquez, F., Fernández, J. J., and Garzón, E. M. (2012). Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. *Parallel Comput*. 38, 408–420. doi: 10.1016/j.parco.2011.08.003

Verschoor, M., and Jalba, A. C. (2012). Analysis and performance estimation of the conjugate gradient method on multiple GPUs. *Parallel Comput*. 38, 552–575. doi: 10.1016/j.parco.2012.07.002

Vitay, J., Dinkelbach, H. Ü., and Hamker, F. H. (2015). Annarchy: a code generation approach to neural simulations on parallel hardware. *Front. Neuroinformatics* 9, 19. doi: 10.3389/fninf.2015.00019

Vuduc, R., Demmel, J. W., and Yelick, K. A. (2005). OSKI: a library of automatically tuned sparse matrix kernels. *J. Phys*. 16, 521–530. doi: 10.1088/1742-6596/16/1/071

Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001). Automated emperical optimization of software and the atlas project. *Parallel Comput*. 27, 3–35. doi: 10.1016/S0167-8191(00)00087-9

Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. (2009). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput*. 35, 178–194. doi: 10.1016/j.parco.2008.12.006

Yavuz, E., Turner, J., and Nowotny, T. (2016). Genn: a code generation framework for accelerated brain simulations. *Sci. Rep*. 6, 18854. doi: 10.1038/srep18854

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Neuroinformatics

ORIGINAL RESEARCH
published: 30 May 2022
doi: 10.3389/fninf.2022.883360

Mapping and Validating a Point Neuron Model on Intel's Neuromorphic Hardware Loihi

Srijanie Dey and Alexander Dimitrov**

Department of Mathematics, Washington State University, Vancouver, WA, United States

Neuromorphic hardware is based on emulating the natural biological structure of the brain. Since its computational model is similar to standard neural models, it could serve as a computational accelerator for research projects in the field of neuroscience and artificial intelligence, including biomedical applications. However, in order to exploit this new generation of computer chips, we ought to perform rigorous simulation and consequent validation of neuromorphic models against their conventional implementations. In this work, we lay out the numeric groundwork to enable a comparison between neuromorphic and conventional platforms. "Loihi"—Intel's fifth generation neuromorphic chip, which is based on the idea of Spiking Neural Networks (SNNs) emulating the activity of neurons in the brain, serves as our neuromorphic platform. The work here focuses on Leaky Integrate and Fire (LIF) models based on neurons in the mouse primary visual cortex and matched to a rich data set of anatomical, physiological and behavioral constraints. Simulations on classical hardware serve as the validation platform for the neuromorphic implementation. We find that Loihi replicates classical simulations very efficiently with high precision. As a by-product, we also investigate Loihi's potential in terms of scalability and performance and find that it scales notably well in terms of run-time performance as the simulated networks become larger.

Keywords: neuromorphic computing, LIF models, neural simulations, validation, performance analysis

Edited by:
James B. Aimone,
Sandia National Laboratories (DOE),
United States

Reviewed by:
Oliver Rhodes,
The University of Manchester,
United Kingdom
Fredrick Henry Rothganger,
Sandia National Laboratories,
United States

****Correspondence:***
Srijanie Dey
srijanie.dey@wsu.edu
Alexander Dimitrov
alex.dimitrov@wsu.edu

Received: *25 February 2022*
Accepted: *25 April 2022*
Published: *30 May 2022*

Citation:
Dey S and Dimitrov A (2022) Mapping and Validating a Point Neuron Model on Intel's Neuromorphic Hardware Loihi. Front. Neuroinform. 16:883360. doi: 10.3389/fninf.2022.883360

1. INTRODUCTION

The human brain is a rich complex organ made up of numerous neurons and synapses. Replicating the brain structure and functionality in classical hardware is an ongoing challenge given the complexity of the brain and limitations of hardware. The advent of supercomputers now allows for complex neural models, but at a huge cost of both software complexity and energy consumption.

A recent intense focus on brain studies, with the BRAIN initiative at the US (Insel et al., 2013), the Human Brain Project (HBP) in Europe (Markram et al., 2011), and philanthropic endeavors like Janelia Research Campus (Winnubst et al., 2019), and the Allen Institute for Brain Science (AIBS) (Lein et al., 2007), has produced a wealth of new data and knowledge, from records of neuronal and network dynamics, to fine-grained data on network micro- and nano-structure, bringing in the era of big neural data. At the same time, advances in electronics and the search for post-von Neumann computational paradigms has led to the creation of neuromorphic systems like Intel's Loihi (Davies et al., 2018), IBM's TrueNorth (Akopyan et al., 2015; DeBole et al., 2019; Löhr et al., 2020) and HBP's SpiNNaker (Khan et al., 2008), and BrainScaleS (Grübl et al., 2020).

Neuromorphic chips, as the name suggests—"like the brain"—can mimic the brain's function in a truer sense as their design is analogous to the brain (Thakur et al., 2018; Roy et al., 2019). Inspired by its architecture, we work on developing a principled approach toward obtaining simulations of biologically relevant neural network models on a novel neuromorphic commercial hardware platform.

Computers today are limited in this respect because of the way they have been built historically and the way they process data leading toward more energy and resource consumption in order to maintain versatility (Nawrocki et al., 2016; Ou et al., 2020). Neuromorphic chips on the other hand claim to be faster and more efficient for a set of specialized tasks (Bhuiyan et al., 2010; Sharp and Furber, 2013). In this study, we lay out a numeric groundwork to validate this assertion based on neural models derived from the primary visual cortex (VISp) of the mouse brain, as seen in recent work done on SpiNNaker (Knight and Furber, 2016; Rhodes et al., 2019). Intel's neuromorphic chip "Loihi" serves as our neuromorphic platform. Results obtained in Loihi are validated against classical simulations (Rossant et al., 2010; Nandi et al., 2020; Wang et al., 2020) given by AIBS's software package the Brain Modeling Toolkit (BMTK) (Dai et al., 2020).

In this manuscript we focus on the Loihi architecture, as it is at present one of the most powerful platforms with specialized digital hardware and significant software support. While TrueNorth has a similar combination of hardware and programming support, its inter-neuron connectivity capability is relatively limited; Loihi approaches the human-scale connectivity density of interest to our research. SpiNNaker has similar capabilities, but is constructed of standard CPU hardware. Loihi's capabilities on the other hand, are built-in on a chip, thus forcing us to explore new programming paradigms. And recent and current state of the art hybrid analog-digital platforms, like Neurogrid (Benjamin et al., 2014), Braindrop (Neckar et al., 2019), DYNAP-SE2 (Moradi et al., 2017), and BrainScaleS(2) (Pehle et al., 2022) are beyond the scope of this manuscript. However, we believe that the simulation and programming paradigms developed on the Loihi platform can generalize to these analog platforms as well, and thus decrease the development time on these unfamiliar architectures.

We present one of our main motivations for this project in **Figure 1**, which highlights Loihi's advantage in performance when compared to standard simulations. Overall, our initial implementation indicates that Loihi is quite efficient in terms of compute-time in context of large brain network simulations and thus shapes our central motivation for this work (see **Figure 1** and **Table 3**). This manuscript mainly focuses on the trade-offs necessitated by these implementations, that is, how precise are the Loihi simulations when validated against BMTK simulations, given their very different hardware and programming architectures?

As a starting point, we focus on a class of neural network building blocks: point neuronal models as used in large AIBS simulations of biological neural networks. We do so because the Generalized Leaky Integrate and Fire Models (GLIFs, Teeter et al., 2018) have been found to be appropriate for reproducing cellular data under standardized physiological conditions. The data used for this study is made available by the AIBS (AIBS, 2020).

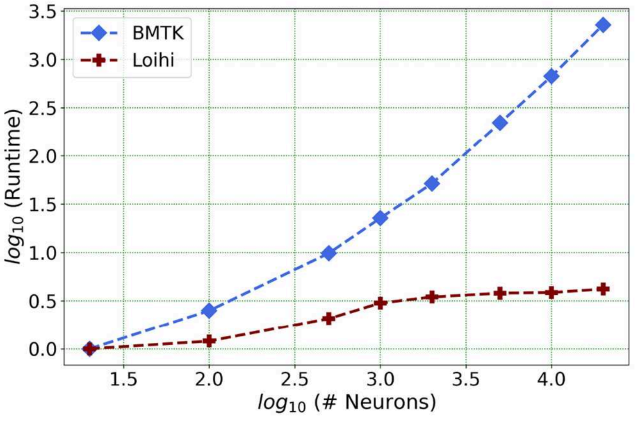


FIGURE 1 | As the network size increases, Loihi outperforms consistently in terms of time. The figure shows runtime comparison of 500 *ms* of dynamics for up to 20,000 neurons for Loihi and BMTK, with the values scaled by the respective smallest runtime. Loihi has a maximum runtime of up to 12 ms, whereas BMTK runtime goes up to 273 s (See **Table 3** for the explicit runtime values and Section 4.4 for further details about the network.).

The paper is organized as follows. In Section 2, we describe in detail the features of Loihi and the differences between the neuromorphic and classical hardware that form the basis for this study. Section 3 explains the implementation of the continuous LIF equation on classical computational architecture using BMTK vs. the discrete Loihi setting. Also, we list the validation methods and the cost function that is used to draw comparisons between the implementations. In Section 4, we list out and explain the various results leading to a qualitative and quantitative assessment between the two platforms based in part on methods from Gutzen et al. (2018). Finally, Section 5 lays the ground for future work with expected improvements based on the second generation of the Loihi chip, Loihi 2 (Intel, 2022).

2. COMPARISON BETWEEN CLASSICAL AND NEUROMORPHIC PLATFORMS

At present, various simulators are available for implementing spiking neural networks (Brette et al., 2008). In this section, we lay out the details of the mathematical model and the platforms we use for our work. For the classical simulation, we use the Brain Modeling Toolkit (BMTK) (Dai et al., 2020) developed by the AIBS. Being open source, these resources enable us to experiment with a varied range of data and thus support our extensive validation of neuronal models in Loihi. Intel's fifth-generation chip Loihi provides us with the tools to implement and test out the various neuromorphic features. The output provided by Loihi simulations is then compared to the output of classical simulations implemented in BMTK.

2.1. The Brain Modeling Toolkit (BMTK)

The BMTK is a python-based software package for creating and simulating large-scale neural networks. It supports models of different resolutions, namely, Biophysical Models, Point Models, Filter Models, and Population Models along with the use of the rich data sets of the Allen Cells Database (Lein et al., 2007; AIBS, 2020). It leverages the modeling file format SONATA which includes details on cell, connectivity and activity properties of a network along with being compatible with the neurophysiology data format Neurodata Without Borders (NWB), thus allowing easy access to a vast repertoire of experimental data.

In this study, we work with the Point Neuron Models with simulations supported by the BMTK module PointNet *via* NEST 2.16 (Kunkel et al., 2017; Linssen et al., 2018). For analysis and visualization, we use the HDF5 output format, underlying both SONATA and NWB's spike and time series storage.

The classical BMTK simulation are instantiated and run on a single node of Kamiak, the institutional high performance computing cluster. A typical Kamiak node contains 2 Intel Xeon E5-2660 v3 CPUs at 2.60 GHz, with 20 cores and 128–256 GB RAM (CIRC and WSU, 2021).

2.2. Loihi

Neuromorphic hardware inspired by the structure and functionality of the brain, envisioned to provide advantages such as low power consumption, high fault tolerance and massive parallelism for the next generation of computers, is called neuromorphic hardware. Toward the end of 2017, Intel Corporation unveiled its experimental neuromorphic chip called Loihi. We provide a summary of the platform here.

As of its 2020 rendition, the version on which these results were evaluated, Loihi is a 60-mm^2 chip that implements 131,072 leaky-integrate-and-fire neurons. According to Davies et al. (2018), it uses an asynchronous spiking neural network (SNN), comprising of 128 neuromorphic cores, each with 1,024 neural computational units; 3 × 86 cores; along with several off-chip communication interfaces that provide connectivity to other chips. As Loihi advances the modeling of SNNs in silicon, it comprises of a large number of features necessary for their implementation viz., hierarchical connectivity, dendritic compartments, synaptic delays and synaptic learning rules. Each neuron is represented as a compartment in the Loihi architecture, i.e., it is designed to resemble an actual biological neuron model comprising of all the functional units (**Figure 2**). The SYNAPSE unit processes all the incoming spikes from the previous compartment/neuron and captures the synaptic weight from the memory. The DENDRITE unit updates the different state variables. The AXON unit generates the spike message to be carried ahead by the fan out cores. The LEARNING unit updates the synaptic weights based on a learning rule and is not used in this project.

The aim of this study is to establish the groundwork required to execute an ambitious plan of simulating about ∼250,000 neurons with ∼500M synapses in the future, which encapsulates much of the experimentally observed dynamics in the mouse visual cortex available to the AIBS, thus providing a close functional replica of the mouse visual cortex. Loihi's specialized hardware features hold promise for a real-time, low-powered version of such an implementation.

2.3. Leaky Integrate and Fire Model (LIF)

A typical neuron consists of a soma, dendrites, and a single axon. Neurons send signals along an axon to a dendrite through junctions called synapses. The classical Leaky Integrate and Fire (LIF) equation (Gerstner and Kistler, 2002) is a point neuron model which reduces much of the neural geometry and dynamics in order to achieve computational efficiency. It is one of the simplest and rather efficient representations of the dynamics of the neuron, while still providing reasonable approximation of biological neural dynamics for *some* classes of neurons (Teeter et al., 2018). It is stated mathematically as:

$$V'(t) = \frac{1}{C}\left[I_e(t) - \frac{1}{R}(V(t) - E_L)\right] \tag{1}$$

$$V(t) \leftarrow V_r, \quad \text{if} \quad V(t) > \Theta \tag{2}$$

where,

$$\begin{aligned} V(t) &= \text{membrane potential (state)} \\ C &= \text{membrane capacitance (parameter)} \\ R &= \text{membrane resistance (parameter)} \\ E_L &= \text{resting potential (parameter)} \\ I_e &= \text{trans-membrane current (control and state)} \\ V_r &= \text{reset membrane potential} \\ \Theta &= \text{firing threshold} \end{aligned}$$

Here, $' = d/dt$, t is time in *ms*, the membrane potential $V(t)$ of the neuron is in mV. These specific physical units are followed based on what the AIBS datasets use to define the respective physiology measurements. A LIF neuron fires when $V(t) > \Theta$, i.e., the membrane potential exceeds the firing threshold Θ and subsequently the membrane potential is set to a reset value V_r.

The classical LIF model (point generalized LIF) has been shown to match well the dynamics of some mouse neurons under a variety of conditions (Teeter et al., 2018), as listed in the Allen Cell Types Database (Lein et al., 2007). In addition, this model matches the LIF abstraction in Loihi to some extent (as Loihi uses discrete time discrete state dynamics to emulate the continuous time continuous state dynamics of the model). Thus, we work with this model throughout this study to establish the basis for comparison for the two platforms, determine how closely such a discrete dynamical system can get to simulations of a continuous dynamical system, validate the neuromorphic implementation against the ground truth of a standard implementation, and provide evidence that our neuromorphic platform performs more efficiently.

2.4. Loihi LIF Model

In an SNN, spiking neurons form the primary processing elements. The individual neurons are connected through junctions called synapses and interact with each other through single-bit events called spikes. Each spike train can be represented as a list of event times, e.g., as a sum of Dirac delta functions $\sigma(t) = \sum_i \delta(t - t_i)$ where t_i is the time of the i-th spike.

Since Loihi encapsulates the working of an SNN, one of the computational models it implements is a variation of the LIF

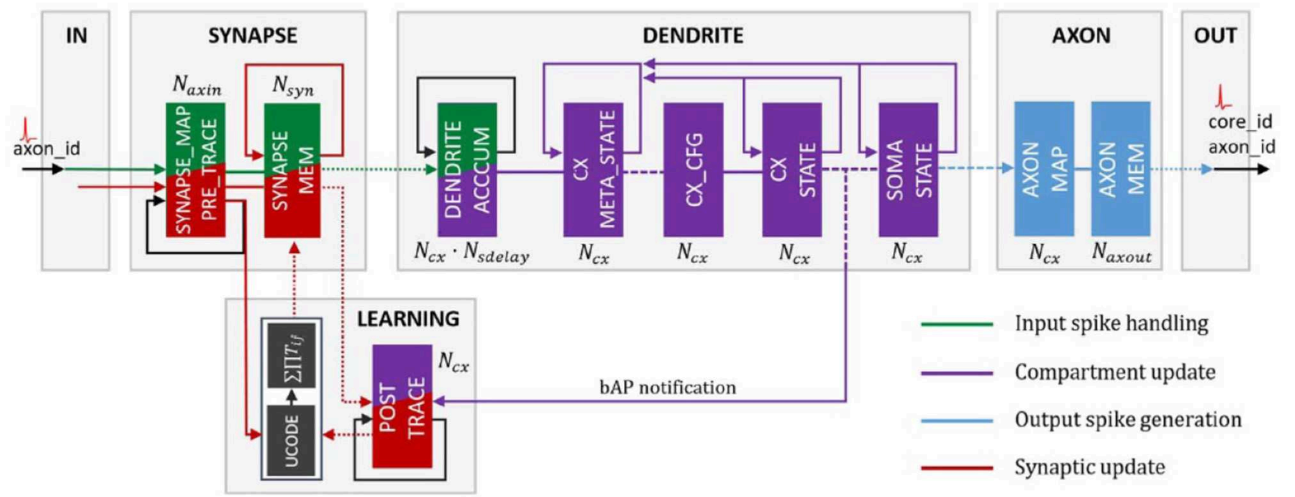


FIGURE 2 | Loihi internal neuron model—Time multiplexed pipeline architecture of a neural unit (Figure 4 in Davies et al., 2018). Reproduced from WikiCommons (2018).

model based on two internal state variables : the synaptic current and the membrane potential (Davies et al., 2018).

$$u(t) = \sum_j w_j(\alpha_j * \sigma_j)(t) + b \tag{3}$$

$$v'(t) = -\frac{1}{\tau_v}v(t) + u(t) \tag{4}$$

$$v(t) \leftarrow 0, \text{ if } \; v(t) > \theta \tag{5}$$

where,

$$\begin{aligned} v(t) &= \text{membrane potential} \\ u(t) &= \text{synaptic current} \\ w &= \text{synaptic weight} \\ \alpha &= \text{synaptic response function} \\ b &= \text{constant bias current} \\ \tau_v &= \text{time constant} \\ \theta &= \text{firing threshold} \end{aligned}$$

A neuron sends out a spike when its membrane potential exceeds its firing threshold θ, i.e., $v(t) > \theta$. After a spike occurs, $v(t)$ is reset to 0. As in the classical LIF model, here $\;' = d/dt$. However, time and membrane potential values here are in arbitrary units.

Loihi follows a fixed-size discrete time-step model, similar to an explicit Euler integration scheme, where the time steps relate to the algorithmic time of the computation. This algorithmic time may differ from the hardware execution time. Moreover, to increase the efficiency of the chip, specific bit-size constraints are imposed on the state variables. We discuss the ones relevant for the LIF model implementation in the following section.

3. METHODS

3.1. Model Setup and Integration

The classical LIF model as represented in Equations (1) and (2) can be rewritten as :

$$V'(t) = -\frac{1}{\tau_v}V(t) + \frac{1}{C}\left[I_e(t) + \frac{1}{R}E_L\right] \tag{6}$$

where $\tau_v = RC$ is membrane time constant of the neuron.

For a non-homogeneous linear differential equation,

$$\frac{df}{dt} = af + g \tag{7}$$

the solution is given by the "variation of constants" method as :

$$f(t) = e^{at}\int_0^t g(s)e^{-as}ds$$

Comparing Equation (6) to Equation (7), we have,

$$\begin{aligned} a &= \frac{1}{\tau_v} \\ f &= V(t) \\ g &= \frac{1}{C}(I_e) + \frac{1}{\tau_v}(E_L) \end{aligned}$$

Here, the postsynaptic current I_e is in the form of an exponent function. However, calculating the above integral at every step i.e., at all grid points $t_i \leq t$ proves to be quite expensive.

BMTK uses NEST as backend to implement the above membrane potential dynamics. To avoid the expensive computations, NEST chooses to use the linear exact integration method (Rotter and Diesmann, 1999), given below as follows :

Equation (6) is rewritten as a multidimensional homogeneous differential equation:

$$\frac{d}{dt}y = Ay \tag{8}$$

where,

$$A = \begin{pmatrix} a_n & a_{n-1} & \cdots & \cdots & a_1 & 0 \\ 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 & 0 \\ 0 & 0 & \ddots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & \frac{1}{C} & \frac{1}{-\tau} \end{pmatrix}$$

The solution is given by :

$$y(t) = e^{At}y_0 \tag{9}$$

$$y_{t+h} = y(t+h) = e^{A(t+h)}y(0) = e^{Ah} \cdot y_t \tag{10}$$

for a fixed time-step h. It saves exorbitant computations since each evaluated step involves multiplication only, and intermediate steps between events do not have to be computed.

3.1.1. Mapping Between BMTK and Loihi Models

In this section, we illustrate the primary step of implementing the BMTK-NEST LIF integration with the Loihi dynamic computational model. Loihi follows a discrete-state, discrete-time computational model, similar to an explicit Euler integration scheme. This allows it more flexibility for integrating non-linear neural model, but, and unlike NEST's exact integration method, Loihi's engine accumulates errors at each time step. The time steps in the Loihi model relate to the algorithmic time of the computation which may differ from hardware execution time. Following the linear exact numerics in the NEST implementation, we implement our model in the Loihi discrete setting using the forward Euler method for guidance, as discussed below.

Step 1

First, we rewrite the standard LIF model in Equation (1) to resemble the Loihi form as given in Equation (4). Since Loihi parameters are unit-less, we introduce a re-scaling parameter V_s, which converts standard physical units used in BMTK to Loihi units.

As we compare Equations (2) and (5), it can be seen that for BMTK the membrane potential reset value is set to V_r whereas it is set to zero for Loihi. To account for that, we shift the BMTK representation by V_r. Thus, the forward transformation from BMTK to Loihi looks as follows :

$$v = (V - V_r)/V_s \tag{11}$$

which produces an inverse transformation, to arrive back at the BMTK values, given by :

$$V = v \cdot V_s + V_r \tag{12}$$

Step 2

Substituting the expression in (12) in (1) and isolating v, we get :

$$V'(t) = \frac{1}{C}\left[I_e(t) - \frac{1}{R}(V(t) - E_L)\right] \qquad |V = vV_s + V_r \implies \tag{13}$$

$$v'(t)V_s = -\frac{1}{RC}(vV_s + V_r - E_L) + \frac{1}{C}I_e(t) \qquad |/V_s \implies \tag{14}$$

$$v'(t) = -\frac{1}{\tau_v}v(t) + \frac{1}{\tau_v}\frac{E_L - V_r}{V_s} + \frac{1}{C}\frac{I_e}{V_s} \tag{15}$$

$$= -\frac{1}{\tau_v}v(t) + u(t) \tag{16}$$

with

$$v(t) = \frac{V(t) - V_r}{V_s}, \tag{17}$$

$$u(t) = \frac{1}{CV_s}I_e(t) + \frac{1}{\tau_v}\frac{E_L - V_r}{V_s}, \tag{18}$$

$$\tau_v = RC, \tag{19}$$

$$\theta = \frac{\Theta - V_r}{V_s} \tag{20}$$

Here, we reintroduce the LIF threshold Θ and the corresponding Loihi threshold θ in Equation (20), which is derived from Θ by the same shift and re-scaling that converted V to v.

To reiterate, Loihi implements the continuous LIF as a discrete finite state machine model (Jin et al., 2008; Mikaitis et al., 2018) implemented in silicon. The actual computation is similar to a forward Euler scheme with some peculiarities reflecting engineering design trade-offs. Specifically, the $v(t)$ state evolves on-chip according to the update rule,

$$v(t+1) = v(t)\left[1 - \frac{\delta_v}{2^{12}}\right] + b + u(t) \tag{21}$$

where δ_v is the membrane potential decay constant and b is the constant bias current listed in Equation (3).

Step 3

Using the forward Euler method :

$$y_{n+1} = y_n + f(t_n, y_n).dt$$

where $y_{n+1} = y(t_{n+1})$ and $t_{n+1} = t_n + dt$ for a fixed time-step dt, we transform the classical LIF model into a form followed in Equation (21). Thus, transforming the LIF model into the discrete form and grouping terms to match the Loihi integration (9) yields the following :

$$\frac{v(t+dt) - v(t)}{dt} = -\frac{1}{\tau_v}v(t) + u(t) \tag{22}$$

$$\implies v(t+dt) = v(t)(1 - \frac{dt}{\tau_v}) + u(t)dt \tag{23}$$

where dt is the fixed time-step with which we can adjust the temporal precision of the Euler integration scheme.

In order to equate the Loihi computation (21) with the Euler scheme (23), we use *dt* with units *ms/Loihi timestep* i.e., 1 BMTK millisecond per Loihi timestep. Thus, comparing Equations (21) and (23) defines the Loihi voltage decay parameter δ_v in terms of the timestep *dt*, i.e.,

$$(2^{12} - \delta_v)\, 2^{-12} = (1 - \frac{dt}{\tau_v}) \tag{24}$$

$$\implies \delta_v = \frac{dt}{\tau_v}\, 2^{12} = \frac{dt}{RC}\, 2^{12} \tag{25}$$

3.1.2. Bit Constraints

Given its discrete setting, there are specific bit-size constraints that Loihi imposes on the state variables and parameters. State variables—membrane potential and current—are allotted ±23 bits each. The membrane potential decay constant δ_v is allotted 12 bits and the membrane potential threshold is assigned 17 bits interpreted as the 17 high bits of a 23 bit word to match the state variables size. Details on other parameters can be found under Table 1 in Davies et al. (2018) and Table 6 in Michaelis et al. (2021).

3.2. Validation Methods

3.2.1. Data

The data used here is provided by the Allen Mouse Brain Atlas (Lein et al., 2007; AIBS, 2020), which is a survey of single cells from the mouse brain, obtained *via* intracellular electrophysiological recordings done through a highly standardized process. We focus on neurons of different types with available GLIF parameters. The data used can be accessed in the Allen Cell Types Database. Our LIF model is implemented and simulated on BMTK based on this data, and these simulations form the ground truth for validating the Loihi implementations.

The datasets used for the simulations in this work can be found in our Github repository (Dey, 2022).

3.2.2. Cost Functions

To quantify the error between the BMTK and Loihi membrane potential values, we use two related cost functions: the Root Mean Square Error (RMSE) and the Pearson correlation coefficient (r) with values as follows :

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_L^i - y_B^i\right)^2} \tag{26}$$

where,

$$\begin{aligned} i &= \text{index of data point} \\ y_L &= \text{transformed Loihi values} \\ y_B &= \text{original BMTK values} \\ n &= \text{number of data points} \end{aligned}$$

and

$$r = \frac{\sum_{i=1}^{n}\left(y_L^i - \bar{y_L}\right)\left(y_B^i - \bar{y_B}\right)}{\sqrt{\sum_{i=1}^{n}\left(y_L^i - \bar{y_L}\right)^2 \sum_{i=1}^{n}\left(y_B^i - \bar{y_B}\right)^2}} \tag{27}$$

where,

$$\begin{aligned} \bar{y_L} &= \text{mean of the transformed Loihi values} \\ \bar{y_B} &= \text{mean of the original BMTK values} \end{aligned}$$

3.2.3. Other Methods

Since BMTK and Loihi run on two different computing environments, visual comparisons in graphs are helpful for diagnostics of discrepancies that may be obscured in the single numbers reported by the cost function. They also contribute to assess the level of similarity between the two implementations.

We compare the simulation dynamics for both implementations based on the following:

- **Distribution Function:** We compare the distributions of attained state values in the two cases. We use density plot as a representation of those distributions, thus allowing us to compare the two implementations in terms of concentration and spread of the values and provide a basis for comparing the collective dynamics of the implementations.
- **Raster Plot:** We evaluate the membrane potential response at each time-step. The X-axis represents the membrane potential and the Y-axis represents the time-step. Raster plot helps to visually communicate similarities between the BMTK and Loihi states, and highlight potential state-localized difference in the dynamics at each step which may otherwise be lost in the average error measures.
- **Scatter Plot:** For examining association between the two implementations, we use color-coded scatter plots identifying the correlation relationships. We add a trend line to illustrate the strength of the relationship and pin down the outliers to improve the simulation results. Since we anticipate an almost perfect linear relationship, we quantify the match with its Pearson correlation coefficient.

4. RESULTS

In order to lay the groundwork for simulating a network of over 250,000 neurons with a connectivity of over 500M synapses in the neuromorphic hardware, we begin by ensuring a high quality replication of individual neural and smaller network models. The replication performance here is evaluated based on membrane potential and current responses, the two state variables. We conjecture that securing a good replica for smaller models will ensure that parameters can be calibrated correctly and thus can be carried forward for the bigger networks needed in biological context (Herz et al., 2006; Gutzen et al., 2018; Trensch et al., 2018).

We begin our work on a single-neuron network[1] sub-threshold dynamics driven by both bias current and external spikes to ensure Loihi is able to handle both stimuli efficiently. Our test suite consists of LIF models based on 20 different parameter sets. We perform rigorous analysis of our results based

[1] A network is the smallest executable structure in Loihi, hence the peculiar term *single-neuron network*.

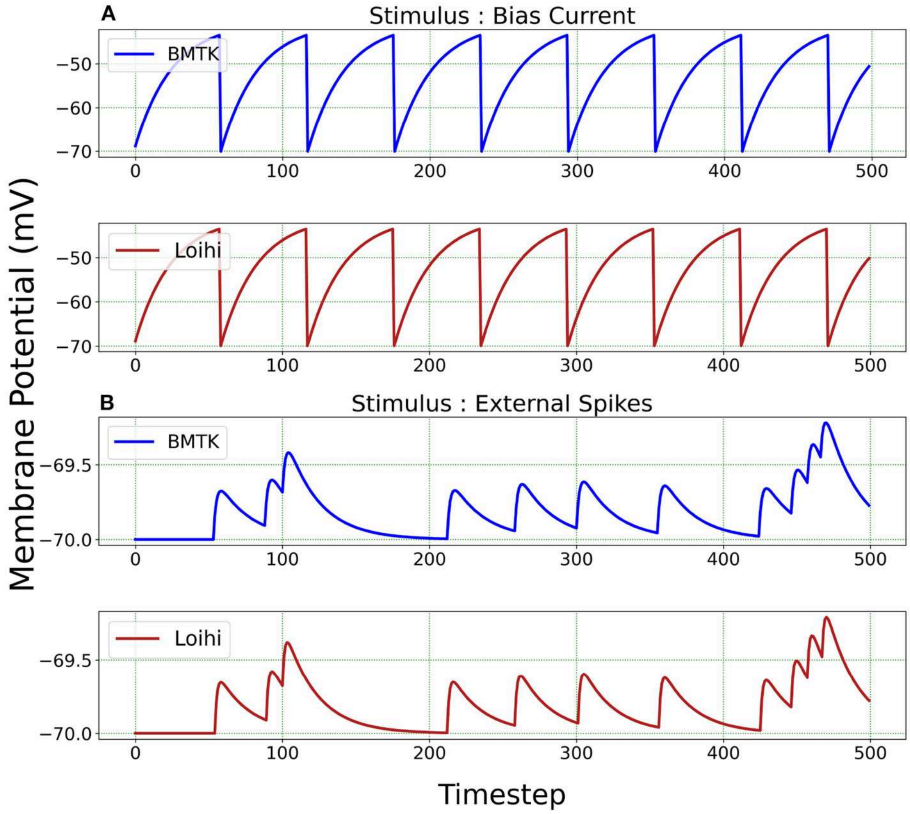


FIGURE 3 | Membrane potential response for single-neuron network based on two different neuron parameters. **(A)** Simulation is driven by bias current. **(B)** Simulation is driven by external spikes.

on various statistical measures and visualizations to demonstrate that we have replication of high quality. It is important to restate here that we test our results based on neurons with different morphologies and biophysics, which attribute to the different parameter sets.

4.1. Simulations of a Single Neuron

We begin by simulating a single-neuron network in BMTK. The simulation is run for 500 *ms*. The classical parameters are translated to Loihi values and the corresponding LIF model is implemented as an one-neuron SNN executed for $500/dt$ time steps in Loihi. The simulations are driven either by bias current or external spikes.

In the Loihi network, neurons are denoted by compartments. The compartment dynamics are hardware-constrained and determined by the parameters *bias current mantissa, membrane potential threshold, membrane potential decay*, and *current decay*. It is worth iterating here that the membrane potential values in Loihi are unit-less as opposed to the BMTK values which are assigned units of millivolts (mV) and milliseconds (ms) based on the AIBS datasets.

We test the precision of our replication, both qualitatively and quantitatively, for all 20 parameters sets and find that the results are consistent with the ones described below. In **Figure 3**, we illustrate the implementations achieved through bias current and external spikes on two different parameter sets (**Table 1**). The remaining 18 parameter sets can be found in our Github page (Dey, 2022). The parameters in the BMTK platform are mapped to Loihi using the transformations described in Section 3.1.1 with respect to the bit constraints described in Section 3.1.2.

TABLE 1 | Parameter set for LIF models.

Parameters	Dataset (1)	Dataset (2)	Units
Membrane time constant	25.0	22.0	*ms*
Membrane potential threshold	−43.0	−43.0	*mV*
Resting potential	−70.0	−70.0	*mV*
Voltage reset	−70.0	−70.0	*mV*
Current	200.0	0.0	*pA*
Membrane capacitance	170.21	170.0	*pF*

It is to be noted here that stimulus bias current acts as one of the parameters of the LIF model and hence is mapped into Loihi according to Equation (18). When stimulating with external spikes as stimulus, we make use of the fixed-time step *dt* that we introduce in Equation (23). Here, the external spike-times are in "*ms*" and we assign unit "ms/Loihi time-step" to *dt*. Thus, the external spike-times are scaled as spike-time/*dt* and then injected into a Loihi neural unit for each time-point, with *dt* guiding the temporal precision scale. **Table 2** shows the external spike-times used in the simulations, which are generated by five spike sources using a random Poisson spike generator with a max firing rate of 5Hz and then frozen to stimulate the different models in both BMTK and Loihi.

For a qualitative comparison, it can be seen from **Figure 3** that Loihi implementations simulate BMTK results very closely. We have close correspondence in terms of spike frequency, spike amplitude, and response values. Since Loihi membrane potential values are unit-less, we map them back to BMTK values (mV, ms) before performing the comparison. The inverse mapping from Loihi to BMTK is performed based on Equation (12), i.e.,

$$V = v \cdot V_s + V_r$$

We perform a quantitative assessment of the replication using RMSE and correlation coefficient between the values obtained from the two platforms. As seen from **Table 3**, the values are highly correlated with a relatively small RMSE.

Figure 4 illustrates the comparison of Loihi implementations against the BMTK implementations for the two different stimuli using various graphing data—(a) Distribution function approximating the membrane potential dynamics, (b) Raster plot of the spiking network activity, (c) Scatter Plot highlighting the positive coefficient between the two implementations. These visualizations help us track discrepancies which might remain unobserved based on single quantitative averages given by the cost function or the correlation coefficient.

TABLE 2 | External spike-time values.

Source	Spike-times (*ms*)
0	446
1	355
2	53, 258, 300, 424, 457
3	88, 466
4	100, 212

TABLE 3 | Correlation and RMSE between BMTK and Loihi membrane potential values.

Stimulus	Correlation	RMSE
Bias current	0.999992	1.1374×10^{-4} mV/*ms*
External spikes	0.999942	4.208×10^{-5} mV/*ms*

4.2. Simulation Using Varied Precision

As already stated, Loihi follows a fixed-size discrete time-step model along with bit-size constraints for the different parameters. Thus, we examine how the numerics of Loihi affect its ability to faithfully implement neuron models. More precisely, we investigate how changing the precision of the time scale and the neuron state values affects the accuracy of the simulations. We explore this property for the two state variables—membrane potential and current.

Figure 5 illustrates the membrane potential and current responses of a single neuron model in BMTK which form the basis of our comparison for the results below.

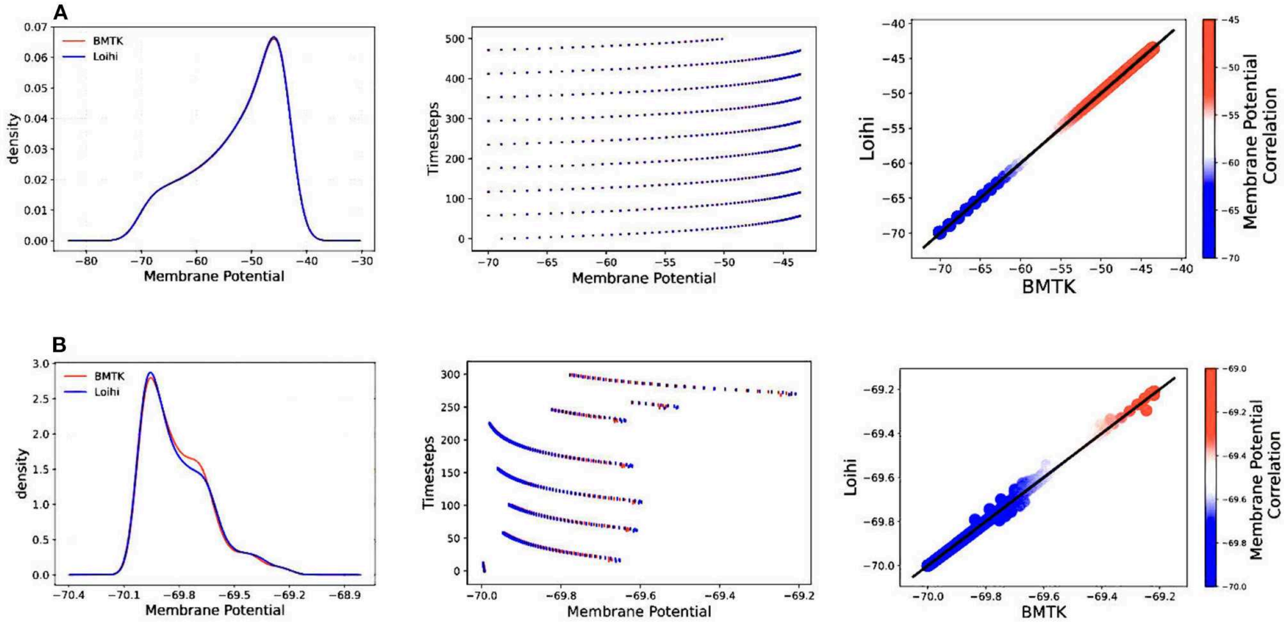


FIGURE 4 | Validation plots for simulations based on two different stimuli—**(A)** Validation plots for bias current stimulus **(B)** Validation plots for external spike stimulus, based on the Distribution Function, Raster Plot, and Scatter Plot, respectively.

4.2.1. Simulation Using Varied Temporal Precision

For Loihi's fixed simulation time-step, we assign different time units to each step and test the corresponding simulation precision. This is achieved through the "dt" parameters available in our equations while transforming the classical LIF model to Loihi neural model. It enables us to experiment with several time units (Hopkins and Furber, 2015). Following Equation (24), the change of a time-step while working with the Loihi neural model necessitates a corresponding variation of the time constant "τ_v" to yield the desired results.

We check the results for $dt = 0.1, 1.0$ and $10.0(ms/timestep)$. As mentioned earlier, we run the simulation for 500 ms, thus the corresponding number of time steps in Loihi for $dt = 0.1$ and $dt = 10.0$ becomes 5000 and 50 respectively, and for $dt = 1.0$ it

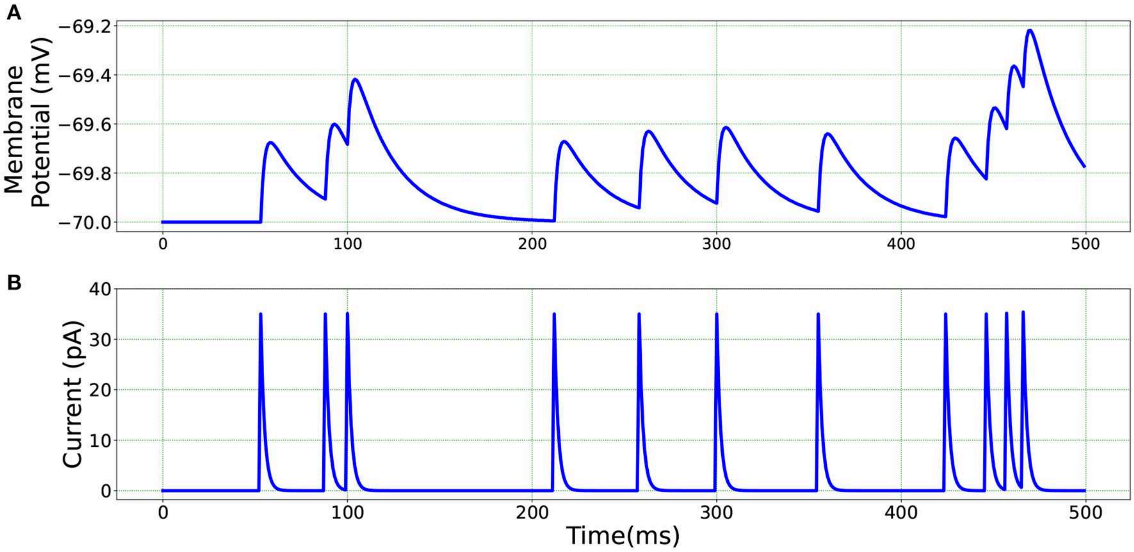


FIGURE 5 | Single neuron model in BMTK—**(A)** Membrane potential response. **(B)** Current response.

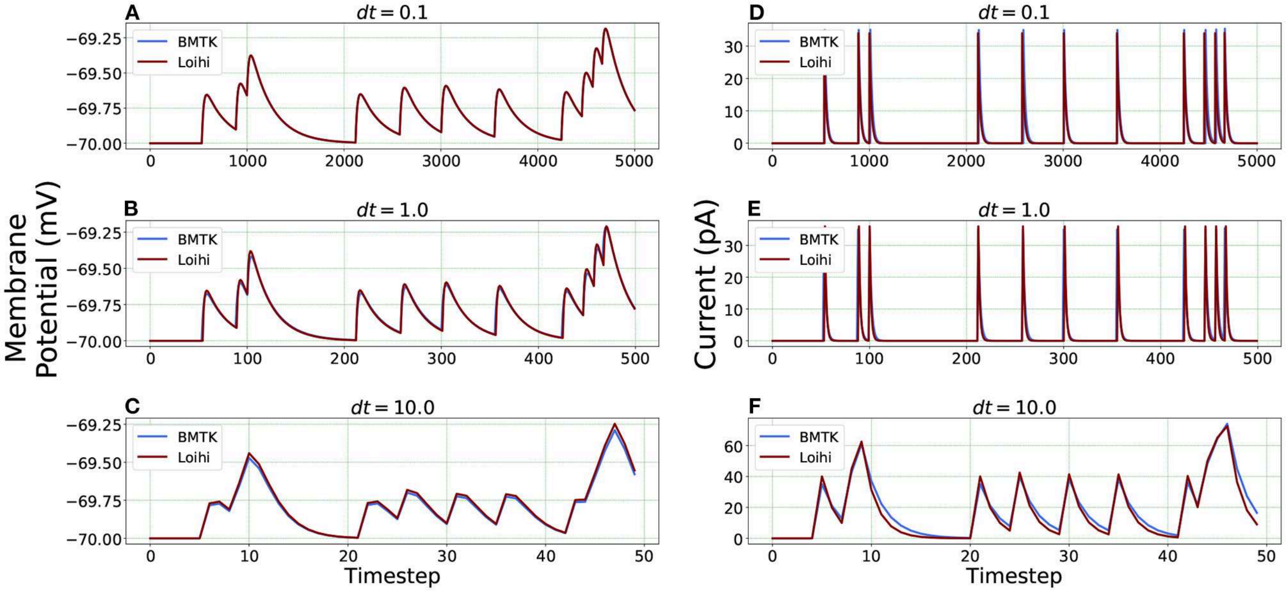


FIGURE 6 | Comparison of membrane potential and current plots with different temporal precisions in Loihi. Membrane potential plots are on the left with **(A)** $dt = 0.1$ **(B)** $dt = 1.0$ **(C)** $dt = 10.0$. Current plots are on the right with **(D)** $dt = 0.1$ **(E)** $dt = 1.0$ **(F)** $dt = 10.0$. For $dt = 10.0$, number of time-steps are 50 and for $dt = 0.1$, number of time-steps are 5,000.

remains at 500, (i.e., $500/dt$ for each dt). **Figure 6** illustrates the related Loihi simulation for membrane potential and current.

Error comparison for temporal precision Unlike continuous analysis in which the error decreases monotonically with dt, Loihi's discrete-time, discrete-state simulation dynamics suggests that there may be an optimal dt which minimizes the LIF dynamics error.

We compare the simulations in Loihi with different temporal precisions against the simulations in BMTK. We calculate the RMSE to be able to deduce the result. As can be seen from **Figure 7**, the error is lowest when 1 *ms* of simulation time in BMTK equates to 1 time-step in Loihi for membrane potential and current. Thus, for the LIF model simulations, representing 1*ms* with a Loihi hardware time step provides the best match between the two simulations. As to using larger dt for efficiency, panels (C) and (F) clearly show that large time steps (larger than the synaptic time constant in this case) significantly degrade the quality of simulations.

It should be noted that the selected simulation timestep dt affects the range of physical time constants τ_v that can be represented in Loihi. Since $\delta_v = \frac{dt}{\tau_v} 2^{12}$ (Equation 24), then $\tau_v = \frac{dt}{\delta_v} 2^{12}$. In Loihi, $\delta_v \in [1, 2^{12}]$ (stored as a 12-bit word,

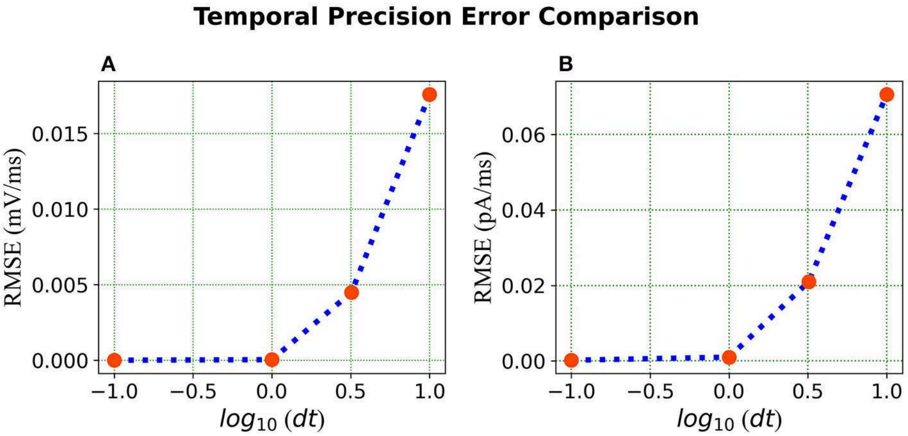


FIGURE 7 | Error comparison for different temporal precisions—**(A)** Membrane potential error. **(B)** Current error. In both panels, the RMSE for the corresponding state is plotted against the *log* of the temporal precision *dt*.

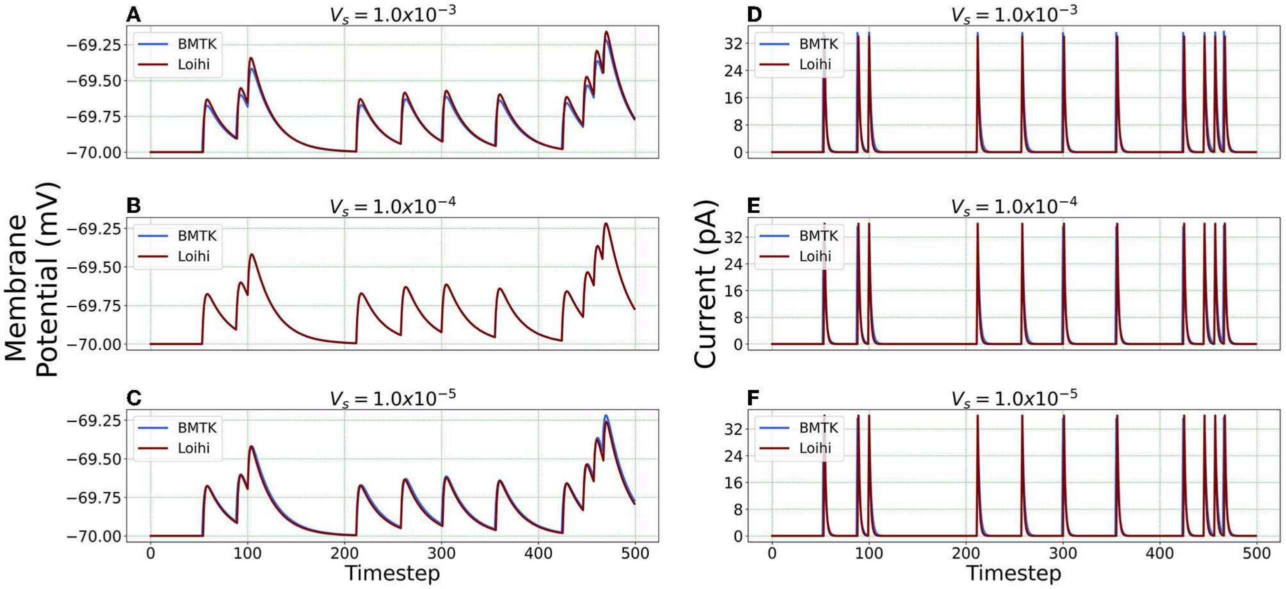


FIGURE 8 | Comparison of membrane potential and current plots with different voltage precisions. Membrane potential plots are on the left with **(A)** $V_s = 1.0 \times 10^{-3}$ **(B)** $V_s = 1.0 \times 10^{-4}$ **(C)** $V_s = 1.0 \times 10^{-5}$. Current plots are on the right with **(D)** $V_s = 1.0 \times 10^{-3}$ **(E)** $V_s = 1.0 \times 10^{-4}$ **(F)** $V_s = 1.0 \times 10^{-5}$.

with 0 representing $\delta_v = 2^{12}$). Hence, $\tau_v \in [1, 2^{12}]dt$, that is, the highest physical time constant that can be represented is $\tau_v = 2^{12}dt \approx 4,100dt$. This is not a big constraint for $dt = 1ms$, but e.g., a higher simulation precision of $dt = 0.01ms$ can be performed only for neurons with time constants $\tau_v < 41ms$, which already excludes some models found in the Allen Institute's Cell Types Database. This shortcoming of this Loihi 1 platform is being addressed by Intel in subsequent hardware like Loihi 2 (Intel, 2022), and the new Lava SDK. Similarly affected are potential spike propagation delays (not used here). Loihi supports ranges from 1 to 62 time steps, which translate to dt to $62dtms$ of physical time. This is a minor constraint for $dt = 1ms$, but quickly becomes a significant constraint for short dt.

4.2.2. Simulation Using Varied Voltage Precision

We repeat the precision study by changing the voltage precision values using the re-scaling parameter V_s. To check different precision results, we try 1K/mV, 10K/mV and 100K/mV (state level/mV) by using $V_s = 1.0 \times 10^{-3}, 1.0 \times 10^{-4}$ and 1.0×10^{-5}

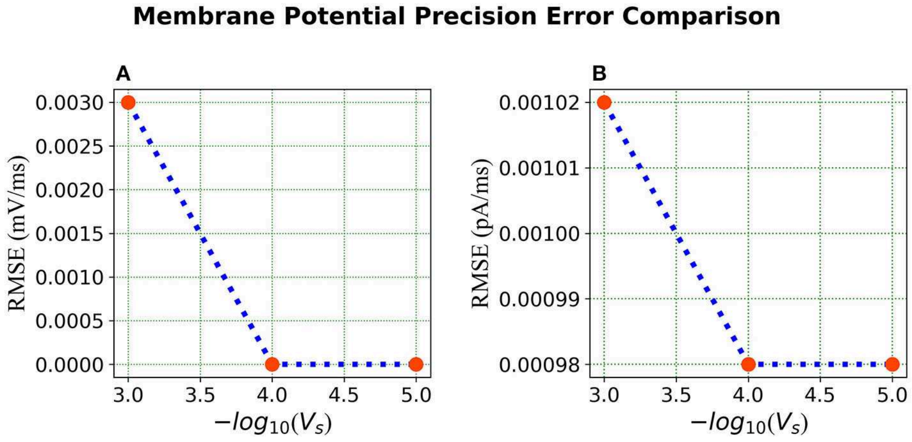


FIGURE 9 | Error comparison for different membrane potential precisions—**(A)** Membrane potential error. **(B)** Current error. In both panels, the RMSE for the corresponding state is plotted against the *–log* of the voltage scale V_s.

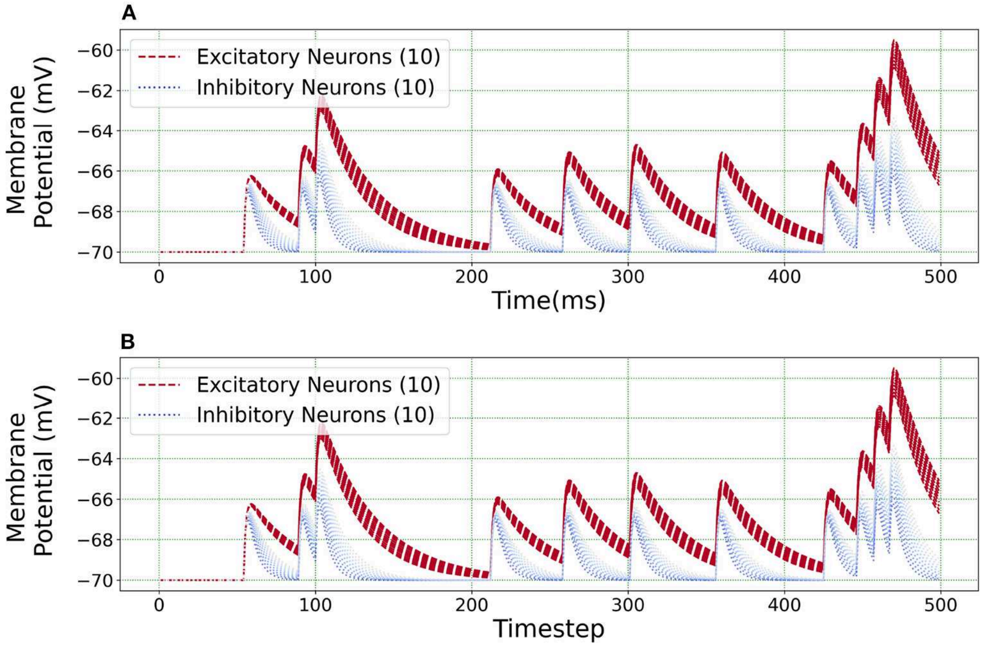


FIGURE 10 | Loihi replicates various neuron class responses of BMTK. **(A)** BMTK simulation of 20 neuron classes. **(B)** Loihi simulation of 20 neuron classes.

respectively. **Figure 8** illustrates the neuron state simulations based on different voltage precisions.

Error Comparison for Voltage Precision As can be seen from **Figure 9**, for membrane potential—error decreases significantly as the precision increases from $V_s = 1.0 \times 10^{-3}$ to $V_s = 1.0 \times 10^{-4}$. However, the error is extremely small for the current simulation and remains the same for $V_s = 1.0 \times 10^{-4}$ and $V_s = 1.0 \times 10^{-5}$.

4.2.3. Effects of State Precision on Simulations

In conclusion, the effect of precision on both scales depends on the model parameters and the information needing to be preserved. However, there are important performance differences. Increased voltage precision is essentially free, as it does not tax the hardware resources any further, and the sole risk is from computation overflow in cases of the Loihi voltage state nearing the capacity of the voltage register. Increased time precision on the other hand has two important drawbacks: it increases simulation time (proportionately to increased precision), and it decreases the range of voltage decay timescales that can be represented (again, proportionately to increased precision). Thus, the choice of simulation time step and corresponding precision should be weighed against these tradeoffs.

4.3. Simulation of Different Neuron Classes

After establishing and verifying the calibrated Loihi parameters for a single neuron, we extend our simulation to an ensemble of neurons comprising of different neuron classes with varying parameters.

Figure 10 illustrates an equivalent simulation for 20 different neuron classes between BMTK and Loihi indicating that Loihi is capable of emulating BMTK results in spite of varying parameters. Here, we found an average correlation of 0.99985 with an RMSE of 0.57 $\times 10^{-4}$ mV/ms (**Table 4**). This also validates the fact that the calibration of parameters for a single neuron done earlier is valid.

The scatter plots in **Figure 11** capture the range of the parameters—**Figure 11(A)** C_m vs. τ_v and **Figure 11(B)** I_{bias} vs. τ_v, in the (E)xcitatory and (I)nhibitory classes used for the simulations. The size of the markers represents RMSE errors for those models, with ranges as indicated on the legend. This lays the foundation for building more complicated networks encompassing different neuron classes.

We reiterate here that Loihi imposes certain bit constraints on the parameters. For instance, membrane potential threshold ranges from 0 to $\pm\ 2^{23}$, membrane time constant allows 0 to 2^{12} bits. The membrane capacitance is integrated with bias current (Equation 18) with bias mantissa allowed a range between $[-2^{12}, 2^{12}]$ and bias exponent a range between [0, 7]. Thus, a good range of parameters can be mapped well into Loihi and a limit to the "exactness" can be attributed to the low-fixed-precision nature of Loihi as most state and configuration variables are in the range of 8–24 bits.

TABLE 4 | Correlation and RMSE for different neuron classes.

Neuron class	Correlation	RMSE
Excitatory	0.999989	0.532×10^{-4} mV/ms
Inhibitory	0.999982	0.612×10^{-4} mV/ms

5. CONCLUSION AND FUTURE WORK

Inspired by the brain, neuromorphic computing holds great potential in tackling tasks with extremely low power and high efficiency. Many large-scale efforts including the TrueNorth, SpiNNaker and BrainScaleS have been demonstrated as a tool for neural simulations, each replete with its own strengths

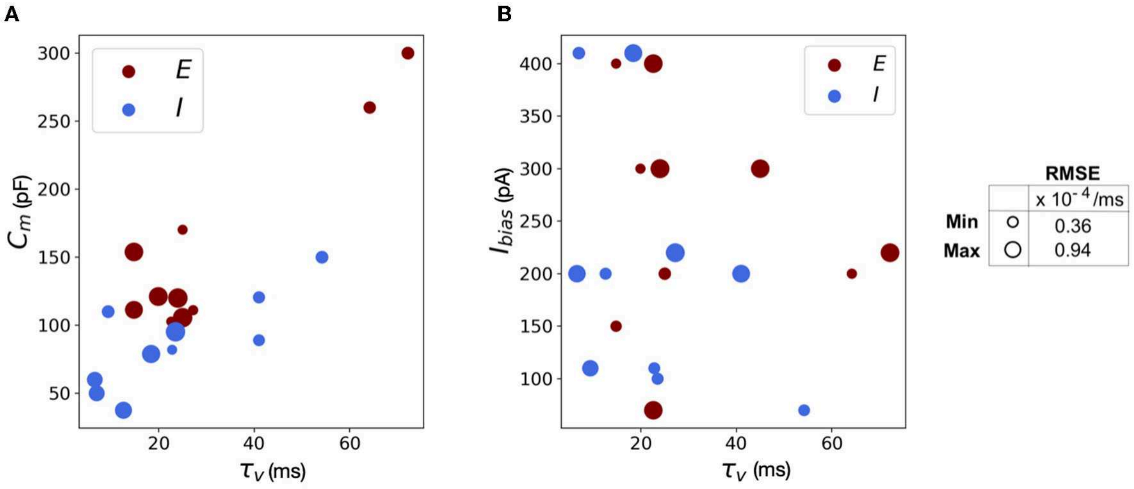


FIGURE 11 | Scatter plots showing the range of parameters for the 20 neurons classes comprising of both excitatory and inhibitory neurons grouped by RMSE of the simulations. **(A)** Scatter plot for membrane capacitance (C_m) vs. membrane time constant (τ_v). **(B)** Scatter plot for bias current (I_{bias}) vs. membrane time constant (τ_v). The marker size is determined by the corresponding RMSE.

and constraints. Fabricated with Intel's 14 nm technology, Loihi is a forward-looking and continuously evolving state-of-the-art architecture for modeling spiking neural networks in silicon. As opposed to its predecessors, Loihi encompasses a wide range of novel features such as hierarchical connectivity, dendritic compartments, synaptic delays and programming synaptic learning rule. These features, together with solid SDK support by Intel, and a growing research community, make Loihi an effective platform to explore a wealth of neuromorphic features in more detail than before.

In this work, we have demonstrated that Loihi is capable of replicating the continuous dynamics of point neuronal models with high degree of precision and does so with much greater efficiency in terms of time and energy. The work comes with its challenges as simulations built on the conventional chips cannot be trivially mapped to the neuromorphic platform as its architecture differs remarkably from the conventional hardware. Classical simulations from the Brain Modeling Toolkit (BMTK) developed by the Allen Institute of Brain Science (AIBS) serves as the foundation of our neuromorphic validation.

For comparison between the conventional and the neuromorphic platforms, we use both qualitative and quantitative measures. It can be seen that Loihi replicates BMTK very closely in terms of both membrane potential and current, the two state variables on which the Loihi LIF model evolves. We use different validation methods and quantitative measures to assess the equivalence and identify sets of parameters which maximize precision while retaining high performance levels. Furthermore, simulation results indicate Loihi is highly efficient in terms of speed and scalability as compared to BMTK.

This work demonstrates that classical simulations based on Generalized Leaky Integrate-and-Fire (GLIF) point neuronal models can be successfully replicated on Loihi with a reasonable degree of precision.

Our future work is motivated by runtime performance comparisons for larger networks between the two platforms. As Loihi and BMTK are based on very different hardware systems that follow distinct dynamics and network-setup regimes, we use the runtime of the simulations to compare the performance of these implementations. As has been mentioned in the introduction, performance of Loihi far exceeds that of BMTK. **Figure 12** compares the runtime of Loihi and BMTK, for running a network of randomly connected neurons with the same parameters. The network consists of excitatory and inhibitory neurons in a 1:1 ratio driven by bias current, with connection probability set at 0.1.

As can be seen from **Figure 12** and **Table 5**, Loihi easily scales up to larger network sizes with a minuscule rise in runtime

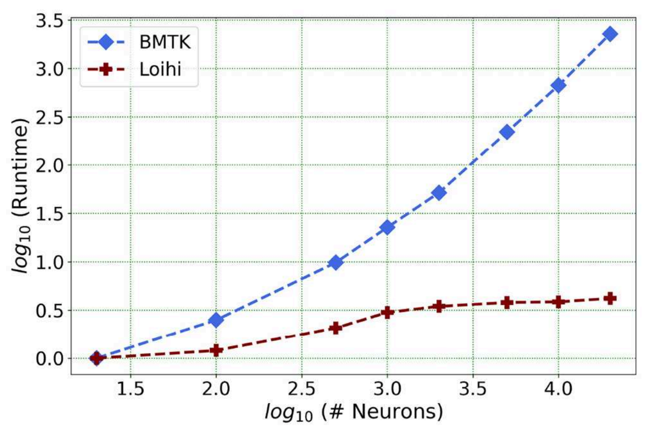


FIGURE 12 | Performance comparison between BMTK and Loihi for network sizes ranging from 1 to 20,000 for the simulation of 500 *ms* of dynamics. The values for each curve are scaled by the respective smallest runtime. The Loihi runtime units are in "*milliseconds*" and BMTK runtime is in "*seconds*".

TABLE 5 | Simulation runtime in Loihi and BMTK.

Network size	Loihi time (ms)	BMTK time (s)
20	2.52	0.12
100	3.03	0.3
500	5.21	1.13
1,000	7.56	2.72
5,000	9.57	26.47
10,000	9.73	80.45

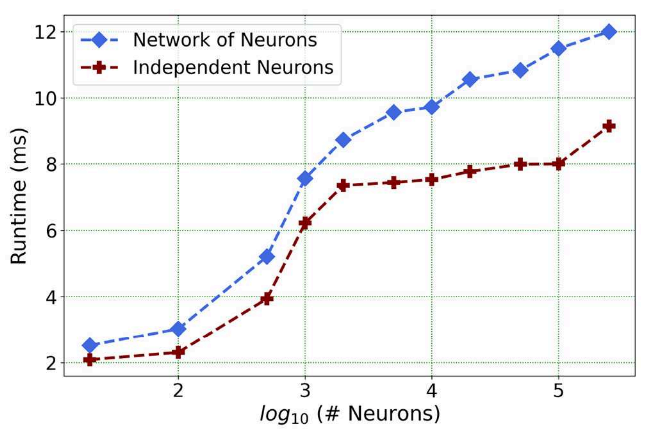


FIGURE 13 | Loihi runtime for a network of upto 250K neurons for the simulation of 500 *ms* of dynamics.

TABLE 6 | Simulation runtime in Loihi for independent neurons vs. connected network.

Network size	Connected network (ms)	Independent neurons (ms)
20	2.52	2.09
100	3.03	2.31
500	5.21	3.94
1,000	7.56	6.22
5,000	9.57	7.35
10,000	9.73	7.53
50,000	10.84	7.98
100,000	11.49	8.00
250,000	11.98	9.16

whereas for BMTK the increase is quite rapid. While both seem to exhibit a power-law scaling (string line on this graph), Loihi's scaling power is much smaller. It is also worth noting here that for Loihi the unit for the runtime are in "*milliseconds*" whereas for BMTK they are in "*seconds*". Here we stop at 20,000 neurons as it can be inferred from the graph that increasing the network size would increase the time cost for BMTK substantially.

Furthermore, following the above outcome, we extend our network size *in Loihi only* to 250K neurons in order to investigate what potential Loihi holds to execute the final goal of simulating about ∼250,000 neurons with ∼500M synapses in the future, a simulation scale comprising much of the experimentally observed dynamics in the mouse visual cortex available to the AIBS. We record our observations for a randomly connected network of neurons as well as an independent set of unconnected neurons. From **Figure 13** and **Table 6**, we can infer that the runtime remains consistent with the above result, with the independent set of neurons completing the simulation marginally faster.

This shows that Loihi performs well for connected networks, setting the stage for our main aim for neural simulations. Additionally, it also works well for independent set of neurons which contribute to solutions of problems that require on-chip parameter and meta-parameter searches, e.g., for Evolutionary Programming (Schuman et al., 2020).

We do not asses the state-based cost for these networks as their large sizes require multi-chip simulations which we expect to be better supported on Loihi 2 (Intel, 2022). Furthermore, other research groups have firmly established that we cannot expect exact replication of subthreshold network states between simulators except for few very simple small networks (van Albada et al., 2018; Crook et al., 2020). Thus, on the network level we need to develop cost functions that capture appropriate network activity details on different scales (e.g., average spike rates and correlations on the coarsest levels, as in van Albada et al., 2018).

In closing, we want to highlight that with the advent of Loihi 2 (Intel, 2022), we aim to address the limitations of the larger networks and carry out the next steps of our work in this new hardware. We are planning to investigate the full GLIF dynamics as we would have better support for more complex network topology and spiking dynamics. In addition, we hope to implement a connected network of 250K neurons with specific synaptic variables as available in the AIBS dataset. We also plan to investigate the control and performance of temporal precision choices. Till date, our limited conclusion for these cases is that ∼ 1*ms* timestep is sufficient. This need not generalize to networks in which other precision may be needed, with corresponding tradeoffs to changes in the parameters. We intend to explore this question further. Lastly, we are working on performing a sensitivity analysis on the GLIF parameters to assess the robustness of the model.

DATA AVAILABILITY STATEMENT

The original contribution in the study are included in the article. The datasets used are available in https://github.com/srijanie03/bmtk_loihi_utils. Further inquiries can be directed to the corresponding author/s.

AUTHOR CONTRIBUTIONS

SD and AD contributed to conception and design of the study. SD performed the simulations and analysis and wrote the first draft of the manuscript. AD wrote parts of the manuscript, edited, and reviewed. All authors contributed to manuscript revision, read, and approved the submitted version.

FUNDING

This work received support from WSU Vancouver Office of Research and Graduate Education to SD.

ACKNOWLEDGMENTS

We thank Dr. Kael Dai for his helpful suggestions and advice regarding the use of BMTK. We thank Intel for providing us with access to Loihi and the Intel technical support team for helpful feedback on technical issues. This work used resources from the Center for Institutional Research Computing at Washington State University (CIRC and WSU, 2021).

REFERENCES

AIBS (2020). *Allen Brain Atlas*. Available online at: https://celltypes.brain-map.org/ (accessed February 22, 2022).

Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., et al. (2015). Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Trans. Comput. Aided Design Integr. Circ. Syst*. 34, 1537–1557. doi: 10.1109/TCAD.2015.2474396

Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J.-M., et al. (2014). Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* 102, 699–716. doi: 10.1109/JPROC.2014.2313565

Bhuiyan, M. A., Nallamuthu, A., Smith, M. C., and Pallipuram, V. K. (2010). "Optimization and performance study of large-scale biological networks for reconfigurable computing," in *2010 Fourth International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)* (New Orleans, LA), 1–9. doi: 10.1109/HPRCTA.2010.5670796

Brette, R., Lilith, M., Carnevale, T., Hines, M., Beeman, D., Bower, J., et al. (2008). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci*. 23, 349–398. doi: 10.1007/s10827-007-0038-6

CIRC and WSU (2021). *What Is Kamiak*? Available online at: https://hpc.wsu.edu/kamiak-hpc/what-is-kamiak/ (accessed February 22, 2022).

Crook, S. M., Davison, A. P., McDougal, R. A., and Plesser, H. E. (2020). Editorial: reproducibility and rigour in computational neuroscience. *Front. Neuroinform*. 14, 23. doi: 10.3389/fninf.2020.00023

Dai, K., Gratiy, S. L., Billeh, Y. N., Xu, R., Cai, B., Cain, N., et al. (2020). Brain modeling toolkit: an open source software suite for multiscale modeling of brain circuits. *PLoS Comput. Biol*. 16, e1008386. doi: 10.1371/journal.pcbi.1008386

Davies, M., Srinivasa, N., Lin, T. H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

DeBole, M., Taba, B., Amir, A., Akopyan, F., Andreopoulos, A., Risk, W., et al. (2019). Truenorth: Accelerating from zero to 64 million neurons in 10 years. *Computer* 52, 20–29. doi: 10.1109/MC.2019.2903009

Dey, S. (2022). *BMTK-Loihi Data*. Available online at: https://github.com/srijanie03/bmtk_loihi_utils (accessed February 22, 2022).

Gerstner, W., and Kistler, W. M. (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press. doi: 10.1017/CBO9780511815706

Grübl, A., Billaudelle, S., Cramer, B., Karasenko, V., and Schemmel, J. (2020). Verification and design methods for the brainscales neuromorphic hardware system. *J. Signal Process. Syst*. 92, 1277–1292. doi: 10.1007/s11265-020-01558-7

Gutzen, R., von Papen, M., Trensch, G., Quaglio, P., Grün, S., and Denker, M. (2018). Reproducible neural network simulations: statistical methods for model validation on the level of network activity data. *Front. Neuroinform*. 12, 90. doi: 10.3389/fninf.2018.00090

Herz, A. V. M., Gollisch, T., Machens, C. K., and Jaeger, D. (2006). Modeling single-neuron dynamics and computations: a balance of detail and abstraction. *Science* 314, 80–85. doi: 10.1126/science.1127240

Hopkins, M., and Furber, S. (2015). Accuracy and efficiency in fixed-point neural ODE solvers. *Neural Comput*. 27, 2148–2182. doi: 10.1162/NECO_a_00772

Insel, T. R., Landis, S. C., and Collins, F. S. (2013). The NIH BRAIN initiative. *Science* 340, 687–688. doi: 10.1126/science.1239276

Intel (2022). *Intel Lab's Loihi 2 Chip*. Technical report, Intel Corporation.

Jin, X., Furber, S. B., and Woods, J. V. (2008). "Efficient modelling of spiking neural networks on a scalable chip multiprocessor," in *2008 IEEE International Joint Conference on Neural Networks* (Hong Kong: IEEE World Congress on Computational Intelligence), 2812–2819. doi: 10.1109/IJCNN.2008.4634194

Khan, M. M., Lester, D. R., Plana, L. A., Rast, A., Jin, X., Painkras, E., et al. (2008). "Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor," in *2008 IEEE International Joint Conference on Neural Networks* (Hong Kong: IEEE World Congress on Computational Intelligence), 2849–2856. doi: 10.1109/IJCNN.2008.4634199

Knight, J. C., and Furber, S. (2016). Synapse-centric mapping of cortical models to the spinnaker neuromorphic architecture. *Frontiers in Neuroscience* 10. doi: 10.3389/fnins.2016.00420

Kunkel, S., Morrison, A., Weidel, P., Eppler, J. M., Sinha, A., Schenck, W., et al. (2017). *Nest 2.12.10*. Zenodo.

Lein, E., Hawrylycz, M., Ao, N., Ayres, M., Bensinger, A., Bernard, A., et al. (2007). Genome-wide atlas of gene expression in the adult mouse brain. *Nature* 445, 168–176. doi: 10.1038/nature05453

Linssen, C., Lepperød, M. E., Mitchell, J., Pronold, J., Eppler, J. M., Keup, C., et al. (2018). NEST 2.16.0. *Zenodo*. doi: 10.5281/zenodo.1400175

Löhr, M. P. R., Jarvers, C., and Neumann, H. (2020). "Complex neuron dynamics on the IBM truenorth neurosynaptic system," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)* (Genao), 113–117. doi: 10.1109/AICAS48895.2020.9073903

Markram, H., Meier, K., Lippert, T., Grillner, S., Frackowiak, R., Dehaene, S., et al. (2011). Introducing the human brain project. *Proc. Comput. Sci*. 7, 39–42. doi: 10.1016/j.procs.2011.12.015

Michaelis, C., Lehr, A. B., Oed, W., and Tetzlaff, C. (2021). Brian2Loihi: an emulator for the neuromorphic chip loihi using the spiking neural network simulator brian. *arXiv preprint arXiv:2109.12308*. doi: 10.48550/ARXIV.2109.12308

Mikaitis, M., Lester, D., Shang, D., Furber, S., Liu, G., Garside, J., et al. (2018). "Approximate fixed-point elementary function accelerator for the spinnaker-2 neuromorphic chip," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)* (Amherst, MA). doi: 10.1109/ARITH.2018.8464785

Moradi, S., Qiao, N., Stefanini, F., and Indiveri, G. (2017). A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Trans. Biomed. Circ. Syst*. 12, 106–122. doi: 10.1109/TBCAS.2017.2759700

Nandi, A., Chartrand, T., Geit, W. V., Buchin, A., Yao, Z., Lee, S. Y., et al. (2020). Single-neuron models linking electrophysiology, morphology and transcriptomics across cortical cell types. *bioRxiv [Preprints]*. doi: 10.1101/2020.04.09.030239

Nawrocki, R. A., Voyles, R. M., and Shaheen, S. E. (2016). A mini review of neuromorphic architectures and implementations. *IEEE Trans. Electron Dev*. 63, 3819–3829. doi: 10.1109/TED.2016.2598413

Neckar, A., Fok, S., Benjamin, B. V., Stewart, T. C., Oza, N. N., Voelker, A. R., et al. (2019). Braindrop: a mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proc. IEEE* 107, 144–164. doi: 10.1109/JPROC.2018.2881432

Ou, Q.-F., Xiong, B.-S., Yu, L., Wen, J., Wang, L., and Tong, Y. (2020). In-memory logic operations and neuromorphic computing in non-volatile random access memory. *Materials* 13, 3532. doi: 10.3390/ma13163532

Pehle, C., Billaudelle, S., Cramer, B., Kaiser, J., Schreiber, K., Stradmann, Y., et al. (2022). The brainscaleS-2 accelerated neuromorphic system with hybrid plasticity. *Front. Neurosci*. 16, 795876. doi: 10.3389/fnins.2022.795876

Rhodes, O., Peres, L., Rowley, A., Gait, A., Plana, L., Brenninkmeijer, C. Y. A., et al. (2019). Real-time cortical simulation on neuromorphic hardware. *Philos. Trans. Ser. A Math. Phys. Eng. Sci*. 378:20190160. doi: 10.1098/rsta.2019.0160

Rossant, C., Goodman, D., Platkiewicz, J., and Brette, R. (2010). Automatic fitting of spiking neuron models to electrophysiological recordings. *Front. Neuroinform*. 4:2. doi: 10.3389/neuro.11.002.2010

Rotter, S., and Diesmann, M. (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biol. Cybernet*. 81, 381–402. doi: 10.1007/s004220050570

Roy, K., Jaiswal, A., and Panda, P. (2019). Towards spike-based machine intelligence with neuromorphic computing. *Nature* 575, 607–617. doi: 10.1038/s41586-019-1677-2

Schuman, C. D., Mitchell, J. P., Patton, R. M., Potok, T. E., and Plank, J. S. (2020). "Evolutionary optimization for neuromorphic systems," in *Proceedings of the Neuro-inspired Computational Elements Workshop, NICE '20* (New York, NY: Association for Computing Machinery), 1–9. doi: 10.1145/3381755.3381758

Sharp, T., and Furber, S. (2013). "Correctness and performance of the spinnaker architecture," in *The 2013 International Joint Conference on Neural Networks (IJCNN)* (Dallas, TX), 1–8. doi: 10.1109/IJCNN.2013.6706988

Teeter, C., Iyer, R., Menon, V., Gouwens, N., Feng, D., Berg, J., et al. (2018). Generalized leaky integrate-and-fire models classify multiple neuron types. *Nat. Commun*. 9, 709. doi: 10.1038/s41467-017-02717-4

Thakur, C. S., Molin, J. L., Cauwenberghs, G., Indiveri, G., Kumar, K., Qiao, N., et al. (2018). Large-scale neuromorphic spiking array processors: a quest to mimic the brain. *Front. Neurosci*. 12, 891. doi: 10.3389/fnins.2018.00891

Trensch, G., Gutzen, R., Blundell, I., Denker, M., and Morrison, A. (2018). Rigorous neural network simulations: a model substantiation methodology for increasing the correctness of simulation results in the absence of experimental validation data. *Front. Neuroinform*. 12, 81. doi: 10.3389/fninf.2018.00081

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Front. Neurosci*. 12, 291. doi: 10.3389/fnins.2018.00291

Wang, Q., Ding, S.-L., Li, Y., Royall, J., Feng, D., Lesnar, P., et al. (2020). The allen mouse brain common coordinate framework: a 3D reference atlas. *Cell* 181, 936.e20–953.e20. doi: 10.1016/j.cell.2020.04.007

WikiCommons (2018). *Core Top-Level Microarchitecture*. Available online at: https://commons.wikimedia.org/wiki/File:Core_Top-Level_Microarchitecture.png (accessed March 17, 2022).

Winnubst, J., Bas, E., Ferreira, T. A., Wu, Z., Economo, M. N., Edson, P., et al. (2019). Reconstruction of 1,000 projection neurons reveals new cell types and organization of long-range connectivity in the mouse brain. *Cell* 179, 268.e13–281.e13. doi: 10.1016/j.cell.2019.07.042

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

ORIGINAL RESEARCH
published: 02 June 2022
doi: 10.3389/fnins.2022.873935

Benchmarking Neuromorphic Hardware and Its Energy Expenditure

Christoph Ostrau, Christian Klarhorst, Michael Thies and Ulrich Rückert*

Technical Faculty, Bielefeld University, Bielefeld, Germany

We propose and discuss a platform overarching benchmark suite for neuromorphic hardware. This suite covers benchmarks from low-level characterization to high-level application evaluation using benchmark specific metrics. With this rather broad approach we are able to compare various hardware systems including mixed-signal and fully digital neuromorphic architectures. Selected benchmarks are discussed and results for several target platforms are presented revealing characteristic differences between the various systems. Furthermore, a proposed energy model allows to combine benchmark performance metrics with energy efficiency. This model enables the prediction of the energy expenditure of a network on a target system without actually having access to it. To quantify the efficiency gap between neuromorphics and the biological paragon of the human brain, the energy model is used to estimate the energy required for a full brain simulation. This reveals that current neuromorphic systems are at least four orders of magnitude less efficient. It is argued, that even with a modern fabrication process, two to three orders of magnitude are remaining. Finally, for selected benchmarks the performance and efficiency of the neuromorphic solution is compared to standard approaches.

Keywords: neuromorphic hardware, spiking neural network (SNN), benchmark, deep neural network (DNN), energy model

OPEN ACCESS

Edited by:
James Courtney Knight,
University of Sussex, United Kingdom

Reviewed by:
Lyes Khacef,
University of Groningen, Netherlands
Johannes Partzsch,
Technical University Dresden, Germany

****Correspondence:***
Christoph Ostrau
costrau@techfak.uni-bielefeld.de

Specialty section:
This article was submitted to Neuroscience, a section of the journal Frontiers in Neuroscience

Received: *11 February 2022*
Accepted: *27 April 2022*
Published: *02 June 2022*

Citation:
Ostrau C, Klarhorst C, Thies M and Rückert U (2022) Benchmarking Neuromorphic Hardware and Its Energy Expenditure. Front. Neurosci. 16:873935. doi: 10.3389/fnins.2022.873935

1. INTRODUCTION

With the increasing maturity of spiking neural network (SNN) simulation tools and neuromorphic hardware systems for acceleration, there is an increasing demand of potential end-users for platform comparison and performance estimation (Davies, 2019). Typical questions include the demand for speed-up of large-scale networks, potentially including plasticity rules for learning, and efficient implementations for so-called edge computing use-cases. For large-scale high-performance computing, a typical workload for comparing implementations is the full-scale cortical microcircuit model, which has been demonstrated on various platforms and forms the de-facto standard (van Albada et al., 2018; Rhodes et al., 2020; Golosio et al., 2021; Knight and Nowotny, 2021). Around the Intel Loihi chip (Davies et al., 2018) there has been a lot of work comparing SNNs to classical algorithmic approaches on standard of-the-shelf hardware systems (Davies et al., 2021). The current work is situated in between these two approaches of benchmarking individual implementations on (large scale) systems and comparing a single neuromorphic system to classical computation. It fills the gap with small to medium-scale neuromorphic benchmarks. We present our benchmark framework SNABSuite (Spiking Neural Architecture Benchmark Suite), which is publicly available. The suite currently supports simulations using NEST (Jordan et al., 2019) (CPU—single and multithreaded), GeNN (Yavuz et al., 2016) (single threaded CPU, GPU), digital (SpiNNaker; Furber et al., 2013), and analogue [Spikey; (Pfeil et al., 2013), BrainScaleS (Schmitt et al., 2017)] neuromorphic hardware. SNABSuite focuses on cross-platform benchmarking using a backend agnostic implementation of SNNs coupled to backend specific configurations (e.g., setting neuron model and parameters or network size), allowing direct cross-platform comparisons of benchmark specific performance metrics. We present results from low-level benchmarks to application related tasks like solving constraint-satisfaction problems (CSP). As an example, solving Sudoku puzzles is a representative for this class of problems and can be realized using a winner-takes-all (WTA) like implementation (Maass, 2014). This implementation is scalable, and thus it can be adapted to size constraints of neuromorphic hardware. Furthermore, different implementations of the WTA structure allow emulating the network on substrates with limited and restricted connectivity demonstrating not only how fast a system can find a solution, but also which kind of network is mappable to the system at all. Consequently, from this application a benchmark candidate for the category of computational kernel benchmarks naturally emerges: the evaluation of the various implementations of WTA networks as a building block for a broader range of applications (in addition to the CSP class of problems there is for example the spiking SLAM algorithm; Kreiser et al., 2018b). Even closer to the hardware system is the first category of benchmarks targeting lower-level features of the system and characterizing its basic properties. These properties, an example is the spike-bandwidth between neurons, are effectively limiting all networks and as such are relevant when designing a network for a specific system. They are not solely given by pure theoretical considerations but depend on several factors: runtime optimizations in the internal event routing, the chosen connectivity and combined spike rates impact these characteristics. For example, a given connectivity might fit onto the neuromorphic system, however, when operating at its limits, spike loss might still occur.

Another very common approach of using SNNs, which is applicable to all target platforms, is the conversion of pre-trained artificial networks (ANN) into SNNs (Rueckauer et al., 2017). Here, we support rate-based as well as time-to-first-spike-based encodings, and through different network layouts and sizes we are able to fully utilize the small-scale Spikey chip as well as the larger SpiNNaker system, allowing a fair comparison of key characteristics like time and energy per inference. A related sub-task is measuring the resemblance of the neuron activation curve to the ReLU function used in the ANN.

Due to a mixture of qualitative and quantitative benchmarks, the suite does not provide an oversimplifying benchmark score as known from suites in classical computation. Furthermore, a recent addition to the framework is an energy model which allows to estimate the energy expenditure of neuromorphic systems by running simulations in, e.g., GeNN or NEST on standard hardware. The estimated results closely resemble previously published values and are confirmed by newer measurements. All in all, this results in a benchmark suite which is reflecting, up to a certain extent, the current state of the art of SNN algorithms that are applicable to the aforementioned neuromorphic platforms and simulators. Hence, the suite fulfils the major requirement of being representative and relevant to our key audience of potential end-users (from neuroscience).

The remainder of this paper is structured as follows: the methods section introduces the neuromorphic systems and SNN simulators used in this work. The benchmark suite and its design are discussed as well as selected benchmark networks. Before elaborating the energy model and contributions to the energy expenditure of the human brain, selected benchmarks of the proposed suite are discussed. Results of the latter are detailed in the respective chapter. The energy model is validated using several of these benchmarks and a naive upscaling of related energy costs allows to compare the hardware systems to the human brain. Finally, the performance and efficiency is compared to classical approaches (algorithms or ANN accelerators) where applicable. The last section provides a summary and an outlook.

2. METHODS

This section provides an overview of the employed SNN simulators and neuromorphic hardware systems before discussing the benchmark suite, selected benchmarks, and the energy model.

2.1. Neuromorphic Systems and Simulators

In the following, all neuromorphic systems and simulators used in this work are reviewed. A summarizing table is provided in **Supplementary Table 1**. When it comes to standard of-the-shelf hardware, like CPUs and GPUs, our benchmarks utilize two simulators. The **NEST** simulator (Gewaltig and Diesmann, 2007) is suited for large scale simulations of SNNs on multiple computation nodes (in HPC systems) or multithreaded simulations on a single node. The simulation code as well as the neuron models are written in C++ code and pre-compiled at installation time. Through a Python interface [PyNEST (Eppler, 2008) or PyNN (Davison, 2008)] the user can build networks of neuron populations and spike or current sources to provide input to the simulation. **GeNN** (Yavuz et al., 2016) supports both single threaded CPU simulations and GPU simulations using CUDA or OpenCL. In this work we test only an NVIDIA RTX 2070[1], thus we stick with the CUDA backend. Similar to NEST, GeNN allows building networks within Python, but the direct interface is written in C++. Neuron and synapse models are programmed in an imperative way and are compiled at runtime. In contrast to NEST, which was created to reproduce exact spike trains with accurate simulations and hence using a fourth order Runge-Kutta-Fehlberg integrator for the LIF neuron models, GeNN uses a closed-form representation assuming a fixed input current over the integration step, which is usually set to 0.1ms. Especially with larger time steps, this can lead to numerical artifacts in the membrane voltage as well as in spike times (see Hopkins and Furber, 2015 for a discussion of the precision of various numerical solvers).

[1] Details of the NVIDIA RTX 2070 can be found at https://www.nvidia.com/en-me/geforce/graphics-cards/rtx-2070/.

Closest specialized hardware system to CPU/GPU simulators is the **SpiNNaker** platform (Furber et al., 2013). Fabricated in a 130nm CMOS process, the SpiNNaker chip consists of 18 general purpose ARM968 cores with 16 cores being used for the simulation of spiking neurons. Each core features 64KB memory for data and 32KB for instructions, each chip has access to additional 128MB of off-die SDRAM. Several chips are connected in a toroidal way and combined to form a small scale four chip board (SpiNN3) or a 48 chip system (SpiNN5). Between cores and chips, the spike communication is using address event representation, where a single packet contains the address of the sending neuron and the spike time is modeled by the time of appearance (Furber et al., 2014). For accessing the hardware, a PyNN interface is provided which is coupled to the various components of the SpiNNaker software stack SpiNNTools (Rowley et al., 2018) and sPyNNaker (Rhodes et al., 2018). The software stack maps the individual networks at runtime to the attached system, placing at most 255 neurons on a single core. Similar to GeNN, the numerical integration is using a closed form solution for the LIF model and assumes constant currents during the full time step. When using an algorithmic timestep of 1ms, the full simulation is running in realtime, which means that 1s of model time is simulated in 1s of wall-clock time. When reducing the algorithmic timestep to 0.1ms to increase the accuracy of the simulation and to potentially reduce the amount of spikes per machine timestep, the system slows down the simulation by a factor of 10.

Finally, two mixed-signal systems from Heidelberg are used within this work. The **Spikey** system (Pfeil et al., 2013) employs above threshold analogue circuitry implemented in a 180 nm CMOS process. Spikey consists of single chip featuring two blocks of 192 neurons each supporting up to 256 independent synaptic inputs. Because of the digital communication of spikes, these neurons can be connected quite flexibly with some constraints regarding cross-chip connectivity and enforcing the separation of excitatory and inhibitory neurons. The chip emulates neurons with a fixed acceleration factor of $10,000$, which means that 1s of model time is emulated in 0.1ms. To counter the analogue mismatch between the neuron circuitry, the software interface has a built-in calibration and maps high-level parameters of LIF neurons in the PyNN interface to adapted low-level hardware parameters. Spikey emulates conductance-based LIF neurons with some restrictions on neuron parameters, and weights are encoded with 4 bit precision and a fixed range of values. As the neuron model is implemented as circuitry, there is no flexibility in changing the neuron model itself. The successor system, **BrainScaleS**, is implemented using updated HICANN chips and supports much larger networks using wafer-scale integration (Schemmel et al., 2010; Petrovici et al., 2014). The implemented neuron model is a conductance-based LIF model with an optional adaptive exponential extension. A single HICANN chip consists of 512 neuron circuits each supporting 220 synapses. Up to 64 neuron circuits (multiples of two) can be combined to form a single virtual neuron, increasing the connectivity per neuron and the robustness against noise. Wafer-scale integration is used to combine 384 accessible chips into a single addressable system, allowing to emulate networks with up to nearly 100,000 neurons. The system comes with neuron calibration to compensate for device mismatch which also provides blacklisting capabilities to exclude circuits and neurons that are not working at all or are misbehaving in some way. Other properties, like acceleration factor or fabrication technology, are similar to those of the Spikey system.

2.2. Benchmark Framework

The first issue encountered when developing a black-box benchmark for neuromorphic hardware is related to the various interfaces to the hardware systems. First introduced in Stöckel et al. (2017), the Cypress[2] library is a C++ framework allowing to access all systems in a backend agnostic manner. The structure and input of an SNN is defined in an abstract interface that is quite similar to the PyNN interface. After compilation, the target backend can be chosen at runtime as long as the respective software packages (and neuromorphic systems) are installed on the working machine. Hence, network definition and data analysis can be decoupled from the actual target backend. However, it is still possible to change some low level properties at runtime, like, e.g., the number of neurons per core on SpiNNaker or the simulation time step on all digital simulators. This platform configuration is appended to the simulator string, which is provided as a command line argument, using the JSON format. At the time of writing, Cypress supports all systems reviewed in Section 2.1, and an overview of the framework is given in **Figure 1**. The simulation flow is the following:

- The network with its neurons and populations is set up in Cypress specific data structures.
- When `run` is called in the C++ source, some compatibility checks are done, e.g., regarding supported neuron models on a specific backend.
- Next, the Cypress network is mapped to the backend specific API, e.g., by creating a mirrored network in the target framework.
- Connection tables and input spikes are generated wherever a target does not natively support it.
- The backend executes the simulation and recorded data, like spikes or voltage traces, is written into the Cypress network instance.

After the simulation, spike data or voltage traces are provided in the same format and can be interpreted by the end-user.

While this abstracts away the individual backends from the main implementation of the benchmark suite, the suite itself needs to provide more flexibility regarding platform specific configuration of the benchmarks. There are two main reasons why the SNNs need to be configurable depending on the target backend: first, the different simulators/emulators support different network sizes and connectivity. Thus, to fully utilize every platform (in case of a scalable benchmark) the suite needs to include mechanisms to incorporate these properties into the configuration of the network. This is comparable to implementations of classical benchmarks as the

[2] https://github.com/hbp-unibi/Cypress

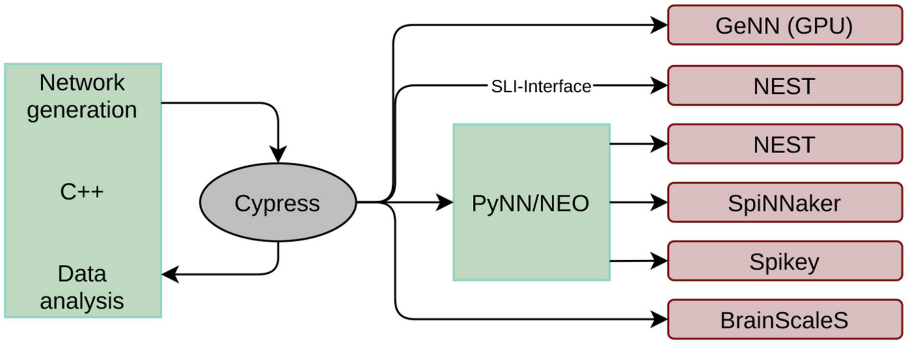


FIGURE 1 | Structure of the Cypress library. Network generations and data analysis is done in C++ code. The library takes care of mapping the created network to the target platforms listed on the right side using the individual platform APIs.

High Performance Linpack benchmark (Dongarra et al., 2003), where problem sizes can be adapted to the target system. For neuromorphic systems, further limitations might be related to supported neuron models, restricted parameter space or bandwidth limitations. Second, the work in Stöckel et al. (2017) demonstrated that using the same neural parameters for all target platforms might give an unfair advantage to the platform for which these parameters have been tuned to. All in all, this requires us to factor out the benchmark configuration, too. Changing network sizes would usually require to recompile the whole network, which is why we included a mechanism to parse all these parameters from JSON files. Each of these benchmark specific files contain a section for every platform and the suite allows to define a default set of parameters. Configuration options depend on the individual benchmark and may include network size, neuron model and parameters or maximal spike rates. On the one hand, this allows fair comparison between different platforms. On the other hand, the actual executed workload might differ between platforms and has to be kept in mind. We see this as a compromise between real "black-box" benchmarking and individual implementations for every platform. This has been accounted for in the Spiking Neural Architecture Benchmark Suite (SNABSuite) and its architecture was already proposed in Ostrau et al. (2020b) combined with a coarse overview of the benchmark approach. Its modular structure factored out all backend specific configuration and the benchmark implementation. Furthermore, through a common API to all benchmarks, these can be interfaced by other applications e.g. for parameter sweeps optimizing the configuration, besides the mere consecutive execution of all benchmarks. To address different sizes of the systems, SNABSuite supports defining several sizes of benchmarks using the benchmark index. Thus, sizes fall into four categories: single core, single chip, small scale, and large scale system. For systems like Spikey, the first index refers to the first available category, which in this case is the single chip.

The next section will introduce selected benchmarks of the SNABSuite, called SNAB (Spiking Neural Architecture Benchmark).

2.3. Neuromorphic Benchmarks

When choosing benchmarks for integration into SNABSuite we are limited by the main criterion: a potential benchmark has to be portable to as many of our target platforms as possible. Otherwise, the benchmark metric cannot be compared between platforms. Ideally, the potential candidate has been already successfully demonstrated. Here, it becomes clear that the implemented benchmarks can only lag behind state-of-the-art SNNs, as there is either missing support for newly introduced neuron models or learning rules (thus, these are not implemented yet or not integrated into the analogue circuitry), or there is some adaptation required for mapping the networks to the hardware. To overcome this issue at least partially, SNABSuite integrates several levels of benchmarks, categorized from low-level characterization benchmark, which measures basic hardware properties as, e.g., maximal spike rate of neurons, up to high-level application benchmarks, that measure the performance of a selected workload with reduced extrapolatory meaning to new benchmarks. These categories are described in **Figure 2**.

2.3.1. Low-Level Characterization Benchmark

Low-level benchmarks target some basic characteristics of a target platform. The resulting performance metrics are quite universal and applicable for a wide range of applications. The first example measures the maximal output frequency of neurons. The metric of average output rate per neuron is a limiting factor in many applications, especially when using a rate encoding of data. This can have an influence on WTA networks as well, as the current winner population might spike at high rates during its winning period. For measuring the maximal output rate, neurons are put in a state where they fire by themselves by setting the resting potential above threshold. Not all simulators/emulators allow to set the reset potential above the threshold, which is why in those cases a small membrane time constant will lead to high firing rates. In addition, the output rate is limited by the configurable refractory state, providing an upper limit of the maximal measurable rate. This specific benchmark comes in several implementations, either using a single neuron, or a partially/fully recorded population of neurons. Here, the

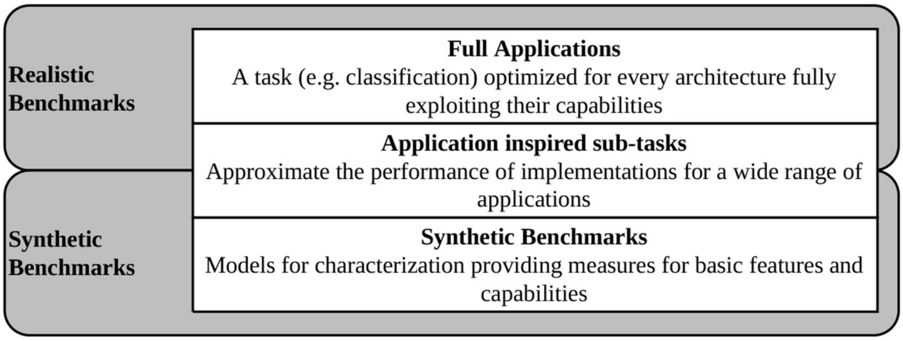

Realistic Benchmarks
Synthetic Benchmarks
Full Applications
A task (e.g. classification) optimized for every architecture fully exploiting their capabilities
Application inspired sub-tasks
Approximate the performance of implementations for a wide range of applications
Synthetic Benchmarks
Models for characterization providing measures for basic features and capabilities

FIGURE 2 | Different categories of benchmarks implemented in SNABSuite.

benchmark reveals whether the output rate decreases with an increased number of neurons, and whether partially recording selected neurons will have a positive influence on the measurable rate.

Other examples of this category of benchmarks include the maximal spike insertion benchmark, wherewith varying connectivity the maximal number of spikes inserted into the network can be measured, or a benchmark for spike transmission between populations.

When comparing the measured (output) rates between the platforms one has to keep in mind that these rates are evaluated in the biological time domain and do not account for the acceleration factor of, e.g., the analogue systems. Thus, a comparably small (output) rate does not immediately hint at low bandwidth between neurons on hardware.

2.3.2. Application Inspired Sub-task

Here, we introduce the class of application inspired sub-tasks. These networks do not yet perform a real world application, however, they are building blocks of the latter. The aim of this kind of benchmark is to provide measures related to these applications, but having broader applicability at the same time: quite often measurements of full application benchmarks can not be extrapolated to other neural algorithms or related fields. This is where the proposed class of benchmarks steps in.

The first example is the class of WTA networks. WTA architectures play a major role in several tasks most notably solving constraint satisfaction problems (Maass, 2014), to implement competing behavior in self-organizing networks (e.g., Diehl and Cook, 2015) or in approaches to neuromorphic simultaneous localization and mapping (SLAM) (Kreiser et al., 2018b). Here, we test three different architectures to account for the different constraints of our target systems (compare **Figure 3**). The simplest instantiation of a two population WTA network uses direct cross-inhibition and self-excitation. Every population gets individual random noise via Poisson spikes source using a one-to-one connectivity scheme. However, this simplest style infringes the constraint of separating excitation and inhibition, which is mandatory for the Spikey platform. The two alternative implementations use external inhibition by either having a global inhibitory population or by using mirror neurons. Benchmark metrics include the maximal winning streak of any population, the number of state changes, and the amount of time for which no winner could be determined. These metrics allow us to qualitatively assess the performance of the WTA dynamics on a substrate by identifying too stable or too fragile winner populations. The respective winner population is determined by counting the spikes per population within a 15ms time window.

A second example is the similarity of activations curves to the rectifying linear unit (ReLU) activation function known from ANN. The motivation is clear: when converting pre-trained ANNs to SNNs a required feature is that neuron output rates increase linearly with increasing activation (Cao et al., 2015). This benchmark samples through different input spike rates measuring the output and calculating the similarity between both curves. As a main metric we chose the average deviation from the target curve for every frequency, which is then again averaged across the different frequencies. A low deviation testifies that the neural substrate is capable of reproducing the ANN activation curve. A second metric is the averaged standard deviation. Here, a high value indicates larger variances across different neurons.

2.3.3. Full Application Benchmark

Full application benchmarks build the last category of benchmarks. Here, a high-level task is solving a certain problem using SNNs and benchmark metrics are usually related to accuracy. For this work we evaluated selected applications/algorithms for which the only requirement is that all target systems are able to potentially support it. This excludes networks with, e.g., continuous access to membrane potential. As an example for this category, the spiking binary associative memory benchmark (BiNAM) (Stöckel et al., 2017) calculates the retrieved amount of information in bits and compares it to the non-spiking variant. The BiNAM is trained offline and used as a synaptic connection matrix in the SNNs. Neuron parameters are tuned to reach maximal capacity, which is equal or below the capacity of the non-spiking variant. To reduce the computation time of this benchmark and in contrast to the analysis in Stöckel et al. (2017), larger networks are tested with a subset of samples only, which approximates the real relative capacity.

A second example is a spiking Sudoku solver (Ostrau et al., 2019), where the Sudokus are representative of the class

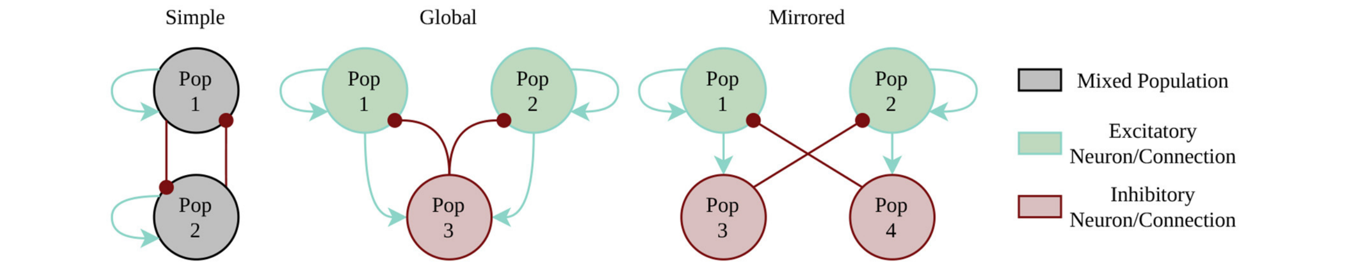


FIGURE 3 | Winner-Takes-All implementation styles. Random spike source are not included in the picture, as every neuron of the excitatory populations (1 and 2) has a one-to-one connection to its individual Poisson spike source.

of constraint-satisfaction problems. As mentioned above, this network uses WTA structures to implement the solver. Here, every possible number in the Sudoku puzzle is represented by a population of neurons. Sudoku rules, interpreted as constraints on all numbers, are implemented using inhibitory connections between the different numbers. Hence, there is inhibition between the numbers situated in a single cell, same numbers in a row, column, and sub-box of the Sudoku. Every neuron in this network has its own Poisson spike source. For analysis, spikes are binned and the respective winner in a Sudoku cell is determined. The benchmark metric is the bio-time to solution, which returns the value of the first time bin in which the solution is complete. The previous publication (Ostrau et al., 2019) analyzed the time-to-solution of 100 assorted Sudoku puzzles for every Sudoku size. Here, we reduce the analysis to a single puzzle to reduce benchmark time, but add GeNN as additional backend and compare the time and energy to solution to algorithmic approaches. Furthermore, the model is used in the validation of the proposed energy model.

A third benchmark is the conversion of deep neural networks to SNNs (Ostrau et al., 2020a). For this conversion, DNNs are trained using ReLU activation functions without biases, and for simplicity, only densely connected layers. This pre-trained network is converted to a SNN by rescaling the weights and converting the input data into rates (Diehl et al., 2015). Currently, this procedure is only evaluated for the MNIST handwritten digits dataset. To reach optimal performance, neuron parameters have to be adapted. For the analogue Spikey system, it was necessary to scale down the MNIST images by 3 × 3 average pooling to create a network that maps on the substrate. This network has as 81 × 100 × 10 layout and further employs excitatory connections only. A second network included in this analysis is the 784 × 1,200 × 1,200 × 10 network published with Diehl et al. (2015). To reach higher accuracies on the analogue system and encounter neuron to neuron variability, a hardware in the loop retraining approach is applied, similar to the one presented in Schmitt et al. (2017). Furthermore, we extended this set of benchmarks by also employing time-to-first spike encoding, where normalized input values are mapped to spikes using $f(x) = (1 - x) \cdot T$, where T is a configurable timescale. The original analysis of converted and pre-trained DNNs in Ostrau et al. (2020a) is extended here by a time-to-first spike encoding and an energy-per-inference comparison to standard accelerators for DNN inference.

A basic building block for the Neural Engineering Framework (Eliasmith and Anderson, 2004), but also an application by itself, is the approximation of functions via activation curves of neurons. For this, we feed spike rates into a population of neurons (one-to-all connection) and measure the response function of individual neurons. Given a certain variability across the population, one finds different response curves (compare **Figure 4**) that can be used as basis of the function space of continuous functions.

$$f(x) \approx \sum_{i}^{\#\text{Neurons}} a_i g_i(x) \tag{1}$$

Here, x is a number that has to be mapped to the available rate interval, $g_i(x)$ is the decoded response rate of a neuron, and a_i are the neuron specific coefficients. For encoding x into rates, we normalize the input interval and linearly transform it to rates (given a maximal rate as a parameter). The response to that rate is decoded to values in the unit interval, again given a maximal frequency. For fixing the coefficients in Equation (1), it is required to have at least as many sampling points as neurons in the target population. For testing the approximation capabilities, more sampling points are used, including points in between the original ones for fitting, from a second simulation/emulation. This is shown in **Figure 4** on the right side. Since the output activity of neurons should be more or less the same in several runs, we can use the same set of input/output rates to fit several functions. In theory, one could see the coefficients as decoding weights, and using a second matrix for encoding, we could chain these approximation populations to approximate more complex calculations, which is not currently covered in this benchmark.

An application from robotics is the spiking localization and mapping algorithm. Existing work proposes a sort of spiking state machine to track the current position and head direction of the (virtual) robot, and learning a map of the surroundings using spike-timing dependent plasticity (STDP) and a bumper sensor (Kreiser et al., 2018a,b, 2020). We adopt this network and use it as a benchmark using the accuracy of the learnt map as a benchmark metric (pixel-wise false positives for a learnt obstacle that does not exist in the simulation and false

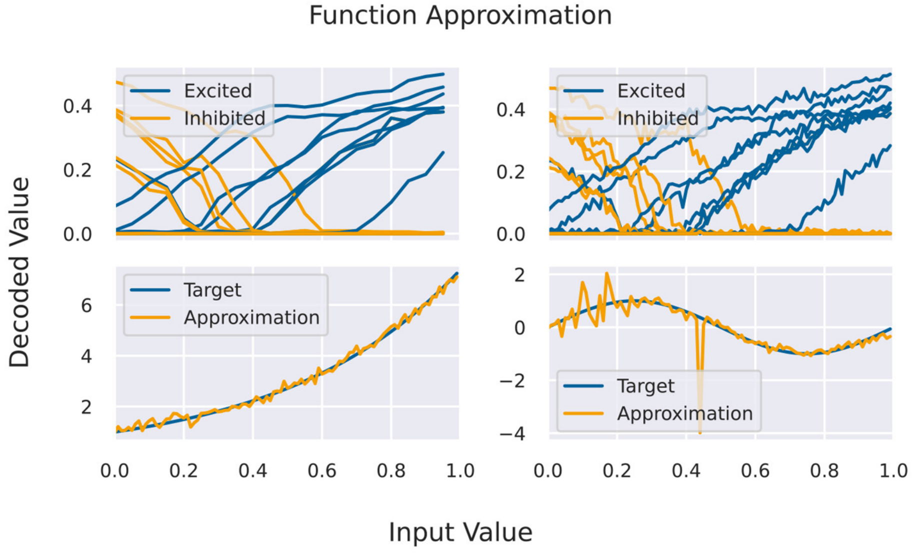


FIGURE 4 | (Top) Activation curves for various neurons for fitting (left) and testing. Every second neuron has a positive bias, and the input is inserted via inhibitory connections (depicted in yellow). **(Bottom)** Results of the function approximation for exponential and sinus function using the testing activation curves.

negative for a not learnt obstacle). The WTA populations used for tracking the current state of the robot could not be reliably tuned on analogue hardware using population-wise neuron parameters and would require neuron-specific tuning of 364 neurons, which is why this is currently not included. Here, only an automated approach would lead to reproducible results. The original work however targeted analogue hardware, demonstrating that the proposed algorithm is indeed suited for this kind of system.

2.4. Comparison to DNN Benchmarks

The original meaning of DNN benchmarks is related to datasets for benchmark DNN algorithms and network topologies, comparing the efficiency and accuracy of networks. Since our target is benchmarking hardware systems, we focus on hardware benchmarks, where the learning algorithm and network topologies are typically fixed. One of the first benchmark suites for DNN acceleration is Baidu Deep Bench[3]. This suite was introduced when the field of DNN accelerators began to grow while the DNN community was in fast progress, and representatives of an application domain are yet to exist. Thus, full applications benchmarks could not be integrated into the suite. Furthermore, every hardware system came with its own deep learning library, increasing the effort to maintain such a suite. Consequently, the authors decided to do some low-level benchmarking using typical core workloads of deep learning. These are comparable to our aforementioned low-level benchmarks. Instead of benchmarking dense or convolutional connectivity schemes or vectorized application of the activation function, our benchmark suite targets spike input and output rates, as well as the bandwidth between populations using various connectivity schemes.

Later, DAWNBench (Coleman et al., 2017, 2019) was introduced to the community. In comparison to DeepBench, the workloads cover various application categories instead of computational kernels. Besides benchmarking mere execution speed of these DNNs, its benchmark metrics account for potential differences in precision and accuracy of accelerators. As with less precise data formats the inference speed can be increased but usually at the cost of impeded accuracy. For training the network, the time required to reach a pre-defined accuracy is measured, which should account for the variances in speed and accuracy. If the criterion is met, the trained network could be used for inference, where delay between input and output is the main metric. DAWNBench built the basis for the current state-of-the-art benchmark suite MLPerf, which not only includes several application domains for DNNs, it also separates training and inference benchmarks and consists of several execution domains, from embedded to sever scale machine learning (Mattson et al., 2020; Reddi et al., 2020). In all application domains, a state-of-the-art network topology has been developed by the community, and as such, the suite can be considered to be representative of the field. Furthermore, most currently developed hardware accelerators support the feature set required to execute the selected networks. This is not the case for neuromorphic computing: There is a broad range of learning algorithms and workarounds for the back-prop algorithm, let alone the available neuron and synapse models. As such, there are only few commonly agreed learning algorithms that can be ported to all hardware platforms. Hence, our benchmark suite relies on low-level benchmarks too which, as a class of benchmarks, have been historically the first step to DNN benchmarking.

[3] https://github.com/baidu-research/DeepBench (accessed November 3, 2021).

2.5. Energy Model for Neuromorphic Hardware

To estimate the power consumption of a network simulated on a target system we identified the following workloads as core contributions to the overall energy expenditure:

- The static power consumption of the idle neuromorphic system
- The energy required for a virtual spike source neuron to emit an event
- The power required to simulate/emulate an idle neuron
- The energy used for a real neuron to emit a spike
- The energy expenditure of transmitting a single spike
- The static power required for activating STDP and the energy per synaptic event

By subsequently activating different parts of a full network we are able to calculate the different contributions from individual processes. In more detail, we start measuring idle power consumption. Next, idle neurons are simulated. For calculating the power per simulated neuron, the idle power is subtracted. Following this line, the energy per generated action potential is measured by simulating neurons that fire by themselves after subtracting the power for idle neurons and idle hardware. In the end, the different processes are mapped to a power/energy budget. Given a network simulation, the overall energy expenditure can be calculated. For simplicity of the model, we do not account for different channels of spike communication. For example, whether source and target neuron are situated on the same chip or in close physical neighborhood plays no role in our model.

For SpiNNaker, the power measurement is done using a Ruideng UM25C USB meter. It measures the power for a supply voltage up to 20V, which allows the measurement of both the small SpiNN3 board (5V) and the larger SpiNN5 board (12V). For Spikey, the sample rate of the USB meter is insufficient. Thus, we fix the power supply (Aim TTi CPX200DP) to 5V and measure the supply current using a Fluke 289. The NVIDIA GPU allows to directly read out the current power consumption using monitoring tools. All setups allow automation of the measurement process using either the provided Bluetooth or serial interface. For the final calculation, every value is an average covering 20 measurements. Furthermore, the devices have been plugged in for several minutes to assure that devices reach their idle temperature.

To compare the energy expenditure of neuromorphic hardware to its biological counterpart, we use the data acquired by Attwell and Laughlin (2001), which calculate the amount of ATP molecules required to maintain resting potential and for active signaling. These values were calculated for the rat's neocortex and have to be adapted for the human brain. According to Lennie (2003), the human brain consists of larger neurons, which can be accounted for by using factors of 2.6 for the energy expenditure of maintaining resting potential and 3.3 for action potential generation. Finally, Howarth et al. (2012) argues that the overlap of sodium and potassium fluxes during action potential generation in the human brain is actually smaller than originally assumed, correcting the original value of 4 from Attwell and Laughlin (2001) to 1.24. Furthermore, the costs of auxiliary functions in the brain ("housekeeping") is about 33% of the signaling costs (Howarth et al., 2012). **Table 1** lists the different contributions to the overall energy expenditure of a single neuron. The resting potential expenditure includes the costs of glia cells under the assumption of a one-to-one correspondence. The contribution of "other pre-synaptic loads" includes the costs for vesicle cycling and Ca^{2+} recycling. For the overall power consumption we assume (following Attwell and Laughlin, 2001) an average spike rate of 4 Hz while the average fan out of a neuron is 2,000. For converting the amount of freed energy per ATP molecule, Rosing and Slater (1972) reports $4.6495 - 5.5628 \cdot 10^{-20}$ J ATP^{-1}. Hence, we used $5 \cdot 10^{-20}$ J ATP^{-1} for our calculations. When scaling the energy expenditure up to full brain size using $8.61 \cdot 10^{10} \pm 8.12 \cdot 10^{9}$ neuron cells of the human brain (Azevedo et al., 2009) we end up with an overall energy expenditure of 21.5W.

TABLE 1 | Various contributions to the overall energy expenditure of the human brain.

Action	ATP molecules	Energy expenditure in W
Action potential	1.57×10^{9}	7.86×10^{-11}
Resting potential	1.15×10^{9}	5.77×10^{-11}
Post-synaptic receptors	1.40×10^{5}	7.00×10^{-15}
Neurotransmitter recycling	1.14×10^{4}	5.70×10^{-16}
Other pre-synaptic loads	1.20×10^{4}	6.00×10^{-16}
Single neuron	4.03×10^{9}	2.02×10^{-10}
Single neuron + housekeeping	4.98×10^{9}	2.49×10^{-10}

Single neuron refers to a spike rate of 4Hz and a fan-out of 2,000.

3. EXPERIMENTS AND RESULTS

This section provides results for the three categories of benchmarks. Major outcomes are discussed and evaluated. Furthermore, the proposed energy model is validated on selected benchmarks. Since individual contributions to the energy model are known, the model allows us to trivially upscale networks to brain size and compare neuromorphic to biological energy efficiency. Finally, the efficiency is also compared to algorithmic approaches for the Sudoku network and to ANN accelerators for the converted pre-trained networks.

3.1. Characterization Benchmarks

In this category of benchmarks we look at two examples. The first one measures the maximal output rate of a set of neurons. **Figure 5** demonstrates the behavior of our target neuromorphic systems regarding this metric. CPU/GPU simulations are not included, as these do not suffer from any spike loss. Here, the maximal output rate is only limited by time resolution and the refractory period of the simulated model. Quite similar, the SpiNNaker system does not show any output bandwidth issues.

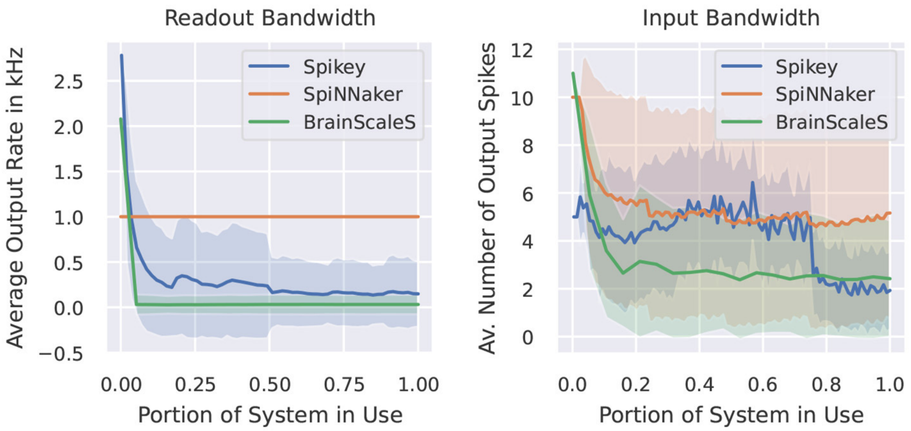


FIGURE 5 | Measured output rate (left) and the indirectly measured input rate from virtual spike source neurons using a one-to-one connection scheme (right). Light coloring indicates the empirical standard deviation across neurons.

The maximally required output buffers can be calculated before the simulation and the SpiNNaker software stack will ensure that spikes are copied to the host machine when reaching these limits. Both analogue systems suffer from bandwidth restrictions, as spikes are communicated via the on-chip network before reaching the target storage. However, the systems run in an accelerated manner, so in wall-clock terms the actual rates are increased by a factor of 10^4.

The second benchmark indirectly measures the amount of output spikes measurable when all neurons receive a spike at the same time via one-to-one connection. Ten spikes per neuron are inserted during the simulation and we provide the average number of output spikes. On SpiNNaker, the loss usually appears at the receiving neuron. If a core (simulating 255 neurons) receives too many inputs within the same timestep, the computation of that core lags behind the global timer. In our case, the computationally more expensive conductance-based LIF neuron model requires more resources than the current-based one, thus these limits depend on the neuron/synapse model in use. Note, that if running into such problems, the SpiNNaker software stack provides several configuration options to reduce the computational load of individual cores, including a slow-down of the simulation or the reduction of simulated neurons per core (which then requires more cores for the simulation of the network). For the Spikey system, the quite prominent drop in output spikes is related to the usage of the second block of neurons of the system. When using a single block only, the average amount of spikes is more or less constant, indicating that the origin of spikes loss is not a bandwidth issue, but more related to the neuron to neuron variability. For BrainScaleS, the amount of spikes for few neurons reaches the target of 10. Here, we expect that the closeness of output spikes reaches some output bandwidth bottlenecks. The overall constant curve for larger networks indicates that there is no additional constraint when using multiple HICANN chips.

3.2. Application Inspired Subtasks

The WTA behavior is tested for two populations only to check the general capability of the system to demonstrate the appropriate behavior. Results for this set of benchmarks are provided in **Supplementary Table 2**. For analogue platforms we see in spike raster plots, that random neurons are activated quite often. Especially for Spikey one can distinguish several neurons that emit spikes more easily compared to neighboring neurons. This reduces the capability of the substrate to simulate two equally probable winner populations and acts as a bias. To account for variances across simulation due to random seeds and trial-to-trial variation in analogue systems, benchmark metrics are averaged using ten simulations. The chosen metrics for this kind of network do vary a lot across simulations, as the random input noise to neurons is different for every instance. For BrainScaleS the obtained values are comparably worse, even in the one-to-one comparison to Spikey. This is basically due to different days of the evaluation and parameter tuning leading to different results and this should not be seen as a general deficit of the hardware platform. The second part of the table (in the **Supplementary Material**) shows values for the exact same benchmarks, but using only as few neurons as possible [simulators: one (source) neuron; Spikey: two neurons, doubled number of source neurons; BrainScaleS: two source neurons, but four neurons per population]. Here, all platforms demonstrate the capability of embedding WTA dynamics, with analogue hardware tending to have a higher variation in winners compared to the simulators.

Figure 6 shows measured activation curves. The aim is to reuse these curves to approximate the ReLU activation function known from the field of deep learning. Thus, the aim is to measure the capability of a system to simulate converted pre-trained networks with rate-encoding. Using a single input neuron that is connected to a simulator specific number of neurons, all target systems demonstrate the ability to approximate the ReLU function in certain frequency boundaries. For simulators,

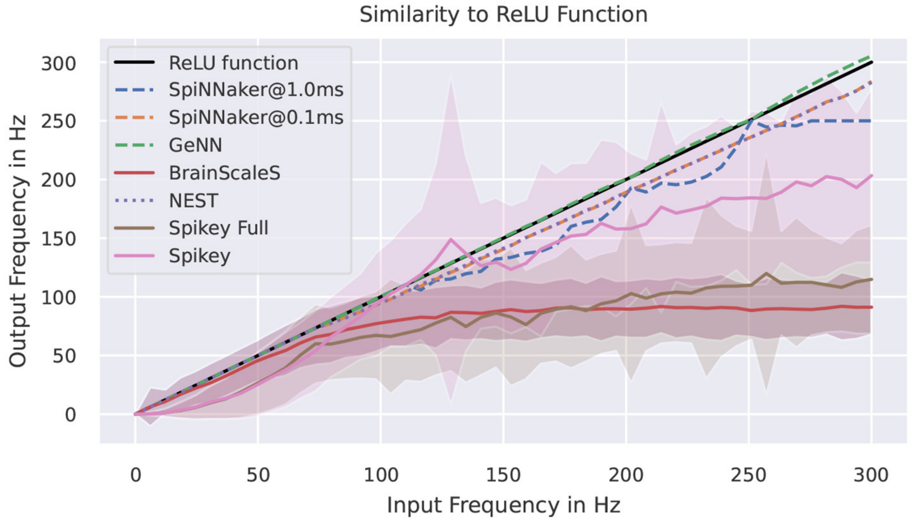


FIGURE 6 | Activation curves for various simulators. "Spikey Full" refers to the usage of both neuron blocks on the Spikey system. The target curve is depicted in black. SpiNNaker is tested with two different simulation timesteps.

this maximal frequency is, besides potential bottlenecks, limited by the refractory period and the timestep of the simulation. For SpiNNaker, no bottlenecks should be triggered in this setup, thus the deviations are results of the timestep of 1ms for running the network in realtime. The curves for the full Spikey system (employing both neuron blocks of the system) and the BrainScaleS system resemble each other. The rate limitation is a consequence of the large speedup and readout restrictions. The latter is due to shared priority encoders with limited maximal output rate of packages into the digital network effectively limiting the measurable number of spikes. If using only one neuron block of the Spikey system, the sensible range of frequencies is larger and meets expectations from the previously discussed low-level benchmarks. Putting these results into numbers, the **Supplementary Table 3** of results in the **Supplementary Material** validates the discussed observations. Note, that by reducing the measured frequency corridor the average deviation is expected to decrease.

3.3. Applications

The BiNAM benchmark comes in four different implementation styles, using bursts and/or populations to represent bits in input and output. Here, we present the results from the simple variant only, although (Stöckel et al., 2017) demonstrated that the analogue platforms potentially benefit from averaging the output over several neurons. The results are provided in **Supplementary Table 4**, and we summarize the most important findings here. In comparison to the old publication, several things have changed in our evaluation. First, we reduced the amount of samples used in the spiking recall phase. While training the BiNAM, we still use the theoretical prediction for the optimal sample count for determining the amount of binary patterns to store. For recall, we make use of the first few randomly generated patterns only and the exact amount of samples is configurable. Since false positives and negatives are on average equally distributed across these random samples, results are still in overall accordance with the full recall. This effectively reduces the simulation times of the larger networks. Second, the new set of results include GeNN and BrainScaleS platforms. While simulators perform close to the original, non-spiking model of the BiNAM, analogue platforms show reduced accuracy. For Spikey, an increased amount of false positives demonstrate that input bottlenecks are not the problem. With an increased amount of output spikes due to false positives, a potential pitfall is that correct positives are lost while false positives are recorded. This seems to be more of an issue with the BrainScaleS system: Here, the smallest network produces no false positives, but already some false negatives, which are mostly related to the neuron to neuron variability. Higher accuracy could only be reached using a neuron specific training of parameters. Spreading the network to two or more HICANNs, the performance of the network is degraded. An increased amount of false positives implies an issue with the read-out causing the increasing number of false negatives. When using larger networks that are spread across the wafer, the amount of false negatives decreases. We can only guess that spreading the activated neurons across more HICANN reduces the average load on individual priority encoders and routers.

The Sudoku benchmark comes in three styles: the first one is how a possible end-user would program such a network. Every possible number is presented by its own population. Between populations, high-level connectors are used (e.g., all-to-all connectors) to implement the direct inhibition. The second benchmark implements exactly the same network but uses a single population which includes all neurons. Connections are realized using custom connection lists, only the random noisy input is connected via one-to-one connectors. Comparing results (see **Table 2**) of both implementations reveals how good the

TABLE 2 | Results of the Sudoku benchmark for several implementation styles and two Sudoku sizes.

Platform	Bio runtime in ms	Sudoku size	Bio time to solution in ms	Wall-clock time to solution in ms
Simple Sudoku				
GeNN-CPU	5,000	2 × 2	20	0.81 ± 0.06
	10,000	3 × 3	4420	3,645.26 ± 219.17
GeNN-GPU	5,000	2 × 2	20	2.8 ± 0.04
NEST	5,000	2 × 2	20	9.77 ± 0.70
	10,000	3 × 3	2980	24,106.13 ± 3,447.74
BrainScaleS	50,000	2 × 2	6264 ± 6333.54	0.63 ± 0.63
SpiNNaker	5,000	2 × 2	20	200.01 ± 0.00
Simple Sudoku—single population				
GeNN-CPU	5,000	2 × 2	20	0.44 ± 0.06
	10,000	3 × 3	4420	1,212.41 ± 82.48
GeNN-GPU	5,000	2 × 2	20	1.31 ± 0.02
	10,000	3 × 3	4420	376.47 ± 6.44
NEST	5,000	2 × 2	20	6.30 ± 0.44
	10,000	3 × 3	2980	9,553.49 ± 51.32
BrainScaleS	50,000	2 × 2	6780 ± 10899.87	0.68 ± 1.09
SpiNNaker	5,000	2 × 2	20	200.01 ± 0.00
	10,000	3 × 3	1660	16,600.33 ± 0.00
Mirrored inhibition Sudoku				
GeNN-CPU	5,000	2 × 2	120	423.41 ± 2.28
GeNN-GPU	5,000	2 × 2	120	25.26 ± 2.07
NEST	5,000	2 × 2	100	52.79 ± 0.25
Spikey	30,000	2 × 2	363 ± 288.60	0.04 ± 0.03
SpiNNaker	5,000	2 × 2	140	4,200.17 ± 0.00

2 × 2 refers to a Sudoku featuring numbers 1–4, 3 × 3 is the standard Sudoku size.

underlying software can merge neuron groups. On SpiNNaker, the individual populations are mapped to individual cores, thus every core simulates only few neurons. This highly inefficient usage leads to the larger network not being available on the SpiNNaker platform. Furthermore, the simulation tools GeNN and NEST do also benefit from such a merging of populations, as the wall clock to solution is lower. For the GeNN GPU, the larger Sudoku network would require lots of working memory at compile time, which is why it is not included. Otherwise, we see that the biological time to solution is unaffected by the implementation style (when keeping the seeds for random input generation fixed). For BrainScaleS, the simulation is evaluated ten times to measure the influence of trial-to-trial variations of the analogue substrate. For comparison, using random seed in a GeNN simulation for the generation of input noise results in a time-to-solution of 58.0 ± 53.7ms for the small and 2716.0 ± 1869.5ms for the large Sudoku. Thus, the variance through trial-to-trial variation matches to some extent the variation due to different random input. While mean time to solution is similar between both implementation styles, the standard deviation is larger for the merged implementation. We are not aware of any specific issue that might be causing this and guess that this is related to the mapping to the hardware system: The networks is mapped to a restricted list of HICANNs, but we leave the placement of individual neurons to the BrainScaleS software stack. Between repetitions, this mapping is kept fixed. Due to the different layout of both networks we cannot assume that the same virtual neuron is placed to the same hardware neuron in both implementations. Thus, one can assume that the simple implementation style was mapped in favor of this specific Sudoku or that by revealing the internal structure of the network, the mapping reduces the amount of spike loss appearing during the emulation.

For the larger Sudoku network, we were not able to reliably emulated it on the BrainScaleS system. However, using neuron specific parameters and a hardware in-the-loop training should result in decreased time-to-solution and reliable solving for larger Sudokus, too. The last implementation benchmarked includes a workaround for the Spikey system avoiding direct inhibition. Here, we see that fast solving of such constraint problems is possible on analogue hardware, and find the shortest wall-clock time to solution.

Next, we evaluate rate-coded converted deep neural networks. We focus on two networks, the first being created to be mapped to the Spikey platform using a 81 × 100 × 10 layout without bias and inhibition. The input images are rescaled using 3 × 3 average pooling. The second network has been published by Diehl et al. (2015). The aim is to reach an accuracy close to

TABLE 3 | Results of pre-trained and converted DNNs.

Platform	Parallel instances	Accuracy in %	Sim. time in s	Bio time/inf. in ms
Spikey network 90.13%				
GeNN-CPU	1	**89.11**	6.83 ± 0.25	500
	100	88.87	4.29 ± 0.02	500
GeNN-GPU	1	89.10	35.64 ± 029	500
	100	88.87	0.70 ± 0.01	500
NEST	1	88.98	86.43 ± 2.09	500
	20	88.98	62.83 ± 3.70	500
BrainScaleS	1	57.92 ± 5.92	0.95	900
Spikey	1	65.23 ± 0.78	**0.35**	300
SpiNNaker	1	88.41	6677.20	500
	239	88.40	235.22	500
In-the-loop retraining				
BrainScaleS	1	83.03	0.95	900
Spikey	1	**85.16**	**0.22**	180
Diehl network 98.84%				
GeNN-CPU	1	**98.85**	276.23 ± 1.24	500
	36	**98.85**	325.56 ± 3.12	500
GeNN-GPU	1	**98.85**	46.89 ± 0.66	500
	36	**98.85**	**9.85 ± 0.01**	500
NEST	1	98.82	1763.54 ± 22.17	500
	53	98.82	2646.66 ± 246.88	500
SpiNNaker	1	98.73	13695.06	500
	53	98.77	1724.87	500
Diehl network (TTFS) 98.84%				
GeNN-CPU	1	97.59	42.49 ± 1.14	9.12 ± 1.02
	10	**97.60**	40.99 ± 0.40	9.12 ± 1.02
GeNN-GPU	1	**97.60**	30.66 ± 0.54	9.12 ± 1.02
	10	**97.60**	**4.37 ± 0.02**	9.12 ± 1.02
NEST	1	97.59	540.95 ± 7.54	9.94 ± 1.01
	10	97.57	581.39 ± 1.80	9.94 ± 1.01
SpiNNaker	1	97.57	4817.04	**9.05 ± 1.08**
	61	97.56	626.44	**9.05 ± 1.08**

Only those benchmarks in the lowest part of the table employ the sparse time-to-first-spike (TTFS) instead of rate encoding. An extended table is found in ***Supplementary Table 5****. The best results are highlighted in bold.*

the original ANN accuracy. Thus, we used the 10 first images of the training set to coarsely optimize SNN parameters using parameter sweeps. Afterwards, we increased to number of images to 100 to do a more fine-grained optimization of the most fragile parameters like the maximal frequency for encoding inputs and the scaling parameter for pre-trained weights. To fully utilize the systems, several parallel instances of the same network evaluate mutually exclusive parts of the test set of 10,000 images. Regarding the loss of accuracy during the conversion process, we find a drop in accuracy of up to 1.5% for digital platforms and the Spikey network. For the analogue systems this loss is significantly larger which can be accounted for using the HIL retraining. Nevertheless, the loss is about 5%. Note, that for this retraining the inference time per sample has been adapted to reach maximal accuracy (see last column of **Table 3**). When comparing simulation times, the advantages of accelerated analogue neuromorphic computing come into play. Only the GPU with massively parallel instances is on a comparable level (at a much larger power consumption).

The network proposed by Diehl et al. (2015) features a smaller conversion loss. Here, the rate-coded variant suffers from up to 0.1% loss. Curiously, the GeNN simulation even improves the accuracy on one image which is most likely due to a lucky circumstance in the parameter/rate conversion process[4]. For SpiNNaker, the number of neurons per core was reduced to 180 (200 for the largest network on the SpiNN5 board). The machine timescale factor was increased to two (not for the largest network) effectively slowing down the simulation. Otherwise, the workload per core, due to the employed rate-coding, would overload and lead to lost spikes or even a break-down of the simulation. Thus, SpiNNaker is as fast as a single threaded NEST simulation, one order of magnitude slower than the GeNN CPU simulation and two orders of magnitude slower than the GPU simulation. When switching to time-to-first-spike (TTFS) encoding, the number of neurons per core is set to default while the timestep is decreased to 0.1ms slowing down the simulation by a factor of ten. This is due to the increased time precision required by this encoding. The overall loss during the conversion process is a bit larger, which might be due to the employed conductance-based synapse model [the original publication (Rueckauer and Liu, 2018) uses a simpler synapse model]. The overall response time (time between inserting the first spike of an TTFS encoded image) to the first and classifying spike in the last layer is about 9ms. The performance comparison to DNN accelerators is provided in Section 3.5.

Results for the function approximation benchmark are shown in **Table 4**. Simpler functions, like the linear function, can be approximated with a very small overall deviation. Furthermore, all target platform show quite similar approximation errors. For digital simulators this is realized using two distinct diversification mechanisms. First, a constant rate of spikes is fed to target neurons using random connections weights (both inhibitory and excitatory). Second, random but fixed input rates are inserted using fixed weights. On analogue hardware this is not necessary due to the naturally occurring neuron-to-neuron variability. Here, higher approximation errors are most likely related to trial-to-trial variances.

The last benchmark performs partial workloads of a SLAM algorithm. **Figure 7** visualizes not only the coverage of the random trail of the robot in its virtual map, but also the learnt map of the simulators. This test environment features a 15 × 15 map with four obstacles. All tested platforms demonstrate the successful learning of the surroundings using the STDP (a spike pair rule with additive weight dependence) enabled connection. Small deviations from the target map occur due to not or only once visited spots in the map.

[4] This is reproduced in the quantization process for the neural compute stick in **Table 8**.

TABLE 4 | Average deviations for selected functions in the function approximation benchmark.

Platform	$f(x) = x$	$f(x) = 10 + x$	$f(x) = \sin(2\pi x)$	$f(x) = \cos(2\pi x)$	$f(x) = \exp(2x)$
GeNN-CPU	0.01 ± 0.01	0.37 ± 0.42	0.30 ± 0.56	0.16 ± 0.27	0.17 ± 0.29
GeNN-GPU	0.02 ± 0.01	0.38 ± 0.41	0.16 ± 0.14	0.18 ± 0.20	0.12 ± 0.10
NEST	0.02 ± 0.02	0.68 ± 0.82	0.29 ± 0.27	0.17 ± 0.14	0.19 ± 0.22
BrainScaleS	0.14 ± 0.38	1.54 ± 3.78	0.77 ± 1.92	0.52 ± 1.44	0.67 ± 1.76
Spikey	0.05 ± 0.06	0.51 ± 0.54	0.27 ± 0.29	0.25 ± 0.26	0.24 ± 0.25
SpiNNaker	0.02 ± 0.02	0.41 ± 0.38	0.32 ± 0.63	0.29 ± 0.44	0.23 ± 0.30

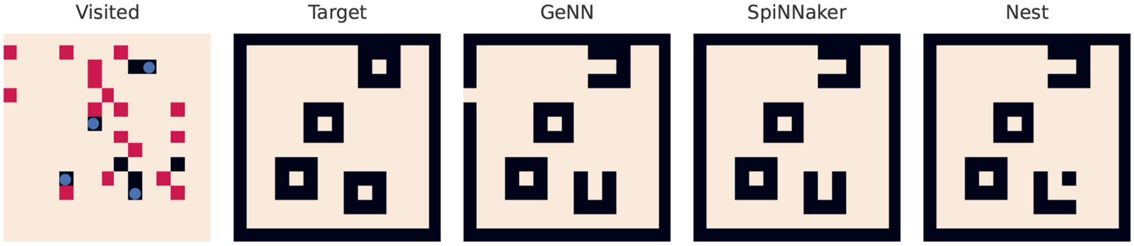


FIGURE 7 | Result of the spiking SLAM benchmark. The leftmost map visualizes the map with the obstacles (blue circles). Black pixels are not visited by the virtual agent while red ones are visited once or twice. The best achievable representation is missing two pixels (see the SpiNNaker result), as two points close to the right obstacles are not visited by the virtual agent.

TABLE 5 | Results of the energy model in comparison to the biological counterparts.

	Brain	Spikey	SpiNNaker	R2600X	Intel mobile	RTX2070
Housekeeping	4.75E-11	1.37E-06	1.66E-04	4.49E-04	1.23E-04	**9.76E-07**
Resting potential	5.77E-11	**3.83E-08**	8.99E-05	4.77E-05	4.25E-05	3.63E-06
Action potential	1.96E-11	**4.39E-10**	1.04E-08	3.04E-08	4.46E-09	4.71E-09
Transmission	8.17E-15	**1.08E-11**	9.59E-09	5.82E-08	2.14E-08	3.40E-09
Single neuron	2.49E-10	**1.49E-06**	3.33E-04	9.62E-04	3.37E-04	3.18E-05
Full brain	2.15E+01	**1.29E+05**	2.87E+07	8.29E+07	2.90E+07	2.74E+06

Values for the simulation of 1s of model time are reported in Joule. The single neuron and full brain estimates assume a fan-out of 2,000 synapses and a spike rate of 4Hz. R2600X: AMD Ryzen 2600X. Intel mobile: Intel Core i7-4710MQ. RTX2070: NVIDIA RTX 2070. Both CPUs are measured using a PeakTech power meter. The lowest values from simulators/emulators are highlighted in bold.

3.4. Energy

First, the energy expenditure of neuromorphic hardware is compared to the human brain. **Table 5** summarizes costs for the various contributions to the overall energy expenditure. Values for the brain are adapted from **Table 1**. For neuromorphic hardware, “housekeeping” refers to the scaled system’s idle power. For transmission, the values for random connectivity schemes are used. To scale values up to a full brain simulation, we assume that every neuron is connected to 2000 neurons and firing at 4Hz similar to Attwell and Laughlin (2001). The results demonstrate the superiority of the analogue system in regard to efficiency. Only the housekeeping costs are lower for the GPU due to the large number of neurons simulated. SpiNNaker performs on par with both CPUs and is less efficient compared to the GPU. Comparing the values for a single neuron or the full brain simulation, the biological paragon is four orders of magnitude more efficient than the analogue implementation. This huge difference cannot be compensated by using a modern fabrication process: According to Sun et al. (2019), the performance per watt doubles every 3–4 years. Applying the same scaling factor to the SpiNNaker system, this results in more than 8-fold improvement, which closes the gap to the GPU implementation. The step from 180 to 22/28nm FDSOI sub-threshold process would result in 50-fold improvement (Rubino et al., 2019) for analogue implementations, which most likely applies to the above-threshold circuits in Spikey, too. This results in ∼2.6KW, which is still two orders of magnitude above the values found in biology. Note, that this is a naive upscaling only, as the system neither supports simulation of such many neurons nor do they provide the infrastructure to connect these.

To validate the proposed energy model, predicted values for several networks are compared to measured ones. These results are summarized in **Table 6**. For the Spikey system, predicted values are quite close to measured values and deviations are <10% but are not covered by the statistical

TABLE 6 | Validation of the energy model.

Platform	Acc.	E/Inf.	Prediction of E/Inference in mJ			
	in %	in mJ	GeNN	Spikey	SpiNN3	SpiNN5
Spikey network with parallelism 1						
GPU	86.04	160.8	46.8 ± 1.3	0.19 ± 0.00	939.9 ± 2.5	8071.8 ± 16.6
Spikey	68.89	0.2	–	0.19 ± 0.00	939.9 ± 2.5	8071.7 ± 16.5
SpiNN3	87.07	950.8	–	0.19 ± 0.00	939.9 ± 2.5	8071.7 ± 16.5
SpiNN5	87.07	8148.1	–	0.19 ± 0.01	939.9 ± 2.5	8071.8 ± 16.6
Spikey network with parallelism 239						
CPU	86.03	–	–	0.00 ± 0.00	89.6 ± 1.3	38.6 ± 6.4
SpiNN5	87.04	38.2	–	0.00 ± 0.00	89.4 ± 1.3	38.5 ± 6.3
Diehl network with parallelism 4						
GPU	98.83	217.0	265.4 ± 16.3	–	1871.1 ± 43.1	7232.5 ± 205.5
SpiNN3	98.73	993.5	–	–	1806.0 ± 41.1	7181.3 ± 196.5
SpiNN5	98.74	6597.8	–	–	1806.0 ± 41.1	7181.3 ± 196.5
Diehl network with parallelism 53						
CPU	98.86	–	–	–	1046.1 ± 39.6	1013.5 ± 185.4
SpiNN5	98.77	488.5	–	–	979.8 ± 37.6	961.3 ± 176.1
	—	**in J**	**in J**	**in mJ**	**in J**	**in J**
Mirror Inhibition						
GeNN-GPU		56.0	378.5 ± 57.0	2.7 ± 0.0	14.2 ± 59.8	160.3 ± 71.3
GeNN-GPU†		62.6	261.2 ± 82.8	2.8 ± 0.0	88.0 ± 22.9	187.8 ± 114.7
Spikey		0.0029	–	2.7 ± 0.0	155.7± 6.5	1174.1 ± 33.4
Single population						
GeNN-GPU		15.3	7.5 ± 0.2	2.9 ± 0.1	134.7 ± 0.4	1153.7 ± 2.4
SpiNN3		134.3	–	2.9 ± 0.1	134.7 ± 0.4	1153.7 ± 2.4
SpiNN5		1171.6	–	2.9 ± 0.1	134.7 ± 0.4	1153.7 ± 2.4
Spiking SLAM						
GeNN-GPU		303.1	108.8 ± 2.8	–	45.7 ± 0.12	390.6 ± 0.8
SpiNN3		46.0	–	–	45.7 ± 0.12	390.6 ± 0.8
SpiNN5		398.0	–	–	45.7 ± 0.12	390.6 ± 0.8

For selected networks the measured energy (3rd column) is compared to the prediction of the energy model (right part of the table). For the Spikey network, input rates have been adapted to values used for the Spikey system to improve comparability. Thus, the reported accuracy might be reduced compared to ***Table 3****. Missing values are either due to missing runtime for the GPU simulation, or due to the network not being compatible with the Spikey system. More data points are found in* ***Supplementary Table 6****.*

†Simulation using three neurons per population similar to the Spikey parameter set.

error. Even better, relative deviation for the SpiNNaker system is <3%. This however is not true for the Diehl network, where low-level settings like the number of neurons have been changed explaining larger deviations. For GPU simulations the prediction is usually correct in order of magnitude, but severely deviates from actual measurements. Here, some features, like dynamic voltage and frequency scaling or temperature dependent clock rates, are not covered by the proposed energy model. Nevertheless, for Spikey and SpiNNaker the proposed model predicts the energy expenditure of a network simulation even though the network has been executed with, e.g., the GeNN simulator. More interestingly, there is an overall agreement of predictions based on analogue emulation and digital simulation. Thus, one can use the Spikey system to estimate the energy expenditure of a SpiNNaker simulation and vice versa (if the network maps to both systems).

3.5. Comparison to Classical Solutions

Finally, we address the comparison to classical algorithmic approaches for solving a Sudoku or ANN inference. In **Table 7**, the time and energy to solution of the former application are compared between a Raspberry Pi 4 with 2GB of RAM and neuromorphic systems. On the Pi 4 the Coin-Or Cbc[5] was employed to efficiently solve the Sudoku. For the small Sudoku puzzle, Spikey is the most efficient platform in regard to both time and energy to solution. The GPU is faster, but also more energy consuming compared to the algorithmic implementation. For the larger Sudoku, the latter outperforms SpiNNaker and the GPU implementation. This is most likely due to the SpiNNaker system and the GPU not being fully utilized. With a more up-to-date

[5] Found at https://github.com/coin-or/Cbc.

manufacturing process (see Section 4 above), the SpiNNaker implementation would be on the same level of efficiency as the RPI 4.

For deep network inference (see **Table 8**), the Spikey system is again the fastest and most efficient system. However, the ANN accelerators perform similar at a higher accuracy. Again, the rather old technology in Spikey is accountable for at least an order of magnitude in efficiency. The Intel Neural Compute Stick 2 (NCS)[6] benefits from a larger batchsize, which is defined at compile time. For the Edge TPU[7] a batchsize could not be configured. Both accelerators were able to simulate the full network without doing computation on the host machine. The GPU simulation requires one order of magnitude more energy, while SpiNNaker (even with full utilization) requires two orders of magnitude more energy. For the larger Diehl network, SpiNNaker, and the GPU simulation are on the same level of efficiency, while the latter being significantly faster. Switching to TTFS encoding and only counting the energy expenditure until the first, classifying spike appears closes the gap between both platforms and ANN accelerators. Here, a modern manufacturing process would result in SpiNNaker being the most efficient system. Note, that the slightly reduced accuracy in SNN simulations with TTFS encoding might be encountered by using SNN specific training methods (e.g., Neftci et al., 2019).

TABLE 7 | Comparing Sudoku solving on a Raspberry Pi 4 using an algorithmic approach with SNN solvers.

System	Time to solution in ms	Energy to solution in J
2 × 2 Sudoku		
RPI 4 2GB	5.00	0.016
SpiNN3	200.00	0.537
Spikey	**0.04**	10^{-4} × **2.105**
GeNN-GPU	1.42	0.061
3 × 3 Sudoku		
RPI 4 2GB	**261.0**	**0.91**
SpinNN3	560.0	16.77
GeNN-GPU	370.6	26.72

Metrics are time and energy to solution. The fastest and most efficient values are highlighted in bold.

4. DISCUSSION

To tackle the problem of missing cross-platform performance assessment in neuromorphic computing, we presented SNABSuite, an open-source benchmark suite. SNABSuite features a set of workloads implemented in a platform-agnostic way, but also providing mechanisms for benchmark and platform configuration. This allows to account for varying neuron models, parameter inaccuracies, and platform sizes. The suite has been

TABLE 8 | The table reports time and energy per inference in SNNs compared to ANN accelerators.

System	Batch-	Parallel	Accuracy in %		Time per Inf. in ms		E per Inf. in mJ	
	size	netw.	value	to Spikey	value	to Spikey	value	to Spikey
Spikey network (90.13% ANN accuracy)								
Coral edge TPU	1		**90.20**	5.04	0.05	0.03	0.3	0.1
Intel NCS 2	1		90.10	4.94	1.92	1.90	10.6	10.4
	200		90.10	4.94	0.12	0.10	0.6	0.4
GeNN-GPU		100	88.87	3.71	0.07	0.05	3.7	3.5
SpiNNaker		239	88.40	3.24	23.50	23.48	38.2	38.0
Spikey		1	85.16	0.00	**0.02**	0.00	**0.2**	0.0
				to SpiNN		to SpiNN		to SpiNN
Diehl network (98.84% ANN accuracy)								
Coral edge TPU	1		**98.85**	1.29	1.43	−4.83	7.7	3.9
Intel NCS 2	1		98.84	1.28	2.53	−3.73	13.8	10.0
	200		98.84	1.28	0.71	−5.55	**3.8**	0.0
GeNN-GPU		36	**98.85**	1.29	1.00	−5.26	181.6	177.8
SpiNNaker		53	98.77	1.21	172.49	166.23	188.5	184.7
GENN-GPU (TTFS)		10	97.60	0.04	**0.44**	−5.82	4.7	0.9
SpiNNaker (TTFS)		61	97.56	0.00	6.26	0.00	**3.8**	0.0

The batchsize is configured for ANN accelerators. The number of parallel instances of the same network is used for SNN simulations. Values are compared to the most efficient neuromorphic solution, too. Highlighted are the most accurate and the fastest/most efficient results.

[6] Results created with Open Vino 2021.3.

[7] The TPU was interfaced with the Pycoral Frogfish Release from February 2021.

deployed to a range of neuromorphic systems (from mixed-signal to fully digital) and SNN simulators, demonstrating the capabilities of the framework. Selected benchmarks have been presented and evaluated revealing hardware specific constraints for neural modeling and potential workarounds for issues encountered when using these systems. Benchmarks belong to three categories with varying closeness to full applications and extrapolation capabilities: low-level benchmarks revealed constraints that influence the available SNNs deployable to a given system. These constraints hold for every application, thus their relevance is quite broad. Application kernels, like the presented WTA architectures, represent a full class of networks, but still not solve a real task like object detection or CSP solving. These belong to the class of full application benchmarks, providing natural benchmark metrics but having only limited meaning for other applications implemented on neuromorphic hardware. We presented results for DNN inference, function approximation, spiking Sudoku solving, and SLAM. More results can be found in the **Supplementary Material**.

For future development of our benchmark framework, two directions are possible: due to the modular structure and the hardware abstraction layer, adding new platforms to the comparison is eased up to a certain extent. Here, possible candidates include Intel Loihi (Davies et al., 2018), BrainDrop (Neckar et al., 2019), or DYNAPs (Moradi et al., 2018). Furthermore, successors have been announced for systems discussed here (Billaudelle et al., 2019; Mayr et al., 2019). The second direction covers the implementation of new benchmarks. Most interesting is the embedding of the various direct training methods published within the recent years. These methods allow, similar to the hardware-in-the-loop approach discussed above, to encounter the neuron variability found in analogue circuitry.

One major argument for neuromorphic computing is the improvement in efficiency compared to algorithmic or standard DNN implementations. To validate this argument, we proposed a simple energy model relating costs of high-level SNN operations (e.g., action potential generation) to low-level energy costs. This model successfully predicts the energy budget of networks emulated on Spikey or simulated on SpiNNaker as long as low-level configurations would not deviate from the default. Here, especially changing the number of neurons per core on the SpiNNaker system leads to larger deviations, as the idle cost per neuron is increased. The energy model does not cover all features of a modern digital processor, thus energy predictions for GPU simulations were found to be insufficient. Nevertheless, the model allowed us to scale up the energy budget to a full brain simulation. Comparing these to the costs of the human brain we found mixed-signal hardware, being the most of efficient system in consideration, to lag behind by four orders of magnitude (even in this very optimistic and simplified upscaling). Furthermore, switching to a modernized fabrication technology, this gap cannot be closed in the short run. To conclude the energy related discussion, we presented a comparison to ANN accelerators and to a Raspberry Pi 4 for selected benchmarks. We demonstrated, that for small scaled Sudokus the SpiNNaker system was not fully utilized and thus the RPI4 is performing better. Only the mixed-signal system had superior time and energy to solution metrics. Similarly, this system is most efficient at DNN inference at the cost of accuracy. For SpiNNaker, switching to TTFS encoding resulted in an efficiency competitive to ANN accelerators, despite the rather old technology in which SpiNNaker cores are fabricated in.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found at: http://yann.lecun.com/exdb/mnist/; https://github.com/dannyneil/spiking_relu_conversion.

AUTHOR CONTRIBUTIONS

CO and CK conducted the experiments. MT and UR supervised this work and contributed with various corrections and comments. All authors contributed to writing the article and approved the submitted version.

FUNDING

This research leading to these results has received funding from the European Union Seventh Framework Programme (FP7) under grant agreement no 604102 and the EU's Horizon 2020 research and innovation programme under grant agreements Nos. 720270 and 785907 (Human Brain Project, HBP). It has been further supported by the Cluster of Excellence Cognitive Interaction Technology CITEC (EXC 277) at Bielefeld University, which was funded by the German Research Foundation (DFG). We acknowledge support for the publication costs by the Open Access Publication Fund of Bielefeld University and the Deutsche Forschungsgemeinschaft (DFG).

ACKNOWLEDGMENTS

We thank the Electronic Vision(s) group from Heidelberg University and Advanced Processor Technologies Research Group from Manchester University for access to their hardware systems and continuous support. We thank James Knight from the University of Sussex for support regarding the GeNN implementation. Tables and Figures have been previously published with the dissertation thesis of CO (Ostrau, 2022).

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fnins.2022.873935/full#supplementary-material

REFERENCES

Attwell, D., and Laughlin, S. B. (2001). An energy budget for signaling in the grey matter of the brain. *J. Cereb. Blood Flow Metab*. 21, 1133–1145. doi: 10.1097/00004647-200110000-00001

Azevedo, F. A. C., Carvalho, L. R. B., Grinberg, L. T., Farfel, J. M., Ferretti, R. E. L., Leite, R. E. P., et al. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *J. Compar. Neurol*. 513, 532–541. doi: 10.1002/cne.21974

Billaudelle, S., Stradmann, Y., Schreiber, K., Cramer, B., Baumbach, A., Dold, D., et al. (2019). Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate. *arXiv preprint arXiv:1912.12980*. doi: 10.1109/ISCAS45731.2020.9180741

Cao, Y., Chen, Y., and Khosla, D. (2015). Spiking deep convolutional neural networks for energy-efficient object recognition. *Int. J. Comput. Vis*. 113, 54–66. doi: 10.1007/s11263-014-0788-3

Coleman, C., Kang, D., Narayanan, D., Nardi, L., Zhao, T., Zhang, J., et al. (2019). Analysis of DAWNBench, a time-to-accuracy machine learning performance benchmark. *ACM SIGOPS Oper. Syst. Rev*. 53, 14–25. doi: 10.1145/3352020.3352024

Coleman, C., Narayanan, D., Kang, D., Zhao, T., Zhang, J., Nardi, L., et al. (2017). *DAWNBench: An End-to-End Deep Learning Benchmark and Competition*, Technical Report.

Davies, M. (2019). Benchmarks for progress in neuromorphic computing. *Nat. Mach. Intell*. 1, 386–388. doi: 10.1038/s42256-019-0097-1

Davies, M., Srinivasa, N., Lin, T.-H. H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Davies, M., Wild, A., Orchard, G., Sandamirskaya, Y., Guerra, G. A. F., Joshi, P., et al. (2021). Advancing neuromorphic computing with Loihi: a survey of results and outlook. *Proc. IEEE* 10, 1–24. doi: 10.1109/JPROC.2021.3067593

Davison, A. P. (2008). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform*. 2:11. doi: 10.3389/neuro.11.011.2008

Diehl, P., and Cook, M. (2015). Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Front. Comput. Neurosci*. 9:99. doi: 10.3389/fncom.2015.00099

Diehl, P. U., Neil, D., Binas, J., Cook, M., Liu, S.-C. C., Pfeiffer, M., et al. (2015). "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," *Proceedings of the International Joint Conference on Neural Networks*, doi: 10.1109/IJCNN.2015.7280696

Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The LINPACK Benchmark: past, present and future. *Concurr. Comput*. 15, 803–820. doi: 10.1002/cpe.728

Eliasmith, C., and Anderson, C. H. (2004). Neural engineering.

Eppler, J. M. (2008). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform*. 2:12. doi: 10.3389/neuro.11.012.2008

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The SpiNNaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., et al. (2013). Overview of the SpiNNaker system architecture. *IEEE Trans. Comput*. 62, 2454–2467. doi: 10.1109/TC.2012.142

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural simulation tool). *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430

Golosio, B., Tiddia, G., De Luca, C., Pastorelli, E., Simula, F., and Paolucci, P. S. (2021). Fast simulations of highly-connected spiking cortical models using GPUs. *Front. Comput. Neurosci*. 15:627620. doi: 10.3389/fncom.2021.627620

Hopkins, M., and Furber, S. (2015). Accuracy and efficiency in fixed-point neural ODE solvers. *Neural Comput*. 27, 2148–2182. doi: 10.1162/NECO_a_00772

Howarth, C., Gleeson, P., and Attwell, D. (2012). Updated energy budgets for neural computation in the neocortex and cerebellum. *J. Cereb. Blood Flow Metab*. 32, 1222–1232. doi: 10.1038/jcbfm.2012.35

Jordan, J., Mørk, H., Vennemo, S. B., Terhorst, D., Peyser, A., Ippen, T., et al. (2019). *NEST 2.18.0* (2.18.0). *Zenodo*. doi: 10.5281/zenodo.2605422

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci*. 1, 136–142. doi: 10.1038/s43588-020-00022-7

Kreiser, R., Cartiglia, M., Martel, J. N., Conradt, J., and Sandamirskaya, Y. (2018a). "A neuromorphic approach to path integration: a head-direction spiking neural network with vision-driven reset," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)* (Florence), 1–5. doi: 10.1109/ISCAS.2018.8351509

Kreiser, R., Renner, A., Leite, V. R. C., Serhan, B., Bartolozzi, C., Glover, A., et al. (2020). An on-chip spiking neural network for estimation of the head pose of the iCub robot. *Front. Neurosci*. 14:551. doi: 10.3389/fnins.2020.00551

Kreiser, R., Renner, A., Sandamirskaya, Y., and Pienroj, P. (2018b). "Pose estimation and map formation with spiking neural networks: towards neuromorphic SLAM," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Madrid), 2159–2166. doi: 10.1109/IROS.2018.8594228

Lennie, P. (2003). The cost of cortical computation. *Curr. Biol*. 13, 493–497. doi: 10.1016/S0960-9822(03)00135-0

Maass, W. (2014). Noise as a resource for computation and learning in networks of spiking neurons. *Proc. IEEE* 102, 860–880. doi: 10.1109/JPROC.2014.2310593

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., et al. (2020). "MLPerf training benchmark," in *Proceedings of Machine Learning and Systems, Vol. 2*, eds I. Dhillon, D. Papailiopoulos, and V. Sze, p. 336–349. Available online at: https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf

Mayr, C., Hoeppner, S., and Furber, S. (2019). SpiNNaker 2: A 10 million core processor system for brain simulation and machine learning. *arXiv preprint arXiv:1911.02385*, 10–13.

Moradi, S., Qiao, N., Stefanini, F., and Indiveri, G. (2018). A scalable multi-core architecture with heterogeneous memory structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs). *IEEE Trans. Biomed. Circuits Syst*. 12, 106–122. doi: 10.1109/TBCAS.2017.2759700

Neckar, A., Fok, S., Benjamin, B. V., Stewart, T. C., Oza, N. N., Voelker, A. R., et al. (2019). Braindrop: a mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proc. IEEE* 107, 144–164. doi: 10.1109/JPROC.2018.2881432

Neftci, E. O., Mostafa, H., and Zenke, F. (2019). Surrogate gradient learning in spiking neural networks: bringing the power of gradient-based optimization to spiking neural networks. *IEEE Sign. Process. Mag*. 36, 51–63. doi: 10.1109/MSP.2019.2931595

Ostrau, C. (2022). Energy and Performance Estimation for Neuromorphic Systems. *Dissertation Thesis*. Bielefeld University. doi: 10.4119/unibi/2962759

Ostrau, C., Homburg, J., Klarhorst, C., Thies, M., and Rückert, U. (2020a). "Benchmarking deep spiking neural networks on neuromorphic hardware," in *Artificial Neural Networks and Machine Learning-ICANN 2020* (Bratislava: Springer International Publishing), 610–621. doi: 10.1007/978-3-030-61616-8_49

Ostrau, C., Klarhorst, C., Thies, M., and Rückert, U. (2019). "Comparing neuromorphic systems by solving sudoku problems," in *2019 International Conference on High Performance Computing & Simulation (HPCS)* (Dublin), 521–527. doi: 10.1109/HPCS48598.2019.9188207

Ostrau, C., Klarhorst, C., Thies, M., and Rückert, U. (2020b). "Benchmarking of neuromorphic hardware systems," in *Neuro-inspired Computational Elements Workshop (NICE'20)* (Heidelberg), 1–4. doi: 10.1145/3381755.3381772

Petrovici, M. A., Vogginger, B., Müller, P., Breitwieser, O., Lundqvist, M., Muller, L., et al. (2014). Characterization and compensation of network-level anomalies in mixed-signal neuromorphic modeling platforms. *PLoS ONE* 9:e108590. doi: 10.1371/journal.pone.0108590

Pfeil, T., Grübl, A., Jeltsch, S., Müller, E., Müller, P., Petrovici, M. A., et al. (2013). Six networks on a universal neuromorphic computing substrate. *Front. Neurosci*. 7:11. doi: 10.3389/fnins.2013.00011

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., et al. (2020). "MLPerf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (Valencia), 446–459. doi: 10.1109/ISCA45697.2020.00045

Rhodes, O., Bogdan, A., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., et al. (2018). *sPyNNaker: A Software Package for Running PyNN Simulations on SpiNNaker*. doi: 10.3389/fnins.2018.00816

Rhodes, O., Peres, L., Rowley, A. G. D., Gait, A., Plana, L. A., Brenninkmeijer, C., et al. (2020). Real-time cortical simulation on neuromorphic hardware. *Philos. Trans. R. Soc. A* 378:20190160. doi: 10.1098/rsta.2019.0160

Rosing, J., and Slater, E. (1972). The value of ΔG° for the hydrolysis of ATP. *Biochim. Biophys. Acta* 267, 275–290. doi: 10.1016/0005-2728(72)90116-8

Rowley, A. G. D., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., Lester, D. R., et al. (2018). SpiNNTools: the execution engine for the SpiNNaker platform. *Front Neurosci*. 13:231. doi: 10.3389/fnins.2019.00231

Rubino, A., Payvand, M., and Indiveri, G. (2019). "Ultra-low power silicon neuron circuit for extreme-edge neuromorphic intelligence," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (Genoa), 458–461. doi: 10.1109/ICECS46596.2019.8964713

Rueckauer, B., and Liu, S.-C. (2018). "Conversion of analog to spiking neural networks using sparse temporal coding," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)* (Florence), 1–5. doi: 10.1109/ISCAS.2018.8351295

Rueckauer, B., Lungu, I.-A., Hu, Y., Pfeiffer, M., and Liu, S.-C. (2017). Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Front. Neurosci*. 11:682. doi: 10.3389/fnins.2017.00682

Schemmel, J., Briiderle, D., Griibl, A., Hock, M., Meier, K., and Millner, S. (2010). "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems* (Paris), 1947–1950. doi: 10.1109/ISCAS.2010.5536970

Schmitt, S., Klahn, J., Bellec, G., Grubl, A., Guttler, M., Hartel, A., et al. (2017). "Neuromorphic hardware in the loop: training a deep spiking network on the BrainScaleS wafer-scale system," in *2017 International Joint Conference on Neural Networks (IJCNN)* (Anchorage, AK), 2227–2234. IEEE. doi: 10.1109/IJCNN.2017.7966125

Stöckel, A., Jenzen, C., Thies, M., and Rückert, U. (2017). Binary associative memories as a benchmark for spiking neuromorphic hardware. *Front. Comput. Neurosci*. 11:71. doi: 10.3389/fncom.2017.00071

Sun, Y., Agostini, N. B., Dong, S., and Kaeli, D. (2019). Summarizing CPU and GPU design trends with product data. *arXiv preprint arXiv:1911.11313.*

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Front. Neurosci*. 12:291. doi: 10.3389/fnins.2018.00291

Yavuz, E., Turner, J., and Nowotny, T. (2016). GeNN: a code generation framework for accelerated brain simulations. *Sci. Rep*. 6:18854. doi: 10.1038/srep18854

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

TECHNOLOGY AND CODE
published: 17 June 2022
doi: 10.3389/fninf.2022.882552

Scaling and Benchmarking an Evolutionary Algorithm for Constructing Biophysical Neuronal Models

Alexander Ladd[1]*, *Kyung Geun Kim*[1], *Jan Balewski*[2], *Kristofer Bouchard*[3,4] and *Roy Ben-Shalom*[5]*

[1] Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA, United States, [2] NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA, United States, [3] Helen Wills Neuroscience Institute & Redwood Center for Theoretical Neuroscience, University of California, Berkeley, Berkeley, CA, United States, [4] Scientific Data Division and Biological Systems and Engineering Division, Lawrence Berkeley National Laboratory, Berkeley, CA, United States, [5] Neurology Department, MIND Institute, University of California, Davis, Sacramento, CA, United States

OPEN ACCESS

Edited by:
Felix Schürmann,
Swiss Federal Institute of Technology
Lausanne, Switzerland

Reviewed by:
Michael Hines,
Yale University, United States
Nathan Gouwens,
Allen Institute for Brain Science,
United States
Werner Van Geit,
Swiss Federal Institute of Technology
Lausanne, Switzerland

***Correspondence:**
Alexander Ladd
zladd@berkeley.edu
Roy Ben-Shalom
rbenshalom@ucdavis.edu

Received: 23 February 2022
Accepted: 18 May 2022
Published: 17 June 2022

Citation:
Ladd A, Kim KG, Balewski J, Bouchard K and Ben-Shalom R (2022) Scaling and Benchmarking an Evolutionary Algorithm for Constructing Biophysical Neuronal Models.
Front. Neuroinform. 16:882552.
doi: 10.3389/fninf.2022.882552

Single neuron models are fundamental for computational modeling of the brain's neuronal networks, and understanding how ion channel dynamics mediate neural function. A challenge in defining such models is determining biophysically realistic channel distributions. Here, we present an efficient, highly parallel evolutionary algorithm for developing such models, named *NeuroGPU-EA*. *NeuroGPU-EA* uses CPUs and GPUs concurrently to simulate and evaluate neuron membrane potentials with respect to multiple stimuli. We demonstrate a logarithmic cost for scaling the stimuli used in the fitting procedure. *NeuroGPU-EA* outperforms the typically used CPU based evolutionary algorithm by a factor of 10 on a series of scaling benchmarks. We report observed performance bottlenecks and propose mitigation strategies. Finally, we also discuss the potential of this method for efficient simulation and evaluation of electrophysiological waveforms.

Keywords: biophysical neuron model, high performance computing, evolutionary algorithms, non-convex optimization, strong scaling, weak scaling, electrophysiology

1. INTRODUCTION

Since Hodgkin and Huxley's seminal work on recording and mathematically formulating the activity of the giant squid axon (Hodgkin and Huxley, 1952), great progress has been made in understanding the electrical properties of single neuron models. The development of the patch-clamp technique (Sakmann and Neher, 1984), which enabled recording neurons in finer resolution, and the work by Rall (1959, 1962, 1964) on modeling the cable properties of dendritic trees have been the basis of extensive research in numerical methods for compartmental neuron models (Rall, 2009). The formulation of electrical properties of neurons in digital computers (Hines, 1984; Carnevale and Hines, 2006; Bower and Beeman, 2012) enabled simulating experimental observation in computational models (Traub et al., 1991, 2005; De Schutter and Bower, 1994; Mainen et al., 1995). These advancements have brought the field of computational neuroscience closer to realistically modeling biological neurons on computers (Markram et al., 2015; Nogaret et al., 2016; Ben-Shalom et al., 2017; Bouchard et al., 2018; Gouwens et al., 2018; Daou and Margoliash, 2020; Spratt et al., 2021). Multi-compartmental biophysical models, such as the Mainen & Sejnowski model (Mainen and Sejnowski, 1996) and those comprising the large-scale neocortical column simulation (Markram et al., 2015; Ramaswamy et al., 2015; Gouwens et al., 2018; Billeh et al., 2020), aim to simulate the electrical properties of single neuron. The NEURON (Hines et al., 2004; Carnevale and Hines, 2006) simulation environment is a commonly used software for simulating how different channel conductances contribute to the electrical activity of the neuron. However, constraining the conductance of various membrane ion channels and biophysical properties of the membrane remains a major obstacle in fitting these models to experimental data (Prinz et al., 2004; Druckmann et al., 2007; Almog and Korngreen, 2016; Nogaret et al., 2016). As the number of free parameters that characterize the neuronal model increase, so does the cardinality of the optimization search space, thus making the optimization less tractable. Adding more parameters that govern channel and membrane dynamics makes the simulated neuron more specific to a physical neuron, but also adds more unknown variables with complex relationships. Thus, there exist trade-offs between the amount of detail in the model, computation time, computational power, and the questions that need to be answered by such models (Eliasmith and Trujillo, 2014; Almog and Korngreen, 2016; Sáray et al., 2020). Researchers must make limiting assumptions to constrain the number of free parameters to maintain feasible simulation and model fitting times.

With increasing model complexity comes the need for more efficient optimization methods. One challenge with constraining the parameters of electrophysiological neuron models is that the search space of possible model parameters is large. Furthermore, neurons with substantially different parameters can produce qualitatively similar responses (Goldman et al., 2001; Golowasch et al., 2002; Prinz et al., 2003). However, a small perturbation in the conductance of a single channel parameter can have a significant impact on the simulated voltage trace. In constraining single neuron parameters, there are several approaches including brute force search, Monte Carlo optimization algorithms such as evolutionary algorithms and simulated annealing, or heuristic algorithms (Keren et al., 2005; Druckmann et al., 2007; Van Geit et al., 2007, 2008, 2016). For the construction of biophysical neuron models in this paper, we chose to use the evolutionary algorithm (EA), a prevalent method for this optimization problem (Vanier and Bower, 1999; Keren et al., 2005; Druckmann et al., 2011; Masoli et al., 2017; Gouwens et al., 2018; Ben-Shalom et al., 2020). Our objective function is constructed from score functions comparing electrophysiological firing properties between simulated and experimental target voltages (Druckmann et al., 2007). This multi-objective optimization (MOO) is formulated using the Indicator-Based evolutionary algorithm (IBEA) (Zitzler and Künzli, 2004). EA searches for solutions that present optimal trade-offs between electrophysiological score functions. We focused on efficiently scaling EA to mitigate computational bottlenecks and highlight potential benefits. We considered the construction of the objective function outside the scope of this work. We show the motivation for accelerating this algorithm through scaling the parameter search algorithm on a motivating example model.

Advancements in chip capacity (Schaller, 1997) and software for high performance computing (HPC) platforms (Fan et al., 2004; Strohmaier et al., 2015) have the potential to accelerate electrophysiological simulation (Bouchard et al., 2018) and consequently the EA algorithm. We focused on benchmarking two classes of software modules—neuron simulators and electrophysiological spike train feature extractors, due to their central importance in EA. While it is important to experiment with performance benchmarks that are specific to individual modules it is also important to develop benchmarks that assess the application of combinations of modules. This study draws from previous work in benchmarking for computer science (Hoefler and Belli, 2015; Bouchard et al., 2016; Coleman et al., 2019; Wu et al., 2019) by applying established performance benchmarks to software for neuron simulation and biophysical modeling. These experiments utilize two well-established benchmarking strategies: strong scaling and weak scaling (Bailey et al., 2010; Balasubramanian et al., 2020). In total, we used three benchmarks:

- "Compute Fixed and Problem Scales": The number of neuron models used in EA increases across experiments but the allocation of computing nodes, cores, and/or GPUs available is fixed.
- "Compute Scales and Problem Fixed" or strong scaling: The allocation of computing nodes, cores, and/or GPUs increases across experiments but the number of neuron models used in EA is fixed.
- "Compute Scales and Problem Scales" or weak scaling: The allocation of computing nodes, cores, and/or GPUs and the number of neuron models used in EA both increase across experiments at a fixed ratio.

These experiments investigate the impact of modularizing EA using different software tools for simulation and electrophysiological feature extractors. Using this experimental design in conjunction with various software and hardware configurations demonstrates the state of the art, challenges, and opportunities, related to efficiently utilizing HPC resources for complex biophysical neuronal modeling.

Adapting well-known performance benchmarks to EA helps understand how the algorithm can scale using different configurations of computational resources and software modules. While (Knight and Nowotny, 2018; Van Albada et al., 2018; Criado et al., 2020; Kulkarni et al., 2021), all provide relevant examples of benchmarking simulation modules and computational platforms, such as neuromorphic hardware, there is a gap in benchmarking the performance of such simulators applied to the neuron fitting problem. This work aims to address this gap by evaluating the run time performance of the evolutionary algorithm as a method to construct biophysical neuron models. Thus, the principal contributions of this paper are as follows:

1. We present an optimized implementation of the evolutionary algorithm, *NeuroGPU-EA*, that aims to accelerate the time it takes to fit a biophysical neuronal model by leveraging parallelism on high performance GPU nodes.

2. We benchmark the run time of this algorithm using well-established performance benchmarks *weak scaling* and *strong scaling*.
3. We vary implementation by:

   3a. Running experiments on CPU only nodes with the *CPU-EA* algorithm or CPU-GPU experiments with *NeuroGPU-EA* algorithm.
   3b. Using different electrophysiological feature extraction libraries.
   3c. Using different GPU-based neuron simulation modules, such as CoreNeuron in *CoreNeuron-EA*.

In the following sections of this paper, we first give a brief overview of the implementation of the evolutionary algorithm and how simulation and feature extraction drive the algorithm toward increasingly realistic neuronal modeling. Next, we specify the hardware and software on the machines we used. Then we give a description of National Energy Research Scientific Computer Center's (NERSC) supercomputer Cori[1], which was used to test the scaling of each variation of this algorithm. The experimental design allows for the comparison of different algorithms, using GPU and CPU, as well as different software modules in the simulate-evaluate loop. Subsequently, we demonstrate the results of such experiments and discuss the implications. We show how scaling the evolutionary algorithm for an example cell results in a more realistic model. Finally, we discuss challenges faced in benchmarking EA and future steps for analysis.

2. METHODS

2.1. Evolutionary Algorithm

The optimization problem considered in this paper is the fitting of biophysically accurate parameters of a neuron using evolutionary algorithms (EAs). EAs are a class of optimization methods that rely on natural selection in a population through biologically inspired operators such as mutation, crossover, and fitness-based selection (Mitchell, 1998). This version of EA encodes solutions to an optimization problem into continuous vector representations of neuron model parameters. We refer to this group of parameterized neuron models as the "population" and a single model as an "individual". EAs represent the quality of these vector representations by evaluating an objective function that takes this population as an input and compares the models' responses to experimental data. The algorithm is known as the (μ, λ) evolutionary algorithm (Beyer and Schwefel, 2002; Beyer, 2007) and is implemented using DEAP (Fortin et al., 2012) and BluePyOpt[2] (Van Geit et al., 2016). In this implementation, μ and λ are the size of the parent population and the number of offspring to produce for the next generation, respectively. The parameter *cxRate* is the probability that an offspring was produced by crossover and the parameter *mutRate* is the probability that an offspring is produced *via* mutation[3]. Mutation is a perturbation of one or more parameters and crossover is a combination between a pair of parameter sets. The function VARIATION in the EA algorithm, **Algorithm 1**, applies mutation, reproduction, or crossover exclusively to each individual, or pair in the case of crossover, to produce λ new offspring from a μ sized parent generation.

Algorithm 1 Evolutionary Algorithm

```
procedure OPTIMIZE($\mu$, $\lambda$, cxRate, mutRate nGenerations)
    parents ← INITIALIZE()
    hof ← []
    parents.scores ← EVALUATE(parents)
    for generation ← 1, nGenerations do
        offspring ← VARIATION(parents, $\lambda$, cxRate, mutRate)
        offspring.scores ← EVALUATE(offspring)
        population ← parents + offspring
        parents ← SELECT(population, $\mu$)    ▷ keep $\mu$ individuals using indicator value tournament selection (Zitzler and Künzli, 2004)
        hof ← hof.update(population) ▷ hof tracks 10 lowest scoring models
    end
    return $\underset{hof_i \in hof}{\operatorname{argmin}}$ sum(hof_i.scores) ▷ the best model has the lowest sum of scores
```

Algorithm 2 Objective Function

```
procedure EVALUATE(offspring)
    scores ← {}
    for all stim ∈ Stims do    ▷ stimuli parallelism
        responses ← SIMULATE(offspring,stim)
        for all scoreFunction ∈ scoreFunctions do    ▷ score function parallelism
            scores[scoreFunction] ← scoreFunction(responses, target)
        end
    end
    return scores
```

Formally, the optimization problem posed in this paper defines an individual i as $\boldsymbol{x_i} \in \mathbb{R}^{13}$. Boldface $\boldsymbol{x}$ denotes a one-dimensional vector. The entire population is defined as $X \in \mathbb{R}^{13 \times N}$, where 13 is the number of free parameters of the neuron model and N is the size of the population (typically 50–5,000). The OBJECTIVE FUNCTION is computed using electrophysiological score functions, thus the term "score function" refers to one electrophysiological feature and the term objective function refers to the function characterizing the joint optimization such score functions (MOO). Initially, a model $\boldsymbol{x_i}$ is simulated

[1] https://docs.nersc.gov/systems/cori/

[2] https://github.com/BlueBrain/BluePyOpt

[3] https://deap.readthedocs.io/en/master/api/algo.html#deap.algorithms.eaMuPlusLambda

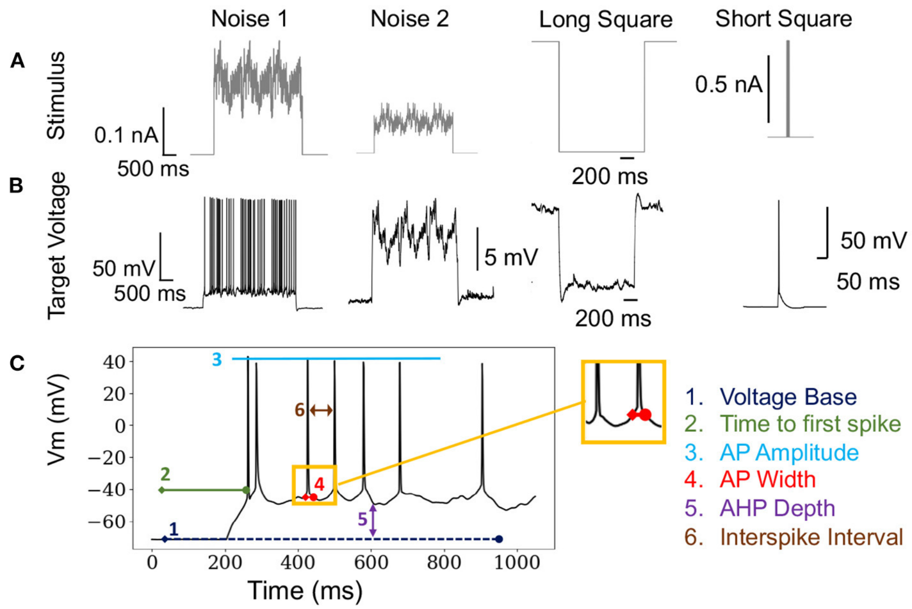


FIGURE 1 | Stimuli and electrophysiological score functions used in algorithm: **(A)** Various stimuli used in the fitting procedure of EA. **(B)** Corresponding target voltages that are recorded from patch clamp experiments as a result of the stimuli in **(A)**. **(C)** Demonstrates how electrophysiological score functions are computed on a single trace. These score functions are used to compare target and simulated firing traces.

using $s \in S$ stimuli, shown in **Figure 1A**, and evaluated against an experimental waveform, shown in **Figure 1B**, using F electrophysiological score functions, shown in **Figure 1C**. This procedure results in a set of scores for each individual. These scores are computed across each stimuli and score function (Druckmann et al., 2007). Then, BluePyOpt (Van Geit et al., 2016) uses an indicator based objective function (IBEA) that computes binary comparisons between individuals and their respective electrophysiological scores. These comparisons are calculated using the sum of indicator functions of the form $I(\{\boldsymbol{x}_i\}, \{\boldsymbol{x}_j\})$, resulting in an indicator based fitness value, as referenced in line 6 of **Algorithm 1**. This definition of fitness is derived from Zitzler and Künzli (2004). The aforementioned μ individuals are obtained through an iterative process that acquires the winner of a tournament of binary comparisons between all individuals until μ individuals have been selected. The selected μ individuals, termed the "parents", are used to produce a new set of offspring for the subsequent generation, as demonstrated in line 4 of **Algorithm 1**. After all individuals in the population, consisting of offspring and parents, are scored, the 10 individuals with the lowest sum of score functions are added to a hall of fame. The hall of fame has no impact on the evolution of the population, as it tracks the 10 lowest scoring individuals over all generations of EA. When the EA algorithm has terminated, on line 11 **Algorithm 1**, the lowest scoring individual is selected from the hall of fame.

In total, we used a set 18 stimulations consisting of 8 long square, 6 noisy, 2 short square, and 2 ramp stimuli, as represented in **Figure 1A**. In benchmarking tasks 3.2,3.3, 3.4 and 3.5, stimuli were chosen in a random order. The optimization in Section 3.6 used the same stimuli as benchmarking tasks 3.2,3.3, 3.4. However, the optimization in Section 3.7 only utilized the 8 long square stimuli. We chose to benchmark a diverse set of stimuli as the practice of EA for fitting neuron model parameters utilizes a wide range of stimuli, including passive stimuli not represented in this study. The full list of score functions is included as a **Supplementary Material**.

2.2. Implementations

In our implementation of EA, we used 20 scoring electrophysiological score functions from Blue Brain Project's Electrophys Feature Extraction Library (eFEL) library[4] (Van Geit et al., 2016). The total offspring score is the unweighted sum of the selected score functions. **Figure 1C** is an illustration of how these scoring functions are computed on a single trace. The size of EA is defined as having cardinality $N \times S \times F$, which represents the population size × the number of stimuli presented × the number of score functions used. As mentioned above, the population for the evolutionary algorithm is comprised of parameter sets for the multi-compartment neuron model. We used a layer 5 thick tufted pyramidal neuron from the Blue Brain Project (Ramaswamy et al., 2015) with 13 different ionic channel parameters in the axon, soma, and dendrite. This cell morphology and parameterization can be found in Blue Brain Model portal[5] as L5 TTPC1 cADpyr232 1. The **Supplementary Table 1** shows how the parameters were distributed across axonal, somatic sections, as well as the

[4] https://efel.readthedocs.io/en/latest/

[5] https://bbp.epfl.ch/nmc-portal/downloads.html

upper and lower optimization bounds for each conductance. **Supplementary Table 1** also shows that some of these parameters were modeled separately in the soma and the axon. The model used in Section 3.7, **Figure 7**, and **Supplementary Figure 1** does not include a parameter for non-specific cation current I_h but the model used in the benchmarking Sections 3.2, 3.3, and 3.4 did include this parameter. This channel was not included in the optimization to reduce the complexity of the optimization task.

In the objective function **Algorithm 2** there are three opportunities to implement parallelism:

1. **Population level parallelism**: run the simulate-evaluate loop in parallel across the entire population.
2. **Stimuli parallelism**: run all the simulations for each stimulus in parallel.
3. **Score function parallelism**: run all the score functions in parallel.

In the objective function **Algorithm 2**, scores and responses are lists containing the voltages and scores for each individual of the population respectively. The objective function can be implemented as a triple `for` loop by including an initial loop over the population. Alternatively, **Algorithm 2** implements a double `for` loop by defining `scores` as a vector of scores corresponding to each individual. Each stimulus response and score are computed without using information about other simulations, other electrophysiological score functions, or individuals in the population. Thus, the problem is embarrassingly parallel (Herlihy and Shavit, 2012). For reference, the sequential representation is summarized in **Figure 2A**. Our *CPU-EA* and *NeuroGPU-EA* algorithm took advantage of this feature in the following ways.

- *NeuroGPU-EA* employed all three approaches to implement parallelism, as demonstrated in **Figure 2C**. (i) The population level parallelism was achieved by dividing the entire population (`MPI_SCATTER`) across nodes and then aggregating (`MPI_GATHER`) at the rank 0 node at the end of evaluation. (ii) The simulations for each stimuli were computed in parallel across each available GPU. (iii) Each electrophysiological score function was computed in parallel on CPUs once the simulation responses were obtained.
- *CPU-EA* implementation, shown in **Figure 2B**, was parallelized over the population and one CPU core per individual was allocated using IPyParallel[6]. The parallelized *CPU-EA* procedure was run in parallel across the entire population (`MAP`) and aggregated (`REDUCE`) into a list once all scores have been calculated. Thus, *CPU-EA* leverages population level parallelism across all available CPU cores.

For *NeuroGPU-EA*, if there are more stimuli than GPUs available, it is necessary to launch batches of simulations while the CPU cores handle electrophysiological score function evaluation for the previous batch of stimuli. This case is demonstrated in **Figure 2D** and will be referenced in Section 3.5 in experiments that scale up the number stimuli used in EA. We compute scores on CPU and acquire additional CPU-GPU data transfer cost because we did not have access to a GPU implementation of the eFEL library.

2.3. Hardware

The experiments presented here were executed on the Cori-GPU cluster at NERSC[7]. Each Cori GPU node has two sockets of 20-core Intel Xeon Gold 6148 ("Skylake") CPUs with 384 GB DDR4 RAM memory and a clock rate of 2.40 GHz. Each of these nodes also has 8 NVIDIA Tesla V100 ("Volta") GPUs, connected with NVLink interconnect, each with 16 GB HBM2 memory. We used Cray's Programming Environment version 2.6.2. Allocated nodes were chosen by the batch system (SLURM 20.11.8) and were allocated exclusively to eliminate on-node interference. The system uses InfiniBand host network adapters (HCA) and network interface cards (NICs). Each Cori CPU node has two sockets, each socket is populated with a 2.3 GHz 16-core Haswell Intel Xeon Processor E52698 v3. Each core supports 2 hyperthreads, and has two 256-bit-wide vector units 36.8 Gflops/core (theoretical peak), 1.2 TFlops/node (theoretical peak) and 2.81 PFlops total (theoretical peak). Each node has 128 GB DDR4 2133 MHz memory (four 16 GB DIMMs per socket) and 298.5 TB total aggregate memory. The interconnect is Cray Aries with Dragonfly topology with 45 TB/s global peak bisection bandwidth. We used Cray's Programming Environment version 2.6.2. Allocated nodes were chosen by the batch system (SLURM 20.11.8) and were allocated exclusively to eliminate on-node interference. For all experiments, we used Cori SCRATCH which is a Lustre file system designed for high performance temporary storage of large files. All experiments were run on x86 64 computing architecture, SUSE Linux Enterprise 15 and kernel 4.12.14-150.75-default.

2.4. Software

We used GCC compiler version 8.3.0, CUDA version 11.1.1, OpenMPI version 4.0.3, Python 3.8.6. As in previous work (Ben-Shalom et al., 2012, 2013, 2022), the evolutionary algorithm was implemented using the DEAP 1.3 (Fortin et al., 2012) and BluePyOpt 1.9.126 (Van Geit et al., 2016) python libraries. Score functions were implemented using Blue Brain Project's Electrophys Feature Extract Library 3.2.4 (eFEL)[8] (Van Geit et al., 2016) and Allen Institutes's IPFX[9]. The CPU based neuron simulations were run using NEURON 7.6.7 .mod files and the NEURON python interface[10]. the software versions and requirements are added as **Supplementary Materials**. For GPU based neuron simulations we used NeuroGPU 1.0[11] (Ben-Shalom et al., 2022) and CoreNeuron 1.0[12] (Kumbhar et al., 2019). For the installation of CoreNeuron we used the Intel PGI compiler, version 20.11-0 and we used Cori's cray-python version 3.7.3.2 to avoid compilation issues with anaconda python used in other experiments.

[6] https://ipyparallel.readthedocs.io/en/latest/

[7] https://docs-dev.nersc.gov/cgpu/

[8] https://github.com/BlueBrain/eFEL

[9] https://github.com/AllenInstitute/ipfx

[10] https://neuron.yale.edu/neuron/

[11] https://github.com/roybens/NeuroGPU

[12] https://github.com/BlueBrain/CoreNeuron

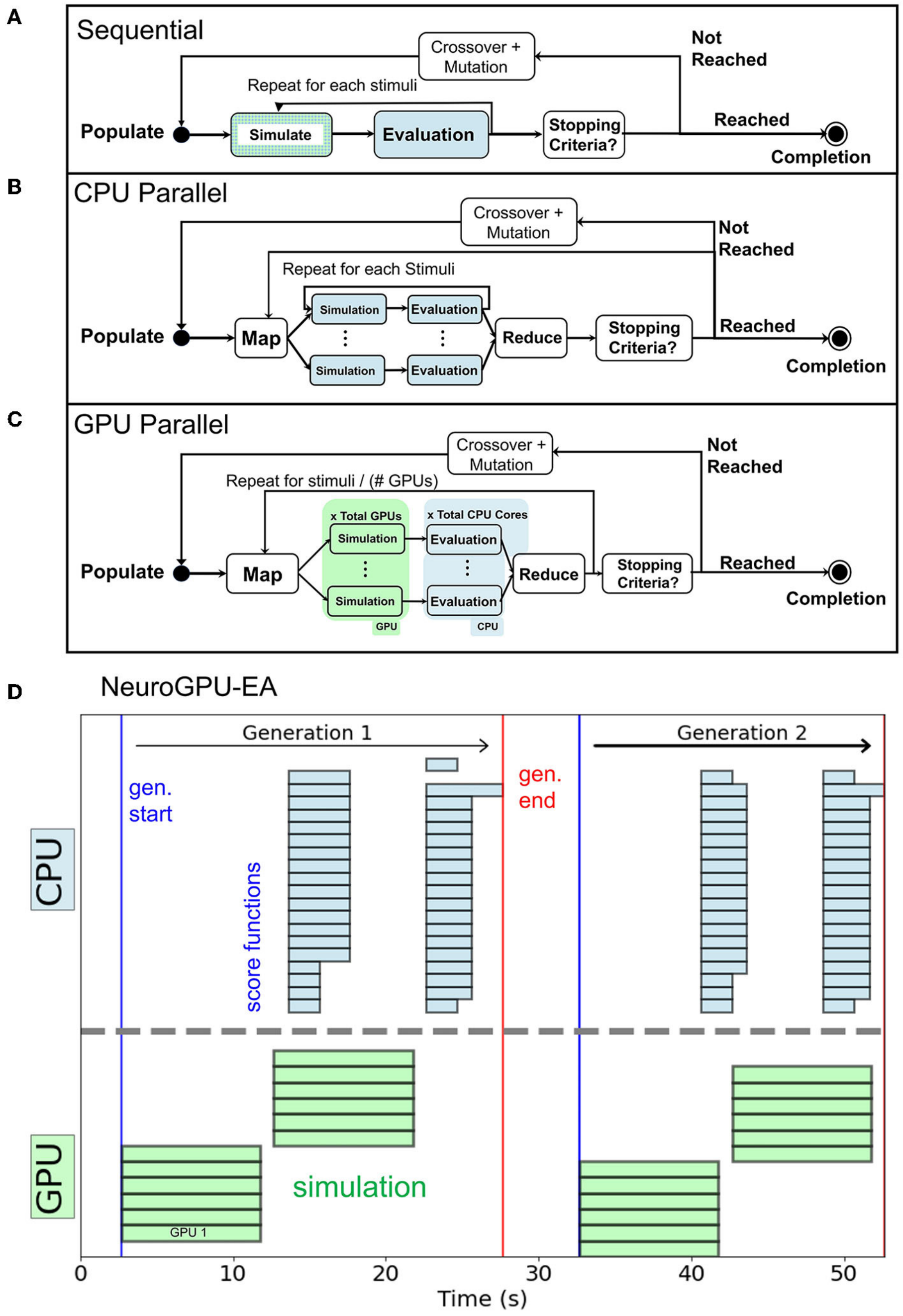


FIGURE 2 | (A) Sequential execution EA. **(B)** *CPU-EA* maps the simulation/evaluation of a model to a single core. **(C)** *GPU-EA* maps each stimuli to a GPU, then scores the simulation in parallel on each CPU core. **(D)** Timeline of *NeuroGPU-EA* for two generations. The algorithm starts new stimuli on GPUs while the CPUs are still completing the previous ones.

3. RESULTS

3.1. Experimental Design

The primary metric of EA performance was the wall time needed to complete one simulation-evaluation step. The three main experimental contexts are *NeuroGPU-EA*, *CoreNeuron-EA*, and *CPU-EA*. The version of *NeuroGPU-EA* that uses CoreNeuron is termed *CoreNeuron-EA*. We refer to both *NeuroGPU-EA* and *CoreNeuron-EA* as *GPU-EA* to represent GPU based evolutionary algorithms. *CPU-EA* experiments are run on CPU only nodes with 64 single-threaded cores. Unlike simulators used in *GPU-EA*, *CPU-EA* using NEURON offers an adaptive timestep option, with command `h.cvode_active(1)`, which allows the simulator to perform fewer integration solves when there

TABLE 1 | Compute fixed and problem scales: Stimuli and score functions are fixed 8 and 20, respectively.

Population	NeuroGPU-EA run time (s)	CPU-EA run time (s)	CoreNeuronGPU-EA run time (s)
500	36.8 ± 5.71	401 ± 82.4	58.7 ± 2.72
1,000	70.6 ± 8.83	679 ± 98.6	89.2 ± 5.25
1,500	91.6 ± 5.52	1,000 ± 159	123 ± 12.8
2,000	123 ± 8.88	1,380 ± 285	151 ± 53
2,500	150 ± 6.48	1,740 ± 296	182 ± 6.92
3,000	182 ± 3.98	1,930 ± 490	210 ± 3.58
3,500	212 ± 3.56	2,270 ± 494	242 ± 3.72
4,000	242 ± 10.4	-	272 ± 3.78
4,500	270 ± 4.87	-	304 ± 7.77
5,000	295 ± 11.8	-	333 ± 9.52

Each experiment uses one node. CPU node has 64 cores. GPU nodes have 80 CPU cores and 8 GPUs. ± values indicate 1 standard deviation.

are fewer spikes. *CPU-EA* uses the `h.cvode_active(1)` setting for applicable stimuli as this setting accelerates NEURON simulation time. As demonstrated in **Supplementary Figure 1**, NEURON with adaptive timestep had a notably faster average simulation time than standard NEURON settings. For benchmarking experiments, 50 trials were run using an initial population with the same seed. **Supplementary Figure 2** shows that running *NeuroGPU-EA* trials with multiple seeds resulted in a slight speedup for the 500 and 1,000 populations, but also resulted in more deviation between recorded times for these population sizes. For all experiments, the first generation of every optimization was discarded so the time spent loading the morphology of model neurons was not included. Morphology loading time was not included for *GPU-EA* because NeuroGPU begins with a mapping of the model in the GPU, while CoreNeuron had an initial cost of 0.35 s per model to load the morphology[13]. Further benchmarking of how different topologies, models, and morphologies affect simulation run time can be found in Figures 3, 4 of previous work (Ben-Shalom et al., 2013). The *GPU-EA* model transfer to CPU was not intentionally benchmarked either, as the NeuroGPU model only exists on the GPU. However, logs from CoreNeuron trials indicate an average cost of 3 ms for moving a single model to the GPU. Furthermore, CoreNeuron outputs indicated that a single model used 560 kB of memory. CPU experiments that were not ran for enough trials are not represented. We report the mean and standard deviation of the run time. We provide run time lower bounds as ideal scaling measures, in accordance with Hoefler and Belli (2015). To confirm these benchmarks are practicable, we include the optimized model responses and statistics at key generations for EA with population size 1,000 in **Supplementary Materials**.

3.2. Benchmark 1

The "Compute Fixed Problem Scales" benchmark measures the computational capacity of the algorithm with a fixed resource allocation. The problem scales with increases in the population size, N, at increments of 500 until the population size reaches 5,000. "Compute Fixed" means using 64 CPUs on one node for the *CPU-EA* algorithm and using 80 CPUs together with 8 GPUs for the *GPU-EA* algorithm. The results from this benchmark experiment are shown in **Figure 3A** and **Table 1**. Across all population sizes, *CPU-EA* took 10x the amount of time it took *NeuroGPU-EA* to complete a simulation-evaluation step and 7x the amount of time it took *CoreNeuron-EA* to complete a simulation-evaluation step. The comparative performance of *CoreNeuron-EA* and *CPU-EA* aligns with previous benchmarking studies showing CoreNeuron's 2-7x decrease of NEURON execution time (Kumbhar et al., 2019). Between *GPU-EA* experiments in **Figure 3A**, *NeuroGPU-EA* had approximately 20% speed-up when compared against *CoreNeuron-EA*. **Supplementary Figure 3A** shows that both feature extraction libraries had similar scaling performance, with Allen IPFX extractor running slightly faster in general, exhibiting a speed up of about 10%. **Supplementary Figure 4A** shows this experimental design applied to *NeuroGPU-EA* using Oak Ridge National Lab's (ORNL) Summit[14] computing cluster. Experiments ran on Cori showed a speed-up of around 20% when compared to those ran on Summit. These experiments characterize the rate in which run time of simulation-evaluation loop grows as the population size scales up. There are several possible scaling bounds such as logarithmic $O(log(N))$, linear $O(N)$, polynomial $O(N^k)$, and exponential $O(k^N)$ where k is a constant and N is the population size. Our expectation is that the run time of the algorithm should increase directly proportional to the increase of the population size. This would be a linear relationship or $O(N)$. **Figure 3A** and **Supplementary Figures 3A, 4A** all confirm a close alignment between expected scaling performance and actualized scaling performance. To further investigate the factors that drive an increase in run time in the application, additional experiments analyzing single node performance were required.

Further motivation for scaling population size on a single node is that this analysis can identify bottlenecks that occur at different problem sizes. **Figure 4** shows a set of experiments ranging from 187 to 3,000 neurons models per node using *GPU-EA* and *CoreNeuron-EA*. These experiments measured the run time for simulating (GPU) and evaluating (CPU). In this figure, both GPU computations and CPU computations are potential bottlenecks. Starting at around 375 individuals per node, up to twice as much time is spent running simulations on the GPU than evaluating them on the CPU. The proportion of run time on the CPU to run time on the GPU increases with the amount of population per node. At 3,000 individuals per node, the CPU evaluation takes twice as long as the GPU evaluation time. For both *CoreNeuron-EA* and *NeuroGPU-EA*, when the population size is larger than 1,000, the CPU is the bottleneck. Thus, predicting the simulate-evaluate run time as population per node increases becomes increasingly dependent on CPU run time.

[13] https://github.com/BlueBrain/CoreNeuron/issues/642

[14] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

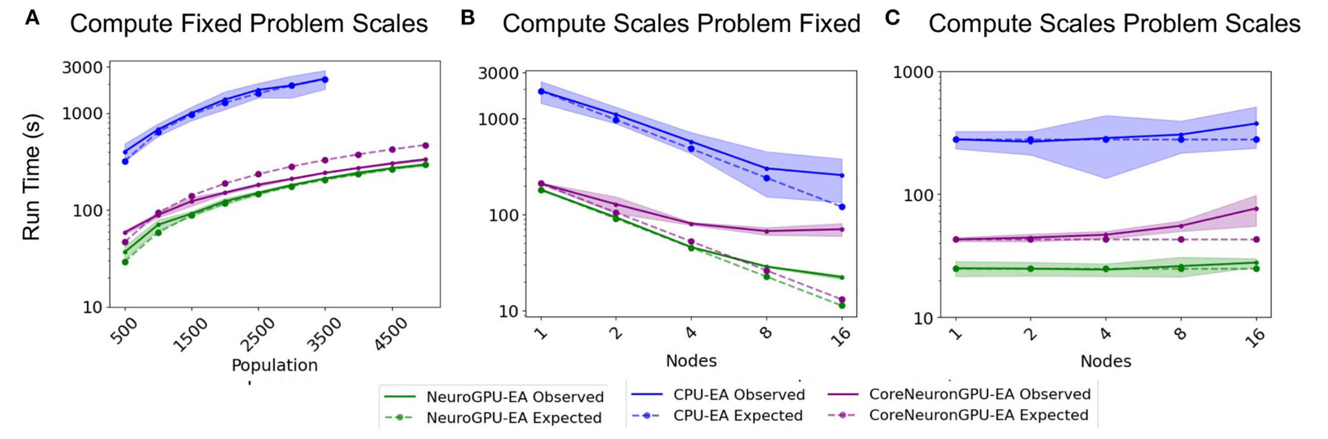


FIGURE 3 | Simulation-evaluation scaling CPU vs. GPU: Experiments measuring the time it takes to run one simulation-evaluation step using *NeuroGPU-EA*, *CoreNeuron-EA*, and *CPU-EA*. **(A)** One compute node and population size increases, as in **Table 1**. **(B)** Increases compute nodes and population size is constant, as in **Table 2**. **(C)** Increasing compute nodes and population size, as in **Table 3**.

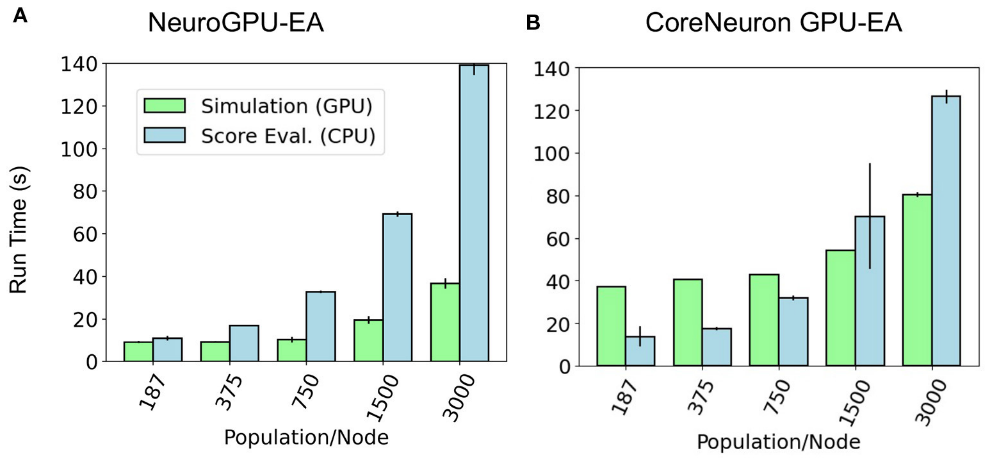


FIGURE 4 | GPU bottleneck shifts to CPU as population per node increases: at large population sizes the CPU operation for score functions is the bottleneck—denoted by relatively taller bars for CPU Eval. At smaller population sizes in **(B)** the GPU simulation is the bottleneck for *CoreNeuron-EA*—denoted by relatively taller bars for simulation. CPU and GPU time are balanced at small population sizes for *NeuroGPU-EA* in **(A)**.

3.3. Benchmark 2

The second benchmark, "Compute Scales Problem Fixed", determines the *strong scaling* of the application. Keeping the problem size constant and increasing the number of allocated CPU/GPU resources quantifies the potential for parallelism to accelerate the simulation-evaluation step. In this experimental design, specified by **Table 2**, the population size, N, is fixed at 3,000, while the number of allocated nodes scales up by a factor of 2. The outcomes for scaling N exponentially are represented in **Figure 3B**. The expectation is that run time decreases exponentially by a factor of 2, corresponding to the compute scaling rate. For all GPU-based algorithms, after 4 nodes, or 2 nodes in the case of *CoreNeuron-EA*, run time acceleration per node starts to decrease and no longer match expected scaling. This demonstrates a limit to which parallelism in *GPU-EA* can efficiently leverage available resources. As shown in **Figures 4A,B** and **Table 2**, at 187 individuals per node the time to complete evaluation is around 14 and 16 s for 375 individuals. This demonstrates Amdahl's law (Amdahl, 1967) which states that the overall improvement gained by parallelized code is limited to the fraction of time that code is in use. The benefit of parallelizing across the population and electrophysiological score functions is limited by the time the slowest score function takes to complete. Similarly, for simulation, both *GPU-EA* and *CPU-EA* show marginal decrease in run time after population size begins decreasing below 750 individuals per node. These results support the analysis shown in **Figure 3B**, as the benefit of using more than 4 nodes to simulate and evaluate 3,000 neurons is limited by the speed of the software modules deployed in the respective tasks. The equivalent

TABLE 2 | Compute scales and problem fixed: Stimuli and electrophysiological score functions are fixed to 8 and 20, respectively.

	CPU node	GPU node		Run time (s)		
Nodes	**Total CPUs**	**Total CPUs**	**Total GPUs**	**CPU-EA**	**NeuroGPU-EA**	**CoreNeuronGPU-EA**
1	64	80	8	1930 ± 490	182 ± 3.98	210 ± 3.58
2	128	160	16	1100 ± 212	93.6 ± 1.52	128 ± 24.9
4	256	320	32	573 ± 141	45.9 ± 0.586	80.6 ± 2.64
8	512	640	64	302 ± 149	28.9 ± 0.526	67.4 ± 5.96
16	1,024	1280	128	257 ± 123	22.4 ± 1.05	70.4 ± 10.5

Population size is fixed to 3,000. each.

TABLE 3 | Compute scales and problem scales: see **Table 2** for other details.

	CPU node	GPU node		Run time (s)			
Nodes	**Total CPUs**	**Total CPUs**	**Total GPUs**	**Population**	**CPU-EA**	**NeuroGPU-EA**	**CoreNeuronGPU-EA**
1	64	80	8	250	279 ± 44.9	25.0 ± 3.49	42.9 ± 1.53
2	128	160	16	500	267 ± 58	24.8 ± 3.13	44.3 ± 3.08
4	256	320	32	1,000	285 ± 151	24.4 ± 2.85	46.8 ± 3.24
8	512	640	64	2,000	305 ± 88.4	26.0 ± 4.75	55.3 ± 5.29
16	1,024	1,280	128	4,000	374 ± 137	27.7 ± 2.24	76.5 ± 21.4

experiments, shown in **Supplementary Figures 3B, 4B**, using IPFX electrophysiological score functions and different computing architecture demonstrate the same limitations in using more than 4 nodes to simulate 3,000 neuron models. The next benchmark illustrates how scaling the problem size enables efficient utilization of larger resource allocations.

3.4. Benchmark 3

The third benchmark, "Compute Scales Problem Scales", determines the *weak scaling* of the application. In this experimental design, specified by **Table 3**, the initial trial sets a scaling factor 250 population ($N = 250$) per node. The subsequent trials increase the number of CPU/GPU nodes and population size proportionally. The expectation is that run time remains constant. These experiments demonstrate how multi-node parallelism can accommodate the scaling of population size in the evolutionary algorithm. As demonstrated in **Figure 3C**, scaling at 250 individuals per node allows the run time of algorithm to remain approximately constant for up to 10 nodes. We chose to scale at 250 individuals per node because in this allocation the time spent on the GPU and CPU are nearly balanced for *GPU-EA*. Furthermore, this choice of scaling factor resulted in a higher average GPU utilization, at around 70%, as demonstrated in **Supplementary Figure 6**. This figure demonstrates the proportion of time spent running computations on the GPU and CPU compared to the total run time. With a scaling constant of 250 individuals, at more than 10 nodes the run time starts to marginally increase with each trial. In *CPU-EA*, the increase in run time is marginal. **Supplementary Figure 3C** demonstrates that eFEL score functions and Allen IPFX provide both match the expected constant scaling and the performance is nearly identical. The IPFX library is a few seconds faster than eFEL. Further experiments, shown in **Supplementary Figures 3C, 4C**, demonstrate that overhead is incurred when *NeuroGPU-EA* is run on larger allocations of GPU nodes (64–128 Nodes) using the Summit computing cluster. In the Section 4, further consideration is taken toward the explaining implications of successful large-scale optimization runs and the software/hardware that powers such runs.

3.5. Scaling Stimuli and Electrophysiological Score Functions

The set of experiments above only changes the problem size using population size, N. To further explore the axes of scaling *GPU-EA* problem space, we ran scaling experiments on *GPU-EA* with *NeuroGPU-EA* where electrophysiological score functions are set to 20, population size is set to 500 but the number of stimuli used in EA increases from 1 to 18. This experiment is shown in **Figure 5A**. In this figure, we use big O notation to denote worst case scaling of running time. The $O(\frac{log(n)}{2})$ and $O(\frac{log(n)}{4})$ lines show the starting run time scaled by the log transform of the expected increase in run time. This figure shows that *GPU-EA* scales logarithmically with the number of stimuli used. Furthermore, we ran an experiment on *GPU-EA* where the number of stimuli is fixed to 8, population size is fixed to 500 but the number of electrophysiological score functions used in EA increases from 1 to 180. This experiment is shown in **Figure 5B**. In this figure, there is constant scaling for up to 80 score functions. Once the number of electrophysiological score functions exceeds 80 they can no longer run entirely in parallel and the algorithm begins to scale at a constant linear rate—$O(\frac{n}{3})$. These results in scaling different dimensions of the EA problem size further demonstrates how computational resources can be

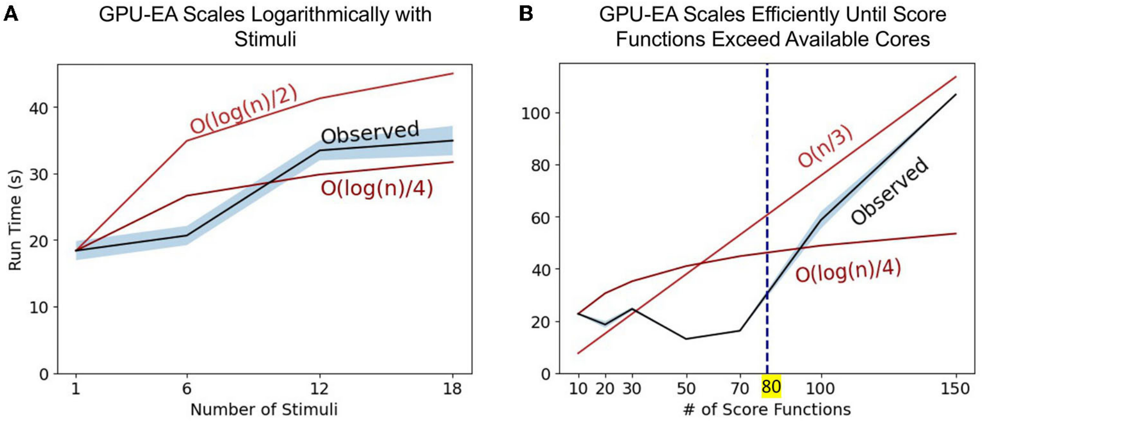


FIGURE 5 | Scaling stimuli and electrophysiological scoring functions: Panel **(A)** represents the observed run time as the number of stimuli used in the algorithm increases. We provide two lines for scaling reference $\mathcal{O}(\frac{log(n)}{2})$ and $\mathcal{O}(\frac{log(n)}{4})$. Panel **(B)** represents the observed run time as the number of score functions used in the algorithm increases. We provide two lines for reference, $\mathcal{O}(\frac{n}{3})$ and $\mathcal{O}(\frac{log(n)}{4})$.

leveraged using parallelism and concurrency to achieve efficient scaling in our *GPU-EA* algorithmic design.

3.6. Benchmark Model Fit

Figure 6 shows the model neuron, specified in Section 2, that was fit using GPU-GA. Congruent with benchmarks 1, 2, and 3, the EA used to fit this model was constrained to use 5,000 population size 20 score functions and 8 stimuli. Models are fitted against publicly available experimental data and stimuli from the Allen Cell Types Database (Gouwens et al., 2019) specimen 488683425. The experimentally recorded cell, plotted in black in **Figure 6**, is a layer 5 thick tufted pyramidal visual cortex cell. The best model, obtained according to the procedure in Section 2, is plotted in red. **Figure 6** shows that the fitted model neuron demonstrates a similar firing rate and spike onsets that are well-aligned with those of experimental data. While the simulated model waveforms closely align with experimental data, the voltage base and after hyperpolarization depth (AHP) vary from those produced by the experimental neuron. As shown in **Supplementary Figures 5A,E**, the voltage base is indicative of a limitation of the passive dynamics of the optimized model, such as g_pas and e_pas. These dynamics could be alleviated through the use of more appropriate passive score functions. The generalized response of the model to stimuli that were not used in the optimization is also demonstrated in **Supplementary Figures 5F–H**. These results show the quality of model that can be achieved with the simple EA design and stimuli used in the benchmarks, however there are many aspects of EA optimization that can be tuned to achieve an improved neuron model. This is why a general understanding of optimization quality from different EA configurations is important. For instance, the beneficial impact of scaling EA population size is exemplified in the next section.

3.7. Effect of Scaling Up EA Population

To demonstrate the practical impact of scaling the evolutionary algorithm, we set up an experiment on the layer 5 thick tufted pyramidal neuron from the Blue Brain Project (Ramaswamy et al., 2015) described in Section 2. The purpose of this experiment is to demonstrate the benefit of increasing population size on the resulting optimized model. In this experiment, we ran 150 generations of the evolutionary algorithm for three different trials with population sizes 1,000, 5,000, 10,000, respectively. Unlike the EA in Section 3.6 which utilized the 8 stimuli from benchmarks 1–3, this version of EA utilized 8 different

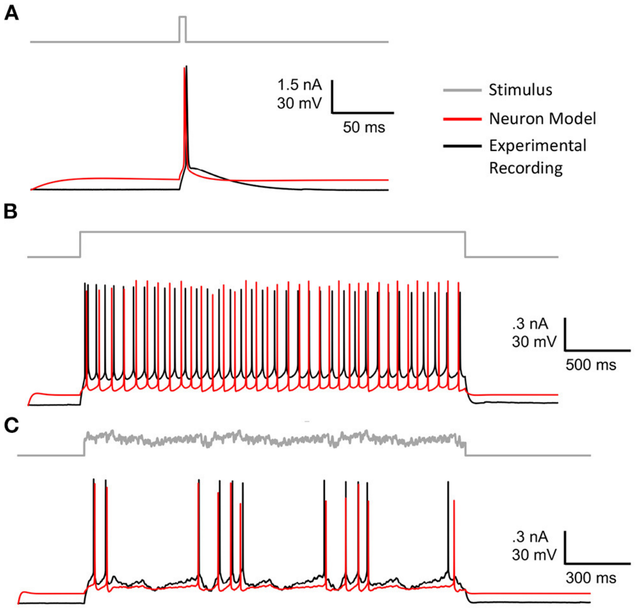


FIGURE 6 | Best fitted model after 50 generations of EA using 8 stimuli and 20 score functions (red) plotted against experimental data (black). **(A)** Long square stimulus. **(B)** Short square stimulus. **(C)** Noisy stimulus. The remaining stimuli and scores are shown in **Supplementary Figure 5** and **Supplementary Table 1**, respectively.

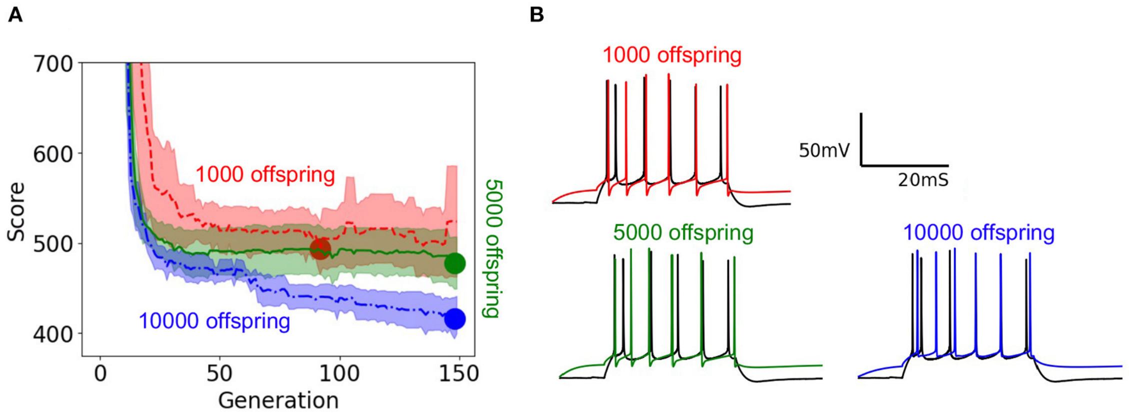


FIGURE 7 | EA score and model fit both improve with larger population size: **(A)** Objective function optimization trajectory in EA with varying population sizes. Scores start around 2,500 but the y-axis is constrained to clearly show results. Lower scores indicate a closer fit to experimental data. The minima of the objective function are denoted by large circles and the lower the minima the more the best simulated response resembles the experimentally recorded waveform. Confidence intervals are computed using 10 random initializations. Panel **(B)** illustrates the neuron model responses corresponding to varying population sizes.

long square currents, as shown in **Figure 1A**, and 8 score functions per stimulus. Six of the score functions are represented in **Figure 1C** and the other two are minor variations of the rendered score functions. For each population size, we ran 10 trials using 10 different random seeds for EA. As a Monte Carlo method, the trajectory of EA is stochastic, thus using random seeds ensures the score trajectories follow a reproducible trend. Notably, the run time per generation increased as EA progressed toward more optimal parameter sets. By breaking down the cumulative run time associated with running multiple generations of EA, **Supplementary Figure 8** shows that as EA progresses, the electrophysiological features take more time to compute on average. Regarding the optimization procedure, **Figure 7A** demonstrates the mean and 95% confidence interval of the score of the objective function for the evolutionary algorithm at each generation. The 10,000 individual EA achieves a better fit to experimental data, resulting in the lowest achieved value for the objective function. The value for objective function represents a penalty against simulated neurons where the electrophysiological features of voltage traces differ from those of an experimentally recorded target waveform. The lower scores achieved by the 10,000 individual EA indicate this configuration finds comparatively more optimal models for generations 70–150. Compared to 1,000 individual EA, the 5,000 individual EA achieved a lower mean score over 10 random seeds, but this difference was not statistically significant. Furthermore, alignment of the experimentally recorded neuron membrane potential and that of the best simulated neuron model substantiates the impact of improved optimization of the objective function due to larger population size in the EA. In **Figure 7B**, as the population sizes increase, models show improvement in the depth and timing of the after hyperpolarization (AHP). The AHP depth is the maximum level of depolarization after the action potential has peaked and re-polarized to resting potential. In the 10,000 individual optimization, the AHP depth is not greater than that of the experimentally recorded target waveform. The duration of the hyperpolarization is also more similar to the target waveform for the 10,000 individual optimization than the smaller population size EAs. **Figure 7B** qualitatively demonstrates that population sizes that allow EA to explore more potential parameter sets construct a model that better characterizes the experimental data (Ben-Shalom et al., 2012). **Figure 7A** quantitatively supports this claim by showing that EA with 10,000 individuals finds the most optimal solution when compared with smaller population EAs.

4. DISCUSSION

The most central comparison drawn in this paper is between CPU and GPU based simulation-evaluation loops. GPU based simulation is markedly faster and scales better than CPU based simulation. These results suggest that *CPU-EA* may be a reasonable choice for fitting simple electrophysiological neuron models, but that researchers should use caution in more computationally complex optimization problems that require scaling. For these complex problems, leveraging parallel code design and GPU neuron simulation can reduce EA optimization time from weeks or days to hours. Based off the CPU experiments in the Compute Fixed Problem Scales section, using a single desktop computer without a GPU would limit researchers to a population size of 1,000 or smaller. Based off the single node GPU example, the addition of a single GPU allows for a researcher to complete 50 iterations of a population size of 3,000 in several days. A conservative estimate is that using *GPU-EA* with a workstation that has 8 GPUs and over 40 cores enables a researcher to complete 50 iterations of an EA with 3,000 individuals in a day. Using the maximum amount of resources available, 128 nodes on Summit, we show in **Supplementary Figure 4C** that we can simulate and evaluate a population size of 32,000 in 35 s. This allocation makes it feasible to reach 50 generations of 32,000 individuals within the course of a few hours. While these estimates demonstrate the potential scale and advantages of *GPU-EA*, there are instances where *CPU-EA* may be more optimal. In Ben-Shalom et al. (2022), we show that simulating models with less compartments and channels are not substantially accelerated by GPUs. Furthermore, for EA optimizations with only one stimulus and a nominal number of score functions will not benefit from the score function level parallelism and stimuli level parallelism discussed in the Section 2. Finally, for researchers attempting to simulate many different models, NEURON provides the highest level of compatibility with most available models in ModelDB (Hines et al., 2004), though CoreNeuron is expanding its compatibility with NEURON.

Our choice of an appropriate scaling factor was critical in achieving large scale simulation and evaluation. We used the experiment shown in **Figure 4A** in the *Compute Fixed Problem Scales* section to determine a reasonable scaling constant of 250 neuron models per node. We chose this constant as the simulation and evaluation time were approximately balanced so neither simulation or evaluation would dominate run time. Relative to other configurations, 250 models per node led to the most efficient GPU utilization, at around 60%. 60% is the highest achieved GPU utilization using *GPU-EA* because the GPU must be idle while the evaluation step is finishing. Once all the models are scored and a new population evolves, the GPU resumes activity at 100% utilization. This is shown in the plot of GPU utilization over time in **Supplementary Figure 7**. Future work could involve implementations that achieve higher GPU utilization through different implementations, such as parallel EAs (PEAs) that evolve multiple sub-populations simultaneously (Cantú-Paz, 2001; Du et al., 2013). It may be necessary to modify GPU simulation modules in order to adapt *GPU-EA* to enable PEAs, or simultaneous EAs with different seeds. Based on **Figure 4A**, 250 neuron models per node was a conservative choice of a scaling constant, as the CPU did not start to become a bottleneck until 750 population per node and 1,500 population per node in **Figure 4B**. This conservative choice ensured we would be able to efficiently scale the problem size with number of computing nodes, which we aim to demonstrate for the purpose of benchmarking. In practice, researchers might choose larger scaling constants.

While *GPU-EA*'s ability to leverage parallelized kernel computation for fast simulation is one advantage. Another advantage is that *GPU-EA*, using NeuroGPU, also adds concurrency to the algorithm described in Section 2. Concurrency, defined as the capacity to run separate tasks at the same time (Roscoe, 1998), is different than the achieved levels of multi-node and single node parallelism. This algorithm is concurrent when simulation of the remaining batches of stimuli begin as soon as the first set of stimuli finish. The GPU does not remain idle as the CPU finishes evaluating the first batch of simulations. Thus, while the CPU is evaluating the quality of the simulations, the GPU begins the next batch of simulations. This is shown in **Figure 2C** as the CPU and GPU are running at the same time. The result of concurrency in *GPU-EA* is shown in **Figure 5A** where the algorithm scales logarithmically with the number of stimuli used in the algorithm. Logarithmic scaling is enabled by NeuroGPU's capacity to run stimuli in parallel across GPUs as well the algorithmic design to simulate a second set of neuronal models while the previous set of stimuli is being evaluated. This logarithmic scaling enables the objective function of evolutionary algorithm to incorporate multiple stimuli. Consequently, models that are fit using multiple stimuli will generalize better to new unseen stimuli. While state of the art fitting procedures like (Gouwens et al., 2018) are currently designed to use a single stimulus in the optimization algorithm, the addition of simulate-evaluate concurrency can enhance these methods using more stimuli with minimal cost in run time. A challenge with incorporating more stimuli in the objective function is that simulators that don't permit concurrent execution will need to simulate and evaluate sequentially.

In the section *Compute Fixed Problem Scales* and **Figure 4**, we showed that at too large of a population size, the score functions will bottleneck simulation-evaluation run time. We also found that this bottleneck in the evaluation step can be mitigated or worsened by the number of electrophysiological scoring functions used. **Figure 5B** showed constant scaling for up to 80 score functions. This happened because there were 80 cores available on a single Cori node. Once the number of score functions exceeds the number of cores available to a node they can no longer be run entirely in parallel and the algorithm begins to scale at a constant linear rate—O(n/3). Thus, if researchers intend to use multiple score functions for multi-objective optimization, as in Druckmann et al. (2007), we recommend they consider using fewer score functions than cores available in *GPU-EA*. Even in the case where there are fewer score functions than cores, **Supplementary Figure 8** shows that as EA progresses, the evaluation step takes more time to complete. A potential cause for increased evaluation time is that in later generations there are more spiking neuron models to be evaluated and eFEL score functions take longer on traces with more spikes. These results demonstrate a distinct advantage in simulating larger populations of neurons on GPU nodes as there are many opportunities to implement parallelism and concurrency. However, CPU processing capacity for scoring electrophysiological features hinders the efficient scaling of the *GPU-EA* algorithm. A potential strategy to alleviate this bottleneck could involve loading simulated traces into the score function library before mapping the score functions to be evaluated in parallel. Currently, traces are loaded separately for each score function. Another potential mitigation could be using GPU feature extraction. To the best of the author's knowledge, there are no available GPU based software toolkits for scoring features of simulated spike trains based on a target train. There are several GPU-based applications that are used for real time analysis of Electroencephalography (EEG) waveform data, Magnetoencephalography (MEG) data, and Multi-electrode Arrays (MEA) signals (Tadel et al., 2011; Guzman et al., 2014; Sahoo et al., 2016), but none that exist for evaluating simulated neuron firing traces. Software capable of scoring electrophysiological traces on a GPU would considerably enhance the performance of *GPU-EA* configurations where score functions are the bottleneck. The prospective advantage of GPU accelerated electrophysiological feature extraction presents an opportunity for researchers. Because the Blue Brain Project's eFEL (Van Geit et al., 2016) score function library is developed in C++, it has the potential to be adapted to the GPU through tools like OpenACC's GPU directives.

A critical consideration in attempting to generalize benchmarks, whether between simulation software, HPC platforms, or algorithms, is that factors from the hardware and software environment to the number of spiking neurons in a population can have a substantial impact on the run time of the application. In **Supplementary Figure 8**, the time to evaluate score functions increased as the EA algorithm produced more spiking neurons. The stochasticity in the optimization in EA is not a desirable property for benchmarking as it can be difficult to tell if scoring is taking longer to complete because it is slower or because an instantiation of EA is producing offspring that spike more. We mitigate this issue by benchmarking one initial population in 3.2, 3.3, and 3.4. Another example of variability in performance occurred in our comparison between Cori and Summit. Initial experiments demonstrated a much more dramatic speedup, but after upgrading to use GCC 8.3.0 (version used on CORI), the performance on Summit improved considerably. Kumbhar et al. (2019) shows a notable increase in performance of CoreNeuron using the Intel C/C++ compiler instead of GCC/G++. Moving from benchmarking stand-alone software modules to applications means there are more dependencies that can be affected by the installation environment. With this consideration we provide a simplified code example[15] to run one simulation evaluation loop without HPC or EA. We also provide the entire code suite[16], which we hope to further extend to be a platform capable of benchmarking of more tools in computational neuroscience.

In future work, we aim to apply this benchmarking framework across several other software modules of interest. A simple extension of this work would be to run experiments comparing Allen IPFX and BluePyopt's eFEL to the widely adopted python electrophysiological toolkit Elephant (Denker et al., 2018). While Elephant has fewer statistic-based features, it offers correlative measures between spike trains. Also, Elephant has a parallel extension which can further advantage HPC resources. Another simple extension of this work could involve benchmarking the biophysical neuron simulator LFPy (Lindén et al., 2014) or Arbor (Abi Akar et al., 2019). A more complex extension of this work would involve benchmarking the same simulate-evaluate loop, but as it applies to spiking neural networks (SNN) instead of the evolutionary algorithm. There are several well-documented and widely adopted SNN packages such as Brian (Goodman and Brette, 2009), NEST (Gewaltig and Diesmann, 2007), and SpiNNaker (Furber et al., 2014) that would be appropriate to benchmark using this experimental design. Finally, we are interested in generalizing this benchmarking experimental design to a wider range of single neuron and network optimization tasks. Any algorithm that involves simulation or feed-forward stage and then an evaluation/feedback/learning stage is amenable to the analysis conducted in this paper. This generalizability extends to many methods commonly used in machine learning and optimization.

5. CONCLUSION

This work demonstrates the potential of efficiently parallelized simulation and evaluation software for electrophysiological modeling. Specifically, applications that leverage GPU utilization demonstrate the capacity to run larger fitting optimizations. In turn, these optimizations can result in a larger search of the parameter space, and consequently, a more accurate model. As the processor count continues to increase on hyper-threaded and multi-core chips, computational methods that leverage parallelism can continue to leverage new innovations in high performance computing to generate more detailed and accurate neuronal models. While this progression is beneficial, it is ever relevant to apply established benchmarks such as weak scaling and strong scaling for neuroscientists to get the most value out of new computing resources.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found below: https://portal.nersc.gov/cfs/m2043/benchmarking_ea.tar.gz.

AUTHOR CONTRIBUTIONS

JB, KB, and RB-S helped the conceptualize experiments. AL and KK designed the software to run, process, and visualize experiments. AL wrote the original draft. RB-S helped with visualization. RB-S and KB funded the project and provided supercomputing hours. All authors have read and agreed to the published version of the manuscript.

FUNDING

This research was supported by the MIND Institute and Neurology Department at the University of California Davis and the Action Potential grant from The FamiliesSCN2A Foundation.

ACKNOWLEDGMENTS

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award M2043. Thank you to Albert Vasquez for helping me test out reproducible examples locally on his 3090 GPU.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2022.882552/full#supplementary-material

[15] https://github.com/xanderladd/benchmarking_examples
[16] https://github.com/xanderladd/EA_benchmarking

REFERENCES

Abi Akar, N., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor–a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia: IEEE), 274–282. doi: 10.1109/EMPDP.2019.8671560

Almog, M., and Korngreen, A. (2016). Is realistic neuronal modeling realistic? *J. Neurophysiol*. 116, 2180–2209. doi: 10.1152/jn.00360.2016

Amdahl, G. M. (1967). "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of Spring Joint Computer Conference* (New Jersey), 483–485. doi: 10.1145/1465482.1465560

Bailey, D. H., Lucas, R. F., and Williams, S. (2010). *Performance Tuning of Scientific Applications*. Florida: CRC Press. doi: 10.1201/b10509

Balasubramanian, M., Ruiz, T. D., Cook, B., Prabhat, M., Bhattacharyya, S., Shrivastava, A., et al. (2020). "Scaling of union of intersections for inference of granger causal networks from observational data," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Louisiana: IEEE), 264–273. doi: 10.1109/IPDPS47924.2020.00036

Ben-Shalom, R., Athreya, N. S., Cross, C., Sanghevi, H., Kim, K. G., Ladd, A., et al. (2020). NeuroGPU, software for NEURON modeling in GPU-based hardware. *bioRxiv* 366, 727560. doi: 10.1101/727560

Ben-Shalom, R., Aviv, A., Razon, B., and Korngreen, A. (2012). Optimizing ion channel models using a parallel genetic algorithm on graphical processors. *J. Neurosci. Methods* 206, 183–194. doi: 10.1016/j.jneumeth.2012.02.024

Ben-Shalom, R., Keeshen, C. M., Berrios, K. N., An, J. Y., Sanders, S. J., and Bender, K. J. (2017). Opposing effects on Na v1. 2 function underlie differences between SCN2A variants observed in individuals with autism spectrum disorder or infantile seizures. *Biol. Psychiatry* 82, 224–232. doi: 10.1016/j.biopsych.2017.01.009

Ben-Shalom, R., Ladd, A., Artherya, N. S., Cross, C., Kim, K. G., Sanghevi, H., et al. (2022). NeuroGPU: accelerating multi-compartment, biophysically detailed neuron simulations on GPUs. *J. Neurosci. Methods* 366, 109400. doi: 10.1016/j.jneumeth.2021.109400

Ben-Shalom, R., Liberman, G., and Korngreen, A. (2013). Accelerating compartmental modeling on a graphical processing unit. *Front. Neuroinform*. 7, 4. doi: 10.3389/fninf.2013.00004

Beyer, H. (2007). Evolution strategies. *Scholarpedia* 2, 1965. doi: 10.4249/scholarpedia.1965

Beyer, H.-G., and Schwefel, H.-P. (2002). Evolution strategies-a comprehensive introduction. *Natural Comput*. 1, 3–52. doi: 10.1023/A:1015059928466

Billeh, Y. N., Cai, B., Gratiy, S. L., Dai, K., Iyer, R., Gouwens, N. W., et al. (2020). Systematic integration of structural and functional data into multi-scale models of mouse primary visual cortex. *Neuron* 106, 388–403. doi: 10.1016/j.neuron.2020.01.040

Bouchard, K. E., Aimone, J. B., Chun, M., Dean, T., Denker, M., Diesmann, M., et al. (2016). High-performance computing in neuroscience for data-driven discovery, integration, and dissemination. *Neuron* 92, 628–631. doi: 10.1016/j.neuron.2016.10.035

Bouchard, K. E., Aimone, J. B., Chun, M., Dean, T., Denker, M., Diesmann, M., et al. (2018). International neuroscience initiatives through the lens of high-performance computing. *Computer* 51, 50–59. doi: 10.1109/MC.2018.2141039

Bower, J. M., and Beeman, D. (2012). *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. California: Springer Science & Business Media.

Cantú-Paz, E. (2001). Migration policies, selection pressure, and parallel evolutionary algorithms. *J. Heurist*. 7, 311–334. doi: 10.1023/A:1011375326814

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Connecticut: Cambridge University Press. doi: 10.1017/CBO9780511541612

Coleman, C., Kang, D., Narayanan, D., Nardi, L., Zhao, T., Zhang, J., et al. (2019). Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev*. 53, 14–25. doi: 10.1145/3352020.3352024

Criado, J., Garcia-Gasulla, M., Kumbhar, P., Awile, O., Magkanaris, I., and Mantovani, F. (2020). "CoreNEURON: performance and energy efficiency evaluation on intel and arm CPUs," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)* (Kobe), 540–548. doi: 10.1109/CLUSTER49012.2020.00077

Daou, A., and Margoliash, D. (2020). Intrinsic neuronal properties represent song and error in zebra finch vocal learning. *Nat. Commun*. 11, 1–17. doi: 10.1038/s41467-020-14738-7

De Schutter, E., and Bower, J. M. (1994). An active membrane model of the cerebellar purkinje cell. I. Simulation of current clamps in slice. *J. Neurophysiol*. 71, 375–400. doi: 10.1152/jn.1994.71.1.375

Denker, M., Yegenoglu, A., and Grun, S. (2018). "Collaborative HPC-enabled workflows on the HBP Collaboratory using the Elephant framework," in *Neuroinformatics 2018* (Montreal, QC), p. 19.

Druckmann, S., Banitt, Y., Gidon, A. A., Schürmann, F., Markram, H., and Segev, I. (2007). A novel multiple objective optimization framework for constraining conductance-based neuron models by experimental data. *Front. Neurosci*. 1, 7–18. doi: 10.3389/neuro.01.1.1.001.2007

Druckmann, S., Berger, T. K., Schürmann, F., Hill, S., Markram, H., and Segev, I. (2011). Effective stimuli for constructing reliable neuron models. *PLoS Comput. Biol*. 7, e1002133. doi: 10.1371/journal.pcbi.1002133

Du, X., Ni, Y., Yao, Z., Xiao, R., and Xie, D. (2013). High performance parallel evolutionary algorithm model based on mapreduce framework. *Int. J. Comput. Appl. Technol*. 46, 290–295. doi: 10.1504/IJCAT.2013.052807

Eliasmith, C., and Trujillo, O. (2014). The use and abuse of large-scale brain models. *Curr. Opin. Neurobiol*. 25, 1–6. doi: 10.1016/j.conb.2013.09.009

Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. (2004). "GPU cluster for high performance computing," in *SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (Pennsylvania), 47.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., and Gagné, C. (2012). DEAP: evolutionary algorithms made easy. *J. Mach. Learn. Res*. 13, 2171–2175. Available online at: https://www.jmlr.org/papers/v13/fortin12a.html

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The Spinnaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (neural simulation tool). *Scholarpedia* 2, 1430. doi: 10.4249/scholarpedia.1430

Goldman, M. S., Golowasch, J., Marder, E., and Abbott, L. (2001). Global structure, robustness, and modulation of neuronal models. *J. Neurosci*. 21, 5229–5238. doi: 10.1523/JNEUROSCI.21-14-05229.2001

Golowasch, J., Goldman, M. S., Abbott, L., and Marder, E. (2002). Failure of averaging in the construction of a conductance-based neuron model. *J. Neurophysiol*. 87, 1129–1131. doi: 10.1152/jn.00412.2001

Goodman, D. F., and Brette, R. (2009). The brian simulator. *Front. Neurosci*. 3, 192–197. doi: 10.3389/neuro.01.026.2009

Gouwens, N. W., Berg, J., Feng, D., Sorensen, S. A., Zeng, H., Hawrylycz, M. J., et al. (2018). Systematic generation of biophysically detailed models for diverse cortical neuron types. *Nat. Commun*. 9, 1–13. doi: 10.1038/s41467-017-02718-3

Gouwens, N. W., Sorensen, S. A., Berg, J., Lee, C., Jarsky, T., Ting, J., et al. (2019). Classification of electrophysiological and morphological neuron types in the mouse visual cortex. *Nat. Neurosci*. 22, 1182–1195. doi: 10.1038/s41593-019-0417-0

Guzman, S., Schlogl, A., and Schmidt-Hieber, C. (2014). Stimfit: quantifying electrophysiological data with python. *Front. Neuroinform*. 8, 16. doi: 10.3389/fninf.2014.00016

Herlihy, M., and Shavit, N. (2012). *The Art of Multiprocessor Programming, 1st Edn*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Hines, M. (1984). Efficient computation of branched nerve equations. *Int. J. Biomed. Comput*. 15, 69–76. doi: 10.1016/0020-7101(84)90008-4

Hines, M. L., Morse, T., Migliore, M., Carnevale, N. T., and Shepherd, G. M. (2004). ModelDB: a database to support computational neuroscience. *J. Comput. Neurosci*. 17, 7–11. doi: 10.1023/B:JCNS.0000023869.22017.2e

Hodgkin, A. L., and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol*. 117, 500–544. doi: 10.1113/jphysiol.1952.sp004764

Hoefler, T., and Belli, R. (2015). "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Texas), 1–12. doi: 10.1145/2807591.2807644

Keren, N., Peled, N., and Korngreen, A. (2005). Constraining compartmental models using multiple voltage recordings and genetic algorithms. *J. Neurophysiol*. 94, 3730–3742. doi: 10.1152/jn.00408.2005

Knight, J. C., and Nowotny, T. (2018). Gpus outperform current hpc and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Front. Neurosci.* 12, 941. doi: 10.3389/fnins.2018.00941

Kulkarni, S. R., Parsa, M., Mitchell, J. P., and Schuman, C. D. (2021). Benchmarking the performance of neuromorphic and spiking neural network simulators. *Neurocomputing* 447, 145–160. doi: 10.1016/j.neucom.2021.03.028

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., et al. (2019). Coreneuron: an optimized compute engine for the neuron simulator. *Front. Neuroinform.* 13, 63. doi: 10.3389/fninf.2019.00063

Lindén, H., Hagen, E., Leski, S., Norheim, E. S., Pettersen, K. H., and Einevoll, G. T. (2014). Lfpy: a tool for biophysical simulation of extracellular potentials generated by detailed model neurons. *Front. Neuroinform.* 7, 41. doi: 10.3389/fninf.2013.00041

Mainen, Z. F., Joerges, J., Huguenard, J. R., and Sejnowski, T. J. (1995). A model of spike initiation in neocortical pyramidal neurons. *Neuron* 15, 1427–1439. doi: 10.1016/0896-6273(95)90020-9

Mainen, Z. F., and Sejnowski, T. J. (1996). Influence of dendritic structure on firing pattern in model neocortical neurons. *Nature* 382, 363–366. doi: 10.1038/382363a0

Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

Masoli, S., Rizza, M. F., Sgritta, M., Van Geit, W., Schürmann, F., and D'Angelo, E. (2017). Single neuron optimization as a basis for accurate biophysical modeling: the case of cerebellar granule cells. *Front. Cell. Neurosci.* 11, 71. doi: 10.3389/fncel.2017.00071

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. Massachusetts: MIT Press. doi: 10.7551/mitpress/3927.001.0001

Nogaret, A., Meliza, C. D., Margoliash, D., and Abarbanel, H. D. (2016). Automatic construction of predictive neuron models through large scale assimilation of electrophysiological data. *Sci. Rep.* 6, 1–14. doi: 10.1038/srep32749

Prinz, A. A., Billimoria, C. P., and Marder, E. (2003). Alternative to hand-tuning conductance-based models: construction and analysis of databases of model neurons. *J. Neurophysiol.* 90, 3998–4015. doi: 10.1152/jn.00641.2003

Prinz, A. A., Bucher, D., and Marder, E. (2004). Similar network activity from disparate circuit parameters. *Nat. Neurosci.* 7, 1345–1352. doi: 10.1038/nn1352

Rall, W. (1959). Branching dendritic trees and motoneuron membrane resistivity. *Exp. Neurol.* 1, 491–527. doi: 10.1016/0014-4886(59)90046-9

Rall, W. (1962). Electrophysiology of a dendritic neuron model. *Biophys. J.* 2(2 Pt 2), 145. doi: 10.1016/S0006-3495(62)86953-7

Rall, W. (1964). "Theoretical significance of dendritic trees for neuronal input-output relations," in *Neural Theory and Modeling*, eds I. Segev, J. Rinzel, and G. M. Shephard (Cambridge: MIT Press), 73–97.

Rall, W. (2009). Rall model. *Scholarpedia* 4, 1369. doi: 10.4249/scholarpedia.1369

Ramaswamy, S., Courcol, J.-D., Abdellah, M., Adaszewski, S. R., Antille, N., Arsever, S., et al. (2015). The neocortical microcircuit collaboration portal: a resource for rat somatosensory cortex. *Front. Neural Circ.* 9, 44. doi: 10.3389/fncir.2015.00044

Roscoe, B. (1998). *The Theory and Practice of Concurrency*. New Jersey: Prentice-Hall (Pearson).

Sahoo, S. S., Wei, A., Valdez, J., Wang, L., Zonjy, B., Tatsuoka, C., et al. (2016). NeuroPigPen: a scalable toolkit for processing electrophysiological signal data in neuroscience applications using apache pig. *Front. Neuroinform.* 10, 18. doi: 10.3389/fninf.2016.00018

Sakmann, B., and Neher, E. (1984). Patch clamp techniques for studying ionic channels in excitable membranes. *Annu. Rev. Physiol.* 46, 455–472. doi: 10.1146/annurev.ph.46.030184.002323

Sáray, S., Rössert, C. A., Appukuttan, S., Migliore, R., Vitale, P., Lupascu, C. A., et al. (2020). Systematic comparison and automated validation of detailed models of hippocampal neurons. *bioRxiv [Preprint]*. doi: 10.1101/2020.07.02.184333

Schaller, R. R. (1997). Moore's law: past, present and future. *IEEE Spectrum* 34, 52–59. doi: 10.1109/6.591665

Spratt, P. W., Alexander, R. P., Ben-Shalom, R., Sahagun, A., Kyoung, H., Keeshen, C. M., et al. (2021). Paradoxical hyperexcitability from Na v1. 2 sodium channel loss in neocortical pyramidal cells. *Cell Rep.* 36, 109483. doi: 10.1016/j.celrep.2021.109483

Strohmaier, E., Meuer, H. W., Dongarra, J., and Simon, H. D. (2015). The top500 list and progress in high-performance computing. *Computer* 48, 42–49. doi: 10.1109/MC.2015.338

Tadel, F., Baillet, S., Mosher, J. C., Pantazis, D., and Leahy, R. M. (2011). Brainstorm: a user-friendly application for MEG/EEG analysis. *Comput. Intell. Neurosci.* 2011, 879716. doi: 10.1155/2011/879716

Traub, R. D., Contreras, D., Cunningham, M. O., Murray, H., LeBeau, F. E., Roopun, A., et al. (2005). Single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles, and epileptogenic bursts. *J. Neurophysiol.* 93, 2194–2232. doi: 10.1152/jn.00983.2004

Traub, R. D., Wong, R. K., Miles, R., and Michelson, H. (1991). A model of a ca3 hippocampal pyramidal neuron incorporating voltage-clamp data on intrinsic conductances. *J. Neurophysiol.* 66, 635–650. doi: 10.1152/jn.1991.66.2.635

Van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware spinnaker and the neural network simulation software nest for a full-scale cortical microcircuit model. *Front. Neurosci.* 12, 291. doi: 10.3389/fnins.2018.00291

Van Geit, W., Achard, P., and De Schutter, E. (2007). Neurofitter: a parameter tuning package for a wide range of electrophysiological neuron models. *Front. Neuroinform.* 1, 1. doi: 10.3389/neuro.11.001.2007

Van Geit, W., De Schutter, E., and Achard, P. (2008). Automated neuron model optimization techniques: a review. *Biol. Cybernet.* 99, 241–251. doi: 10.1007/s00422-008-0257-6

Van Geit, W., Gevaert, M., Chindemi, G., Rossert, C., Courcol, J.-D., Muller, E. B., et al. (2016). BluePyOpt: leveraging open source software and cloud infrastructure to optimise model parameters in neuroscience. *Front. Neuroinform.* 10, 17. doi: 10.3389/fninf.2016.00017

Vanier, M. C., and Bower, J. M. (1999). A comparative survey of automated parameter-search methods for compartmental neural models. *J. Comput. Neurosci.* 7, 149–171. doi: 10.1023/A:1008972005316

Wu, X., Taylor, V., Wozniak, J. M., Stevens, R., Brettin, T., and Xia, F. (2019). "Performance, energy, and scalability analysis and improvement of parallel cancer deep learning candle benchmarks," in *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto), 1–11. doi: 10.1145/3337821.3337905

Zitzler, E., and Künzli, S. (2004). "Indicator-based selection in multiobjective search," in *International Conference on Parallel Problem Solving from Nature* (Berlin; Heidelberg: Springer), 832–842. doi: 10.1007/978-3-540-30217-9_84

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

TECHNOLOGY AND CODE
published: 27 June 2022
doi: 10.3389/fninf.2022.884046

Modernizing the NEURON Simulator for Sustainability, Portability, and Performance

Omar Awile[1†], *Pramod Kumbhar*[1†], *Nicolas Cornu*[1], *Salvador Dura-Bernal*[2,3], *James Gonzalo King*[1], *Olli Lupton*[1], *Ioannis Magkanaris*[1], *Robert A. McDougal*[4,5,6], *Adam J. H. Newton*[2,4], *Fernando Pereira*[1], *Alexandru Săvulescu*[1], *Nicholas T. Carnevale*[7‡], *William W. Lytton*[3‡], *Michael L. Hines*[7‡] and *Felix Schürmann*[1*‡]

[1] Blue Brain Project, École Polytechnique Fédérale de Lausanne (EPFL), Geneva, Switzerland, [2] Department Physiology and Pharmacology, SUNY Downstate, Brooklyn, NY, United States, [3] Center for Biomedical Imaging and Neuromodulation, Nathan Kline Institute for Psychiatric Research, Orangeburg, NY, United States, [4] Department of Biostatistics, Yale School of Public Health, New Haven, CT, United States, [5] Program in Computational Biology and Bioinformatics, Yale University, New Haven, CT, United States, [6] Yale Center for Medical Informatics, Yale University, New Haven, CT, United States, [7] Department of Neuroscience, Yale University, New Haven, CT, United States

Edited by:
Mike Hawrylycz,
Allen Institute for Brain Science,
United States

Reviewed by:
Daniele Linaro,
Politecnico di Milano, Italy
Mikael Djurfeldt,
Royal Institute of Technology, Sweden

***Correspondence:**
Felix Schürmann
felix.schuermann@epfl.ch

† These authors share first authorship
‡ These authors share senior authorship

Received: 25 February 2022
Accepted: 26 May 2022
Published: 27 June 2022

Citation:
Awile O, Kumbhar P, Cornu N, Dura-Bernal S, King JG, Lupton O, Magkanaris I, McDougal RA, Newton AJH, Pereira F, Săvulescu A, Carnevale NT, Lytton WW, Hines ML and Schürmann F (2022) Modernizing the NEURON Simulator for Sustainability, Portability, and Performance.
Front. Neuroinform. 16:884046.
doi: 10.3389/fninf.2022.884046

The need for reproducible, credible, multiscale biological modeling has led to the development of standardized simulation platforms, such as the widely-used NEURON environment for computational neuroscience. Developing and maintaining NEURON over several decades has required attention to the competing needs of backwards compatibility, evolving computer architectures, the addition of new scales and physical processes, accessibility to new users, and efficiency and flexibility for specialists. In order to meet these challenges, we have now substantially modernized NEURON, providing continuous integration, an improved build system and release workflow, and better documentation. With the help of a new source-to-source compiler of the NMODL domain-specific language we have enhanced NEURON's ability to run efficiently, via the CoreNEURON simulation engine, on a variety of hardware platforms, including GPUs. Through the implementation of an optimized in-memory transfer mechanism this performance optimized backend is made easily accessible to users, providing training and model-development paths from laptop to workstation to supercomputer and cloud platform. Similarly, we have been able to accelerate NEURON's reaction-diffusion simulation performance through the use of just-in-time compilation. We show that these efforts have led to a growing developer base, a simpler and more robust software distribution, a wider range of supported computer architectures, a better integration of NEURON with other scientific workflows, and substantially improved performance for the simulation of biophysical and biochemical models.

Keywords: NEURON, simulation, neuronal networks, multiscale computer modeling, systems biology, computational neuroscience

1. INTRODUCTION

NEURON is an open-source simulation environment that is particularly well suited for models of individual neurons and networks of neurons in which biophysical and anatomical complexity have important functional roles (Hines and Carnevale, 1997). Its development started in the laboratory of John Moore at Duke University in the mid-1980s as a tool for studying spike initiation and propagation in squid axons. Subsequently it underwent massive enhancements in features and performance and it is now used for models that range in scale from subcellular (McDougal et al., 2013) to large networks (Migliore et al., 2006). Today it is one of the most widely used simulation environments for biologically detailed neurosimulations (Tikidji-Hamburyan et al., 2017).

Einevoll et al. (2019) have argued that the central role of simulation software in neuroscience is analogous to physical infrastructure in other scientific domains, such as astronomical observatories and particle accelerators, and that the resources required to build and maintain software should be considered in this context. The increasing importance of software in science is, however, not specific to neuroscience. Crouch et al. (2013) and Hettrick et al. (2014) found that there is a general trend that science relies more and more on software with the capability to automate complex processes and perform quantitative calculations for prediction and analysis. Unfortunately, this reliance on software also has inherent and increasing risks (Miller, 2006). The need for better software sustainability, correctness and reproducibility (McDougal et al., 2016; Mulugeta et al., 2018) has prompted initiatives and proposals suggesting better practices when developing scientific software (Crouch et al., 2013; Erdemir et al., 2020) and when publishing computational results (Heroux, 2015; Willenbring, 2015). In practice, however, it remains difficult to always have the right training, resources and overall understanding to develop good software and use it correctly. Bartlett et al. (2012) and Gewaltig and Cannon (2014) further illustrate how productive use of a software application can lead to development and use beyond its original scope. This, in turn, increases its complexity and can render a once-straightforward implementation unwieldy and hard to maintain.

Another challenge is that of software portability. A user may, rightfully, expect that a scientific software runs on different operating systems and makes good use of all the installed hardware, which in today's systems often means a combination of a multi-core CPU and a powerful graphics processing unit (GPU). The number and diversity of these hardware architectures is expected to continue to increase as hardware architects seek to further exploit problem specificities in their designs (Hennessy and Patterson, 2017). The increasing difficulty in miniaturizing transistors will amplify the trend toward architectural heterogeneity (Hennessy and Patterson, 2019). From a software point of view, maintaining portability for this diversity of platforms is a fundamental challenge. The more target platforms that need to be supported, the bigger the risk that this leads to multiple redundant code segments with potentially different programming syntax, compilation configurations, and deployment mechanisms, which are error-prone and labor-intensive to maintain. In the software development world, mechanisms and paradigms have been found that facilitate writing more portable software, such as programming paradigms that support multiple architectures (Wolfe, 2021), and modern continuous integration mechanisms (Meyer, 2014). If we want to be able to keep benefiting from future hardware developments in neuroscience, neurosimulator software will have to fully engage with the portability challenge.

Another important challenge is running computational models quickly and efficiently, including those of large size. This requires understanding the computational nature of the scientific problem, which computer system is best suited (Cremonesi and Schürmann, 2020; Cremonesi et al., 2020), and optimization of data structures and algorithms for specific hardware architectures (Jordan et al., 2018; Kumbhar et al., 2019). Given the multitude of computational models and diversity of computer architectures, it has become necessary to use various automated approaches to generate optimized versions of the software. Examples include modern compiler techniques for code generation from domain specific languages (e.g., Blundell et al., 2018; Akar et al., 2019; Kumbhar et al., 2020), or just in time compilation (Lam et al., 2015), as well as the use of platform-optimized libraries (e.g., Agullo et al., 2009; Carter Edwards et al., 2014) and new abstraction layers that anticipate heterogeneous architectures (Beckingsale et al., 2019).

A major confounding factor to the aforementioned challenges is that popular scientific codes, such as NEURON (Hines and Carnevale, 1997) or NEST (Gewaltig and Diesmann, 2007), have often been developed over long periods and include key source code that was written without the benefit of modern development tools, libraries and software programming practices. The necessity of modernizing scientific codes is increasingly recognized (Neely et al., 2017; de Verdière, 2020) and does not spare brain simulator software projects (Brette et al., 2007). In the case of NEST these modernizations happened over the past few years, spanning a wide range of both algorithmic and technical improvements (Pronold et al., 2022). Others, such as the Brian Simulator (Goodman, 2009), have decided to rewrite their codes from scratch, taking the opportunity to overcome limitations of their previous implementations, such as allowing for flexibility in model specification while improving simulator performance (Stimberg et al., 2019). It also prompted the inception of new simulator projects, such as the Arbor simulator (Akar et al., 2019), where the developers sought to start from a design philosophy that prefers standard library data structures, code generation, and which minimizes external dependencies. The flip side of such a fresh start is that it is difficult to maintain full backward compatibility with existing models.

Lastly, with more complex scientific workflows, the notion of software being a single do-it-all tool is slowly waning and one should rather think of it as a building block in a larger eco-system. This trend can be seen in efforts like the EBRAINS research infrastructure[1], where multiple tools are combined into intricate scientific workflows (Schirner et al., 2022). The Open Source Brain (Gleeson et al., 2019) and tools such as NetPyNE (Dura-Bernal et al., 2019), LFPy (Lindén et al., 2014), Bionet (Gratiy et al., 2018) and BluePyOpt (Van Geit et al., 2016) use NEURON as a library and augment it with additional features. As an example, NetPyNE is a high-level Python interface to NEURON that facilitates the development, parallel simulation, optimization and analysis of multiscale neural circuit models.

Here, we report on our efforts to modernize the widely-used NEURON simulator, improving its sustainability, portability,

[1] https://ebrains.eu/

and performance. We have overhauled NEURON's overall code organization, testing, documentation and build system with the aim of increasing the code's sustainability. We have integrated NEURON with an efficient and scalable simulation engine (CoreNEURON) and a modern source-to-source compiler (NMODL) capable of targeting both CPUs and GPUs. We demonstrate the performance of several large-scale models using different multi-CPU and multi-GPU configurations, including running simulations on Google Cloud using NetPyNE. Finally, we updated NEURON's reaction-diffusion simulation capabilities with just-in-time compilation, more seamless integration with the rest of NEURON, support for exporting to the SBML format, and support for 3D intra- and extra-cellular simulation.

2. METHODS

Over the years the NEURON simulator has been developed to accommodate new simulation use-cases, support community tools and file formats, and adopt new programming paradigms to benefit from emerging computing technologies. During this process, various software components have been developed and external libraries have been integrated into the codebase. Like many scientific software packages, maintaining a codebase developed over four decades poses a significant software engineering challenge.

Since the 7.8 release the NEURON developer community has launched a variety of initiatives to future-proof the simulator codebase. **Figure 1** summarizes the high-level functional components of the NEURON simulator and the various changes described in the rest of this section. These developments happened since 2020 over the course of 2 years starting with the refactoring of the build system and the introduction of a new continuous integration (CI) system. While especially the latter is an ongoing effort, both developments have allowed various improvements to documentation, testing and packaging. The tighter integration of CoreNEURON into NEURON and its various performance and hardware portability improvements were implemented in parallel and were able to quickly benefit from the new build system and CI made available in NEURON.

2.1. Improving Software Sustainability Through Code Modernization and Quality Assurance

To address some of the shortcomings of NEURON's codebase accumulated over four decades of continued development, we have implemented a number of far-reaching changes and adopted a new modern development process with the aim of streamlining the handling of code contributions, while at the same time improving code quality and documentation. First, we have replaced the legacy GNU Autotools based build system with CMake. Second, this allowed us to introduce a comprehensive automated build and CI system using GitHub Actions. Third, these two changes allowed us to create a modern binary release system based on Python wheels. Finally, we have further extended these components to integrate a code coverage monitoring service and automatically build user and developer documentation.

2.1.1. Modern Build System Adoption

Until recently the build system of NEURON used GNU Autotools. Autotools, the de-facto standard on Unix-like systems, is a build system used to assist the various build steps of software packages. CMake is a modern alternative to Autotools that offers many advantages and features important for the continued development of NEURON. First, it has extensive support for customizing C and C++ builds, from language standards, to fine-tuning compile and link-time arguments. Furthermore, it supports build portability across hardware platforms (i.e., x86_64, ARM, GPUs), operating systems (i.e., Linux, macOS, Windows) and compilers (GCC, Clang, Intel, NVIDIA, etc.). It also allows a more robust integration with external dependencies. Finally, CMake is being actively developed and supported by a large community of open source and industry developers.

We decided, therefore, to replace NEURON's legacy Autotools build system by CMake and reimplemented the entire configure and build process using CMake for NEURON as well as its libraries such as CoreNEURON and Interviews. The build-system reimplementation allowed us to refactor large parts of the auxiliary code used for configuring, packaging and installing NEURON in order to make it more robust and maintainable.

Using CMake we are able to provide a build configuration that goes far beyond GNU Autotools in several respects. For instance, we have included the ability to automatically clone and integrate other CMake based libraries like CoreNEURON and NMODL using CMake options. The build is organized in various build targets producing multiple shared libraries for the interpreter, solvers, simulator, the native interfaces of the Python API, RxD, CoreNEURON, and the main neuron executable, `nrniv`.

2.1.2. Continuous Integration and Build Automation

Continuous integration (CI) is crucial for the development process of any software project. It allows the development team to check the correctness of code changes over the course of the project's development life cycle. To support our work in modernizing the NEURON codebase and opening up the development process to a wider community it was, therefore, important to first put a CI in place. **Figure 2A** gives an overview of the CI workflow using GitHub Actions and Azure. Every time a new *Pull Request* is opened on the NEURON repository, CI pipelines for building and testing NEURON are executed on Linux, Windows and macOS. These also rebuild the documentation, executing all embedded code snippets and generate test code coverage reports, which are provided to the code reviewer to aid them in evaluating the proposed change to the code. As part of the CI workflows, we also need to build and test binary installers as well as python wheels for various platforms. As Github Actions provides limited concurrent builds for open source projects, we use Azure CI workflows for building artifacts such as installers and python wheels. This helps us to reduce overall CI turnaround time.

Using the *Pull Request* CI workflow, we ensure that the proposed change will not introduce bugs or unintended side-effect, by testing the majority of build options along with different compiler versions. The jobs also build Python wheel packages (see Section 2.1.4 for more details) and test the integrity

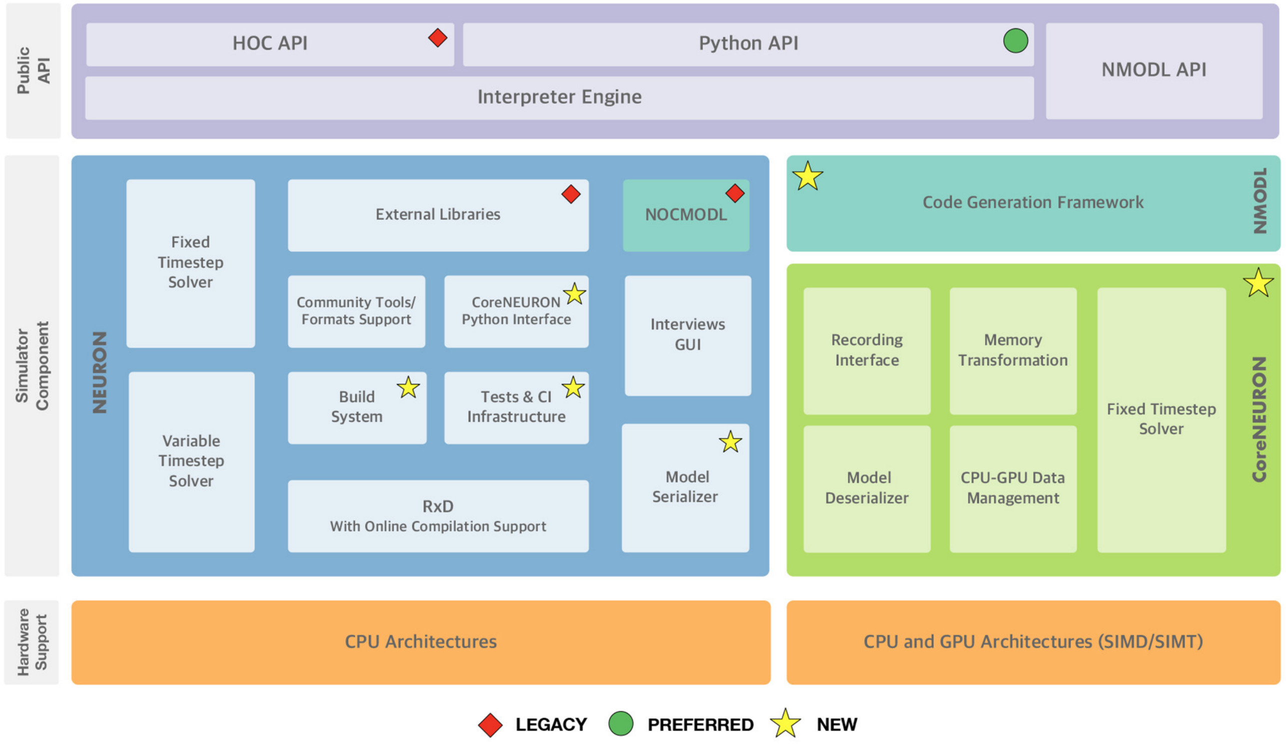


FIGURE 1 | NEURON Simulator Overview: At the top, the "Public API" layer shows NEURON's three application programming interfaces exposed to end-users: the legacy HOC scripting interface, the preferred Python interface, and the NMODL DSL for defining channel and synapse models. In the middle, the "Simulator Component" layer shows the main three different software components and their interal sub-components: NEURON, the main modeling and simulation environment, CoreNEURON, a compute engine for NEURON targetted at modern hardware architectures including GPUs, and NMODL, a modern compiler framework for the NMODL DSL. At the bottom, supported hardware architectures are shown. Software components that are newly added or are deprecated are highlighted.

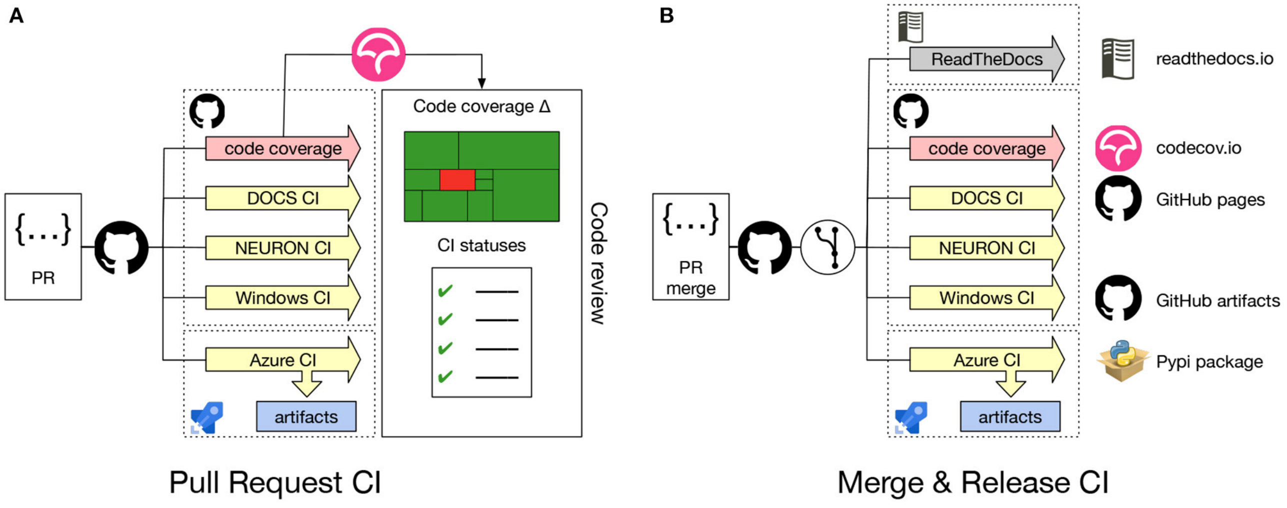


FIGURE 2 | (A) *Pull Request* CI workflow: Whenever a *Pull Request* is opened jobs are started to build and test NEURON on Linux, Windows and macOS. Several combinations of build options and versions of dependencies are tested. Also, Python wheels are built, the documentation is regenerated, executing embedded code snippets and code coverage metrics are determined. All CI job results are reported to the code reviewer. **(B)** Merge & Release CI workflow: Automated CI jobs are also started after a PR has been merged, nightly or on a new version release. These jobs produce artifacts that are delivered into appropriate channels.

of the resulting packages by running NEURON's unit and integration test suites. These jobs are executed both on Linux and macOS systems. Because Windows is a substantially different environment, we needed to develop a separate CI to build and test the NEURON installer package using the `MinGW` compiler and `MSYS2` environment[2].

Contrary to the *Pull Request* CI, the *Merge & Release* CI (**Figure 2B**) is executed on the latest master branch of the NEURON codebase or with a new release. Artifacts such as the latest documentation, Python wheels, binary packages and the Windows installer are built.

Once we had set up a robust build system and CI workflow, we were able to extend the framework to offer additional features both for the developer and the user community. First, as the name suggests, the *Docs CI* builds a NEURON Python wheel package and rebuilds the NEURON documentation (described in more detail in Section 2.1.3). Second, we have added a *code coverage* workflow, which builds NEURON with all features enabled and runs all tests tracking code coverage using `lcov` and `pytest-cov`.

2.1.3. Documentation Generation

NEURON's documentation consists of a number of resources covering the Interviews graphical interface, the HOC and Python APIs, the NMODL domain specific language (DSL), and development best practices.

As part of our effort to streamline the development process and modernize the organization of NEURON, we consolidated NEURON's documentation sources into the main repository and automated the documentation building. Additional resources, such as tutorials, in the form of Jupyter notebooks, were also gathered and integrated into the NEURON documentation.

We integrated documentation building into the CMake build system and CI pipelines. In the CI pipelines, we start by executing the Jupyter notebooks using a freshly built NEURON wheel, ensuring that existing notebooks are compatible with the latest code. Once the notebooks have been successfully executed, they are then converted to HTML. Next, Doxygen code documentation is generated. Finally, the manual and developers guide are re-built with Sphinx, embedding the previously generated Jupyter notebooks and Doxygen. This documentation is then published on the ReadTheDocs[3] where we provide versioned documentation starting with the NEURON 8.0.0 release.

2.1.4. A Modern NEURON Python Package

NEURON was originally packaged as a traditional software application, made available as a binary package for mainstream operating systems and alternatively as a source tarball. Alternatively, NEURON could also be installed as a Python package through a laborious multistep process that lacked flexibility and was error prone. With the introduction of a standard, complete Python interface (Hines et al., 2009), NEURON could be more readily used from a Python shell. Thanks to the user friendliness and strong scientific ecosystem of the Python language, this API quickly became popular in the NEURON community. At the same time, the Python wheel package format has become an extremely popular means of distributing Python software packages, allowing the user to install Python packages using `pip install`.

To provide the flexibility of `pip`-based installation, using the CMake build system we implemented a new NEURON Python package shown in **Figure 3**. This package is comprised of C/C++ extensions providing the legacy `hoc` language, the Python interface, reaction-diffusion module (`rxd`) with Cython extension, and several pure Python modules. To build functional Python extensions it is important to provide the build system with the correct build flags and paths compatible with the target Python framework. To achieve this we extended the `Extension` class of `setuptools`.

One of the challenges faced when building the NEURON Python package was distribution of binary executables and support for compiling `MOD` files on the user's machine. Python extensions are typically built as shared libraries and automatically placed in the correct location in the package path to be found at runtime. However, compiled executables such as `nrniv` are treated by `setuptools` as binary data and not installed into the Python framework's `bin` folder. Also, we needed to support the `nrnivmodl` workflow where the user can compile mechanism and synapse models from `MOD` files and load the corresponding library at runtime. In order to support this, we created Python shims that take the place of the actual NEURON executables in the `bin` folder. These shims prepare the runtime environment and call the homonymous NEURON binary executable via the `execv` routine, which substitutes the shim process with the NEURON executable.

2.2. Integration of CoreNEURON Within NEURON

CoreNEURON (Kumbhar et al., 2019) is a simulation engine for NEURON optimized for modern hardware architectures including CPUs and GPUs. CoreNEURON is developed and maintained in its own public repository on GitHub. Previously it was up to the user to obtain API-compatible versions and build NEURON and CoreNEURON from source separately. The use of CoreNEURON was also not straightforward as it had to be run as a separate executable. To simplify the usage, CoreNEURON and NEURON are now coupled more tightly in terms of code organization, build system and implementation level.

First of all, we have integrated CoreNEURON (along with NMODL) as git submodules of the `nrn` repository, allowing us to build single software distribution packages containing optimized CPU and GPU support via CoreNEURON. This also simplifies tracking changes in the various repositories and making sure that the correct code revisions are distributed and built together. Secondly, we have integrated CoreNEURON and NMODL building into NEURON's CMake build system. This is possible thanks to CMake's robust support of subprojects that allow easy popagation of build parameters across the various code bases. Thirdly we have implemented an in-memory model transfer,

[2] https://www.msys2.org/

[3] https://readthedocs.org/

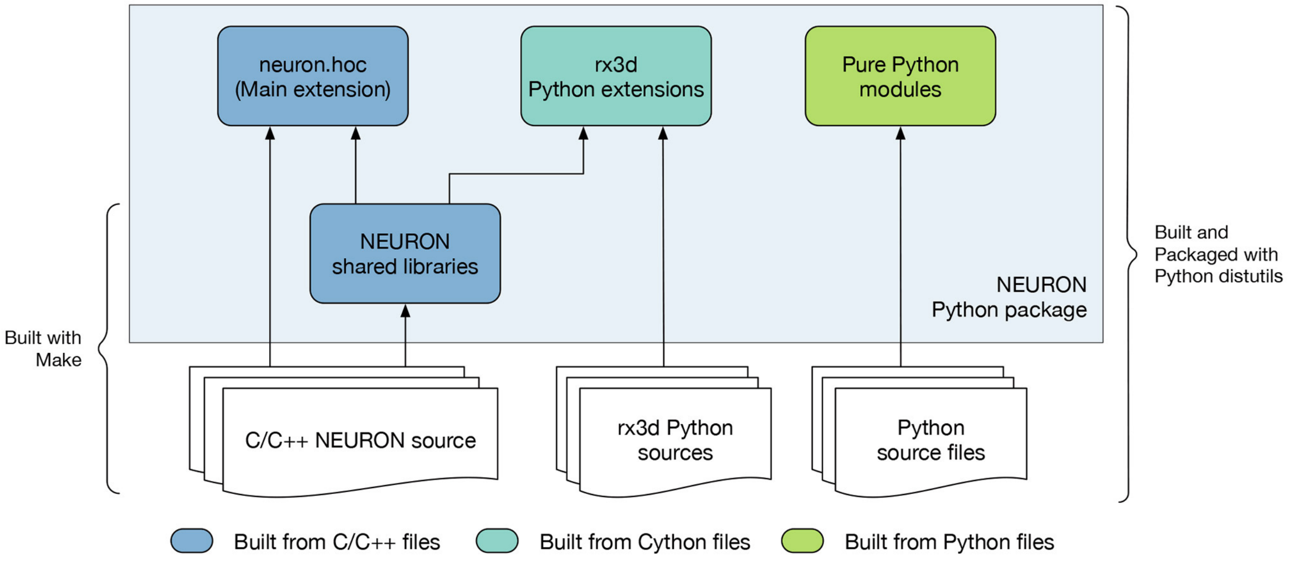


FIGURE 3 | Overview of the NEURON Python package. The package is comprised of pure Python modules and extension code. The main NEURON extension is written in C/C++ and provides the API for the NEURON interpreter. Additionally, the rx3d modules are written in Cython providing the reaction-diffusion solvers of NEURON.

improved GPU support in CoreNEURON, and integrated our `MOD` file code generation pipelines. Some of the important changes are discussed in the remainder of this section.

2.2.1. Transparent Execution via Coreneuron Using In-memory Model Transfer

While CoreNEURON can be run as a standalone application, it still requires the model created by NEURON as an input. This model can be written to disk and transferred to CoreNEURON via files. This approach has the advantage that a large model can be constructed once by NEURON and stored to files, which can be later run by CoreNEURON many times (for example for ensemble runs). Also, this allows the models with huge memory requirement to be constructed in smaller pieces by NEURON before being executed simultaneously by CoreNEURON, thanks to it having a $5-6\times$ smaller memory footprint than NEURON. However, we found that in practice this workflow is not flexible enough for many users.

To address this we have now implemented two-way in-memory data transfer between NEURON and CoreNEURON, which greatly simplifies CoreNEURON usage. With this, it is now possible to record cell or mechanism properties (e.g., voltage, current, variables of type `STATE, PARAMETER, RANGE, ASSIGNED` defined in `MOD` files), unlike with the file-transfer mode where only spikes can be recorded. In case of NEURON, all of the data structures representing a model are laid out in an Array-of-Structures (AoS) memory layout. This allows easy manipulation of sections, channels and cells at runtime, but it is not optimal for memory access and Single Instruction Multiple Data (SIMD) execution on modern CPU/GPU architectures. Hence, NEURON data structures are serialized and transferred to CoreNEURON where they are transposed into a Structure-of-Array (SoA) memory layout. This allows efficient code vectorization and favors coalesced memory access, which is important for runtime performance. In addition to data structures representing cells and network connectivity, event queues are now also copied back and forth between NEURON and CoreNEURON, allowing simulations to be run partly with NEURON and partly with CoreNEURON if desired. For end users, all this functionality is now exposed via a new Python module named `coreneuron`. The API and new options are discussed in Section 3.3.

2.2.2. Enabling GPU Offloading in NEURON Simulations

It is now possible to offload NEURON simulations transparently to GPUs using CoreNEURON. This support is implemented using the OpenACC programming model. When CoreNEURON GPU support is enabled, all data structures representing the model are copied to GPU memory when initializing CoreNEURON. State variables for the Random123 (Salmon et al., 2011) library are also allocated on the GPU using CUDA unified memory. Once the data are transferred, they reside in the GPU memory throughout the simulation. This simplifies memory management and reduces expensive CPU-GPU memory transfers.

All computationally intensive kernels of the main simulation loop are offloaded to the GPU, including state and current updates in `MOD` files, and the Hines solver (Hines, 1984). Even though the spike detection kernels are not computationally expensive, they are offloaded to GPU too, to benefit from data locality and avoid additional data transfers. Only the spike events generated on the GPU and user-requested state variables are copied back to CPU. Similarly, the spikes communicated by other processes are first queued on the CPU and then transferred to GPU.

For some models the Hines solver can consume a significant fraction of the total simulation time when running on GPUs. This is often due to the limited parallelism and non-coalesced memory access arising from heterogeneous, branched tree structures in neuron morphologies. As presented by Kumbhar et al. (2019), two node ordering schemes (called cell permutations) were developed to improve the parallelism and coalesced memory accesses. We have further improved this implementation, reducing solver execution time on GPU by an additional 15 — —20%. Finally, simulations running on GPU can now utilize multiple GPUs available on a compute node. The available GPUs are uniformly distributed across MPI processes and threads.

2.2.3. Integration of Code Generation Pipelines

An integral part of NEURON's modeling capability is provided via the NMODL DSL. This allows the user to describe in their models a wide range of membrane and intracellular submodels such as voltage and ligand gated channels, ionic accumulation and diffusion, and synapse models. Such models are written in `MOD` files and then translated to lower level C or C++ code using a source-to-source compiler (transpiler).

In order to support execution via both NEURON or CoreNEURON, each `MOD` file is now translated twice: first into C code for NEURON, and then into C++ for CoreNEURON. This workflow is illustrated in **Figure 4**. `nrnivmodl` remains our main tool for processing user provided `MOD` files. By default, all input `MOD` files are translated into C code by NEURON's legacy NOCMODL transpiler. These files are then compiled to create a library for NEURON called `libnrnmech`. If a user provides the `-coreneuron` CLI option then either the default MOD2C or the new NMODL transpiler (Kumbhar et al., 2020) is used to translate `MOD` files into C++ files. These files are then compiled to create a library for CoreNEURON called `libcorenrnmech`. The NMODL transpiler generates modern, optimized C++ code that can be compiled efficiently on CPUs or GPUs. These two libraries are finally linked into an executable called `special`. When running on the CPU the user has the choice between using `python`, `nrniv` or the `special` executable to launch simulations. When running on a GPU, however, one must use the `special` executable to launch simulations due to limitations of the NVIDIA compiler toolchain when using OpenACC together with shared libraries.

2.3. Modular NEURON: The Example of NetPyNE

NetPyNE (Dura-Bernal et al., 2019) is a high-level declarative NEURON wrapper used to develop a wide range of neural circuit models (Metzner et al., 2020; Anwar et al., 2021; Bryson et al., 2021; Pimentel et al., 2021; Ranieri et al., 2021; Romaro et al., 2021; Volk et al., 2021; Borges et al., 2022; Dura-Bernal et al., 2022a,b; Medlock et al., 2022)[4], and also as a resource for teaching neurobiology and computational neuroscience.

The core of NetPyNE consists of a standardized JSON-like declarative language that allows the user to specify all aspects of the model across different scales: cell morphology and biophysics (including molecular reaction-diffusion), connectivity, inputs and stimulation, and simulation parameters. The NetPyNE API can then be used to automatically generate the corresponding NEURON network, run parallel simulations, optimize and explore network parameters through automated batch runs, and visualize and analyze the results using a wide range of built-in functions. NetPyNE can calculate local field potentials (LFPs) recorded at arbitrary locations and has recently been extended to calculate current dipoles and electroencephalogram (EEG) signals using LFPykit (Hagen et al., 2018).

NetPyNE also facilitates model sharing by exporting to and importing from the NeuroML and SONATA standardized formats. All of this functionality is also available via a web-based user interface[5]. Both Jupyter notebooks and graphical interfaces are integrated and available via the Open Source Brain (Gleeson et al., 2019) and the EBRAINS (Amunts et al., 2019) platforms.

Simulating large models in NEURON/NetPyNE is computationally very expensive. Thus, enabling CoreNEURON within NetPyNE is very attractive and may provide large gains. Thanks to the tighter integration of CoreNEURON into NEURON (see Section 2.2), we were able to easily integrate CoreNEURON solver support into NetPyNE. We used the `coreneuron` Python module provided by NEURON and added three new configuration options in the simulation configuration object of NetPyNE: `coreneuron` to enable CoreNEURON execution, `gpu` to enable or disable GPU support, and `random123` in order to enable Random123-based random number generators. Users can now enable these features by simply setting the above options in their NetPyNE simulation configuration file.

2.4. Enabling New Use-Cases With Reaction-Diffusion Integration

The NEURON reaction-diffusion module (RxD, McDougal et al., 2013) provides a consistent formalism for specifying, simulating, and analyzing models incorporating both chemical signaling (chemophysiology) and electrophysiology. Such models are common in neuroscience as, for example, calcium concentration in the cytosol affects the activity of calcium-gated potassium channels. Before NEURON's RxD module, these models incorporated chemical effects in any of a variety of ways using custom NMODL code; this variation unfortunately made some such models incompatible with each other and posed challenges when combining the custom code with NEURON's built-in tools. Due to the use of custom solutions, they also generally combined simulation methodology with model description; NEURON's RxD module, by contrast, explicitly separates the two, allowing, for example, the same model to be used for both 1D and 3D simulation. Recent enhancements to RxD have focused on improving its domain of applicability and usability through changes to the interface and redesigning the backend for more flexibility and faster simulation.

[4] http://netpyne.org/models

[5] http://gui.netpyne.org

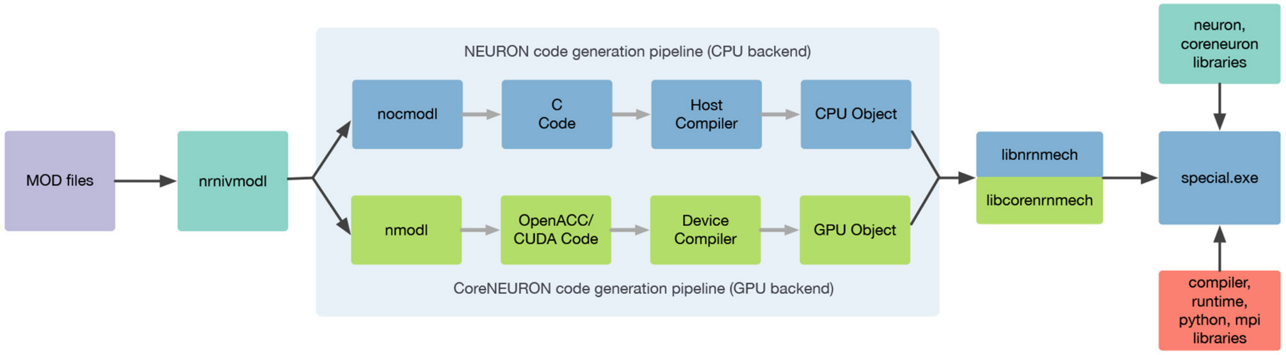

NEURON code generation pipeline (CPU backend)
MOD files
nrnivmodl
nocmodl
C Code
Host Compiler
CPU Object
nmodl
OpenACC/ CUDA Code
Device Compiler
GPU Object
CoreNEURON code generation pipeline (GPU backend)
libnrnmech
libcorenrnmech
special.exe
neuron, coreneuron libraries
compiler, runtime, python, mpi libraries

FIGURE 4 | `nrnivmodl` workflow where input `MOD` files are translated into C/C++ code for NEURON and CoreNEURON targeting different hardware platforms via NOCMODL and NMODL. The output of this workflow is a `special` executable containing NEURON and CoreNEURON specific libraries.

A number of features have been implemented to expand the ability of RxD to better represent a researcher's conceptual model. An `Extracellular` region type (Newton et al., 2018) provides support for studying cellular interactions through changes in the extracellular space (e.g., in ischemic stroke or between neurons and astrocytes), simulated using a macroscopic volume averaging approach. Three-dimensional intracellular simulation (McDougal et al., 2022) allows study of microdomains and the sensitivity to precise positioning of synapses. Importantly, each of these extensions was designed to fit within the broader RxD context; reaction and diffusion rules are specified and interpreted in the same way for 1D and 3D simulation and for intra- and extracellular simulation. Current through NMODL-DSL-specified ion channels generates a flux in the corresponding extracellular compartment and ionic Nernst potentials are updated based on the 3D extracellular concentration. Both 3D intra- and extra-cellular dynamics are calculated using a parallelized adaptation of the Douglas-Gunn alternating direction implicit method (Douglas and Gunn, 1964). For models not requiring full 3D simulation, it is still sometimes advantageous to account for geometry changes (e.g., in a model using nested shells to account for radial variation, a spine most naturally connects to only the outer-most shell); to allow modelers to address this connection, we added a `MultipleGeometry` to explicitly bridge across geometry changes.

Other interface enhancements focused on extending RxD's usability. We have worked to make existing NEURON tools work directly with RxD objects. For example, `h.distance` computes path distance between two points, whether they are RxD nodes or segments. Likewise, the NEURON-specific graph types (`h.PlotShape` and `h.RangeVarPlot` for visualizing concentration across an image of the cell and along a path) can plot traditional NEURON variables (e.g., `v` or `cai`) as well as RxD chemical `Species` in whatever region (cytosol, ER, etc.) when using the graph's matplotlib or pyplot backends. We added a `neuron.units` submodule with conversion factors to facilitate specifying models with rate constants most naturally expressed in specific units (e.g., circadian models involve protein concentrations that change over hours whereas the gating variable on a sodium channel may have a time constant of milliseconds). A new `rxd.v` variable allows using the RxD infrastructure to include dynamics driven by membrane potential, offering an alternative to specifying ion channel kinetics through NMODL files. To allow studying NEURON RxD models with other tools, we introduced `neuron.rxd.export.sbml` which allows exporting reaction dynamics at a point to SBML, a standard for representing systems biology models (Keating et al., 2020).

Additionally, we modified the backend for `h.SaveState` to introduce an extensible architecture for storing and restoring new types of state variables. Python functions within the `neuron` module allow registering SaveState extension types, which consist of specifying a unique identifier, a function to call that serializes the corresponding states, and a function that expands a serialized representation. We implemented an RxD extension that registers itself when RxD states are defined (in particular, importing the module alone does not trigger the registration). When no extensions (including RxD) are present in the model, the saved file is bitwise identical to previous (NEURON 7.x) versions; when extensions are present, the saved data includes a new version identifier, is otherwise identical to the previous version, but ends with binary encoded data representing the number of save extensions used in the model, an identifier for each used save extension, the length, and data to be passed to the extension. In the case of the RxD SaveState extension, as the state data is potentially voluminous, the serialized data is zlib compressed.

We redesigned the RxD backend to improve the flexibility of interactive model specification and debugging. `Region` objects now take an optional `name` argument that can be specified at creation or after to help distinguish them during debugging. For both intra- and extracellular 3D simulation, each `Species`, `State`, and `Parameter` has its values stored in independent memory locations accessed by a pointer to the corresponding mesh. This architecture allows pointers to be preserved as new `Species`, etc. are defined and old ones are removed. To improve the portability of cells between models, we replaced the requirement that a given species (e.g., calcium) could only be defined once with a requirement that there be no overlapping versions of the same species. This change now allows each cell object in NEURON to fully specify its kinetics, including the reaction-diffusion aspects, thus allowing such cells to be reused in other models without further code changes.

With successive releases of NEURON, we iteratively improved the performance of RxD simulation. We moved all simulation code to C++ and compile reaction specifications to eliminate Python overhead. Reactions continue to be specified in Python as before, but now contain a method that generates a corresponding C file; this method is automatically called when first needed and the corresponding file is compiled and dynamically loaded into NEURON. We replaced the matrix solving algorithm used in 3D RxD simulations with one that exploits the decoupled nature of the reaction-contribution to the Jacobian that arises from reactions only happening between molecules at the same spatial location. Multi-threading support was added to 3D simulations. 3D diffusion rates depend on the concentration at a node and up to six of its neighbors. To minimize cache-miss latency when accessing neighboring voxels when simulating extracellular diffusion, `__builtin_prefetch` was used to move data into a cache before accessing it. For extracellular diffusion, prefetching provides a modest improvement depending of the size of the simulation, e.g., 5% speed-up in a 128^3 cube of voxels. NEURON does not use prefetching with intracellular simulation, as in practice we observed no comparable speedup. Finally, to accelerate both the initialization and simulation of models with reaction-diffusion dynamics to be studied in full 3D, we now construct voxel based representations of each of its constituent convex components (frusta and their joins) on a common mesh and merge them together (McDougal et al., 2022), instead of constructing a voxel-based representation of an entire neuron morphology at once.

3. RESULTS

3.1. Sustainability Improvements Through Modern Development Practices

3.1.1. Toward a Development Community

As described in Sections 2.1, 2.1.1, and 2.1.2, we have radically updated NEURON's development life cycle to be a modern and collaborative process. First, new developers are now able to quickly get started thanks to improved documentation (Section 2.1.3). Users can access the latest release documentation at https://nrn.readthedocs.io/en/latest/ and a nightly documentation snapshot at https://neuronsimulator.github.io/nrn/. Second, a modernized build system eases local setup and testing of proposed code changes. A single repository, https://github.com/neuronsimulator/nrn, now provides access to all software components including Interviews, CoreNEURON, NMODL, tutorials, and documentation. The integrated CMake build system across these components provides a uniform interface to build all components with ease. Third, code contributions are automatically checked using a comprehensive CI suite. This increases programmer confidence and helps reviewers to more quickly evaluate proposed changes.

These improvements have directly led to the adoption of a collaborative development process with a lively community. As an example, **Figure 5** depicts commits over time since the `nrn` git repository was started in November 2007. We can see that NEURON has been receiving more and more contributions from new developers in the last 3 years.

3.1.2. Software Sustainability Through Development Ecosystem Modernization

Build system modernization, removing obsolete dependencies, and the introduction of CI pipelines have lead to a vastly streamlined development process. First, using fewer external dependencies sped up and simplified the build process and reduced build times. Second, replacing the Autotools build system with CMake has allowed us to more simply integrate Interviews, CoreNEURON and NMODL in the build and has made build system changes more maintainable. Third, the removal of Autotools and support for legacy Python 2 has simplified the overall code structure, and subsequent code refactoring has become less complex. Finally, as a consequence of the described build system improvements, the CI configuration has been simplified as fewer build combinations need to be tested, which in turn has enabled us to integrate additional build jobs into the CI, such as automated Python wheel builds, Windows installer builds, test code coverage and documentation generation.

Thanks to having centralized documentation and automated documentation building in the CI, developers are now able to more easily find information on how to build, install, configure, debug, profile, measure test code coverage, manage releases and versioning, and build Python wheels. The user documentation has also become more accessible and searchable, which makes NEURON more accessible to the community.

Since the introduction of test code coverage tracking we were able to increase test coverage by almost 18%. For the code components that are being maintained and developed by the NEURON community, more than half the code is covered by tests. Introducing systematic testing and coverage reports has allowed us to keep track of our progress and facilitate the refactoring and maintenance of the code. Up to date code coverage reports are available at https://app.codecov.io/gh/neuronsimulator/nrn.

3.2. Improved Software and Hardware Portability

3.2.1. Streamlined NEURON Software Distributions

Building distributions (Windows and macOS installers, Debian packages) and testing them reliably on different platforms against different software toolchains has historically been a major hurdle to NEURON releases. This had been extending release cycles and delaying the introduction of new features. With the CI pipelines described in Section 2.1.2 and a modernized CMake-based build system, we are now able to automatically build portable NEURON Python wheels for Linux and macOS (including Apple M1) and an installer for Windows. With the infrastructure in place we are able to offer both nightly wheels and installers, containing the latest changes, as well as more rigorously tested,

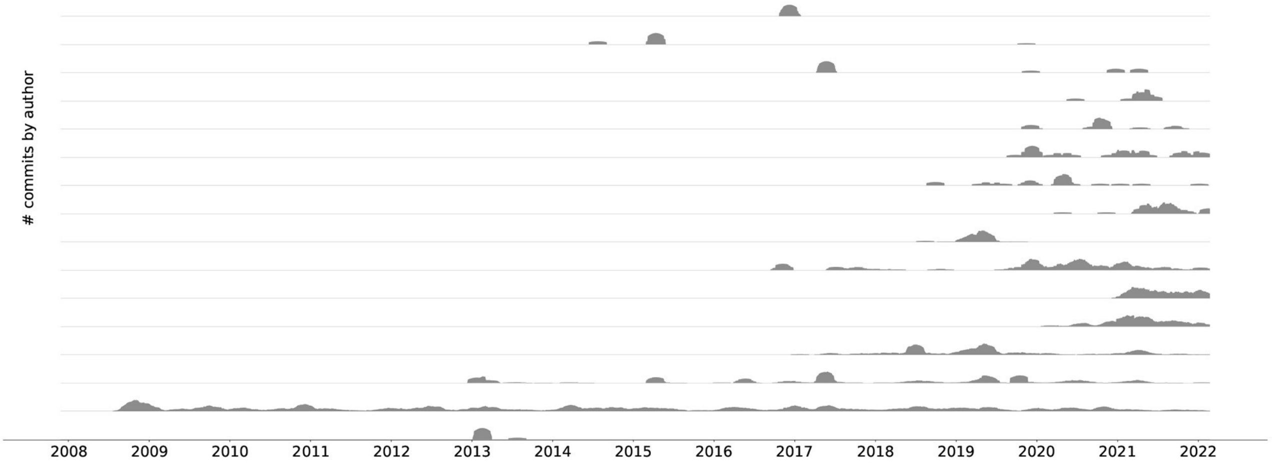


FIGURE 5 | An aligned commit series[6] plot showing the top 16 contributors since the beginning of the git repository history. For better visibility the different time series do not share y-axes. A clear trend toward collaborative development can be seen.

and hence stable, release builds. By simplifying, automating and documenting the build steps, we have streamlined the process of creating new NEURON releases.

Python's dominance in scientific workflows and the widespread use of Python-based data processing, analysis and visualization tools in the scientific community can be attributed to the ease-of-use and portability of these packages. By distributing NEURON as a Python package we are embracing this trend and making it easier to adopt NEURON. These Python wheels, as well as binary installers, provide a full, portable distribution of NEURON targeting desktop environments, cloud instances and HPC clusters alike. Although we do not currently provide a Python wheel for Windows, users can make use of the Windows Installer or the Linux Python wheel using Windows Subsystem for Linux (WSL). All these distributions have support for dynamically loading MPI, Python, and Interviews graphics. This ease of use has made distribution via Python wheels the preferred way of installing NEURON. For Python wheels, we currently see an average of around 3000 downloads/month, and over 17,000 downloads in the last 6 months. Released wheels are available via https://pypi.org/project/NEURON.

3.2.2. Improved Hardware Portability

Supporting a wide range of use-cases requires strong hardware support for architectures ranging from laptops to cloud and HPC platforms. Thanks to its updated and improved build system it is straightforward to build NEURON and CoreNEURON for a variety of hardware platforms including x86, ARM64, POWER and NVIDIA PTX. The respective vendor compilers are able to take advantage of CoreNEURON's improved data-structures and produce optimized code. In order to make CoreNEURON's GPU backend accessible to the wider user community we have additionally created NEURON Python wheels with GPU acceleration enabled. Currently these specialized wheels can only be used on environments with NVIDIA GPUs with compute capability 6, 7 or 8 and the NVIDIA HPC SDK version 22.1 with CUDA 11.5. These wheels can be downloaded from https://pypi.org/project/NEURON-gpu.

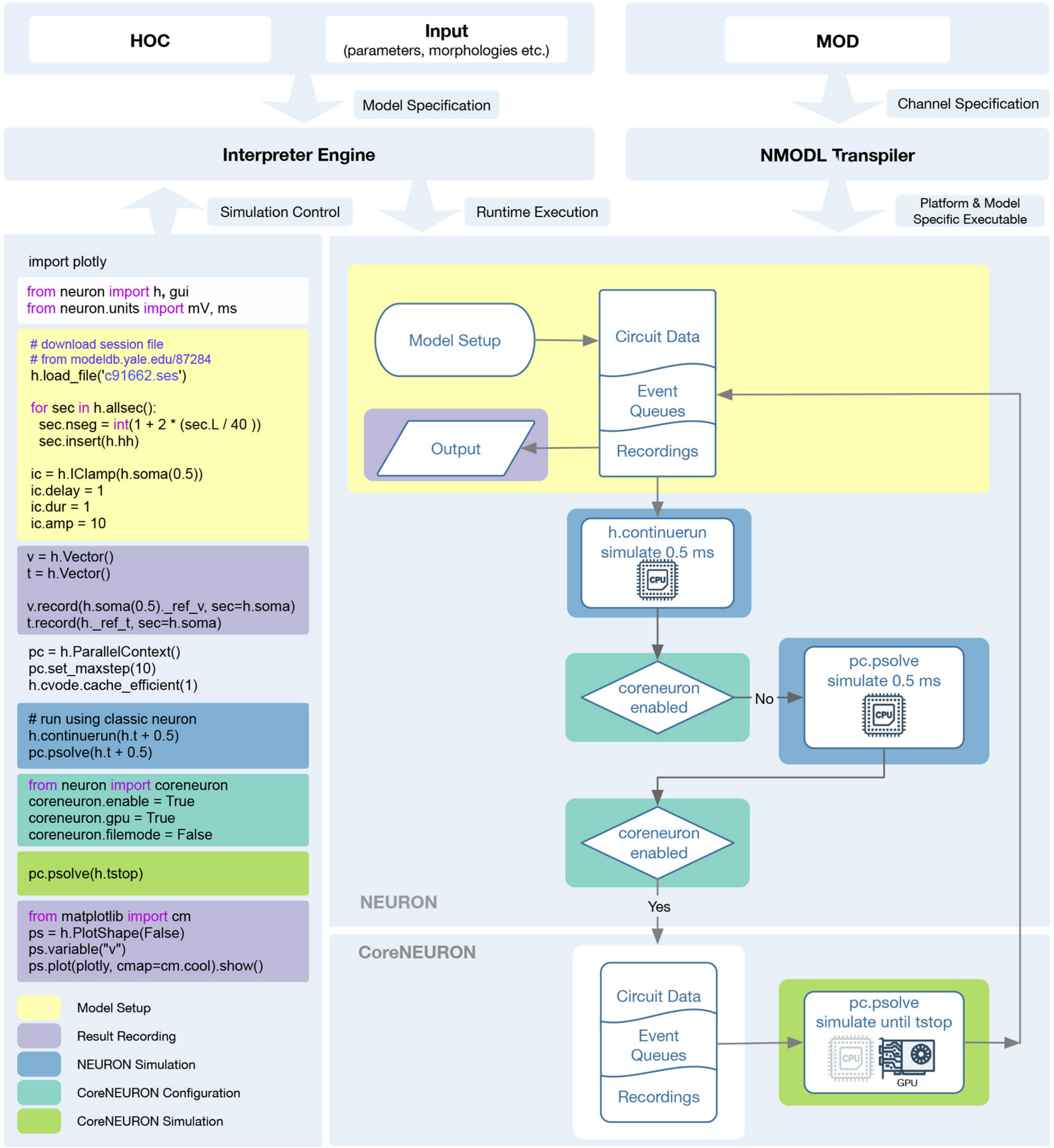
3.3. Performance Improvements Through Tighter Integration

As presented in Section 2.2, we have greatly improved the integration between NEURON, CoreNEURON and NMODL both on the level of the code organization and their ease of use at runtime. CoreNEURON and NMODL are now git submodules of the `nrn` repository, allowing us to build single software distribution packages containing optimized CPU and GPU support via CoreNEURON. This allows the user to transparently take advantage of modern hardware platforms such as GPUs, and recent hardware features such as AVX-512 on Intel CPUs. To this end, CoreNEURON's GPU implementation has been made production-ready, allowing easy offloading to NVIDIA GPUs. More importantly, the newly introduced in-memory transfer mode allows CoreNEURON simulations to be directly called from NEURON and the model state to be passed back and forth between NEURON and CoreNEURON. The workflow to utilize these new features is illustrated in **Figure 6**.

After constructing and initializing the model using NEURON's session file, simulation is first run in NEURON with `h.continuerun()` and `pc.psolve()`. GPU support via CoreNEURON is enabled using the newly introduced `coreneuron` Python module. Having CoreNEURON enabled will cause `pc.psolve` to use the CoreNEURON solver instead of the default NEURON solver. All necessary model state is automatically transferred to CoreNEURON before the simulation is continued using the CoreNEURON solver. At the end of the solver call, the model state is transferred back to NEURON, allowing user to save necessary recording results.

With this tighter integration, it is now possible to easily switch a large number of models, notably those using the fixed timestep

[6] Adapted from https://github.com/src-d/hercules.



```
import plotly
from neuron import h, gui
from neuron.units import mV, ms
# download session file
# from modeldb.yale.edu/87284
h.load_file('c91662.ses')

for sec in h.allsec():
  sec.nseg = int(1 + 2 * (sec.L / 40 ))
  sec.insert(h.hh)

ic = h.IClamp(h.soma(0.5))
ic.delay = 1
ic.dur = 1
ic.amp = 10
v = h.Vector()
t = h.Vector()

v.record(h.soma(0.5)._ref_v, sec=h.soma)
t.record(h._ref_t, sec=h.soma)
pc = h.ParallelContext()
pc.set_maxstep(10)
h.cvode.cache_efficient(1)
# run using classic neuron
h.continuerun(h.t + 0.5)
pc.psolve(h.t + 0.5)
from neuron import coreneuron
coreneuron.enable = True
coreneuron.gpu = True
coreneuron.filemode = False
pc.psolve(h.tstop)
from matplotlib import cm
ps = h.PlotShape(False)
ps.variable("v")
ps.plot(plotly, cmap=cm.cool).show()
```

FIGURE 6 | Diagram showing the NEURON and CoreNEURON execution workflow using the Python API: The Python code on the left demonstrates the CoreNEURON Python API usage and interoperability between NEURON and CoreNEURON solvers. The different shaded areas in the code correspond to the boxes of the same color on the right. First, the script sets up the model in NEURON and defines the entities to be recorded. In the following `h.continuerun()` statement the script starts by running the simulation in NEURON for 0.5 ms. Since the `coreneuron` and `gpu` options have not been enabled yet, the following call to `pc.psolve()` advances the simulation for 0.5 ms using NEURON. The call to `pc.psolve()` that is executed after enabling CoreNEURON, however, first copies the model to CoreNEURON using the direct-mode transfer and then runs the simulation until the prescribed `h.tstop` using CoreNEURON on GPUs. After finishing the CoreNEURON simulation step all variables and events are transferred back to NEURON.

TABLE 1 | Details of benchmarking systems: Blue Brain 5 (Phase 2) supercomputer and Google Cloud Platform.

BB5	CPU	2x20 core Intel Xeon Gold Cascade Lake 6248 @ 2.5 GHz
	GPU	NVIDIA V100 16GB
	Memory	768 GB DDR4 RAM
	CPU compiler	Intel C++ Compiler 19.1.2
	GPU compiler	NVHPC 21.2
	CUDA	11.0.2
	MPI	HPE MPI (SGI MPT) 2.25
	Python	3.8.3
	Network	InfiniBand 100G EDR
GCP	CPU	2x20 core Intel Xeon Cascade Lake
	GPU	NVIDIA V100 16GB
	Memory	320 GB DDR4 RAM
	CPU compiler	Intel C++ Compiler Classic 2022.0.2
	GPU compiler	NVHPC 21.2
	CUDA	11.0.2
	MPI	Intel MPI 2021.5.0
	Python	3.8.6

method, to use the optimized CoreNEURON solver. To quantify the performance benefits, in this section we showcase three different models that were ported to use CoreNEURON. We will compare the performance of NEURON and CoreNEURON running on CPU and GPU on the olfactory 3D bulb model 3.3.1, the rat CA1 hippocampus model 3.3.2 and the rodent motor (M1) cortical model 3.3.3.

The benchmarking system, Blue Brain 5 (Phase 2), with its hardware and software toolchains is summarized in **Table 1**. This system is based on an HPE SGI 8600 platform (HPE, 2022) and is housed at the Swiss National Supercomputing Center (CSCS). The GPU partition of the system has compute nodes with Intel Cascade Lake processors and NVIDIA Volta 100 GPUs. We used vendor-provided compiler toolchains and MPI libraries for the benchmarking. Unless otherwise specified, measurements were performed using two compute nodes, providing a total of 80 physical cores. We ran all CPU benchmarks in pure MPI mode by pinning one MPI rank per core. For GPU executions we reduced the number of MPI ranks to 16 (eight ranks per node) in order to achieve better utilization, enabled the CUDA Multi-Process Service (MPS) and used two or four NVIDIA V100 GPUs on each node. For the CPU measurements, NEURON and CoreNEURON were compiled using the Intel C++ compiler, while for GPU measurements CoreNEURON was compiled using the NVIDIA C++ compiler. All CoreNEURON benchmarks were both performed using the legacy MOD2C transpiler and the next-generation NMODL transpiler. All reported speedups were averaged over ten runs.

3.3.1. Accelerating 3D Olfactory Bulb Model Simulations via CoreNEURON

The olfactory bulb microcircuit developed by Migliore et al. (2014), serves as a model for studying the functional consequences of the laminar organization observed in cortical systems. The model was developed using realistic three-dimensional inputs, cell morphologies and network connectivity. The original model uses Python 2 and is available on ModelDB; we updated this model to use Python 3 and CoreNEURON to run our benchmarks. The updated version is publicly available from the GitHub repository of the Human Brain Project https://github.com/HumanBrainProject/olfactory-bulb-3d. The full model consists of 191,410 cells, 3,388,282 synapses and a total of 9,118,745 compartments. We simulated the default model configuration with a biological duration of 1050 ms and a timestep of 46.875 µs.

Figure 7 shows the performance difference between NEURON and CoreNEURON solvers when executing on CPU and GPU hardware. As can be observed in **Figure 7A**, the simulation using CoreNEURON on two full CPU nodes is 3.5× faster than the baseline NEURON benchmark run on the same hardware. The achieved acceleration is due to the use of SIMD instructions, which is enabled by the efficient internal data structures and the SoA (Structure of Array) memory layout used by CoreNEURON. When GPU offloading is enabled in CoreNEURON then with two GPUs per node the speedup increases to 21.4× compared with the baseline NEURON benchmark. Performance does not scale linearly when doubling the number of GPUs per node to four, and we see a maximum speedup of 30.4×. This is due to more time spent in the communication between the eight GPUs and the size of the model reaching the strong-scaling limit. In order to understand the performance differences between CoreNEURON executing on CPU and GPU **Figure 7B** shows a comparison of the two runtime profiles broken down into the most relevant execution regions. We have normalized the time of each region with the total execution time of the simulation. On the one hand, **Figure 7B** shows that the relative time spent in the most compute intensive parts such as the `Current calculation` and the `State update` is reduced significantly when executing on the GPU. The Hines `Matrix solver` does not currently benefit from GPU acceleration. This is due to data dependencies and limited parallelism inherent to the algorithm. On the other hand, we can see that the event delivery and CPU-GPU data transfer incur an additional cost compared to the CPU execution. Also, the spike exchange routines take a larger share in runtime on the GPU than on the CPU. This shows that the highly parallelizable compute operations are readily accelerated on the GPU while data movement and code with higher execution divergence are favored by the CPU.

3.3.2. Accelerating Rat CA1 Hippocampus Simulations Using GPUs

Another interesting example of a NEURON based simulation is the full-scale model of the rat hippocampus CA1 region built as part of the European Human Brain Project (manuscript in preparation). A recent draft of this model contains 456,378 neurons with 12 morphological types and 16 morpho-electrical types. The CA1 neurons in this model employ up to 11 active conductance classes, with up to 9 of those classes used in the dendrites. This model is used for running various large scale

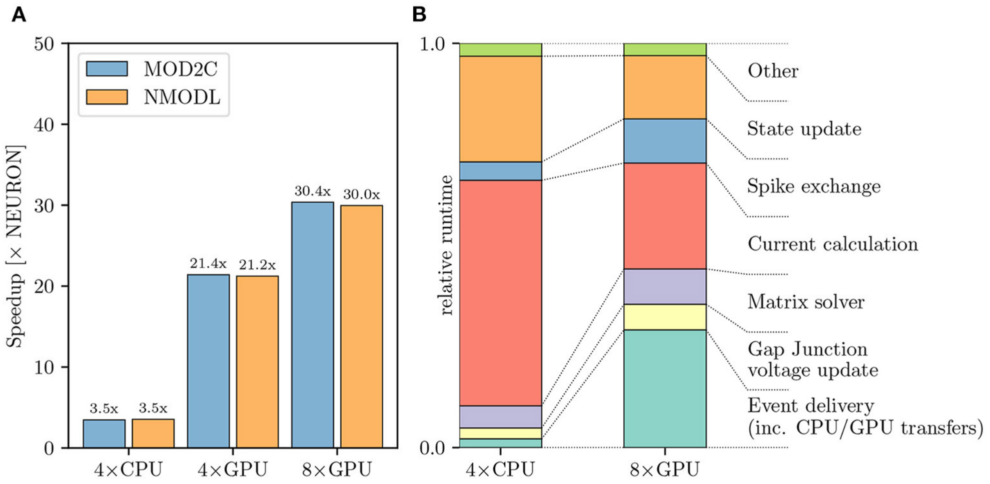


FIGURE 7 | Olfactory 3D bulb model performance comparison: **(A)** Improvement in the simulation time using CoreNEURON on CPU and GPU, with respect to NEURON running on CPU. We show the speedups using both MOD2C and NMODL transpilers. **(B)** Comparison of the two runtime profiles of the CoreNEURON run on CPU and GPU. The relative time (normalized to the total execution time) of the different execution regions in one timestep is shown. All benchmarks were run on two compute nodes with a total of 80 MPI ranks.

in-silico experiments on different European HPC systems. A reduced but representative version of this model is used for and available as part of the Hippocampus Microcircuit Massive Open Online Course[7] offered on edx.org.

Simulating the full rat CA1 hippocampus model is computationally expensive and requires large-scale HPC systems, due to its scale and biologically detailed nature. It represents, therefore, an ideal showcase for the improvements in hardware portability described above, most notably the GPU support provided by CoreNEURON. For the purpose of this showcase, we used aforementioned reduced model of the CA1 hippocampus containing 18,198 cells with 11,401,920 compartments and 107,237,482 synapses. Furthermore, we simulated a biological duration of 1000 ms with a timestep of 25 µs. Finally, the runtime configuration for the GPU benchmarks had to be adjusted so that 80 (instead of 16) MPI ranks were used. This was necessary due to a technical limit on the number of artificial cells of a given type that CoreNEURON can simulate in a single MPI rank.

Figure 8A shows that using CoreNEURON yields a performance improvement of $3-4\times$ compared with NEURON when executing on four Cascade Lake CPUs. When enabling GPU offloading in CoreNEURON it is possible to achieve up to 52× improvement compared with the NEURON baseline. It is worth mentioning that the new NMODL transpiler shows significant improvement in execution time over the legacy MOD2C transpiler (i.e., a speedup of 52× vs. 42× when using eight GPUs). This is due to NMODL's analytic solver generation, which is based on SymPy (Meurer et al., 2017) and the Eigen library (Guennebaud and Jacob, 2010). The performance improvement can be mainly attributed to the fact that `State update` kernels represent the evaluation of `DERIVATIVE` blocks in the `MOD` files, which contain ODEs that are now efficiently solved by NMODL. It is apparent that due to the computationally expensive nature of this model it scales linearly from four to eight GPUs and executing on GPUs provides a substantial benefit over the baseline NEURON benchmark. Analogous to **Figures 7B**, **8B** shows the relative breakdown of the total execution time for the most relevant regions. In contrast to the previous example, however, we see here that the current calculation and state update remain dominant on the GPU, while event delivery and CPU-GPU transfer play a smaller role. We interpret this result as an indication that this simulation is better suited for the strengths of the GPU hardware than the olfactory bulb model.

3.3.3. Simulating Large-Scale Cortical Models With NetPyNE

Here we report on the integration of NetPyNE with the CoreNEURON solver, thus taking full advantage of modern CPU/GPU architectures to reduce the simulation time. To illustrate this, we compared the performance of a large-scale and biologically realistic model of mouse motor (M1) cortical circuit on Google Cloud Platform (GCP) as well as Blue Brain 5. The mouse M1 model simulates a 300 µ m cylinder of cortical tissue including 10651 neurons, of three excitatory and four inhibitory types, and 38 million synapses, with 7000 spike generators (NetStims) simulating long-range inputs from seven thalamic and cortical regions. All model neuronal densities, classes, morphologies, biophysics, and connectivity were derived from experimental data. The model is being used to study neural coding, learning mechanisms and brain disease in motor cortex circuits (Sivagnanam et al., 2020; Dura-Bernal et al., 2022b).

GCP offers a wide variety of machine types tailored to different applications. For our benchmarks we allocated `n2-standard-80` and `nvidia-tesla-v100` nodes. The

[7] https://www.edx.org/course/simulating-a-hippocampus-microcircuit

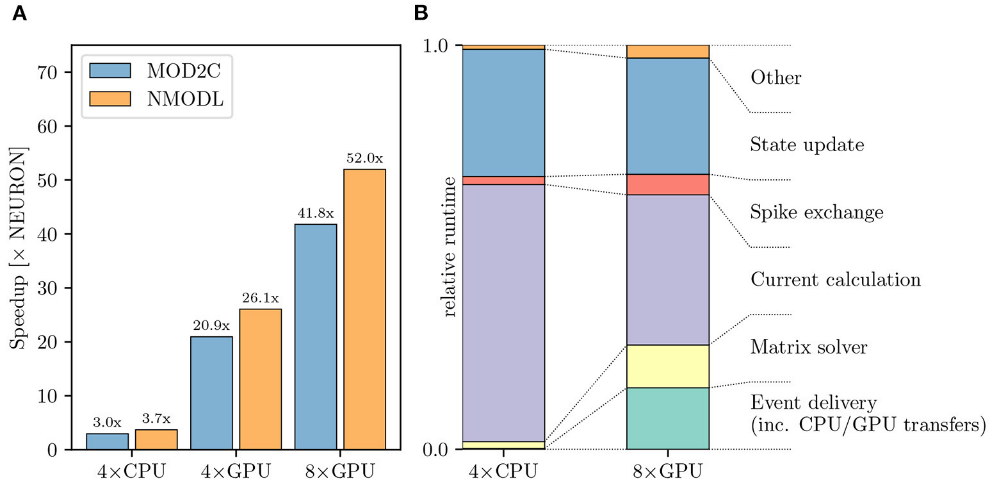


FIGURE 8 | Rat CA1 hippocampus model performance comparison: **(A)** Improvement in the simulation time using CoreNEURON on CPU and GPU, with respect to NEURON running on CPU. NMODL shows a significant improvement compared to the legacy MOD2C transpiler, which is due to analytic solver support in NMODL. **(B)** Comparison of the two runtime profiles of the CoreNEURON run on CPU and GPU. The relative time (normalized with the total execution time) of the different execution regions in one timestep. All benchmarks were run on two compute nodes with a total of 80 MPI ranks.

`n2-standard-80` are dual-socket Intel Cascade Lake based nodes while `nvidia-tesla-v100` compute nodes consist of dual-socket Cascade Lake based systems with 8 NVIDIA V100 GPUs. NEURON and CoreNEURON CPU measurements of the M1 NetPyNE benchmarks were run using 80 MPI ranks evenly distributed over two `n2-standard-80` nodes, while GPU runs were performed with 16 MPI ranks on one node pinning one MPI rank per core and distributing the ranks evenly between 4 and 8 GPUs. To allow for a fairer comparison between on-premise HPC hardware and cloud platforms we adjusted the benchmark configuration accordingly on Blue Brain 5 running with 16 MPI ranks for the GPU runs while maintaining the 80 ranks for the CPU runs. **Figure 9A** shows benchmark performance on Blue Brain 5. The CoreNEURON solver is 4× faster compared to NEURON when executing on CPU only. When GPU support is enabled then we achieve speedup of 26× and 39× with four and eight GPUs, respectively. Similarly to Section 3.3.1 the suboptimal scaling from four to eight GPUs suggests that the model's size and computational intensity do not fully saturate the allocated hardware and do not fully benefit from it. **Figure 9B** shows the performance improvements achieved on GCP. The measured speedups compared with the baseline NEURON CPU runs are 3.6×, 21×, and 30× for CoreNEURON on CPU, on four and eight GPUs, respectively. While the achieved speedups on GCP are slightly lower than on Blue Brain 5, they still show a clear performance advantage when using CoreNEURON with GPU support. Furthermore, this shows that one can get improved performance in cloud environments just as on traditional, on-premise cluster systems. Finally, we note that the use of NEURON Python wheels with their built-in GPU support drastically simplifies the efforts to setup the NEURON simulator toolchain with GPU support in cloud environments.

3.3.4. Improvements in RxD Performance

The developments described in Section 2.4 substantially improved the performance and scalability of reaction-diffusion simulations in NEURON both for 1D and 3D simulation. Using the same model code to simulate pure diffusion in a 1D test cylinder of length 200 μ m and diameter 2 μm, with 1,001 segments runs 4.2× faster in NEURON 8.0 than in NEURON 7.6.7. We note that "1D" is a slight misnomer here, as NEURON's version of 1D simulation uses the Hines algorithm (Hines, 1984) which provides $\mathcal{O}(n)$ scaling for implicit simulation and supports an arbitrary tree-like morphology where multiple 1D sections may connect at the same point. Likewise, a NEURON implementation of the circadian model of Leloup et al. (1999), a 10-species model with 21 reactions and no diffusion, ran 17.3× faster in NEURON 8.0 than in 7.6.7 (**Figure 10**). In the NEURON tutorial, we provide a version of this model that requires recent versions of NEURON due to its use of the `neuron.units` submodule to cleanly specify units appropriate for this model, however the ModelDB entry for this paper provides an equivalent representation that runs in both older and recent versions of NEURON.

Thanks to the improved voxelization method, the construction of voxel representations is now significantly faster. For example, the morphology (with voxel size 0.25 μ m in each dimension) of the soma and apical dendrite of a rat CA1 pyramidal neuron (Ascoli et al., 2007; Malik et al., 2016) within 70 μ m of the soma completed in 69 s in NEURON 8.0 and 349 s in NEURON 7.6.7, approximately a 5-fold speedup. The relative volume error (computed by comparing to the volume calculated as above vs. with voxel size 0.05 μ m in each dimension) was 0.15% in NEURON 8.0 vs. 16% in NEURON 7.6.7, a reduction made possibly by switching from including the full volume of boundary voxels to a subsampled fractional volume.

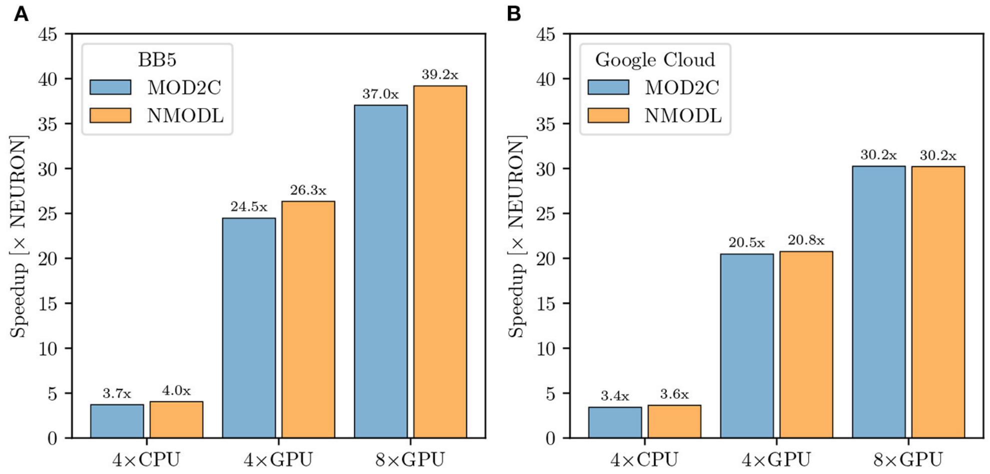


FIGURE 9 | M1 cortical model performance comparison: **(A)** Improvement in simulation time running CoreNEURON on CPU and GPU on the Blue Brain 5 supercomputer with respect to NEURON running on CPU. CPU benchmarks were run with 80 MPI ranks evenly distributed over two nodes, one rank per core. GPU benchmarks used 16 MPI ranks evenly distributed over two nodes. **(B)** Improvement in simulation time running CoreNEURON on CPU and GPU on Google Cloud Platform compute resources. The same configuration was used, except for GPU benchmarks being executed on one node.

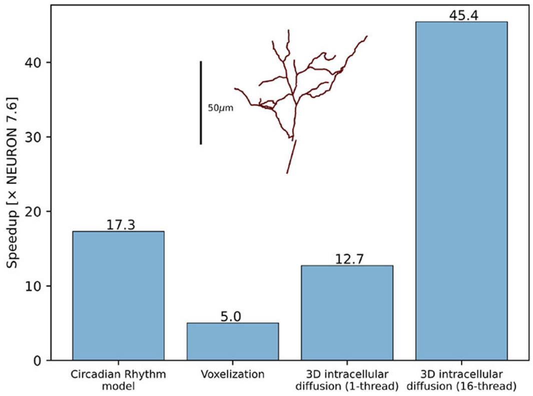


FIGURE 10 | RxD performance improvements for the Circadian Rhythm reaction model with just-in-time compilation, 3D morphology voxelization and intracellular diffusion using the DG-ADI method. The speedup was measured between NEURON 7.6.7 and NEURON 8.0. Inset shows the morphology used for voxelization and 3D diffusion.

3D simulation times are likewise reduced in NEURON 8.0, through the combined effects of eliminating Python from the simulation-time code, compilation of reaction specifications, and the replacement of the simulation algorithm (7.6.7 used SciPy's Virtanen et al., 2020 Biconjugate Gradient Stabilized iterative solver for matrix inversion; 8.0 uses a threaded Douglas-Gunn Alternating Direction Implicit method). Pure diffusion over 1 s of simulated time on the cell volume described above with initial conditions of 1 mM in a section of the apical dendrites and 0 mM elsewhere required 18,349 s of real time in NEURON 7.6.7, 1,442 s of real time in NEURON 8.0 with 1 thread (a 12.7× speedup), and only 426 s in NEURON 8.0 when using 16 threads (a 45.4× speedup). (We specify the initial conditions here as the iterative solver used in 7.6.7 would potentially require a different number of iterations depending on the initial conditions; the solver used in 8.0 performs the same calculations each time regardless of the concentrations).

4. DISCUSSION

4.1. Sustainability of the NEURON Simulator

Over the years, the scientific community has used NEURON to study a wide range of biophysical questions across various spatial and temporal scales, with over 2500 publications reporting such usage [8]. More than 750 published NEURON model source codes are available through SenseLab's ModelDB repository (McDougal et al., 2017), which provides curated metadata describing the biological assumptions. It is fair to say that for biophysically detailed models of neurons and networks, NEURON has become the de facto standard in the community. In light of this, it is quite obvious that continuity of the NEURON project is of great importance to the community.

However, for the majority of its more than 35 years of history, NEURON has been developed under what Gewaltig and Cannon (2014) dubbed a "heroic development model." While open source from its inception, until recently, the majority of contributions to the NEURON software came from its original author, Michael Hines, and a small group of collaborators from Duke and Yale University. We have now established a radically updated development model and life cycle of the NEURON software toward a modern and collaborative process, facilitating the contribution of modifications from other developers. In the language of Gewaltig and Cannon (2014) we were able to evolve

[8] https://www.neuron.yale.edu/neuron/publications/neuron-bibliography

NEURON's "heroic development model" to a "collaborative development model" with community engagement.

Sustainability of a software project, or its continued development, is of course not a goal in and of itself. A software with the history of NEURON should maintain as much backward compatibility with previously published models as possible, while catering for new use cases. In this respect, we demonstrated how it is possible to process the widely used NMODL language with a modern, compiler-based approach capable of producing highly optimized code, while keeping maximal backward compatibility.

4.2. From Desktop to Supercomputers

For first time users, NEURON is often introduced through interactive explorations on a desktop or laptop, combining small bits of Python with NEURON's built-in graphical user interface (GUI). This usage remains common in practice, with roughly one-third of NEURON models on ModelDB (McDougal et al., 2017) containing a `.ses` file generated from NEURON's GUI. Our improved continuous integration and build automation caters to this demand and produces pre-compiled binaries for the most common consumer computing architectures (Windows, macOS, and Linux), which allows users to easily install NEURON, including its graphical interface, on their personal systems.

With increasing use, it is common to complement or replace GUI-based use with the power and flexibility of scripting. This allows for easy handling of more diverse models and settings, parameter sweeps, or the evaluation of model variants to establish distributions and robustness of results. Over the last decade, Python has become the language of choice in the practice of computational science, and neurosimulations have been no different (Muller et al., 2015). Since this early integration, both neurosimulators and Python, have evolved substantially, as has their usage pattern. This requires that we express the functionality of NEURON in a Pythonic style and enable simple installation of platform-specific NEURON modules. Here, the adoption of modern Python wheels has dramatically simplified the use of NEURON and user-specific extensions in the Python environment.

As models and simulations become bigger, it is important that the simulator can optimally use the available hardware. In these cases it is particularly useful to be able to rely on NEURON binaries tailored to the computer architecture. This can be achieved through specialized pre-built NEURON binaries or through compiling NEURON with platform-specific parameters, including the support of the CPU's vector instructions, multiple threads or the use of GPUs (see also Section 4.4).

While there is already a wide range of computer architectures in use today, it is to be expected that the diversity will further increase. This general trend is driven by the challenges of further miniaturizing transistors and performance gains of future computer architectures in part will have to come from specialization (Hennessy and Patterson, 2019), already prominently visible in the field of machine learning (Reuther et al., 2019). Not all of these specializations will necessarily be useful for neurosimulations, but the better we understand the computational costs of our models in neuroscience (Cremonesi and Schürmann, 2020; Cremonesi et al., 2020), the more we may be interested in adopting some of those platforms for specific applications. Here, our work on translating the computationally intensive parts of a neuron model described in the NMODL language into source code that can be compiled for a wide range of computer hardware, in combination with the reduced memory footprint of CoreNEURON, is a major step forward to leverage these developments in the computer industry for neuroscience purposes.

With pre-compiled binaries for all major operating systems (Windows, macOS, Linux), Python scriptability, and built-in support for serial, threaded, MPI, and GPU accelerated calculations, NEURON can readily be used in many computing environments. The new NMODL framework furthermore can be extended to support future computer architectures without having to compromise performance on other platforms.

These efforts for efficiently using today's computer architectures, are complemented with NEURON's ability to run on large number of networked nodes, so-called clusters. In previously published work (Hines et al., 2011b), NEURON has run simulations with up to 128,000 processes catering even to the largest models. Nowadays, many universities maintain their own high performance computing environments for such purposes and a growing number of e-infrastructure providers offer high performance computing. For example, smaller numbers of computing resources are freely available for neuroscience simulation with NEURON and other simulators through a Neuroscience Gateway account (Sivagnanam et al., 2013). The EBRAINS research infrastructure provides large supercomputer allocations with preinstalled NEURON and even models for approved research projects.

4.3. NEURON as a Building Block for Scientific Workflows

Cloud-based Jupyter notebook providers have recently become another accessible way to use NEURON. EBRAINS, developed through the Human Brain Project, provides a Jupyter notebook cloud environment with NEURON and other simulator software pre-installed. Additionally, many public Jupyter servers, including Google Colab, allow installation of Python packages including NEURON via `pip`. Using NEURON through a cloud-based Jupyter server makes it accessible through any computing device with a modern browser, including phones and tablets, and facilitates sharing and collaborating on whole models and examples. To increase NEURON's usability in Jupyter notebooks, we have added built-in support for Python graphics (including via Plotly and Matplotlib) to NEURON's `ShapePlot` and `RangeVarPlot` classes, which provide, respectively, a 3D false-color view of the cell and a plot of state values along a path.

Building on top of the modular structure and APIs of NEURON, there are various community tools that have incorporated NEURON as a building block to develop higher level tools. For example, the Human Neocortical Neurosolver (Neymotin et al., 2020) provides its own graphical interface and launches NEURON in a separate process to simulate the underlying neocortical model. NetPyNE's online graphical interface launches virtual machines with NEURON on demand in the cloud and uses NEURON or CoreNEURON to run simulations. The example of the M1 model demonstrates that in this setting we observe performance increases when switching the simulator from NEURON to CoreNEURON (4× on a tightly coupled cluster, and 3.6× on a cloud instance). Also, we see good scaling when using multiple GPUs both for the tightly coupled cluster and the cloud; see the next section for a more detailed analysis. These results show that we can tap into the performance improvements of NEURON when invoking it from other tools. Thus, the same improvements in packaging and platform specific optimization that were demonstrated above will benefit also other tools, such as LFPy (Lindén et al., 2014) and BluePyOpt (Van Geit et al., 2016), which rely on NEURON's simulation capability.

4.4. Increased Performance for Tackling New Scientific Questions

In recent years, several large-scale and biophysically detailed models have been developed (Markram et al., 2015; Casali et al., 2019; Billeh et al., 2020; Hjorth et al., 2020; Borges et al., 2022; Dura-Bernal et al., 2022a,b). Some of these models are no longer only models of networks and their signal processing but are actually biophysical models of the tissue itself (Markram et al., 2015), capturing a diverse set of properties of the modeled brain region, also referred to as "digital twins." Such models have proved useful to bridge anatomy and physiology across multiple spatial and time scales (Reimann et al., 2017; Newton et al., 2021).

These types of models have become possible because of more quantitative data, new computational approaches to predicting and inferring missing parameters, and the increase in computer performance. In fact, a large number of improvements to the NEURON software over the last decade were motivated by these types of models: a major common theme in these developments was functionality to support parallel execution on multiple compute nodes (Migliore et al., 2006; Hines et al., 2008a,b, 2011a; Lytton et al., 2016), complemented by platform-specific optimizations (Ewart et al., 2015; Kumbhar et al., 2016). In particular, the platform-specific optimizations underwent a disturbing trend: optimizations of NEURON for the first vector computers had been discontinued in favor of memory optimizations for out-of-order CPUs with intricate cache hierarchies, only to return to SIMD analogous structure-of-array memory layouts for today's CPUs/GPUs with wide vector units. Previously, this led to special code versions with significant development efforts to adapt the code for different generations of hardware platforms and programming models.

Our work on CoreNEURON (Kumbhar et al., 2019) and NMODL (Kumbhar et al., 2020) has made it possible to contain the platform specific optimizations in the code generation framework, requiring less platform specific code in NEURON and allowing models to remain comparatively free of platform-specific details. Here, we introduced two recent advances. First, NMODL and CoreNEURON are now able to automatically generate code not only for CPUs but also GPUs. Second, the transparent integration with NEURON makes it possible to leverage this capability simply as an accelerator for simulations on a user's desktop, or to dramatically speed up large-scale simulations on supercomputers.

We compared running three different large scale models with NEURON and CoreNEURON in different hardware configurations. Compared to the baseline running with NEURON, transparently offloading to CoreNEURON achieves a four-fold speed-up on the same CPU hardware. This performance increase is mostly due to the better utilization of the vector units, a more cache efficient memory layout, and less data transfer between the CPU and main memory (Kumbhar et al., 2019). We furthermore demonstrate that it is now also possible to seamlessly make use of NVIDIA GPU hardware. We demonstrated a speed-up of 30×, 39× and 52× when using eight GPUs for the olfactory bulb model, the cortical M1 model and the hippocampus CA1 model, respectively, compared to four full CPUs. For the hippocampus CA1 model, we see ideal scaling up to 52× when doubling the GPU number. Both the thalamocortical M1 model and the olfactory bulb model show suboptimal scaling when moving from four to eight GPUs (39× and 30×, respectively), which we attribute to their lower computational cost compared with the hippocampus CA1 model, leading to lower utilization of the GPUs, and an overall lower compute-to-communicate ratio seen in the relatively longer time spent in event delivery and CPU-GPU data transfers.

These numbers should not be used for comparing the CPU's suitability for neurosimulations with that of a GPU. As the two architectures have wildly different characteristics in terms of total floating point performance and memory bandwidth, a deeper analysis is required to establish the efficiency of the simulations on the respective platforms (Lee et al., 2010). However, what can clearly be demonstrated with these numbers is that it is now possible for any user to readily make use of NVIDIA GPUs if they are installed.

4.5. Simulations in the Cloud

Low-cost virtual machines or dedicated servers are now also available from commercial providers billed by the second—typically referred to as "the cloud." Using NEURON in these environments requires that one can quickly configure NEURON there. In these environments, it is furthermore desirable to save intermediate results in a database to allow examining the results mid-calculation and to facilitate resuming in the event of timeouts and other issues, which can be readily done with database functionality integrated in Python (such as SQLite3). This is where our work on a straightforward `pip install` for NEURON is particularly useful, as more generally described in Sivagnanam et al. (2020) as a practical approach for both small sets of simulations and very large ones. Not only could we demonstrate the feasibility of running in the cloud using the Google Compute Cloud and NetPyNE with CoreNEURON as the compute backend, but we could also demonstrate that the achievable performance on a moderate set of nodes (moderate when compared to clusters and supercomputers) is on par with that of bare metal simulations on dedicated cluster. Using CoreNEURON showed a speedup that is comparable (3.6× vs. 4.0×) with the one achieved on a dedicated cluster. Using GPU offloading with CoreNEURON offered a 30× speedup compared with the baseline NEURON simulation. These results should be put in the context of simple on-demand access to such compute resources, NEURON's integration in an ecosystem of computational neuroscience, data processing and analysis tools.

4.6. Efficiently Integrating Subcellular and Extracellular Detail Into Neurosimulations

Some of the greatest health challenges of our time like stroke (the #2 cause of death worldwide) and dementia (#7 cause of death worldwide) are driven by the interaction of effects spanning from the sub-cellular level to neuron, network, and organism levels. These scales have often been addressed separately using different techniques with computational neuroscientists focusing on the neuron and network level while systems biologists study the protein interactions and systems-level questions, but this split has long been viewed as artificial and possibly problematic (see e.g., De Schutter, 2008). Exporting point dynamics to SBML for exploration with systems biology tools is an explicit bridge across this divide, but we believe that within NEURON the recent enhancements to NEURON's reaction-diffusion (RxD) support provide a conceptual bridge, supporting studies that were previously impractical. Faster simulations in both 1D and 3D are more than just a convenience for the modeler: they allow more detailed dynamics to be simulated in the same amount of time featuring a more complete representation of molecular interactions. They allow parameter sweeps at a higher resolution of detail, and they allow building more detailed training sets for machine-learning powered approximations of complex biophysics (e.g., Pham et al., 2021). The recently added ability to use NEURON's SaveState class with RxD models will facilitate running multiple experiments from a complex initial state and investigating steady state dependence on parameter values. New NEURON features like 3D extracellular simulation allow exploring how detailed cell models interact with each other through the extracellular environment and provide opportunities to include the effects of astrocytes (associated with multiple neurological diseases including Alzheimer's and Parkinson's), blood vessels, and other considerations historically under-studied by computational neuroscience.

4.7. Outlook

The updated development model, improved build system and enhanced software testing regime presented in this study provide the foundations for further modernization and re-engineering of NEURON. These improvements will enable more complex changes to be made while maintaining correctness and performance. The performance improvements reported here, coupled with the ease-of-use of the newly released Python wheels, show that CoreNEURON could soon become the default simulation engine for NEURON models, and that the improved integration between NEURON and CoreNEURON that we have presented is just the starting point. Further integration will require more migration of NEURON code to C++ and the development of modern data structures that can be exchanged between NEURON and CoreNEURON more easily and efficiently. The next-generation NMODL framework source-to-source compiler is able to parse the entire NMODL language and generate efficient and portable code for most existing `MOD` files. By further extending its code generation capabilities we will be able to replace the legacy NOCMODL and MOD2C DSL translators entirely. Ultimately this will allow the neuroscience community to use NEURON to simulate increasingly complex brain models in more accessible ways on systems ranging from desktops to supercomputers.

DATA AVAILABILITY STATEMENT

The NEURON simulator, with all the features and improvements described in this paper, is available as version 8.1 in the NEURON GitHub repository[9]. NetPyNE with CoreNEURON support is released as version 1.0.1 and available in the NetPyNE GitHub repository[10]. From the performance benchmarking studies in Section 3.3, the 3D Olfactory bulb model is available in the Human Brain Project GitHub repository[11], the Rat CA1 Hippocampus model is available as part of the Hippocampus Microcircuit Massive Open Online Course[12] offered on edx.org, and the M1 cortical circuit is available in the SUNY Downstate Medical Center GitHub repository[13]. All the benchmarking scripts, performance measurement data and figures are available in the NEURON GitHub repository[14].

AUTHOR CONTRIBUTIONS

NTC, WL, MH, and FS conceptualized and led the study. MH and PK led the overall software development on NEURON, CoreNEURON, and NMODL. OA, PK, NC, JK, OL, IM, FP, AS, and MH contributed to aspects of NEURON, CoreNEURON, and NMODL software development. OL and IM led GPU support integration and performance improvements. AS and NC led software engineering efforts including refactoring, CI and testing. FP implemented support for portable Python wheels for CPU and GPU platforms. IM and SD-B performed and validated the NetPyNE benchmarks. IM, OL, PK, and OA performed and validated the 3D olfactory bulb and Hippocampus benchmarks. RM and AN led and performed software development on RxD. OA, PK, and FS wrote the manuscript. NC, SD-B, JK, OL, IM, RM, AN, FP, AS, NTC, WL, and MH contributed to the manuscript. All authors gave feedback to the manuscript.

FUNDING

Research reported in this publication was supported by the National Institute for Neurological Disorders and Stroke, the National Institute for Mental Health, and the National Institute

[9] https://github.com/neuronsimulator/nrn
[10] https://github.com/suny-downstate-medical-center/netpyne
[11] https://github.com/HumanBrainProject/olfactory-bulb-3d
[12] https://www.edx.org/course/simulating-a-hippocampus-microcircuit
[13] https://github.com/suny-downstate-medical-center/M1_NEURON_paper/
[14] https://github.com/neuronsimulator/neuron_frontiers_2022_artifacts

of Biomedical Imaging and Bioengineering of the National Institutes of Health under award numbers R01NS011613, R01MH086638, and U24EB028998, National Science Foundation 1904444-1042C, New York State Spinal Cord Injury Research Board (SCIRB) DOH01-C32250GG-3450000, funding to the Blue Brain Project, a research center of the École Polytechnique Fédérale de Lausanne (EPFL), from the Swiss government's ETH Board of the Swiss Federal Institutes of Technology, the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement Nos. 785907 and 945539 (Human Brain Project SGA2 and SGA3).

ACKNOWLEDGMENTS

We thank Alessandro Cattabiani, Christos Kotsalos, and Tristan Carel for improving AST visitors and solver support in NMODL. We thank Jorge Blanco Alonso and Christos Kotsalos for improving the reports interface and GPU solver performance in CoreNEURON, respectively. We thank Evan Blasy, Lia Eggleston, and Cameron Conte for their respective contributions in SBML export, 3D voxelization, and 3D simulation functionalities in RxD. We thank Michele Migliore for providing the Olfactory bulb model.

REFERENCES

Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., et al. (2009). Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *J. Phys.* 180, 012037. doi: 10.1088/1742-6596/180/1/012037

Akar, N. A., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor–a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia), 274–282.

Amunts, K., Knoll, A. C., Lippert, T., Pennartz, C. M., Ryvlin, P., Destexhe, A., et al. (2019). The human brain project–synergy between neuroscience, computing, informatics, and brain-inspired technologies. *PLoS Biol.* 17, e3000344. doi: 10.1371/journal.pbio.3000344

Anwar, H., Caby, S., Dura-Bernal, S., D'Onofrio, D., Hasegan, D., Deible, M., et al. (2021). Training a spiking neuronal network model of visual-motor cortex to play a virtual racket-ball game using reinforcement learning. *bioRxiv*. doi: 10.1101/2021.07.29.454361

Ascoli, G. A., Donohue, D. E., and Halavi, M. (2007). NeuroMorpho.Org: a central resource for neuronal morphologies. *J. Neurosci.* 27, 9247–9251. doi: 10.1523/JNEUROSCI.2055-07.2007

Bartlett, R. A., Heroux, M. A., and Willenbring, J. M. (2012). "Overview of the TriBITS lifecycle model: a Lean/Agile software lifecycle model for research-based computational science and engineering software," in *2012 IEEE 8th International Conference on E-Science* (Chicago, IL: IEEE), 1–8.

Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., et al. (2019). "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (Denver, CO: IEEE), 71–81.

Billeh, Y. N., Cai, B., Gratiy, S. L., Dai, K., Iyer, R., Gouwens, N. W., et al. (2020). Systematic integration of structural and functional data into multi-scale models of mouse primary visual cortex. *Neuron* 106, 388.e18–403.e18. doi: 10.1016/j.neuron.2020.01.040

Blundell, I., Brette, R., Cleland, T. A., Close, T. G., Coca, D., Davison, A. P., et al. (2018). Code generation in computational neuroscience: a review of tools and techniques. *Front. Neuroinform.* 12, 68. doi: 10.3389/fninf.2018.00068

Borges, F., d,. S., Moreira, J. V., Takarabe, L. M., Lytton, W. W., and Dura-Bernal, S. (2022). Large-scale biophysically detailed model of somatosensory thalamocortical circuits in NetPyNE. *bioRxiv*. doi: 10.1101/2022.02.03.479029

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398. doi: 10.1007/s10827-007-0038-6

Bryson, A., Berkovic, S. F., Petrou, S., and Grayden, D. B. (2021). State transitions through inhibitory interneurons in a cortical network model. *PLoS Comput. Biol.* 17, e1009521. doi: 10.1371/journal.pcbi.1009521

Carter Edwards, H., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distribut. Comput.* 74, 3202–3216. doi: 10.1016/j.jpdc.2014.07.003

Casali, S., Marenzi, E., Medini, C., Casellato, C., and D'Angelo, E. (2019). Reconstruction and simulation of a scaffold model of the cerebellar network. *Front. Neuroinform.* 13, 37. doi: 10.3389/fninf.2019.00037

Cremonesi, F., Hager, G., Wellein, G., and Schürmann, F. (2020). Analytic performance modeling and analysis of detailed neuron simulations. *Int. J. High Perform. Comput. Appl.* 34, 428–449. doi: 10.1177/1094342020912528

Cremonesi, F., and Schürmann, F. (2020). Understanding computational costs of cellular-level brain tissue simulations through analytical performance models. *Neuroinformatics* 18, 407–428. doi: 10.1007/s12021-019-09451-w

Crouch, S., Hong, N. C., Hettrick, S., Jackson, M., Pawlik, A., Sufi, S., et al. (2013). The software sustainability institute: changing research software attitudes and practices. *Comput. Sci. Eng.* 15, 74–80. doi: 10.1109/MCSE.2013.133

De Schutter, E. (2008). Why are computational neuroscience and systems biology so separate? *PLoS Comput. Biol.* 4, e1000078. doi: 10.1371/journal.pcbi.1000078

de Verdière, G. C. (2020). *Recommendations of the "Extreme Data and Computing Initiative-2" Project, Assessment for Legacy Code and Software Modernisation*. Available online at: https://exdci.eu/sites/default/files/public/files/d4.5f.pdf (accessed September 14, 2021).

Douglas, J., and Gunn, J. E. (1964). A general formulation of alternating direction methods. *Numerische mathematik* 6, 428–453. doi: 10.1007/BF01386093

Dura-Bernal, S., Griffith, E. Y., Barczak, A., O'Connell, M. N., McGinnis, T., Schroeder, C. E., et al. (2022a). Data-driven multiscale model of macaque auditory thalamocortical circuits reproduces in vivo dynamics. *bioRxiv*. doi: 10.1101/2022.02.03.479036

Dura-Bernal, S., Neymotin, S. A., Suter, B. A., Dacre, J., Schiemann, J., Duguid, I., et al. (2022b). Multiscale model of primary motor cortex circuits reproduces in vivo cell type-specific dynamics associated with behavior. *bioRxiv*. doi: 10.1101/2022.02.03.479040

Dura-Bernal, S., Suter, B. A., Gleeson, P., Cantarelli, M., Quintana, A., Rodriguez, F., et al. (2019). NetPyNE, a tool for data-driven multiscale modeling of brain circuits. *Elife* 8, e44494. doi: 10.7554/eLife.44494

Einevoll, G. T., Destexhe, A., Diesmann, M., Grün, S., Jirsa, V., de Kamps, M., et al. (2019). The scientific case for brain simulations. *Neuron* 102, 735–744. doi: 10.1016/j.neuron.2019.03.027

Erdemir, A., Mulugeta, L., Ku, J. P., Drach, A., Horner, M., Morrison, T. M., et al. (2020). Credible practice of modeling and simulation in healthcare: ten rules from a multidisciplinary perspective. *J. Transl. Med.* 18, 369. doi: 10.1186/s12967-020-02540-4

Ewart, T., Yates, S., Cremonesi, F., Kumbhar, P., Schürmann, F., and Delalondre, F. (2015). "Performance evaluation of the IBM POWER8 architecture to support computational neuroscientific application using morphologically detailed neurons," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems - PMBS '15* (Austin, TX: ACM Press), 1–11.

Gewaltig, M., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430. doi: 10.4249/scholarpedia.1430

Gewaltig, M.-O., and Cannon, R. (2014). Current practice in software development for computational neuroscience and how to improve it. *PLoS Comput. Biol.* 10, e1003376. doi: 10.1371/journal.pcbi.1003376

Gleeson, P., Cantarelli, M., Marin, B., Quintana, A., Earnshaw, M., Sadeh, S., et al. (2019). Open source brain: a collaborative resource for visualizing, analyzing, simulating, and developing standardized models of neurons and circuits. *Neuron* 103, 395–411. doi: 10.1016/j.neuron.2019.05.019

Goodman, D. F. M. (2009). The brian simulator. *Front. Neurosci.* 3, 192–197. doi: 10.3389/neuro.01.026.2009

Gratiy, S. L., Billeh, Y. N., Dai, K., Mitelut, C., Feng, D., Gouwens, N. W., et al. (2018). Bionet: a python interface to neuron for modeling large-scale networks. *PLoS ONE* 13, e0201630. doi: 10.1371/journal.pone.0201630

Guennebaud, G., and Jacob, B.. (2010). *Eigen v3*. Available online at: http://eigen.tuxfamily.org

Hagen, E., Næss, S., Ness, T. V., and Einevoll, G. T. (2018). Multimodal modeling of neural network activity: computing lfp, ecog, eeg, and meg signals with lfpy 2.0. *Front. Neuroinform.* 12, 92. doi: 10.3389/fninf.2018.00092

Hennessy, J. L., and Patterson, D. A. (2017). *Computer Architecture, Sixth Edition: A Quantitative Approach, 6th Edn*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Hennessy, J. L., and Patterson, D. A. (2019). A new golden age for computer architecture. *Commun. ACM* 62, 48–60. doi: 10.1145/3282307

Heroux, M. A. (2015). Editorial: ACM TOMS replicated computational results initiative. *ACM Trans. Math. Softw.* 41, 1–5. doi: 10.1145/2743015

Hettrick, S., Antonioletti, M., Carr, L., Chue Hong, N., Crouch, S., De Roure, D., et al. (2014). Uk research software survey 2014. doi: 10.5281/zenodo.608046

Hines, M. (1984). Efficient computation of branched nerve equations. *Int. J. Biomed. Comput.* 15, 69–76. doi: 10.1016/0020-7101(84)90008-4

Hines, M., Davison, A., and Muller, E. (2009). NEURON and python. *Front. Neuroinform.* 3, 1. doi: 10.3389/neuro.11.001.2009

Hines, M., Kumar, S., and Schürmann, F. (2011a). Comparison of neuronal spike exchange methods on a Blue Gene/P supercomputer. *Front. Comput. Neurosci.* 5, 49. doi: 10.3389/fncom.2011.00049

Hines, M., Kumar, S., and Schürmann, F. (2011b). Comparison of neuronal spike exchange methods on a blue gene/p supercomputer. *Front. Comput. Neurosci.* 5, 49. doi: 10.3389/fncom.2011.00049

Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput.* 9, 1179–1209. doi: 10.1162/neco.1997.9.6.1179

Hines, M. L., Eichner, H., and Schürmann, F. (2008a). Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. *J. Comput. Neurosci.* 25, 203–210. doi: 10.1007/s10827-007-0073-3

Hines, M. L., Markram, H., and Schürmann, F. (2008b). Fully implicit parallel simulation of single neurons. *J. Comput. Neurosci.* 25, 439–448. doi: 10.1007/s10827-008-0087-5

Hjorth, J. J. J., Kozlov, A., Carannante, I., Frost Nylén, J., Lindroos, R., Johansson, Y., et al. (2020). The microcircuits of striatum in silico. *Proc. Natl. Acad. Sci. U.S.A.* 117, 9554–9565. doi: 10.1073/pnas.2000671117

HPE (2022). *Hpe sgi 8600 System*. Available online at: https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-a00025339en_us (accessed January 05, 2022).

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Front. Neuroinform.* 12, 2. doi: 10.3389/fninf.2018.00002

Keating, S. M., Waltemath, D., König, M., Zhang, F., Dräger, A., Chaouiya, C., et al. (2020). Sbml level 3: an extensible format for the exchange and reuse of biological models. *Mol. Syst. Biol.* 16, e9110. doi: 10.15252/msb.20199110

Kumbhar, P., Awile, O., Keegan, L., Alonso, J. B., King, J., Hines, M., et al. (2020). "An optimizing multi-platform source-to-source compiler framework for the NEURON MODeling language," in *Computational Science—ICCS 2020, Lecture Notes in Computer Science*, eds V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira (Cham: Springer International Publishing), 45–58.

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., et al. (2019). CoreNEURON : an optimized compute engine for the NEURON simulator. *Front. Neuroinform.* 13, 63. doi: 10.3389/fninf.2019.00063

Kumbhar, P., Hines, M., Ovcharenko, A., Mallon, D. A., King, J., Sainz, F., et al. (2016). "Leveraging a cluster-booster architecture for brain-scale simulations," in *High Performance Computing, Vol. 9697*, eds J. M. Kunkel, P. Balaji, and J. Dongarra (Cham: Springer International Publishing), 363–380.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). "Numba: a llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15* (New York, NY: Association for Computing Machinery).

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., et al. (2010). Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM Sigarch Comput. Arch. News* 38, 451–460. doi: 10.1145/1816038.1816021

Leloup, J.-C., Gonze, D., and Goldbeter, A. (1999). Limit cycle models for circadian rhythms based on transcriptional regulation in drosophila and neurospora. *J. Biol. Rhythms* 14, 433–448. doi: 10.1177/074873099129000948

Lindén, H., Hagen, E., Leski, S., Norheim, E. S., Pettersen, K. H., and Einevoll, G. T. (2014). Lfpy: a tool for biophysical simulation of extracellular potentials generated by detailed model neurons. *Front. Neuroinform.* 7, 41. doi: 10.3389/fninf.2013.00041

Lytton, W. W., Seidenstein, A. H., Dura-Bernal, S., McDougal, R. A., Schürmann, F., and Hines, M. L. (2016). Simulation neurotechnologies for advancing brain research: parallelizing large networks in NEURON. *Neural Comput.* 28, 2063–2090. doi: 10.1162/NECO_a_00876

Malik, R., Dougherty, K. A., Parikh, K., Byrne, C., and Johnston, D. (2016). Mapping the electrophysiological and morphological properties of ca 1 pyramidal neurons along the longitudinal hippocampal axis. *Hippocampus* 26, 341–361. doi: 10.1002/hipo.22526

Markram, H., Muller, E., Ramaswamy, S., Reimann, M., Abdellah, M., Sanchez, C., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

McDougal, R. A., Bulanova, A. S., and Lytton, W. W. (2016). Reproducibility in computational neuroscience models and simulations. *IEEE Trans. Biomed. Eng.* 63, 2021–2035. doi: 10.1109/TBME.2016.2539602

McDougal, R. A., Conte, C., Eggleston, L., Newton, A. J. H., and Galijasevic, H. (2022). Efficient simulation of 3D reaction-diffusion in models of neurons and networks. *Front. Neuroinform.* 16, 847108. doi: 10.3389/fninf.2022.847108

McDougal, R. A., Hines, M. L., and Lytton, W. W. (2013). Reaction-diffusion in the neuron simulator. *Front. Neuroinform.* 7, 28. doi: 10.3389/fninf.2013.00028

McDougal, R. A., Morse, T. M., Carnevale, T., Marenco, L., Wang, R., Migliore, M., et al. (2017). Twenty years of ModelDB and beyond: building essential modeling tools for the future of neuroscience. *J. Comput. Neurosci.* 42, 1–10. doi: 10.1007/s10827-016-0623-7

Medlock, L., Sekiguchi, K., Hong, S., Dura-Bernal, S., Lytton, W. W., and Prescott, S. A. (2022). Multi- scale computer model of the spinal dorsal horn reveals changes in network processing associated with chronic pain. *J. Neurosci.* 42, 3133–3149. doi: 10.1523/JNEUROSCI.1199-21.2022

Metzner, C., Mäki-Marttunen, T., Karni, G., McMahon-Cole, H., and Steuber, V. (2020). The effect of alterations of schizophrenia-associated genes on gamma band oscillations. *Schizophrenia* 8, 46. doi: 10.1101/2020.09.28.316737

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., et al. (2017). SymPy: symbolic computing in python. *PeerJ Comput. Sci.* 3, e103. doi: 10.7717/peerj-cs.103

Meyer, M. (2014). Continuous integration and Its tools. *IEEE Software* 31, 14–16. doi: 10.1109/MS.2014.58

Migliore, M., Cannia, C., Lytton, W. W., Markram, H., and Hines, M. L. (2006). Parallel network simulations with NEURON. *J. Comput. Neurosci.* 21, 119–129. doi: 10.1007/s10827-006-7949-5

Migliore, M., Cavarretta, F., Hines, M. L., and Shepherd, G. M. (2014). Distributed organization of a brain microcircuit analyzed by three-dimensional modeling: the olfactory bulb. *Front. Comput. Neurosci.* 8, 50. doi: 10.3389/fncom.2014.00050

Miller, G. (2006). A scientist's nightmare: software problem leads to five retractions. *Science* 314, 1856–1857. doi: 10.1126/science.314.5807.1856

Muller, E., Bednar, J. A., Diesmann, M., Gewaltig, M.-O., Hines, M., and Davison, A. P. (2015). Python in neuroscience. *Front. Neuroinform.* 9, 11. doi: 10.3389/fninf.2015.00011

Mulugeta, L., Drach, A., Erdemir, A., Hunt, C. A., Horner, M., Ku, J. P., et al. (2018). Credibility, replicability, and reproducibility in simulation for biomedicine and clinical applications in neuroscience. *Front. Neuroinform.* 12, 18. doi: 10.3389/fninf.2018.00018

Neely, J., de Supinski, B. R., and Still, C. H. (2017). Application modernization for the exascale era. *Comput. Sci. Eng.* 19, 6–8. doi: 10.1109/MCSE.2017.3421548

Newton, A. J., McDougal, R. A., Hines, M. L., and Lytton, W. W. (2018). Using neuron for reaction-diffusion modeling of extracellular dynamics. *Front. Neuroinform*. 12, 41. doi: 10.3389/fninf.2018.00041

Newton, T. H., Reimann, M. W., Abdellah, M., Chevtchenko, G., Muller, E. B., and Markram, H. (2021). In silico voltage-sensitive dye imaging reveals the emergent dynamics of cortical populations. *Nat. Commun*. 12, 3630. doi: 10.1038/s41467-021-23901-7

Neymotin, S. A., Daniels, D. S., Caldwell, B., McDougal, R. A., Carnevale, N. T., Jas, M., et al. (2020). Human neocortical neurosolver (hnn), a new software tool for interpreting the cellular and network origin of human meg/eeg data. *Elife* 9, e51214. doi: 10.7554/eLife.51214

Pham, D.-T. J., Yu, G. J., Bouteiller, J.-M. C., and Berger, T. W. (2021). Bridging hierarchies in multi-scale models of neural systems: look-up tables enable computationally efficient simulations of non-linear synaptic dynamics. *Front. Comput. Neurosci*. 88, 733155. doi: 10.3389/fncom.2021.733155

Pimentel, J. M., Moioli, R. C., de Araujo, M. F. P., Ranieri, C. M., Romero, R. A. F., Broz, F., et al. (2021). Neuro4PD: An initial neurorobotics model of parkinson's disease. *Front. Neurorobot*. 15, 640449. doi: 10.3389/fnbot.2021.640449

Pronold, J., Jordan, J., Wylie, B. J. N., Kitayama, I., Diesmann, M., and Kunkel, S. (2022). Routing brain traffic through the von neumann bottleneck: parallel sorting and refactoring. *Front. Neuroinform*. 15, 785068. doi: 10.3389/fninf.2021.785068

Ranieri, C. M., Pimentel, J. M., Romano, M. R., Elias, L. A., Romero, R. A. F., Lones, M. A., et al. (2021). A data-driven biophysical computational model of parkinson's disease based on marmoset monkeys. *IEEE Access* 9, 122548–122567. doi: 10.1109/ACCESS.2021.3108682

Reimann, M. W., Nolte, M., Scolamiero, M., Turner, K., Perin, R., Chindemi, G., et al. (2017). Cliques of neurons bound into cavities provide a missing link between structure and function. *Front. Comput. Neurosci*. 11, 48. doi: 10.3389/fncom.2017.00048

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., and Kepner, J. (2019). "Survey and benchmarking of machine learning accelerators," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)* (Waltham, MA: IEEE), 1–9.

Romaro, C., Najman, F. A., Lytton, W. W., Roque, A. C., and Dura-Bernal, S. (2021). NetPyNE implementation and scaling of the Potjans-Diesmann cortical microcircuit model. *Neural Comput*. 33, 1993–2032. doi: 10.1162/neco_a_01400

Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. (2011). "Parallel random numbers: as easy as 1, 2, 3," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11* (New York, NY: Association for Computing Machinery), 1–12.

Schirner, M., Domide, L., Perdikis, D., Triebkorn, P., Stefanovski, L., Pai, R., et al. (2022). Brain simulation as a cloud service: the virtual brain on ebrains. *Neuroimage* 251, 118973. doi: 10.1016/j.neuroimage.2022.118973

Sivagnanam, S., Gorman, W., Doherty, D., Neymotin, S. A., Fang, S., Hovhannisyan, H., et al. (2020). "Simulating large-scale models of brain neuronal circuits using google cloud platform," in *Practice and Experience in Advanced Research Computing, PEARC '20* (New York, NY: Association for Computing Machinery), 505–509.

Sivagnanam, S., Majumdar, A., Yoshimoto, K., Astakhov, V., Bandrowski, A. E., Martone, M. E., et al. (2013). "Introducing the neuroscience gateway," in *IWSG* (Zurich), 993.

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *Elife* 8, e47314. doi: 10.7554/eLife.47314.028

Tikidji-Hamburyan, R. A., Narayana, V., Bozkus, Z., and El-Ghazawi, T. A. (2017). Software for brain network simulations: a comparative study. *Front. Neuroinform*. 11, 46. doi: 10.3389/fninf.2017.00046

Van Geit, W., Gevaert, M., Chindemi, G., Rössert, C., Courcol, J.-D., Muller, E. B., et al. (2016). BluePyOpt: leveraging open source software and cloud infrastructure to optimise model parameters in neuroscience. *Front. Neuroinform*. 10, 17. doi: 10.3389/fninf.2016.00017

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., et al. (2020). Scipy 1.0: fundamental algorithms for scientific computing in python. *Nat. Methods* 17, 261–272. doi: 10.1038/s41592-019-0686-2

Volk, V. L., Hamilton, L. D., Hume, D. R., Shelburne, K. B., and Fitzpatrick, C. K. (2021). Integration of neural architecture within a finite element framework for improved neuromusculoskeletal modeling. *Sci. Rep*. 11, 22983. doi: 10.1038/s41598-021-02298-9

Willenbring, J. M. (2015). Replicated computational results (RCR) report for "BLIS: a framework for rapidly instantiating BLAS functionality". *ACM Trans. Math. Softw*. 41, 1–4. doi: 10.1145/2738033

Wolfe, M. (2021). Performant, portable, and productive parallel programming with standard languages. *Comput. Sci. Eng*. 23, 39–45. doi: 10.1109/MCSE.2021.3097167

Author Disclaimer: The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Neuroinformatics

ORIGINAL RESEARCH
published: 29 June 2022
doi: 10.3389/fninf.2022.884033

A System-on-Chip Based Hybrid Neuromorphic Compute Node Architecture for Reproducible Hyper-Real-Time Simulations of Spiking Neural Networks

Guido Trensch[1,2]* *and Abigail Morrison*[1,2,3]

[1] Simulation and Data Laboratory Neuroscience, Jülich Supercomputing Centre, Institute for Advanced Simulation, Jülich Research Centre, Jülich, Germany, [2] Department of Computer Science 3—Software Engineering, RWTH Aachen University, Aachen, Germany, [3] Institute of Neuroscience and Medicine (INM-6), Institute for Advanced Simulation (IAS-6), JARA-Institute Brain Structure-Function Relationship (JBI-1/INM-10), Research Centre Jülich, Jülich, Germany

OPEN ACCESS

Edited by:
James Courtney Knight,
University of Sussex, United Kingdom

Reviewed by:
Runchun Mark Wang,
Western Sydney University, Australia
Bernhard Vogginger,
Technical University Dresden, Germany

****Correspondence:***
Guido Trensch
g.trensch@fz-juelich.de

Received: *25 February 2022*
Accepted: *23 May 2022*
Published: *29 June 2022*

Citation:
Trensch G and Morrison A (2022) A System-on-Chip Based Hybrid Neuromorphic Compute Node Architecture for Reproducible Hyper-Real-Time Simulations of Spiking Neural Networks. Front. Neuroinform. 16:884033. doi: 10.3389/fninf.2022.884033

Despite the great strides neuroscience has made in recent decades, the underlying principles of brain function remain largely unknown. Advancing the field strongly depends on the ability to study large-scale neural networks and perform complex simulations. In this context, simulations in hyper-real-time are of high interest, as they would enable both comprehensive parameter scans and the study of slow processes, such as learning and long-term memory. Not even the fastest supercomputer available today is able to meet the challenge of accurate and reproducible simulation with hyper-real acceleration. The development of novel neuromorphic computer architectures holds out promise, but the high costs and long development cycles for application-specific hardware solutions makes it difficult to keep pace with the rapid developments in neuroscience. However, advances in System-on-Chip (SoC) device technology and tools are now providing interesting new design possibilities for application-specific implementations. Here, we present a novel hybrid software-hardware architecture approach for a neuromorphic compute node intended to work in a multi-node cluster configuration. The node design builds on the Xilinx Zynq-7000 SoC device architecture that combines a powerful programmable logic gate array (FPGA) and a dual-core ARM Cortex-A9 processor extension on a single chip. Our proposed architecture makes use of both and takes advantage of their tight coupling. We show that available SoC device technology can be used to build smaller neuromorphic computing clusters that enable hyper-real-time simulation of networks consisting of tens of thousands of neurons, and are thus capable of meeting the high demands for modeling and simulation in neuroscience.

Keywords: neuromorphic computing, compute node, FPGA, SoC, spiking neural networks, simulation, performance, parallel computing

1. INTRODUCTION

In the process of gaining insight into the underlying principles of neural computation, the tools and methods developed and provided by computational neuroscience play a key role. In particular, we rely on the mathematical modeling of neuron, synapse, and neural network models and their numerical simulation to study their complex interaction and network dynamics. Community software for modeling, such as NeuroML (Gleeson et al., 2010), NMODL (Hines and Carnevale, 2000), and NESTML (Plotnikov et al., 2016), and for simulation, such as NEURON (Hines and Carnevale, 1997), Arbor (Akar et al., 2019), NEST (Gewaltig and Diesmann, 2007), and Brian (Goodman and Brette, 2008) provide such tools. They are complemented by numerical tools for statistical analysis, such as the Electrophysiology Analysis Toolkit *Elephant*[1] as well as tool support for model validation methodologies, for example, the validation framework *NetworkUnit*[2] (Gutzen et al., 2018). The requirements in regard to efficiency, correctness, and replicability and reproducibility of the outcomes place high demands on the whole software ecosystem.

When investigating large scale networks, in general one would like to simulate them as fast as possible. Whereas, real-time simulation is interesting because of the possibility of interacting with real-world applications, hyper-real-time would enable the study of slow processes, such as structural plasticity and long-term memory, and permit researchers to perform more comprehensive parameter scans of faster processes. This is still a major technical challenge (Friedmann et al., 2017), and not even the fastest supercomputer available today is up to the task.

Consequently, neuromorphic computing and application-specific novel hardware architectures are very attractive as they promise significant acceleration. However, the technical hurdles to making neuromorphic computing a useful tool for neuroscientists are not insignificant either. Crucially, flexibility and efficiency, which are both required for such a system, are opposing goals in the choice of technology (e.g., GPP[3], FPGA[4] or ASIC[5]; Noll et al., 2010). Optimal flexibility is achieved with traditional general purpose processors. The SpiNNaker system (Furber et al., 2013) is an example for a neuromorphic massively parallel computing platform that is based on digital multi-core chips using ARM processing cores. It is fully programmable, thus flexible in the choice and implementation of the numerical models, and allows large-scale simulations to be performed in real-time. The Heidelberg BrainScaleS system (Schemmel et al., 2010) and its successor BrainScales-2 (Pehle et al., 2022), in contrast, are capable of running simulations orders of magnitude faster than real-time. To achieve this, the architecture builds on the physical, i.e., analog, emulation of neuron and synapse models (Schemmel et al., 2017) in dedicated mixed-signal circuits combined with digital plasticity processors (BrainsScaleS-2) using a "hybrid plasticity" scheme (Friedmann et al., 2017). Physical, analog emulation thereby restricts the system to its built-in, "silicon-frozen" analog models, and use-cases where technology-related effects, such as fabrication tolerances and thermal noise, are acceptable.

During recent years, programmable device technology and tools have greatly increased in functionality, benefiting from the continued advances in semiconductor technology. Modern field programmable gate arrays (FPGAs) provide a large number of chip resources (e.g., logic cells and memories) allowing to implement complex hardware designs at affordable costs. High-level synthesis (HLS) tools allow the developer to generate hardware implementations from algorithmic descriptions, thus reducing development time and making the technology accessible to non hardware experts. Although the design effort remains high, programmable device technology offers a good compromise between flexibility and efficiency and has therefore been widely recognized as potentially well-suited to neural network simulation. This has been exploited by a number of digital neuromorphic architectures developed in recent years.

In an earlier study, Maguire et al. (2007) made an inventory and revealed the challenges associated with implementing large-scale spiking neural networks on FPGAs, emphasizing the importance of design decisions on system level and its impact on the final performance. Since then, a number of architectural approaches and implementations for different use cases have been published. A scalable modular architecture for closed-loop experiments with *in vitro* cultures is presented in Pani et al. (2017). The platform is able to simulate small-to-medium size networks in real-time, implementing 1,440 Izhikevich neurons. *Bluehive* (Moore et al., 2012)—a scalable custom 64-FPGA machine—is dedicated to the simulation of large-scale networks with demanding communication requirements. On a single FPGA, *Bluehive* can simulate 64,000 Izhikevich neurons in real-time. *NeuroFlow* (Cheung et al., 2016) is a platform that builds on top of Maxeler's[6] Dataflow Engine (DFE) technology. A 6-FPGA system can simulate a network of 600,000 neurons. Real-time performance is achieved when simulating a network consisting of 400,000 neurons. The simulation of a plastic 1,000 neuron two-population Izhikevich model for 24 h biological time can be completed in 1,435 s, thus achieving a ~60-fold acceleration. The platform supports several neuron and synapse model types and a spike time dependent plasticity (STDP) rule. *NeuroFlow* also provides a PyNN interface (Davison et al., 2009)—a common Python interface for neural network simulators. In Wang et al. (2014) and Wang et al. (2018), an architecture is proposed that uses a procedural "on-the-fly" generation scheme for parameters and connections and is able to simulate 20 million to 2.6 billion leaky integrate and fire (IAF) neurons in real-time on a single Stratix V FPGA.

Such large scales come at a price and can only be achieved by accepting limitations regarding functionality, model complexity and simulation accuracy. These limitations may well represent acceptable trade-offs for the intended specific use cases, but can

[1] RRID:SCR_003833; http://neuralensemble.org/elephant.

[2] RRID:SCR_016543; https://github.com/INM-6/NetworkUnit.

[3] General Purpose Processor.

[4] Field Programmable Gate Array.

[5] Application-Specific Integrated Circuit.

[6] Maxeler Technologies: www.maxeler.com.

be severe with respect to the requirements of a platform for general neuroscience simulations. For example, in order to save hardware resources and reduce both computational costs and the amount of data to be processed, hardware implementations often use a large update interval of $h = 1$ ms to progress neuron model dynamics (e.g., Moore et al., 2012; Cheung et al., 2016; Wang et al., 2018). This is 10 times larger than the *de facto* standard used in digital simulations, and comes at the cost of numerical accuracy, especially for neuron models with stiff equations (Hansel et al., 1998; Morrison et al., 2007; Blundell et al., 2018b; Pauli et al., 2018). A further commonly-used trade-off with similar advantages and disadvantages is to represent neuron state variables in a low-precision fixed-point data format (e.g., Moore et al., 2012; Wang et al., 2018). It has been shown, for example, that the accuracy of the numerical integration of the Izhikevich neuron model dynamics is insufficient when a s16.15 representation, i.e., a 32-bit signed fixed-point data format is used (Gutzen et al., 2018; Trensch et al., 2018). Model complexity is reduced in the architecture proposed in Wang et al. (2014) and Wang et al. (2018) where individual synaptic connection delays are replaced by an axonal delay, thus avoiding the large memory structures and computational costs required to delay and accumulate incoming spike events.

These examples clearly demonstrate that it is challenging to reach design decisions that are simultaneously performant and flexible. The plethora of neuron and synapse models makes it difficult to come to design decisions that satisfy all requirements equally. There are also many questions relevant for the design which still lack an unambiguous answer and thus keep design decisions in a state of uncertainty. One example is the required numerical precision, which determines the specification of data types and the implementation of arithmetic operations—a design decision that effects implementation complexity, chip area and power efficiency. So far, only a few studies have examined the effects of numerical accuracy on simulation outcomes (e.g., Pfeil et al., 2012; Trensch et al., 2018; Dasbach et al., 2021).

Promising new design possibilities are also enabled by the integration on a single chip of FPGAs together with processor cores and other components to System-on-Chip (SoC) devices. This paves the way toward novel hybrid software-hardware approaches for application-specific implementations and new neuromorphic computing systems, such as the IBM Neural Computer INC-3000; a highly scalable parallel processing system. A single-cage system clusters 432 Xilinx Zynq SoC devices in a high bandwidth 3D mesh communication network (Narayanan et al., 2020). The system is highly flexible and applications can off-load algorithms and accelerate them using the programmable logic of the Zynq SoC devices. An example of such an application is the implementation of the cortical microcircuit model (Potjans and Diesmann, 2014) on the INC-3000 presented in Heittmann et al. (2022)—a reproduction of an equivalent NEST implementation and on the SpiNNaker neuromorphic system (cf. van Albada et al., 2018). The model consists of $0.8 \cdot 10^5$ neurons and $0.3 \cdot 10^9$ synaptic connections, was implemented in HLS, and utilizes 305 FPGAs. The simulation achieves an approx. four times speed-up compared with the biological time domain.

In this article, we introduce a novel SoC-based hybrid software and hardware mixed architecture approach for a neuromorphic compute node (henceforth *HNC node*) which is intended to work in a multi-node cluster configuration and capable of meeting the high demands for modeling and simulation in neuroscience. The development builds on the Xilinx Zynq-7000 SoC device architecture (Xilinx, 2021) and takes advantage of the tight coupling of a powerful FPGA device and a dual-core ARM Cortex-A9 processor core. The primary goal of the development is to provide a flexible platform for the accelerated simulation of neural network models which may consist of up to a few tens of thousands of neurons, a scale which covers the vast majority of current spiking neural network modeling studies. With the neuroscience requirement-driven design of the HNC node architecture, our development is to be seen as a complementary yet distinct approach to the neuromorphic developments aiming at brain-inspired and highly efficient novel computer architectures for solving real-world tasks.

We show that such a system can indeed be built, and that acceleration factors with respect to real-time in the order of 10–50 are realistically achievable for moderate workloads, with even higher factors possible for low workloads. We further demonstrate that the use of workload and performance models allow us to predict the performance characteristics of such a system under varying assumptions regarding workload and hardware design choices, some of which showing great potential as a substrate for neural simulations.

This article is organized as follows. Section 2 first gives an overview of the HNC node high-level architecture and the main design ideas. Section 3 presents the results of our performance measurements and an evaluation of the performance characteristics. A detailed presentation of the HNC node hardware and software architecture can be found in Section 4, with a focus on microarchitecture details critical to performance. In Section 5, we develop a workload and performance model to understand the performance characteristics of the HNC node and predict them for alternative assumptions in design space.

2. OVERVIEW OF THE HYBRID NEUROMORPHIC COMPUTE (HNC) NODE

The HNC node architecture concept combines software-based and hardware-based implementations for the building blocks of a neural network simulation engine, and tightly couples both implementation types on a single chip; specifically, on a device of the Xilinx Zynq-7000 SoC family (Xilinx, 2021).

The underlying algorithms and the functional principle of the HNC node concept do not differ from those that are typically used in pure software implementations for time-discrete neural network simulations of point neuron models. It follows a hybrid strategy where neuron states are updated synchronously, time-driven, and at fixed intervals (e.g., $\Delta t = 0.1$ ms) and synapses are updated asynchronously and event-driven, triggered when a synapse's presynaptic neuron emits a spike (Morrison et al., 2005).

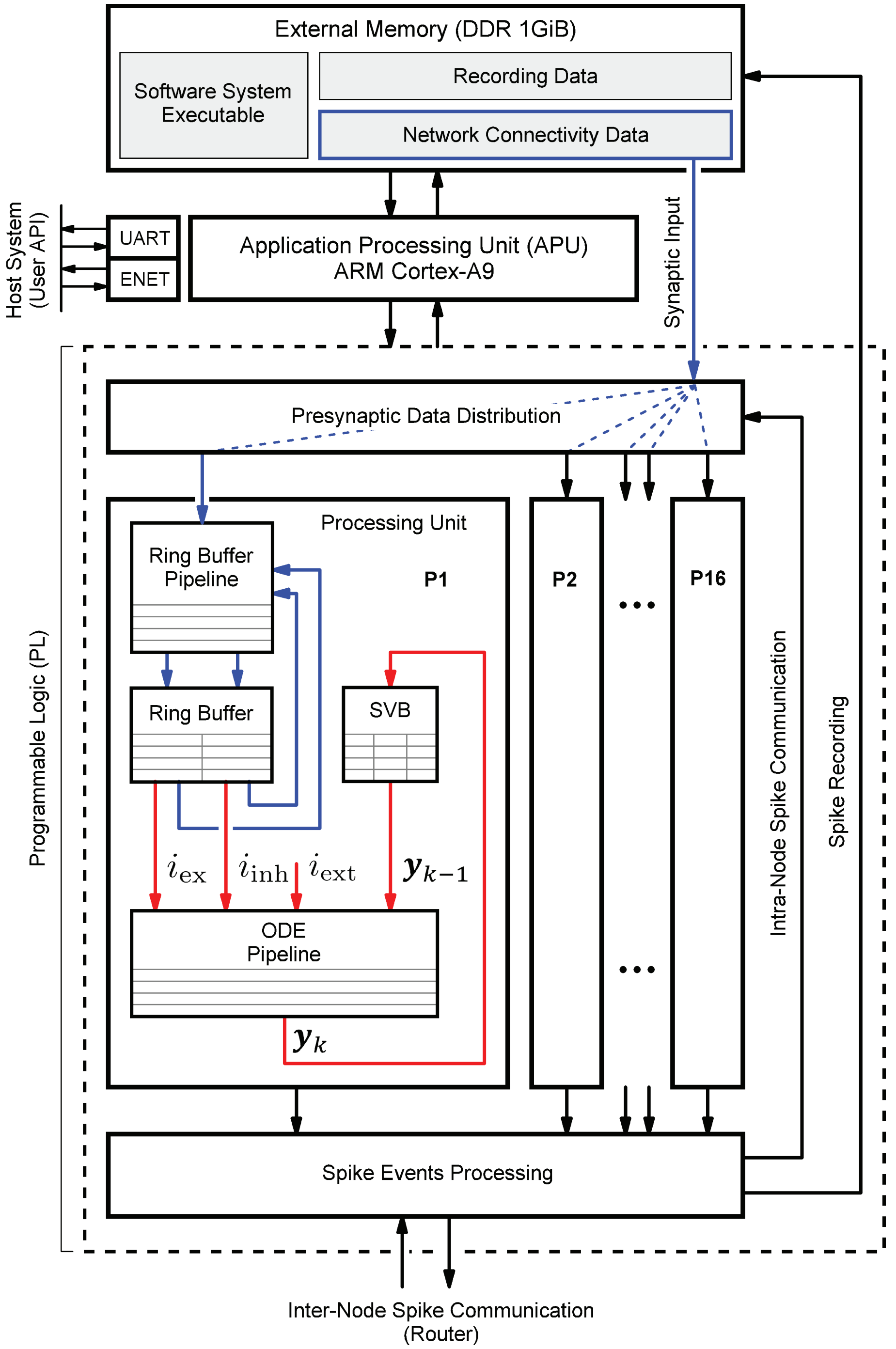


FIGURE 1 | Hybrid neuromorphic compute (HNC) node high-level architecture. The highest architectural level of the HNC node comprises three main components: an off-chip external memory (top), an application processing unit (APU; middle), and a programmable logic part (PL; lower dashed box). In order to distribute the workload and parallelize operations, the PL implements 16 identical processing units (P1, P2,.., P16). The red and blue arrows indicate two distinct processes that are critical to performance and primarily determine the performance characteristics and achievable acceleration factors. Red arrows: the process of the neuron and synapse model state update performed by the ordinary differential equations solver pipelines (ODE pipelines) which operate on fast on-chip block RAM memories that constitute the state variables buffer (SVBs). Blue arrows: the process of the presynaptic data distribution and processing which hold the data it operates on in the slow external off-chip memory.

While it is sufficient to implement performance non-critical tasks in software and let them be executed by general purpose processors, the performance-critical algorithms profit from mapping them to hardware. Non-critical tasks are, for example, the processes of node configuration, operation and simulation control, data type conversion, network instantiation, and user interaction. Critical to performance and simulation efficiency are the spike events processing and presynaptic data distribution, and the neuron and synapse model computations. The algorithms implemented in hardware bring the data and the operations performed on them close together and can thus alleviate problems which are inevitable with conventional systems, such as the von Neumann bottleneck.

Figure 1 shows the HNC node high-level architecture concept, which consists of three main components: (i) an off-chip external memory (top); (ii) an application processing unit (APU; middle); and (iii) a programmable logic part (PL; dashed box). A more detailed description of the high-level system architecture and the microarchitecture is given in Section 4.1.

Both the APU and the PL are connected to the off-chip external memory. It contains the node control software (Section 4.2) which is executed by the APU orchestrating the overall node operation, and also holds the node-local connectivity data of the neural network being simulated and buffers the recorded spike data. Storing the connectivity data in a slow, external memory is one of the decisive performance limiting factors of the system. This aspect is discussed in detail Section 4.3.1. However, there are two important factors leading to this design decision. The first is a functional requirement: even though the current development does not yet include plasticity, in order to be able to cope with synaptic and structural plasticity algorithms in future, the synaptic connections must be stored, accessible, and changeable. In contrast, for static networks, performance-efficient solutions have been developed which makes use of a procedural connectivity generation approach (Knight and Nowotny, 2021; Heittmann et al., 2022) where the synaptic connections are determined algorithmically during the simulation, thus avoiding having to retrieve them from memory. The second is a resource constraint due to technical limitations of the technology: fast, low-latency, on-chip block RAM (BRAM) would be ideal to hold this data, but BRAM is a limited FPGA resource and the memory requirement for storing a network's connectivity data is demanding. For example, given a 64-bit data item to represent a single connection, a natural dense network, such as the cortical microcircuit model (Potjans and Diesmann, 2014) comprising $0.8 \cdot 10^5$ neurons and $0.3 \cdot 10^9$ connections requires 2.4 Gbyte of memory in total. That is 24 Mbyte per compute node if a single node processes 10^3 neurons. The Xilinx Zynq-7000 SoC device used in this work provides only 19.2 Mbit of BRAM, i.e., 10 times less than required.

The PL, i.e., the FPGA part of the SoC, implements 16 identical hardware processing units (P1, P2,.., P16). Each is capable of carrying out the computations for $N^P = 64$ neurons. This allows a total of $N = 1,024$ neurons to be processed on a single chip or HNC node, respectively. The PL and APU are closely coupled through high performance streaming and memory mapped interfaces which allow an efficient data exchange between the two parts. The PL is also directly connected to the off-chip external memory, thus enabling APU-independent memory read and write operations.

Each processing unit processes its 64 neurons in a pipeline fashion, updating the neuron states at fixed intervals of $\Delta t = 0.1$ ms. The neuron states $\boldsymbol{y}_k$ are thereby held in state vector buffers (SVB) which are implemented as fast block RAM (BRAM) memories on the PL. The associated data paths of this time-driven process are indicated in **Figure 1** by the red arrows.

The blue arrows in **Figure 1** mark the data paths involved in the event-driven presynaptic data processing. The post-synaptic spike events (up to 16 spike events can occur in parallel at a time; one per processing unit) are serialized and packed for communication and recording. This is handled by the spike events processing module. If a spike event occurs, it initiates read operations from external memory to obtain the network's connectivity data, i.e., the node-local synaptic connections of the firing neuron, from which the synaptic inputs are derived. The presynaptic data distribution module parallelizes this data and delivers the synaptic inputs to the processing units (P1, P2,.., P16); this is indicated by the dashed blue lines in **Figure 1**, thereby distributing the workload generated by the incoming presynaptic spike events. The ring buffers (RB) implement the synaptic transmission delays and store the accumulated synaptic inputs, i.e., the lumped excitatory i_{ex} and inhibitory i_{inh} values. Since the number of synapses by far predominates, the whole process of presynaptic data distribution and processing is critical to performance.

3. RESULTS

3.1. Single Node Performance

In the following, we consider an isolated HNC node that is not embedded in a multi-node system for which otherwise inter-node communication and synchronization latencies cannot be ignored. For an isolated node, the previously explained two distinct processes will exclusively determine performance where the neuron state update process (red arrows in **Figure 1**), and the process of presynaptic data distribution and processing (blue arrows in **Figure 1**), contribute to different performance relevant aspects. In Section 5.2, a performance model is presented that is based on the HNC node microarchitecture implementation details explained in Section 4.3. By additionally taking communication latencies, inevitably occurring in a multi-node system, into account, the model will also allow conclusions to be drawn about the acceleration factors achievable for larger network sizes and workloads.

The current HNC node design implements $N^{\text{M}} = 1024$ neurons and allows $C^{\text{M}} = 128$ target connections per source neuron and node. This is in agreement with a connection probability value of approx. $\epsilon = 0.1$ observed in Braitenberg and Schüz (1998). Note that the possible number of a source neuron's target connections is not restricted to the value of C^{M}. It scales linearly with the number of HNC nodes M in a cluster, i.e., it yields MC^{M}. A typical cortical neuron connects to between $1,000$ and $10,000$ other neurons. Consequently, a network of $N = 10^5$ where each neuron has 10^4 connections represents an upper limit with regard to memory requirements and workload; beyond this, the total number of synapses in a network scales linearly rather than quadratically.

In order to evaluate the HNC node's capability to perform in different workload situations, we investigate a two-population

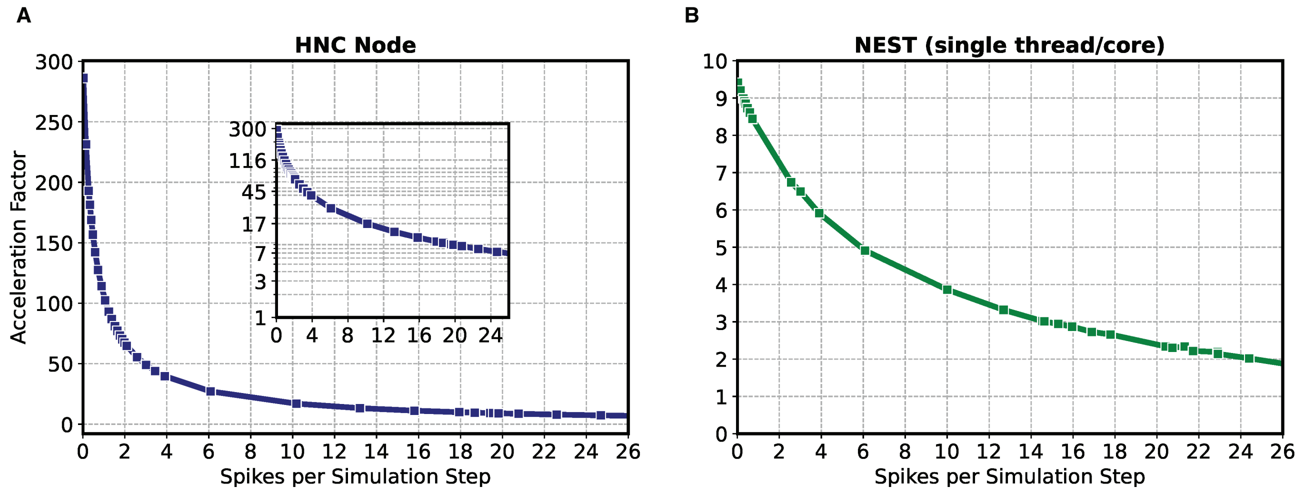


FIGURE 2 | Performance as a function of workload for the HNC node and NEST. The acceleration factor (wall clock time divided by the biological time) as a function of the average number of spike events per simulation time step $\bar{\nu}_k$ of the HNC node using a PL clock frequency of $f_{clk} = 200$ MHz **(A)** and the neural simulation tool NEST on an Intel(R) Core(TM) i7-7700K CPU 4.20 GHz (Kaby Lake architecture) **(B)**. The measurements were carried out with $h =$ 0.1 ms simulation resolution. In consecutive simulation runs of 5 min simulated biological time, the 1,000 neuron two-population Izhikevich neural network model described in Section 5.3 was stimulated with an increasing external offset current $i_{ext} = \{-3.0$ pA, .., $+100$ pA$\}$. Inset in **(A)** gives a log-lin representation.

network model consisting of 1,000 neurons (see Section 5.3). We measure the time to simulate the network and calculate the acceleration factor as the quotient of the measured simulation duration in wall clock time and the simulated biological time of 300 s. We systematically vary the external input current from $i_{ext} = -3.0$ pA to 100.0 pA. The increasing external offset current causes the network to run through a wide range of activity, from quiescence up to an average firing rate of $\bar{\nu} = 300$ spks/s and thus an increase in the workload. According to the workload model described in Sec. 5.1, this results in an average number of spike events per simulation time step ($h =$ 0.1 ms) ranging from $\bar{\nu}_k = 0$ to $\bar{\nu}_k = 30$.

The result of the HNC node performance measurement is shown in **Figure 2A**. For comparison, **Figure 2B** shows the results for the same model implemented in NEST 2.20.1 (Fardet et al., 2020) and executed on an Intel(R) Core(TM) i7-7700K CPU 4.20 GHz (Kaby Lake architecture). If the workload is in the range of a few spike events per simulation time step, the HNC node outperforms the NEST simulation on the Intel Kaby Lake CPU, and this even at ~4.5 W power consumption (see the power report given in the **Supplementary Material**)—with the Intel Kaby Lake CPU, a power consumption of several tens of Watts is to be expected. If the external current is set to zero, the network fires with an average rate of $\bar{\nu} = 7.0$ spks/s, which corresponds to a number of spikes per time step of $\bar{\nu}_k = 0.7$. For this workload, the acceleration factor achieved for the NEST simulation is 8.4 compared to a factor of 127.0 measured for the HNC node. The NEST simulator used for the comparison is a runtime-optimized and flexible tool for a wide range of neural network simulations and as such, is a good reference in this regard. Clearly, a CPU-optimized implementation of the specific network model can achieve even better results[7]. However, the difference in performance and efficiency is such that the HNC node performance is beyond the reach of any CPU implementation. At low workloads, the hardware implementation can fully utilize its capabilities. Pipelining and the parallelization of operations increases throughput and reduce latencies. This is mainly to be ascribed to the process of the neuron state update, indicated by the red arrows in **Figure 1**. Its implementation benefits from data-locality that is achieved by storing variables in fast, low-latency on-chip BRAM memories.

As the workload increases, the NEST implementation undergoes a moderate degradation in performance. In contrast, the performance deteriorates rapidly on the HNC node. This is a trivial consequence of the data access latency and limited bandwidth of the external memory decelerating the process of the pre-synaptic data distribution and processing (marked by the blue arrows in **Figure 1**), which now dominates operation. This is examined in greater detail below for different hardware design choices. Moreover, the measurements of the single HNC node and CPU core performances only give an upper baseline. For the simulation of larger networks on multi-node or many-core systems in the following we examine the effect on performance of the additional latencies arising from synchronization and communication.

3.2. Performance Characteristics

Based on the HNC node microarchitecture (Section 4.3) and their operating latencies (Section 4.3.3) a performance model

[7] A C-implementation of the network model is provided on GitHub: https://github.com/gtrensch/RigorousNeuralNetworkSimulations.

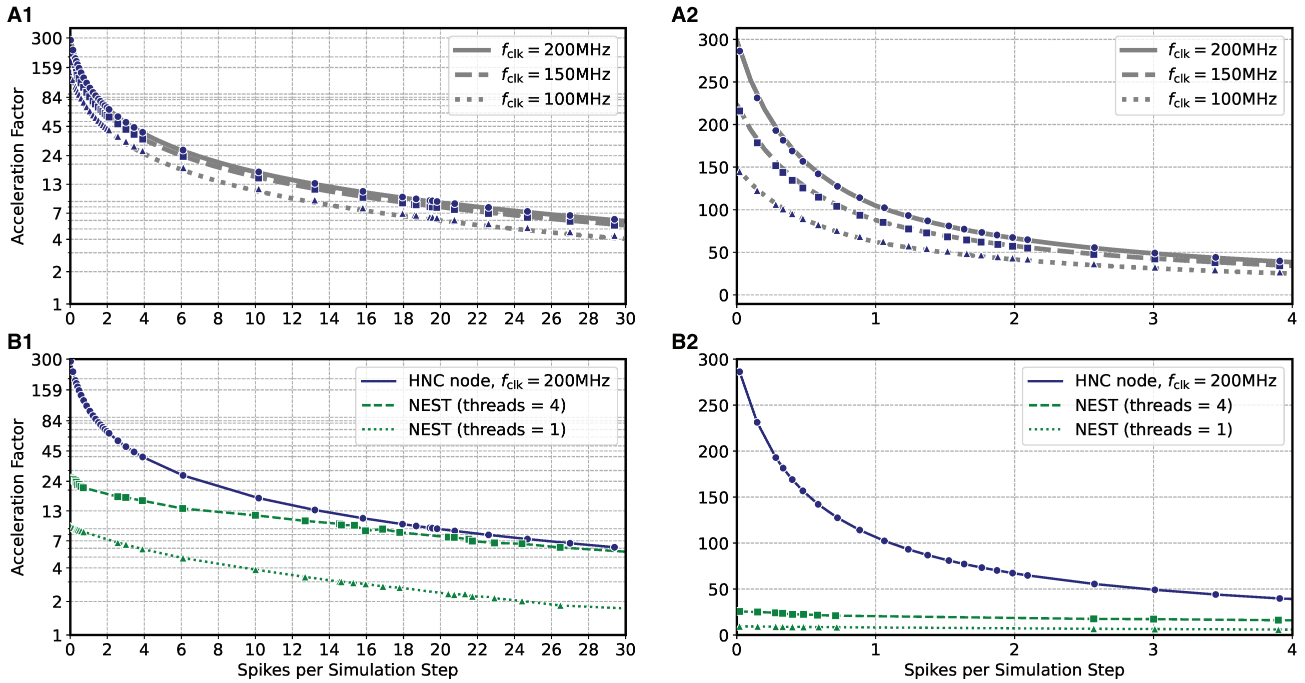


FIGURE 3 | Performance as a function of PL clock frequency and workload for the HNC node and NEST. **(A)** Measured acceleration factors of the HNC node (blue markers) as a function of workload for three different clock frequencies in log-lin **(A1)** and linear **(A2)** representation. Gray curves show the predictions of the performance model (Section 5.2). **(B)** As in **(A)**, but comparing the performance of the HNC node running at a PL clock frequency of $f_{clk} = 200$ MHz to that of a NEST implementation using one or four threads on an Intel(R) Core(TM) i7-7700K CPU 4.20 GHz.

is developed in Section 5.2. This model is used in the following to evaluate the performance characteristics of the HNC node as a stand-alone compute node and when operating in a cluster configuration. In order to verify the correctness of the performance model, we repeated the measurements carried out in the previous section using three different PL clock frequencies $f_{\text{clk}} = 100/150/200$ MHz. The results are shown in **Figures 3A1,A2** where the blue markers indicate the measured acceleration factors and the gray curves are calculated from Equation (7) of the performance model. The predicted acceleration factors are in almost perfect agreement with the measured values.

The results also reveal that as the workload increases, the achievable acceleration factor is increasingly determined—and thus limited—by external memory access times (i.e., by the term $\bar{\nu}_k L_{\text{DS}}$ in Equation 7). This can not be compensated by a higher PL clock frequency. However, an acceleration factor of ~100 is achieved for moderate workloads, i.e, $\bar{\nu}_{\text{k}} \approx 1$, $h = 0.1$ ms. Such a workload is created, for example, by a network consisting of $N = 5,000$ neurons with an average firing rate of $\bar{\nu} \approx 2$ spks/s.

Figures 3B1,B2 compares the HNC node measurements at a PL clock frequency of $f_{\text{clk}} = 200$ MHz to the equivalent simulation in NEST on a four-core Intel CPU. At low workloads, the HNC node is an order of magnitude faster than the NEST/CPU implementation. Even at high workloads, the HNC node still simulates substantially faster than a single state-of-the-art processor core. Such high workloads are not only of theoretical interest in benchmarking tasks. As $\bar{\nu}_{\text{k}}$ increases linearly with the network size N (Section 5.1), from a single-node workload perspective and assuming a fixed number of neurons per node, a small network at high average firing rates is equivalent to a large network utilizing multiple nodes and exhibiting a low average firing rate.

For example, for the cortical microcircuit model (Potjans and Diesmann, 2014) which consists of $N \approx 0.8 \cdot 10^5$ neurons, a value of $\bar{\nu}_{\text{k}} \approx 23$ can be expected[8]. At this workload the HNC node achieves an acceleration factor of ~7 while for a single-threaded NEST simulation a factor of ~2 was measured. If the NEST workload is distributed, in the sense of strong-scaling utilizing all four cores of the Intel CPU, the NEST simulation is nearly as fast as the HNC node. Note that $\bar{\nu}_{\text{k}} \approx 23$ is a theoretical value in this case, as the current single node implementation cannot accommodate a network as large as the cortical microcircuit model.

Even though power efficiency was not considered in this work, it is worth mentioning that the SoC device's power consumption is in the order of just a few Watts, and thus achieves a much higher simulation efficiency than the Intel core.

If the HNC node is to be operated in a cluster, the adverse effect that additional inter-node communication has on performance could influence design decisions such as the number

[8] A value of $\bar{\nu}_{\text{k}} = 23.24$ spikes per simulation time step was determined experimentally from a 15 minutes NEST simulation using the implementation by van Albada et al. (2018).

TABLE 1 | Parameter sets.

Parameters	Prototype	High data stream parallelism	High proc. units parallelism	Low proc. units parallelism
Number of data streams, *DS*	2	16	16	16
Data stream latency, L_{DS}	110	14	14	14
Number processing units, *P*	16	16	32	8
Number of neurons per processing unit, N^{P}	64	64	32	128
ODE pipeline iteration latency, IL_{N}	64	64	32	128
Acceleration factors w/o communication				
Maximum, $F_{\mathrm{S}}^{\mathrm{MAX}} = F_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}} = 0)$	298.5	298.5	571.4	152.7
Low workload, $F_{\mathrm{S}}(1.0)$	104.7	177.0	246.9	113.0
Medium workload, $F_{\mathrm{S}}(10.0)$	16.9	84.5	97.7	66.5
High workload, $F_{\mathrm{S}}(20.0)$	8.8	52.4	58.4	45.6
Acceleration factors with communication				
Maximum, $F_{\mathrm{C}}^{\mathrm{MAX}} = F_{\mathrm{C}}(0)$	119.8	119.8	148.1	86.6
Low workload, $F_{\mathrm{C}}(1.0)$	67.6	91.7	107.5	70.9
Medium workload, $F_{\mathrm{C}}(10.0)$	15.0	51.7	56.4	44.4
High workload, $F_{\mathrm{C}}(20.0)$	8.1	34.8	36.9	31.3

The acceleration factors are calculated using the performance model (Section 5.2) for four different parameter sets prototype; high data stream parallelism; high processing units parallelism; low processing units parallelism, and for three different workload situations, low, medium, and high as well as with and without inter-node communication. The number of neurons per node $N^{\mathrm{M}} = PN^{\mathrm{P}} = 1024$*, the PL clock frequency* $f_{\mathrm{clk}} = 200$ MHz*, the transmission latency time* $T_{\mathrm{COM}} = 500$ ns*, and the per spike event transmission latency factor* $\alpha = 0.05$ *(see main text and Section 5.2 for description) are the same for all parameter sets.*

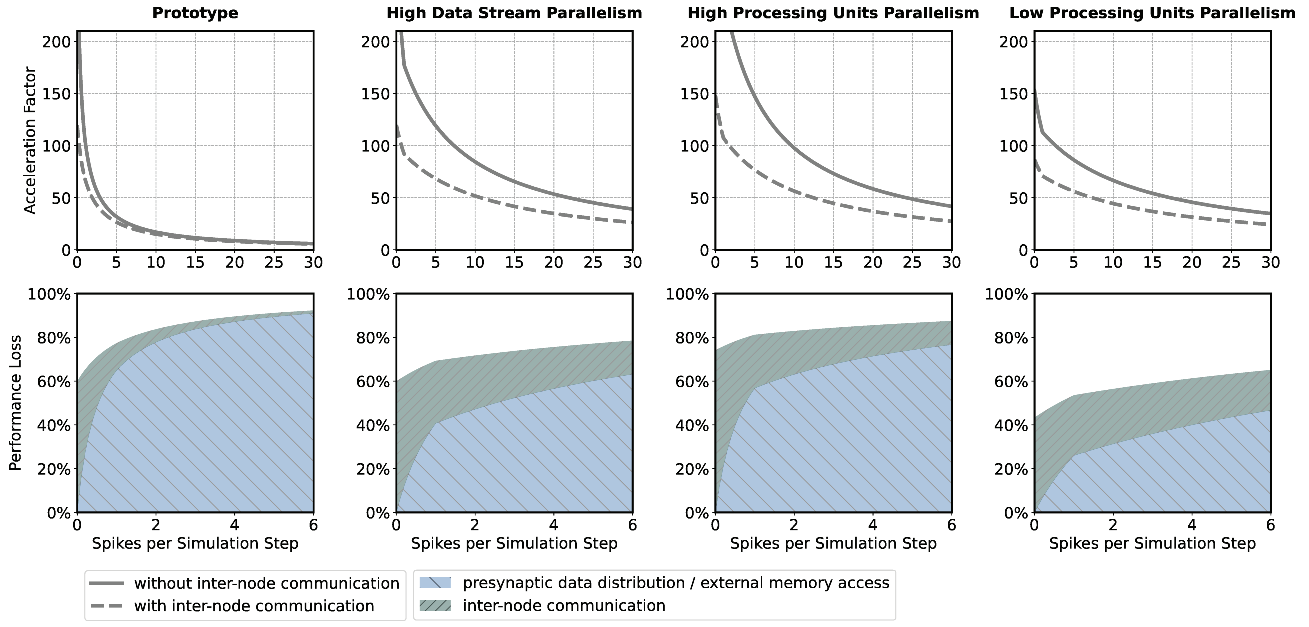


FIGURE 4 | Performance characteristics estimation. Performance characteristics of the HNC node are calculated using the performance model (Section 5.2) for the parameter sets *prototype*; *high data stream parallelism*; *high processing units parallelism*; *low processing units parallelism*. See main text and **Table 1** for details. The upper panels show the achievable acceleration factors as a function of workload with inter-node communication $F_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}})$ (dashed curves) and without inter-node communication $F_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}})$ (solid curves); the lower panels show the stacked plots of the respective contributions to the loss of performance with respect to the maximum achievable single-node acceleration factor F_{S}^{MAX} of the inter-node communication $P_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}})$ (green) and presynaptic data distribution $P_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}})$ (blue) (see Section 5.2).

of neurons per node and processing unit. For illustration, we consider four sets of design parameters. These are as follows: **prototype**—the parameter set corresponding to the prototypical implementation generating the measurements presented above, implementing $P = 16$ processing units, $DS = 2$ data streams (marked S1 and S2 in **Figures 8**, **9**), and $N^{\mathrm{P}} = 64$ neurons per ODE pipeline; **high data stream parallelism**—as for *prototype* but assuming that each processing unit connects to its own data stream ($P = 16$, $DS = 16$, $N^{\mathrm{P}} = 64$) introducing a factor eight times reduction of external memory access latency; **high processing units parallelism**—as for *high data stream parallelism* but implementing twice the number of processing units in order to halve the ODE pipeline iteration latency and increase the maximum achievable single-node acceleration factor ($P = 32$, $DS = 16$, $N^{\mathrm{P}} = 32$); and **low processing units parallelism**—the opposite of *high processing units parallelism*, reducing the number of processing units ($P = 8$, $DS = 16$, $N^{\mathrm{P}} = 128$). The parameter sets are shown in **Table 1**. Note that the parameter sets, with the exception of the *prototype* configuration, have not been applied to the HNC node. The SoC device selected for this study is limited to the *prototype* configuration in terms of number of data streams.

The number of neurons per node ($N^{\mathrm{M}} = 1024$) and the PL clock frequency ($f_{\mathrm{clk}} = 200$ MHz) are kept constant across the parameter sets. To describe the effect of inter-node communication on performance, the performance model developed in Section 5.2 introduces two parameters: the transmission latency time T_{COM}, and the per spike event transmission latency factor α (for a description of the parameters see Section 5.2). Their values were set to $T_{\mathrm{COM}} = 500$ ns and $\alpha = 0.05$. They are the same for all parameter sets. The choice for the transmission latency time is motivated by the temporal resolution of $h = 0.1$ ms and an envisioned acceleration factor of 100, which would be a major breakthrough for reproducible large-scale neuroscience simulations. This assigns T_{COM} half of the wall clock time that would be available to complete a single simulation step. The value of the per spike event transmission latency factor was arbitrarily chosen and corresponds to 5 additional clock cycles per spike event at a given PL clock frequency of $f_{\mathrm{clk}} = 200$ MHz.

The upper panels in **Figure 4** show the acceleration factors as a function of the workload calculated according to the performance model (Section 5.2, Equations 7, 8) both with and without inter-node communication. In addition, the lower panels in **Figure 4** provide an alternative representation of the curves, namely as the respective proportion of performance loss (with respect to the maximum achievable single-node acceleration factor for the corresponding parameter configuration) caused by inter-node communication and by the process of the presynaptic data distribution—mainly the effect of external memory access latency (Section 5.2, Equations 11, 12). **Table 1** shows the calculated acceleration factors for low, medium, and high workload.

As one would expect, the additional communication latency reduces the maximum achievable acceleration factors. For the *prototype* configuration (**Figure 4**, prototype, upper panel), for example, the factor decreases from 298.5 to 119.8 (**Table 1**). As the workload increases, the effect becomes progressively smaller. For the *prototype* configuration, for low workload, the factor decreases by 35.5%, for medium workload by 11.2%, and for high workload by 7.9%. For low workload, the achievable acceleration is now determined by inter-node communication latency, but toward higher workload external memory access time is still the main contributor to performance degradation (**Figure 4**, prototype, lower panel).

In the *high data streaming parallelism* configuration, we therefore assign each processing unit its own data stream, and by this means, introduce eight times higher parallelism in the presynaptic data distribution—the two data streams S1 and S2 (**Figure 8**) are each split into eight streams, thus reducing external memory access times by a factor of eight. **Figure 4** (high data stream parallelism, upper and lower panel) illustrate the effect. For medium workload and with inter-node communication, the acceleration factor increases from 15.0 (for the prototype configuration) to 51.7, i.e., by a factor of 3.4.

One may try to further improve performance by an increase in the parallelism of the neuron and synapse model processing, i.e., by introducing a higher number of processing units. The *high processing units parallelism* configuration doubles the number of processing units. This configuration achieves a very high maximum acceleration factor of 571.4 for the single node without inter-node communication. In a cluster such high acceleration cannot be realized, even for low workload. Bound by inter-node communication latency, the performance loss in relation to the maximum acceleration is 74%, and for low workload 81.2%. However, for high workload, external memory access time is still the main limiting factor (**Figure 4**, high processing units parallelism, upper and lower panel).

With regard to the hardware footprint and the required FPGA resources—which is an important aspect of hardware designs—the effect of a reduction of the number of processing units is also of interest. The *low processing units parallelism* configuration, therefore, implements half of the processing units of the *prototype* configuration (**Figure 4**, low processing units parallelism, upper and lower panel). For low workload and in comparison to the *high processing units parallelism* configuration, the acceleration factor decreases from 107.5 to 70.9, i.e., by 34%. For high workload, the acceleration factor decreases from 36.9 to 31.3. This is a loss of only 15.2% and might be an acceptable degradation when making design decisions oriented toward a high workload scenario, given that thereby 75% of ODE pipeline hardware resource, namely digital signal processing (DSP) units, can be saved with this configuration. Saving hardware resources reduces power consumption and thus increases simulation efficiency. Considering the above, for medium workload the *high data stream parallelism* configuration can be a compromise with regard to the achievable acceleration factors for different workload situations and the required chip resources. For the HNC node prototype implementation the utilization of the SoC chip resource are given in the **Supplementary Material**.

The current implementation of the HNC node configured with the *prototype* parameter set and operated in a cluster would achieve an acceleration factor in the order of 10–50 for medium and small workloads. Such a workload is created, for example, by a network consisting of $N = 10,000$ neurons with an average firing rate of $\bar{\nu} \approx 2..10$ spks/s. To simulate such a network, 10 HNC nodes would need to be clustered.

3.3. Correctness

In order to meet the requirement of an accurate and reproducible simulation we evaluated the equivalence of the simulation results produced by the HNC node and a ground truth.

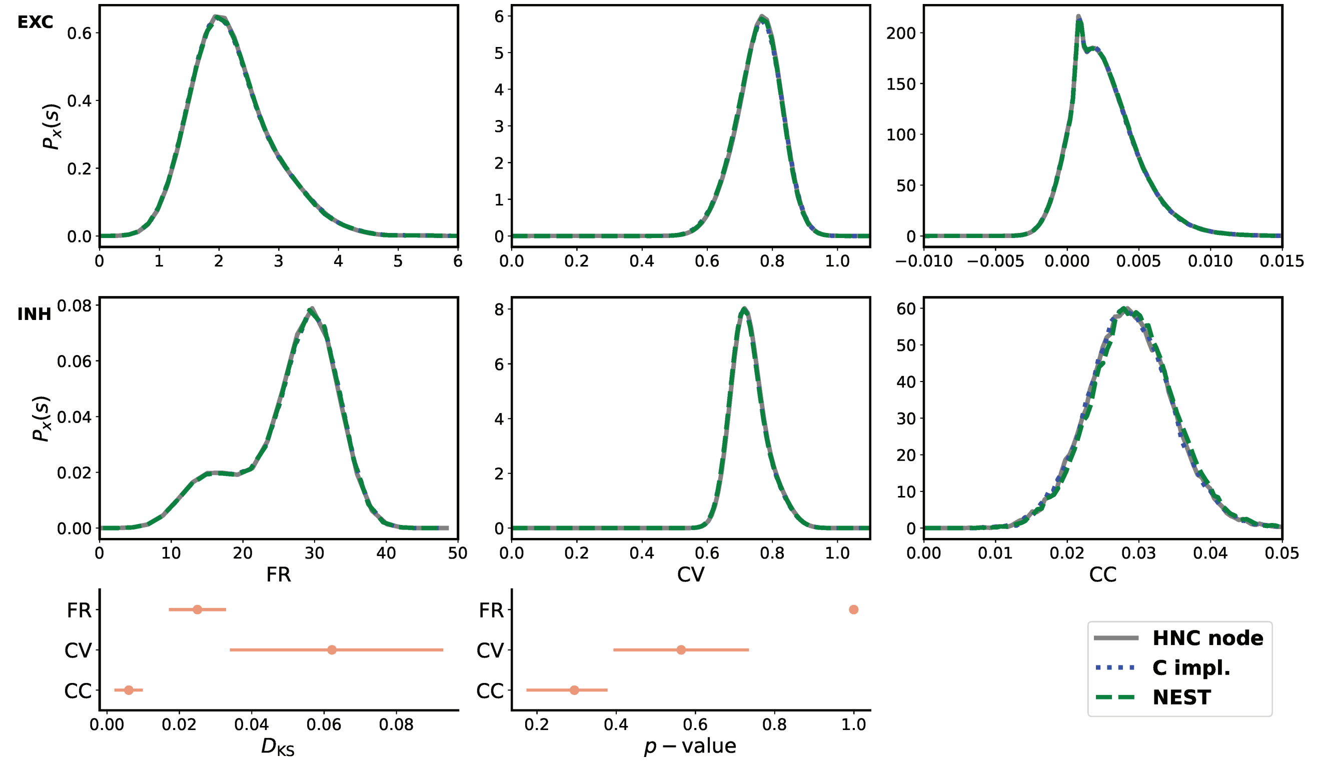


FIGURE 5 | Quantitative comparison of statistical measures. Upper two rows from left to right: probability distribution of average firing rate (FR), coefficient of variation (CV), and Pearson's correlation coefficient (CC) for the excitatory (EXC) and the inhibitory (INH) population. The measures were calculated from 30 min simulated time. For the calculation of CC, spike trains were binned at 2 ms. In order to derive the probability distributions from the calculated measures, the Freedman-Diaconis rule was applied to select the width of the bins of the distribution histograms, and a Gaussian kernel was used for density smoothing. The bottom row shows the Kolmogorov-Smirnov statistics calculated from the raw samples of the calculated statistical measures. During the simulations performed on the HNC node, using the NEST simulator, and carried out using the reference C implementation, the network was stimulated with a different random input—a limitation of the HNC node prototype and hardware implementation of the PRNG. All three simulations used the same explicit Forward Euler integration method with an integration step size of $h = 0.1$ ms. All measures are in close agreement and show statistical equivalence.

In this validation process we aimed for the reproduction of the dynamics of a selected network state obtained from a reference implementation of the two-population Izhikevich network described in Section 5.3. This reference implementation was written in the C language and developed as part of an earlier study (Trensch et al., 2018). The source code is available online[9]. To create the network state, the ground truth, the network was trained for 1 h biological time using a spike time dependent plasticity (STDP) rule (see the description of the network given in the **Supplementary Material**). After 1 h of simulated network time, the current state of the network was captured by exporting the network's connectivity data. The connectivity data was then imported back into the C simulation, and with the STDP rule turned off, from 30 min simulated time the spikes were recorded while the network was stimulated with a random input. This recorded network activity data defined the ground truth, that is, the captured network state that defines a reliable reference. For reproduction, we loaded the connectivity data into the HNC node and repeated the simulation. To provide further evidence and to substantiate the correctness of the simulation result generated by the HNC node, the connectivity data was also imported into the NEST simulator and we repeated the simulation again. When simulating a network, it is sufficient to communicate spike events at intervals less or equal to the minimum synaptic delay in the network. The NEST implementation makes use of this and propagates spike events on a 1 ms grid - the minimal synaptic delay in the two-population Izhikevich network. In contrast, the HNC node communicates spike events at 0.1 ms intervals. For progressing neuron model dynamics, an integration step size of $h = 1$ ms would not be sufficient to achieve the necessary numerical accuracy (Pauli et al., 2018). Therefore, both NEST and the HNC node use an integration step size of $h = 0.1$ ms. The NEST Izhikevich neuron model implementation was adapted accordingly. The simulation script and the source code is available online[9].

From the three obtained data sets of network activity, the probability distribution of the firing rates (FR), the coefficient of variation (CV), and the Pearson's correlation coefficient (CC) were calculated and compared. The statistical measures are described in the **Supplementary Material**. The result of the

[9] https://github.com/gtrensch/RigorousNeuralNetworkSimulations

comparison is shown in **Figure 5**. All measures are in close agreement and show statistical equivalence.

Simulation results must not only be reproducible and in agreement with a reliable reference, but also replicable, i.e., spike-identical in repeated simulations. Replicability was tested by repeatedly simulating the two-population Izhikevich network for 20 minutes simulated time. Due to limited numerical precision and rounding errors, operations are not commutative. Therefore, and to strengthen the tests, the network was also logically shifted across processing units in order to assign a logical neuron-id to different hardware resources, and thus force a different spike ordering and scheduling of operations. The simulation results were successfully validated for spike-identicality (data not shown).

4. ARCHITECTURE

4.1. System-Level Architecture

We chose the XCZ7045 SoC from the Xilinx Zynq-7000 SoC device family (Xilinx, 2021) for the technical implementation, and all work presented in this article was carried out on a Xilinx Zynq-7000 SoC ZC706 development board (Xilinx, 2019e). The XCZ7045 integrates a dual-core ARM Cortex-A9 processor (up to 1 GHz) and a freely programmable and re-configurable logic device, i.e., an FPGA with the size of $350,000$ configurable logic blocks (CLBs). It provides ~$218,000$ look-up tables (LUTs), ~$437,000$ flip-flops (FFs), 19.2 Mbit of fast static block RAM (BRAM) that can be customized for different configurations, and 900 digital signal processing (DSP) blocks for the implementation of arithmetic operations.

Figure 6 shows the system-level view of the implemented HNC node architecture. It details the major components and modules, their interaction and functional assignments. The operation of the HNC node is software-controlled. The program executable is located in the external memory (top right) and executed by the processing system's (PS) application processing unit (APU) (upper dashed box). For user interaction, debugging and data exchange, the HNC node is connected to a Linux host system (upper left) *via* an Ethernet (ENET) connection for the read-out of recorded spike events, a JTAG[10] connection for programming and debugging, and a serial UART[11] user console interface.

The simulation engine's core components are realized in programmable logic (PL). They are shown in the lower dashed box in **Figure 6**. Function-wise, the hardware components can be assigned to four distinct steps in the process of carrying out a simulation cycle: (i) presynaptic data distribution; (ii) presynaptic data processing; (iii) neuron and synapse model update; and (iv) spike events processing.

Presynaptic data distribution: triggered by postsynaptic spike events, the *PS/PL Data Transfer Module* initiates read operations from the external memory to obtain the node-local connectivity information (see Section 4.3.1) of the firing neurons. In order to do so and make optimal use of the read bandwidth of the external memory, the *PS/PL Data Transfer Module* is connected to the PS *via* a pair of high performance ports (HP1, HP3) capable of working independently of one another. At its outputs, the module connects to a series of first-in-first-out (FIFO) buffers (in **Figure 6** referred to as RB FIFOs) which compensate for latencies and to which the presynaptic date is distributed. The RB FIFOs connect the *PS/PL Data Transfer Module* to 16 identical processing units (P1, P2,.., P16). The processing units parallelize and pipeline the computations for the presynaptic data processing and the neuron and synapse model dynamics.

Presynaptic data processing: In order to derive the synaptic inputs i_{ex} and i_{inh} from the presynaptic data, the presynaptic data is fetched from the RB FIFOs and passed through the RB pipelines. The RB pipelines operate on the ring buffers (RBs) and accumulate the synaptic inputs, the values of which are stored and delayed for further processing by placing them into the RBs.

Neuron and synapse model update: The ordinary differential equation solver pipelines (ODE pipelines) retrieve the accumulated synaptic input values from the RBs and progresses the neuron and synapse model dynamics; updating the models' state vectors in the state variables buffers (SVBs). In addition, an XNOR-shift PRNG can provide a random external network stimulus $\{i_{\text{ext}}^{\text{P1}}, .., i_{\text{ext}}^{\text{P16}}\}$ which is directly applied to the neurons in the ODE pipelines.

Spike events processing: In principle, there can be as many spike events occurring in each unit, and in a single simulation time step k, as the number of neurons processed in a pipeline. In other words, in extreme, $16 \cdot N^{\text{P}} = N^{\text{M}} = 1,024$ spike events need be buffered, serialized and packed for local (intra-node) and external (inter-node) spike communication, as well as for recording. The associated components that are related to this process are shown at the lower right in **Figure 6**.

In order to enable the APU to perform software-controlled read and write operations on the SVBs to access the state variables, all processing units are chained to one another and connected to a direct memory access (DMA) controller.

The aforementioned modules mainly represent the data paths or operate on them. To orchestrate the control flow, additional components are required for configuration, simulation control, and synchronization. For configuration and simulation control, a bank of 32-bit registers store node control and status information (shown at the mid left in **Figure 6**). All registers are mapped into the APU's address space and thus accessible by the node software. Their settings steer the operation of a finite state machine (FSM) responsible for generating all control signal sequences for the different operating modes (e.g., load state variables, progress simulation by k steps, unload state variables). To preserve the temporal causality and ensure the correct sequence of operations, all spike events of a simulation step k must have been delivered and the RB buffer updates must have been completed before the next simulation step $k+1$ can be initiated. This is ensured by an intra-node synchronization logic which monitors the operating status of all modules. The module is shown at the lower left in **Figure 6**. Technically, it implements a barrier mechanism that synchronizes the overall processing at the end of every simulation step. In a multi-node configuration this extends to an inter-node barrier message—software simulators, such as

[10] JTAG is an industry standard named after the Joint Test Action Group.

[11] Universal Asynchronous Receiver Transmitter.

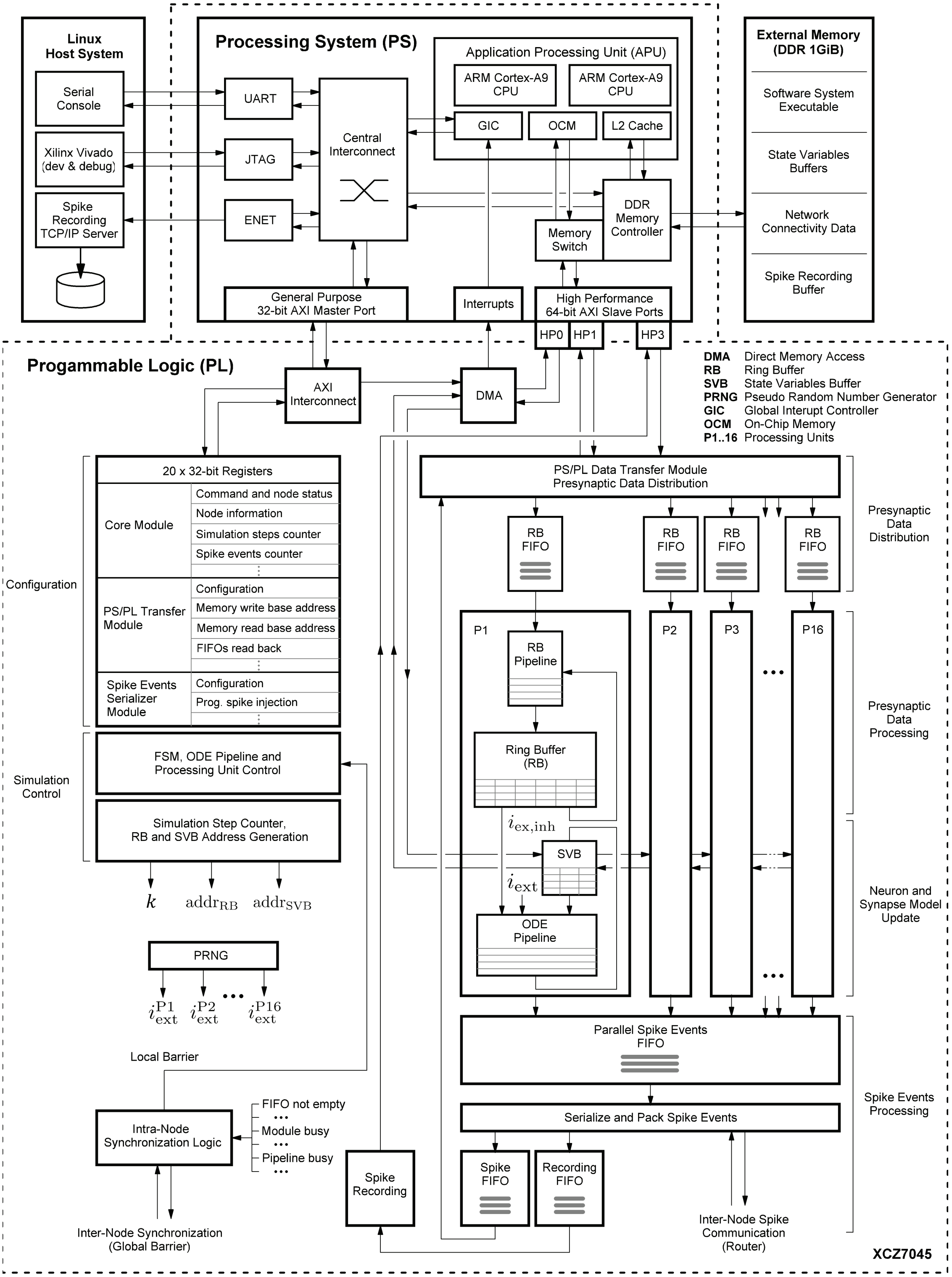

Linux Host System
Serial Console
Xilinx Vivado (dev & debug)
Spike Recording TCP/IP Server
Processing System (PS)
UART
JTAG
ENET
Central Interconnect
Application Processing Unit (APU)
ARM Cortex-A9 CPU
ARM Cortex-A9 CPU
GIC
OCM
L2 Cache
DDR Memory Controller
Memory Switch
General Purpose 32-bit AXI Master Port
Interrupts
High Performance 64-bit AXI Slave Ports
HP0
HP1
HP3
External Memory (DDR 1GiB)
Software System Executable
State Variables Buffers
Network Connectivity Data
Spike Recording Buffer
Progammable Logic (PL)
AXI Interconnect
DMA
DMA Direct Memory Access
RB Ring Buffer
SVB State Variables Buffer
PRNG Pseudo Random Number Generator
GIC Global Interupt Controller
OCM On-Chip Memory
P1..16 Processing Units
20 x 32-bit Registers
Core Module
Command and node status
Node information
Simulation steps counter
Spike events counter
Configuration
PS/PL Transfer Module
Configuration
Memory write base address
Memory read base address
FIFOs read back
Spike Events Serializer Module
Configuration
Prog. spike injection
Simulation Control
FSM, ODE Pipeline and Processing Unit Control
Simulation Step Counter, RB and SVB Address Generation
k
$\mathrm{addr}_{\mathrm{RB}}$
$\mathrm{addr}_{\mathrm{SVB}}$
PRNG
$i_{\mathrm{ext}}^{\mathrm{P1}}$
$i_{\mathrm{ext}}^{\mathrm{P2}}$
$i_{\mathrm{ext}}^{\mathrm{P16}}$
Local Barrier
Intra-Node Synchronization Logic
FIFO not empty
Module busy
Pipeline busy
Inter-Node Synchronization (Global Barrier)
Spike Recording
PS/PL Data Transfer Module Presynaptic Data Distribution
RB FIFO
RB FIFO
RB FIFO
RB FIFO
Presynaptic Data Distribution
P1
P2
P3
P16
RB Pipeline
Ring Buffer (RB)
$i_{\mathrm{ex,inh}}$
SVB
i_{ext}
ODE Pipeline
Presynaptic Data Processing
Neuron and Synapse Model Update
Parallel Spike Events FIFO
Serialize and Pack Spike Events
Spike FIFO
Recording FIFO
Inter-Node Spike Communication (Router)
Spike Events Processing
XCZ7045

FIGURE 6 | System-level view of the HNC node hardware architecture. The on-chip components are framed by the dashed lines. The lower frame encloses all modules that have been implemented in programmable logic (PL), while in the upper frame the components of the processing system (PS) are shown. Attached to it is an external 1 GiB DDR RAM module (upper right). It stores the node software system executable and the data structures required for operation, for example, the state variables and connectivity information. The external memory also functions as buffer for the recorded spike data. The PS is further connected to a Linux host system (upper left) which provides a serial console to operate the HNC node, the Xilinx Vivado environment for development, and a TCP/IP server to collect the recorded binary spike data.

NEST (Gewaltig and Diesmann, 2007) use MPI[12] barrier calls for this purpose.

The entire hardware design—with the exception of the DMA controller and the FIFO blocks for which Xilinx soft IP cores were used—was implemented on the register transfer level (RTL) in VHDL. The decision to take this more arduous and time consuming approach—rather than a high level synthesis (HLS) implementation (Xilinx, 2019c)—is motivated by the endeavor to maximize control over the microarchitecture details in order to optimize the timing behavior. The current HNC node implementation works stable up to a PL clock frequency of $f_{\text{clk}} = 200$ MHz. The software implementation was carried out in the C language. For the development process the Xilinx Vivado Design Suite (Xilinx, 2019d) and the Xilinx Vivado SDK and embedded system tools (Xilinx, 2019b) were used which provide the development tools for hardware-software co-design, synthesis and analysis.

4.2. Software System Architecture

Figure 7 outlines the basic architecture of the HNC node software system, which is executed by the SoC's integrated APU. At its lowest level, an abstraction layer provides fundamental routines to drive the hardware functions, for example, to reset and initialize components, to handle interrupts, to establish a basic serial console and TCP/IP communication, and to initiate direct memory access (DMA) transfer operations. Helper- and low-level simulator functions, such as routines to load and unload the state variable buffers, build on top of this layer providing the foundation for the actual simulator functions—the kernel of the software system. The main components here are the `Neuron Manager`, responsible for the instantiation of neurons, and the `Connection Manager`, responsible for creating the synaptic connections. At the highest level, a C-API provides `Create(..)`, `Connect(..)`, and `Simulate(..)` function calls, which represent a minimal set of functions required to instantiate and simulate a network. Besides the simulator core-functionality, we implemented functions for system configuration, testing and debugging as well as for user-interactive node control. Access to those is given through a serial user console interface. To minimize the resources footprint and achieve best possible performance, the software system was implemented as bare-metal application, running natively without the use of any underlying operating system. When executed, it makes use of one of the two ARM Cortex-A9 cores that the APU provides. During the execution of a `Simulate(..)` function call, no operations on the external memory are performed by the APU. This allows the PL to make optimal use of the bandwidth of the external memory while a simulation is running.

4.2.1. Node-Local Network Instantiation

The current HNC node prototype requires that the neural network model is formulated as a sequence of `Create` and `Connect` function calls, which needs to be compiled to an executable. In this object-format it is loaded into the external memory and executed when a `Simulate` function call is issued. Each `Create` instantiates a single neuron. The function takes as its arguments a model name, the initial values of the neuron's state variables, and a logical neuron-id, which identifies the neuron on the node. The `Create` function calls are processed by the `Neuron Manager`. It maps the logical neuron-id to a dedicated hardware resource identified by a resource-id, i.e., a processing unit and a position in the ODE pipeline. This process mainly consists of setting up the data structures for state variables in memory while administering byte-orders and data type conversions according to the model-specific hardware implementation. The DMA controller operates directly on these data structures when the processing units are *"loaded"* and the state variables are moved to the SVBs—and also vice versa when *"unloaded"* and the data is read back to external memory. In the current implementation, an interrupt-controlled DMA operation takes $\approx 30\mu s$ to fill the SVBs while 16KiB of data is transferred in order to load or unload the states of $N^{\text{P}} = 1024$ neurons.

Analogous to the `Create` function call for the instantiation of a neuron, a `Connect` function call creates a single connection. It expects in its argument list a logical source neuron-id (for a multi-node system extended by a node-id), a logical target neuron-id, as well as the synapse parameters, i.e., a weight and a delay. From the sequence of `Connect` function calls, the `Connection Manager` builds the data structures in the external memory that represent the network connectivity. This structure associates each source neuron with a list of synapse target connections.

4.2.2. Recording

The HNC node implements two different solutions for recording the network activity data, one for recording spikes and one for recording state variables. Recording spike events is a fully asynchronous process which is decoupled from the simulation scheduling. During a simulation, the spike events are grouped together as they occur and packed to 64-bit values which are buffered in the `Recording FIFO` (shown at the bottom in **Figure 6**) before being written to external memory. The high performance port HP3 is used for the write operations. Its read channel is already assigned to the retrieving of the presynaptic data. Sharing the port—and the external memory—does not create any visible read-write contention. Performance measurements carried out with and without spike recording did not show any degradation in performance with active spike recording and led to comparable results for the measured acceleration factors, even at high spike rates. The current design implements a recording buffer with a size of 60MiB capable of caching ~15M spike events. This buffer is written by the recording hardware in a round robin manner and emptied by the simulation kernel's `Recording Client` (**Figure 7**), which transfers the data *via* a TCP connection to a TCP server running on the Linux host system. For the client implementation on the HNC node the open-source *lightweight IP*[13] (lwIP) TCP/IP stack was used, which comes with the Xilinx board support package,

[12] Message Passing Interface, https://www.mpi-forum.org/.

[13] http://savannah.nongnu.org/projects/lwip/

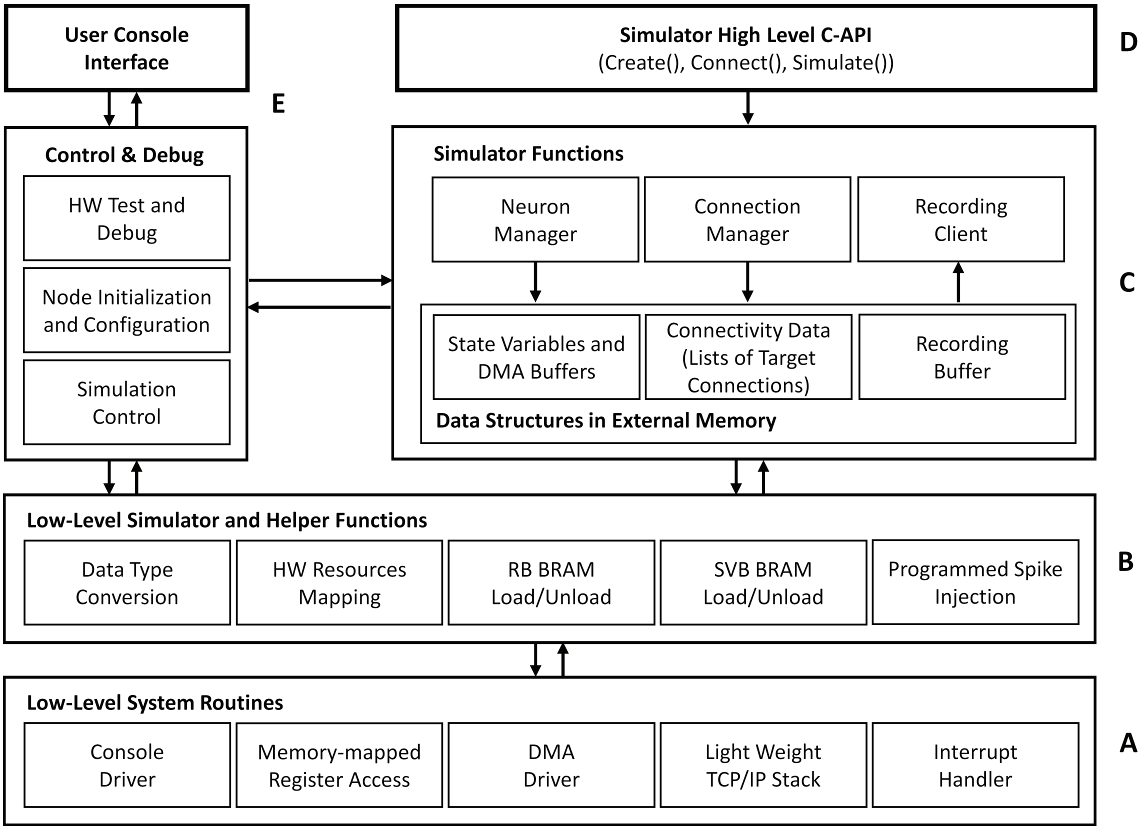

User Console Interface
Simulator High Level C-API
(Create(), Connect(), Simulate())
D
E
Control & Debug
HW Test and Debug
Node Initialization and Configuration
Simulation Control
Simulator Functions
Neuron Manager
Connection Manager
Recording Client
C
State Variables and DMA Buffers
Connectivity Data (Lists of Target Connections)
Recording Buffer
Data Structures in External Memory
Low-Level Simulator and Helper Functions
Data Type Conversion
HW Resources Mapping
RB BRAM Load/Unload
SVB BRAM Load/Unload
Programmed Spike Injection
B
Low-Level System Routines
Console Driver
Memory-mapped Register Access
DMA Driver
Light Weight TCP/IP Stack
Interrupt Handler
A

FIGURE 7 | HNC node software system architecture. The tiered architecture provides abstraction at different functional levels. **(A)** The low-level system routines hide technical details about the operation of the implemented hardware components. Based on this, low-level simulator and helper functions **(B)** form the foundation for the core component of the HNC node software **(C)**, that is, the simulator functions. **(D)** At the highest level, a minimal set of functions is provided to instantiate and simulate a network. **(E)** In addition, the software system implements components that are responsible for control, testing, and debugging and also enable user interaction.

and is included in the Vivado SDK. In order to read out state variables, a running simulation must be halted to allow the DMA controller to access the SVBs. Consequently, capturing state variables significantly reduces performance. On the other hand, the DMA provides the APU with an efficient way to access all state variables at once and at any desired interval.

4.3. Microarchitecture

The module microarchitectures presented in this section try to bring the data and the operations performed on them as close together as possible. The implementations aim at optimal low-latency solutions utilizing SoC device features, such as low-latency BRAM and high-performance streaming interfaces for external memory access.

4.3.1. Connectivity Representation and Presynaptic Data Distribution

The structure in which the network connectivity data is stored in memory is determined by the microarchitecture of the *PS/PL Data Transfer Module*, which is shown in **Figure 8**. Upon the arrival of a spike event, it retrieves the list of synapse target connections C_j associated with a source neuron n_j, and distributes the data items to the RB FIFO buffers for further processing by the RB pipelines (see also **Figure 6**). Such a retrieved list constitutes the *presynaptic data*. It is represented by a list of quadruples $C_j = \{(s_{ij}, n_i, w_{ij}, d_{ij}), .., ()\}$, where n_i specifies the target neuron, w_{ij} and d_{ij} denote the synaptic weight and delay values, and s_{ij} is a data path control value assigning a data item to its associated RB FIFO buffer by controlling the demultiplexer circuits (DMUX, **Figure 8**). The data format of the synaptic target list items is detailed in **Supplementary Figure S1**. The demultiplexers connect the data paths alternately with the RB FIFO buffers and thus the processing units. This architecture detail comes in handy when removing, adding, or combining processing units, as it helps to maintain a balanced load on the high-performance ports. The design and implementation of the module aim at lowest possible data access latency and an optimal utilization of the available read bandwidth of the external memory. Therefore, the *PS/PL Data Transfer Module*, residing in the PL, is interfaced with the PS, and thus with the external memory, through the two high-performance ports HP1 and HP3. This splits the target list into the two lists C_j^{S1} and C_j^{S2} assigned to HP1 and HP3, respectively. Their assignment (and associated data paths) are indicated in red and blue in

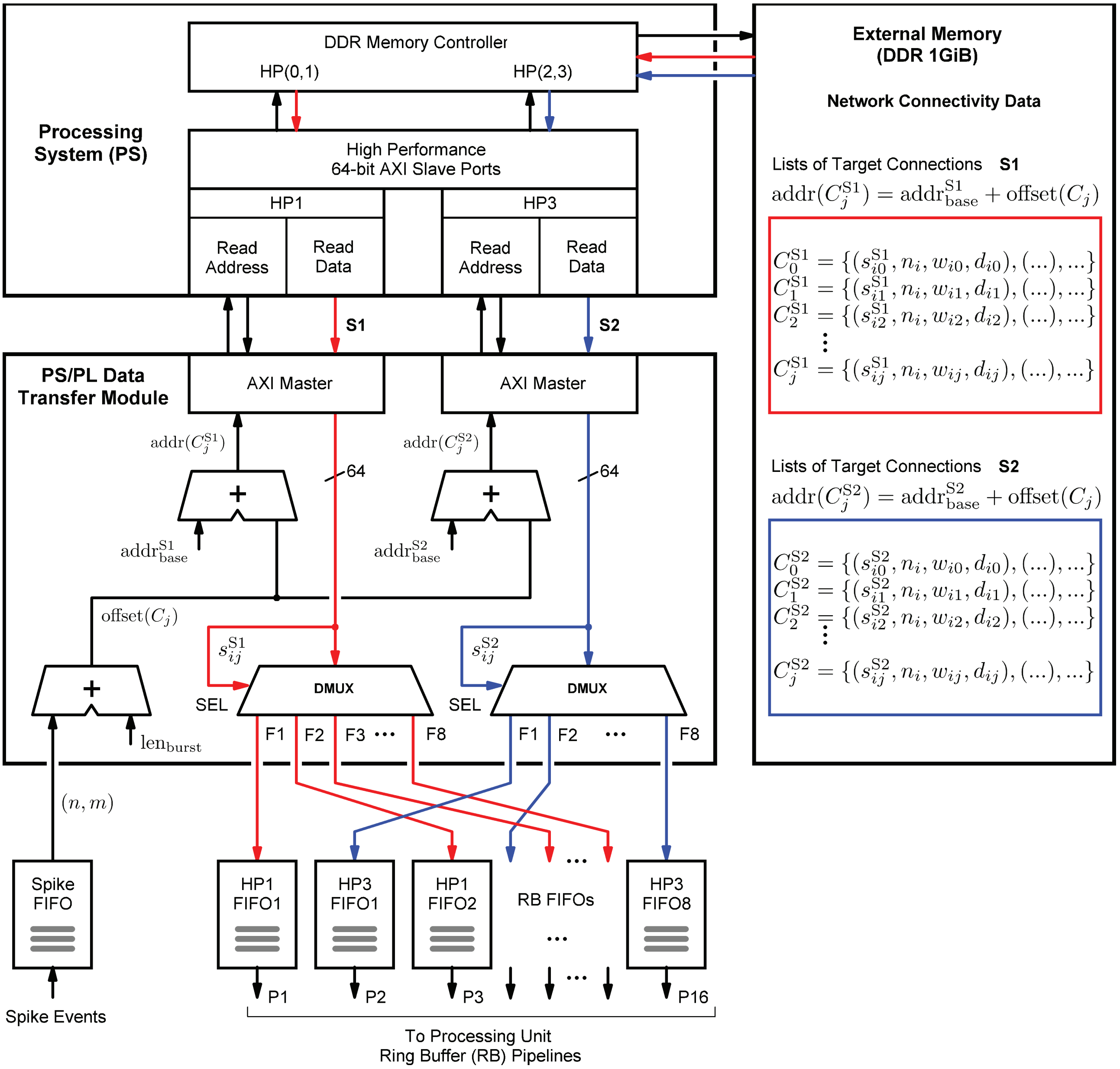

DDR Memory Controller
HP(0,1)
HP(2,3)
Processing System (PS)
High Performance 64-bit AXI Slave Ports
HP1
HP3
Read Address
Read Data
Read Address
Read Data
S1
S2
PS/PL Data Transfer Module
AXI Master
AXI Master
$\mathrm{addr}(C_j^{\mathrm{S1}})$
$\mathrm{addr}(C_j^{\mathrm{S2}})$
64
64
+
+
$\mathrm{addr}_{\mathrm{base}}^{\mathrm{S1}}$
$\mathrm{addr}_{\mathrm{base}}^{\mathrm{S2}}$
$\mathrm{offset}(C_j)$
s_{ij}^{S1}
s_{ij}^{S2}
DMUX
DMUX
SEL
SEL
+
$\mathrm{len}_{\mathrm{burst}}$
F1
F2
F3 ···
F8
F1
F2
···
F8
(n, m)
Spike FIFO
Spike Events
HP1 FIFO1
HP3 FIFO1
HP1 FIFO2
RB FIFOs
HP3 FIFO8
P1
P2
P3
P16
To Processing Unit Ring Buffer (RB) Pipelines
External Memory (DDR 1GiB)
Network Connectivity Data
Lists of Target Connections S1
$\mathrm{addr}(C_j^{\mathrm{S1}}) = \mathrm{addr}_{\mathrm{base}}^{\mathrm{S1}} + \mathrm{offset}(C_j)$
$C_0^{\mathrm{S1}} = \{(s_{i0}^{\mathrm{S1}}, n_i, w_{i0}, d_{i0}), (...), ...\}$
$C_1^{\mathrm{S1}} = \{(s_{i1}^{\mathrm{S1}}, n_i, w_{i1}, d_{i1}), (...), ...\}$
$C_2^{\mathrm{S1}} = \{(s_{i2}^{\mathrm{S1}}, n_i, w_{i2}, d_{i2}), (...), ...\}$
$C_j^{\mathrm{S1}} = \{(s_{ij}^{\mathrm{S1}}, n_i, w_{ij}, d_{ij}), (...), ...\}$
Lists of Target Connections S2
$\mathrm{addr}(C_j^{\mathrm{S2}}) = \mathrm{addr}_{\mathrm{base}}^{\mathrm{S2}} + \mathrm{offset}(C_j)$
$C_0^{\mathrm{S2}} = \{(s_{i0}^{\mathrm{S2}}, n_i, w_{i0}, d_{i0}), (...), ...\}$
$C_1^{\mathrm{S2}} = \{(s_{i1}^{\mathrm{S2}}, n_i, w_{i1}, d_{i1}), (...), ...\}$
$C_2^{\mathrm{S2}} = \{(s_{i2}^{\mathrm{S2}}, n_i, w_{i2}, d_{i2}), (...), ...\}$
$C_j^{\mathrm{S2}} = \{(s_{ij}^{\mathrm{S2}}, n_i, w_{ij}, d_{ij}), (...), ...\}$

FIGURE 8 | Presynaptic data distribution. Upon the arrival of a spike event, the presynaptic data is read from the external memory in two independent parallel data streams S1 and S2, indicated by the red and blue arrows, and distributed to the RB FIFOs by the demultiplexers (DMUX). While the *Processing System (PS)* performs the external storage operations bypassing the APU (not shown in the figure), the *PS/PL Data Transfer Module* controls the two AXI data streams and the high-performance ports HP1 and HP3 through which it connects to the PS. It calculates the memory addresses of two lists, C_j^{S1} and C_j^{S2} which constitute the two data streams, that is, the presynaptic data associated with the neuron that has emitted the spike. This data is stored in two different memory regions, marked by the red and blue boxes.

Figure 8. The two high-performance ports are capable of working in parallel and independently of one another, while for example, the ports HP0 and HP1 would share the same PS resources, hindering full parallelism.

In terms of implementation, the port interfaces follow the *Advanced eXtensible Interface*[14] (AXI) standard (Arm Limited, 2021). More precisely, they provide 64-bit AXI3 Slave interfaces. On the PL, the *PS/PL Data Transfer Module* architecture bundles two AXI Master stream interface implementations that constitute their counterparts. The AXI protocol is based on data bursts. The presynaptic data to be retrieved upon the occurrence of a single spike event is transmitted in two parallel sequences of four bursts, i.e., four bursts on each port, where a burst consists of 16 64-bit data items. The principle is shown in **Figure 9**. The red and blue colors correspond to the datapath coloring in **Figure 8**.

[14] The Advanced eXtensible Interface (AXI) standard is an extension of the Advanced Microcontroller Bus Architecture (AMBA), which is an open standard.

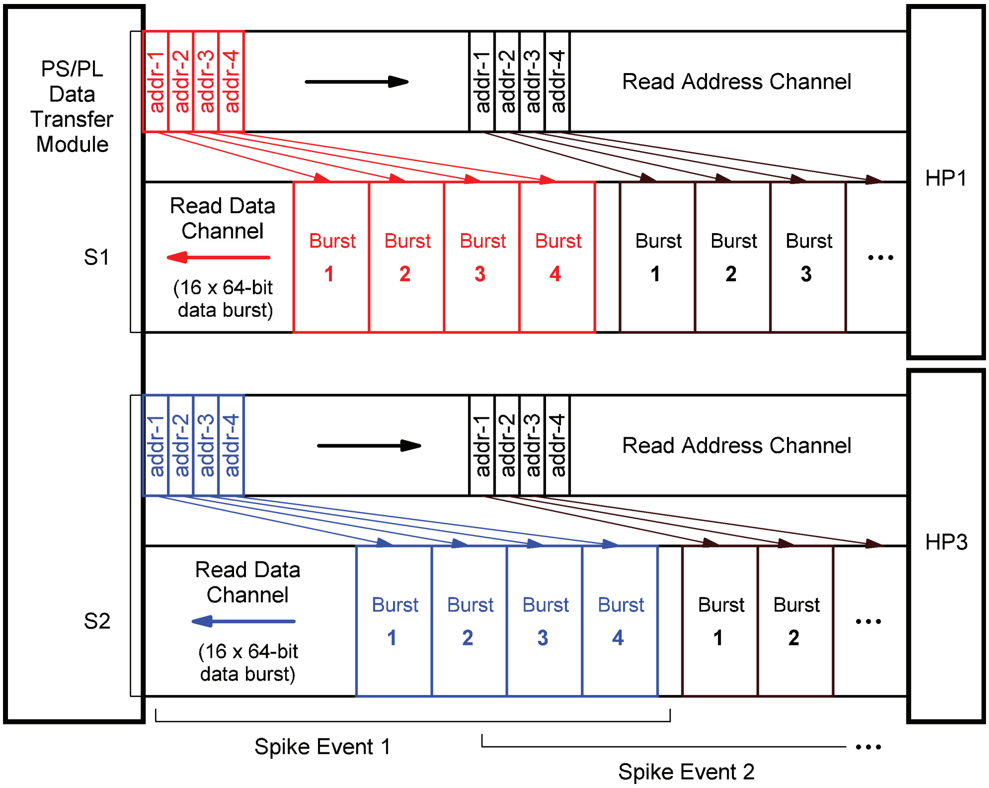


FIGURE 9 | AXI stream protocol implementation. To create data streams that are as continuous as possible, data transfers are already scheduled without waiting for the preceding transfer to complete. Per spike event, the transfer of a sequence of four data bursts is initiated on each of the two read data channels associated with the two streams S1 and S2 (marked red and blue). For this purpose, the memory read base addresses of the four burst data packets are transmitted as a block on the read address channels.

The read address channels describe the address and control information of the data bursts transferred on the read data channels. The addresses $\left(\mathrm{addr}(C_j^{S1}), \mathrm{addr}(C_j^{S2})\right)$ are calculated from the neuron-id and node-id (n_j, m) of the source neuron that emitted the spike, the burst length $(\mathrm{len}_{\mathrm{burst}})$, and the memory base addresses of the two target lists $\left(\mathrm{addr}_{\mathrm{base}}^{S1}, \mathrm{addr}_{\mathrm{base}}^{S2}\right)$.

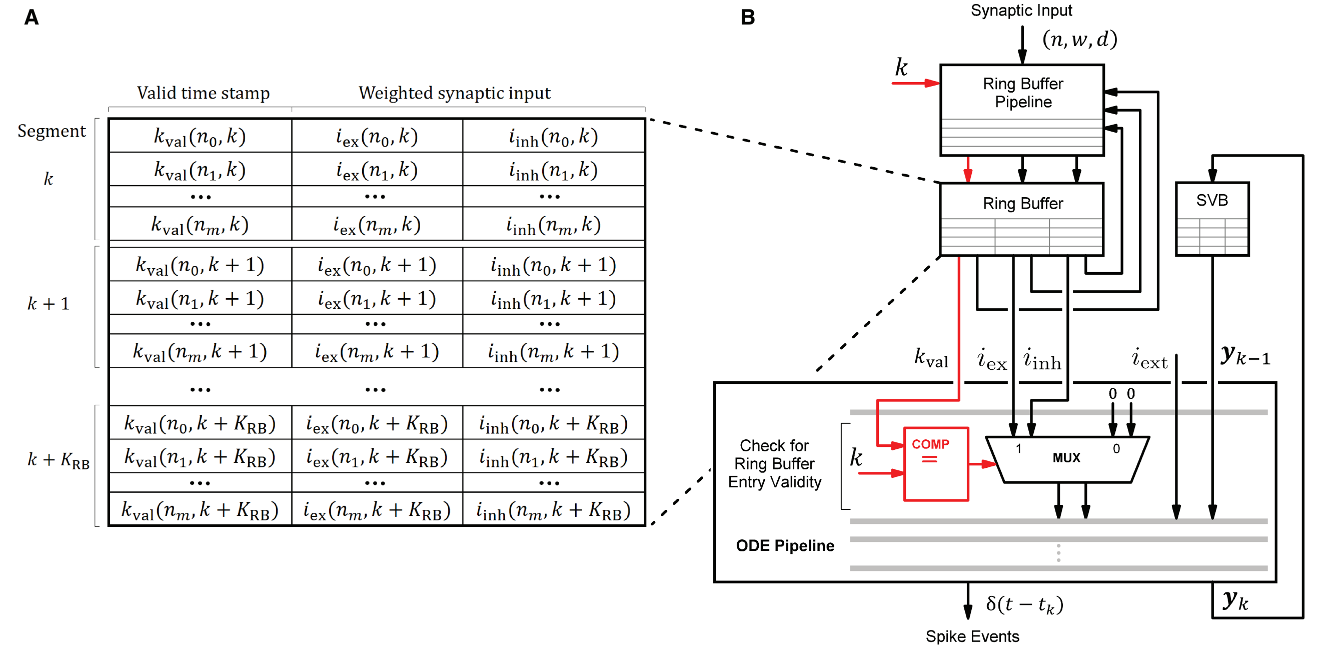
In order to generate data streams that are as continuous as possible, read operations that are triggered by subsequent spike events are already scheduled even though the read data channels are still occupied. By this means, the two data streams S1 and S2 are created. Every spike event triggers a transfer of a 1KiB data packet from external memory. For a single data packet, an average transmission time of ~550 ns ($f_{\mathrm{clk}} = 200$ MHz) was measured. This corresponds to a data transfer rate of 1818 MiB/s which is a much higher throughput than achievable with a Xilinx AXI DMA soft IP core (Xilinx, 2019a)—the common solution for high-bandwidth direct memory access. The DMA soft IP core throughput is specified with 399.04 MB/s at 100 MHz clock frequency (Xilinx, 2019a).

The transfer parameters, the number of bursts and the size of a burst, are configurable in control registers. They were set as discussed above allowing a source neuron to make 128 synapses on a node. In the current prototypical implementation, the transferred data packets are of same size for all spike events. Unused list entries are read from memory but they are not distributed.

The RB FIFO buffers which connect the *PS/PL Data Transfer Module* with the processing units serve two purposes. First, they buffer the synaptic input derived from incoming spike events for the time that the ODE pipelines are operating on the ring buffers (RBs) and blocking them for parallel read operations, and second, they allow a clock domain crossing. We have not yet investigated the latter, but it would allow the *PS/PL Data Transfer Module* to operate at a higher clock frequency than the processing units, which could have a positive impact on the latency of external memory data access.

4.3.2. Ring Buffer Processing and Ordinary Differential Equation Solver Pipeline

The HNC node's processing units draw their ability to accelerate computations primarily from the pipelined processing when accumulating the synaptic inputs in the ring buffers, and when progressing the neuron and synapse model dynamics in the ODE pipelines. This capacity builds on the usage of fast, low-latency on-chip BRAM for storing local variables. **Figure 10** shows the involved components and their interaction for a single processing unit. In every simulation time step, the ODE



Segment	Valid time stamp	Weighted synaptic input	
k	$k_{\mathrm{val}}(n_0, k)$	$i_{\mathrm{ex}}(n_0, k)$	$i_{\mathrm{inh}}(n_0, k)$
	$k_{\mathrm{val}}(n_1, k)$	$i_{\mathrm{ex}}(n_1, k)$	$i_{\mathrm{inh}}(n_1, k)$
	...	...	...
	$k_{\mathrm{val}}(n_m, k)$	$i_{\mathrm{ex}}(n_m, k)$	$i_{\mathrm{inh}}(n_m, k)$
$k+1$	$k_{\mathrm{val}}(n_0, k+1)$	$i_{\mathrm{ex}}(n_0, k+1)$	$i_{\mathrm{inh}}(n_0, k+1)$
	$k_{\mathrm{val}}(n_1, k+1)$	$i_{\mathrm{ex}}(n_1, k+1)$	$i_{\mathrm{inh}}(n_1, k+1)$
	...	...	...
	$k_{\mathrm{val}}(n_m, k+1)$	$i_{\mathrm{ex}}(n_m, k+1)$	$i_{\mathrm{inh}}(n_m, k+1)$
	...	...	...
$k+K_{\mathrm{RB}}$	$k_{\mathrm{val}}(n_0, k+K_{\mathrm{RB}})$	$i_{\mathrm{ex}}(n_0, k+K_{\mathrm{RB}})$	$i_{\mathrm{inh}}(n_0, k+K_{\mathrm{RB}})$
	$k_{\mathrm{val}}(n_1, k+K_{\mathrm{RB}})$	$i_{\mathrm{ex}}(n_1, k+K_{\mathrm{RB}})$	$i_{\mathrm{inh}}(n_1, k+K_{\mathrm{RB}})$
	...	...	...
	$k_{\mathrm{val}}(n_m, k+K_{\mathrm{RB}})$	$i_{\mathrm{ex}}(n_m, k+K_{\mathrm{RB}})$	$i_{\mathrm{inh}}(n_m, k+K_{\mathrm{RB}})$

FIGURE 10 | Ring buffer (RB) architecture and interaction of components. Local variables are stored in the ring buffer (RB) and the state variables buffer (SVB). For their implementation fast, low-latency on-chip BRAM memories were used. Shown is the interaction of the RB pipeline and the ODE pipeline which both operate on the RB. To avoid additional write operations by the ODE pipeline to invalidate the RB entries when processed, each entry is provided with a time stamp k_{val} that indicates the RB cycle for which it is valid. The value of k_{val} is calculated by the RB pipeline—see also the RB update algorithm (**Figure 11**)—and compared with the current simulation time step to verify an entry's validity when read by the ODE pipeline for processing. The principle is illustrated in **(B)** where the data path is marked in red. The corresponding RB layout is shown in **(A)**.

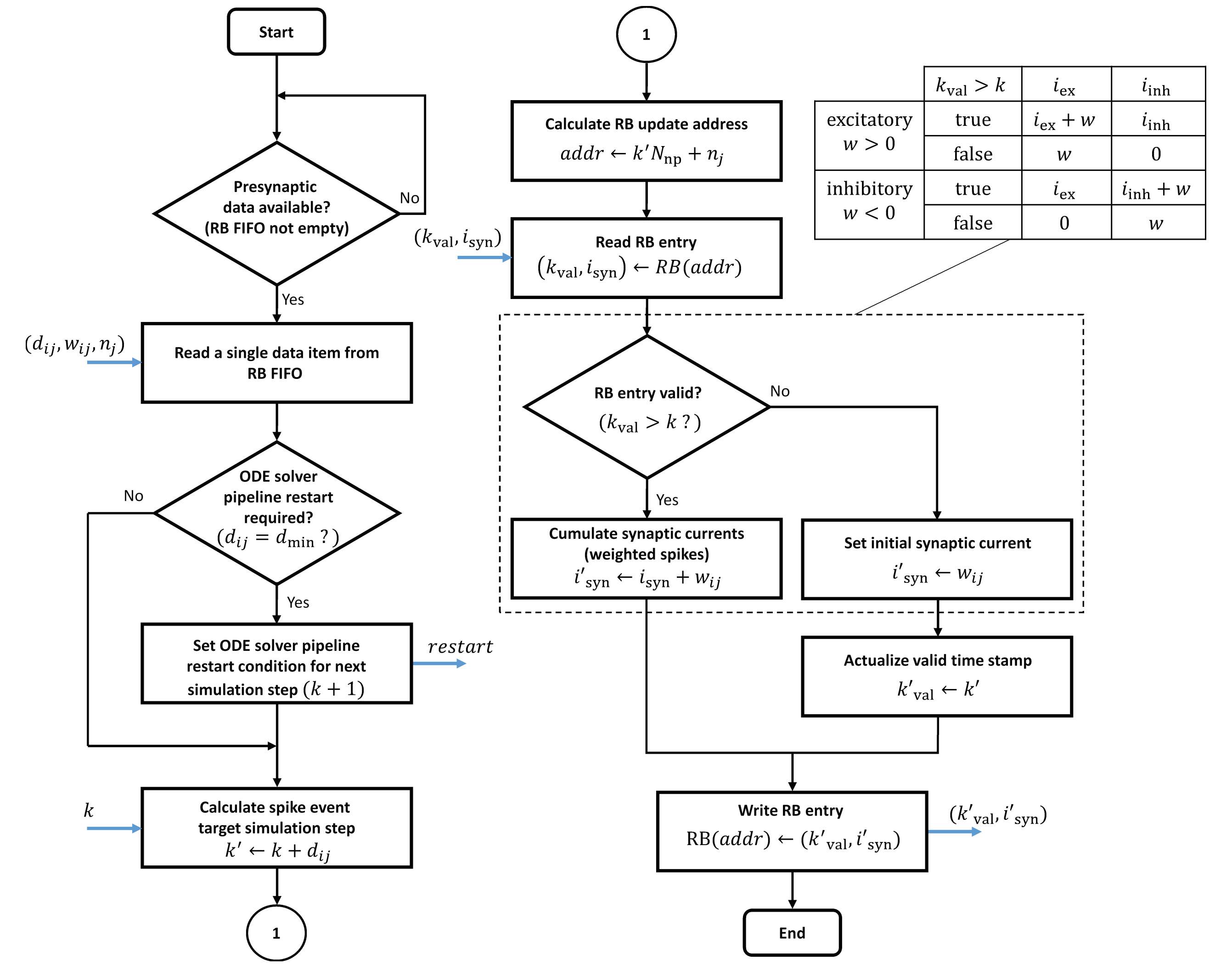
pipeline updates the state vectors of 64 neurons ($\boldsymbol{y}_k \rightarrow \boldsymbol{y}_{k+1}$) while operating on the state variables buffer (SVB). The SVB is implemented as true dual-port BRAM enabling high pipeline-throughput and minimal iteration latency. The state vectors are implemented as 128-bit data words, where 120 bits are available for use to store the state variables and 8 bits are required for pipeline control. The representation of the 120-bit data word, in terms of the number of state variables, their length and type, is determined by the model's hardware implementation and its counterpart in the software system, namely the function for neuron instantiation as part of the neuron manager. This generic approach allows a certain flexibility with regard to the choice of data types and operations according to the numerical precision required by the model to be implemented. This architecture is open to extensions, as the ODE pipeline module can be exchanged to support a wide variety of neuron and synapse models. An example implementation is given in **Supplementary Figure S3**. It shows the microarchitecture of the Izhikevich model implementation used for the performance evaluation and validation task conducted in this work.

An ODE pipeline retrieves the accumulated synaptic inputs i_{ex} and i_{inh} from the RB and may also receive input from an external source, such as a PRNG. Like the SVB, the RB is also implemented as true dual-port BRAM. The buffer layout, shown in **Figure 10A**, consists of K_{RB} segments subdivided into $N^{\mathrm{P}} = 64$ entries - the number of neurons in the pipeline. The RB is read in a round-robin fashion by the ODE pipeline, such that a segment is re-addressed after $k + K_{\mathrm{RB}}$ simulation time steps. The delay resolution—the minimum of which is given by the simulation resolution, i.e., $d_{\mathrm{min}} = h = 0.1$ ms—and the number of segments K_{RB} determine the maximum possible synaptic delay.

RB entries that have already been processed, and are thus outdated, remain in the buffer and may be erroneously reprocessed by the ODE pipeline in subsequent RB cycles. In order to avoid having to add an additional write operation to the ODE pipeline to mark an entry as processed, and thus invalid, we implemented a solution which turns this approach around. When updated, an entry is marked with a time stamp k_{val} that indicates the RB cycle for which the entry is valid. The principle is shown in **Figure 10B**. This *valid time stamp* is derived from the calculated target simulation step $k' \leftarrow k + d_{ij}$ excluding the lower $\log_2(K_{\mathrm{RB}})$ digits. Upon entering the ODE pipeline, the higher order bits of k and the value of k_{val} are checked for equality. If this is the case, i_{ex} and i_{inh} are valid synaptic inputs. This method further avoids the restoring of RB entries in the situation of an ODE pipeline restart (see below). The disadvantage of this solution is a higher consumption of the scarce BRAM resources.

In contrast to the ODE pipelines that are controlled by a finite state machine, an RB pipeline works in a purely event-driven fashion. When not stalled by ODE pipeline operations, the presynaptic data buffered in the RB FIFO is being fetched. It is then passed through the RB pipeline which executes the ring buffer algorithm detailed in the flow diagram in **Figure 11**.

The proposed design raises two issues of potential read-before-write conflicts which need to be taken into consideration.

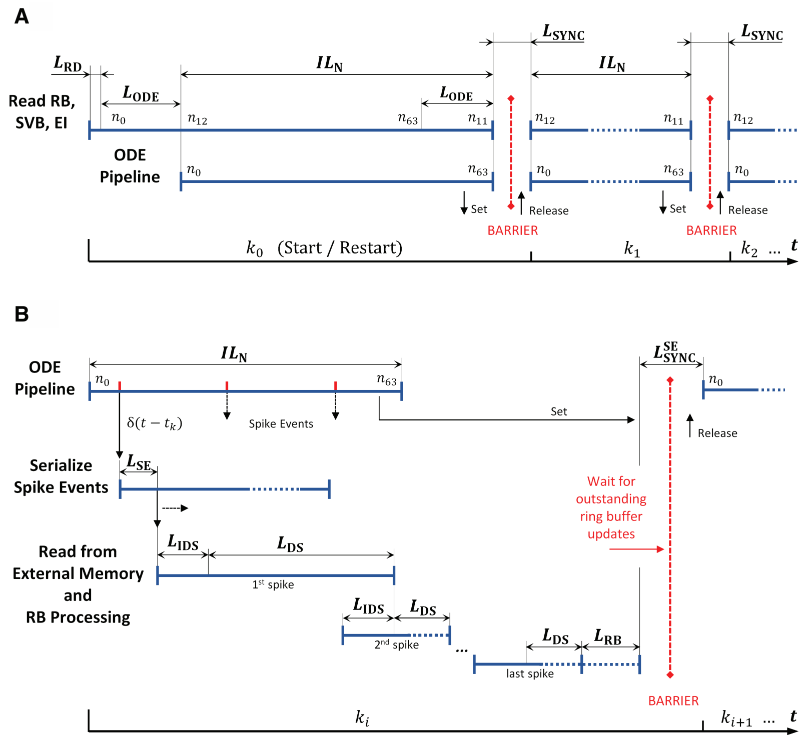

Start
Presynaptic data available? (RB FIFO not empty)
No
Yes
(d_{ij}, w_{ij}, n_j)
Read a single data item from RB FIFO
ODE solver pipeline restart required? $(d_{ij} = d_{\min}\ ?)$
No
Yes
Set ODE solver pipeline restart condition for next simulation step $(k+1)$
$restart$
k
Calculate spike event target simulation step $k' \leftarrow k + d_{ij}$
1
1
Calculate RB update address $addr \leftarrow k' N_{\text{np}} + n_j$
$(k_{\text{val}}, i_{\text{syn}})$
Read RB entry $(k_{\text{val}}, i_{\text{syn}}) \leftarrow RB(addr)$
RB entry valid? $(k_{\text{val}} > k\ ?)$
No
Yes
Cumulate synaptic currents (weighted spikes) $i'_{\text{syn}} \leftarrow i_{\text{syn}} + w_{ij}$
Set initial synaptic current $i'_{\text{syn}} \leftarrow w_{ij}$
Actualize valid time stamp $k'_{\text{val}} \leftarrow k'$
Write RB entry $\text{RB}(addr) \leftarrow (k'_{\text{val}}, i'_{\text{syn}})$
$(k'_{\text{val}}, i'_{\text{syn}})$
End

		$k_{\text{val}} > k$	i_{ex}	i_{inh}
excitatory $w > 0$	true	$i_{\text{ex}} + w$	i_{inh}
	false	w	0
inhibitory $w < 0$	true	i_{ex}	$i_{\text{inh}} + w$
	false	0	w

FIGURE 11 | Ring buffer update algorithm. The algorithm is executed by the RB pipelines. The blue arrows indicate in- and out-going data items at different pipeline stages. The dashed box is for simplified illustration and shows the algorithm for the exceptional case of static synapses where a neuron's excitatory and inhibitory synaptic input can be lumped together. In all other cases, the algorithm expands according to the table on the upper right.

Even though RB update operations never address an RB segment that is processed in the current time step k, it may nevertheless happen that an RB write operation is not considered in further processing. This can be the case if a presynaptic data item represents a synapse with $d_{ij} = d_{\min}$, i.e., a 0.1 ms delay. The initiated update on the $k + 1$ RB segment may have no effect as it is already being fetched into the ODE pipeline for the next simulation step. In such a case, the ODE pipelines must be reset and restarted. This adds an additional latency L_{ODE} to the processing, where L_{ODE} denotes the pipeline depth. In the proposed design the ODE pipeline restart is software controlled. Whether a restart condition is indicated or not depends on the synaptic delay value and is encoded in the presynaptic data (see table in **Supplementary Figure S2**). This information is passed to the finite state machine that is controlling the ODE pipeline operation and considered when the next simulation step is initiated. Another read-before-write conflict arises in the RB pipeline itself, caused by BRAM read, write, and operation latencies. These must be taken into account if consecutive presynaptic data items initiate updates on the same RB entry. The reading of an entry for which a previous write operation has not yet completed will lead to a wrong synaptic input value. This problem may only arise with multapses. It can also be solved in software by rearranging the lists of synaptic targets in memory.

It is also worth mentioning that a ring buffer shares its read ports between the RB and ODE pipelines. We have investigated the impact of read contention on performance due to concurrent read operations. When not considering an asynchronous external spike input and long ODE pipeline iteration latencies, only an early arriving spike event may find the RB pipeline stalled when placing data in the RB FIFOs. The additional latency is minimal (in the order of a few clock cycles per simulation time step) and thus can be neglected.



		Clock Cycles	Description
L_{RD}		1	Ring buffer (RB), state variables buffer (SVB) and external input buffer (EI) initial read latency
L_{ODE}		12	Latency to fill the ODE pipeline
IL_N		64	ODE pipeline iteration latency, $N^P = 64$ neurons per processing unit
L_{SYNC}		3	Synchronization latency between simulation steps when no spike events occur
L_{SE}	L_{SEP}	4	Parallel spike events FIFO latency
	L_{SES}	2	Serialize and pack spike events
	L_{SEF}	2	Spike events FIFO latency
L_{IDS}	L_{IDSCAL}	4	Calculate memory read addresses and initiate PS to PL data transfer
	L_{IDSADR}	9	Read address channel latency (send memory read addresses)
L_{DS}		≈80/90/ 110	Measured per spike data stream latency (128 conn.) (value depends on f_{clk}: 100/150/ 200Mhz)
L_{RB}	L_{RBF}	2	RB FIFO latency
	L_{RBP}	5	RB pipeline latency
L^{SE}_{SYNC}		8	Synchronization latency between simulation steps when processing spike events

FIGURE 12 | Operating latencies. The scheduling of operations and the latencies associated with it, distinguishes two basic cases. **(A)** If no spike events occur, operation mainly reduces to ODE pipeline processing. **(B)** Normally, spike events have to be processed which changes and adds latencies. Postsynaptic spike events must be serialized, and for incoming presynaptic spike events the presynaptic data must be retrieved from external memory. **(C)** Table listing relevant latencies. The value of L_{DS} cannot be derived from the microarchitecture; its average value was determined using an external logic analyzer.

4.3.3. Operating Latencies

In order to further examine the design, we extracted the operating latencies from the microarchitecture VHDL implementation, or where that was not possible, measured them with an external logic analyzer. The timing diagrams and the table in **Figure 12** details the performance relevant operating latencies of the HNC node in clock cycles and show the timing of the operation scheduling.

At simulation start (and restart) the ODE pipelines are empty. An initial memory read operation that fetches the first data items, and the process of filling the ODE pipelines results in the latencies L_{RD} and L_{ODE}. This is illustrated in **Figure 12A** for the case of two simulation steps in which no spike events occur. The latency L_{ODE} corresponds to the depth of the ODE pipelines and may differ depending on the implemented model. The same holds for the iteration latency IL_N, which is the number of clock cycles required to process all $N^P = 64$ neurons assigned to a pipeline. At the end of a simulation step a few clock cycles L_{SYNC} are required for synchronization.

Spike events can occur in every clock cycle of the ODE pipeline operation, as depicted in **Figure 12B**. They are serialized and packed, resulting in a latency of L_{SE} (see also the table in **Figure 12C**). Before the presynaptic data can be read from external memory, its memory addresses have to be calculated. The latencies created by this process are summarized in L_{IDS}. The high-performance ports and the memory controller on the PS, as well as the external memory itself, determine the overall read access latency, and hence the value of L_{DS} as the data is streamed into the RB FIFOs by the *PS/PL Data Transfer Module*. The components involved are connected to different clock domains and contribute with latencies that are determined by the SoC technology rather than by the implemented user logic. We therefore measured the value as the number of PL clock cycles required for the transfer of a 1KiB data packet—the amount of data which is read from external memory upon the occurrence of a single spike event—for three PL clock frequencies $f_{clk} = 100/150/200$ MHz.

At the end of a simulation step in which spike events had to be processed, the RB pipelines might still be filled, and pending RB updates must be finalized. This adds the latencies summarized in L_{RB}. Finally, the HNC node goes into synchronization to prepare for the next simulation step. This requires a few clock cycles at the end of a simulation step compared to the situation where no spike event occurred. This adds the latency L^{SE}_{SYNC} to the processing.

In a multi-node system, the total latency would be extended by inter-node synchronization times. This is not explicitly included in the timing diagrams in **Figure 12** but indicated by the red barriers.

5. METHODS AND MATERIALS

5.1. Workload Model

The synchronous, time-driven neuron states update process (red arrows in **Figure 1**) generates a computational cost determined almost exclusively by the number of neurons processed by a single processing unit, and thus adds a constant operating latency. In contrast, the computational cost of the asynchronous, event-driven process of presynaptic data distribution and processing (blue arrows in **Figure 1**) depends mainly on the amount of presynaptic data to be processed and retrieved from external memory. The amount of data is determined by the average number of synapses on a node that a source neuron connects to C^{M}, as well as the total number of spike events processed by the node. For a given number of neurons per node N^{M}—which is a hardware design parameter and a constant—a certain number of nodes M is required to simulate a network of size N. The connection probability ϵ of the network determines the average in/out-degree $K = \epsilon N$, i.e., the number of in- and out-going synaptic connections of a neuron, which grows with the network size. Since the connections are distributed across the nodes, the average number of synapses on a node that a source neuron connects to remains constant for a given ϵ, even if the network size is growing. This is expressed by Equation (1).

$$C^{\mathrm{M}} = \epsilon N^{\mathrm{M}} = \frac{\epsilon N}{M} \tag{1}$$

Because C^{M} is a constant, the average amount of presynaptic data retrieved from external memory is consequently of same size for every spike event. It is therefore practical to consider as an indicator of computational workload the average number of spike events processed per simulation time step k:

$$\bar{\nu}_{\mathrm{k}} = N\bar{\nu}h, \ \text{ with } \ \bar{\nu} = \frac{1}{N}\sum_{\mathrm{N}} \frac{n_{\mathrm{sp}}(T)}{T} \tag{2}$$

$$\bar{\nu}_{\mathrm{k}} = \frac{h}{T}\sum_{\mathrm{N}} n_{\mathrm{sp}}(T) \tag{3}$$

where $\bar{\nu}$ is the average firing rate calculated over all neurons in the network, n_{sp} is a neuron's total spike count in the interval T, and h defines the temporal resolution, the step size, of the grid-based simulation, i.e., the time interval $h = \Delta t = t_{k+1} - t_k$. Note that this metric is initially independent of the number of neurons simulated.

5.2. Performance Model

We exploit knowledge of the HNC node microarchitecture latencies to derive a performance model that allows conclusions to be drawn about the performance characteristics in different scenarios regarding the workload and design and technology parameters. We make the following assumptions that represent a scenario that maximally challenges the hardware:

- *All neurons have at least one target connection with a synaptic delay value $d_{ij} = d_{\mathrm{min}}$.*
  Every spike event will initiate an ODE pipeline restart. This adds the latencies L_{RD} and L_{ODE} (**Figure 12**) to every simulation step.
- *Spike events are distributed uniformly across the neurons in an ODE pipeline and over pipeline iterations.*
  We assume that the expected value for the timing of a spike event is the middle of an ODE pipeline iteration, i.e., at $IL_{\mathrm{N}}/2$. This is justified by the two-population Izhikevich network used for the benchmarking (Sec. 5.3), and the placement of the neurons on the processing units.
- *All lists of synaptic target connections are the same length.*
  This is justified by the current design (Section 4.3.1). Upon every spike event, a 1KiB data packet is transferred from external memory to the RB FIFOs.

As explained in the previous section, we take the average number of spike events $\bar{\nu}_{\mathrm{k}}$ processed in a single simulation step k as a measure of the workload. The time span to perform a single simulation step becomes minimal if no spike events occur, and is predominantly determined by the number of serially processed neurons assigned to an ODE pipeline. This is reflected in the ODE pipeline iteration latency IL_{N}. Together with the synchronization latency L_{SYNC}, it sets the upper bound for the single-node acceleration factor $F_{\mathrm{S}}^{\mathrm{MAX}}$ at a given clock frequency f_{clk}. From the timing diagram in **Figure 12A** we derive:

$$F_{\mathrm{S}}^{\mathrm{MAX}} = \frac{khf_{\mathrm{clk}}}{L_{\mathrm{RD}} + L_{\mathrm{ODE}} + k(IL_{\mathrm{N}} + L_{\mathrm{SYNC}})} \tag{4}$$

where k denotes the number of simulation steps, and h specifies the temporal resolution of the simulation, i.e., $h = \Delta t = 0.1$ ms. For $k \gg 1$ this simplifies to

$$F_{\mathrm{S}}^{\mathrm{MAX}} = \frac{hf_{\mathrm{clk}}}{L_{\Sigma}} \tag{5}$$

where $L_{\Sigma} = IL_{\mathrm{N}} + L_{\mathrm{SYNC}}$. Analogously to L_{Σ}, which summarizes the processing latencies for the non-spiking case, latencies arising from processing spiking events can be summarized according to the timing diagrams and process scheduling shown in **Figures 12A,B**. This consists of the sum of the latencies for the spike events serialization and buffering process ($L_{\mathrm{SE}} = L_{\mathrm{SEP}} + L_{\mathrm{SES}} + L_{\mathrm{SEF}}$), the latencies incurred by the initiation of the data streams S1 and S2 ($L_{\mathrm{IDS}} = L_{\mathrm{IDSCAL}} + L_{\mathrm{IDSADR}}$), see **Figure 9**, and the latencies resulting from the processing of outstanding presynaptic data items at the end of a simulation step ($L_{\mathrm{RB}} = L_{\mathrm{RBF}} + L_{\mathrm{RBP}}$). The number of clock cycles for each latency, as well as its description, can be found in **Figure 12C**. Altogether, this results in

$$L_{\Sigma}^{\mathrm{SE}} = L_{\mathrm{RD}} + L_{\mathrm{ODE}} + \frac{IL_{\mathrm{N}}}{2} + L_{\mathrm{SE}} + L_{\mathrm{IDS}} + L_{\mathrm{RB}} + L_{\mathrm{SYNC}}^{\mathrm{SE}} \tag{6}$$

The term $IL_{\mathrm{N}}/2$ in Equation (6) reflects the assumption of a uniform distribution of the spike events.

For an isolated node with no inter-node communication, the acceleration factor as a function of the average number of spike events per simulation step can now be formulated as follows:

$$F_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}}) = \begin{cases} \dfrac{hf_{\mathrm{clk}}}{\bar{\nu}_{\mathrm{k}}(L_{\Sigma}^{\mathrm{SE}} + L_{\mathrm{DS}}) + (1-\bar{\nu}_{\mathrm{k}})L_{\Sigma}} & \text{if } \bar{\nu}_{\mathrm{k}} < 1 \\ \dfrac{hf_{\mathrm{clk}}}{L_{\Sigma}^{\mathrm{SE}} + \bar{\nu}_{\mathrm{k}} L_{\mathrm{DS}}} & \text{otherwise} \end{cases} \tag{7}$$

For $\bar{\nu}_{\mathrm{k}} < 1$, the denominator in Equation (7) consists of two terms corresponding to the spiking $[\bar{\nu}_{\mathrm{k}}(L_{\Sigma}^{\mathrm{SE}} + L_{\mathrm{DS}})]$ and the non-spiking $((1-\bar{\nu}_{\mathrm{k}})L_{\Sigma})$ case, where L_{DS} denotes the per spike event data stream latency. The two branches are equal for $\bar{\nu}_{\mathrm{k}} = 1$. In the absence of spike events, $F_{\mathrm{S}}(0) = F_{\mathrm{S}}^{\mathrm{MAX}}$ applies (see Equation 5).

Please note that Equation (7) does not consider C^{M} - the average number of synapses on a node that a source neuron connects to. The value of C^{M} determines the value of L_{DS} that was measured since it cannot be derived from the microarchitecture. This becomes relevant, for example, when the number of neurons per node N^{M}, and thus also C^{M} changes (Equation 1). Furthermore, it neglects the possibility that a presynaptic data transfer could complete before all neurons are processed, i.e., $IL_{\mathrm{N}}/2 - L_{\mathrm{DS}} > 0$. To account for this, the value of L_{Σ}^{SE} would have to be corrected by adding $IL_{\mathrm{N}}/2 - L_{\mathrm{DS}}$. However, one would only see an effect at very low spike rates because $\bar{\nu}_{\mathrm{k}} L_{\mathrm{DS}} \gg IL_{\mathrm{N}}/2 - L_{\mathrm{DS}}$ applies. Equation (7), therefore, represents a good estimate of the acceleration factors that can be achieved with the proposed HNC node design under different workloads.

Currently, only a single-node prototype exists. In order to estimate the performance characteristics of a multi-node system, we expand the performance model to include inter-node communication latencies. Strongly simplifying the complex effects of communication network topologies, protocols, and low-latency interconnects, we propose three basic assumptions:

- Spike events are broadcasted, i.e, communicated to all nodes.
- Inter-node connections all have the same and fixed transmission latency time T_{COM}, which adds to every simulation step. In addition to the times needed to communicate the spike events between nodes, T_{COM} also includes inter-node synchronization latencies, i.e., barrier messaging times.
- To take into account that inter-node communication increase with workload, every spike event adds a transmission latency to the communication, i.e., a variable, workload dependent portion defined as a small fraction of the transmission latency time. It is specified by a factor α and results for a given workload in $\bar{\nu}_{\mathrm{k}} \alpha T_{\mathrm{COM}}$.

Adding inter-node communication latencies to Equation (7) results in

$$F_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}}) = \begin{cases} \dfrac{hf_{clk}}{\bar{\nu}_{\mathrm{k}}(L_{\Sigma}^{\mathrm{SE}} + L_{\mathrm{DS}} + \alpha L_{\mathrm{COM}}) + (1-\bar{\nu}_{\mathrm{k}})L_{\Sigma} + L_{\mathrm{COM}}} & \text{if } \bar{\nu}_{\mathrm{k}} < 1 \\ \dfrac{hf_{clk}}{L_{\Sigma}^{\mathrm{SE}} + \bar{\nu}_{\mathrm{k}}(L_{\mathrm{DS}} + \alpha L_{\mathrm{COM}}) + L_{\mathrm{COM}}} & \text{otherwise} \end{cases} \tag{8}$$

where L_{COM} denotes the transmission latency in PL clock cycles derived from the transmission latency time, i.e., $L_{\mathrm{COM}} = f_{\mathrm{clk}} T_{\mathrm{COM}}$. Note that even in the absence of spike events, L_{COM} does not vanish as it includes inter-node synchronization times. According to Equation (4), the upper bound for the acceleration factor with inter-node communication then becomes:

$$F_{\mathrm{C}}^{\mathrm{MAX}} = \frac{hf_{\mathrm{clk}}}{L_{\Sigma} + L_{\mathrm{COM}}} \tag{9}$$

From the performance characteristics derived above, the total relative performance loss P_{TOT} with respect to the maximum achievable acceleration can be estimated for different workloads as follows:

$$P_{\mathrm{TOT}}(\bar{\nu}_{\mathrm{k}}) = P_{\mathrm{S}} + P_{\mathrm{C}} = \left(1 - \frac{F_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}})}{F_{\mathrm{S}}^{\mathrm{MAX}}}\right) \cdot 100\%. \tag{10}$$

The total performance loss can be further subdivided into the losses caused by the HNC node-local spike processing (which mainly consists of retrieving and distributing the presynaptic data)

$$P_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}}) = \left(1 - \frac{F_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}})}{F_{\mathrm{S}}^{\mathrm{MAX}}}\right) \cdot 100\% \tag{11}$$

and the loss caused by the inter-node communication

$$P_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}}) = \frac{F_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}}) - F_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}})}{F_{\mathrm{S}}^{\mathrm{MAX}}} \cdot 100\%. \tag{12}$$

5.3. Verification, Validation, and Benchmarking Model: Two-Population Izhikevich Network

We use a simple two-population model as the basis for both the performance measurements and the verification and validation of the correctness of the HNC node hardware and software implementation. The network consists of 1,000 Izhikevich-type neurons (Izhikevich, 2003), which follow the dynamics

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + i_{\mathrm{syn}}(t) + i_{\mathrm{ext}}(t), \text{ with } i_{\mathrm{syn}}(t) = i_{\mathrm{ex}} + i_{\mathrm{inh}} \tag{13}$$

$$\frac{du}{dt} = a(bv - u) \tag{14}$$

$$\text{if } v \geq 30\text{mV, then} \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{15}$$

The network consists of 800 excitatory regular spiking neurons $\big[(a, b, c, d) = (0.02, 0.2, -65.0, 8.0)\big]$ and 200 inhibitory fast spiking neurons $\big[(a, b, c, d) = (0.1, 0.2, -65.0, 2.0)\big]$. The excitatory population makes random connections to the inhibitory population and to itself. The inhibitory population only projects to the excitatory population. All neurons in the network draw their connections with a fixed in-degree of $K_{\text{in}} = 100$ and receive additional input from an external source. A detailed description of the network is given in the **Supplementary Material**.

The choice of this model was motivated by our previous work, where we subjected a two-population Izhikevich network implementation on the SpiNNaker system to a rigorous verification and validation task (Gutzen et al., 2018; Trensch et al., 2018).

6. DISCUSSION

We presented an SoC-based hybrid software-hardware architecture of a neuromorphic computing node. This is to be seen as a complementary yet distinct approach to the neuromorphic developments aiming at brain-inspired and highly efficient novel computer architectures for solving real-world tasks. The requirements for achieving reproducible hyper-real-time neuroscience simulations are different, so also the technical challenges. We examined the extent to which the proposed architecture and Xilinx Zynq SoC device technology is capable of meeting the high demands of modeling and simulation in neuroscience in terms of flexibility, accuracy, and simulation performance.

6.1. Flexibility

The HNC node design exploits the trade-off between flexibility and efficiency offered by the Xilinx Zynq SoC device technology. The tight coupling of programmable logic with a general purpose processor gives the developer the flexibility to cope with rapid developments in neuroscience and changing requirements. For example, the plethora of neuron and synapse models require that the operations and their scheduling performed by the ODE pipelines can be adapted in terms of the implemented numerical algorithms and data types. The application of code generation techniques (Blundell et al., 2018a) can abstract hardware implementation details away from a neuron and synapse modeling task. Therefore, the ODE pipeline architecture was implemented as a replaceable VHDL-module having a defined port interface. This makes the neuron and synapse model hardware implementations accessible to tools, such as NESTML (Plotnikov et al., 2016). By this means a wide variety of neuron and synapse models can be supported.

The availability of powerful, node-local processor cores also allows us to decentralize; moving tasks onto the neuromorphic compute nodes that are typically carried out on a host system. For example, the generation of the network connectivity could be carried out on a conventional system using established tools, such as PyNN (Davison et al., 2009) or PyNEST (Eppler et al., 2009), while the network instantiation process is parallelized by being delegated to the processor cores of the neuromorphic compute nodes. This would reduce network building times, especially when repeated simulations are performed (e.g., parameter scans). Moreover, the integration with the existing workflows for neural network modeling and simulation becomes easier to reach.

The HNC node architecture is open for extension, for example, the implementation of synaptic plasticity rules. Although plasticity models were deliberately left out for the current HNC node prototype, it was considered in the design decisions. In future developments, we intend to exploit the hybrid software-hardware architecture concept of the HNC node in such a way that plasticity algorithms programmed in software run on a dedicated plasticity processor—executed on the APU using the second, so far unused, ARM processor core—supported by accelerators implemented in programmable logic. To enable the implementation of spike-based plasticity rules (Morrison et al., 2008), the network connectivity data as well as the recorded spike events are stored in the external memory, thus keeping synaptic weights adjustable and spike history accessible to the processor cores. There are a number of different forms of plasticity (Magee and Grienberger, 2020) and a rapid development in the field which entails some technical challenges. The HNC node provides here a flexible platform as a means to explore novel architecture concepts to implement plasticity algorithms.

6.2. Numerical Precision

Particular care must be taken with respect to mathematical operations. Both the choice of data types and algorithms as well as their technical implementation require special attention. The design decisions made regarding the example Izhikevich neuron model ODE pipeline implementation (see Section 4 in the **Supplementary Material**), e.g., the data types and the numeric integration scheme, are based on the results of our earlier studies (Gutzen et al., 2018; Trensch et al., 2018). By conducting a calculation verification task[15], we concluded that a 32-bit signed fixed-point data type (s16.15) does not provide the necessary numerical precision to capture the dynamics of the Izhikevich neuron model (Izhikevich, 2003) with sufficient accuracy. For the processing unit's ODE pipelines, we therefore implemented a 40-bit signed fixed-point data type (s16.23)—a decision also made to avoid expensive floating point operations. In combination with an explicit Forward Euler ODE solver method and an integration step size of $h = 0.1$ ms, we achieve sufficient accuracy—even though it is the simplest numerical method available. Analogously to the calculation verification task carried out in the studies mentioned, we verified the ODE pipeline operation by comparing the subthreshold dynamics and the spike timing to the results of an explicit Runge-Kutta-Fehlberg(4, 5) method with an absolute integration error of 10^{-6}.

6.3. Verification of Implementations

During implementation, hardware and software components cannot be considered independently of each other and must therefore be developed in parallel in a co-development process. The HNC node software system is written in C and almost all hardware components were developed in VHDL. In contrast to a high-level synthesis approach, where a hardware design is formulated at an algorithmic level in the C language, for example, and the synthesis tool chain generates a reliable hardware

[15] Calculation verification tasks assess the level of error that arises from various sources of error in numerical simulations as well as to identify and remove them (Thacker et al., 2004).

description from it, the implementation in VHDL at the RTL level is rather error-prone. A well thought-out test strategy is therefore essential. It must consider the verification of the correctness of the technical implementation of the hardware and software components as well as the validation of the outcome of the simulations performed on the HNC node.

Our approach was that of an embedded hardware-software co-verification, in the sense of a directed software-controlled functional testing. For this purpose, the hardware components under test were connected to the APU of the SoC device through memory-mapped AXI-interfaces and subjected to a series of hierarchical functional tests written in C. These tests range from simple to complex and are executed on the APU. They include basic hardware and software functional tests, integration tests, as well as complex functional tests that also became part of the HNC node software system. Examples of such complex tests are memory read-write pattern tests. They ensure the correct implementation and operation of the DMA data transfer to and from the SVBs and verify data type and endianness conversion. Another example of a complex test scenario is the functional verification of the RB pipeline and RB buffer operation, where a software-controlled spike injection and a subsequent RB read out is used to verify the correctness of the presynaptic data processing.

6.4. Performance

Software developers of spiking neural network simulation tools invest much effort in the optimization of their codes to achieve best possible performance and simulation efficiency. They are well aware of the performance-critical nature of retrieving the presynaptic data from memory and its distribution, its accumulation in the ring buffers, and the update process of the neuron and synapse model dynamics performed at every simulation time step. The challenges in finding optimal solutions and implementations are manifold. For example, in large-scale networks, synaptic processing substantially dominates the computational load, and the irregular, random access pattern in retrieving the presynaptic data reduce a processor's cache hit rate and increases data access latencies (see e.g., Kunkel et al., 2014). The tools of trade here are algorithms that implement high parallelism in computations, "cache-friendly" data structures, and the application of techniques for latency hiding, such as data prefetching (Pronold et al., 2022). The proposed HNC node design aims to address these problems—which on conventional computer architectures are a consequence of the von Neumann bottleneck—by implementing performance-critical tasks in hardware. Specifically, the process of neuron and synapse model update benefits from the data-locality of state variables stored in fast on-chip BRAM memories. Storing the network connectivity data in an external memory, however, undermines this concept, and toward higher workloads, performance will be bound by external memory access latency. For larger systems and higher workloads, it is therefore crucial to aim for an architecture design that also allows data-locality for the presynaptic data processing. The design of the HNC node is constrained in this respect by the limited BRAM resources.

The ability to model the performance behavior for different design parameters is of great value as it can guide future developments and design decisions. We developed such a model for the HNC node architecture. The implementation strategy, based on the hardware description on register-transfer level (RTL) in the VHDL language has allowed us to derive an accurate performance model from the implemented microarchitecture. To this end we made several simplifying assumptions, in particular, with respect to the inter-node communication latencies. Network technologies are typically optimized for throughput, but not for latency. The value of the transmission latency time ($T_{\mathrm{COM}} = 500$ ns) assumed for the performance evaluation is already ambitious. However, low-latency inter-node communication is as important for performance as data-locality is for the computations. Despite these simplifications, the model achieves a good approximation of the performance characteristics. Extrapolating from the single node performance, we predict that small clusters capable of simulating in hyper-real-time networks comprising a few tens of thousands of neurons would achieve acceleration factors in the order of 10 to 50.

6.5. Cluster Operation

Although cluster operation is not the focus of this article, some related considerations that influenced design decisions are worth mentioning. Three communication bottlenecks can be identified in the simulation flow that are relevant to the overall performance of a cluster system: the spike exchange between nodes, inter-node synchronization, and external communication for system configuration and operation including the unload of recorded simulation data. The requirements of these tasks differ in regard to latency and bandwidth. Inter-node spike communication and node synchronization require an ultra-low latency interconnect but not high bandwidth. The demand for external communication is completely different. For loading and unloading larger amounts of data, high bandwidth is desirable to achieve low system setup times and eventually real-time recording capability. We are therefore aiming at three different solutions tailored to the respective task, although our cluster concept is not yet fully developed. The HNC node encodes spike events using an *Address Event Representation* (AER; Mahowald, 1992). AER-based communication is well established in neuromorphic computing and the basis for low-latency spike-communication. In order to achieve the predicted cluster performance (cf. Section 3.2), it is crucial that the transmission latency time of $T_{\mathrm{COM}} = 500$ ns for inter-node spike communication assumed by the performance model can be attained in a cluster consisting of a few tens of HNC nodes. The Xilinx Zynq SoC device used for the implementation of the HNC node prototype provides various hardware interfaces that would allow us to establish an efficient chip-to-chip communication, for example, a number of serial gigabit transceivers (GTX/GTH), PCI Express, and low-voltage differential signaling (LVDS) user I/Os (Xilinx, 2021). A solution for a low-latency spike communication in a 64-node FPGA cluster is, for example, presented in Moore et al. (2012). It exploits high-speed serial links and achieves a hop-latency of 50 ns in a 3D torus topology. For inter-node synchronization, we favor a simple one-wire (e.g., wired-or) solution where a global barrier signal is derived from the intra-node synchronization logic (cf. **Figure 6**). External communication with the HNC node is established using a 10/100/1000 Mb/s tri-speed Ethernet PHY and the TCP protocol—currently used only to stream the recorded simulation data to a host system. For a cluster, we aim at a *parallel data move* solution, in which each HNC node is connected to its own host system or host node, respectively.

The proposed technology and architecture is an ideal basis for prototyping and design space exploration—the primary domain of programmable logic devices—and for elaborating novel architectures. The reconfigurable logic allows extensive freedom in the implementation of the numerical models while the processor cores opens an elegant way to achieve system integration. They can be an intermediate step toward next-generation neuromorphic systems and neuroscience simulation platforms. In this sense, the proposed HNC node design complements the existing neuromorphic system architecture approaches of SpiNNaker and BrainScales, in regards both to technology and the trade-off between flexibility and efficiency.

DATA AVAILABILITY STATEMENT

The simulation scripts and source codes used in this work to demonstrate correctness are available online at: https://github.com/gtrensch/RigorousNeuralNetworkSimulations (doi: 10.5281/zenodo.6591552).

AUTHOR CONTRIBUTIONS

GT developed the System-on-Chip based hybrid architecture and implemented the prototype, developed the workload and performance model, and performed the experiments. GT and AM designed the experiments and wrote the paper. All authors contributed to the article and approved the submitted version.

FUNDING

This project has received funding from the Helmholtz Association's Initiative and Networking Fund under project number SO-092 (Advanced Computing Architectures, ACA). Open access publication funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—491111487.

ACKNOWLEDGMENTS

We are grateful to Tobias Noll, Georgia Psychou, Eqbal Maraqa, Tom Tetzlaff, and Michael Schiek for the fruitful discussions throughout the project. We would especially like to thank Arne Heittmann for his comments on an earlier version of the manuscript and to our colleagues in the Advanced Computing Architectures (ACA) Project and the Simulation and Data Laboratory Neuroscience for continuous collaboration.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2022.884033/full#supplementary-material

REFERENCES

Akar, N. A., Cumming, B., Karakasis, V., Kusters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor-a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia: IEEE), 274–282. doi: 10.1109/EMPDP.2019.8671560

Arm Limited (2021). *AMBA AXI and ACE Protocol Specification*. Arm Limited. Available online at: www.arm.com.

Blundell, I., Brette, R., Cleland, T. A., Close, T. G., Coca, D., Davison, A. P., et al. (2018a). Code generation in computational neuroscience: a review of tools and techniques. *Front. Neuroinform*. 12:68. doi: 10.3389/fninf.2018.00068

Blundell, I., Plotnikov, D., Eppler, J. M., and Morrison, A. (2018b). Automatically selecting an optimal integration scheme for systems of differential equations in neuron models. *Front. Neuroinform*. 12:50 doi: 10.3389/fninf.2018.00050

Braitenberg, V., and Schüz, A. (1998). *Cortex: Statistics and Geometry of Neuronal Connectivity*. Berlin; Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-662-03733-1

Cheung, K., Schultz, S. R., and Luk, W. (2016). NeuroFlow: a general purpose spiking neural network simulation platform using customizable processors. *Front. Neurosci*. 9:516. doi: 10.3389/fnins.2015.00516

Dasbach, S., Tetzlaff, T., Diesmann, M., and Senk, J. (2021). Dynamical characteristics of recurrent neuronal networks are robust against low synaptic weight resolution. *Front. Neurosci*. 15:757790. doi: 10.3389/fnins.2021.757790

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform*. 2:11. doi: 10.3389/neuro.11.011.2008

Eppler, J., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2009). PyNEST: a convenient interface to the nest simulator. *Front. Neuroinform*. 2:12. doi: 10.3389/neuro.11.012.2008

Fardet, T., Vennemo, S. B., Mitchell, J., Mòrk, H., Graber, S., Hahne, J., et al. (2020). Nest 2.20.1. *Zenodo*.

Friedmann, S., Schemmel, J., Grübl, A., Hartel, A., Hock, M., and Meier, K. (2017). Demonstrating hybrid learning in a flexible neuromorphic hardware system. *IEEE Trans. Biomed. Circuits Syst*. 11, 128–142. doi: 10.1109/TBCAS.2016.2579164

Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., et al. (2013). Overview of the spinnaker system architecture. *IEEE Trans. Comput*. 62, 2454–2467. doi: 10.1109/TC.2012.142

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., et al. (2010). NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comput. Biol*. 6:e1000815. doi: 10.1371/journal.pcbi.1000815

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Front. Neuroinform*. 2:5. doi: 10.3389/neuro.11.005.2008

Gutzen, R., von Papen, M., Trensch, G., Quaglio, P., Grün, S., and Denker, M. (2018). Reproducible neural network simulations: Statistical methods for model validation on the level of network activity data. *Front. Neuroinform*. 12:90. doi: 10.3389/fninf.2018.00090

Hansel, D., Mato, G., Meunier, C., and Neltner, L. (1998). On numerical simulations of integrate-and-fire neural networks. *Neural Comput*. 10, 467–483. doi: 10.1162/089976698300017845

Heittmann, A., Psychou, G., Trensch, G., Cox, C. E., Wilcke, W. W., Diesmann, M., et al. (2022). Simulating the cortical microcircuit significantly faster than real time on the IBM INC-3000 neural supercomputer. *Front. Neurosci*. 15:728460. doi: 10.3389/fnins.2021.728460

Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput*. 9, 1179–1209. doi: 10.1162/neco.1997.9.6.1179

Hines, M. L., and Carnevale, N. T. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Comput*. 12, 995–1007. doi: 10.1162/089976600300015475

Izhikevich, E. M. (2003). Simple model of spiking neurons. *Trans. Neur. Netw*. 14, 1569–1572. doi: 10.1109/TNN.2003.820440

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci*. 1, 136–142. doi: 10.1038/s43588-020-00022-7

Kunkel, S., Schmidt, M., Eppler, J. M., Plesser, H. E., Masumoto, G., Igarashi, J., et al. (2014). Spiking network simulation code for petascale computers. *Front. Neuroinform*. 8:78. doi: 10.3389/fninf.2014.00078

Magee, J. C., and Grienberger, C. (2020). Synaptic plasticity forms and functions. *Annu. Rev. Neurosci*. 43, 95–117. doi: 10.1146/annurev-neuro-090919-022842

Maguire, L., McGinnity, T., Glackin, B., Ghani, A., Belatreche, A., and Harkin, J. (2007). Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing* 71, 13–29. doi: 10.1016/j.neucom.2006.11.029

Mahowald, M. (1992). *VLSI analogs of neuronal visual processing: a synthesis of form and function* (Ph.D. thesis). Califpionia Institute of Technology, Pasadella, CA, United States.

Moore, S. W., Fox, P. J., Marsh, S. J., Markettos, A. T., and Mujumdar, A. (2012). "Bluehive - A field-programable custom computing machine for extremescale real-time neural network simulation," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines* (Toronto, ON: IEEE), 133-140. doi: 10.1109/FCCM.2012.32

Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biol. Cybern*. 98, 459–478. doi: 10.1007/s00422-008-0233-1

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Comput*. 17, 1776–1801. doi: 10.1162/0899766054026648

Morrison, A., Straube, S., Plesser, H. E., and Diesmann, M. (2007). Exact subthreshold integration with continuous spike times in discrete time neural network simulations. *Neural Comput*. 19, 47–79. doi: 10.1162/neco.2007.19.1.47

Narayanan, P., Cox, C. E., Asseman, A., Antoine, N., Huels, H., Wilcke, W. W., et al. (2020). Overview of the IBM neural computer architecture. *arXiv:2003.11178 [cs]*. *arXiv: 2003.11178*. doi: 10.48550/ARXIV.2003.11178

Noll, T. G., von Sydow, T., Neumann, B., Schleifer, J., Coenen, T., and Kappen, G. (2010). "Chapter 2: Reconfigurable components for application-specific processor architectures," in *Dynamically Reconfigurable Systems*, eds M. Platzner, J. Teich, and N. Wehn (Heidelberg: Springer), 25–49. doi: 10.1007/978-90-481-3485-4_2

Pani, D., Meloni, P., Tuveri, G., Palumbo, F., Massobrio, P., and Raffo, L. (2017). An FPGA platform for real-time simulation of spiking neuronal networks. *Front. Neurosci*. 11:90. doi: 10.3389/fnins.2017.00090

Pauli, R., Weidel, P., Kunkel, S., and Morrison, A. (2018). Reproducing polychronization: a guide to maximizing the reproducibility of spiking network models. *Front. Neuroinform*. 12:46. doi: 10.3389/fninf.2018.00046

Pehle, C., Billaudelle, S., Cramer, B., Kaiser, J., Schreiber, K., Stradmann, Y., et al. (2022). The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity. *Front. Neurosci*. 16:795876. doi: 10.3389/fnins.2022.795876

Pfeil, T., Potjans, T., Schrader, S., Potjans, W., Schemmel, J., Diesmann, M., et al. (2012). Is a 4-bit synaptic weight resolution enough? - constraints on enabling spike-timing dependent plasticity in neuromorphic hardware. *Front. Neurosci*. 6:90. doi: 10.3389/fnins.2012.00090

Plotnikov, D., Blundell, I., Ippen, T., Eppler, J. M., Morrison, A., and Rumpe, B.(2016). "NESTML: a modeling language for spiking neurons," in *Modellierung 2016* (Karlsruhe), 93–108.

Potjans, T. C., and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Pronold, J., Jordan, J., Wylie, B. J. N., Kitayama, I., Diesmann, M., and Kunkel, S. (2022). Routing brain traffic through the von neumann bottleneck: Parallel sorting and refactoring. *Front. Neuroinform*. 15:785068. doi: 10.3389/fninf.2021.785068

Schemmel, J., Brüderle, D., Grübl, A., Hock, M., Meier, K., and Millner, S. (2010). "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems* (Paris: IEEE), 1947–1950. doi: 10.1109/ISCAS.2010.5536970

Schemmel, J., Kriener, L., Muller, P., and Meier, K. (2017). "An accelerated analog neuromorphic hardware system emulating NMDA- and calcium based non-linear dendrites," in *2017 International Joint Conference on Neural Networks* (Anchorage, AK), 2217–2226. doi: 10.1109/IJCNN.2017.7966124

Thacker, B. H., Doebling, S. W., Hemez, F. M., Anderson, M. C., Pepin, J. E., and Rodriguez, E. A. (2004). *Concepts of Model Verification and Validation*. Los Alamos, NM: Los Alamos National Lab.

Trensch, G., Gutzen, R., Blundell, I., Denker, M., and Morrison, A. (2018). Rigorous neural network simulations: a model substantiation methodology for increasing the correctness of simulation results in the absence of experimental validation data. *Front. Neuroinform*. 12:81. doi: 10.3389/fninf.2018.00081

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware spinnaker and the neural network simulation software nest for a full-scale cortical microcircuit model. *Front. Neurosci*. 12:291. doi: 10.3389/fnins.2018.00291

Wang, R., Hamilton, T. J., Tapson, J., and van Schaik, A. (2014). "An FPGA design framework for large-scale spiking neural networks," in *2014 IEEE International Symposium on Circuits and Systems* (Melbourne, VIC: IEEE), 457–460. doi: 10.1109/ISCAS.2014.6865169

Wang, R. M., Thakur, C. S., and van Schaik, A. (2018). An FPGA-based massively parallel neuromorphic cortex simulator. *Front. Neurosci*. 12:213. doi: 10.3389/fnins.2018.00213

Xilinx (2019b). *Embedded System Tools Reference Manual v2019.2 (UG1043)*. Available online at: www.xilinx.com (accessed January 13, 2022).

Xilinx (2019c). *Vivado Design Suite User Guide High-Level Synthesis v2019.1 (UG902)*. Available online at: www.xilinx.com (accessed January 13, 2022).

Xilinx (2019d). *Vivado Design Suite User Guide v2019.1 (UG893)*. Available online at: www.xilinx.com (accessed January 13, 2022).

Xilinx (2019e). *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC User Guide (UG945)*. Available online at: www.xilinx.com (accessed January 13, 2022).

Xilinx (2021). *Zynq-7000 SoC Technical Reference Manual (UG585)*. Available online at: www.xilinx.com.

Xilinx. AXI DMA v7.1 LogiCORE IP Product Guide (2019a). Available online at: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf (accessed January 13, 2022).

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Neuroinformatics

ORIGINAL RESEARCH
published: 04 July 2022
doi: 10.3389/fninf.2022.883333

Fast Simulation of a Multi-Area Spiking Network Model of Macaque Cortex on an MPI-GPU Cluster

Gianmarco Tiddia[1,2], Bruno Golosio[1,2]*, Jasper Albers[3,4], Johanna Senk[3], Francesco Simula[5], Jari Pronold[3,4], Viviana Fanti[1,2], Elena Pastorelli[5], Pier Stanislao Paolucci[5] and Sacha J. van Albada[3,6]

[1] Department of Physics, University of Cagliari, Monserrato, Italy, [2] Istituto Nazionale di Fisica Nucleare (INFN), Sezione di Cagliari, Monserrato, Italy, [3] Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6) and JARA-Institute Brain Structure-Function Relationships (INM-10), Jülich Research Centre, Jülich, Germany, [4] RWTH Aachen University, Aachen, Germany, [5] Istituto Nazionale di Fisica Nucleare (INFN), Sezione di Roma, Rome, Italy, [6] Faculty of Mathematics and Natural Sciences, Institute of Zoology, University of Cologne, Cologne, Germany

Spiking neural network models are increasingly establishing themselves as an effective tool for simulating the dynamics of neuronal populations and for understanding the relationship between these dynamics and brain function. Furthermore, the continuous development of parallel computing technologies and the growing availability of computational resources are leading to an era of large-scale simulations capable of describing regions of the brain of ever larger dimensions at increasing detail. Recently, the possibility to use MPI-based parallel codes on GPU-equipped clusters to run such complex simulations has emerged, opening up novel paths to further speed-ups. NEST GPU is a GPU library written in CUDA-C/C++ for large-scale simulations of spiking neural networks, which was recently extended with a novel algorithm for remote spike communication through MPI on a GPU cluster. In this work we evaluate its performance on the simulation of a multi-area model of macaque vision-related cortex, made up of about 4 million neurons and 24 billion synapses and representing 32 mm^2 surface area of the macaque cortex. The outcome of the simulations is compared against that obtained using the well-known CPU-based spiking neural network simulator NEST on a high-performance computing cluster. The results show not only an optimal match with the NEST statistical measures of the neural activity in terms of three informative distributions, but also remarkable achievements in terms of simulation time per second of biological activity. Indeed, NEST GPU was able to simulate a second of biological time of the full-scale macaque cortex model in its metastable state 3.1× faster than NEST using 32 compute nodes equipped with an NVIDIA V100 GPU each. Using the same configuration, the ground state of the full-scale macaque cortex model was simulated 2.4× faster than NEST.

Keywords: computational neuroscience, spiking neural networks, simulations, GPU (CUDA), primate cortex, multi-area model of cerebral cortex, message passing interface (MPI), high performance computing (HPC)

OPEN ACCESS

Edited by:
James B. Aimone,
Sandia National Laboratories (DOE),
United States

Reviewed by:
Fred H. Hamker,
Chemnitz University of Technology,
Germany
Georgios Detorakis,
Independent Researcher, Irvine, CA,
United States

****Correspondence:***
Bruno Golosio
golosio@unica.it

Received: *24 February 2022*
Accepted: *02 June 2022*
Published: *04 July 2022*

Citation:
Tiddia G, Golosio B, Albers J, Senk J, Simula F, Pronold J, Fanti V, Pastorelli E, Paolucci PS and van Albada SJ (2022) Fast Simulation of a Multi-Area Spiking Network Model of Macaque Cortex on an MPI-GPU Cluster.
Front. Neuroinform. 16:883333.
doi: 10.3389/fninf.2022.883333

1. INTRODUCTION

Large-scale spiking neural networks are of growing research interest because of their ability to mimic brain dynamics and function more and more accurately. However, the task of accurately simulating natural neural networks is arduous: the human brain contains around 86 × 10^9 neurons (Azevedo et al., 2009) and on the order of 10^4–10^5 synapses per neuron in the cerebral cortex (Cragg, 1975; Alonso-Nanclares et al., 2008). Moreover, the brain presents a plethora of different neurotransmitters, receptors, and neuron types connected with specific probabilities and patterns. For these reasons, even a simulation of a small fraction of the brain could be computationally prohibitive if the details of axonal and dendritic arborizations were accounted for or if, adding further complexity, accurate descriptions of biochemical processes were included. In this work, focused on enabling the simulation of multi-area cortical models on up to a few tens of compute nodes, we treat spiking simulations with point neurons and simplified synaptic rules. This level of abstraction greatly reduces computational demands while still capturing essential aspects of the neural network behavior. However, achieving short simulation times for a multi-area spiking network model is nevertheless nontrivial even on high-performance hardware with highly performant software tools.

Some simulators such as NEST (Hahne et al., 2021), NEURON (Carnevale and Hines, 2006), Brian 2 (Stimberg et al., 2019) and ANNarchy (Vitay et al., 2015) are capable of simulating a large variety of neuron and synapse models. These simulators support multithreaded parallel execution on general-purpose CPU-based systems. Furthermore, NEST and NEURON also support distributed computing *via* MPI.

Meanwhile in the last decades, to efficiently simulate large-scale neural networks in terms of both speed and energy consumption, neuromorphic hardware has been developed by taking inspiration from brain architecture. Among these systems, we can mention Loihi (Davies et al., 2018) and TrueNorth (Akopyan et al., 2015), which are entering the realm of large-scale neural network simulations, and BrainScaleS (Grübl et al., 2020), which is based on analog emulations of simplified models of spiking neurons and synapses, with digital connectivity. The system enables energy-efficient neuronal network simulations, offering highly accelerated operations. Another promising project in this field is SpiNNaker (Furber et al., 2014), which recently achieved biological real-time simulations of a cortical microcircuit model (Rhodes et al., 2020) proposed by Potjans and Diesmann (2014) (which has since been simulated sub-realtime with NEST (Kurth et al., 2022) and with an FPGA-based neural supercomputer (Heittmann et al., 2022). This result was made possible by its architecture designed for efficient spike communication, performed with an optimized transmission system of small data packets. BrainScaleS and SpiNNaker are freely available to the scientific community through the EBRAINS Neuromorphic Computing service. Nevertheless, neuromorphic systems still require a significant amount of system-specific skills. Even if the simulation speed they can provide is impressive, the flexibility and simplicity of programming environments available for such neuromorphic systems are still low compared to their general-purpose counterparts. On neuromorphic systems adopting analog design techniques, advantages in speed, area, and energy consumption are associated with the difficulties of managing manufacturing fluctuations, unavoidable in analog substrates, and with the effects of electronic noise emerging in the dynamics of analog circuits. Porting neural simulations from digital systems to analog neuromorphic platforms is not a trivial task. Overcoming such difficulties and turning them into advantages is an emerging field of research (Wunderlich et al., 2019). Furthermore, as soon as the number of synapses established by each neuron reaches biological scales (i.e., several thousands per neuron), the current generation of neuromorphic systems often experience significant slowdown, whereas a new generation capable of coping with such issues is still under development. For example, in its maximum configuration, the first-generation BrainScaleS system hosts 1 billion synapses and 4 million neurons (250 synapses/neuron) on 20 silicon wafers (Güttler, 2017), and a similar synapse-per-neuron ratio is the sweet spot for optimal execution on SpiNNaker, well below the typical 10K synapses/neuron characteristic for pyramidal cortical neurons or >100K synapses/neuron sported by cerebellar Purkinje cells.

Lately some systems based on graphical processing units (GPUs) have emerged (Sanders and Kandrot, 2010; Garrido et al., 2011; Brette and Goodman, 2012; Vitay et al., 2015; Yavuz et al., 2016). These systems grant a higher flexibility compared to neuromorphic systems, because of the current technological constraints of the latter and because of the software support offered by platforms like CUDA (Compute Unified Device Architecture) (Sanders and Kandrot, 2010), created by NVIDIA to take advantage of the large compute resources of GPUs. As a matter of fact, spiking neural network simulations could reap large benefits from the high degree of parallelism of GPU systems, which allows for thousands of simultaneous arithmetic operations even for a single GPU. However, the effective speed-up made possible by parallelization on GPUs can be limited by sequential parts and operations like I/O of spike recordings and feeding inputs into the network model, which inevitably require data transfer between CPU and GPU memory.

Among GPU-based simulators we can mention CARLSim4 (Chou et al., 2018), a spiking neural network simulator written in C++ with a multi-GPU implementation, and NCS6 (Hoang et al., 2013), a CPU/GPU simulator specifically designed to run on high-performance computing clusters. More recently, CoreNEURON (Kumbhar et al., 2019) was developed as an optimized compute engine for the NEURON simulator. It is able to both reduce memory usage and increase simulator performance with respect to the NEURON simulator by taking advantage of architectures like NVIDIA GPUs and many-core CPUs. One of the most popular GPU-based simulators for spiking neural networks is GeNN (Yavuz et al., 2016), which has achieved fast simulations of the cortical microcircuit model of Potjans and Diesmann (Knight and Nowotny, 2018; Knight et al., 2021). Recently the same simulator, running on a single high-end GPU, has shown better performance compared to what was obtained with a CPU-based cluster (Knight and Nowotny, 2021) in the simulation of a multi-area spiking network model of macaque cortex (Schuecker et al., 2017; Schmidt et al., 2018a,b). This result was reached thanks to the procedural connectivity approach, consisting in generating the model connectivity and its synaptic weights only when spikes need to be transmitted, without storing any connectivity data in the GPU memory. As a matter of fact, one of the most constraining features of GPUs is the size of the built-in memory, which in spiking neural network simulations can be a severe limitation. The possibility of generating the connections on demand enables performing a large-scale simulation even with a single GPU. However, procedural connectivity is a suitable approach only with static synapses. Indeed, plastic synapses require data to be stored since their synaptic weights change their value during the simulation. The inclusion of plastic synapses is essential for many investigations, e.g., when learning or the interplay between synaptic changes and brain dynamics are of interest (Capone et al., 2019; Golosio et al., 2021a). GeNN allows and supports models with synaptic plasticity, but for such models the procedural connectivity approach is thus prevented.

NEST GPU (previously named NeuronGPU) (Golosio et al., 2021b) is a GPU-MPI library written in CUDA for large-scale simulations of spiking neural networks, which was recently included in the NEST Initiative with the aim of integrating it within the NEST spiking network simulator, in order to allow for simulations on GPU hardware. In this work we evaluate the performance of NEST GPU on simulations that exploit multiple GPUs on MPI clusters. The library implements a novel MPI-optimized algorithm for spike communication across processes that also leverages some of the delivery techniques already investigated for CPU-based distributed computing platforms. Currently, NEST GPU exploits the neuron distribution among processes as described in Pastorelli et al. (2019): neurons are allocated on processes taking into account their spatial locality, instead of using a round-robin approach. Spike delivery takes advantage of this distribution mode, resulting in an efficient and optimized algorithm. NEST GPU supports a large variety of neuron models and synapses, both static and plastic. In this work we compare the outcomes of NEST GPU and NEST for the full-scale multi-area spiking network model of macaque cortex simulated on a high-performance computing (HPC) cluster with both GPU- and CPU-equipped compute nodes. To this end the distributions of firing rates, coefficients of variation of interspike intervals (CV ISI), and Pearson correlations between spike trains obtained by the two simulators are examined. We further evaluate the performance in terms of simulation time per second of biological activity.

2. MATERIALS AND METHODS

2.1. NEST GPU Spike Communication and Delivery Algorithm

In this section the algorithm exploited by NEST GPU for spike communication between MPI processes and for spike delivery is briefly introduced. For an in-depth description of the spike delivery algorithm please see Golosio et al. (2021b).

In NEST GPU, the output connections of each neuron (or other spiking device) are organized in groups, all connections in the same group having the same delay. For each neuron there is a spike buffer, which is structured as a queue used to store the spikes emitted by the neuron. Each spike is represented by a structure with three member variables: a time index t_s, which starts from 0 and is incremented at every time step; a connection group index i_g, which also starts from zero and is increased every time the spike matches a connection group, i.e., when the time index corresponds to the connection group delay; and a multiplicity, i.e., the number of spikes emitted by the neuron in a single time step. Keeping a connection group index and having connection groups ordered according to their delays is useful for reducing the computational cost, because it avoids the need for a nested loop to compare the time index of the spike with all the connection delays. When the time index of a spike matches a connection group delay, spike information (i.e., source neuron index, connection group index, multiplicity) is inserted in a global spike array and the connection group index is increased. A spike is removed from the queue when i_g becomes greater than the number of connection groups of that neuron, i.e., when the time index becomes greater than the maximum delay. The final delivery from the global spike array to the target neurons is done in a loop, so no additional memory is required. When a source neuron is connected to target neurons belonging to a different MPI process, a spike buffer, similar to the local one, is created in the remote MPI process. When the source node fires a spike, this is sent to the spike buffer of the remote MPI process, which will deliver the spike to all target neurons after proper delays. The remote spikes, i.e., the spikes that must be transferred to remote MPI processes, are communicated through non-blocking MPI send and receive functions at the end of every simulation time step. Let N be the number of MPI processes. The whole procedure consists of three stages:

1. Each MPI process initiates a non-blocking receive (`MPI_Irecv`) on $N - 1$ receiving buffers (one for each remote MPI process), so that all receiving buffers are ready more or less simultaneously;
2. Each MPI process initiates forwarding of the remote spikes to all other $N - 1$ processes by calling a non-blocking send (`MPI_Isend`);
3. Each MPI process initializes a list with the indexes of the other $N - 1$ processes, and starts checking all the items in the list in an endless loop with `MPI_Test`. When the transfer from the i-th MPI process is complete, the corresponding index i is removed from the list. The loop is interrupted when the list is empty.

The spike buffer for a single network node and the spike handling and delivery for multiple MPI processes are depicted in **Figure 1**.

2.2. NEST GPU Spike Recording Algorithm

In this section the NEST GPU algorithm for spike time recording is introduced. The spike times are initially recorded in the GPU

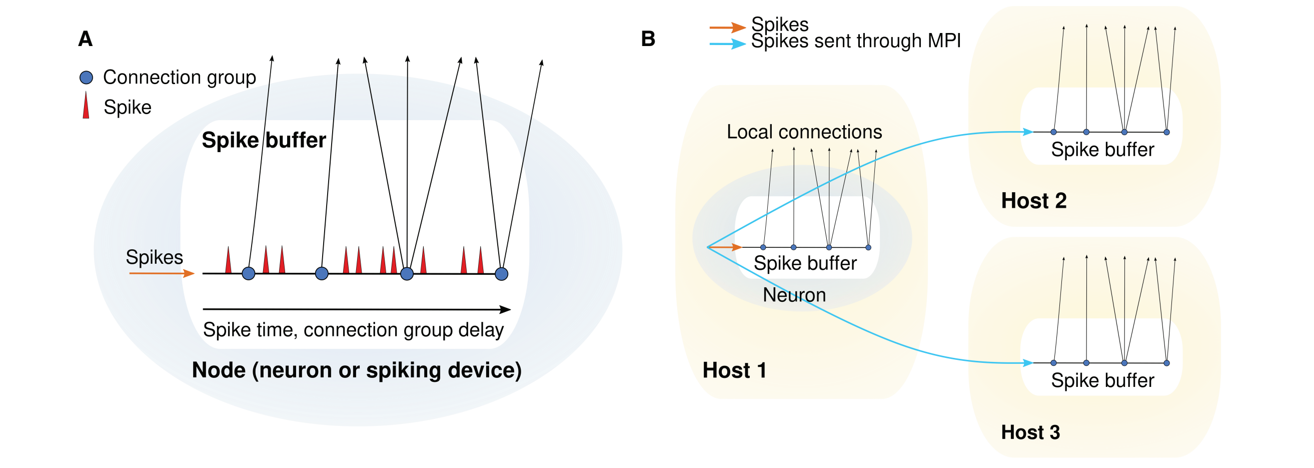


FIGURE 1 | Spike handling and delivery schemes. **(A)** Structure of a single spike buffer. **(B)** Schematic depicting MPI communication between spike buffers for different hosts.

memory in a two-dimensional array, with the number of rows equal to the number of neurons and the number of columns equal to the maximum number of spikes that can be recorded before each extraction. Since the number of spikes per neuron is typically much smaller than the maximum, most entries of this array are zero. To compress the information, the spike times are periodically packed into contiguous positions of a one-dimensional buffer, which is copied from GPU memory to RAM along with a one-dimensional array indicating the positions at which the spikes for each neuron start. The packing algorithm works as follows:

1. Let N_i be the number of recorded spikes of the i-th neuron, and C_i its cumulative sum (also called prefix scan):

$$C_0 = 0; \qquad C_i = \sum_{k=0}^{i-1} N_k \quad \text{for } i = 1, ..., n \tag{1}$$

   where n is the number of neurons. Note that C_i has $n+1$ elements, one more than N_i, and that it is sorted by construction. C_i is computed in parallel with CUDA using the algorithm described by Nguyen (2007, Chapter 39) as implemented in https://github.com/mattdean1/cuda. The last element of C_i, $N_{\text{tot}} = C_n$, is the total number of recorded spikes of all neurons;
2. Let $t_{i,j}$ be the time of the j-th recorded spike of the i-th neuron. The packed spike array A_m $(m = 0, \ldots, N_{\text{tot}} - 1)$ is computed from $t_{i,j}$ using a one-dimensional CUDA kernel with N_{tot} threads. m is set equal to the thread index. Since C_i is sorted, a binary-search algorithm can be used to find the largest index i such that

$$C_i \leq m < C_{i+1} \tag{2}$$

   C_i will be the index of the first spike of the i-th neuron in the packed spike array, therefore the spike m in this array will correspond to the spike i, j in the original two-dimensional array $t_{i,j}$, where j is simply

$$j = m - C_i \tag{3}$$

   Once i and j are computed from m, it is possible to set

$$A_m = t_{i,j} \tag{4}$$

Packing of recorded spikes and transfer to the RAM can be performed after a certain number of simulation time steps depending on GPU memory availability.

2.3. Multi-Area Model

We consider the dynamics of a model of all vision-related areas in one hemisphere of macaque cortex (Schmidt et al., 2018a,b) (**Figure 2**). Here, we briefly summarize the model; all details and parameter values can be found in the original publications. Following the parcellation of Felleman and Van Essen (1991), the model includes 32 areas that either have visual function or are strongly interconnected with visual areas. To yield a tractable model size, only 1 mm^2 of cortex is represented within each area, albeit with the full local density of neurons and synapses. This leads to a total of about 4.1 million neurons and 24 billion synapses. The areas have a laminar structure, layers 2/3, 4, 5, and 6 each containing one population of excitatory (E) and one population of inhibitory (I) neurons (area TH lacks layer 4); hence the total number of populations in the network is 254. The neuron model is the leaky integrate-and-fire model with exponential current-based synapses, and all neurons have the same electrophysiological parameter values. The initial membrane potentials are normally distributed. Input from non-modeled brain regions is represented by homogeneous Poisson spike trains with area-, layer- and population-specific rates.

The numbers of neurons are determined from a combination of empirically measured neuron densities, cytoarchitectural type definitions of areas, and the thicknesses of the cortical layers. The connectivity of the local microcircuits consists of scaled versions of the connectivity of a microcircuit model of early sensory cortex (Potjans and Diesmann, 2014). The inter-area connectivity is based on axonal tracing data collected in the CoCoMac database (Bakker et al., 2012), complemented with the quantitative tracing data of Markov et al. (2011, 2014). Gaps in the data are filled by predictions of overall connection densities from inter-area distances, and laminar patterns from relative neuron densities of source and target areas. The synapses are statistically mapped to target neurons based on the extent of the dendritic trees of morphologically reconstructed neurons of each type in each layer (Binzegger et al., 2004). A mean-field-based method slightly adjusts the connectivity to support plausible spike rates (Schuecker et al., 2017).

When the cortico-cortical synapses have the same strength as the local synapses, this leads to a stationary "ground state" of activity without substantial rate fluctuations or inter-area interactions (**Figure 2B**). As this state does not match experimental resting-state recordings of spiking activity and functional connectivity between areas, the cortico-cortical synaptic strengths are increased, especially onto inhibitory neurons, in order to generate substantial inter-area interactions while maintaining balance. Poised just below a transition to a high-activity state, the spiking activity is irregular with low synchrony apart from population events of variable duration. In this "metastable state" (**Figure 2C**), aspects of both microscopic and macroscopic resting-state activity in lightly anesthetized monkeys are well reproduced: the spectrum and spike rate distribution of the modeled spiking activity of primary visual cortex (V1) are close to those from parallel spike train recordings (Chu et al., 2014a,b); and the functional connectivity between areas approximates that obtained from fMRI recordings (Babapoor-Farrokhran et al., 2013).

For further details we refer to the original publications (Schmidt et al., 2018a,b).

3. RESULTS

In this section we first verify the correctness of the simulations performed by NEST GPU, using NEST 3.0 as a reference. Afterwards, the performance evaluation is presented in terms of build (i.e., network construction) and simulation time.

To this end, we used the HPC cluster JUSUF (von St. Vieth, 2021). In particular, the NEST GPU simulations employed 32 accelerated compute nodes, each of them equipped with two AMD EPYC 7742 (2 × 64 cores, 2.25 GHz), 256 GB of DDR4 RAM (3,200 MHz), and an NVIDIA V100 GPU with 16 GB HBM2e; inter-node communication is enabled *via* InfiniBand HDR100 (Connect-X6). The NEST simulations were run on standard compute nodes of the HPC cluster JURECA-DC (Thörnig and von St. Vieth, 2021), which uses the same CPUs and interconnect as JUSUF but has 512 GB of DDR4 RAM per node available.

3.1. Comparison of Model Results Between NEST and NEST GPU

In Golosio et al. (2021b) some of us have compared the simulation outcomes between NEST GPU and NEST for the cortical microcircuit model of Potjans and Diesmann (2014), showing an optimal match between the results of both simulators. The validation approach follows that of van Albada et al. (2018) and Knight and Nowotny (2018). In this section we present a similar procedure in order to validate the NEST GPU outcome for the multi-area model considered here.

Firstly, for each of the executed simulations, we simulated 10 s of biological activity of the full-scale multi-area model in both NEST and NEST GPU. All the simulations were performed with a time step of 0.1 ms. We simulated both the ground state (showing asynchronous irregular spiking with stationary rate) and the metastable state of the model (better representing the resting-state activity of the cortex) in order to compare the results of both configurations. To avoid transients due for instance to initial synchronization, a pre-simulation time of 500 ms was employed for all the simulations. This enhances the independence of the derived activity statistics from the total simulation time.

We executed 10 simulations for each simulator, recording the spike times. The 10 simulations differ in the chosen seed for the random number generation, so that there is no pairwise matching of seeds between NEST and NEST GPU simulations. Furthermore, we performed another set of 10 simulations with NEST to estimate the differences that arise only because of the different seeds used. Taking their outcome as a reference for both NEST GPU and NEST simulations, it was possible to evaluate NEST-NEST and NEST-NEST GPU comparisons.

To compare the simulation outcomes using the recorded spike times, we selected and extracted the distributions of three quantities for each population:

- The time-averaged firing rate of each neuron;
- The coefficient of variation of inter-spike intervals (CV ISI), i.e., the ratio between the standard deviation and the average of inter-spike time intervals of each neuron;
- The pairwise Pearson correlation between the spike trains obtained from a subset of 200 neurons for each population, in order to grant a reasonable computing time.

The spike trains were binned with a time step of 2 ms, corresponding to the refractory time, so that at most one spike could occur in each bin. Considering a binned spike train b_i for neuron i with mean value μ_i, the correlation coefficient between two spike trains b_i and b_j is defined as:

$$C[i,j] = \langle b_i - \mu_i, b_j - \mu_j \rangle / \sqrt{\langle b_i - \mu_i, b_i - \mu_i \rangle \cdot \langle b_j - \mu_j, b_j - \mu_j \rangle} \tag{5}$$

where $\langle , \rangle$ represents the scalar product. Hence a 200×200 matrix is built and the distribution of the Pearson correlations can be evaluated as the distribution of the off-diagonal elements. All aforementioned distributions were computed using the Elephant package (Denker et al., 2018).

The raw distributions were smoothed using Kernel Density Estimation (KDE) (Rosenblatt, 1956; Parzen,

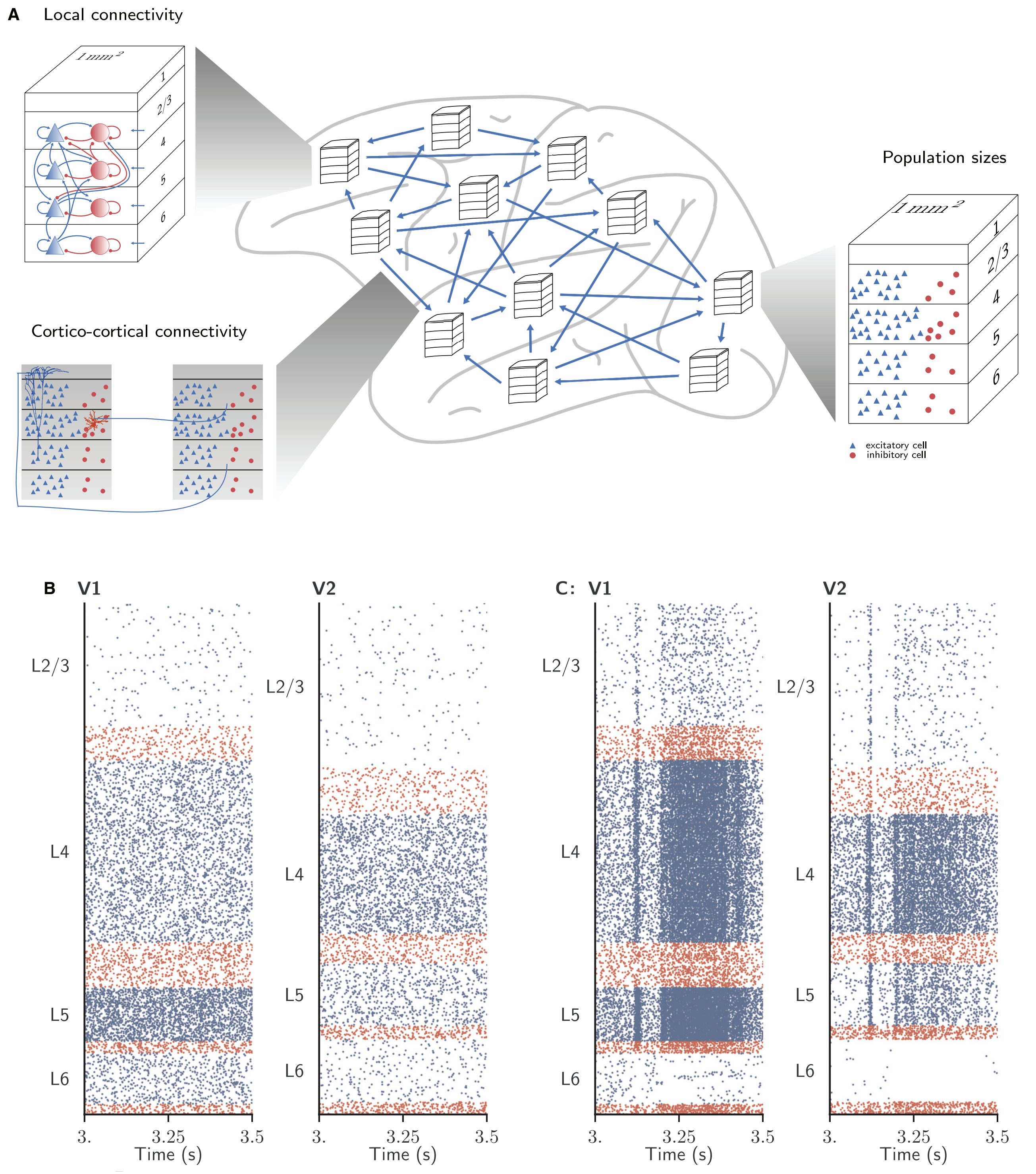


FIGURE 2 | Spiking neuronal network model used to evaluate simulator performance in this study. **(A)** Schematic overview of the model. The multi-area model represents 32 areas of macaque vision-related cortex, each modeled by four cortical layers with a size of 1 mm^2. Local connectivity, cortico-cortical connectivity, and population sizes are adapted for each area. **(B)** Network activity of areas V1 and V2 in the ground state. **(C)** Network activity of the same areas in the metastable state. Figure adapted from Schmidt et al. (2018a) and Schmidt et al. (2018b).

1962). The KDE method was applied with the `sklearn.neighbors.KernelDensity` function of the scikit-learn Python library (Pedregosa et al., 2011) (version 0.24.2). Specifically, we performed KDE with a Gaussian kernel, optimized with a bandwidth obtained using the Silverman method (Silverman, 1986).

With this procedure we obtained 762 distributions for each simulation. For each of the 254 populations we determined the average and standard deviation of these distributions across each set of 10 simulations. To gain an impression of the similarity of the simulation outcomes of NEST and NEST GPU, example distributions are shown in **Figures 3**, **4**.

As can be observed, the distributions obtained with the two simulators closely match each other in the ground state (**Figure 3**), and also the error bands are negligible because of the small variability of the state. In the metastable state, the variability between the NEST and NEST GPU distributions is larger (**Figure 4**). Due to the increased variability, we decided to depict an additional NEST distribution to show the substantial fluctuations that can arise between two sets of NEST simulations.

To provide an overview over the distributions for the entire model, averaged distributions for each layer and area were computed. These data were plotted with the `seaborn.violinplot` function of the Seaborn Python library (Waskom, 2021) (version 0.11.1), which returns KDE-smoothed distributions optimized with the Silverman method, matching our calculation of the distributions. The distributions thus obtained were compared by placing them side by side in the split violin plots shown in **Figure 5**, also showing median and interquartile range for every distribution.

The area-averaged distributions compared in **Figure 5** are nearly indistinguishable. The same holds for each of the 254 population-level distributions separately[1].

To quantify the similarity between the distributions, the Earth Mover's Distance (EMD) was computed. This metric evaluates the distance between two probability distributions, and its name stems from an analogy with the reshaping of soil. The two distributions may be thought of as, respectively, a given amount of earth located in a certain space and the same amount of earth that has to be arranged properly. The Earth Mover's Distance can thus be seen as the minimum amount of work needed to obtain the desired distribution from the original one. It is equivalent to the 1st Wasserstein distance between two distributions (see **Supplementary Material**). In this work it has been computed using the `scipy.stats.wasserstein_distance` function of the Python scientific library SciPy (Virtanen et al., 2020) (version 1.5.2). We opted for this measure instead of the Kullback-Leibler divergence adopted in the procedure described in Golosio et al. (2021b) because of the metric properties of the EMD, which makes it not only more specific in detecting the degree of dissimilarity among distributions but also symmetric.

To verify the equivalence between the simulators we analyzed the box plots obtained from the set of 10 EMD values for each population, given by the pairwise comparison of each of the 10 simulations. This way, we take into consideration the possible variability due to the different random number generator seeds. The random connectivity, membrane potential initialization, and external drive mean that one expects a nonzero EMD between simulations with different random seeds even with the same simulator. Furthermore, the different order of the operations in the two simulators combined with the chaotic dynamical state imply that nonzero differences would be expected even with the same random seeds for different simulators. Since EMD has the same units as the variables over which the distributions are computed, it is possible to directly estimate the relevance of the corresponding values.

Figure 6 shows the EMD box plots obtained from the comparisons NEST-NEST and NEST-NEST GPU for the three distributions calculated for area V1, respectively, for the ground state and the metastable state. The EMD values for the NEST-NEST GPU comparison are distributed similarly to those for the NEST-NEST comparison, meaning that the differences that arise due to the choice of simulator are statistically similar to those between NEST simulations with different random number generator seeds. Thus, using NEST GPU instead of NEST (with different random numbers) does not add variability compared to using different random seeds with the same simulator. This is a further indication that NEST and NEST GPU yield statistically closely similar results. EMD values obtained by the comparison of the ground state distributions are smaller than the EMD values obtained for the metastable state. This is due to the increased fluctuations in the latter state of the model. In some cases, the whiskers for the NEST-NEST and NEST-NEST GPU comparisons have different extents. This may be related to long-tailed distributions of the corresponding activity statistics, especially for correlations (cf. **Figure 5**). Differences in the tails of the distributions caused by only a few data points can lead to large differences in EMD values because the probability mass needs to be moved over large distances to turn one distribution into another. However, the EMD values are marginal compared to the values within the distributions shown in **Figure 5**, revealing a negligible difference between the NEST and NEST GPU simulation results. This statement is also true for the other areas of the model, as shown in the **Supplementary Material**.

3.2. Performance Evaluation

Hitherto we showed that NEST and NEST GPU simulation outcomes are comparable. In this section the performance of NEST GPU is evaluated and compared with that of NEST 3.0.

We divided the total execution time into build and simulation time. The former includes the time needed to allocate memory for the network components (i.e., neurons, synapses, and all devices, such as Poisson generators and spike detectors), and to establish the connections. The simulation time measures how long it takes to propagate the network dynamics for the specified amount of biological time once the model has been set up.

The simulation time for NEST and NEST GPU was further divided to reflect the following subtasks:

[1] The distributions are available at https://github.com/gmtiddia/ngpu_multi_area_model_simulation/tree/main/analysis/dist_plots/Areas.

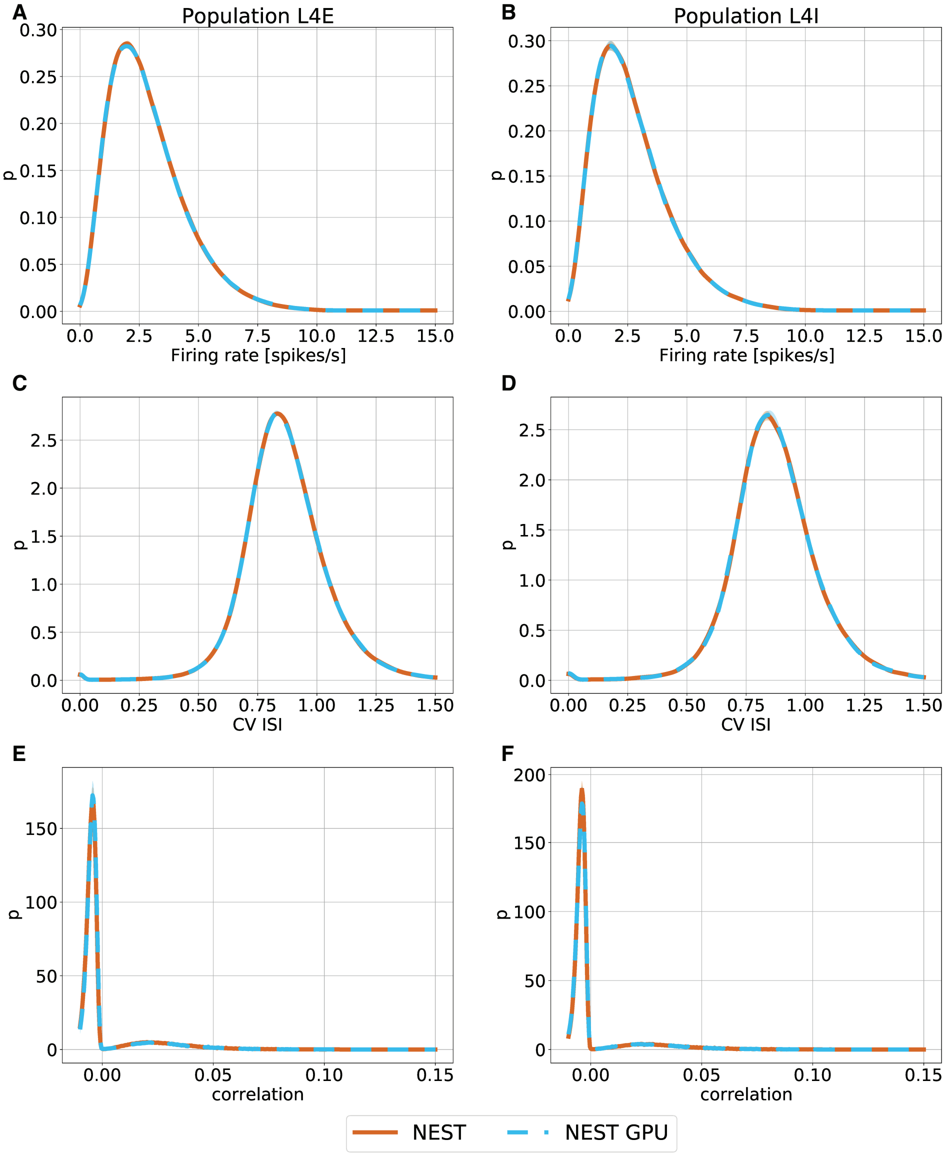


FIGURE 3 | Ground state distributions of firing rate **(A,B)**, CV ISI **(C,D)** and Pearson correlation of the spike trains **(E,F)** for the populations L4E and L4I of area V1. The distributions are averaged over 10 simulations with NEST (orange) and NEST GPU (sky blue). Every averaged distribution has an error band representing its standard deviation.

- Delivery, describing the time for local spike handling and delivery;
- MPI communication, describing the time for remote spike handling and delivery;
- Collocation, i.e., the time employed for the preparation of the MPI send buffers;
- Update, i.e., the dynamics update time;
- Other, a general subtask in which other contributions to the overall simulation time are taken into account.

As reported in Golosio et al. (2021b), NEST GPU creates the model connections in the RAM, and thereafter copies them to the GPU memory. For this reason, the build phase, i.e., the phase related to the network construction, does not take advantage of any speed-up due to the use of GPUs. However, the build phase does not depend on the biological time, meaning that the more biological time is simulated, the less relevance the build time has for the overall duration of the simulation.

The simulations performed on JUSUF by NEST GPU used 32 compute nodes with one MPI process each and 8 threads

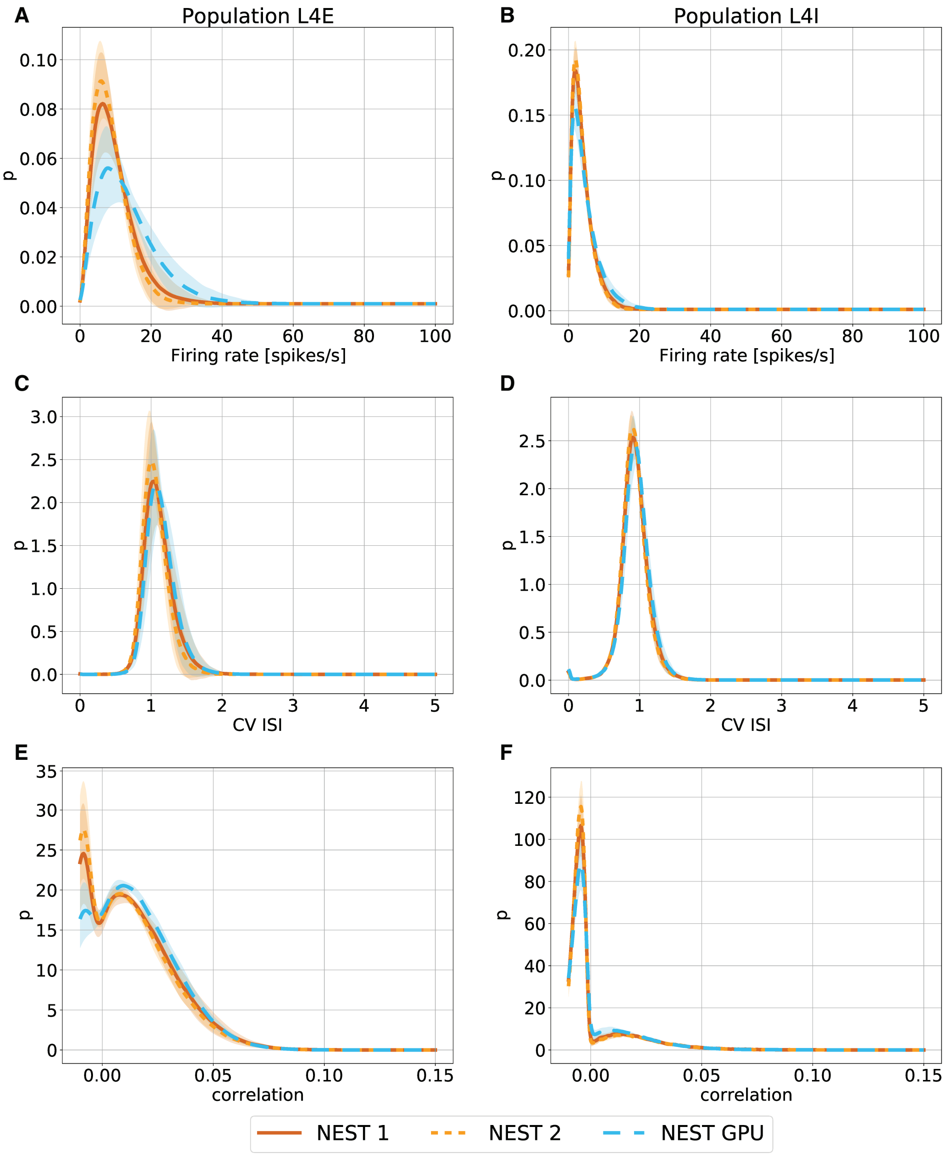


FIGURE 4 | Metastable state distributions of firing rate **(A,B)**, CV ISI **(C,D)** and Pearson correlation of the spike trains **(E,F)** for the populations L4E and L4I of area V1. The distributions are averaged over 10 simulations with NEST (orange lines) and NEST GPU (sky blue dashed line). Every averaged distribution has an error band representing its standard deviation. An additional set of NEST simulation distributions is also shown.

per MPI process. It should be noted that while NEST uses MPI and thread parallelism during both build and state propagation phases, the number of threads per MPI process in NEST GPU affects only the build time, because the connections are initially created in parallel by different OpenMP threads in CPU memory, as stated above. This parallel setup, which permits the simulation of an area for each compute node, was the most efficient in terms of compute time, because the NVIDIA V100 GPU memory can hold one model area at most and also because in this setup only inter-area communications have to be carried out by MPI. Indeed, it is known that one of the most significant bottlenecks in parallel computation is the communication between MPI processes (Marjanović et al., 2010), and herein the way NEST GPU handles spike delivery and distributes model areas between MPI processes (i.e., an area for each MPI process) grants an efficient parallel optimization.

Performance was evaluated using 10 simulations of 10 s of biological time for both NEST and NEST GPU, averaging over random number generator seeds. In contrast to the previous simulations, spike recording was disabled. To obtain a single set

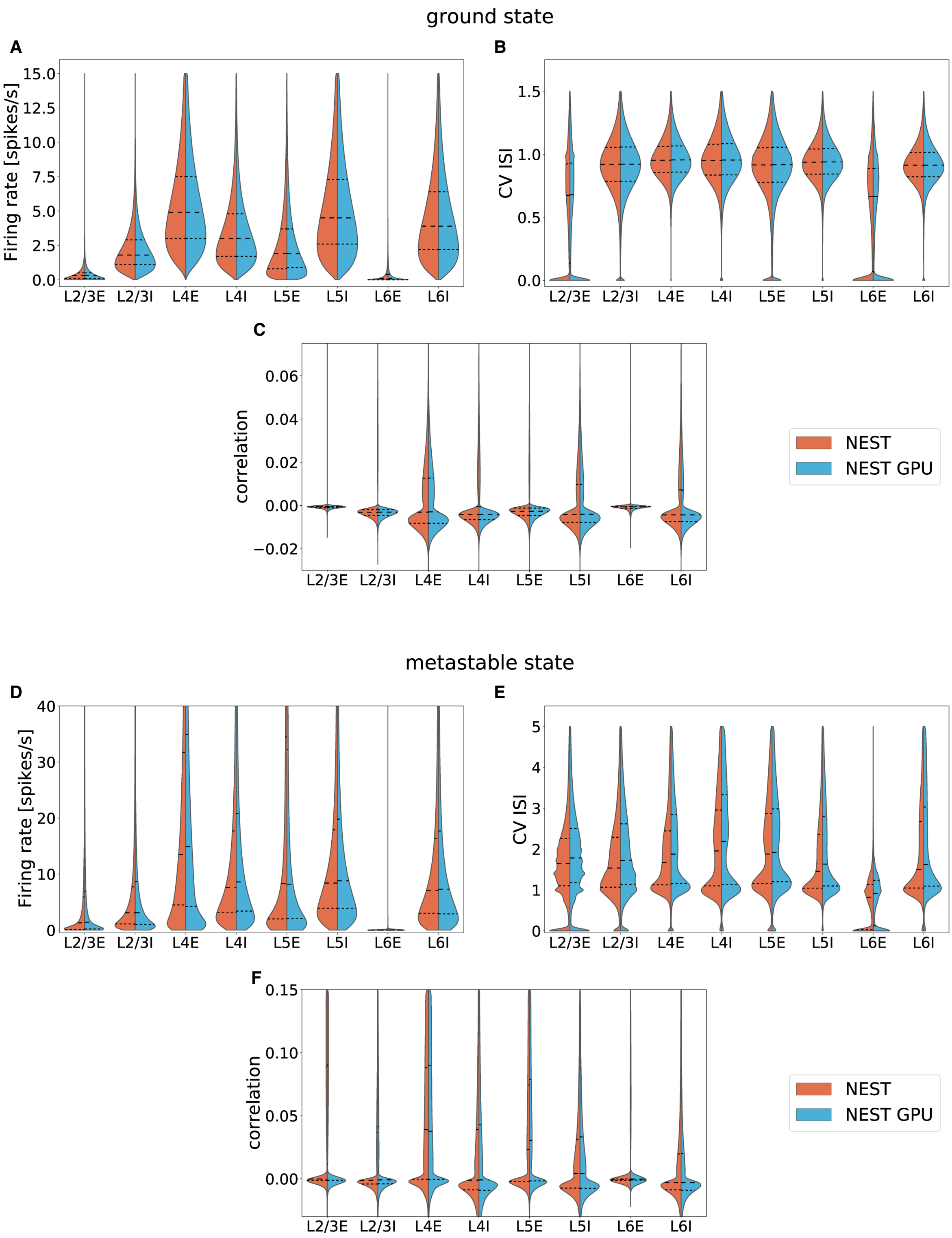


FIGURE 5 | Averaged distributions of the ground state and the metastable state of the model for all 32 areas obtained using NEST (orange, left side) and NEST GPU (sky blue, right side) and compared with split violin plots. The central dashed line represents the distribution's median, whereas the other two dashed lines represent the interquartile range. **(A,D)** average firing rate, **(B,E)** average CV ISI, **(C,F)** average Pearson correlation of the spike trains.

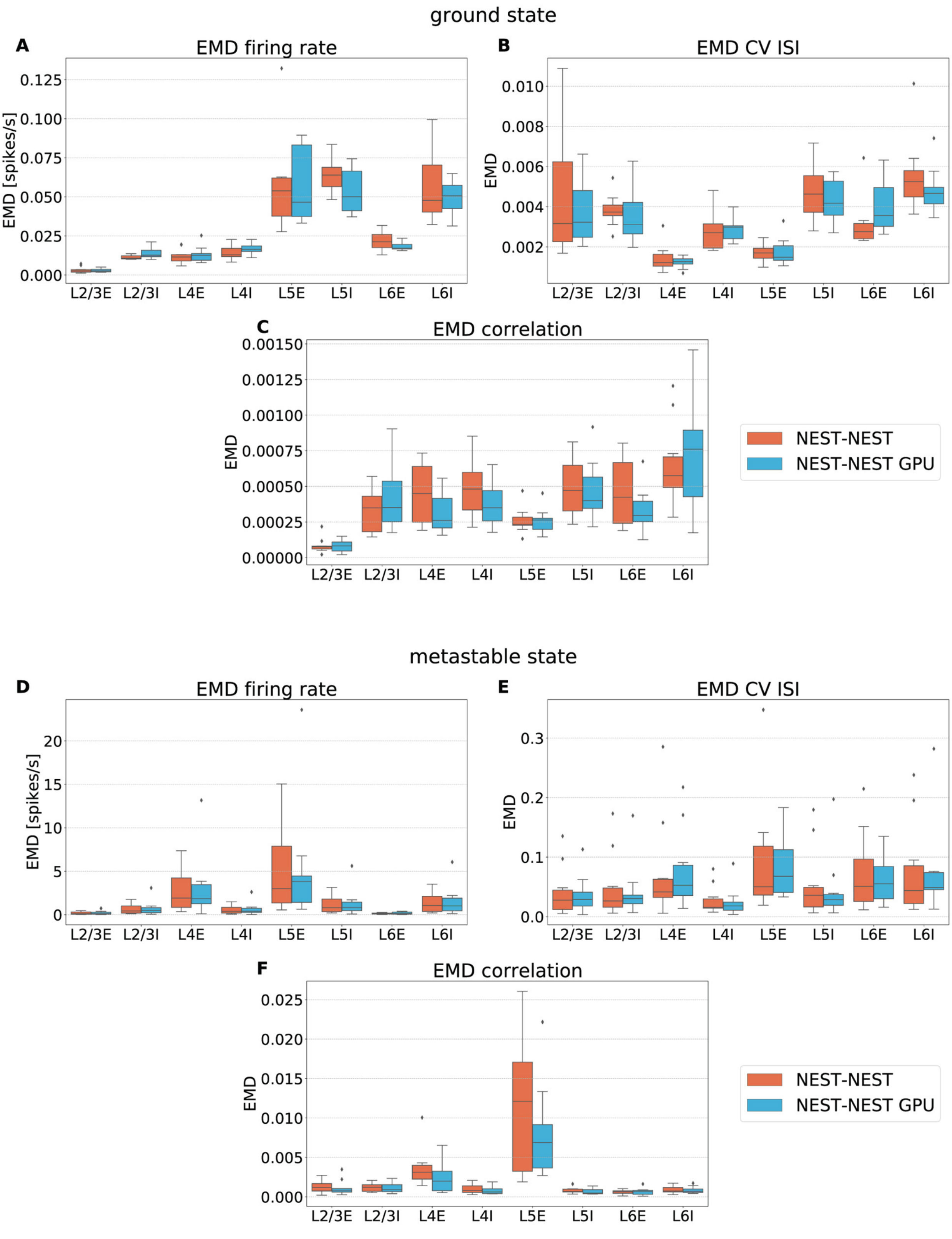


FIGURE 6 | Earth Mover's Distance between distributions of firing rate **(A,D)**, CV ISI **(B,E)** and correlation of the spike trains **(C,F)** obtained for area V1 of the model in the ground state and the metastable state. NEST-NEST (orange, left) and NEST-NEST GPU (sky blue, right) data are placed side by side.

of values for each simulation we performed time measurement on each employed compute node separately and then we averaged the obtained values. Since the MPI processes are synchronized by NEST GPU after each simulation time step, the overall simulation time for each node is the same; however, the time taken by individual subtasks differs somewhat across the MPI processes due to the differences between the areas of the model, such as number of neurons, density of connections, and activity rate. These subtask differences across the MPI processes are discussed later in this section. Once we extracted a single set of timings for each simulation we computed their mean and standard deviation to obtain a unique set of values.

Figure 7 shows the performance benchmarks of the multi-area model on CPUs conducted on JURECA-DC using the benchmarking framework beNNch (Albers et al., 2022). For both network states, the optimal configuration of the hybrid parallelization is achieved with 8 MPI processes per node and 16 threads per task, thus making use of every physical core of the machine while avoiding hyperthreading. NEST distributes neurons in a round-robin fashion across virtual processes. This implements a simple form of static load balancing as neuronal populations are distributed evenly. Larger error bars in **Figure 7B** demonstrate the increased dependence on initial conditions and decreased stability of network activity of the metastable state. For the network simulations of the ground and metastable states, the scalings plateau at 12 nodes and 32 nodes, respectively. As discussed in Jordan et al. (2018), plateau is expected in strong scaling experiments once the MPI communication dominates. **Figure 7** shows that indeed all contributions except the communication get smaller for increasing numbers of MPI processes.

In the ground state simulations, comparing the configurations with 32 nodes, the network construction times were 951 ± 29 s and 80 ± 7 s (mean ± st.dev.) for NEST GPU and NEST, respectively. Simulations of the multi-area model in its metastable state revealed similar network construction times of 957 ± 41 s for NEST GPU and 69.5 ± 0.4 s for NEST.

In terms of state propagation time, ground state simulations took 6.5 ± 0.1 s using NEST GPU, whereas NEST took 15.6 ± 2.1 s, both measured per second of biological model time. In the metastable state NEST GPU was able to compute a second of biological activity in 15.3 ± 0.9 s, whereas NEST took 47.9 ± 7.7 s. The longer simulation time taken for the metastable state is explained by the higher firing rates and synchrony in this state.

In case of enabled spike recording using NEST GPU the simulation time increases up to 5% when recording from all neurons. In these simulations, packing of recorded spikes and transfer to the CPU memory is performed every 2,000 simulation time steps (i.e., every 200 ms of biological time). This overhead is strongly dependent on the model simulated and the amount of GPU memory available. In fact a larger GPU memory would support larger buffers of recorded spikes, diminishing the frequency of copy operations from GPU memory to CPU memory. Furthermore, the overhead can be reduced by recording spikes from only a fraction of the neurons.

Figure 8A shows the various contributions to the simulation time for NEST and NEST GPU. The main difference between the simulators appears in the time taken by spike communication, evincing the advantage of exploiting a neuron distribution among MPI processes that takes into account spatial locality. The round-robin distribution of neurons in NEST necessitates a larger degree of parallelization and hence communication to reach optimal performance. This increased communication is needed regardless of whether MPI or OpenMP parallelism is used. Indeed, replacing the 8 MPI processes per node by a further 8 threads incurs an even greater performance penalty (data not shown). We here compare both simulators in configurations which yield optimal performance.

The relative contributions of the various phases do not differ strongly between the ground and metastable states. The contribution of the communication of spikes between different MPI processes for the metastable state of the model is around 8.0 and 29.7 s per second of biological time for NEST GPU and NEST, respectively. The contribution of update, delivery, and other operations, excluding the communication of spikes between different MPI processes, is around 7.3 s for NEST GPU and 18.0 s for NEST. We can therefore observe that the better performance of NEST GPU compared to NEST is mainly due to a reduction in the communication time of the spikes between MPI processes, although there is an improvement also in the time associated with the update and delivery of local spikes.

Regarding the differences in computation time across MPI processes in NEST GPU, as mentioned above, the time taken by individual subtasks can vary across MPI processes because of differences between the areas of the model. However, since MPI processes are synchronized at the end of every simulation time step, the overall simulation time shown by every MPI process is the same. The resulting latency due to the difference between model areas is embedded in the Communication subtask. We measured that, within a simulation, the contribution of the spike communication between the 32 MPI processes (i.e., the 32 areas of the model) can vary up to 25% with respect to its average shown in **Figure 8A** and the contribution of the local spike delivery subtask shows comparable variations. The rest of the subtasks (i.e., Collocation, Update and Other) do not change significantly across the MPI processes, as shown in **Figure 8B**.

4. DISCUSSION

In this work we have compared the simulators NEST GPU and NEST on a full-scale multi-area spiking network model of macaque cortex with 4.1 million neurons and 24 billion synapses (Schmidt et al., 2018a,b). As described at the beginning of the Results section, the NEST GPU simulations used 32 nodes of the HPC cluster JUSUF, each node of which is equipped with an NVIDIA V100 GPU. The NEST simulations used 32 nodes of the JURECA-DC cluster, each of which is equipped with two AMD EPYC 7742 CPUs. We have considered both the ground state of the model and the metastable state, where the latter better represents *in vivo* cortical activity thanks to stronger inter-area connections.

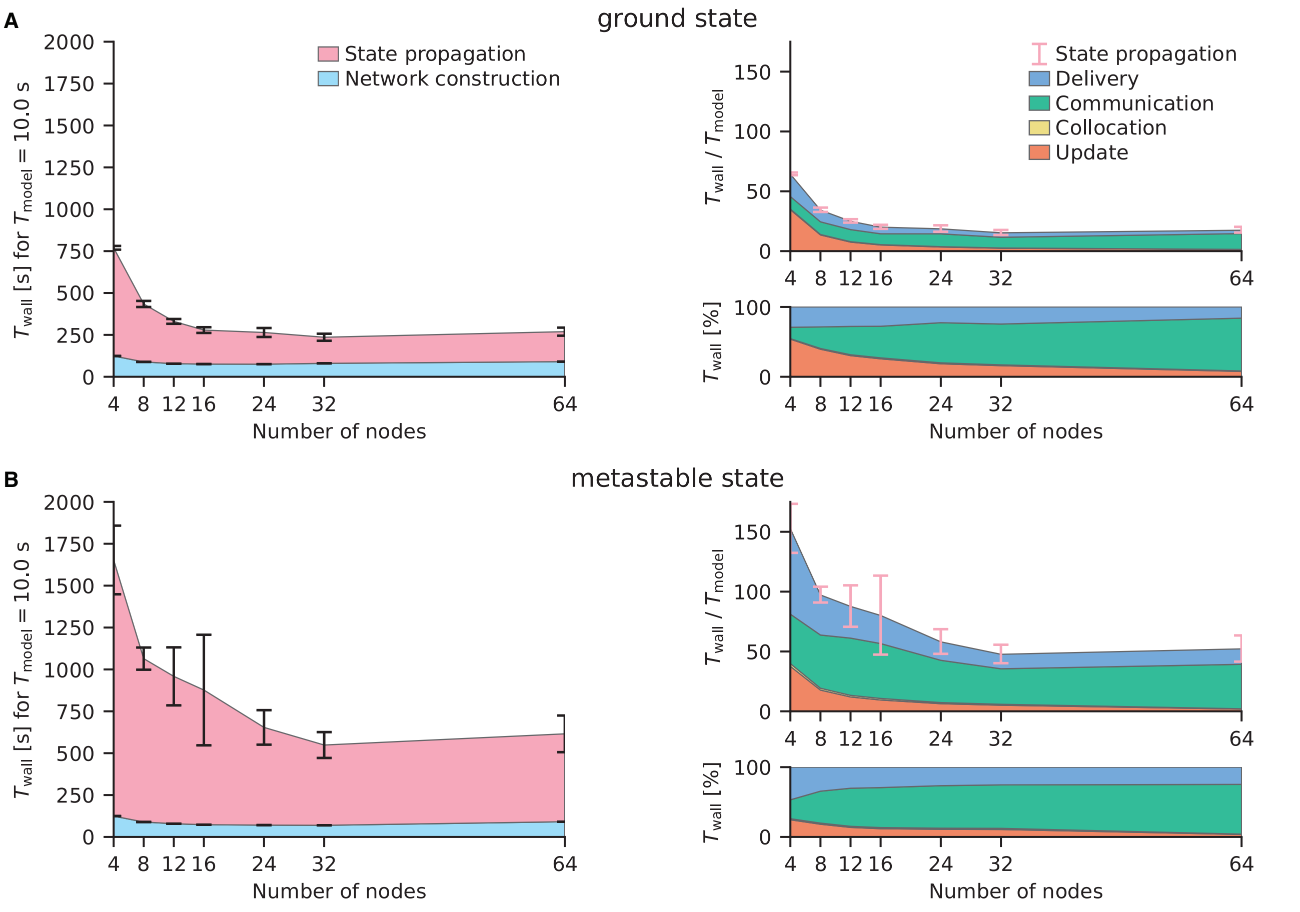


FIGURE 7 | Strong-scaling performance of the multi-area model in its ground and metastable states on JURECA-DC using NEST 3.0. **(A)** Simulated with parameters inducing stable ground state activity in the network. The left sub-panel displays the absolute wall-clock time T_{wall} for the network construction and state propagation in ms for a biological model time $T_{\text{model}} = 10$ s. Error bars indicate the standard deviation of the performance across 10 repeat simulations with different random seeds, the central points of which show the respective mean values. Error bars are shown in pink in the right panels to indicate that they are for the state propagation phase as a whole; the corresponding standard deviations are the same as in the left panels. The top right sub-panel presents the real-time factor defined as $T_{\text{wall}}/T_{\text{model}}$. Detailed timers show the absolute (top right) and relative (bottom right) time spent in the four different phases of the state propagation: update, collocation, communication, and delivery. Where the collocation phase is not discernible, this is due to its shortness. **(B)** Simulated with parameters inducing a metastable state with population bursts of variable duration. Same arrangement as **(A)**.

Figure 5, showing the averaged distributions of firing rate, CV ISI, and Pearson correlation obtained with NEST and NEST GPU, exhibits the compatibility between the outcomes of the two simulators in both states of the network. We have also quantified the differences that arise between a NEST and a NEST GPU simulation using the Earth Mover's Distance (EMD) metric. Specifically, we used EMD to evaluate the differences between the distributions obtained for each population with the two simulators. The results of this analysis show that the differences between NEST and NEST GPU simulations are comparable to those between multiple NEST simulations differing only in terms of their random seeds.

Regarding simulation performance, we observed that the build time of the multi-area model simulations is substantially higher using NEST GPU as compared to NEST. This is due to the fact that NEST GPU builds the network in the RAM and thereafter copies the constructed model to GPU memory. This additional step represents the bottleneck of the network construction phase using NEST GPU. However, since the build time is independent of the biological simulation time, it can be regarded as an overhead with decreasing relevance for longer biological times. A future integration of the network construction phase into the GPU memory could strongly decrease this contribution.

In terms of simulation time, NEST GPU shows a remarkable performance (**Figure 8A**). Simulations of the multi-area model in its ground state achieved a simulation time of 6.5 s per second of biological activity, reaching a speed-up factor of 2.4 compared to NEST. In the metastable state, NEST GPU reached 15.3 s of simulation time per second of biological activity which is approximately 3.1× faster than NEST simulations. Future work

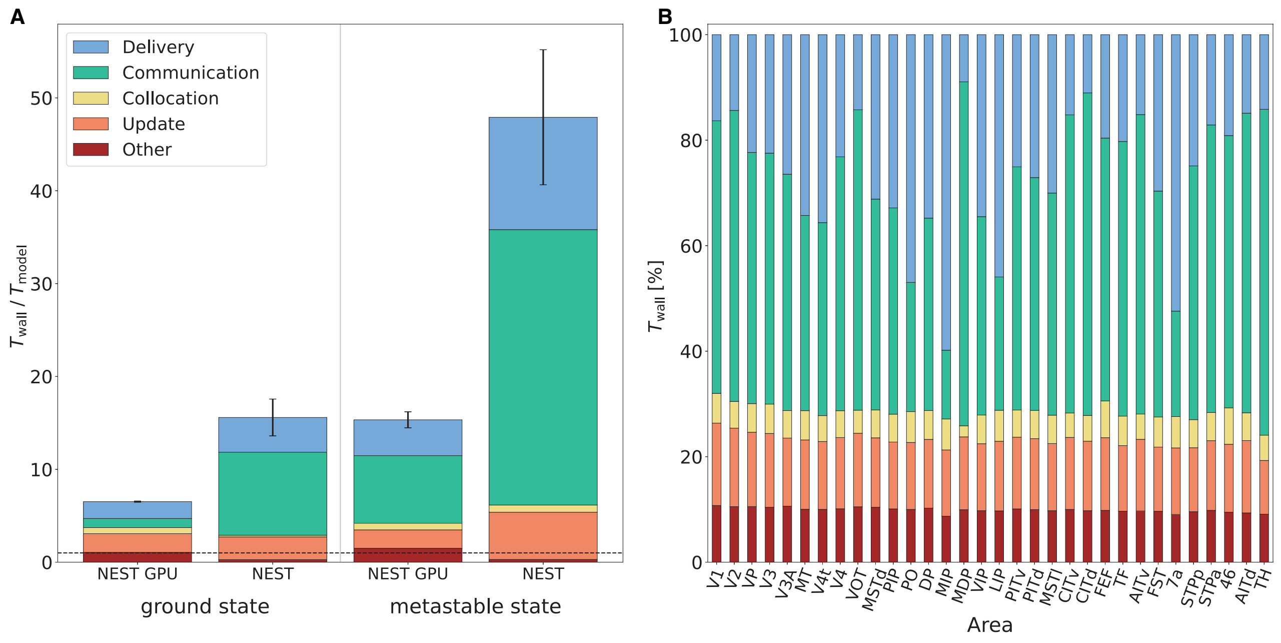


FIGURE 8 | Contributions to the simulation time of the multi-area model. **(A)** Contributions to the simulation time in the ground state and the metastable state for NEST and NEST GPU measured with the real-time factor. Error bars show the standard deviation of the overall performance across 10 simulations with different random seeds. The plot shows the performance obtained by NEST GPU and NEST in the 32-node configuration. NEST simulations were performed on JURECA-DC using 8 MPI processes per node and 16 threads per task, whereas NEST GPU simulations were performed on JUSUF using one MPI process per node and 8 threads per task. The black dashed line indicates the biological time. **(B)** Relative contributions to the simulation time of the multi-area model in the metastable state for every area (i.e., for every MPI process) in a NEST GPU simulation.

can further improve upon this performance: firstly, if each node of the HPC cluster were equipped with more than one GPU, the communication time, and with it the simulation time, would diminish. Secondly, the same simulation performed with a more recent GPU hardware (e.g., NVIDIA A100 GPUs) would permit not only faster simulations but also the possibility to simulate more than one area of the model on the same GPU thanks to enhancements of the GPU memory.

From the Results section it can be observed that the most relevant differences in the performance of NEST and NEST GPU in the simulation of the multi-area model are related to the contribution of the spike communication to the total simulation time. NEST uses a round-robin distribution of the nodes among MPI processes, and a two-tier connection infrastructure for communicating spikes. This infrastructure differentiates between data structures on the presynaptic side, i.e., the MPI process of the sending neuron, and the postsynaptic side, i.e., the MPI process of the receiving neuron. By using the blocking MPI Alltoall, spikes, which are stored in MPI buffers, are routed across MPI processes from pre- to postsynaptic neurons. In the implementation described by?, a spike having target neurons on different threads necessitated communication of spike copies to all these threads. Furthermore, this implementation only allowed MPI buffers to grow, but not to shrink. Albers et al. (2022) identified that this puts unnecessary strain on the MPI communication. They therefore introduced spike compression which only sends one spike to each target MPI process, which has the necessary knowledge on the target threads saved in an additional data structure. The problem of buffer size is solved *via* introducing the possibility of dynamically shrinking and growing the MPI buffers.

Kumar et al. (2010) and Hines et al. (2011) propose and compare several strategies for spike-exchange on systems including up to 128 K communication end-points (fine-grained BlueGene/P cores) leveraging a communication infrastructure based on non-blocking neighborhood collectives. The proposed approach has several points of strength that have not yet been exploited in this paper, for several reasons. First, communication steps are performed every ms (the minimum axo-synaptic delay in their model), while the integration step is set at 0.1 ms. Some of the authors of the present paper already exploited this strategy (e.g., in Pastorelli et al., 2019) demonstrating its substantial merit in reducing the communication/computation time ratio. However, the minimal connection delay in the 32-area model under consideration is not higher than the integration step, so this prevents the application of the method in the current paper. However, this will be considered for multi-area models with inter-areal connection delays substantially longer than the integration step. Second, in Kumar et al. (2010) and Hines et al. (2011) communication and computation are overlapped by further dividing the communication step in two alternating temporal steps (A and B, with spikes produced during the time window A sent during the B window, and vice versa). Substantial minimal inter-areal connection delays are again a precondition for the overlapping of computation and communication, but it must also be supported by adequate infrastructure in the simulation engine. This technique further reduces the overhead of communication down to values that are comparable with the computation cost even for highly simplified neural integration models. The technique should be surely considered for implementation in NEST GPU and NEST.

Concerning the merit of distributing the neurons among nodes according to their spatial locality (as in the NEST GPU implementation), there are several substantial differences between the spike-exchange algorithmic exploration proposed by Hines et al. (2011) (from 8 to 128 K small memory footprint MPI end-points in the BlueGene/P system) and our discussion that uses as end-points of MPI communication 32 large memory node systems. Hines et al. (2011) analyze the effect of round-robin vs. a consecutive distribution of neurons among processing nodes for models with random vs. local connectivity, in two spiking rate regimes, named "Noburst" and "Burst." In the first one each neuron of the network fires with a uniform distribution over the entire simulation time interval, whereas in the second regime groups of contiguous neurons successively fire at five times their normal rate for a 50 ms period. On a system with 16 K communication end-points, they demonstrated that the consecutive distribution is highly advantageous for locally connected networks with homogeneous firing rates, while it is only moderately advantageous when all neurons on a processor show the bursting regime. In our case, with larger memory per node and in general for horizontal projections strongly decaying with spatial distance, mapping the laterally incoming synapses on the memory of a single GPU eliminates the need to use collective communications for a much larger fraction of spikes than when mapping a structured network on a system with 16K communication end-points. Indeed, the average ratio between the number of spikes that an area sends to all the other areas and the total number of spikes that it emits is around 3%, with a maximum across areas of around 16% (see **Supplementary Material**).

Regarding NEST GPU performance on a learning case (i.e., on a network model that employs plastic synapses), in Golosio et al. (2021b) we evaluated the library's performance in the simulation of networks with spike-timing-dependent plasticity (STDP) (Gütig et al., 2003) on a single GPU. In general, multi-GPU/MPI simulation performance can significantly depend on the way synaptic parameters of STDP connections between neurons on different MPI processes are updated. The availability of presynaptic spikes and synaptic representation on the same process as the target neurons, as in NEST GPU (and NEST), enables efficient weight updates because they can be managed locally. However, simulation of plastic networks will be covered in future work.

The inclusion of NEST GPU into the NEST Initiative facilitates further integration with the NEST simulator, opening it up to GPU-based spiking neural network simulations. Currently there is ongoing work oriented to an adaptation of the models to be consistent with the NEST simulator, and a software interface has also been developed (Golosio et al., 2020) which enables creating NEST-NEST GPU hybrid networks. Indeed, as reported in Golosio et al. (2021b), the Python interfaces of NEST and NEST GPU are highly similar, making the porting of NEST scripts to the new simulator quite simple. Not only the possibility of using GPU hardware, but also the optimized MPI algorithm for spike communication will greatly improve user experience in simulating large-scale spiking neural networks. In fact, as shown in **Figure 8**, the time reduction in the communication between MPI processes is the main contributor to the better performance of NEST GPU compared to NEST. A speed-up in the neuron updates and delivery of local spikes is also present, and can be further enhanced with the use of more performant GPU-based HPC solutions.

In summary, the NEST GPU simulator (Golosio et al., 2021b) is able to outperform NEST in the state propagation phase of the simulation of a large-scale spiking model, and this speed-up can be essential for simulations covering long stretches of biological time. The performance might be even further enhanced with the help of the latest GPU hardware, which could lead to a steeper performance difference between CPU-based simulators and GPU-based ones. Indeed the use of multi-GPU nodes in a cluster, together with the increase in GPU memory and therefore the possibility of allocating more neurons on a single GPU card, would allow a considerable reduction in the spike communication time. More generally, the GPU industry is growing rapidly, with excellent prospects for the performance of future cards, which from generation to generation significantly increase performance.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found below: https://github.com/gmtiddia/ngpu_multi_area_model_simulation.

AUTHOR CONTRIBUTIONS

GT, JA, and JP performed the simulations and data analysis with guidance by JS, BG, and SvA. JP, JA, JS, and SvA contributed to the development of the NEST implementation of the multi-area model. BG, FS, GT, EP, VF, and PP contributed to the development of the NEST GPU implementation of the multi-area model. GT, BG, and SvA wrote the first manuscript draft. JS, GT, JA, EP, PP, VF, and SvA revised the manuscript. BG and SvA supervised the project. All authors have read and approved the final manuscript.

FUNDING

This study was supported by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreements No. 945539 (Human Brain Project SGA3) and No. 785907 (Human Brain Project SGA2), the Priority Program 2041 (SPP 2041) Computational Connectomics of the German Research Foundation (DFG), the Helmholtz Association Initiative and Networking Fund under project number SO-092 (Advanced Computing Architectures, ACA), the Joint Lab Supercomputing and Modeling for the Human Brain, and the INFN APE Parallel/Distributed Computing laboratory. We acknowledge the use of Fenix Infrastructure resources, which are partially funded from the European Union's Horizon 2020 research and innovation programme through the ICEI project under the Grant Agreement No. 800858. Open access publication funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 491111487.

ACKNOWLEDGMENTS

We thank Alexander van Meegen for contributing to the testing and maintenance of the NEST code for the multi-area model. We are also grateful to Jose Villamar for his valuable contribution to the improvement of the NEST GPU code. The authors gratefully acknowledge the computing time granted by the JARA Vergabegremium and provided on the JARA Partition part of the supercomputer JURECA at Forschungszentrum Jülich (computation grant JINB33).

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2022.883333/full#supplementary-material

REFERENCES

Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., et al. (2015). Truenorth: design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Trans. Comput. Aided Design Integrat. Circ. Syst*. 34, 1537–1557. doi: 10.1109/TCAD.2015.2474396

Albers, J., Pronold, J., Kurth, A. C., Vennemo, S. B., Haghighi Mood, K., Patronis, A., et al. (2022). A modular workflow for performance benchmarking of neuronal network simulations. *Front. Neuroinform*. 16, 837549. doi: 10.3389/fninf.2022.837549

Alonso-Nanclares, L., Gonzalez-Soriano, J., Rodriguez, J., and DeFelipe, J. (2008). Gender differences in human cortical synaptic density. *Proc. Natl. Acad. Sci. U.S.A.*105, 14615–14619. doi: 10.1073/pnas.0803652105

Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., et al. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *J. Comp. Neurol*. 513, 532–541. doi: 10.1002/cne.21974

Babapoor-Farrokhran, S., Hutchison, R. M., Gati, J. S., Menon, R. S., and Everling, S. (2013). Functional connectivity patterns of medial and lateral macaque frontal eye fields reveal distinct visuomotor networks. *J. Neurophysiol*. 109, 2560–2570. doi: 10.1152/jn.01000.2012

Bakker, R., Thomas, W., and Diesmann, M. (2012). CoCoMac 2.0 and the future of tract-tracing databases. *Front. Neuroinform*. 6, 30. doi: 10.3389/fninf.2012.00030

Binzegger, T., Douglas, R. J., and Martin, K. A. C. (2004). A quantitative map of the circuit of cat primary visual cortex. *J. Neurosci*. 39, 8441–8453. doi: 10.1523/JNEUROSCI.1400-04.2004

Brette, R., and Goodman, D. F. M. (2012). Simulating spiking neural networks on GPU. *Network* 23, 167–182. doi: 10.3109/0954898X.2012.730170

Capone, C., Pastorelli, E., Golosio, B., and Paolucci, P. S. (2019). Sleep-like slow oscillations improve visual classification through synaptic homeostasis and memory association in a thalamo-cortical model. *Sci. Rep*. 9, 8990–8911. doi: 10.1038/s41598-019-45525-0

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge: Cambridge University Press.

Chou, T.-S., Kashyap, H. J., Xing, J., Listopad, S., Rounds, E. L., Beyeler, M., et al. (2018). "CARLsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters," in *2018 International Joint Conference on Neural Networks (IJCNN)* (Rio de Janeiro: IEEE).

Chu, C. C. J., Chien, P. F., and Hung, C. P. (2014b). Tuning dissimilarity explains short distance decline of spontaneous spike correlation in macaque V1. *Vision Res*. 96, 113–132. doi: 10.1016/j.visres.2014.01.008

Chu, C. C. J., Chien, P. F., and Hung, C. P. (2014a). *Multi-Electrode Recordings of Ongoing Activity and Responses to Parametric Stimuli in Macaque V1*. Available online at: https://crcns.org/data-sets/vc/pvc-5/about

Cragg, B. G. (1975). The density of synapses and neurons in normal, mentally defective ageing human brains. *Brain* 98, 81–90. doi: 10.1093/brain/98.1.81

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Denker, M., Yegenoglu, A., and Grün, S. (2018). "Collaborative HPC-enabled workflows on the HBP Collaboratory using the Elephant framework," in *Neuroinformatics 2018* (Jülich), P19.

Felleman, D. J., and Van Essen, D. C. (1991). Distributed hierarchical processing in the primate cerebral cortex. *Cereb. Cortex* 1, 1–47. doi: 10.1093/cercor/1.1.1

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The SpiNNaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Garrido, J. A., Carrillo, R. R., Luque, N. R., and Ros, E. (2011). "Event and time driven hybrid simulation of spiking neural networks," in *Advances in Computational Intelligence* (Berlin; Heidelberg: Springer), 554–561.

Golosio, B., De Luca, C., Pastorelli, E., Simula, F., Tiddia, G., and Paolucci, P. S. (2020). "Toward a possible integration of NeuronGPU in NEST," in *NEST Conference 2020* (Ås), 7.

Golosio, B., De Luca, C., Capone, C., Pastorelli, E., Stegel, G., Tiddia, G., et al. (2021a). Thalamo-cortical spiking model of incremental learning combining perception, context and NREM-sleep. *PLoS Comput. Biol*. 17, 1–26. doi: 10.1371/journal.pcbi.1009045

Golosio, B., Tiddia, G., De Luca, C., Pastorelli, E., Simula, F., and Paolucci, P. S. (2021b). Fast simulations of highly-connected spiking cortical models using GPUs. *Front. Comput. Neurosci*. 15, 13. doi: 10.3389/fncom.2021.627620

Grübl, A., Billaudelle, S., Cramer, B., Karasenko, V., and Schemmel, J. (2020). Verification and design methods for the BrainScaleS neuromorphic hardware system. *J. Signal Process. Syst*. 92, 1277–1292. doi: 10.1007/s11265-020-01558-7

Gütig, R., Aharonov, R., Rotter, S., and Sompolinsky, H. (2003). Learning input correlations through nonlinear temporally asymmetric hebbian plasticity. *J. Neurosci*. 23, 3697–3714. doi: 10.1523/JNEUROSCI.23-09-03697.2003

Güttler, G. M. (2017). *Achieving a Higher Integration Level of Neuromorphic Hardware Using Wafer Embedding*. Heidelberg: Heidelberg University Library. doi: 10.11588/HEIDOK.00023723

Hahne, J., Diaz, S., Patronis, A., Schenck, W., Peyser, A., Graber, S., et al. (2021). NEST 3.0. Available online at: https://zenodo.org/record/4739103/export/hx#.YqHBUiNByYM

Heittmann, A., Psychou, G., Trensch, G., Cox, C. E., Wilcke, W. W., Diesmann, M., et al. (2022). Simulating the cortical microcircuit significantly faster than real time on the ibm inc-3000 neural supercomputer. *Front. Neurosci*. 15, 728460. doi: 10.3389/fnins.2021.728460

Hines, M., Kumar, S., and Schürmann, F. (2011). Comparison of neuronal spike exchange methods on a Blue Gene/P supercomputer. *Front. Comput. Neurosci*. 5, 49. doi: 10.3389/fncom.2011.00049

Hoang, R., Tanna, D., Jayet Bray, L., Dascalu, S., and Harris, F. (2013). A novel cpu/gpu simulation environment for large-scale biologically realistic neural modeling. *Front. Neuroinform*. 7, 19. doi: 10.3389/fninf.2013.00019

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Front. Neuroinformat.* 12:2. doi: 10.3389/fninf.2018.00002

Knight, J. C., Komissarov, A., and Nowotny, T. (2021). PyGeNN: A Python library for GPU-enhanced neural networks. *Front. Neuroinform*. 15, 659005. doi: 10.3389/fninf.2021.659005

Knight, J. C., and Nowotny, T. (2018). GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Front Neurosci*. 12. doi: 10.3389/fnins.2018.00941

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci*. 1, 136–142. doi: 10.1038/s43588-020-00022-7

Kumar, S., Heidelberger, P., Chen, D., and Hines, M. (2010). "Optimization of applications with non-blocking neighborhood collectives *via* multisends on the blue gene/p supercomputer," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (Atlanta, GA: IEEE), 1–11.

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., et al. (2019). Coreneuron: an optimized compute engine for the neuron simulator. *Front. Neuroinform*. 13, 63. doi: 10.3389/fninf.2019.00063

Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2022). Sub-realtime simulation of a neuronal network of natural density. *Neuromorph. Comput. Eng*. 2, 021001. doi: 10.1088/2634-4386/ac55fc

Marjanović, V., Labarta, J., Ayguadé, E., and Valero, M. (2010). "Overlapping communication and computation by using a hybrid mpi/smpss approach," in *Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10* (New York, NY: Association for Computing Machinery), 5–16.

Markov, N. T., Ercsey-Ravasz, M. M., Ribeiro Gomes, A., Lamy, C., Magrou, L., Vezoli, J., et al. (2014). A weighted and directed interareal connectivity matrix for macaque cerebral cortex. *Cereb. Cortex* 24, 17–36. doi: 10.1093/cercor/bhs270

Markov, N. T., Misery, P., Falchier, A., Lamy, C., Vezoli, J., Quilodran, R., et al. (2011). Weight consistency specifies regularities of macaque cortical networks. *Cereb. Cortex* 21, 1254–1272. doi: 10.1093/cercor/bhq201

Nguyen, H. (2007). *Gpu Gems 3. Addison-Wesley Professional, 1st Edn*. Boston, MA: Addison-Wesley

Parzen, E. (1962). On estimation of a probability density function and mode. *Ann. Math. Stat*. 33, 1065–1076. doi: 10.1214/aoms/1177704472

Pastorelli, E., Capone, C., Simula, F., Sanchez-Vives, M. V., Del Giudice, P., Mattia, M., et al. (2019). Scaling of a large-scale simulation of synchronous slow-wave and asynchronous awake-like activity of a cortical model with long-range interconnections. *Front. Syst. Neurosci*. 13, :33. doi: 10.3389/fnsys.2019.00033

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. (2011). Scikit-learn: machine learning in python. *J. Mach. Learn. Res*. 12, 2825–2830. Available online at: https://scikit-learn.org/stable/about.html

Potjans, T. C., and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Rhodes, O., Peres, L., Rowley, A. G. D., Gait, A., Plana, L. A., Brenninkmeijer, C., et al. (2020). Real-time cortical simulation on neuromorphic hardware. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci*. 378, 20190160. doi: 10.1098/rsta.2019.0160

Rosenblatt, M. (1956). Remarks on some nonparametric estimates of a density function. *Ann. Math. Stat*. 27, 832–837. doi: 10.1214/aoms/1177728190

Sanders, J., and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Upper Saddle River, NJ: Addison-Wesley.

Schmidt, M., Bakker, R., Hilgetag, C. C., Diesmann, M., and van Albada, S. J. (2018a). Multi-scale account of the network structure of macaque visual cortex. *Brain Struct. Funct*. 223, 1409–1435. doi: 10.1007/s00429-017-1554-4

Schmidt, M., Bakker, R., Shen, K., Bezgin, G., Diesmann, M., and van Albada, S. J. (2018b). A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas. *PLoS Comput. Biol*. 14, e1006359. doi: 10.1371/journal.pcbi.1006359

Schuecker, J., Schmidt, M., van Albada, S. J., Diesmann, M., and Helias, M. (2017). Fundamental activity constraints lead to specific interpretations of the connectome. *PLoS Comput. Biol*. 13, e1005179. doi: 10.1371/journal.pcbi.1005179

Silverman, B. W. (1986). *Density Estimation for Statistics and Data Analysis*. London: Chapman and Hall.

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *Elife* 8, e47314. doi: 10.7554/eLife.47314

Thörnig, P., and von St. Vieth, B. (2021). JURECA: data centric and booster modules implementing the modular supercomputing architecture at jülich supercomputing centre. *J. Large Scale Res. Facilit*. 7, A182. doi: 10.17815/jlsrf-7-182

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Front. Neurosci*. 12, 291. doi: 10.3389/fnins.2018.00291

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., et al. (2020). SciPy 1.0: fundamental algorithms for scientific computing in python. *Nat. Methods* 17, 261–272. doi: 10.1038/s41592-020-0772-5

Vitay, J., Dinkelbach, H. U., and Hamker, F. H. (2015). ANNarchy: a code generation approach to neural simulations on parallel hardware. *Front. Neuroinform*. 9, 19. doi: 10.3389/fninf.2015.00019

von St. Vieth, B. (2021). Jusuf: Modular tier-2 supercomputing and cloud infrastructure at jülich supercomputing centre. *J. Large Scale Res. Facilit*. 7, A179. doi: 10.17815/jlsrf-7-179

Waskom, M. L. (2021). seaborn: statistical data visualization. *J. Open Source Softw*. 6, 3021. doi: 10.21105/joss.03021

Wunderlich, T., Kungl, A. F., Müller, E., Hartel, A., Stradmann, Y., Aamir, S. A., et al. (2019). Demonstrating advantages of neuromorphic computation: a pilot study. *Front. Neurosci*. 13, 260. doi: 10.3389/fnins.2019.00260

Yavuz, E., Turner, J., and Nowotny, T. (2016). GeNN: a code generation framework for accelerated brain simulations. *Sci Rep*. 6, 18854. doi: 10.1038/srep18854

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

ORIGINAL RESEARCH
published: 05 July 2022
doi: 10.3389/fnins.2022.881598

Beyond LIF Neurons on Neuromorphic Hardware

Mollie Ward and Oliver Rhodes*

Department of Computer Science, University of Manchester, Manchester, United Kingdom

Neuromorphic systems aim to provide accelerated low-power simulation of Spiking Neural Networks (SNNs), typically featuring simple and efficient neuron models such as the Leaky Integrate-and-Fire (LIF) model. Biologically plausible neuron models developed by neuroscientists are largely ignored in neuromorphic computing due to their increased computational costs. This work bridges this gap through implementation and evaluation of a single compartment Hodgkin-Huxley (HH) neuron and a multi-compartment neuron incorporating dendritic computation on the SpiNNaker, and SpiNNaker2 prototype neuromorphic systems. Numerical accuracy of the model implementations is benchmarked against reference models in the NEURON simulation environment, with excellent agreement achieved by both the fixed- and floating-point SpiNNaker implementations. The computational cost is evaluated in terms of timing measurements profiling neural state updates. While the additional model complexity understandably increases computation times relative to LIF models, it was found a wallclock time increase of only 8× was observed for the HH neuron (11× for the mutlicompartment model), demonstrating the potential of hardware accelerators in the next-generation neuromorphic system to optimize implementation of complex neuron models. The benefits of models directly corresponding to biophysiological data are demonstrated: HH neurons are able to express a range of output behaviors not captured by LIF neurons; and the dendritic compartment provides the first implementation of a spiking multi-compartment neuron model with XOR-solving capabilities on neuromorphic hardware. The work paves the way for inclusion of more biologically representative neuron models in neuromorphic systems, and showcases the benefits of hardware accelerators included in the next-generation SpiNNaker2 architecture.

Keywords: SpiNNaker, dendritic computation, Hodgkin-Huxley, neuronal modeling, neuromorphic computing, spiking neural networks

Edited by:
Markus Diesmann,
Helmholtz Association of German Research Centres (HZ), Germany

Reviewed by:
Federico Corradi,
Eindhoven University of Technology, Netherlands
Marcel Stimberg,
Sorbonne Université, France

****Correspondence:***
Mollie Ward
mollie.ward@manchester.ac.uk

Specialty section:
This article was submitted to Neuromorphic Engineering, a section of the journal Frontiers in Neuroscience

Received: *22 February 2022*
Accepted: *10 June 2022*
Published: *05 July 2022*

Citation:
Ward M and Rhodes O (2022) Beyond LIF Neurons on Neuromorphic Hardware.
Front. Neurosci. 16:881598.
doi: 10.3389/fnins.2022.881598

1. INTRODUCTION

A vast array of brain modeling techniques exist to simulate brain activity with a view to gaining understanding of the human brain. These techniques range from mathematical representations of individual molecules within neurons to whole-brain simulations. One widely used method for simulation of brain activity is through the use of neural networks. Spiking Neural Networks (SNNs) use biologically-inspired models of neurons to carry out computation with the aim of simulating neural activity and have applications in a number of research areas including computational neuroscience, machine learning, and robotics.

State-of-the-art large scale SNN simulations such as those described by the Blue Brain Project (Markram et al., 2015) aim to mimic brain activity through the use of complex neuron models to advance understanding of the human brain. Scientists were able to accurately reproduce anatomical and physiological features of real biological networks when simulating 0.29 mm^3 of the rat brain. Despite these recent achievements in the complexity and scale of SNNs simulated, simulation of these SNNs consumes considerable power (megawatts) for simulation of very small regions of the brain (Markram et al., 2015). The full simulation involved over 30,000 different neuron models incorporating 13 different ion-channel models, each neuron comprised on average 20,000 differential equations representing synaptic connections and ion-channels. The full simulation required solving of over two billion equations for every second of biological time (Kumbhar et al., 2019). This power consumption is not required in the human brain which demonstrates a remarkable ability for large amounts of fine-scale computation at a fraction of the power (up to 20 watts) and much faster than SNNs simulated on conventional computer hardware which do not run in real-time (Cox and Dean, 2014).

Energy-efficient neuromorphic systems are designed to mimic the brain and provide low-power platforms for simulation of SNNs, providing a potential solution for the high energy requirements of large scale simulations. As neuromorphic computing platforms target real-time large-scale simulations of SNNs, the biological plausibility of neuron models has been largely ignored in favor of simple, efficient neuron models such as the Leaky-Integrate-and-Fire (LIF) neuron model. Such models are favored due to the ease of solving the equations involved: the differential equations can be solved exactly with a small number of addition and multiplication operations. These simple neurons have allowed large-scale SNNs in real-time such as the cortical microcircuit simulated on SpiNNaker (Rhodes et al., 2020). This SNN simulated $\approx$ 1mm^2 of mamillian neocortex, and while this demonstrated the potential of neuromorphic hardware as a neuroscience research tool, the model does not exhibit the fidelity typically explored by the neuroscience research community.

The LIF model falls short of biological plausibility in two main areas: membrane conductance and structure. Membrane conductance is described with a single term in the model but is actually governed by a number of different ion-channels spanning the neural membrane. The flow of ions through these channels gives biological neurons a wide range of firing capabilities not captured with the LIF model, e.g., the ability to respond to identical inputs differently depending on the current state of the neuron and its ion-channels. Structure is simplified in the LIF model to a single point, however in biology, neurons are complex and elongated and incorporate vast branched extensions called dendrites. Dendrites are active structures capable of generating their own action potentials and are believed to contribute significant computational function to biological neurons (Dayan and Abbott, 2005; Poirazi and Papoutsi, 2020).

Neuron models can increase in complexity to capture these simplified biological features and a wide range of spiking neuron models exist. Hodgkin and Huxley (1952) described a biologically inspired model incorporating equations for sodium and potassium ion channels which govern the progression of the action potential. Other models, such as the Izhikevich model (Izhikevich, 2004), aim to capture certain biological characteristics with more efficient non-biologically plausible equations. However, this lack of biological plausibility takes away the ability to explore the effects of incorporating different ion-channels and more complex morphologies than a single point neuron structure. Accurate and efficient ion-channel modeling on neuromorphic hardware would therefore allow exploration of a wide range of biologically inspired models including multi-compartment models describing complex neural morphologies with dendritic compartments (Markram et al., 2015; Gidon et al., 2020). Implementation of more complex neuron models onto neuromorphic systems could provide low-power solutions for large-scale SNN simulations.

Neuron models with increased complexity have been tested in analog and digital neuromorphic systems, demonstrating the importance of this kind of modeling. For example, individual ion-channels have been modeled in an analog circuit (Abu-Hassan et al., 2019). Here, the aim was to design a biologically accurate neuromorphic circuit that responds identically to a biological neuron under any injected current. The authors were able to reproduce biological voltage recordings with 94–97% accuracy. These neurons were built to demonstrate the potential for making synthetic neurons with therapeutic potential for implementation into the central nervous system, therefore do not easily scale up to large SNNs and do not incorporate structural morphology. However, this work demonstrates accurate representation of ion-channel models on neuromorphic systems. Multi-compartment neuron models have also been tested on neuromorphic systems. BrainScalesS (Schemmel et al., 2010) is an analog neuromorphic system that features an Adaptive-Exponential Integrate-and-Fire (AdExp) neuron model. Schemmel et al. (2017) and more rec ently Müller et al. (2022) and Kaiser et al. (2022) expanded this neuron model to capture dendritic computation in multi-compartment approaches. Intel's Loihi (Davies et al., 2018) also offers support for dendritic computation by offering the opportunity to model neurons with multiple compartments. Here, the additional compartments are effectively identical, the only difference being that the "somatic" compartment generates spike output and the "dendritic" compartments do not. While this does enable a concept of dendritic computation through the ability to distribute synaptic input across individual units, there is a lack of biological plausibility as dendrites are actually much more computationally complex, exhibiting non-linear processing of synaptic inputs (Gidon et al., 2020; Poirazi and Papoutsi, 2020).

1.1. Neuromorphic Hardware

While a range of neuromorphic computing systems are currently developed across industry and academia (Schemmel et al., 2010; Benjamin et al., 2014; Furber et al., 2014; Merolla et al., 2014; Davies et al., 2018; Pei et al., 2019), the application of this technology remains limited. While these systems boast impressive performance figures in terms of energy and processing speed, their bespoke architectures are often tailored to particular applications, making it hard to adapt these systems to emerging research problems. The SpiNNaker neuromorphic computing system is selected as the research platform for this work, as its flexibility enables exploration of the target neural models, while constraints such as co-location of memory and processors mean findings remain relevant for the wider neuromorphic research community. The SpiNNaker system is currently an active research platform, with a 1M core machine operating and maintained by the University of Manchester, UK. In parallel to exploring SNN applications on this system, research and development into next-generation hardware is also on-going in the form of the SpiNNaker2 system (Mayr et al., 2019). The two platforms are explored in this work, implementing models on both the SpiNNaker system and a SpiNNaker2 prototype chip (Jib2), to enable comparison and evaluation of performance.

1.1.1. SpiNNaker

SpiNNaker is a massively-parallel many-core digital computing platform, designed for large-scale real-time simulation of SNNs. The system comprises chips assembled into a two-dimensional mesh network, enabling the system to scale to 1M cores. Each individual chip houses 18 cores, network on chip (NoC) and external RAM controller; while each core contains an ARM968, direct memory access controller, communications controller, two timers and other peripherals. Each core has 32 kB instruction and 64 kB data tightly coupled memory (ITCM and DTCM, respectively), with single cycle access. Each chip has an additional 128 MB shared memory, typically accessed *via* DMA, and used to store larger SNN data-structures such as synaptic matrices. Cores operate at 200 MHz, running an event-driven operating system enabling efficient neural processing (Rhodes et al., 2018). Individual cores simulate a collection of neurons using software to solve mathematical models representing neural dynamics. These models are solved in discrete time, with the goal of matching the simulation timestep to the time required to process the state update, in order to achieve real-time simulation. Models are programmed in C, and compiled into ARM code using the GCC toolchain. As the core has no floating-point unit, all models are coded using fixed-point arithmetic, with the ISO standard *accum* type favored for the majority of variables. This 32-bit type is a signed representation, with 16 integer and 15 fractional bits, and lower/upper limits of 0.000030517578125 and 65535.999969482421875, respectively (Rhodes et al., 2018). While transcendental functions are also not supported in hardware, division and exponential functions are available in software, requiring approximately 100 clock cycles each. This framework enables real-time implementation of multiple neuron models, including the current- and conductance-based LIF and Izhikevich neurons (Rhodes et al., 2018).

1.1.2. Jib2—SpiNNaker2 Prototype

While the architectural principles are similar, the goals of SpiNNaker2 are to increase the number of cores by a factor of 10, and to increase the number of simulated neurons by a factor of 50, while staying within the same power budget. The system will use an ARM cortex M4 core, with adaptive body biasing to enable increased clock frequencies during periods of high load—switchable from 150 to 300 MHz. Additional performance increases are expected from inclusion of hardware accelerators for specific operations common in neural processing, including random number generation, e^x, and a single-precision floating point unit (Mikaitis, 2020). The experiments reported in this work are performed on a SpiNNaker2 prototype system known as Jib2, containing 8 processing elements (PE) arranged in two quad processing elements (QPEs) (Höppner et al., 2021). Each PE has an ARM cortex M4 in addition to the above mentioned accelerators, and runs compiled C code in a similar fashion to SpiNNaker (Section 1.1.1), again compiled with the GCC toolchain. PEs each have 128 kB of fast access SRAM, for combined instruction and data storage. Jib2 has variable voltage-frequency levels enabling low-power operation and workload-dependent scaling of clock frequency. The experiments reported in this work are performed with the core running with voltage-frequency settings of 0.5 V–150 MHz and 0.8 V–300 MHz.

1.2. NEURON Simulation Environment

New models implemented on neuromorphic hardware need to be benchmarked again standard methods used in the industry in order to ensure the models are accurate and valid. NEURON is a widely used platform for simulation of individual neuron models and networks of neurons and was designed specifically to simulate equations describing nerve cells. NEURON was chosen as the benchmark for models as it is a standard tool in the research field. It provides an environment for implementing biologically realistic models with a focus on incorporation of multiple ion-channel models and complex branched neuronal morphologies (Hines and Carnevale, 1997). The activity of neurons is modeled using the cable equation in which neurons are treated as trees consisting of a number of compartments. Each compartment is an unbranched cable which can be split into sections and each section can contain its own biophysical properties through different ion-channels. Each section is described by its membrane potential and a set of coupled differential equations are solved for each section within a neuron to compute the evolution of membrane potential inside the neuron over time. The general form of the cable equation for each section, j is:

$$c_j \frac{dv_j}{dt} + I_m^j = \sum_k \frac{v_k - v_j}{r_{jk}} \quad (1)$$

where c_j is the membrane capacitance of the section, v_j is the membrane voltage of the section, t is time, the ionic component I_m^j includes all currents through ion-channels. $\sum_k \frac{v_k - v_j}{r_{jk}}$ represents the sum of axial currents entering from neighboring sections, v_k is the membrane voltage of the neighboring section and r_{jk} is the resistive coupling between compartments. This differential equation is coupled to an additional set of differential equations describing the active states of any ion-channels incorporated into the model. This

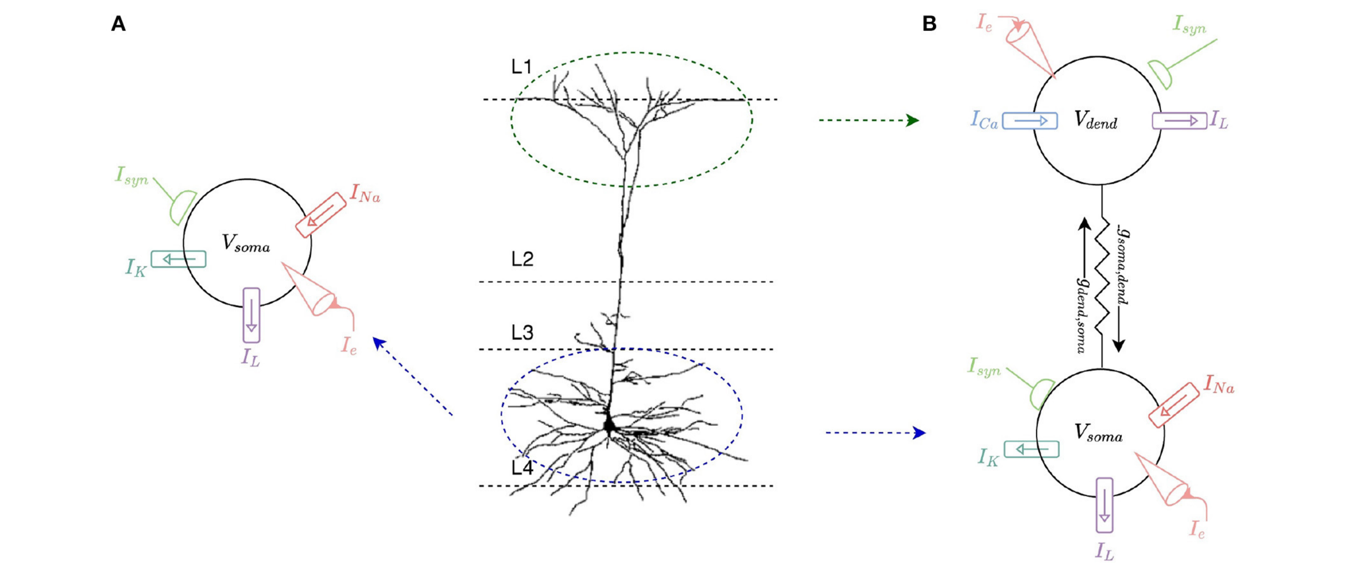


FIGURE 1 | (A) Single-compartment HH model of a L2/L3 pyramidal neuron. The soma of the cell is modeled with a leak current, I_L, as well as sodium, I_{Na}, and potassium, I_K, currents. Current can be injected into the model, I_e, and it can receive multiple synaptic inputs, I_{syn}. **(B)** Two-compartment model of an L2/L3 pyramidal neuron consisting of a somatic and a dendritic compartment. The dendritic compartment incorporates a calcium ion channel current, I_{Ca}, and a leak current, I_L. The somatic compartment incorporates the same leak current, I_L, as well as sodium, I_{Na}, and potassium, I_K, currents. The compartments are connected by coupling conductances, $g_{soma,dend}$ and $g_{dend,soma}$. Current can be injected into either compartment, I_e, and each compartment can receive multiple synaptic inputs, I_{syn}.

leads to a set of coupled differential equations which need to be solved at each simulation time step. NEURON uses a backward Euler implicit integration method as standard (Hines and Carnevale, 1997). Each time step update is divided into a set of operations which are performed in order to progress from one time step to the next. These operations include a spike delivery step where synapses are activated by incoming spikes, a matrix assembly step where the ionic and synaptic currents are calculated, a matrix resolution step in which the membrane potential is calculated, a state variable update step in which the ion-channel states are updated, and a threshold detection step in which membrane voltages are checked against threshold values to determine whether a firing condition has been met (Kumbhar et al., 2019). The NEURON platform was designed specifically to model systems of neurons incorporating easy to configure biological data (branched morphologies and ion-channel models) and is therefore widely used by the computational neuroscience community.

2. METHODS

This work involves modeling a L2/L3 pyramidal neuron[1]. These neurons comprise approximately two-thirds of neurons in the cerebral cortex of human brains and are key for a large number of cognitive processes, making them prime candidates for mathematical modeling and simulation. Differential equations are used in individual models of spiking neurons to calculate a neuron's membrane potential over time. The change of the membrane potential is proportional to the rate of change of charge build up, i.e., the rate of change of ion flow into and out of the cell, and hence is proportional to the amount of current entering the cell. The amount of current entering the cell is based on the membrane and synaptic conductances and any current injected into the cell. The soma of the neuron is first modeled as a single compartment Hodgkin-Huxley (HH) (1952) model with sodium and potassium ion-channels. This single-compartment model is then expanded to a two-compartment model to capture a dendritic compartment which incorporates a calcium ion-channel model (**Figure 1**).

2.1. Ion-Channels and HH Neurons

A single compartment HH neuron model is built to represent the somatic membrane potential of a typical L2/L3 pyramidal neuron. The model describes the region in which action potentials are generated (**Figure 1A**). The somatic model incorporates sodium (I_{Na}), potassium (I_K), and leak (I_L) currents with corresponding maximal conductances $g_{Na} = 0.12$ S/cm^2, $g_K = 0.036$ S/cm^2, and $g_L = 0.0003$ S/cm^2, and reversal potentials $E_{Na} = +50$ mV, $E_K = -77$ mV and $E_L = -54.3$ mV. The rate functions for somatic ion channels are modeled as described by Hodgkin and Huxley (1952) and total membrane current in the somatic compartment is calculated as the sum of these three individual currents:

$$I_{soma} = g_L(V_{soma} - E_L) + g_K n^4(V_{soma} - E_K) + g_{Na} m^3 h(V_{soma} - E_{Na}) \qquad (2)$$

[1] All models discussed in the text are available at https://github.com/mollie-ward/beyondLIFNeurons, and can be compiled and run on any GCC compatible platform.

where n, m and h are gating variables for the ion channels; n and m are activation variables for K$^+$ and Na$^+$ ion channels, respectively, and h is an inactivation variable for Na$^+$ channels. n, m, and h, like V_{soma}, all vary over time and can be modeled by:

$$\tau_n(V)\frac{dn}{dt} = n_\infty(V) - n \tag{3}$$

where

$$\tau_n(V) = \frac{1}{\alpha_n(V) + \beta_n(V)} \quad \text{and} \quad n_\infty(V) = \frac{\alpha_n(V)}{\alpha_n(V) + \beta_n(V)} \tag{4}$$

with similar equations for m and h. $\alpha_n(V)$ and $\beta_n(V)$ are the opening and closing variables for the K$^+$ channel, $\alpha_m(V)$ and $\beta_m(V)$ are the opening and closing variables for the Na$^+$ channel and $\alpha_h(V)$ and $\beta_h(V)$ are the key inactivation and de-inactivation variables for the Na$^+$ channel. Rate functions for K$^+$ and Na$^+$ conductances are parameterized according to (Dayan and Abbott, 2005):

$$\alpha_n = \frac{0.01(V_{soma} + 55)}{1 - \exp(-0.1(V_{soma} + 55))} \quad \text{and} \quad \beta_n = 0.125\exp(-0.0125(V_{soma} + 65)) \tag{5}$$

$$\alpha_m = \frac{0.1(V_{soma} + 40)}{1 - \exp(-0.1(V_{soma} + 40))} \quad \text{and} \quad \beta_m = 4\exp(-0.0556(V_{soma} + 65)) \tag{6}$$

$$\alpha_h = 0.07\exp(-0.05(V_{soma} + 65)) \quad \text{and} \quad \beta_h = \frac{1}{1 + \exp(-0.1(V_{soma} + 35))} \tag{7}$$

The soma fires action potentials in response to injected current (I_e) and excitatory synaptic input (I_{syn}). The progression of the membrane potential is governed by the ion channel currents described. **Figure 2** shows the somatic membrane potential over time in response to two current injections, and corresponding ion-channel parameters m, n, and h. A small, constant current injection can cause one somatic action potential to fire as the ion-channel parameters stabilize and adapt to the injected current (**Figure 2A**). A larger, constant current injection causes repeated firing of somatic action potentials and as the ion-channel parameters do not stabilize, firing is constant (**Figure 2B**). A LIF neuron is not able to adapt in this way and is either firing constantly or not firing at all.

2.1.1. Numerical Methods

The neuron is modeled as an equipotential sphere such that the same electrical potential exists across the whole surface and hence the entire neuron can be described with a single membrane potential in a single compartment model. The ion-channels described in Section 2.1 (Equations 2–4) are incorporated into the general equation for a single compartment neuron in which the membrane voltage (V_{soma}) is modeled over time. The progression of membrane voltage is calculated at discrete timesteps with interval $\Delta t = 0.1$ms.

$$C_m\frac{dV_{soma}}{dt} = -I_m + \frac{I_e}{A} \tag{8}$$

Membrane capacitance (C_m) is uniformly set to 1 μFcm^2 over the neuron. The conductance per unit area (i_m) is defined in Equation (2) (S/cm^2), I_e^μ is the total electrode current flowing into the compartment (nA) and A is the area of the neuron (mm^2). Equation (8), combined with Equations (2), (3), and the corresponding equations for m and h, make up a system of ordinary differential equations (ODEs) where the rates of change of more than one variable are described: membrane voltage (V_{soma}), sodium activation parameter (m), sodium inactivation parameter (h), and potassium activation parameter (n) (**Figure 2**).

2.2. Multi-Compartment Modeling

A two-compartment neuron morphology consisting of a somatic compartment and a dendritic compartment is designed incorporating ion-channel currents. Inspiration is drawn from the multi-compartment neuron model presented by Gidon et al. (2020) with the aim of simplifying this model in order to make it suitable for implementation on neuromorphic hardware while preserving the higher level L2/L3 pyramidal neural cell capabilities demonstrated. The dendritic compartment represents the apical dendrites and the somatic compartment represents the soma and basal dendrites (**Figure 1B**). For the somatic compartment, HH sodium and potassium ion-channels described in Section 2.1 are implemented. For the dendritic compartment, a calcium channel introduced by Gidon et al. (2020) is implemented in an attempt to capture the L2/L3 pyramidal neural cell firing dynamics demonstrated by the authors.

2.2.1. Dendritic Currents

The dendritic compartment is modeled with the same leak current (I_L) as the soma and a calcium current (I_{dCaAP}) as described in Gidon et al. (2020). The calcium current in the dendritic compartment gives the compartment the ability to fire its own action potentials (independent of the somatic action potentials). These dendritic calcium action potentials are known as dCaAPs. The dCaAP current is activated when the dendritic membrane potential (V_{dend}) crosses a threshold value ($V_{thresh} = -36$mV):

$$I_{dCaAP} = -\omega K(v)(A - B) \tag{9}$$

with weight, $\omega = 3$ (dimensionless). When V_{dend} crosses the threshold, V_{thresh}, the dCaAP is activated: the activation function of the dendrite, $K(v)$, is calculated and the time of dCaAP activation, t', is set to the current timestep, t.

$$K(v) = \exp(\frac{-F(V_{dend} - V_{thresh})}{\tau_K}) \tag{10}$$

where F is a normalization factor $F = 1/(V_{thresh} - V_{rest})$ and τ_K is the dCaAP amplitude decay constant $\tau_K = 0.3$ (dimensionless).

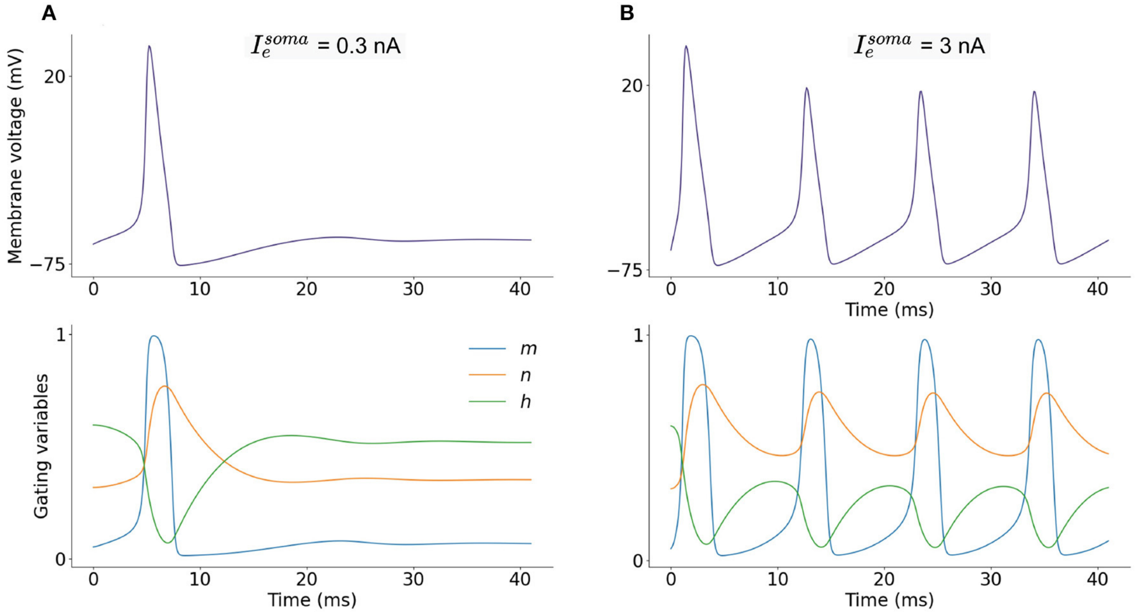


FIGURE 2 | (A) Single-compartment model of a L2/L3 pyramidal neuron with injected current I_e^{soma} = 0.3 nA and corresponding dimensionless ion-channel activation parameters *m*, *n*, and *h* which govern sodium, I_{Na}, and potassium, I_K, currents. **(B)** Current injection of I_e^{soma} = 3 nA and corresponding *m*, *n* and *h* progression.

The dCaAP current has a 200 ms refractory period in which it cannot fire.

A and *B* describe the rise and decay of the dCaAP current and are described by sigmoidal functions:

$$A = \frac{1}{1 + \exp(-\frac{(t-t')}{\tau_A})} \tag{11}$$

$$B = \frac{1}{1 + \exp(-\frac{(t-(t'+\Delta t')}{\tau_A})} \tag{12}$$

where t' is the time of dCaAP activation, $\Delta t' = 21$ms, $\tau_A = 3$ms and $\tau_B = 0.4$ms.

The total membrane current in the dendritic compartment is calculated as the sum of the dCaAP current and the leak current:

$$I_{dend} = g_L(V_{dend} - E_L) + I_{dCaAP} \tag{13}$$

Current flows from the dendritic compartment to the somatic compartment such that injected current into the dendrite can cause somatic action potentials to fire (**Figure 3A**) slightly after dendritic action potentials. Firing dynamics of the two compartment model in response to injected current into each compartment is presented in **Figure 3**. Increasing input to the dendritic compartment causes the amplitude of dCaAPs to decrease, this in turn causes somatic action potentials to stop firing as the current flowing to the somatic compartment will decrease with decreased amplitude of dCaAP (**Figure 4**).

2.2.2. Numerical Methods

The single compartment model is described with a single membrane potential, however, membrane voltages actually vary considerably across the expansive surface of a neuron. It is possible to analyse signal propagation within neurons using a mathematical analysis known as "cable theory" (Dayan and Abbott, 2005). A two-compartment neuron is modeled using cable theory which assumes that the membrane potential varies with longitudinal distance along the axon, *x*, enabling it to be expressed as a partial differential equation (PDE) as a function of *x* and time, *t*, $V(x, t)$:

$$c_m \frac{\partial V}{\partial t} = \frac{a}{2R}\frac{\partial^2 V}{\partial x^2} - I_m + I_e \tag{14}$$

where R is the intracellular resistivity (MΩmm^2) and a is the radius (mm^2). Appropriate boundary conditions are defined as the neuron is split into two compartments (soma and dendrite)—assuming membrane potential does not vary across the surface of the compartment—each with their own voltage (V_{soma} and V_{dend}). This allows the continuous membrane potential, $V(x, t)$, to be approximated by a set of membrane potential values in each compartment. Applying these boundary conditions simplifies the PDE to a system of ODEs for each compartment such that each compartment is described by its own membrane potential. For the somatic compartment, V_{soma}:

$$c_m \frac{dV_{soma}}{dt} = -I_m^{soma} + \frac{I_e^{soma}}{A_{soma}} + g_{dend,soma}(V_{soma} - V_{dend}) \tag{15}$$

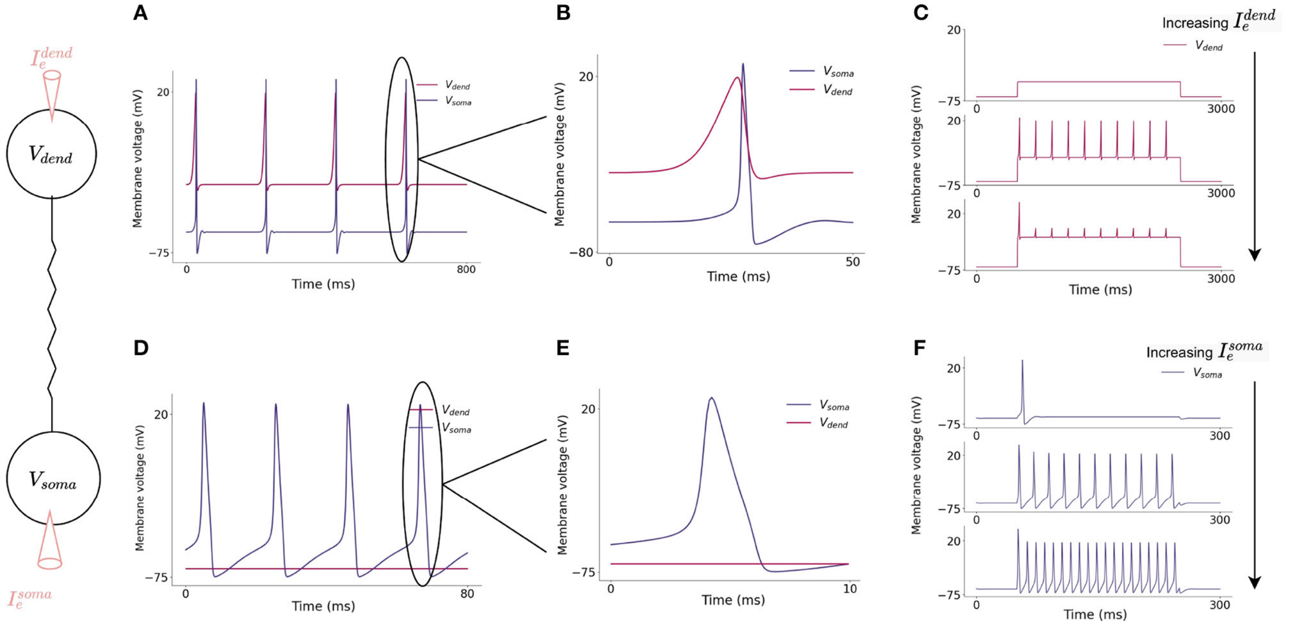


FIGURE 3 | (A) Action potential initiation in dendritic and somatic compartment in response to a 3 nA injected current into the dendritic compartment. Dendritic compartment regularly fires dCaAPs with a refractory period of 200 ms. dCaAPs propagate to the somatic compartment and cause somatic action potentials. The dCaAP therefore precedes somatic action potentials **(B)**. **(C)** Spiking dynamics of the dendritic compartment in response to increasing current injections into the dendrite, the frequency of dendritic spikes remains constant but the amplitude decreases as the current injection increases. **(D)** Action potential initiation in somatic compartment in response to a 10 nA injected current into the somatic compartment, the dendritic compartment does not fire in response to this injected current **(E)**. **(F)** Spiking dynamics of the somatic compartment in response to increasing current injection into the soma, the frequency of somatic spikes increases as the current injection increases.

For the dendritic compartment, V_{dend}:

$$c_m \frac{dV_{dend}}{dt} = -I_m^{dend} + \frac{I_e^{dend}}{A_{dend}} + g_{soma,dend}(V_{dend} - V_{soma}) \quad (16)$$

where I_e^{soma} and I_e^{dend} is the total electrode current flowing into the compartments (nA) and A_{soma} and A_{dend} is the area of the compartments (mm^2). The constants $g_{soma,dend}$ and $g_{dend,soma}$ (nA/mm^2) determine the resistive coupling between neighboring compartments. The membrane current for each compartment, I_m^{soma} and I_m^{dend} are described in Equations (2) and (9). At each timestep, the voltage update equation and corresponding activation parameters (Equations 15 and 16) must be solved, along with the corresponding activation parameters for any present ion channels such as Equation (3) for both compartments. A backwards Euler integration scheme is used due to its robust stability (Hines and Carnevale, 1997) by exploiting the conditional linearity of the ion-channel update equations (Dayan and Abbott, 2005).

2.3. Synaptic Model

Where synapses are incorporated into the model, synaptic currents are modeled as

$$I_{syn} = g_{syn}(V_\mu - E_{syn}) \quad (17)$$

where E_{syn} is the reversal potential for the synaptic current (mV), and g_{syn} is the synaptic conductance (S/cm^2). All synapses model excitatory NMDA connections, therefore $E_{syn} = 0$ mV (Dayan and Abbott, 2005). Synaptic conductance is modeled as:

$$g_{syn} = g_{max} * P_s \quad (18)$$

where $g_{max} = 0.05$ is the maximal conductance and P_s is the probability of neurotransmitter release, modeled as:

$$P_s = P_{max} * (e^{\frac{-t}{\tau_s}}) \quad (19)$$

where the maximal probability of neurotransmitter release $P_{max} = 1$ and $\tau_s = 10$ ms. All synapses were activated at 20 Hz for simulations and are incorporated into dendritic (I_{dend}, Equation 13) or somatic (I_{soma}, Equation 2) currents as an additional term.

2.4. SpiNNaker Implementation

To make the models suitable for implementation on neuromorphic hardware, modifications to the system of equations are sought to decrease the computational load of simulation. Neuron models on SpiNNaker are written in C and compiled into ARM executable code. The SpiNNaker ARM968 CPU provides an energy-efficient core on which to

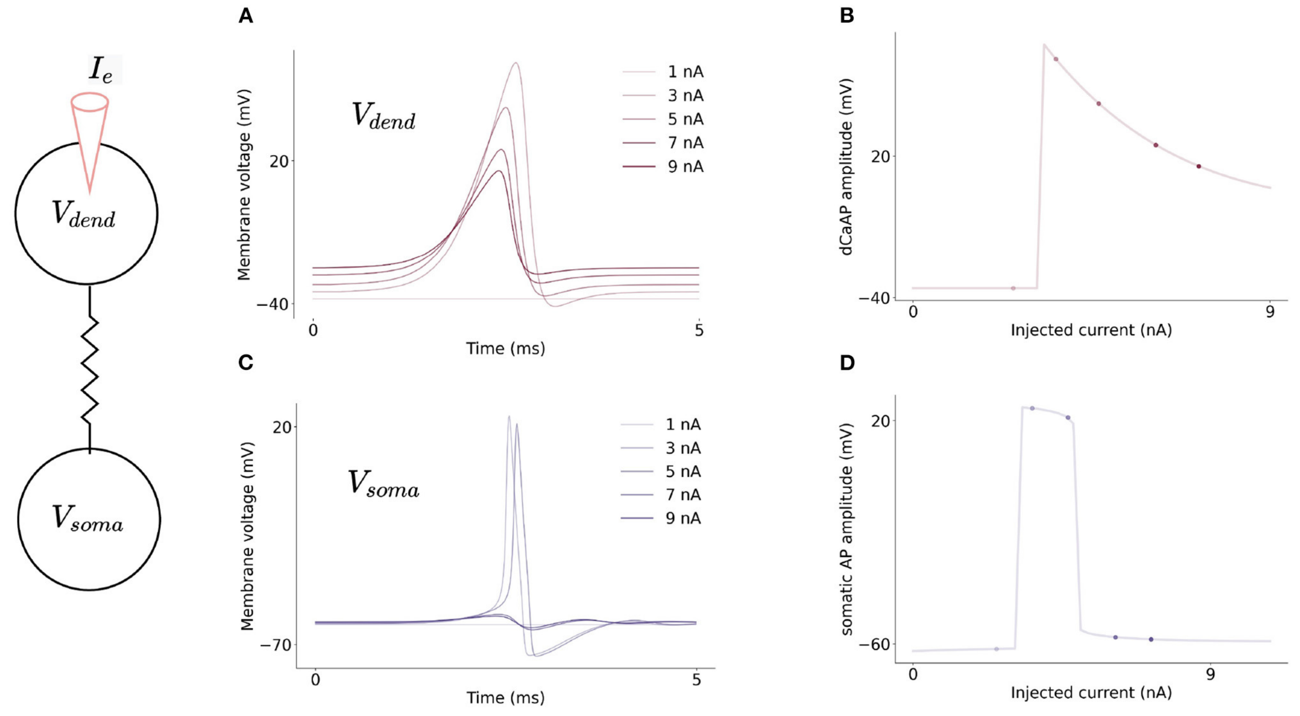


FIGURE 4 | (A) Increasing injected current into the dendritic compartment results in a decrease in dCaAP amplitude. **(B)** The shape of the dCaAP is governed by the dendritic activation function, $K(v)$ (Equation 10), which exhibits a characteristic shape in which the threshold current for dCaAP firing is the maximum dCaAP activation and hence the maximum dCaAP amplitude, the amplitude then decays with decay constant $\tau_K = 0.3$ (dimensionless) (Equation 10). **(C)** Resulting somatic membrane voltage with increasing injected current into the dendritic compartment. Amplitude of somatic action potentials decreases with decreasing dCaAP amplitude. The all-or-nothing nature of these action potentials causes a lack of firing in the soma when the dendritic stimulus intensity gets higher. **(D)** Somatic action potential amplitudes are maximum when the dCaAP activation threshold is reached, they then decrease until the firing threshold for somatic action potentials is no longer reached and the soma stops firing.

TABLE 1 | Time taken to update the membrane voltage in models in μs on the SpiNNaker and Jib2 neuromorphic hardware in one 0.1ms timestep.

		HH		Two comp		LIF
		No LUT	**LUT**	**No LUT**	**LUT**	
SpiNNaker		99.6	8.34	153.67	12.91	0.32
Jib2	300 MHz	2.59	0.73	3.22	1.09	0.09
	150 MHz	5.19	1.45	6.45	2.18	0.19

Values for the time taken to update neuron state in a Leaky-Integrate-and-Fire neuron is also presented for comparison (values from Rhodes et al., 2018).

simulate large-scale neural networks. This core has no floating-point hardware so fixed-point arithmetic is the preferred data representation. Two 32-bit fixed-point arithmetic types are used in this study which are defined in the ISO standard 18037 and are implemented by the GCC compiler. Variables and constants assuming values greater than 1 are defined as an ISO 10837 s16.15 *accum* fixed-point type: a signed 16-integer and 15-fractional bit number. Variables and constant taking values exclusively between 0 and 1 (for example *m*, *n*, and *h*) are defined as ISO 10837 u0.32 *unsigned long fract* fixed-point type: an unsigned 32-fractional bit number. Previous efforts to model more complex neuron models on SpiNNaker (Hopkins and Furber, 2015) reported spike time lag in comparison with reference models, however, later work (Hopkins et al., 2020) demonstrated that errors can be reduced by introducing various rounding techniques including *round-to-nearest* rounding. These methods are implemented here to reduce arithmetic error between the SpiNNaker implementation and the reference model. While most modern processors include hardware support for common arithmetic operations, SpiNNaker lacks hardware support for division and exponential operations. Simplifying assumptions which still give a biologically faithful model were sought enabling pre-calculation of operations such as divisions and exponentials. For example, in Equation (10), F and τ_K are constant to avoid the need to calculate a division at runtime. Lookup tables (LUTs) were also used to eliminate a number of costly calculations: Equations (3)–(7) are replaced by lookup operations.

2.4.1. Ion-Channels and HH Neurons

For potassium and sodium ion-channels, LUTs (using 12 kB of memory) remove nine exponential and twelve division calculations involved in the calculation of gating variables for the ion-channels (Equation 3) which greatly improves the efficiency of the simulation (**Table 1**). Inspiration was taken from the NEURON simulation environment (Hines and Carnevale, 1997), in which LUTs are used as standard for Hodgkin-Huxley style ion channels. In NEURON, values of τ_n, τ_m, τ_h, n_∞, m_∞, and h_∞ are pre-calculated for values of V_{soma} at 1 mV intervals between values of -100 and $+100$ mV and the value of V_{soma} is used with interpolation to retrieve corresponding parameters from the table. Here, a similar table is tested in which instead of values of τ_n, τ_m, and τ_h, values of $\exp(\Delta t/\tau_n)$ (and similar for m and h) with $\Delta t = 0.1$ were pre-calculated for each value of V to further decrease the amount of computation required at each timestep. While use of this table did decrease the computational requirements, the change from NEURON's standard LUT meant that discrepancies were introduced. To rectify this, an identical LUT to NEURON's was created, along with a LUT which stores values of $\exp(\Delta t/\tau_n)$ with $\Delta t = 0.1$ for values of τ_n (and similar for m and h). Therefore, instead of the complex equations required to solve Equation (3) and the similar equations for m and h, each state update then requires only three look-up operations per activation parameter followed by one addition and multiplication.

2.4.2. Two-Compartment Model

The same LUT described in Section 2.4.1 is used for the somatic compartment in the two-compartment model. For the dendritic compartment, another LUT (using 1.6 kB of memory) is used in the calculation of the dCaAP current (I_{dCaAP}) again improving efficiency. Here, A and B in I_{dCaAP} are each described by two divisions and an exponential operation which are particularly costly on SpiNNaker hardware. However, as the two terms are not themselves voltage dependent, calculating each term at every timestep is unnecessary; A and B have characteristic sigmoidal shapes which describe the rise and decay of the dCaAP current which can be pre-calculated and loaded onto the SpiNNaker chips such that the $A - B$ calculation:-

$$\frac{1}{1+\exp(-\frac{(t-t')}{\tau_A})} - \frac{1}{1+\exp(-\frac{(t-(t'+\Delta t')}{\tau_A})} \tag{20}$$

is replaced by a single look-up operation.

3. MODEL VALIDATION

To assess the accuracy of the proposed models on neuromorphic hardware, the models are benchmarked against the NEURON simulation environment in a number of simulations. Benchmarking involves comparison between the membrane potential on each platform at each timestep and comparison of the timing of spikes. Monitoring progression of membrane potential enabled a comparison of the numerical solvers on the different platforms and spike times give a broader comparison as spikes are the fundamental communication method in SNNs. Spike times are recorded as the timestep in which membrane voltage crossed a threshold value, -20 mV, and are compared between the different platforms. In order for direct comparisons to be made, identical simulations are run with SpiNNaker, Jib2 and with NEURON. Despite the mathematical complexity involved in these calculations, SpiNNaker and the Jib2 neuromorphic hardware are still able to model the HH and two-compartment neuron accurately.

3.1. Ion-Channels and HH Neurons

In the somatic model, current injections ranging from 0 to 10 nA are tested for 2 s of simulation time on both SpiNNaker and Jib2. This is long enough for steady state behavior to develop in the neuron and accumulated errors to become visible if present. The membrane voltage is then compared with an identical reference model in the NEURON simulation environment. The maximum error recorded over all current injections over full simulation time for SpiNNaker is 34.6mV, and for Jib2 is 0.106 mV. Spike times are consistent between Jib2 and NEURON, but the increase in error on the SpiNNaker neuromorphic system leads to accumulated errors which results in differences in spike timings between NEURON and SpiNNaker. Despite this, over the range of simulations, spike times on SpiNNaker only differ by one timestep (0.1 ms). In these neuron models the action potential is the most challenging part of the model due to the rapidly changing dynamics, and it is during action potentials that the largest errors between the fixed-point SpiNNaker implementation and the reference model are generated. One source of errors between these systems is the differing number representation: NEURON supports double precision floating point numbers, Jib2 supports single precision floating point units and SpiNNaker supports 32-bit fixed-point representations. During testing, switching the reference model to a CPU implementation and restricting the precision to 32-bit floating point arithmetic resulted in negligible errors between this implementation and Jib2. This shows that the different number representations are a source of error, these results are not included due to brevity. Another source of error in both SpiNNaker and Jib2 relative to NEURON is due to subtle differences in look-up tables being implemented: where NEURON pre-calculates values of τ_n, τ_m, and τ_h and then calculates values of $\exp(\Delta t/\tau_n)$ (and similar for m and h), this exponential and division step is replaced with another LUT in the SpiNNaker implementation to avoid the need for $\exp(\Delta t/\tau_n)$ calculations at each timestep. This source of error is confirmed by altering double precision reference models to mirror the SpiNNaker implementation and observing the decreased error. The accuracy of the somatic compartment is also tested with varying resolutions of LUT to further justify the use of a LUT with 1 mV intervals between values of -100 and $+100$ mV, as in NEURON (Hines and Carnevale, 1997) simulations. The

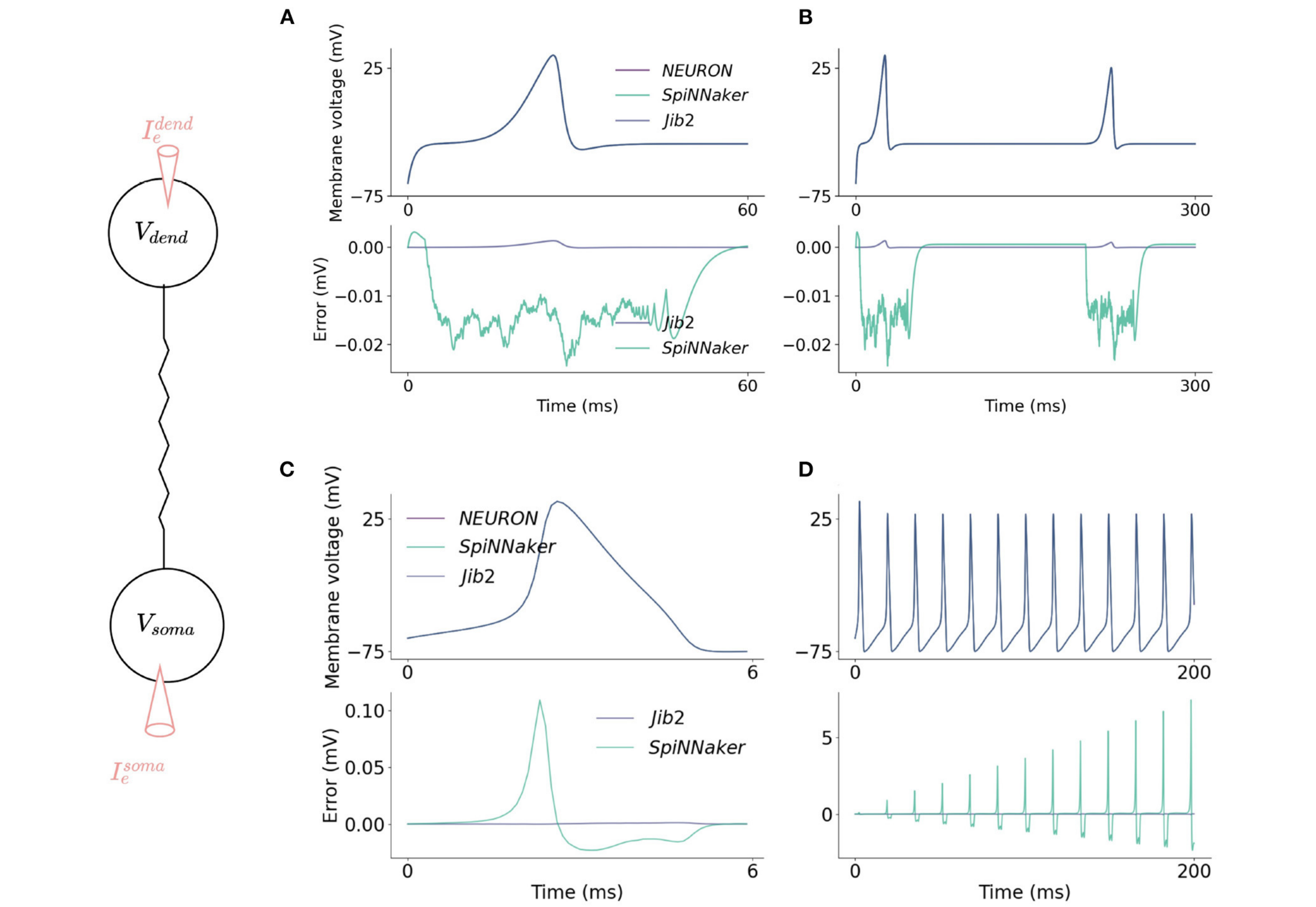


FIGURE 5 | Accuracy comparisons between the two-compartment model implemented on SpiNNaker, Jib2 and NEURON. The dendritic compartment fires a dCaAP in response to a current injection of 3.5nA which is accurately modeled on the neuromorphic platforms **(A)**. The dendritic compartment is modeled accurately over time **(B)**: absolute errors are small throughout the duration of the action potential and drop to 0 mV when the dendritic compartment enters its refractory period. In the somatic compartment, absolute errors are larger because the calculation of I_{soma} is voltage dependent, therefore when errors are produced in this calculation, an error in voltage is calculated which then, in turn, further increases the error in the I_{soma} calculations **(C)**. Because of this, over time, SpiNNaker experiences accumulated errors. With Jib2 errors return to 0mV between spikes, and with do not accumulate over time **(D)**.

model is simulated with no LUT, and with LUTs with 2 and 1 mV voltage intervals.With no LUT, the maximum error was 105.9 mV. Inclusion of LUTs increases the accuracy of models with the 2 mV table resulting in a 59.2 mV maximum error and the 1 mV interval table providing the most accurate solution with a 34.6 mV maximum error. SpiNNaker cores have 64 kB of memory for data storage (DTCM). Finer resolution LUTs are not tested because they occupy more of the SpiNNaker DTCM and the accuracy of spike times using the 1 mV interval is sufficient therefore occupying more DTCM with larger lookup tables is unnecessary. Despite the differing errors, both SpiNNaker and Jib2 can accurately model the HH model in response to a wide range of current injections and are able to maintain this accuracy over time (**Figure 5**).

3.2. Multi-Compartment Modeling

In the dendritic compartment, current injections ranging from 0 to 10 nA are tested for 2 s of simulation time. Membrane voltage was recorded and absolute errors between SpiNNaker, Jib2 and NEURON are calculated, as well as the timings of spikes in both compartments. In the dendritic compartment, the maximum error recorded over all current injections over full simulation time for SpiNNaker is 0.00314 mV, and for SpiNNaker 2 is 0.00137 mV. An example current injection of 0.45 nA into the dendritic compartment and resulting membrane potential recording is shown in **Figure 5A**. The evolution of absolute error over time in the dendritic compartment in response to injected current into the dendrite follows a typical shape each time a dCaAP is fired; the error remains below 0.5μV between spikes when the dendritic compartment is in its refractory period but rises as the membrane voltage rises, following a similar progression as the voltage. After the spike, the look-up table is no longer used and the value of $A - B$ returns to 0, meaning the value of I_{dCaAP} is 0. There are therefore no issues with errors accumulating over time because after each spike the error returns to near zero as there is no active calcium current. The errors for the dendritic compartment simulations are smaller than the somatic model due to a decreased complexity in the equations (the soma contains more ion-channels). Again, errors result here from the differing numerical datatypes used on the different systems. This is demonstrated by restricting the reference model to single precision floats which decreases the error between Jib2 and the reference model. Again, during testing, switching the reference model to a CPU implementation and restricting the precision to 32-bit floating point arithmetic resulted in decreased errors between this implementation and Jib2, these results are not included due to brevity. The errors are not large enough to cause any differences in the timing of spikes between SpiNNake, Jib2 and the NEURON model. Spike times remained consistent across all platforms.

3.3. Performance Profiling

We are interested in accelerating brain simulation with neuromorphic hardware, therefore the time taken to update the state of a neuron in each simulated 0.1 ms timestep is a key metric to evaluate for the different implementations. This is measured through recording the number of clock cycles taken to update the membrane potential in each model, and combining with the clock frequency. Each core on the SpiNNaker chip operates at 200 MHz, meaning one clock cycle takes 5 ns. With Jib2, voltage-frequency settings of 0.5 V–150 MHz and 0.8 V–300 MHz result in clock cycles taking 6.67 and 3.3 ns, respectively. For each model, 100 neurons are profiled for 10,000 timesteps with a representative current injection causing regular somatic spiking with a frequency of 50 Hz for HH and two compartment model. The amount of time taken to update the membrane potential in implemented models, with and without LUTs, is described in **Table 1**. For comparison, the time taken to update the membrane potential of a LIF neuron on SpiNNaker and Jib2 is also presented, the LIF neuron is kept sub-threshold during profiling in order to provide full state updates at each timestep, analogous to the HH and two compartment models.

In the HH model, the calculation of ion-channel parameters, resulting current values from these parameters, and subsequent membrane potential update for each 0.1 ms timestep on SpiNNaker takes 99.6 μs without any lookup tables (LUTs). Inclusion of the LUTs results in the update taking 8.34 μs to compute, meaning the look-up table speeds the implementation up by over 11x. Similar calculations on Jib2 demonstrate the benefits of the hardware accelerators by showing a speed up in processing time. At 300 MHz, the membrane potential update takes 0.73 μs and at 150 MHz, the update takes 1.45 μs. Again, Jib2 illustrates that the LUTs improve the implementation speed, both the HH model and the two-compartment model are over 3x faster with the LUTs (**Table 1**). The HH model on Jib2 with LUTs is within an order of magnitude of the LIF neuron running on Jib2 which takes 0.09 μs.

Addition of the calcium current in the dendritic compartment for the two-compartment model increases the amount of time taken to update the membrane potential. On SpiNNaker, updating the somatic membrane potential takes 12.91 μs with LUTs, without LUTs this takes 153.67 μs, over 11x longer. On Jib2 these membrane potential updates are quicker, taking 1.09 and 2.18 μs with the core operating at 300 and 150 MHz, respectively. Again, this model is 3x faster with the inclusion of the LUTs, with the non-LUT implementations taking 5.19 and 6.45 μs (**Table 1**).

4. RESULTS

4.1. HH Model Increases Expressiveness of Single Compartment Neurons

After models were validated (Section 3), the additional behaviors they bring to neuromorphic hardware were explored which have not been demonstrated previously on these platforms. The sodium and potassium ion-channels incorporated into a HH neuron model give the neuron a number of firing capabilities that are unable to be produced with simple LIF neurons. Izhikevich (2004) identified 20 of the most important neurocomputational spiking features of biological neurons which can be captured with spiking neuron models. The Hodgkin-Huxley model was identified to be able to reproduce all 20 of the firing dynamics, while the LIF neuron model can only reproduce 3: the ability to spike tonically, to increase firing frequency in response to increased input strength and the ability to integrate inputs and fire in response to them (Izhikevich, 2004). The single compartment neuron model here features Hodgkin-Huxley sodium and potassium ion-channels which therefore give this model the ability to produce all 20 neurocomputational features.

Firing features of the somatic model are demonstrated in **Figure 6**, through injection of current directly into the neuron and recording the resulting somatic membrane voltage. It is not possible for a single neuron model to exhibit all properties simultaneously because some features, for example the ability to fire a train of spikes in response to a constant input, and the ability to fire periodic bursts of spikes in response to constant input, are mutually exclusive. For that reason, 10 biologically important firing features are presented that can be exhibited simultaneously without altering parameters from those described in Section 2.2. In response to a constant somatic current injection, the soma can fire a constant train of spikes known as tonic spiking (**Figure 6A**). If the current injection is just strong enough to cause a spike, the neuron demonstrates phasic spiking (**Figure 6B**) where a single spike is fired followed by inactivity. Phasic spikes are often followed by sub-threshold oscillations (**Figure 6C**) caused by ion-channel currents. Inputs to neurons are generally not constant, and neurons can display a number of firing properties in response to different input currents. Neurons can demonstrate accommodation to inputs: presenting the neuron with a slowly increasing current does not produce a spike but presenting the same neuron with a sharply increasing current will produce a spike (**Figure 6D**) due to the ion-channels within the neuron having more time to adapt to the current, meaning the neuron accommodates. Hodgkin-Huxley neurons are Class II excitable neurons meaning they are either inactive or they fire spikes with a high frequency, this is displayed by presenting the neuron with a steady increase in injected current (**Figure 6E**). Adaptation of ion-channel currents also leads to a phenomenon in which the neuron fires a spike after an inhibitory

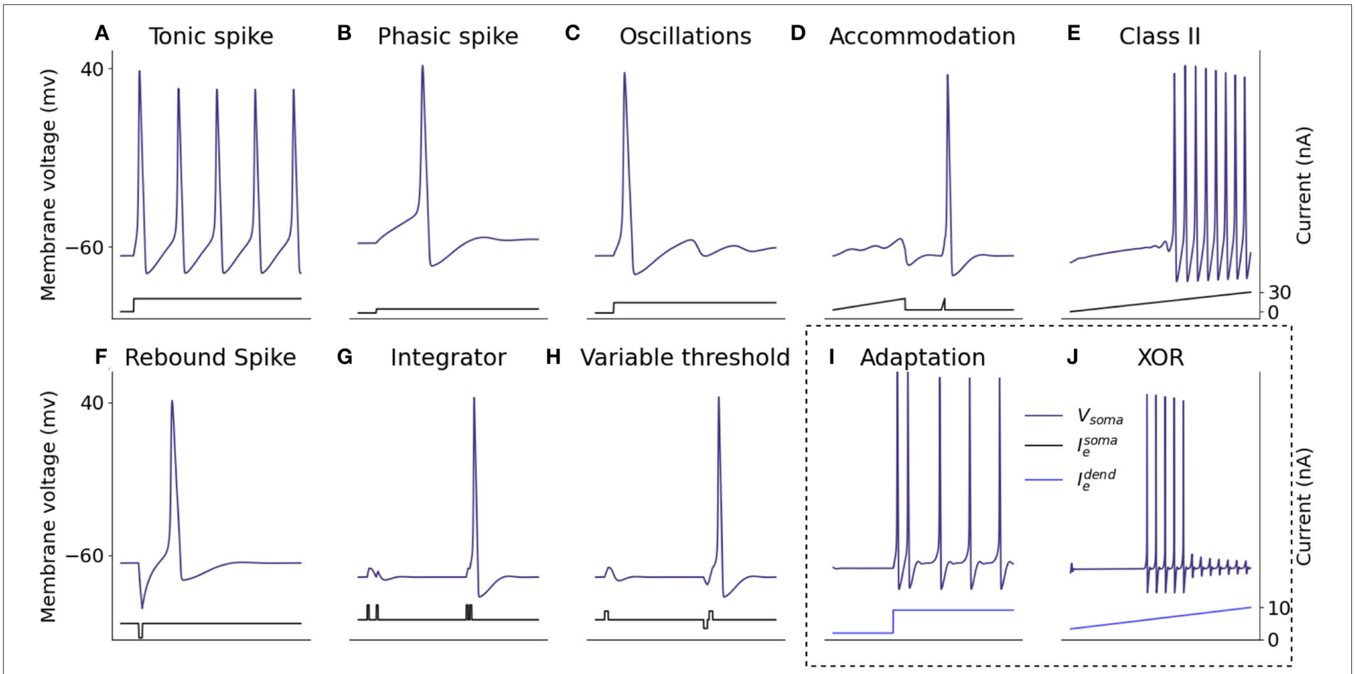


FIGURE 6 | Spiking properties of the somatic compartment. Shown are simulations of the two-compartment model with current injected into the somatic or dendritic compartment to reproduce different firing dynamics capturing a number of biologically representative neural capabilities. The firing dynamics in the box are achieved through injected current into the dendritic compartment rather than the somatic compartment itself. **(A)** Tonic spike. **(B)** Phasic spike. **(C)** Oscillations. **(D)** Accommodation. **(E)** Class II. **(F)** Rebound spike. **(G)** Integrator. **(H)** Variable threshold. **(I)** Adaptation. **(J)** XOR.

current injection (**Figure 6F**) known as a post-inhibitory spike. Neuron models in SNNs generally integrate spiking inputs over time, if inputs are closer together then the neuron is more likely to fire spikes as the firing threshold is passed (**Figure 6G**). While most SNN neuron models have fixed voltage threshold for firing spikes, biological neurons actually have a variable threshold which is determined by the activity of the neuron (**Figure 6H**). Briefly exciting the neuron in **Figure 6H** is not enough to make the neuron fire, however if it is preceded by a brief inhibitory input, this same excitation will cause the neuron to fire. Spike rate adaptation is a phenomenon in which neurons fire tonic spikes with decreasing frequency, this feature is mutually exclusive with the tonic spiking ability discussed above and single-compartment neuron models are unable to display both properties. Here, inclusion of the dendritic compartment allows the soma to display spike-rate adaptation in response to a constant current injected into the dendritic compartment (**Figure 6I**). Injecting the dendritic compartment with a steady increase in injected current leads to remarkably different somatic firing dynamics in which firing starts when input is above threshold but ceases firing when the input continues to rise (**Figure 6J**), this phenomenon is explained in Section 4.2.

Other neurocomputational properties presented by Izhikevich can be captured by altering the parameters described in Section 2.2. These include the ability to burst (rather than tonically) fire and to fire in response to inhibitory (rather than excitatory) inputs. Properties such as these can be captured by altering parameters involved in the differential equations describing ion-channel currents (Kirigeeganage et al., 2019). Sodium channel currents change more rapidly than potassium currents in the beginning of the progression of an action potential, they are described by an activation variable (m) and an inactivation variable (n) (Section 2.1). Therefore, to adjust the neuron to be responsive to inhibitory inputs, modifications to the differential equation describing m can be made to alter the responsiveness of the neuron.

Izhikevich compared 11 spiking neuron models by the ability of the models to produce some of these features and the computational cost of each model (Izhikevich, 2004). The Hodgkin-Huxley model was the only one able to produce all firing properties while also being biophysically meaningful. This biological accuracy leads to higher computational cost of the model which makes it more expensive to implement than other neuron models. However, computational costs can be diminished using a variety of techniques (see Section 2.4). In addition, the biophysical plausibility of the Hodgkin-Huxley model allows incorporation of dendritic morphology and different ion-channels through cable equation modeling, this is not possible with less biologically plausible models. The dendritic modeling in the second compartment gives the neuron additional computational properties to further increase the firing capabilities beyond those identified by Izhikevich, described in Section 4.2.

TABLE 2 | XOR truth table demonstrating the binary output of an XOR operation in response to two binary inputs.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

If the inputs are both 1 (true) or both 0 (false) then the output is 0 (false), if either input is 1 and the other 0 then the output is 1.

4.2. Dendritic Compartment Enables Single Neuron to Function as a Multi-Layer Network

Inclusion of the dendritic compartment further increases the computational properties of the neuron beyond the 20 identified by Izhikevich. The dynamics described by Izhikevich are relatively well-known capabilities for Hodgkin-Huxley neurons and can be reproduced by other neuron models including the model proposed by Izhikevich himself (Izhikevich, 2004). However, the biological plausibility of the Hodgkin-Huxley model enables it to be built upon through the incorporation of more compartments representing dendritic branches which further increase the capabilities of the neuron.

Here, the dendritic compartment gives the neuron the ability to compute a logical operation known as exclusive-or (XOR). Logical operations are performed on binary inputs and produce a binary output. An XOR operation is a logical operation in which an exclusive-or is implemented: the binary output is 1 (or true) when there is only one input to the operation, if both of the inputs are 0 (or false) or both of the inputs are 1 then the output of the XOR operation is 0 (**Table 2**). While simple logic operations such as AND and OR are easily implemented in single units within neural networks, the XOR function is a common problem in neural network research and is widely used as an example of a linearly inseparable problem; it has become a benchmark in machine learning for testing neural network capabilities in solving complex problems. SNN implementation of the XOR operation has thus far required multiple layers of spiking neurons as the nature of spiking neural network architectures is that each layer can only separate data points with a single line (Vogels and Abbott, 2005; Reljan-Delaney and Wall, 2018; Cyr et al., 2020). XOR functions were deemed impossible in single-layer networks—Marvin Minsky and Seymour Papert provided proof that single-layer ANNs could not perform XOR in their 1969 book Perceptrons (Minsky and Papert, 2017) due to the non-linear separability. An XOR gate was demonstrated within a large SNN by (Vogels and Abbott, 2005) who stated that "a functional XOR gate requires ∼220 neurons". Here, the XOR problem is solved with a single neuron model.

The shape of the dendritic activation function allows the XOR problem to be solved here with a single neuron model. The activation function results in the amplitude of dCaAPs decreasing when the input to the dendritic compartment increases above a certain strength; the dCaAP amplitude is maximal when the input to the dendrite crosses the threshold for activation, then decreases as the input increases further (**Figure 7A**). As the dCaAP amplitude decreases, the amount of current flowing from the dendritic compartment to the somatic compartment decreases which in turn decreases the somatic action potential amplitude (**Figure 4B**). As somatic action potentials are an all-or-nothing spike response, when the current flowing from the dendritic compartment to the somatic compartment decreases below a certain value, the soma stops firing action potentials (**Figure 4B**). Therefore, the somatic compartment will start firing when the input to the dendrite is increased to its firing threshold and then will decrease and eventually stop firing as input is increased further. Similar behavior is observed when input to the dendritic compartment is synaptic rather than injected current. Increasing the number of synapses also causes the dCaAP amplitude to increase then to decrease above a certain number of synapses, leading to somatic action potential firing and subsequent cease (**Figure 7**).

While the action potentials arising from calcium currents in the dendritic compartment are responsible for XOR-type computation, the somatic compartment, through integration of sodium and potassium currents, computes standard logical operations for spiking neurons such as AND and OR. The combination of these differing logical operations allow the neuron to act as a multi-layer network, increasing the computational capabilities of a single neuron model in comparison with a leaky integrate-and-fire model.

5. DISCUSSION

This work has provided the first fixed-point implementation of ion-channel, Hodgkin-Huxley, and multi-compartment models on SpiNNaker neuromorphic hardware and the first profiling for both speed and accuracy of such models on SpiNNaker2 prototype neuromorphic hardware, demonstrating the improved performance of the next-generation system through the use of hardware accelerators and floating point arithmetic. The first demonstration of a two-compartment neuron model running on neuromorphic hardware that can solve the XOR problem using a single neuron is also presented through this work.

Neuromorphic systems are designed to provide low-energy platforms for simulation of Spiking Neural Networks (SNNs) but in doing so biologically plausible neuron models have largely been ignored in favor of simple and efficient neuron models such as the Leaky Integrate-and-Fire (LIF) model. In contrast, focus in the computational neuroscience community has been on building models with a high degree of biological accuracy which are in turn accompanied by large computational costs, making the models difficult to scale into SNNs. This work bridges this gap by presenting two biologically inspired neuron models (**Figure 1**), implemented efficiently and accurately on SpiNNaker and Jib2 neuromorphic platforms (**Figure 5**): a single compartment Hodgkin-Huxley (HH) neuron (**Figure 2**) and a multi-compartment neuron incorporating dendritic computation (**Figure 4**).

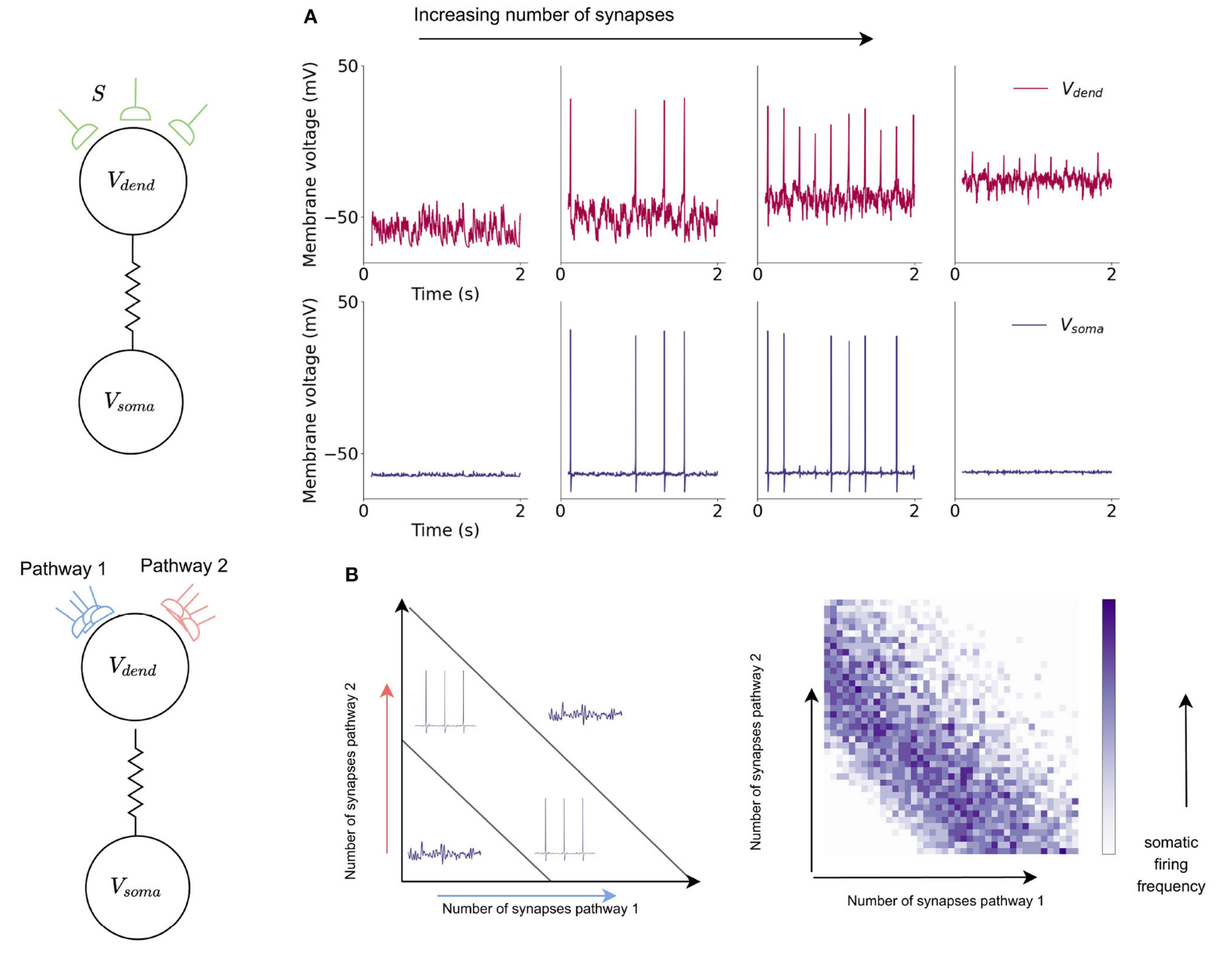


FIGURE 7 | Somatic compartment exhibits XOR response to dendritic input in a single neuron model. **(A)** Dendritic and somatic firing dynamics in response to increased synaptic input into the dendritic compartment. Increased number of synapses leads to an initial increase and then decrease in dCaAP amplitude which subsequently cause the somatic compartment to start firing action potentials then stop. **(B)** The dynamics of the somatic compartment in response to the dendritic inputs provide a solution to the XOR problem in a single neuron model. Somatic compartment exhibits XOR response to dendritic input in a single neuron model. Increased number of synapses leads to an initial increase and then decrease in dCaAP amplitude which subsequently cause a similar increase and cease of somatic action potential firing.

Both SpiNNaker and Jib2 are able to accurately model both neurons over time with identical spike times recorded on Jib2 and a reference model in NEURON (Hines and Carnevale, 1997) and spike times within 0.1 ms on SpiNNaker. Manipulation of equations, pre-calculation of constants and the use of lookup tables (using 12 and 13.6 kB of memory for HH and two-compartment models, respectively) enabled a significant speed up of simulation time of the models (approx 11× for both the single and two-compartment models—**Table 1**). This speed-up is further increased by 3× with implementation on the next-generation Jib2 neuromorphic chip, demonstrating the effectiveness of hardware accelerators for expressions such as exponential operations (**Table 1**) when simulating biologically representative neurons.

Comparison with neuromorphic implementations of the conventional LIF neuron model revealed that both the HH and the multi-compartment neurons were slower to simulate on neuromorphic hardware, due to the increased complexity of the models (**Table 1**). However, the computational capabilities gained justify the increased expense of running the model, and the model on Jib2 is within an order of magnitude of the LIF neuron in terms of computation time. The underlying ion channel models directly correspond to biophysiological data, bringing increased biological relevance to models simulated on neuromorphic hardware. Furthermore, the presented HH model exhibits a wide range of firing characteristics which cannot be captured with LIF neurons (**Figure 6**), and the inclusion of a dendritic compartment enables the a single neuron model to function as a multi-layer network. The multi-compartment model provides the first implementation of a single neuron model capable of solving the XOR problem on neuromorphic hardware (**Figure 7**).

This work has explored simulation and profiling of individual neurons, and their realization on neuromorphic systems. The ultimate goal of implementing these models is harnessing their ability to capture biologically representative features in *large-scale* SNNs, and opening up new applications in bio-inspired AI. To understand how the presented models would scale when included in large networks, it is useful to contrast performance with LIF neurons and biologically representative neural circuits previously evaluated on SpiNNaker. In previous work modeling cortical microcircuits comprising LIF neurons, it was shown that neuron and synapse processing could be parallelized effectively on multicore architectures such as SpiNNaker (Rhodes et al., 2020). Through this parallelization real-time simulation of cortical circuits containing 80k neurons and 300M synapses was demonstrated, with an energy per synaptic event of $\approx 0.6\ \mu$J. The models presented here would impact the *neuron* processing, resulting in a $\approx 40\times$ reduction in neuron density relative to LIF neurons to accommodate the increased model complexity. This indicates that approximately 40× more SpiNNaker chips would be required to simulate the same size of model, leading to the same factor increase in total energy consumption. Projecting these numbers on to SpiNNaker2 requires consideration of the updated performance achieved with the new hardware. The HH and two-compartment neurons occupy 48 and 55 kB, respectively (of the 128 kB fast-access SRAM for combined instruction and data storage on Jib2) with the instructions to update the neuron and the storage of constants, variables and LUTs. Increasing the number of neurons does not significantly increase the storage requirements, as the instructions for updating the neurons are the same and all neurons share common LUTs. While the number of neurons per core determines the amount of state variables to be stored, these datastructures are relatively small compared to those described above (assuming split neuron and synapse processing/storage as described above, Peres and Rhodes, 2022). Therefore the determining factor in the number of neurons which can be simulated on each core is the processing time. As it takes 0.73 μs to update a HH neuron and 1.09 μs for the two-compartment neuron using a 0.1 ms simulation timestep, assuming the goal of real-time simulation, an upper limit of 136 HH neurons or 91 two-compartment neurons could be updated by a single core while maintaining real-time execution. In reality this number is likely to be reduced to enable cores to perform auxilliary operations such as monitoring and data recording, reducing overall neuron density. However, this is likely to remain above the 64 neurons per core utilized in previous cortical simulations on SpiNNaker (Rhodes et al., 2020), enabling real-time cortical simulations containing biologically representative ion-channel-based neuron models (on SpiNNaker2). Furthermore, embedding these models within the SpiNNaker routing and communications fabric should facilitate further expansion of model sizes while maintaining real-time execution. This indicates that the cost of changing from LIF to multicompartment models on SpiNNaker2 will incur a 10× increase in energy, with the overall system significantly more energy efficient—LIF neurons have been profiled at 20 pJ per synaptic event (Höppner et al., 2021).

The model provides a framework for capturing and testing more biologically plausible neural dynamics in an efficient way. For example, different ion-channels can easily be substituted or added to the model, and more complex morphologies can be captured through inclusion of more dendritic compartments. Recent work has demonstrated the potential of multi-compartment neuron models to learn *via* a synaptic learning rule (Bicknell and Häusser, 2021), opening the door to the possibility of training the neuron models presented in this work within large-scale SNNs on neuromorphic hardware, in particular those featuring hardware accelerators to maximize efficiency. Significant computational capabilities are gained with each individual neuron model and neuromorphic architectures can provide energy-efficient platforms for simulations. While this work has focused on demonstrating feasibility through development of software models suitable for execution on SpiNNaker, the developed models also provide the first step toward algorithm-hardware co-design. Hardware requirements such as arithmetic operations and memory use have been identified, providing insights into how future neuromorphic systems could be tailored to further optimize execution.

DATA AVAILABILITY STATEMENT

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

AUTHOR CONTRIBUTIONS

MW designed and completed the project through model creation, implementation, validation, timing measurements, accuracy measurements and testing of models on both neuromorphic systems and NEURON, and wrote the manuscript. OR provided input into model design and implementation, assisted with efficiency improvements and measurements of models on both neuromorphic systems. Both authors reviewed and edited manuscript, and approved the submitted version.

FUNDING

This work was supported by the EU ICT Flagship Human Brain Project (H2020 785907 and 945539).

ACKNOWLEDGMENTS

The authors thank Dr. Luis Plana for technical discussions and assistance with the Jib2 implementation and measurements.

REFERENCES

Abu-Hassan, K., Taylor, J. D., Morris, P. G., Donati, E., Bortolotto, Z. A., Indiveri, G., et al. (2019). Optimal solid state neurons. *Nat. Commun*. 10, 1–13. doi: 10.1038/s41467-019-13177-3

Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J., et al. (2014). Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* 102, 699–716. doi: 10.1109/JPROC.2014.2313565

Bicknell, B. A., and Häusser, M. (2021). A synaptic learning rule for exploiting nonlinear dendritic computation. *Neuron* 109, 4001.e10–4017.e10. doi: 10.1016/j.neuron.2021.09.044

Cox, D. D., and Dean, T. (2014). Neural networks and neuroscience-inspired computer vision. *Curr. Biol.* 24, R921–R929. doi: 10.1016/j.cub.2014.08.026

Cyr, A., Thériault, F., and Chartier, S. (2020). Revisiting the XOR problem: a neurorobotic implementation. *Neural Comput. Appl*. 32, 9965–9973. doi: 10.1007/s00521-019-04522-0

Davies, M., Srinivasa, N., Lin, T., Chinya, G., Cao, Y., Choday, S., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Dayan, P., and Abbott, L. F. (2005). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Cambridge, MA: The MIT Press.

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The SpiNNaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Gidon, A., Zolnik, T. A., Fidzinski, P., Bolduan, F., Papoutsi, A., Poirazi, P., et al. (2020). Dendritic action potentials and computation in human layer 2/3 cortical neurons. *Science* 367, 83–87. doi: 10.1126/science.aax6239

Hines, M. L., and Carnevale, N. T. (1997). The neuron simulation environment. *Neural Comput*. 9, 1179–1209. doi: 10.1162/neco.1997.9.6.1179

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* 117, 500–544. doi: 10.1113/jphysiol.1952.sp004764

Hopkins, M., and Furber, S. (2015). Accuracy and efficiency in fixed-point neural ODE solvers. *Neural Comput*. 28, 2148–2182. doi: 10.1162/NECO_a_00772

Hopkins, M., Mikaitis, M., Lester, D. R., and Furber, S. (2020). Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci*. 378, 2166. doi: 10.1098/rsta.2019.0052

Höppner, S., Yan, Y., Dixius, A., Scholze, S., Partzsch, J., Stolba, M., et al. (2021). The SpiNNaker 2 processing element architecture for hybrid digital neuromorphic computing. *arXiv preprint arXiv:2103.08392*. doi: 10.48550/ARXIV.2103.08392

Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE Trans. Neural Netw*. 15, 1063–1070. doi: 10.1109/TNN.2004.832719

Kaiser, J., Billaudelle, S., Müller, E., Tetzlaff, C., Schemmel, J., and Schmitt, S. (2022). Emulating dendritic computing paradigms on analog neuromorphic hardware. *Neuroscience* 489, 290–300. doi: 10.1016/j.neuroscience.2021.08.013

Kirigeeganage, S., Jackson, D., Zurada, J. M., and Naber, J. (2019). "Modeling the bursting behavior of the Hodgkin-Huxley neurons using genetic algorithm based parameter search," in *2018 IEEE International Symposium on Signal Processing and Information Technology, ISSPIT 2018* (Louisville, KY), 470–475. doi: 10.1109/ISSPIT.2018.8642781

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., et al. (2019). Coreneuron: an optimized compute engine for the neuron simulator. *Front. Neuroinform*. 13, 63. doi: 10.3389/fninf.2019.00063

Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

Mayr, C., Höppner, S., and Furber, S. (2019). SpiNNaker 2: a 10 million core processor system for brain simulation and machine learning. *Concurr. Syst. Eng. Ser*. 70, 277–280. doi: 10.48550/arXiv.1911.02385

Merolla, P. A., Arthur, J. V., Alvarez-icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., et al. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 668–673. doi: 10.1126/science.1254642

Mikaitis, M. (2020). *Arithmetic accelerators for a digital neuromorphic processor* (Ph.D. thesis). Manchester, UK.

Minsky, M., and Papert, S. A. (2017). *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: The MIT Press. doi: 10.7551/mitpress/11301.001.0001

Müller, E., Arnold, E., Breitwieser, O., Czierlinski, M., Emmel, A., Kaiser, J., et al. (2022). A scalable approach to modeling on accelerated neuromorphic hardware. *Front. Neurosci*. 16, 884128. doi: 10.3389/fnins.2022.884128

Pei, J., Deng, L., Song, S., Zhao, M., Zhang, Y., Wu, S., et al. (2019). Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* 572, 106–111. doi: 10.1038/s41586-019-1424-8

Peres, L., and Rhodes, O. (2022). Parallelization of neural processing on neuromorphic hardware. *Front. Neurosci*. 16, 867027. doi: 10.3389/fnins.2022.867027

Poirazi, P., and Papoutsi, A. (2020). Illuminating dendritic function with computational models. *Nat. Rev. Neurosci*. 21, 303–321. doi: 10.1038/s41583-020-0301-7

Reljan-Delaney, M., and Wall, J. (2018). "Solving the linearly inseparable XOR problem with spiking neural networks," in *Proceedings of Computing Conference 2017* (London, UK), 701–705. doi: 10.1109/SAI.2017.8252173

Rhodes, O., Bogdan, P. A., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., et al. (2018). SpyNNaker: a software package for running pynn simulations on spinnaker. *Front. Neurosci*. 12, 816. doi: 10.3389/fnins.2018.00816

Rhodes, O., Peres, L., Rowley, A. G., Gait, A., Plana, L. A., Brenninkmeijer, C., et al. (2020). Real-time cortical simulation on neuromorphic hardware. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci*. 378, 2164. doi: 10.1098/rsta.2019.0160

Schemmel, J., Bruderle, D., Grubl, A., Hock, M., Meier, K., and Millner, S. (2010). "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *2010 IEEE International Symposium on Circuits and Systems (ISCAS)* (Paris), 1947–1950. doi: 10.1109/ISCAS.2010.5536970

Schemmel, J., Kriener, L., Muller, P., and Meier, K. (2017). "An accelerated analog neuromorphic hardware system emulating NMDA- and calcium-based non-linear dendrites," in *Proceedings of the International Joint Conference on Neural Networks* (Alaska, USA), 2217–2226. doi: 10.1109/IJCNN.2017.7966124

Vogels, T. P., and Abbott, L. F. (2005). Signal propagation and logic gating in networks of integrate-and-fire neurons. *J. Neurosci*. 25, 10786–10795. doi: 10.1523/JNEUROSCI.3508-05.2005

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Neuroinformatics

TYPE Technology and Code
PUBLISHED 22 July 2022
DOI 10.3389/fninf.2022.883796

OPEN ACCESS

EDITED BY
Thomas Nowotny,
University of Sussex, United Kingdom

REVIEWED BY
Gennady Cymbalyuk,
Georgia State University, United States
Claudia Casellato,
University of Pavia, Italy

*CORRESPONDENCE
Marc de Kamps
M.deKamps@leeds.ac.uk

RECEIVED 25 February 2022
ACCEPTED 24 June 2022
PUBLISHED 22 July 2022

CITATION
Osborne H and de Kamps M (2022) A numerical population density technique for N-dimensional neuron models.
Front. Neuroinform. 16:883796.
doi: 10.3389/fninf.2022.883796

A numerical population density technique for N-dimensional neuron models

Hugh Osborne[1] and Marc de Kamps[1,2,3]*

[1]School of Computing, University of Leeds, Leeds, United Kingdom, [2]Leeds Institute for Data Analytics, University of Leeds, Leeds, United Kingdom, [3]The Alan Turing Institute, London, United Kingdom

Population density techniques can be used to simulate the behavior of a population of neurons which adhere to a common underlying neuron model. They have previously been used for analyzing models of orientation tuning and decision making tasks. They produce a fully deterministic solution to neural simulations which often involve a non-deterministic or noise component. Until now, numerical population density techniques have been limited to only one- and two-dimensional models. For the first time, we demonstrate a method to take an N-dimensional underlying neuron model and simulate the behavior of a population. The technique enables so-called graceful degradation of the dynamics allowing a balance between accuracy and simulation speed while maintaining important behavioral features such as rate curves and bifurcations. It is an extension of the numerical population density technique implemented in the MIIND software framework that simulates networks of populations of neurons. Here, we describe the extension to N dimensions and simulate populations of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances then demonstrate the effect of degrading the accuracy on the solution. We also simulate two separate populations in an E-I configuration to demonstrate the technique's ability to capture complex behaviors of interacting populations. Finally, we simulate a population of four-dimensional Hodgkin-Huxley neurons under the influence of noise. Though the MIIND software has been used only for neural modeling up to this point, the technique can be used to simulate the behavior of a population of agents adhering to any system of ordinary differential equations under the influence of shot noise. MIIND has been modified to render a visualization of any three of an N-dimensional state space of a population which encourages fast model prototyping and debugging and could prove a useful educational tool for understanding dynamical systems.

KEYWORDS

simulator, neural population, population density, software, Python, dynamical systems, network, visualization

1. Introduction

A common and intuitive method for simulating the behavior of a population of neurons is to directly simulate each individual neuron and aggregate the results (Gewaltig and Diesmann, 2007; Yavuz et al., 2016; Knight et al., 2021). At this level of granularity, the population can be heterogeneous in terms of the neuron model used, parameter values, and connections. The state of each neuron, which may consist of one or many more time or spatially dependent variables, is then integrated forward in time. The benefit of this method of simulation is that it provides a great deal of control over the simulated neurons with the fewest approximations. If required, the state history of each neuron can be inspected. However, this degree of detail can produce results that are overly verbose making it difficult to explain observations. While this can be mitigated by carefully limiting the degrees of freedom (for example, keeping all neurons in the population homogeneous, using point neuron models, or having a well-defined connection heuristic), other simulation methods exist that have such assumptions built in and provide additional benefits like increased computation speed, lower memory requirements, or improved ways to present and interpret the data. For example, so-called neural mass models (Wilson and Cowan, 1972; Jansen and Rit, 1995) eschew the behavior of the individual neurons in a population in favor of a direct definition of the average behavior. These methods are computationally cheap and can be based on empirical measurements but they lack a direct link to the microscopic behavior of the constituent neurons which limits a generalization to populations of different neuron types.

Population density techniques (PDTs) approximate population-level behaviors based on a model definition of the constituent neurons. Most PDTs assume all neurons are homogeneous and unconnected within a discrete population. All neurons are considered point-neurons and adhere to a single neuron model which is made up of one or more variables that describe the state of the neuron at a given time. The state space of the model, as shown in Figure 1, contains all possible states that a neuron in the population could take. For a population of neurons, PDTs frequently define a probability density function or the related probability mass function across the state space which gives the probability of finding a neuron from the population with a given state. PDTs are not concerned with the individual neurons but instead calculate the change to the probability mass function which is governed by two processes: the deterministic dynamics defined by the underlying neuron model, and a non-deterministic noise process representing random incoming spike events.

Methods for solving the deterministic dynamics of a system of ordinary differential equations under the influence of a non-deterministic noise process have been used right back to early studies of Brownian motion. Then in theoretical neuroscience, Johannesma (1969) and Knight (1972) among others used similar techniques to give a formal definition of the effect of stochastic spiking events on a neuron by defining a probability density function of possible somatic membrane potentials. Most often, these involved the assumption of infinitesimal changes in state due to the incoming events, also known as the diffusion approximation. Omurtag et al. (2000) applied the method to a population of unconnected homogeneous neurons. They separated the deterministic dynamics of a common underlying neuron model from the incoming spike train generated by a Poisson process. Originally, the motivation for their work was to more efficiently approximate the behavior of collections of neurons in the visual cortex. Work by Sirovich et al. (1996) showed that there is a lot of redundancy in optical processing in the macaque visual cortex such that on the order of $O(10^4)$ functional visual characteristics or modalities are encoded by $O(10^8)$ neurons. It was, therefore, a reasonable approximation to treat a population of 10^4 neurons as a homogeneous group and investigate the interaction between populations. The technique was employed by Nykamp and Tranchina (2000) to analyse mechanisms for orientation tuning. Bogacz et al. (2006) also used PDTs to model decision making in a forced choice task.

PDTs have since been extended to attend to various shortcomings of the original formulation. For example, there is often an assumption of Poisson distributed input to a population (Omurtag et al., 2000; Mattia and Del Giudice, 2002; Rangan and Cai, 2007) which in certain circumstances is not biologically realistic. Ly and Tranchina (2009) outlined a technique to calculate the distribution of the output spike train of a population of LIF neurons with different input distributions (based on a renewal process - with a function involving the inter-spike interval). Instead of introducing a Poisson process for their noise term, they use a hazard function which defines the probability of an incoming spike given the time since the last spike. This allows them to handle more realistic input distributions such as a gamma distribution for certain situations and calculate the output firing rate. They are also able to derive the output statistics of a population like expected inter-spike interval and spike distribution. Further work has been done to develop so-called quasi-renewal processes (Naud and Gerstner, 2012) which define the probability of the next spike in terms of both the population level activity and the time since the last spike. Such approaches can simulate behaviors such as spike frequency adaptation and refractoriness but there is a weaker link to the underlying neuron model which limits the simulation of populations of neurons with dynamics that produce behaviors like bursting.

PDTs have also often been limited to low-dimensional neuron models with which to derive population level behavior and statistics. The conductance based refractory density (CBRD) approach (Chizhov and Graham, 2007) tracks the distribution of a population of neurons according to the time since they last spiked (often referred to as their age) instead of across the state space of the neuron model. In its most elementary form, the probability density equation, given in terms of time and time since last spike, is dependent on the neuronal dynamics defined by the underlying model and a noise process. Crucially though, the conductance variables defined in the underlying model (such as the sodium gating variables of the Hodgkin-Huxley neuron model) can be approximated to their mean across all neurons with similar age. With this approximation, the dimensionality

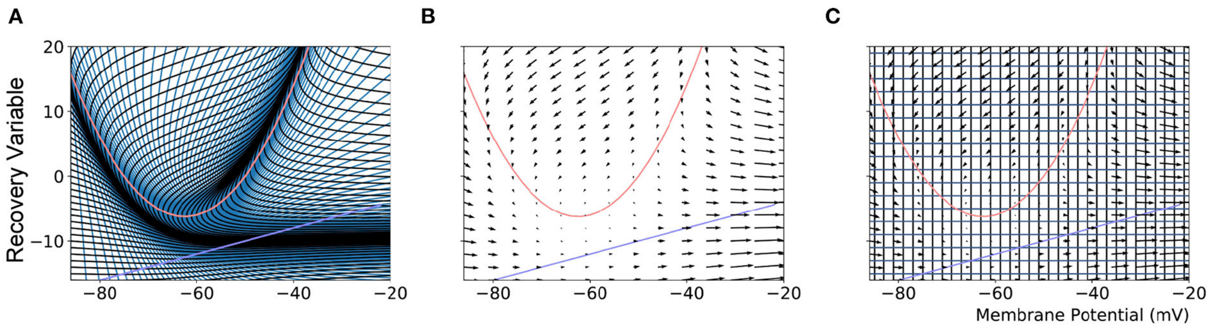


FIGURE 1
(A) The mesh used in MIIND to simulate a population of Izhikevich neurons. The quadratic red curve and blue line are the nullclines where the rate of change of the membrane potential and recovery variable, respectively, are zero. The strips, made up of quadrilateral cells are formed by the characteristic curves of the Izhikevich model for a given parameter set. **(B)** A vector field for the same model showing the direction of movement of probability mass around the state space. **(C)** The state space discretized into a regular grid. The parameters and definition of the Izhikevich model are not given here as it is only required to demonstrate the mesh and grid discretization. As in the original derivation of the model, the recovery variable has no units.

of the problem is reduced to a dependence only on the membrane potential, significantly improving the tractability of such systems. Refractory density approaches (Schwalger and Chizhov, 2019) have been extended further to approximate finite size populations, phenomenological definitions, and bursting behaviors (Schwalger et al., 2017; Chizhov et al., 2019; Schmutz et al., 2020).

Using these techniques for modeling and simulation generally requires a large amount of mathematical and theoretical work to develop a solution for a specific scenario. As we see above, each additional behavior requires at least an extension or even reformulation of a previous approach. The numerical PDT implemented in MIIND (de Kamps et al., 2019; Osborne et al., 2021) requires only a definition of the underlying neuron model plus population and simulation parameters. The definition can be given in the form of a Python function in a similar fashion to direct simulation techniques. However, until now, the PDT has been able to simulate populations of neurons adhering to only a one- or two-dimensional model. Often, this is enough as many different neuronal behaviors can be captured with two variables, for example, the action potential of the Fitzhugh-Nagumo neuron (FitzHugh, 1961; Nagumo et al., 1962), the spike frequency adaptation of the adaptive exponential integrate-and-fire neuron (Brette and Gerstner, 2005), or the bursting behavior of the Izhikevich neuron model (Izhikevich, 2007). Using a one-dimensional neuron model, MIIND has been employed to simulate a network of interacting populations in the spinal cord (York et al., 2022). Populations were based on the exponential integrate-and-fire neuron model and showed how a relatively simple spinal network could explain observed trends in a static leg experiment. The main benefit of using the numerical PDT in this study was to eliminate finite-size variation in the results which would have hindered the subsequent analysis. The MIIND software itself also afforded benefits such as the ability to quickly prototype population network models, and to observe the population states during and after simulation. Osborne et al. (2021) have previously presented the full implementation details of MIIND including the two "flavors" of the numerical PDT, named the mesh and grid methods. The mesh method involves discretizing the state space using a mesh of quadrilateral cells as shown in Figure 1A. The grid method was developed chiefly to improve the flexibility of the PDT to avoid building a mesh. In this method, the state space is discretized into a grid of rectangles which allows for a more automated approach. Here, we extend the grid method to greater than two-dimensional models to expand the repertoire of possible neuron types.

2. Materials and methods

2.1. Recap of the grid method in MIIND

The MIIND algorithm for calculating the change to the probability mass function is covered in detail by de Kamps et al. (2019) and Osborne et al. (2021). However, we will cover the basic algorithm as it is relevant to the extension of the grid method to N dimensions. As a preprocessing step, the state space of the underlying neuron model is discretized such that each discrete volume of state space, or cell, is associated with a probability mass value. The probability mass is assumed to be uniformly distributed across the cell. The discretization can take the form of a mesh as shown in Figure 1A, constructed from the characteristic curves of the underlying neuron model or a regular grid which spans the state space as in Figure 1C.

When generating the grid in MIIND, the user provides the resolution of the grid and the size and location in state space within which the population is expected to remain during

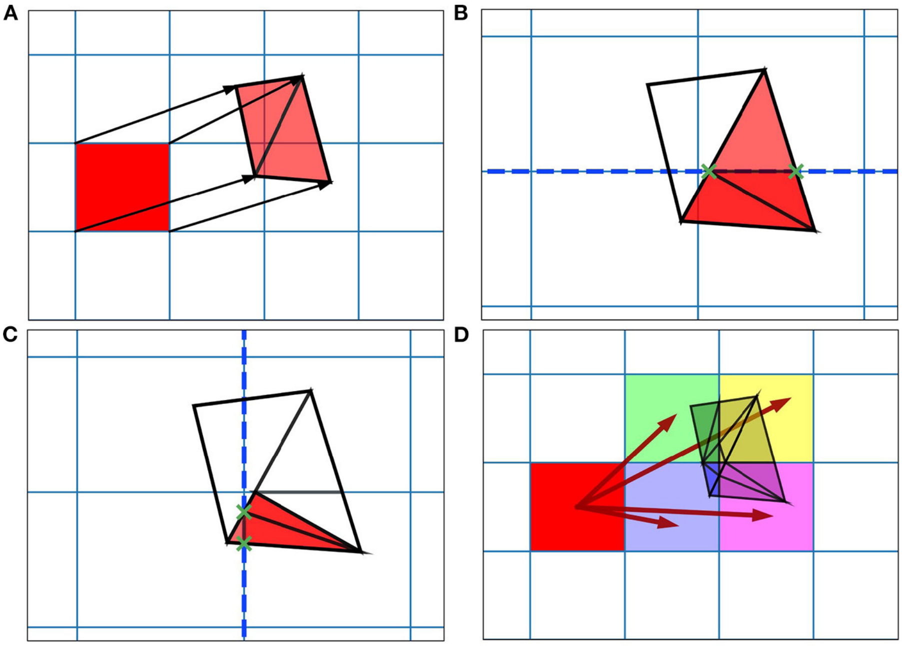


FIGURE 2
Figure showing steps for generating the transition matrix to solve the deterministic dynamics of the underlying model using a two-dimensional grid. Axes are not labeled as they represent arbitrary time-dependent variables. **(A)** For each grid cell (rectangle), the vertices translated according to a single time step of the underlying neuron model and the resulting quadrilateral is triangulated. **(B)** Each triangle is then tested for intersection with the axis-aligned lines of the original grid. The green crosses mark the intersection points between the tested triangle and the dashed line. The resulting subsections are again triangulated. **(C)** The process runs recursively until no more triangulations can be made. **(D)** The resulting triangles each lie within only a single cell of the original grid. The area of each triangle divided by the area of the original quadrilateral gives a proportion of mass to be transferred from the grid cell to the containing cell. From these, the proportions to be transferred can be summed and the totals stored in the file.

simulation. For each iteration of the simulation, the distribution of probability mass across the cells is updated, firstly, according to the deterministic dynamics of the underlying neuron model. For example, in the Izhikevich neuron model (Izhikevich, 2007), as shown in Figure 1B, the vector field below −60 mV indicates that probability mass will move slowly toward −60 mV before quickly accelerating to the right. Because the underlying neuron model does not change, the proportion of probability mass transitioning from each cell according to the deterministic dynamics remains constant throughout any simulation and can therefore be precalculated and stored in a file. To generate the file, the steps illustrated in Figure 2 are performed. For each cell, the aim is to calculate where probability mass will move after one time step of the simulation and how much of the mass is apportioned to each cell. First, the four vertices of the cell are translated according to a single time step of the underlying neuron model to produce a quadrilateral which is assumed to remain convex due to the small distance traveled. The quadrilateral is then split into two triangles and each triangle is then processed separately. Each triangle is tested against the axis-aligned edges of the grid. Because the lines are axis-aligned, this is a trivial test for points on either side of the line. If an intersection occurs, the new vertices are calculated to produce two polygons on either side of the line. Each polygon is triangulated and the process is recursively repeated on all sub-triangles until no more intersections occur. Once all triangles have been tested, the quadrilateral is now split into a collection of triangles which are each entirely contained within one cell of the grid. For each cell which contains one or more triangles, the total area of the triangles is calculated as a proportion of the area of the quadrilateral and this value represents the proportion of probability mass which will be transferred from the originating grid cell after one time step. It is expected that each transformed cell will only overlap with a few others in the grid so that an $N \times N$ matrix of transitions where N is the number of cells should be sparsely populated and can be stored in a file then read into memory. The transitions in the file are applied once every iteration of the simulation. This is a computationally time efficient way to solve the deterministic dynamics.

Once the probability mass distribution has changed according to the deterministic dynamics of the underlying neuron model, the second part of the MIIND algorithm calculates the spread of mass across cells due to random (usually Poisson distributed) incoming spikes. This process is more computationally expensive than the first because the shape of the spread must be recalculated every time step by solving

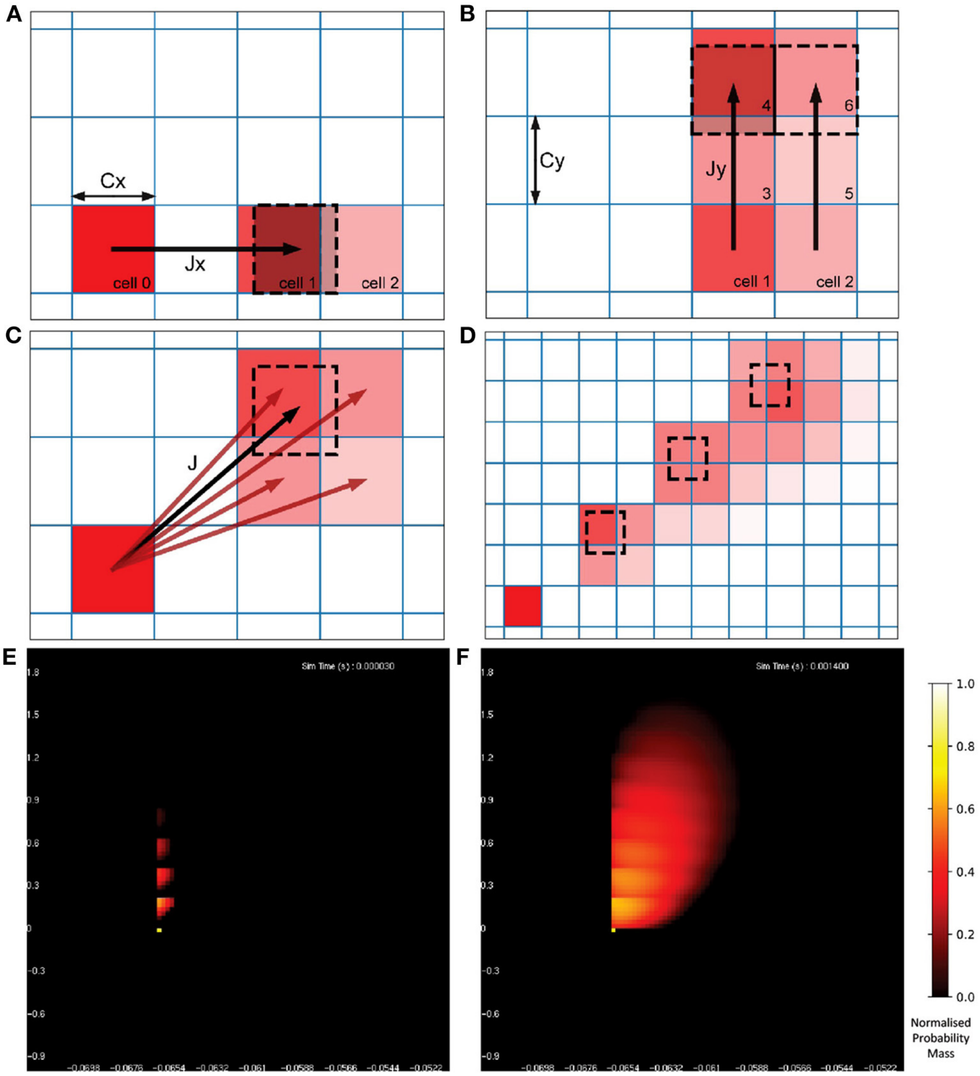


FIGURE 3
(A) The change in state, J, of a neuron due to a single incoming spike can be split into component parts, Jx and Jy for the horizontal and vertical dimensions, respectively. All neurons with a state within cell 0 will be translated by Jx due to a single incoming spike. Because all cells are the same width (Cx), the uniformly distributed probability mass of cell 0 will be shared among a maximum of two cells, cell 1 and cell 2. The offset of cell 1 from cell 0 is equal to $floor(Jx/Cx)$ [for negative Jx, it is $ceil(Jx/Cx)$] with cell 2 being the one beyond that. The proportion of mass transferred from cell 0 to cell 1 is equal to $1 - (Cx \, \% \, Jx)$ and the remainder is transferred to cell 2. **(B)** Once the mass proportions have been calculated in the horizontal direction, the same calculations are made with cells 1 and 2 in the vertical direction using Cy and Jy. The proportion calculated from cell 0 to cell 1 is split between cells 3 and 4. The proportion in cell 2 goes to 5 and 6. **(C)** The proportions of mass to be transferred from cell 0 to the resulting four cells give an approximation of the effect of transition J. With a constant J, this calculation gives the same relative results for every cell and therefore only needs to be performed once. **(D)** Iteratively applying the transitions to all cells in the grid spreads mass further across state space simulating the effect of neurons receiving multiple spikes in a given time step. **(E)** The probability mass function of a population of leaky integrate-and-fire neurons with an excitatory synaptic conductance rendered in MIIND. The color of each cell indicates the amount of probability mass. The value has been normalized to the maximum value of all cells. The effect of an incoming spike is to shift mass 0.2 nS/cm^2 in the vertical direction (producing a change in synaptic conductance). At this early point in the simulation, most neurons would have received zero or one spike (indicated by the bright yellow spots) while only a few would have received up to four spikes. **(F)** As the simulation proceeds, mass continues to be transferred upwards due to incoming spikes but the deterministic dynamics of the model causes mass to also move to the right according to the transitions defined in the matrix file and the population becomes more cohesive.

the Poisson master equation (de Kamps, 2006), which involves iteratively applying a different set of transitions to the probability mass function and is dependent on the incoming rate of spikes. Figure 3 shows how the spread of probability mass can be calculated in two dimensions based on the width of the cells and the change in state due to a single incoming spike. Calculating the transitions for solving the non-deterministic noise process benefits from the fact that all cells are the same size and regularly spaced. It is assumed that a single incoming spike will cause a neuron's state to instantaneously jump by a constant vector, J. Most often this is only in one direction instead of two. For example, many neuron models expect an instantaneous jump in membrane potential or in synaptic conductance. However, calculating the jump transition for any vector is a useful feature to have for models like the Tsodyks-Markram synapse model (Tsodyks and Markram, 1997) for which incoming spikes cause a jump in two variables at once. For a single incoming spike, all probability mass in a cell will shift up or down according to the x component of J, where x is the first variable or dimension of the model. Because all cells are the same size, this shift will result in probability mass being shared among at most two other cells which are adjacent to each other. Calculating which cells receive probability mass and in what proportion requires only knowing the width of the cells in the x dimension and the x component of J. If the J vector has a y component, where y is the second variable or dimension, the same process can be applied to each of the two new cells. The proportion of probability mass to be shared to each cell is itself shared among a further two cells for a maximum of four cells containing probability mass from the original cell. Due to the regularity of the grid, this calculation need only be made once and is applicable to every other cell. To simulate the effect of the incoming Poisson noise process on the probability mass function, the transitions are applied iteratively to each cell.

Figures 3E,F show the resulting probability mass function during a simulation when both deterministic and non-deterministic processes are applied. From the function, average values across the population can be calculated as well as the average firing rate if the underlying model has a threshold-reset mechanism. In that case, after each iteration, mass that has moved into the cells that lie across the threshold potential is transferred to cells at the rest potential according to a mapping generated during the pre-processing steps. The details of this mechanism are described by Osborne et al. (2021).

2.2. Extending the grid to N dimensions

An important observation is that the steps shown in Figure 2 for generating the two-dimensional transition matrix file work similarly in higher dimensions. However, the complexity of the algorithm increases significantly. For a three-dimensional underlying neuron model, the grid is extended such that each cell is a cuboid in state space with eight vertices (Figure 4). For an N-dimensional (ND) neuron model, an N-dimensional grid can be constructed with cells made up of 2^N vertices. The task here is to update the calculations involved in the deterministic and non-deterministic processes described above so that they work generically for any number of dimensions. For illustration purposes, we will use a three-dimensional grid.

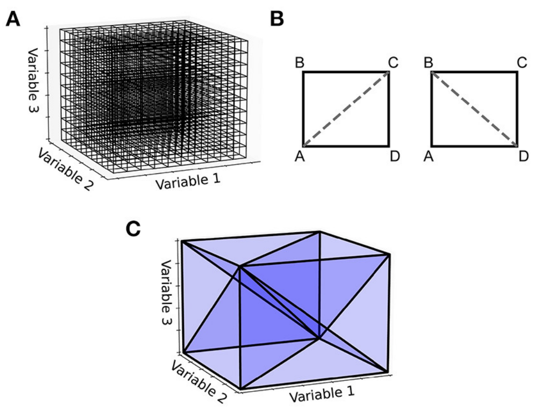


FIGURE 4
(A) With a three-dimensional state space, the grid discretization is made up of cuboids. **(B)** For the two-dimensional case, a rectangle has two possible triangulations, [A,B,C] and [A,C,D] or [A,B,D] and [B,D,C]. **(C)** A cuboid triangulated into six 3-simplices. Other triangulations are possible, some which aim to achieve the minimum number of simplices or to keep the volumes of the simplices as uniform as possible. The Delaunay triangulation makes no guarantees of this kind but is easy to implement and works in N-dimensions.

For the deterministic dynamics, each of the 2^N vertices is again translated according to a single time step of the neuron model and the resulting volume must be triangulated into N-simplices. In three dimensions, a 3-simplex is a tetrahedron. There are many possible triangulations of an N-dimensional cell. As an example, in the simpler two-dimensional case, if the four vertices of a rectangle are labeled A to D in a clockwise fashion as in Figure 4B, the possible triangulations are [A,B,C] and [A,C,D] or [A,B,D] and [B,D,C]. As with the number of possible triangulations, the number of resulting N-simplices increases with dimensionality and there are many algorithms available to generate them (Haiman, 1991). Many algorithms exist to find the so-called Delaunay triangulation of a set of points, which has a specific definition: A set of triangles (or N-simplices) between points such that no point lies within the circumcircle (or hypersphere) of any triangle (or N-simplex) in the set. This definition results in a quite well-formed triangulation (minimizing the number of long and thin triangles). One of the simplest ways to find the Delaunay triangulation of a set of points in N dimensions is to use the quickhull algorithm (Brown, 1979; Barber et al., 1996). The initial triangulation of the transformed cell is calculated using this method. To improve efficiency of this triangulation step, instead of finding the Delaunay triangulation for every translated cell, quickhull can be applied once to a unit N-cube as shown in Figure 4C. Under the assumption that the transformed cell remains a convex hull (not unreasonable given that the time step should

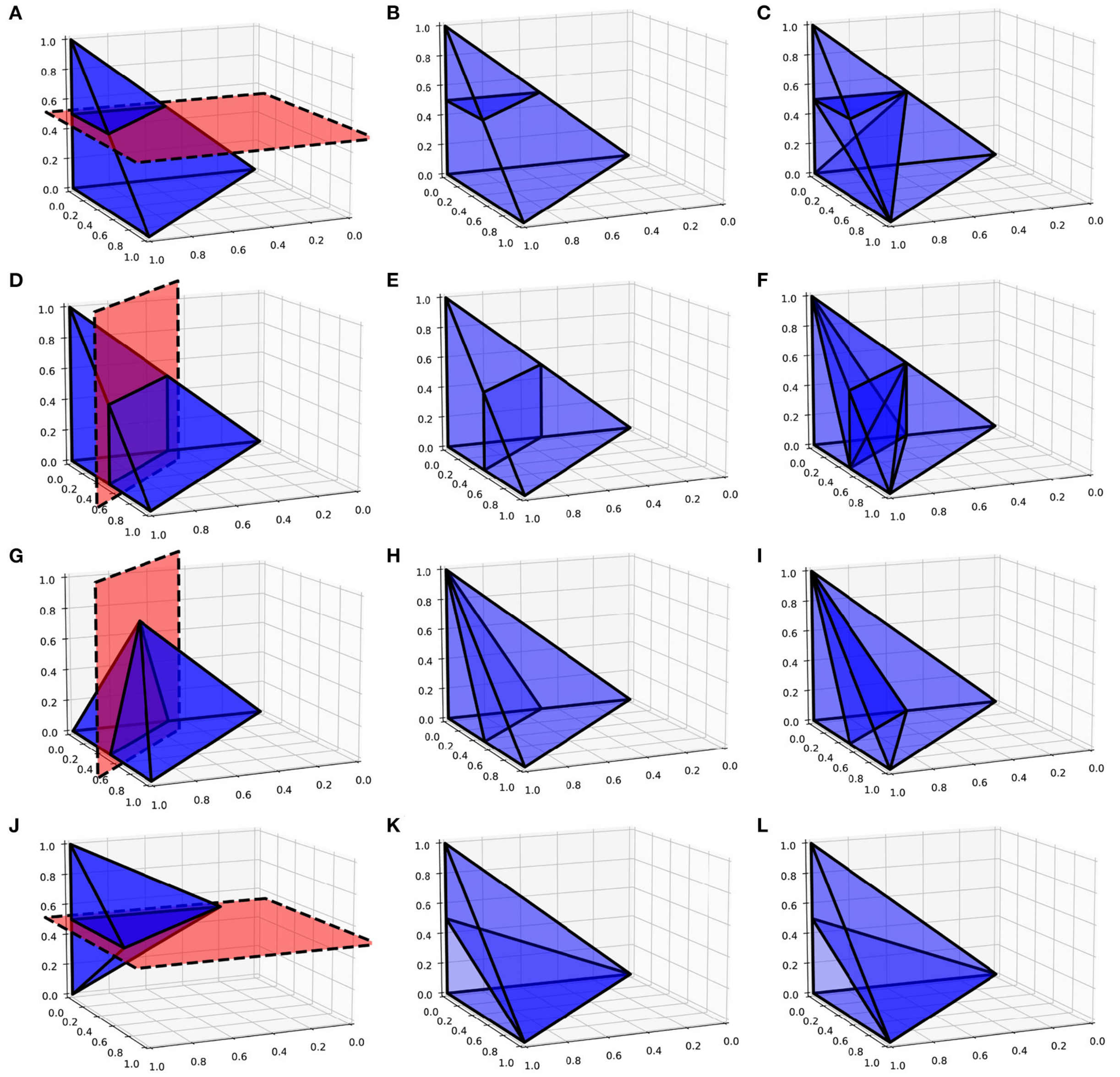


FIGURE 5
(A,D,G,J) Possible plane intersections with a 3-simplex. **(B,E,H,K)** Illustration of how each intersection is represented in the algorithm such that intersections bisect the relevant edges. **(C,F,I,L)** The resulting triangulations of the bisected 3-simplex which can be applied to all intersections of this type when calculating the transitions. **(A–C)** A plane intersection leaving one vertex of the 3-simplex above the plane and three vertices below. **(D–F)** A plane intersection leaving two vertices on either side of the plane. **(G–I)** A plane intersection which goes through one of the vertices leaving one vertex above the plane and two vertices below. **(J–L)** A plane intersection which goes through two vertices leaving one vertex on either side.

be small), the triangulation of the unit N-cube can be applied to every transformed cell without re-calculating.

As with the two-dimensional version, the next step is to recursively test each N-simplex for intersections with hyperplanes of the grid. Figure 5 shows examples of possible plane intersections of a 3-simplex. Finding an intersection, again, trivially involves checking if vertices lie on both sides of the hyperplane. The new vertices resulting from the intersections with the edges of the N-simplex describe two new shapes on either side of the plane. These must again be triangulated into smaller N-simplices. As with the first triangulation of the unit N-cube, pre-calculated triangulations of a unit N-simplex can be mapped to each newly generated N-simplex of the transformed cell. However, as Figure 5 shows, there are multiple ways that an N-simplex can be bisected with each requiring a different triangulation of the resulting shapes. Each type of intersection

TABLE 1 Possible vertex configurations from bisections of a 3-simplex.

Vertices above the plane	Vertices below the plane	Vertices on the plane	Resulting new vertices
1	3	0	3
2	2	0	4
1	2	1	2
1	1	2	1

can be described uniquely with the number of vertices above the hyperplane, below the hyperplane and on the hyperplane. Table 1 gives the possible bisections of a 3-simplex which are illustrated in Figure 5. The terms "above" and "below" are just used here to describe each side of the hyperplane and do not represent a position relative to each other or the hyperplane. Listing 1 gives the programmatic way to find all possible intersections of an N-simplex.

Listing 1 Calculate all possible vertex combinations to uniquely identify each type of intersection of an N-simplex

```
For each possible number of co-planar vertices
    which **is** between 0 **and** 2^N - 2:
    List all possible combinations of the
        remaining vertices above **and** below the
        hyperplane excluding 0
```

For each of the vertex combinations which uniquely identifies a type of intersection, the appropriate triangulation of the resulting shapes can be pre-calculated using the Delaunay triangulation of a unit N-simplex. To do this, the vertices of the N-simplex are assigned to be "above", "below" or "on" according to the vertex combination. At this point, no hyperplane exists to test for intersection points. However, we know that edges that pass between an "above" vertex and a "below" vertex will be intersected so we can choose to bisect that edge to produce a new vertex as shown in Figure 5. This represents a good enough approximation of the eventual N-simplex bisection and the quickhull algorithm can be performed on the resulting two shapes. The full dictionary of vertex combinations to triangulations is stored in a lookup table so that, during the actual subdivision of N-simplices in the grid, all that is required is to find the correct intersection in the table and to apply the triangulation mapping. As before, the algorithm continues recursively until no more triangulations are required and the volumes of all N-simplices are summed to calculate the proportion of probability mass which will be shared among the relevant cells.

Solving the non-deterministic dynamics in N dimensions is precisely the same as for two dimensions. In the same way that the probability mass proportion was recursively shared among two new cells per dimension, the resulting number of cells to which mass is transitioned due to a single incoming spike is at most 2^N. No intersections of triangulations are required for this calculation as only the cell width and the jump value in each dimension is required as shown in Figure 3. The MIIND algorithm proceeds in the same way as it did for two dimensions. First applying the matrix of transitions for the deterministic dynamics to the grid, then iteratively applying the jump transition to each cell multiple times to approximate the spread of probability mass due to Poisson distributed input. If the underlying neuron model has a threshold-reset mechanism, probability mass in the cells at threshold (for a three-dimensional grid, this is a two-dimensional set of cells) is transferred to a set of reset cells according to another pre-calculated mapping.

2.3. Running an ND simulation in MIIND

When implementing the ND extension to the grid method in MIIND, care has been taken to minimize any changes to how the user builds and runs a simulation. Listing 2 shows a MIIND simulation file for defining two neuron populations in an E-I configuration as examined later in Section 2.5.

Listing 2 The XML-style simulation file for an E-I network in MIIND

```
<Simulation>
<WeightType>CustomConnectionParameters</
    WeightType>
<Algorithms>
<Algorithm type="GridAlgorithmGroup" name="
    COND3D" modelfile="cond3d.model"
    tau_refractive="0.002" transformfile="
    cond3d.tmat" start_v="-65" start_w="0.00001
    " start_u="0.00001">
<TimeStep>1e-03</TimeStep>
</Algorithm>
</Algorithms>
<Nodes>
<Node algorithm="COND3D" name="E" type="
    EXCITATORY" />
<Node algorithm="COND3D" name="I" type="
    INHIBITORY" />
</Nodes>
<Connections>
<IncomingConnection Node="E" num_connections="10
    " efficacy="0.15" delay="0.0" dimension="1"
    />
<IncomingConnection Node="I" num_connections="10
    " efficacy="0.15" delay="0.0" dimension="1"
    />

<Connection In="E" Out="E" num_connections="50"
    efficacy="1" delay="0.003" dimension="1"/>
<Connection In="I" Out="E" num_connections="50"
    efficacy="4" delay="0.003" dimension="2"/>

<Connection In="E" Out="I" num_connections="50"
    efficacy="1" delay="0.003" dimension="1"/>
<Connection In="I" Out="I" num_connections="50"
    efficacy="4" delay="0.003" dimension="2"/>
```

```
</Connections>
<Reporting>
    <Display node="E" />
   <Average node="E" t_interval="0.001" />
   <Average node="I" t_interval="0.001" />
   <Rate node="E" t_interval="0.001" />
   <Rate node="I" t_interval="0.001" />
</Reporting>
<SimulationRunParameter>
<master_steps>10</master_steps>
<t_end>TE</t_end>
<t_step>1e-03</t_step>
<name_log>cond.log</name_log>
</SimulationRunParameter>
</Simulation>
```

The full details of the syntax for a simulation file is provided by Osborne et al. (2021). Little in this file has changed to accommodate higher dimensional neuron models. In the definition of the *Algorithm*, COND3D, the attributes *start_v*, *start_w*, and *start_u* allow the user to define the starting position (of a Dirac delta peak) for the population in the three-dimensional space. Similarly-named attributes can be added for higher dimensions. The *modelfile* and *transformfile* attributes should point to the required pre-processed files generated from the algorithm described in Section 2.2.

The *Connection* elements describe the inhibitory and excitatory connections between the two populations (nodes) E and I. As discussed earlier, each population simulated using the numerical PDT is influenced by one or more Poisson noise processes which change the probability mass function to approximate each neuron in the population receiving Poisson distributed spike trains. In MIIND, populations interact via their average output firing rate which becomes the rate parameter of the input Poisson process for the target population. Four such connections are set up here. The *num_connections* attribute indicates how many incoming connections each neuron in the target (*Out*) population receives from the source (*In*) population. This has the effect of multiplying the incoming firing rate parameter. The *efficacy* attribute gives the instantaneous jump value caused by a single incoming spike. The *dimension* attribute has been newly added and gives the direction in which the jump occurs. In this example, spikes from the excitatory population cause a change of 1 nS/cm^2 change in dimension 1 which corresponds to the *w* variable. Finally, the *delay* attribute gives the transmission delay of the instantaneous firing rate between populations which allows MIIND to simulate the complex dynamics which can arise when this is a non-zero value.

All other aspects of the file remain unchanged though the *Display* element which tells MIIND to render the probability mass function of population E during the simulation now causes a three-dimensional rendering of the function in state space. For higher dimensions, which three dimensions to display can be chosen during simulation.

The main change to MIIND to support ND neuron models is the addition of the *generateNdGrid* method in the MIIND Python module. Listing 3 shows a function set up in Python, *cond*, which describes the time evolution of a LIF neuron with excitatory and inhibitory conductances. The *generateNdGrid* method generates the *cond3d.model* and *cond3d.tmat* support files which are referenced in the simulation file above (listing 2). The method takes as parameters:

1. The Python function defining the model dynamics.
2. The name of the generated files.
3. The minimum values in state space.
4. The span of the grid in state space.
5. The resolution of the grid.
6. The threshold potential.
7. The reset potential.
8. Any additional change in state of a neuron after being reset to the reset potential (in this case, there is none).
9. The timescale of the neuron model in seconds.
10. The time step with which to solve the neuron model in seconds.

Listing 3 An example Python script to generate the support files for a three-dimensional LIF neuron population in MIIND.

```
import miind.miindgen as miindgen

def cond(y):
    V_l = -70.6
    V_e = 0.0
    V_i = -75
    C = 281
    g_l = 0.03
    tau_e = 2.728
    tau_i = 10.49

    v = y[2]
    w = y[1]
    u = y[0]

    v_prime = (-g_l*(v - V_l) - w * (v - V_e) -
        u * (v - V_i)) / C
    w_prime = -(w) / tau_e
    u_prime = -(u) / tau_i

    return [u_prime, w_prime, v_prime]

miindgen.generateNdGrid(cond, ’cond3d’,
    [-0.2,-0.2,-80], [5.4,5.4,40.0],
    [50,50,50], -50.4, -70.6, [0.0,0.0,0.0], 1,
     0.001)
```

Running a script such as this performs the steps outlined in Section 2.2. To see further examples of ND simulations in MIIND, once the software has been installed (using pip install miind), the *examples/model_archive* directory of the MIIND repository contains the required files for a number of different three- and four-dimensional neuron model populations. The three experiments presented below are available in the *examples/miind_nd_examples* directory of the MIIND repository.

TABLE 2 Parameters used for Equations (1) and (2).

Parameter name	Values and notes
Equation (1)	**Leaky integrate-and-fire neuron with an excitatory and inhibitory synaptic conductance**
g_l	0.03 nS/cm²
E_l	−70.6 mV
E_e	0.0 mV
E_i	−75 mV
C	281 pF/cm²
τ_e	2.728 ms
τ_i	10.49 ms
Refractive period	2 ms
Threshold potential	−50.4 mV
Reset potential	−70.6 mV
Equation (2)	**Hodgkin-Huxley Neuron**
g_l	0.5 mS/cm²
g_k	30 mS/cm²
g_{na}	100 mS/cm²
V_k	−90 mV
V_{na}	50 mV
V_l	−65 mV
C	1.0 μF/cm²
α_m	$0.32(13 - v + V_t)/(e^{\frac{13-v+V_t}{4}} - 1)$
α_n	$0.032(15 - v + V_t)/(e^{\frac{15-v+V_t}{5}} - 1)$
α_h	$0.128e^{\frac{17-v+V_t}{18}}$
β_m	$0.28(v - V_t - 40)/(e^{\frac{v-V_t-40}{5}} - 1)$
β_n	$0.5e^{\frac{10-v+V_t}{40}}$
β_h	$4/(1 + e^{\frac{40-v+V_t}{5}})$
V_t	-63 mV

2.4. Testing a single population

Initially, a single population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductance variables was simulated in MIIND and compared to a so-called Monte Carlo approach. The definition of the underlying neuron model is given in Equation (1) and the parameters are listed in Table 2. v represents the membrane potential, u represents the conductance of inhibitory synapses which will increase with increased inhibitory input. w represents the conductance of the excitatory synapses. C is the membrane capacitance and g_l is the leak conductance. V_l, V_e, and V_i are the reversal potentials for their respective conductances. The refractory period, during which the state is held constant at the reset potential, has been set to 2 ms. Figure 6 shows a schematic of the neuron model state space in three dimensions and the effect of excitatory and inhibitory input spikes. Due to the dynamics of the model, mass in cells with a high u value will move to lower values of v and mass at high w values will move to higher cells in v.

$$
\begin{aligned}
C\frac{dv}{dt} &= -g_l(v - V_l) - w(v - V_e) - u(v - V_i) \\
\tau_e\frac{w}{dt} &= -w \\
\tau_i\frac{u}{dt} &= -u
\end{aligned}
$$

$$v > threshold \longrightarrow v = reset \tag{1}$$

The Monte Carlo simulation was set up in Python for a population of 10,000 neurons following the dynamical system in Equation (1). For a time step of 1 ms, neurons receive a number of input spikes sampled from a Poisson distribution with a given rate parameter. Each spike causes a 1.5 nS/cm² increase in the excitatory synaptic conductance variable, w. Each neuron also receives excitatory and inhibitory Poisson noise at 50 Hz, again, with each excitatory spike causing a 1.5 nS/cm² increase in w and each inhibitory spike causing a 1.5 nS/cm² increase in u. Both u and w were set to 0 nS/cm² at the start of the simulation.

A MIIND simulation was similarly set up. Six separate grid transition files were generated all according to Equation (1) but with different grid resolutions: 50 × 50 × 50 (for u, w, and v, respectively), 100 × 100 × 100, 150 × 150 × 150, 100 × 100 × 200, 200 × 200 × 100, and 50 × 50 × 300. For all resolutions, the grid spans the model state space for u = −0.2 nS/cm² to 5.2 nS/cm², w = −0.2 nS/cm² to 5.2 nS/cm², and v = −80 to −40 mV. These ranges represent the limits of the values that the variables can take in the MIIND simulation but were chosen because all significant probability mass is contained in this volume throughout. All simulations produced 1.2 s of activity. The average membrane potential, synaptic conductances, and firing rate of the population were recorded.

Though MIIND has not been fully benchmarked, it is instructive to see the relative benefits to computational efficiency with differing grid resolutions. For the grid resolutions, 50 × 50 × 50, 100 × 100 × 100, 150 × 150 × 150, and 50 × 50 × 300, the time from starting the MIIND program to the beginning of the simulation was recorded to give an indication of the effect of load times with greater transition file sizes. Then the time to complete the simulation was recorded. The same simulation from above was performed without recording the membrane potential or firing rate to the hard drive. The machine used to produce the results has a solid state drive (SSD), an Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, and an NVidia Geforce GTX 1060.

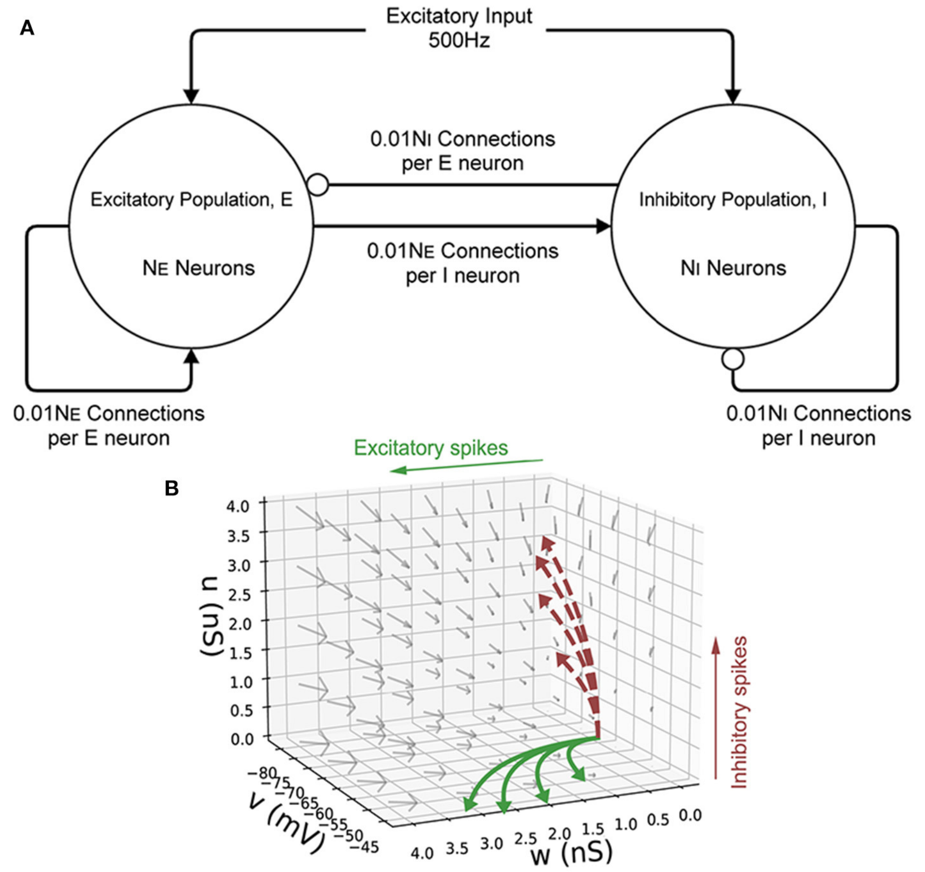


FIGURE 6
(A) A schematic of the E-I population network. The excitatory population, E is made up of N_E neurons. The inhibitory population, I contains $N_I = 10,000 - N_E$ neurons. Each population receives an excitatory external input of 500 Hz. Each neuron in both populations receives $0.01N_E$ excitatory connections and $0.01N_I$ inhibitory connections. Arrows represent an excitatory connection, circles represent an inhibitory connection. **(B)** The three-dimensional state space of the leaky integrate-and-fire neuron with an excitatory and inhibitory synaptic conductance. v is the membrane potential, w is the excitatory synaptic conductance, and u is the inhibitory synaptic conductance. The vector field shows the direction of motion in state space for neurons with no external impulse. Neurons which receive an excitatory input spike are shifted higher in w. Neurons which receive an inhibitory input spike are shifted higher in u. The solid curves show trajectories of neurons under excitatory impulse alone. The dashed curves show trajectories of neurons under inhibitory impulse alone.

2.5. An E-I network

To demonstrate how MIIND is able to simulate the interaction of multiple populations and capture changes in behavior with different parameters, a population network was set up in an E-I configuration (Brunel, 2000). Figure 6 shows the population level connections. In both the MIIND and Monte Carlo simulations, for each connection, the average firing rate of the source population is used as the rate parameter for the Poisson input to the target population. The Monte Carlo simulation was set up in Python for 10,000 neurons following the dynamics of Equation (1). Parameters for the neuron model and E-I network model are adapted from Sukenik et al. (2021). The 10,000 neurons are shared among the two populations according to a ratio parameter of excitatory to inhibitory neurons. That is, the number of inhibitory neurons, N_I was chosen and the number of excitatory neurons, N_E was set equal to $10,000 - N_I$. The excitatory and inhibitory conductance jump values are held constant and a weight is multiplied by the Poisson rate parameter of each connection to reflect that each neuron should receive $0.01N_E$ excitatory connections and $0.01N_I$ inhibitory connections. A transmission delay of 3 ms is applied to all inter-population connections. Finally, each population receives a 500 Hz excitatory Poisson distributed input with each spike causing a 1.5 nS/cm² jump in w. Table 3 gives the full list of parameters for the E-I model. MIIND was set up in the same way using a newly generated grid with resolution 150 × 150 × 150. The grid for this simulation covers a much larger volume of state space as it is expected that there will be large fluctuations in the conductance variables. Therefore, the size of the grid was set to $u = -10$ nS/cm² to 100 nS/cm², $w = -5$ nS/cm² to 25 nS/cm², and $v = -80$ to -40 mV. Across

TABLE 3 Parameters used for the E-I network model.

Parameter name	Values and notes
	Parameters apply to both the MIIND and Monte Carlo simulations
External firing rate	500 Hz to both E and I populations
External excitatory jump	1.5 nS/cm² change in w per incoming spike
N_I	Free parameter in the range 1,000–9,000
N_E	$10,000 - N_I$
Number of E to E connections	$0.01N_E$
Number of E to I connections	$0.01N_E$
Number of I to I connections	$0.01N_I$
Number of I to E connections	$0.01N_I$
Excitatory jump for E to E connections	1 nS/cm² increase in w per incoming spike
Excitatory jump for E to I connections	1 nS/cm² increase in w per incoming spike
Inhibitory jump for I to I connections	4 nS/cm² increase in u per incoming spike
Inhibitory jump for I to E connections	4 nS/cm² increase in u per incoming spike
E to E transmission delay	3 ms
E to I transmission delay	3 ms
I to I transmission delay	3 ms
I to E transmission delay	3 ms

simulation trials, all parameters were kept constant except for N_I.

2.6. A four-dimensional neuron population

To test the performance of MIIND with populations of four-dimensional neurons, we simulated a population of Hodgkin-Huxley neurons (Hodgkin and Huxley, 1952). This gold-standard model has not been simulated with a population density approach before. A fourth time-dependent variable significantly increases the amount of computation required to generate the transition matrix and its size beyond the three-dimensional case above. As before, a Monte-Carlo simulation was set up for comparison. The Hodgkin-Huxley neuron model is defined in Equation (2). As in Equation (1), the neuron has a capacitance, C, and a leak conductance, g_l, with reversal potential, V_l. The potassium and sodium synaptic conductances, g_k and g_{na} remain constant with respective reversal potentials, V_k and V_{na}. However, they are modulated by the three time dependent gating variables, n, m, and h. The definitions of α and β are given in Table 2.

$$\begin{aligned}
C\frac{dv}{dt} &= -g_k n^4(v - V_k) - g_{na}m^3h(v - V_{na}) - g_l(v - V_l)\\
\frac{m}{dt} &= \alpha_m(1 - m) - \beta_m m\\
\frac{n}{dt} &= \alpha_n(1 - n) - \beta_n n\\
\frac{h}{dt} &= \alpha_h(1 - h) - \beta_h h.
\end{aligned} \tag{2}$$

The population was given a Poisson distributed input at various rates between 0 and 40 Hz. The number of input connections to each neuron in the population was set at 100 and can be considered a weight so that the incoming rate would be multiplied by this amount. Each incoming spike produces a 3 mV jump in membrane potential. For MIIND, only one Hodgkin-Huxley grid was generated with dimensions 50 × 50 × 50 × 50 for h, n, m, and v, respectively. This resolution was chosen to keep the total number of cells low. The size of the grid was set between -0.1 and 1.1 for the gating variables, and $v = -100$ to 60 mV.

3. Results

3.1. A single population of three-dimensional neurons

Figure 7 shows the probability mass functions for six different simulations of a population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances. Each cell has a color/brightness and an alpha or transparency value such that cells with a higher probability mass are a brighter yellow, and more opaque than cells with lower probability mass which are darker red and more transparent. This plotting style allows the center of the function volume to be seen from the outside. Cells with zero probability are entirely transparent so that only significant cells are visible. Due to the greater opacity which often appears in the central volume of the function, the MIIND user may also rotate the entire volume to view the function from all angles. In Figures 7A,B, when only an excitatory input is provided, the function remains in the two-dimensional plane at $u = 0$ and is the same function as produced in the purely two-dimensional model demonstrated in de Kamps et al. (2019) and Osborne et al. (2021). Likewise, when only an inhibitory input is provided (Figures 7C,D), the function stays at $w = 0$. Figures 7E,F show the result of both an excitatory and inhibitory input. When enough excitatory input is provided, probability mass reaches the threshold membrane potential and is reset causing a sharp cut-off at those values. The brighter yellow cells in the center of the function's volume indicate that the majority of neurons can be found there traveling from the reset to threshold potential receiving close to the average number of excitatory and inhibitory input spikes. Further out, at higher

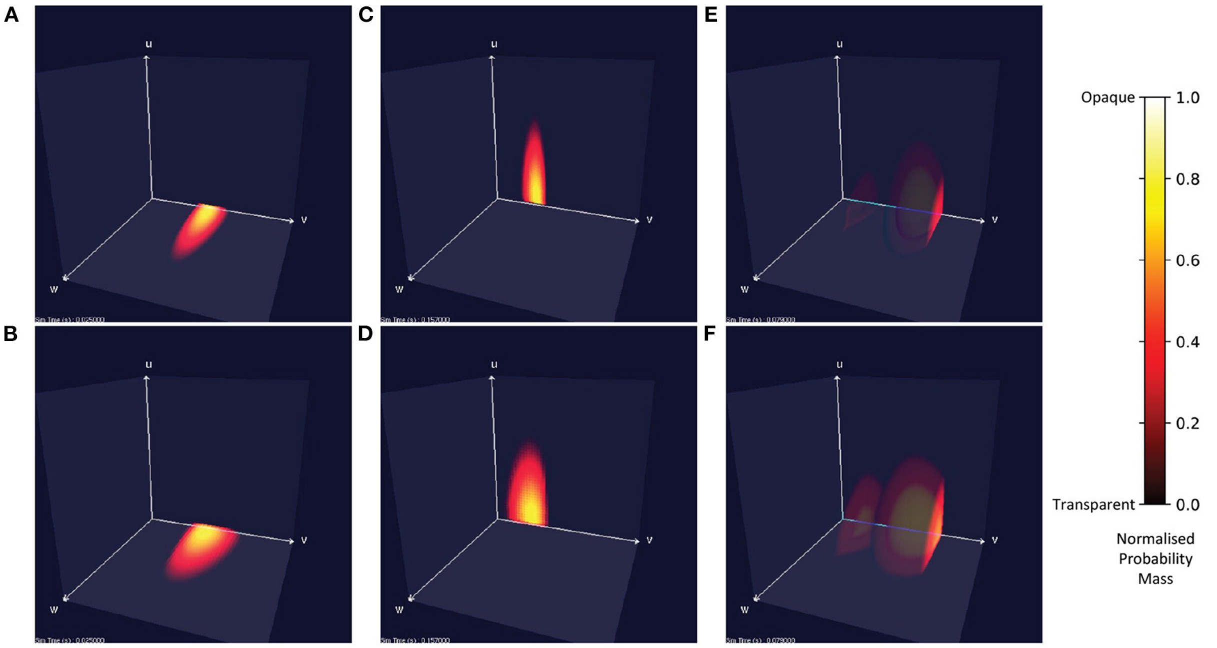


FIGURE 7
Visualizations of a population of leaky integrate-and-fire neurons with an excitatory and inhibitory synaptic conductance in MIIND. Cells with no probability mass are transparent. With increasing probability mass, they become more opaque and change from red to yellow. The color and opacity are normalized to the value of the cell with the highest probability mass. **(A,C,E)** The probability mass function across a 150 × 150 × 150 grid. **(B,D,F)** The probability mass function across a 50 × 50 × 50 grid for the same simulation time as the image above. **(A,B)** When the population receives only excitatory incoming spikes, the probability mass function remains in the plane at $u = 0$. **(C,D)** When the population receives only inhibitory incoming spikes, the probability mass function stays in the plane at $w = 0$. **(E,F)** When the population receives both inhibitory and excitatory incoming spikes, the probability mass function extends into the state space. In this case, the excitatory input is enough to overcome the inhibitory input and the mass function moves across the threshold potential. The bright face shows the probability mass at the threshold. Probability mass which has been reset reappears at the reset potential and moves further into the state space.

values of u and w, the probability of finding a neuron reduces as neurons are less likely to receive many more spikes than average.

Figure 8 shows average membrane potential recorded from multiple simulations of a population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances. The scatter points show the average potential of 10,000 individual neurons simulated using the Monte Carlo approach. The remaining curves show the average potential of populations simulated in MIIND using 3-dimensional grids of different resolutions. For the transient period before the membrane potential reaches a steady state, all the MIIND simulations remain synchronized with the Monte Carlo results. As would be expected, the least accurate result comes from the lowest resolution grid, 50 × 50 × 50. However, even at this resolution, the mean error between the Monte Carlo activity and the MIIND result is only 0.354 mV. The error is reduced significantly for 100 × 100 × 100 (0.115 mV) then further reduced but only slightly for 150 × 150 × 150 (0.063 mV) suggesting a degree of diminishing return for increasing the resolution in an equal fashion across dimensions. The error from the 200 × 200 × 100 grid is the same as the 100 × 100 × 100 grid but the 100 × 100 × 200 grid does better (0.059 mV) indicating that increasing the resolution of the membrane potential dimension is a more efficient way to attain accurate results for this underlying neuron model. To illustrate this further, the 50 × 50 × 300 grid performs the best of the trials with an average error of 0.054 mV despite the low resolution of the conductance dimensions. Over a range of average rates (Figure 8B) of the Poisson distributed input, the steady state membrane potential of the MIIND simulations, again, approaches those of the Monte Carlo results with increasing resolution. For low input rates, when the majority of neurons are subthreshold, the 150 × 150 × 150 grid gives the closest approximation to the Monte Carlo results. However, once the majority of neurons are crossing the threshold and firing, the 50 × 50 × 300 grid gives better agreement. Figure 8C shows the average excitatory conductance variable for the grids across the range of lower input rates (1–10 Hz) in comparison to the Monte Carlo approach. The 50 × 50 × 300 grid underestimates the conductance which could account for the underestimation of the membrane potential for the same input. The 150 × 150 × 150 grid, by contrast, has better agreement with the membrane potential and excitatory conductance for these rates which produce mostly sub-threshold activity in the population.

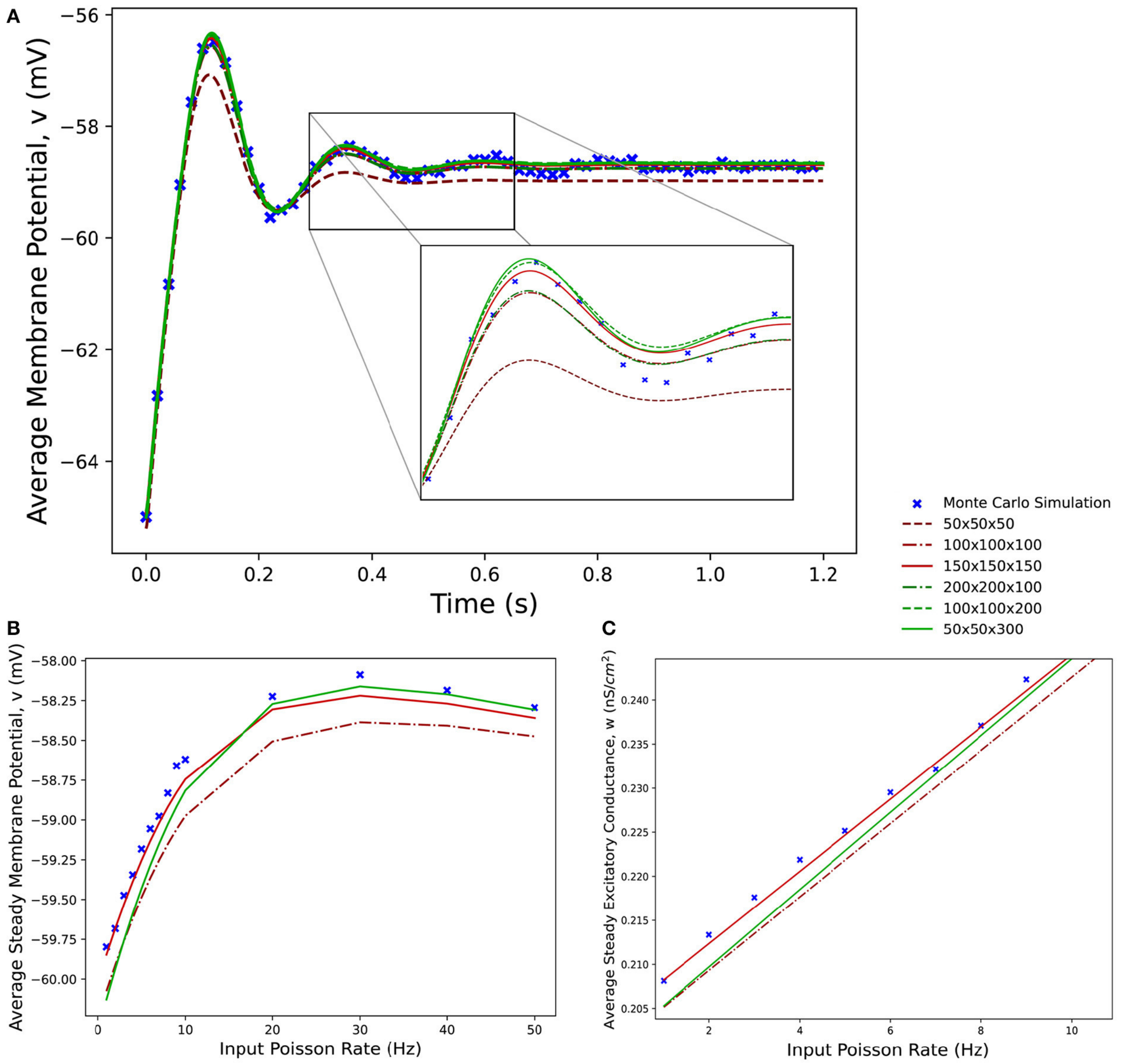


FIGURE 8
(A) The average membrane potential for a single population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances simulated using a Monte Carlo approach and using MIIND with grids of different resolutions. **(B)** The effect on the average steady state membrane potential with different rates of the Poisson distributed input for the Monte Carlo simulation and different MIIND grid resolutions. **(C)** The effect on the average steady state excitatory conductance variable with increasing Poisson input rate. Only the mean of the values for the Monte Carlo simulation are shown here (without a variance or standard deviation) because the MIIND simulation produces no such statistic and so no comparison can be made.

Figure 9 shows the average firing rates of the same Monte Carlo and MIIND simulations. The differences in grid resolution produce a similar trend in error, with the lowest resolution, 50 × 50 × 50 laying furthest away from the Monte Carlo simulation and the 50 × 50 × 300 grid the closest. However even at lower resolutions, all the average firing rates of the MIIND populations are very well matched to direct simulation.

3.2. Simulation speed for different grid resolutions

Table 4 shows the load and simulation times for 1 s of a single population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances. As expected, as the total number of cells increases the load times and simulation times increase. When running multiple short simulations, the load

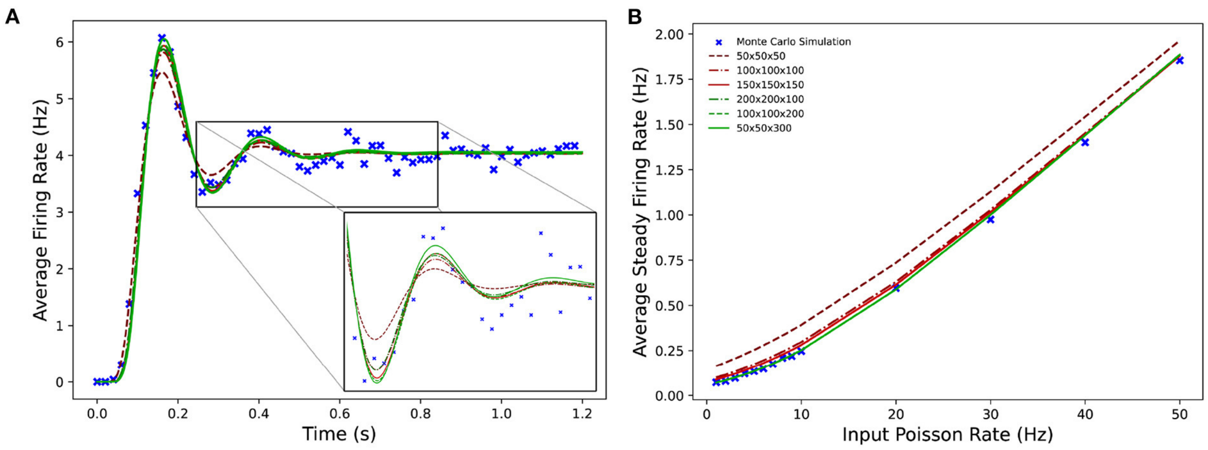


FIGURE 9
(A) The average firing rate of a single population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances simulated using a Monte Carlo approach and using MIIND with grids of different resolutions. **(B)** The effect on the average steady state firing rate of the population with increasing rate of the Poisson distributed input.

TABLE 4 Times to simulate 1 s of a population of leaky integrate-and-fire neurons with excitatory and inhibitory synaptic conductances in MIIND using different grid resolutions.

Grid resolution	Time to load the grid (s)	Time to run the simulation (s)
50 × 50 × 50	4.82	2.71
100 × 100 × 100	35.58	15.18
150 × 150 × 150	126.01	48.7
50 × 50 × 300	27.62	11.93

time becomes a significant consideration. However, only the simulation time is dependent on the required length of the simulation. The load time remains constant.

3.3. Three-dimensional neurons in an E-I population network

For the Monte Carlo simulation of 5,000 excitatory and 5,000 inhibitory neurons (with an average of 50 excitatory and 50 inhibitory incoming connections to each), the two populations reach an equilibrium state after an initial transitory phase. Figure 10A shows excellent agreement between the average membrane potentials from the two approaches. The initial oscillation in the transient period covers nearly 100 nS/cm² in u and 12 nS/cm² in w which requires a much larger volume of state space than the single population simulation because of the large synaptic efficacies and recurrent connections involved in the E-I network. In MIIND, as the oscillations reduce, the state space covered by the probability mass function reduces and is therefore discretized by fewer cells. In other words, the cell density covering the function is lower. However, this only causes a minimal amount of additional damping to the oscillation as the function reaches equilibrium.

In the Monte Carlo simulation, with 8,000 excitatory neurons and 2,000 inhibitory neurons, both populations produce an oscillating pattern as shown in Figure 10B. In the MIIND simulation, in order to match the connection ratios between populations, the number of excitatory and inhibitory connections is set to 80 and 20, respectively. The simulation is also able to produce a similar oscillatory pattern. As would be expected, the population density approach produces a regular oscillation while the Monte Carlo has some variation in the length and amplitude of each oscillation. The double peak of each oscillation can be explained by observing the probability mass function in MIIND during the simulation (Figures 10C–H). The initial peak is produced as the whole population depolarizes and approaches the threshold potential. As mass begins to pass the threshold, the reset mass brings the average membrane potential back down. The probability mass is pushed higher in w and u as the recurrent excitatory input and inhibitory input from the other population increase. The excitatory input has the strongest effect on the probability mass close to the reset potential which begins to push the average membrane potential back up toward a second peak. The inhibitory input has the strongest effect on the probability mass close to threshold and less and less mass reaches threshold. The split probability mass function coalesces once more and the cycle can repeat.

When the ratio of excitatory to inhibitory neurons is 9:1, the Monte Carlo simulation demonstrates how the excitatory

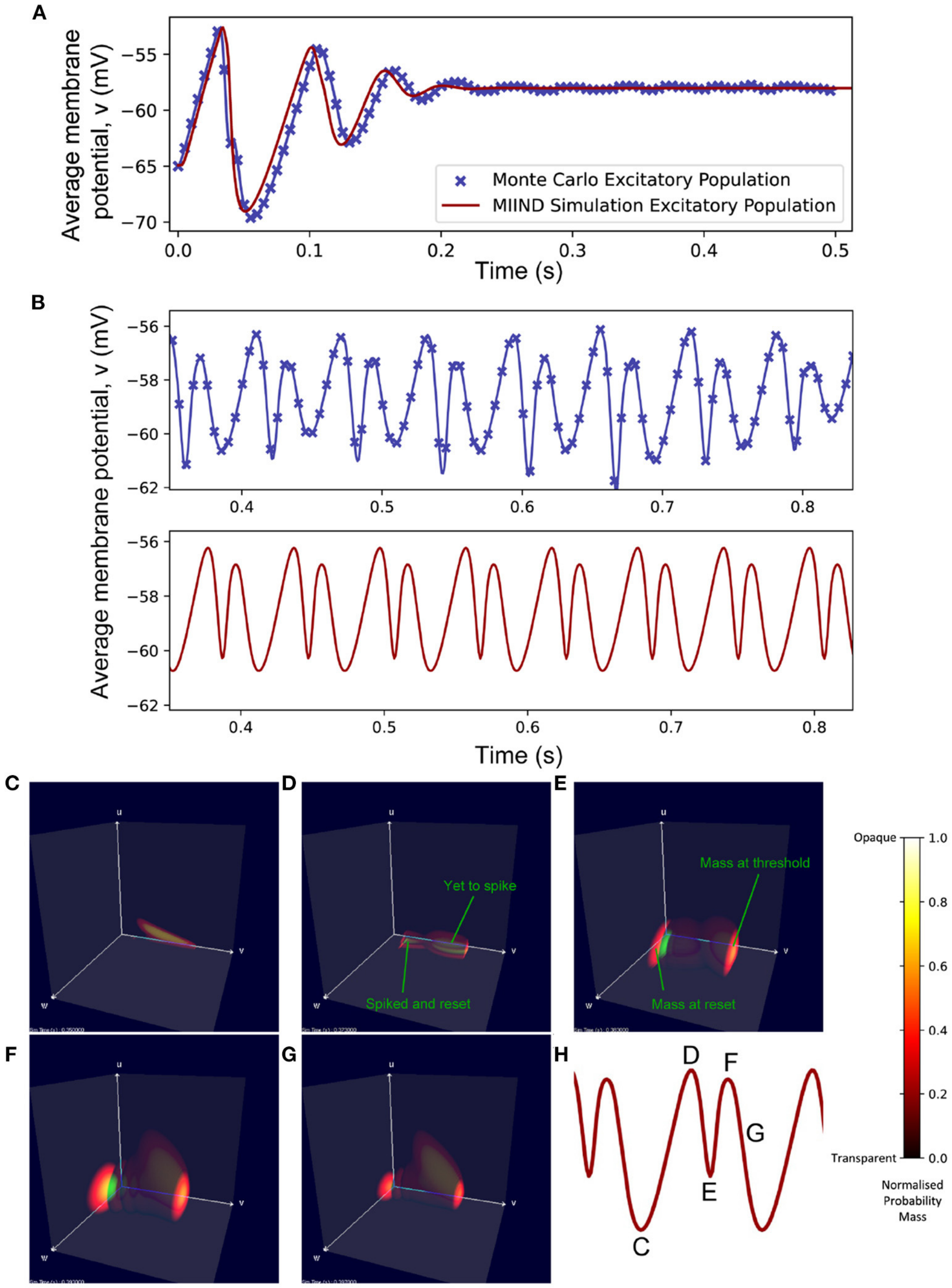


FIGURE 10
(A) The average membrane potential of the excitatory population in the E-I network with a ratio of 1:1 excitatory and inhibitory neurons ($N_E = N_I$). In the MIIND simulation, each connection between populations has the "number of connections" value set to 50. **(B)** The average membrane potential of the excitatory population in the E-I network with a ratio of 8:2 excitatory to inhibitory neurons ($N_E = 8,000, N_I = 2,000$). In the MIIND simulation, the excitatory connections have the "number of connections" value set to 80 and the inhibitory connections have the "number of connections" value set to 20. For clarity, the traces from the Monte Carlo simulation and the MIIND simulation have been separated. **(C–H)** The probability mass function for the excitatory population in MIIND during the double peaked oscillation with a connection ratio of 8:2. **(H)** shows the corresponding points in the oscillation. At **(C)**, The population only experiences the external input of 500 Hz and is pushed toward the threshold. At **(D)**, though some probability mass has passed threshold and been reset, the majority is close to the threshold and so the average membrane potential is at a peak. At **(E)**, probability mass has continued to cross the threshold so that now a large amount is near the reset potential which brings the average back down. The function has also shifted higher in w due to the excitatory self-connections and more probability mass is pushed across threshold. The function also begins moving upwards in u from the increased inhibitory input but this is not enough to overcome the excitation. At **(F)**, the inhibitory input has continued to push the probability mass function higher in u and much less probability mass now crosses the threshold. At **(G)**, the function continues to shift back away from threshold approaching **(C)** once again.

self-connection causes the excitatory population activity to "blow-up" such that the excitatory conductance reaches a maximum value and neurons fire at their maximum rate. This state is a challenge for MIIND to emulate. Firstly, the number of iterations required to solve the Poisson master equation each time step must be increased to 1,000 due to the instability caused by such high firing rates. Secondly, the excitatory conductance variable frequently approaches 80 nS/cm^2 and so the grid must cover a large amount of state space requiring an unreasonable resolution in the w dimension to maintain the same cell density as previous simulations.

3.4. A single population of four-dimensional Hodgkin Huxley neurons

Even with a low resolution of 50 × 50 × 50 × 50, MIIND is able to simulate the probability mass function of a population of Hodgkin Huxley neurons and achieve good agreement with the transient activity and steady state membrane potential of an equivalent Monte Carlo simulation (Figure 11A). Figure 11B shows how, for a range of input firing rates, the resulting average membrane potential at steady state approximates that of the Monte Carlo simulations better for higher frequencies. At low input rates, the membrane potential is overestimated. MIIND displays the probability mass function for three of the four dimensions at a time as shown in Figure 11C. With a key press, the user can change the order of dimensions displayed and see any combination of variables. Figure 11F shows the three gating variables, m, n, and h.

4. Discussion

The original motivation for applying a population density approach to simulate neurons was to reduce the computational complexity when analyzing a large population of homogeneous neurons. This has since been made somewhat redundant with the development of more powerful computers and especially the use of GPGPU architectures. For example, GeNN (Yavuz et al., 2016; Knight et al., 2021) can simulate in real time the well known Potjans-Diesmann microcircuit model (Potjans and Diesmann, 2014) which comprises around 10,000 neurons. The analytical solution for the behavior of a leaky integrate-and-fire population developed by Omurtag et al. (2000) using the diffusion approximation would undoubtedly prove efficient, requiring only a single calculation per population per time interval. However, it would require a lot of manual work to define the full population network and if a different neuron model were to substitute the integrate-and-fire neuron, the entire solution would need to be re-derived. Even with newer techniques such as the refractory density approach, work is required to get the underlying neuron model in a form that can be processed. MIIND uniquely overcomes this limitation allowing the user to define the neuron model without any further manual process to produce a numerical solution to the population density approach. However, solving the master equation for the non-deterministic noise component of the dynamics requires repeated applications of the jump transitions shown in Figure 3. As discussed by Osborne et al. (2021), depending on the model, the time step, and the input firing rate, solving the master equation can require tens or hundreds of iterations per time step of the simulation. This was the case for the EI network in the 8:2 ratio. The sharp changes in firing rate of the two populations combined with large synaptic conductance jumps meant that solving the master equation required 100 iterations per cell per time step to remain stable. The numerical population density approach in MIIND should, therefore, not be used for simulations where computational speed is the most important factor. However, it has been shown (de Kamps et al., 2019) that there is at least an order of magnitude improvement in memory consumption over direct simulation techniques, such as that of NEST, as there is no requirement to store the spike history.

Although computational efficiency is not the primary reason for using the population density approach, there are some benefits to generating the probability mass function over a direct simulation of individual neurons. The probability mass function can be considered the idealized distribution of neuron states. Cells in the grid which have zero mass correspond to volumes of state space where neuron states should never appear. This can be difficult to approximate with a direct simulation of individual neurons for parts of the distribution with a low but non-zero probability mass. Inconsistencies between the behavior of real neurons and a model could be identified more effectively by comparing to the probability mass function. Also due to the idealized probability mass function, the output metrics of a population such as average firing rate and average membrane potential have no variation due to noise or a specific realization of the Poisson distributed input. Therefore, no averaging or smoothing is required to produce more readable results as would be expected from a direct simulation (Figures 8, 9). In the E-I network, 10,000 Monte Carlo neurons was enough to produce a similar result to the MIIND simulation. But when that number is reduced to 1,000, there is greater variation in the firing rate and average membrane potential of the population. In the E-I network, a temporarily high number of spikes from the excitatory population leads to increased excitatory input 3 ms later and increased inhibitory input 3 ms after that. The resulting reduction in average membrane potential and firing rate is therefore exaggerated which produces an overall skew of these metrics. A population in MIIND can be thought of as an infinite number of trials of a single neuron or as an infinite number of neurons performing a single trial once. Because of this, a MIIND simulation is independent of the number of

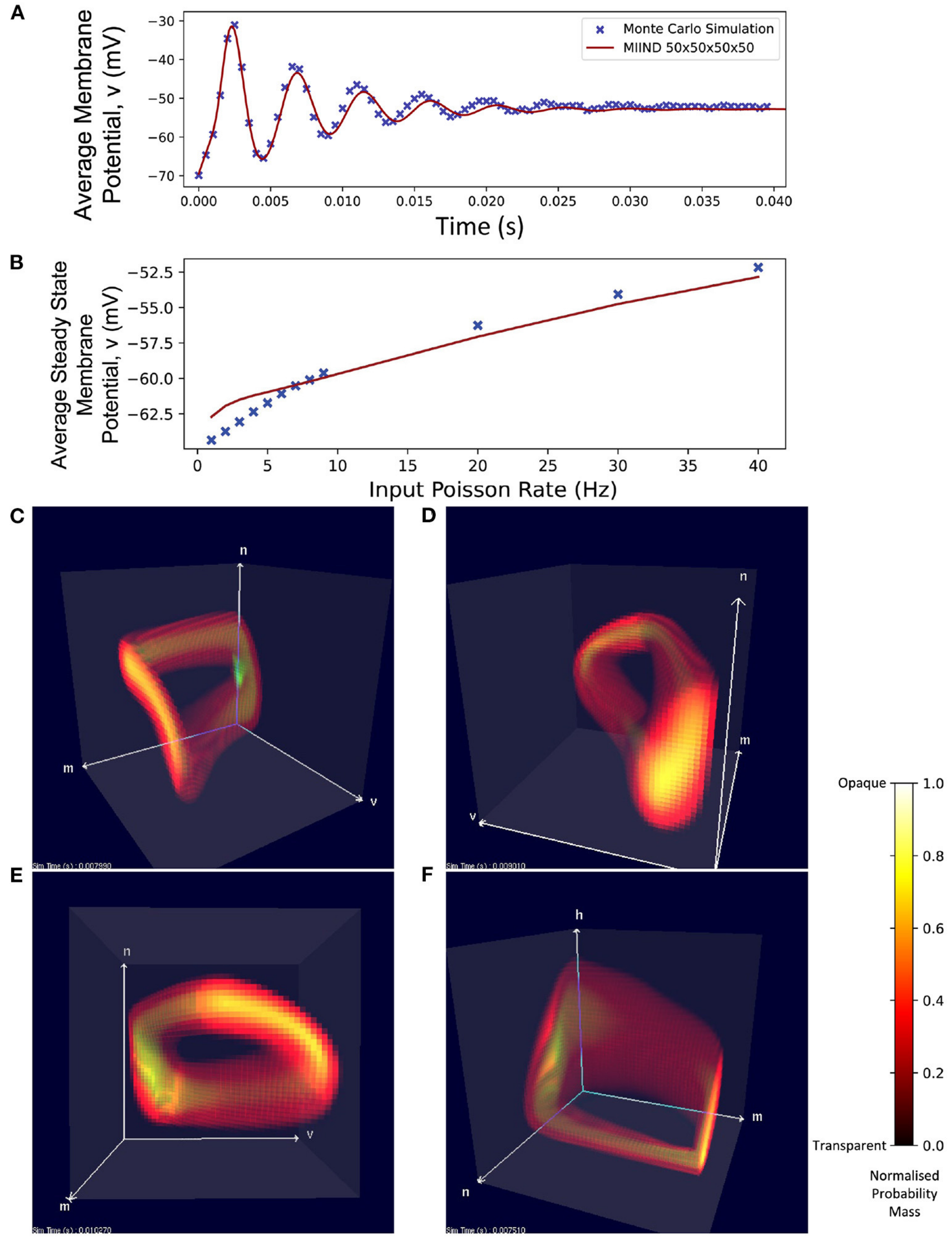


FIGURE 11

(A) The average membrane potential of a single population of Hodgkin-Huxley neurons simulated using a Monte Carlo approach and in MIIND with a $50 \times 50 \times 50 \times 50$ grid. **(B)** The average steady state membrane potential with different rates of the Poisson distributed input. **(C–E)** The three-dimensional marginal probability mass function of the four-dimensional Hodgkin-Huxley neuron population in MIIND having reached a steady state. The membrane potential v, sodium activation variable m, and potassium activation variable n are shown. **(F)** The three-dimensional marginal probability mass function showing the gating variables, w, n, and h (sodium inactivation variable) only.

neurons in the population and cannot produce so-called finite size effects. This can be a useful feature as it is not always as clear from a Monte Carlo simulation what behavior stems from the finite size and what is a population level effect.

Finally, the visualization of the probability mass function in MIIND could prove to be a valuable educational tool for understanding the behavior of neural populations under the influence of random spikes. In fact, any N-dimensional dynamical system under the influence of shot noise could be observed although this has not been attempted. It would be easy enough to plot points in a three-dimensional state space for individually simulated neurons but points at the front of the distribution would obscure those at the back and in the center. Producing a smooth enough distribution and to pick an appropriate transparency value for each cell would require a population of millions of neurons.

The increased time to produce the probability mass function over direct simulation does not negate the usefulness of lower resolution grids to improve the simulation time as shown in Table 4. In particular, using a low resolution grid can greatly improve workflow when designing or prototyping a new model. Building a model which performs as required involves multiple runs of the simulation as parameters are adjusted or when errors are identified. This is another reason why it is convenient that MIIND renders each population's probability mass function while the simulation is running. As shown in Figure 11, viewing the probability mass function across all dimensions from any angle as the simulation progresses gives both insight into how the population behaves and any unexpected behavior is quickly identified.

4.1. What is the theoretical output spike distribution of a population in MIIND?

Different populations in MIIND interact via their average firing rates. For each connection, the average firing rate of the source population is taken as the rate parameter to a Poisson distributed input to the target population. For a one-dimensional neuron model such as a leaky integrate-and-fire neuron for which incoming spikes cause an instantaneous jump in membrane potential, it is reasonable to assume that neurons in the population are pushed over threshold directly and only due to the Poisson distributed input suggesting that the output distribution should also be Poisson distributed. However, in higher dimensional models with, for example, the addition of excitatory and inhibitory synaptic conductances, it becomes clear that neurons can move across threshold without direct influence from the Poisson input. If a sample of neurons are taken from the probability mass distribution at the beginning of a simulation, by definition, the probability that each sampled neuron is above threshold in a given time step is the probability mass which sits above threshold to be transferred to the reset potential. As the behavior of all neurons are independent by virtue of being unconnected and homogeneous, the distribution of spiking neurons from the population at each time step can be considered binomial with p equal to the total probability mass above threshold. Using the average firing rate as the parameter to a Poisson input for each population is therefore a reasonable approximation. Models such as the E-I network which have self-connections and loop-connections invalidates the assumption of independence and further work is required to assess if using Poisson distributed outputs is appropriate under such circumstances.

4.2. Finite size populations

The main function of MIIND is to use the numerical population density approach to simulate population behavior. However, a population of finite size can also be simulated which makes use of the transition matrix file and calculated jump transitions. This hybrid version of the algorithm is closer to direct simulation. A list of M grid coordinates is stored which represents the location in state space of M individual neurons. At each time step, each coordinate is updated to one of the possible transition cells defined in the transition file with probability equal to the proportion of mass in that transition. To capture the non-deterministic dynamics, a Poisson distributed random number of spikes is sampled and the calculated jump transition is applied that many times. Again, the jump transition mass proportions are used as the probability for choosing the coordinate update with each jump. The average firing rate of the population is the number of neurons above threshold (which are then translated to the reset potential) divided by M. Currently direct connections between neurons is not implemented and instead, the average firing rate is used as the Poisson rate parameter applied to all neurons in the target population. Because the Poisson master equation is not required to solve the non-deterministic dynamics, this algorithm is much faster than the population density technique and approaches the speeds of simulations in GeNN although this has not been fully benchmarked. The two main reasons for using the finite size algorithm in MIIND are to further speed up prototyping of new models and to more easily eliminate finite-size effects.

4.3. Other potential models for study

The ability to easily simulate populations of three- and four-dimensional neuron models opens a world of possibilities for the population density approach. The Tsodyks-Markram synapse model (Tsodyks and Markram, 1997), for example, can be combined with a leaky integrate-and-fire neuron model to define a four-dimensional system. In the original work, the model was shown to support both rate coding between neurons and more precise spike timing based on the configuration of resource management in the synapse. For a large population, simulating the rate coding configuration makes more sense but MIIND could also be used to investigate the resilience of the spike timing configuration to noise. Booth and Rinzel (1995) developed a two-compartment minimal motor neuron model. Each compartment requires two dimensions and MIIND would therefore be able to simulate a population of both compartments together. This model can reproduce the bi-stable behavior of motor neurons such that a suitable incoming excitatory burst of spikes can shift the population to an up state where it remains even in the absence of further input. This is a candidate for identifying any finite size effects and, in the presence of noise, estimating the amplitude and duration of the required excitatory and inhibitory bursts to switch states.

4.4. Limitations

The population density approach suffers from the so-called curse of dimensionality. With each additional time-dependent variable in the underlying neuron model, the number of cells in the grid is multiplied by the resolution of the new dimension. Not only does this produce an exponential increase in the number of cells for which the deterministic and non-deterministic dynamics must be solved, but the number of transitions per cell in the transition file also increases in most cases. The 50 × 50 × 50 × 50 transition file for the Hodgkin Huxley model runs to nearly 1.5 Gb all of which must be loaded into graphics memory. There is still work to do to improve the memory management in MIIND but it is likely that a 5-dimensional transition matrix would not fit in the memory of current graphics hardware. In addition, generating the Hodgkin-Huxley transition file takes over 100 h on the four CPU cores of a typical PC. This is a one-time preprocessing requirement that can be mitigated somewhat with high performance computing systems but, again, for higher dimensional models the time required would become unfeasible.

In many cases, the number of cells in the grid that contain a non-zero amount of probability mass at any time during the simulation is much lower than the total number of cells. For higher dimensions it would be possible to calculate the non-deterministic dynamics transitions required for a cell when probability mass is first transferred to it during the simulation. The simulation would be considerably slower at the beginning but would approach the original speeds as more cells are calculated. The memory requirements would only come from the cells involved in the probability mass function. This adaptation would still have an upper limit on the number of dimensions as the number of involved cells would still increase with greater dimensionality but it would be far from the exponential increase currently.

Another potential method for improving performance in both memory and computation speed would be to relax the requirement that all grid cells are the same size. In areas of state space where the dynamics are expected to follow a shallow curve (as opposed to the sharp turns in state space which can occur near unstable stationary points for example), larger cells could be defined. In order to preserve the benefit of equally sized cells when calculating the jump transition, the larger cells could be subdivided at simulation time and the deterministic dynamics transitions into the large cell could be linearly interpolated throughout. While not significantly affecting the computation time, the memory requirements would improve with the reduced number of transitions.

5. Conclusion

We have demonstrated for the first time, a numerical population density technique to simulate populations of N-dimensional neurons. Although models of higher than 5 dimensions are currently technologically out of reach, it is a significant achievement to produce the probability mass function of a population of 4-dimensional Hodgkin-Huxley neurons and to be able to visualize it in such a fashion. Implementing this technique in MIIND results in a very low barrier to entry for new users allowing them to define their desired neuron model in Python, automatically generate the required transition files and run the simulation without expert knowledge of the technique or any involved technical knowledge beyond some basic Python and XML. Although originally conceived as a technique to improve computational efficiency when simulating large populations of neurons, the population density technique cannot achieve the speeds of some other simulation methods. However, there are a number of benefits to using it, particularly in the areas of theoretical neuroscience and as a tool for analysis.

Data availability statement

The MIIND source code and installation packages are available as a github repository at https://github.com/dekamps/miind. MIIND can be installed for use in Python using "pip install miind" on many Linux, MacOS, and Windows machines with python versions ≥3.6. Documentation is available at https://miind.readthedocs.io/. The leaky integrate-and-fire model, E-I network model, and Hodkin-Huxley model can be found in the examples/model archive directory of the MIIND repository on github.

Author contributions

The article and code development was undertaken by HO with support and advice from MK. All authors contributed to the article and approved the submitted version.

Funding

This project received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 785907 (Human Brain Project SGA2) (MK). HO was funded by EPSRC (EP/N509681/1). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Acknowledgments

The authors wish to thank Frank van der Velde and Martin Perez-Guevara for their continued support of the MIIND project.

Conflict of Interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. (1996). The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.* 22, 469–483. doi: 10.1145/235815.235821

Bogacz, R., Brown, E., Moehlis, J., Holmes, P., and Cohen, J. D. (2006). The physics of optimal decision making: a formal analysis of models of performance in two-alternative forced-choice tasks. *Psychol. Rev.* 113, 700. doi: 10.1037/0033-295X.113.4.700

Booth, V., and Rinzel, J. (1995). A minimal, compartmental model for a dendritic origin of bistability of motoneuron firing patterns. *J. Comput. Neurosci.* 2, 299–312. doi: 10.1007/BF00961442

Brette, R., and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.* 94, 3637–3642. doi: 10.1152/jn.00686.2005

Brown, K. Q. (1979). Voronoi diagrams from convex hulls. *Inform. Process. Lett.* 9, 223–228. doi: 10.1016/0020-0190(79)90074-7

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 183–208. doi: 10.1023/A:1008925309027

Chizhov, A., Campillo, F., Desroches, M., Guillamon, A., and Rodrigues, S. (2019). Conductance-based refractory density approach for a population of bursting neurons. *Bull. Math. Biol.* 81, 4124–4143. doi: 10.1007/s11538-019-00643-8

Chizhov, A. V., and Graham, L. J. (2007). Population model of hippocampal pyramidal neurons, linking a refractory density approach to conductance-based neurons. *Phys. Rev. E* 75, 011924. doi: 10.1103/PhysRevE.75.011924

de Kamps, M. (2006). "An analytic solution of the reentrant poisson master equation and its application in the simulation of large groups of spiking neurons," in *The 2006 IEEE International Joint Conference on Neural Network Proceedings* (Vancouver, BC: IEEE), 102–109. doi: 10.1109/IJCNN.2006.246666

De Kamps, M., Lepperød, M., and Lai, Y. M. (2019). Computational geometry for modeling neural populations: from visualization to simulation. *PLoS Comput. Biol.* 15, e1006729. doi: 10.1371/journal.pcbi.1006729

FitzHugh, R. (1961). Impulses and physiological states in theoretical models of nerve membrane. *Biophys. J.* 1, 445. doi: 10.1016/S0006-3495(61)86902-6

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (neural simulation tool). *Scholarpedia* 2, 1430. doi: 10.4249/scholarpedia.1430

Haiman, M. (1991). A simple and relatively efficient triangulation of the n-cube. *Discrete Comput. Geometry* 6, 287–289. doi: 10.1007/BF02574690

Hodgkin, A. L., and Huxley, A. F. (1952). The components of membrane conductance in the giant axon of loligo. *J. Physiol.* 116, 473. doi: 10.1113/jphysiol.1952.sp004718

Izhikevich, E. M. (2007). *Dynamical Systems in Neuroscience*. Cambridge, MA: MIT Press. doi: 10.7551/mitpress/2526.001.0001

Jansen, B. H., and Rit, V. G. (1995). Electroencephalogram and visual evoked potential generation in a mathematical model of coupled cortical columns. *Biol. Cybernet.* 73, 357–366. doi: 10.1007/BF00199471

Johannesma, P. I. M. (1969). *Stochastic neural activity: a theoretical investigation* (Ph.D. thesis). Faculteit der Wiskunde en Natuurwetenschappen, Nijmegen, Netherlands.

Knight, B. W. (1972). Dynamics of encoding in a population of neurons. *J. Gen. Physiol.* 59, 734–766. doi: 10.1085/jgp.59.6.734

Knight, J. C., Komissarov, A., and Nowotny, T. (2021). PyGeNN: a Python library for gpu-enhanced neural networks. *Front. Neuroinform.* 15, 659005. doi: 10.3389/fninf.2021.659005

Ly, C., and Tranchina, D. (2009). Spike train statistics and dynamics with synaptic input from any renewal process: a population density approach. *Neural Comput.* 21, 360–396. doi: 10.1162/neco.2008.03-08-743

Mattia, M., and Del Giudice, P. (2002). Population dynamics of interacting spiking neurons. *Phys. Rev. E* 66, 051917. doi: 10.1103/PhysRevE.66.051917

Nagumo, J., Arimoto, S., and Yoshizawa, S. (1962). An active pulse transmission line simulating nerve axon. *Proc. IRE* 50, 2061–2070. doi: 10.1109/JRPROC.1962.288235

Naud, R., and Gerstner, W. (2012). Coding and decoding with adapting neurons: a population approach to the peri-stimulus time histogram. *PLoS Comput. Biol.* 8, e1002711. doi: 10.1371/journal.pcbi.1002711

Nykamp, D. Q., and Tranchina, D. (2000). A population density approach that facilitates large-scale modeling of neural networks: analysis and an application to orientation tuning. *J. Comput. Neurosci.* 8, 19–50. doi: 10.1023/A:1008912914816

Omurtag, A., Knight, B. W., and Sirovich, L. (2000). On the simulation of large populations of neurons. *J. Comput. Neurosci.* 8, 51–63. doi: 10.1023/A:1008964915724

Osborne, H., Lai, Y. M., Lepperød, M. E., Sichau, D., Deutz, L., and De Kamps, M. (2021). MIIND: a model-agnostic simulator of neural populations. *Front. Neuroinform.* 15, 614881. doi: 10.3389/fninf.2021.614881

Potjans, T. C., and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Rangan, A. V., and Cai, D. (2007). Fast numerical methods for simulating large-scale integrate-and-fire neuronal networks. *J. Comput. Neurosci.* 22, 81–100. doi: 10.1007/s10827-006-8526-7

Schmutz, V., Gerstner, W., and Schwalger, T. (2020). Mesoscopic population equations for spiking neural networks with synaptic short-term plasticity. *J. Math. Neurosci.* 10, 1–32. doi: 10.1186/s13408-020-00082-z

Schwalger, T., and Chizhov, A. V. (2019). Mind the last spike-firing rate models for mesoscopic populations of spiking neurons. *Curr. Opin. Neurobiol.* 58, 155–166. doi: 10.1016/j.conb.2019.08.003

Schwalger, T., Deger, M., and Gerstner, W. (2017). Towards a theory of cortical columns: from spiking neurons to interacting neural populations of finite size. *PLoS Comput. Biol.* 13, e1005507. doi: 10.1371/journal.pcbi.1005507

Sirovich, L., Everson, R., Kaplan, E., Knight, B., O'Brien, E., and Orbach, D. (1996). Modeling the functional organization of the visual cortex. *Phys. D Nonlinear Phenomena* 96, 355–366. doi: 10.1016/0167-2789(96)00033-4

Sukenik, N., Vinogradov, O., Weinreb, E., Segal, M., Levina, A., and Moses, E. (2021). Neuronal circuits overcome imbalance in excitation and inhibition by adjusting connection numbers. *Proc. Natl. Acad. Sci. U.S.A*. 118:e2018459118. doi: 10.1073/pnas.2018459118

Tsodyks, M. V., and Markram, H. (1997). The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. *Proc. Natl. Acad. Sci. U.S.A*. 94, 719–723. doi: 10.1073/pnas.94.2.719

Wilson, H. R., and Cowan, J. D. (1972). Excitatory and inhibitory interactions in localized populations of model neurons. *Biophys. J*. 12, 1–24. doi: 10.1016/S0006-3495(72)86068-5

Yavuz, E., Turner, J., and Nowotny, T. (2016). Genn: a code generation framework for accelerated brain simulations. *Sci. Rep*. 6, 1–14. doi: 10.1038/srep18854

York, G. J. R., Osborne, H., Sriya, P., Astill, S., de Kamps, M., and Chakrabarty, S. (2022). The effect of limb position on a static knee extension task can be explained with a simple spinal cord circuit model. *J. Neurophysiol*. 127, 173–187. doi: 10.1152/jn.00208.2021

frontiers | Frontiers in Neuroinformatics

TYPE Original Research
PUBLISHED 22 September 2022
DOI 10.3389/fninf.2022.884245

OPEN ACCESS

EDITED BY
James Courtney Knight,
University of Sussex, United Kingdom

REVIEWED BY
Srikanth Ramaswamy,
Newcastle University, United Kingdom
Mauricio Girardi-Schappo,
University of Ottawa, Canada

*CORRESPONDENCE
Fernando S. Borges
fernandodasilvaborges@gmail.com
Salvador Dura-Bernal
salvador.dura-bernal@downstate.edu

RECEIVED 25 February 2022
ACCEPTED 27 July 2022
PUBLISHED 22 September 2022

CITATION
Borges FS, Moreira JVS, Takarabe LM, Lytton WW and Dura-Bernal S (2022) Large-scale biophysically detailed model of somatosensory thalamocortical circuits in NetPyNE. *Front. Neuroinform.* 16:884245. doi: 10.3389/fninf.2022.884245

Large-scale biophysically detailed model of somatosensory thalamocortical circuits in NetPyNE

Fernando S. Borges[1,2]*, Joao V. S. Moreira[1], Lavinia M. Takarabe[2], William W. Lytton[1,3,4] and Salvador Dura-Bernal[1,5]*

[1]Department of Physiology and Pharmacology, State University of New York Downstate Health Sciences University, Brooklyn, NY, United States, [2]Center for Mathematics, Computation, and Cognition, Federal University of ABC, São Paulo, Brazil, [3]Department of Neurology, Kings County Hospital Center, Brooklyn, NY, United States, [4]Aligning Science Across Parkinson's (ASAP) Collaborative Research Network, Chevy Chase, MD, United States, [5]Nathan Kline Institute for Psychiatric Research, Orangeburg, NY, United States

The primary somatosensory cortex (S1) of mammals is critically important in the perception of touch and related sensorimotor behaviors. In 2015, the Blue Brain Project (BBP) developed a groundbreaking rat S1 microcircuit simulation with over 31,000 neurons with 207 morpho-electrical neuron types, and 37 million synapses, incorporating anatomical and physiological information from a wide range of experimental studies. We have implemented this highly detailed and complex S1 model in NetPyNE, using the data available in the Neocortical Microcircuit Collaboration Portal. NetPyNE provides a Python high-level interface to NEURON and allows defining complicated multiscale models using an intuitive declarative standardized language. It also facilitates running parallel simulations, automates the optimization and exploration of parameters using supercomputers, and provides a wide range of built-in analysis functions. This will make the S1 model more accessible and simpler to scale, modify and extend in order to explore research questions or interconnect to other existing models. Despite some implementation differences, the NetPyNE model preserved the original cell morphologies, electrophysiological responses and spatial distribution for all 207 cell types; and the connectivity properties of all 1941 pathways, including synaptic dynamics and short-term plasticity (STP). The NetPyNE S1 simulations produced reasonable physiological firing rates and activity patterns across all populations. When STP was included, the network generated a 1 Hz oscillation comparable to the original model *in vitro*-like state. By then reducing the extracellular calcium concentration, the model reproduced the original S1 *in vivo*-like states with asynchronous activity. These results validate the original study using a new modeling tool. Simulated local field potentials (LFPs) exhibited realistic oscillatory patterns and features, including distance- and frequency-dependent attenuation. The model was extended by adding thalamic circuits, including 6 distinct thalamic populations with intrathalamic, thalamocortical (TC) and corticothalamic connectivity derived from experimental data. The thalamic model reproduced single known cell and circuit-level dynamics, including burst and tonic firing modes and oscillatory patterns, providing a more realistic input to cortex and enabling study of TC interactions. Overall, our work provides a widely accessible, data-driven and biophysically-detailed model of the somatosensory TC circuits that can be employed as a community tool for researchers to study neural dynamics, function and disease.

KEYWORDS

somatosensory cortex, thalamocortical circuits, large-scale model, biophysical, cortical, multiscale

Introduction

The primary somatosensory cortex (S1) of mammals is critically important in the perception of touch and works closely with other sensory and motor cortical regions in permitting coordinated activity with tasks involving grasp (Bosman et al., 2011; Petrof et al., 2015; Barthas and Kwan, 2017). Moreover, the communication of these cortical areas with the thalamus is crucial for maintaining functions, such as sleep and wakefulness, considering that the thalamocortical (TC) circuit is essential for cerebral rhythmic activity (O'Reilly et al., 2021). A greater understanding of S1 cortical circuits will help us gain insights into neural coding and be of assistance in determining how disease states such as schizophrenia, epilepsy and Parkinson's disease lead to sensory deficits or uncoordinated movement (Vázquez et al., 2013; Petrof et al., 2015; Azarfar et al., 2018; Peña-Rangel et al., 2021).

There exists an impressive, highly detailed model of rat S1 developed by the Blue Brain Project (BBP) (Markram et al., 2015), incorporating anatomical and physiological information from a wide range of experimental studies. This groundbreaking model includes over 31,000 neurons of 55 layer-specific morphological and 207 morpho-electrical neuron subtypes, and 37 million synapses capturing layer- and cell type-specific connectivity patterns and synaptic dynamics. Simulation results matched *in vitro* and *in vivo* experimental findings, and the model has been used over the years to reproduce additional experimental results and generate predictions of the dynamics and function of cortical microcircuits (Reimann et al., 2015, 2017a,b; Gal et al., 2017; Hagen et al., 2018; Amsalem et al., 2020). Although the BBP S1 model is state-of-the-art, certain constraints limit its reproducibility and use by the community, as well as its extension or modification to connect to other regions or update model features. The size and complexity of any model of this scope is daunting. Due to its scale and complexity, the original model must be run and analyzed on large High Performance Computing platforms (HPCs), which are not available to many users. Although the model is simulated using NEURON (Carnevale and Hines, 2006; Migliore et al., 2006) a widely used platform within the computational neuroscience community, it also requires other custom libraries specifically designed to facilitate this workflow. These libraries are used to build, manage simulations and analyze the model. However, not all of these libraries and workflows are publicly available (Markram et al., 2015), making it somewhat difficult to modify the code, and scale or simplify the model for simulation on smaller computers, overall reducing its accessibility and reproducibility (McDougal et al., 2016).

Here we implemented the original BBP S1 model in NetPyNE (Dura-Bernal et al., 2019) in order to make it more accessible and simpler to scale, modify and extend. NetPyNE is a python package that provides a high-level interface to the NEURON simulator, and allows the definition of complex multiscale models using an intuitive declarative standardized language. NetPyNE translates these specifications into a NEURON model, facilitates running parallel simulations, automates the optimization and exploration of parameters using supercomputers, and provides a wide range of built-in analysis functions.

Conversion to NetPyNE also makes it easier to connect to previous models developed within the platform, such as our primary motor cortex model (Sivagnanam et al., 2020; Dura-Bernal et al., 2022b), and models implemented in other tools (e.g., NEST) by exporting to the NeuroML or SONATA standard formats. In prior work, we ported a classic model of generic sensory cortical circuits (Potjans and Diesmann, 2014) to our NetPyNE platform (Romaro et al., 2021) in order to make it both more scalable and facilitate modification of cell models and network parameters. The original model used integrate-and-fire neurons and we replaced these with more complex multi-compartment neuron models.

Although we have primarily focused on simplifying the network description, we have also made the model more complex, and more complete, by adding the associated somatosensory thalamic circuits and bidirectional connectivity with cortex to allow interplay of these two highly coordinated areas (Meyer et al., 2010). The deepening of knowledge about the cortico-thalamo-cortical loop (Shepherd and Yamawaki, 2021) should contribute to investigations on rhythmic dysfunctions, such as epilepsy and schizophrenia. But in contrast to cortical microcircuitry, few detailed models exist for the thalamus (Hill and Tononi, 2005; Izhikevich and Edelman, 2008; Murray and Anticevic, 2017; Iavarone et al., 2019).

In this study we present a NetPyNE implementation of the BBP S1 model, capturing most of the original single-cell physiology and morphology, synaptic mechanisms, connectivity and basic simulation results. With the addition of detailed thalamic circuits, we extend the results to show synchronous activity across cortical and thalamic populations, and open the door to new investigations on corticothalamic dynamics. The model is able to port readily across machines and can utilize a fast and efficient implementation on CPUs and GPUs using CoreNEURON. This extension allows the original BBP S1 model to be readily available to be used by the wider community to study a wide range of research questions.

Materials and methods

Individual neuron models

Cell reconstructions were based on the compartmental model Hodgkin-Huxley formalism, with membrane properties represented as components of an electric circuit, and ionic channels modeled as variable conductances. To port the somatosensory microcircuit model in NetPyNE (Dura-Bernal et al., 2019), we recreated the single neuron models using cell files from the Neocortical Microcircuit Collaboration (NMCP)[1] (Ramaswamy et al., 2015). The full dataset comprises 207 morpho-electrical (me) cell types, with 5 examples for each, totaling 1,035 cell models, each stored with morphology file, descriptions of ion channels, and a NEURON HOC template to instantiate the cell, which can be imported directly to NetPyNE (**Figure 1**). Neuron morphologies from the BBP S1 model (specifically, L1_DLAC, L4_DBC, L23_PC, and L6_TPC_L4) imported into NetPyNE were visualized using the NetPyNE GUI (**Figure 1A**). The full name of the 207 cell types as well as the corresponding acronym can be found in **Supplementary Table 1** in the **Supplementary material**.

Benchmark testing validated physiological responses (**Figures 1B–E**) at 3 current clamp amplitudes (120%, 130%, and 140% of threshold; only 120% shown). Slight differences were observed in the cell types with a stochastic version of the K^+ channel mechanism (StochKv; **Figure 1D**) where we used a deterministic version of the channel from OpenSourceBrain (Gleeson et al., 2019). The StochKv NMODL (.mod) mechanism required additional code outside of NetPyNE in order to update its state, and the inclusion of stochastic variables in each section of the cells significantly increased the simulation time. In order to understand the effect of StochKv on cell response, we applied a current clamp (0.1 nA, 2s) to the soma of each of the 1,035 cells, and used the Electrophys Feature Extraction Library (eFEL)[2] to compare BBP and NetPyNE mean firing rate (**Figure 1F**) and time to first spike (**Figure 1G**) for those with and without the StochKv channel. As expected, the variability for cells with the StochKv channel in the original model was pronounced. Although present in 54/207 me-types, the StochKv channels are only in 3.63% of all cells. Within each m-type (morphology-type) those with StochKv also correspond to a minority of e-types (electrical) types; for example, only 32% of L4_DBC cells have e-type bIR (with StochKv channels). Given the small proportion of cells with StochKv channels (3.63%), the NetPyNE mean firing rates per m-type population closely matched those of the original BBP model (**Figure 1J**). Furthermore, the deterministic version of StochKv preserved irregular cell spiking patterns (CV_{BBP} = 0.25 ± 0.16; $CV_{NetPyNE}$ = 0.16 ± 0.13; where CV is the inter spike interval coefficient of variation; see **Supplementary Figure 1**) as well as the neural firing rate (FR_{BBP} = 30.24 ± 24.33, $FR_{NetPyNE}$ = 28.67 ± 21.20) in the current-clamp simulation with amplitude 0.1 nA during 2 s. For stimulation amplitude 0.8 nA, the CV (BBP = 0.14 ± 0.17; NetPyNE = 0.15 ± 0.30) and FR (BBP = 115.58 ± 70.20, NetPyNE = 110.15 ± 66.58) were similar in both model implementations (**Supplementary Figure 1**).

The NetPyNE implementation perfectly reproduced the original neuronal intrinsic dynamics since all model parameters were directly imported from the original HOC files, the same NMDOL files were used (except StochKv), and the underlying simulation engine was NEURON in both cases (see **Figures 1B–E**). To validate this, we simulated somatodendritic backpropagating action potentials (**Figure 1H**) and dendrosomatic postsynaptic potentials (**Figure 1I**) in an example L5_TTPC cell. Results were identical in the NetPyNE implementation and the original BBP cell models. To model dendrosomatic postsynaptic potentials (PSPs), we added excitatory connections with 5 and 10 synapses, and an inhibitory connection with 20 synapses, to the L5_TTPC neuron. Additionally, we provided the same three subthreshold

[1] https://bbp.epfl.ch/nmc-portal

[2] https://github.com/BlueBrain/eFEL

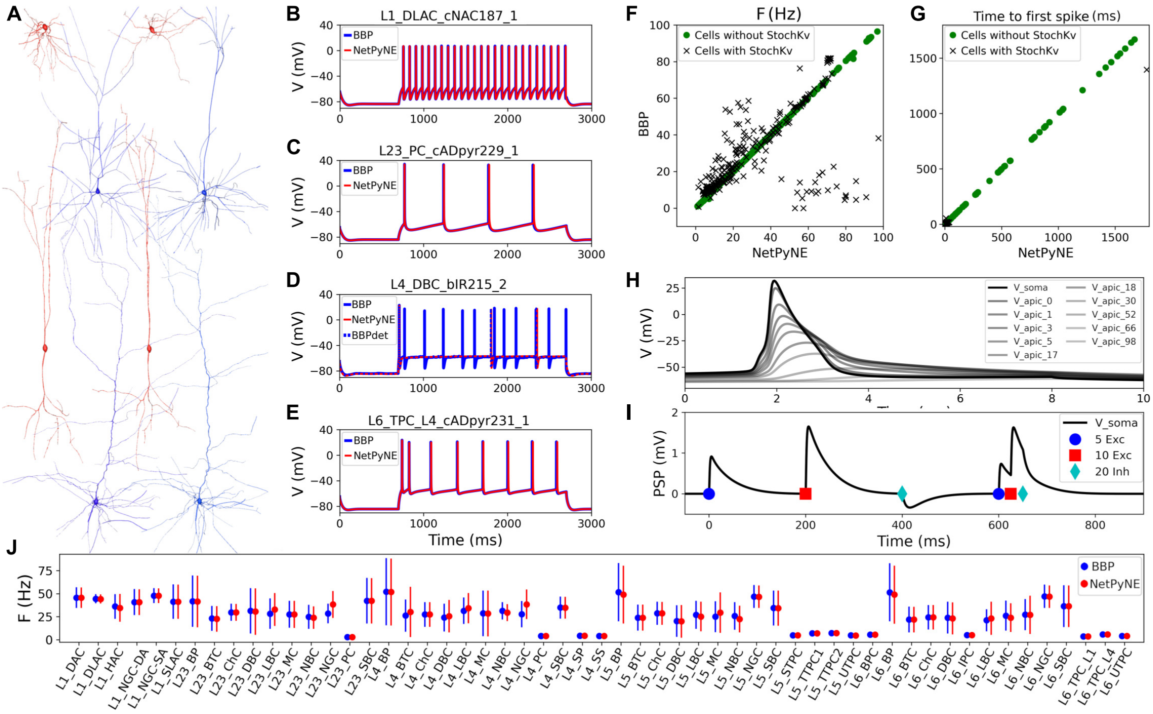


FIGURE 1

Reproduction and validation of BBP S1 cell types in NetPyNE. **(A)** 3D reconstructions of 4 pairs of m-type example neurons visualized using the NetPyNE graphical user interface: inhibitory cells L1_DLAC and L4_DBC (red), and excitatory cells L23_PC and L6_TPC_L4 (blue). **(B–E)** Somatic membrane potential of the neurons in **(A)** under current clamp with amplitude 120% of the neuron firing threshold. NetPyNE results (red) compared to the original BBP model results (blue). For L4_DBC_bIR cells **(D)** we used a deterministic version of the BBP stochastic potassium channel (StochKv) resulting in divergent results; using same deterministic channel in BBP (BBPdet, blue dotted line) restores the match to NetPyNE results. **(F,G)** Comparison of BBP and NetPyNE firing rate and time to first spike in response to current-clamp with amplitude 0.1 nA during 2 s for each cell type. Only some cell types with the StochKv show differences. **(H)** Backpropagating action potential in a L5_TTPC cell. **(I)** Post synaptic potentials (PSPs) of one dendritic connection with 5 excitatory (blue circles), 10 excitatory (red squares), and 20 inhibitory (cyan diamonds) synapses. **(J)** Comparison BBP and NetPyNE mean firing rate for all m-type populations. Due to the small number of cells with StochKv (3.6%), NetPyNE population firing rates closely match those of BBP.

inputs within a short time interval, to demonstrate temporal integration of PSPs (Figure 1I).

Distribution and connectivity of cortical populations

Rather than instantiating the connectivity from a list of individual synapses based on anatomical overlap of neuronal arbors (Reimann et al., 2015), we created our S1 port using probability rules for both neuron distribution and connections. The network consisted of 31,346 cells in a cylindrical volume 2,082 μm height and radius of 210 μm as in the original model (Figure 2). Each population was randomly distributed within its specific layer (L1, L2/3, L4, L5, or L6). The number of cells in each one of 207 me-types was taken from the NMCP (Ramaswamy et al., 2015) the minicolumn data available was not used to distribute cells. A 2D representation of the cell distribution within the cylindrical volume is shown in Figure 2A, with layer thicknesses (in μm) for L1, L23, L4, L5, and L6 set to 165, 502, 190, 525, and 700, respectively. We used the S1 connectome (Gal et al., 2017) from NMCP, following the approach in Reimann et al. (2017b): 7 stochastic instances of a model microcircuit based on averaged measurements of neuron densities were used to calculate distance-dependent probabilities of connection. In each microcircuit instance, we calculated the connection probability for each pair of neurons based on the 2D somatic distance (horizontal XZ-plane) for each of the 1,941 pathways. To estimate the distance-dependent probability, we calculated the probability in evenly spaced intervals starting at 25 ± 25 μm, in 50 μm intervals, up to 375 ± 25 μm. Next, we calculated the mean probability across the 7 microcircuits in evenly spaced intervals and used the mean values to fit the connection probability rules. We evaluated multiple functions for each pathway and selected the one that provided the best fit to the data. Figures 2B–D shows how this approach was used to calculate the connection probability of 3 example projections: data from the 7 microcircuit instances (mcs; cyan circles) was averaged across microcircuits (green diamonds) and fitted to either a single exponential

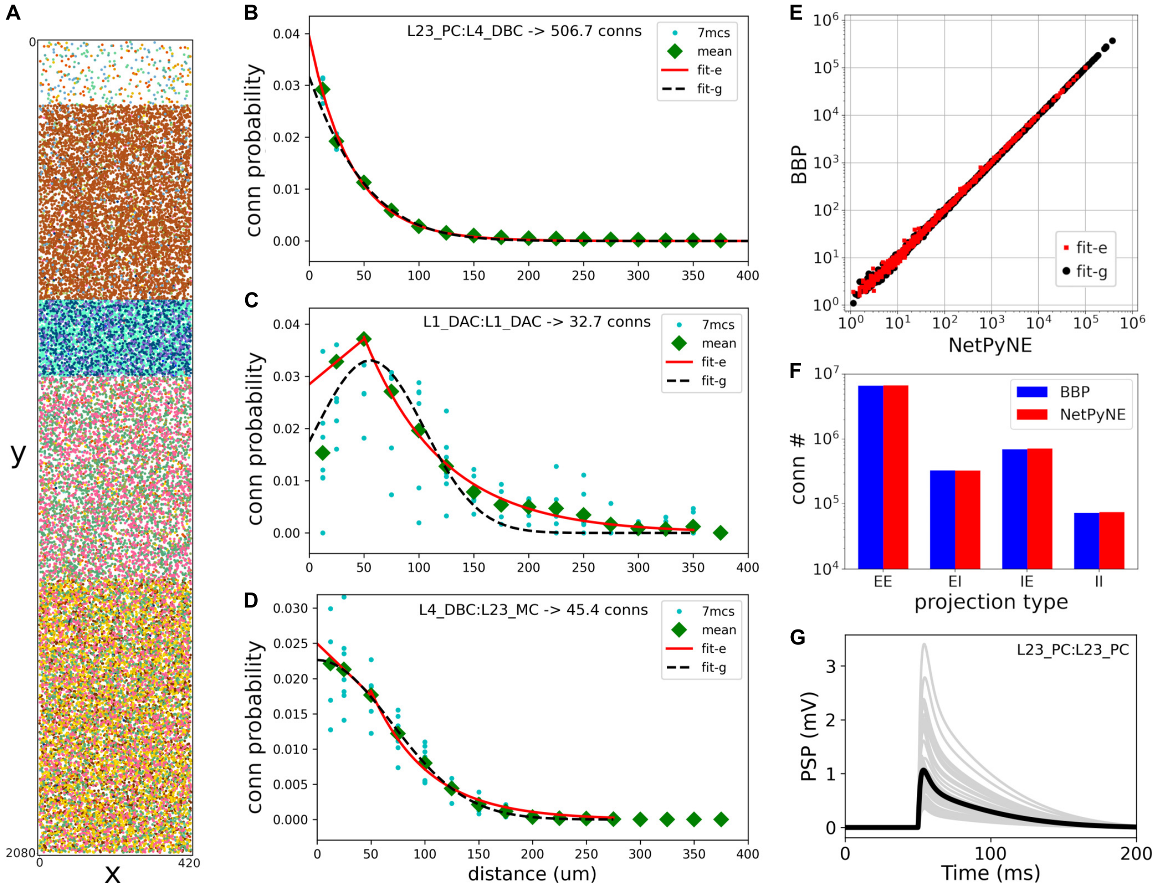


FIGURE 2
Reproduction and validation of BBP S1 neuron distribution and connectivity in NetPyNE. **(A)** 2D representation of the location of 31,346 cells in a cylinder with 2,082 µm height and 210 µm radius, with each subtype (different colors) randomly distributed within its layer (L1, L2/3, L4, L5, or L6). **(B–D)** Probability of connection as a function of neuron pairwise 2D distance for three example pathways, each with a different best fit function: a single exponential (**B**, red line), exponential with a linear saturation rule (**C**, red line), and single gaussian fit (**D**, black dashed line). Cyan circles represent data from 7 microcircuit instances, and green diamonds represent the mean across the 7 instances. **(E,F)** Comparison of the number of connections between NetPyNE and BBP for each of the 1,941 pathways **(E)** and 4 projection types (p-types) **(F)**. **(G)** Postsynaptic potential (PSP) generated by connection between L23_PC neurons (Table 1, #18) simulated in NetPyNE; mean PSP trace (black line) across 20 PSP random instances (gray lines).

(Figure 2B, red line); an exponential with a linear saturation rule (Figure 2C); or a single gaussian (Figure 2D, dashed line). Because the original S1 model shows high variability in the number of synapses per connection, we calculated the mean values for each pathway and used it as a parameter in our model. The result is a representative reconstruction of the S1 column connectivity in NetPyNE, with approximately 27.6 million excitatory synapses and 9.6 million inhibitory synapses.

Using the fitted connectivity rules, we reconstructed an entire S1 column in NetPyNE and compared the two versions using the mean number of connections. To avoid overfitting, we generated 7 different instances using different connectivity seeds for both the NetPyNE and BBP models. The number of connections was similar in both models for each of the 1,941 pathways (Figure 2E) and for each of the four projection types (p-type) (EE, EI, IE, II) (Figure 2F).

Synaptic physiology

The original BBP S1 model included detailed synaptic properties (conductances, post-synaptic potentials (PSP), latencies, rise and decay times, failures, release probabilities, etc.) recapitulating published experimental data. Short-term dynamics were used to classify synapses into the following types (s-types): inhibitory facilitating (I1), inhibitory depressing (I2), inhibitory pseudo-linear (I3), excitatory facilitating (E1), excitatory depressing (E2), and excitatory pseudo-linear (E3). A set of rules were then derived from experimental data to assign an s-type to each broad class of connections. Based on the NMCP data, there were 29 classes of connections as determined by the combination of pre- and post-synaptic me-types. The synaptic properties, s-type and p-type for each class of connections are summarized in Table 1.

TABLE 1 Synaptic properties, s-type, p-type, and rules for each class of connections implemented in NetPyNE.

#	BBP id	s-Type	p-Type	g_{syn} (nS)	τ_{decay} (ms)	U	D (ms)	F (ms)	Pre- and post-syn cell type rules
0	0	I1	II	0.83 ± 0.55	10.40 ± 6.10	0.16 ± 0.100	45 ± 21	376 ± 253	L6:L6_(DBC-LBC-NBC-SBC)
1	3	I1	IE	0.91 ± 0.61	10.40 ± 6.10	0.16 ± 0.100	45 ± 21	376 ± 253	SBC_cAC:Exc or L6_(NBC-LBC):L6_BPC
2	13	I1	IE	0.75 ± 0.32	10.40 ± 6.10	0.41 ± 0.212	162 ± 69	690 ± 5	L6_MC:L6_IPC
3	1	I2	II	0.83 ± 0.55	8.30 ± 2.20	0.25 ± 0.130	706 ± 405	21 ± 9	L1:Excitatory or Inhibitory:Inhibitory
4	4	I2	IE	0.91 ± 0.61	8.30 ± 2.20	0.25 ± 0.130	706 ± 405	21 ± 9	SBC_dNAC:Excitatory
5	8	I2	IE	0.75 ± 0.32	8.30 ± 2.20	0.25 ± 0.130	706 ± 405	21 ± 9	BTC-DBC-BP:Excitatory
6	9	I2	IE	0.75 ± 0.32	8.30 ± 2.20	0.30 ± 0.080	1,250 ± 520	2 ± 4	MC:Excitatory
7	10	I2	IE	0.91 ± 0.61	8.30 ± 2.20	0.14 ± 0.050	875 ± 285	22 ± 5	LBC-NBC_(bAC cAC bNAC dNAC):Excitatory
8	12	I2	IE	2.97 ± 0.95	8.30 ± 2.20	0.25 ± 0.130	706 ± 405	21 ± 9	Chc:Excitatory
9	5	I3	IE	0.91 ± 0.61	6.44 ± 1.70	0.32 ± 0.140	144 ± 80	62 ± 31	SBC_bNAC or LBC-NBC_(cNAC dSTUT cSTUT bSTUT):Excitatory
10	11	I3	IE	0.83 ± 0.55	36.55 ± 0.71	0.25 ± 0.130	706 ± 405	21 ± 9	NGC:Excitatory
11	114	E1	EI	0.43 ± 0.28	1.74 ± 0.18	0.02 ± 0.001	194 ± 10	507 ± 20	Exc:(BP_cAC DBC_cAC BTC_cAC)
12	115	E1	EI	0.72 ± 0.50	1.74 ± 0.18	0.02 ± 0.001	194 ± 10	507 ± 20	Exc:(NBC-LBC)_(cAC cIR bAC bIR cNAC)
13	132	E1	EI	0.72 ± 0.50	1.74 ± 0.18	0.01 ± 0.001	242 ± 15	563 ± 32	L6_TPC_L:L6_(DBC-LBC-NBC-SBC)
14	133	E1	EI	0.11 ± 0.08	1.74 ± 0.18	0.09 ± 0.120	138 ± 211	670 ± 830	Excitatory:MC
15	116	E2	EE	0.72 ± 0.50	1.74 ± 0.18	0.50 ± 0.020	671 ± 17	17 ± 5	Excitatory:Excitatory
16	117	E2	EI	0.43 ± 0.28	1.74 ± 0.18	0.50 ± 0.020	671 ± 17	17 ± 5	Excitatory:[L1-BP_(cNAC bNAC)-DBC_bAC-BTC_(bAC cNAC bIR)]
17	118	E2	EI	0.72 ± 0.50	1.74 ± 0.18	0.50 ± 0.020	671 ± 17	17 ± 5	Excitatory:SBC-ChC
18	119	E2	EE	0.68 ± 0.46	1.74 ± 0.18	0.46 ± 0.260	671 ± 17	17 ± 5	L23_PC:L23_PC
19	120	E2	EE	0.68 ± 0.46	1.74 ± 0.18	0.86 ± 0.049	671 ± 17	17 ± 5	L4_Excitatory:L4_Excitatory
20	121	E2	EE	0.19 ± 0.12	1.74 ± 0.18	0.79 ± 0.040	671 ± 17	17 ± 5	L4_SS:L23_PC
21	122	E2	EE	0.80 ± 0.53	1.74 ± 0.18	0.39 ± 0.030	671 ± 17	17 ± 5	L5_STPC:L5_STPC
22	123	E2	EE	1.50 ± 1.05	1.74 ± 0.18	0.50 ± 0.020	671 ± 17	17 ± 5	L5_TTPC:L5_TTPC
23	127	E2	EE	0.80 ± 0.53	1.74 ± 0.18	0.39 ± 0.134	780 ± 54	51 ± 36	L6_IPC:L6_IPC
24	131	E2	EI	0.72 ± 0.50	1.74 ± 0.18	0.58 ± 0.070	240 ± 43	71 ± 47	L6_IPC:L6_(DBC-LBC-NBC-SBC)
25	134	E2	EI	0.72 ± 0.50	1.74 ± 0.18	0.72 ± 0.065	227 ± 38	14 ± 12	Exc:(NBC-LBC)_(bSTUT dNAC bNAC cSTUT)
26	126	E3	EE	0.80 ± 0.53	1.74 ± 0.18	0.21 ± 0.032	460 ± 53	230 ± 69	L6_TPC_L:L6_TPC_L
27	128	E3	EE	0.80 ± 0.53	1.74 ± 0.18	0.27 ± 0.033	559 ± 238	200 ± 92	L6_IPC:L6_BPC
28	129	E3	EE	0.80 ± 0.53	1.74 ± 0.18	0.22 ± 0.053	535 ± 134	116 ± 81	L6_IPC:L6_TPC_L

s-type, type of short-term dynamics; p-type, type of projection; g_{syn}, peak conductance (ms); τ_{decay}, decay time (ms); U, neurotransmitter release probability; D, time constant for recovery from depression (ms); F, time constant for recovery from facilitation (ms). Values indicate mean ± standard deviation.

The dual-exponential conductance model with rise time (τ_{rise}) 0.2 ms was used for all synapses. Moreover, synaptic properties included the kinetic parameters: peak conductance (g_{syn}; in nS) and decay time (τ_{decay}; in ms); and dynamic parameters: neurotransmitter release probability (U), time constant for recovery from depression (D; in ms) and time constant for recovery from facilitation (F; in ms). The NetPyNE implementation reproduces the original PSP amplitudes from Markram et al. (2015). An example of PSPs for a connection between L23_PC neurons (Table 1, #18) simulated in NetPyNE is shown in **Figure 2G**. The mean PSP peak amplitude across 20 PSPs (with different randomization seeds) was 1.0 mV, which matches the value obtained in Markram et al. (2015). We also included a compact description of the rules to determine what connections belong to each class, based on the pre- and postsynaptic cell types (**Table 1**). For clarity, we rearranged the classes of connections by s-types in the sequence I1, I2, I3, E1, E2, and E3 (from 0 to 28), and included the original BBP class label for reference. The parameters D and F correspond to the synapses with short term plasticity (STP), which could be optionally added to recurrent S1 connections, and connections from thalamus to S1.

The s-types for each class of connections and for each of the 1,941 pathways are color-coded and illustrated in **Figure 3A**. Since pathways depend on m-types but connection classes depend on me-types (each m-type includes multiple me-types), it is possible to have multiple s-types for the same pathway; in those cases we simply labeled it as either I2 or E2. To implement the dynamics of each s-type in NetPyNE we used a deterministic version of the dual-exponential synaptic model (Fuhrmann et al., 2002; Hennig, 2013). Example simulations of the PSP for the different s-types are shown in **Figure 3B**. For each example, we ran 20 simulations with 5 different post-synaptic cells of the same me-type and 4 random synaptic distributions. Pre- and post-synaptic neurons of specific me-types were selected to illustrate each of the six s-types (see **Figure 3**).

Extending the model to include thalamic populations and connectivity

We extended the model to include somatosensory thalamic populations with cell type-specific dynamics, intra-thalamic connectivity and bidirectional projections with cortex. In the original model, thalamic inputs were modeled as spike generators that only provided feedforward inputs to S1. Our somatosensory thalamus model is composed of the excitatory ventral posterolateral (VPL), ventral posteromedial (VPM) and the posteromedial (POm) nuclei, and the inhibitory reticular nucleus (RTN). We used single compartment cell models with dynamics tuned to reproduce previous studies on the interaction between the thalamic relay and reticular cells (Destexhe et al., 1996a), but adjusted to work in large-scale networks (Moreira et al., 2021). The thalamic circuit architecture consisted of six stacked populations as a rough approximation of the thalamic anatomical layout (**Figure 4A**). The top three were inhibitory populations comprising the outer, middle and inner sectors of the RTN, and spanning a height of 78, 78, and 156 μm, respectively. Below these were the three excitatory populations, VPL, VPM, and POm, with heights of 156, 156, and 312 μm, respectively. The horizontal dimensions (XZ-plane) for all populations were 420 μm × 420 μm. Cells were randomly distributed across each nuclei with the number of cells in each population based on cellular density obtained from the Cell Atlas for the Mouse Brain[3] (Erö et al., 2018). Although POm was larger than VPL and VPM, we reduced its cell density by 50%, resulting in approximately the same population size. This lower density accounts for the proportion of coexisting, but functionally isolated, M1-projecting TC cells present in POm with no projections to S1 (Guo et al., 2020).

The intrathalamic connectivity was based on data of axonal and dendritic footprints for each nucleus. The VPL and VPM are considered first-order nuclei (FO), which means they receive afferent information from peripheral sensory organs (not modeled here) and are interconnected with cortex (Sugitani et al., 1990; Ma, 1991; Luczyńska et al., 2003) and RTN (Lam and Sherman, 2011) in a topological fashion. On the other hand, POm is considered a higher-order (HO) nucleus, so input arrives mainly from the cortex, in this case, from S1 L5 and L6 (Ohno et al., 2012; O'Reilly et al., 2021). The connectivity pattern of HO nuclei has not been properly characterized, but literature reports a decreased level of organization of HO nuclei inputs to RTN (Lam and Sherman, 2011), as it sends projections to S1.

We therefore adopted three connectivity strategies. In the first, neurons from FO nuclei projected to RTN with a column-like topological organization. We implemented this by combining a probability of connection that decreased exponentially with the horizontal distance between the pre- and post-synaptic cells with a decay constant proportional to the footprint radius, and which was truncated to 0 outside of the footprint radius (this denotes the maximum distance of connection in the XZ-plane). The following footprint diameters were derived from experimental data (or estimated in the case of no literature reports) for the different axonal footprints of each thalamic projection: RTN→VPL and RTN→VPM: 64.33 μm (Lam and Sherman, 2007); VPL→RTN: 97.67 μm; and VPM→RTN: 103.57 μm (Lam and Sherman, 2011). The second strategy applies to RTN→RTN connectivity and implements a sector-specific distance-dependent connectivity. More specifically, within each RTN sector, the probability of connection decayed exponentially and was truncated to 0 based on a footprint radius of 264.63 μm (Lam et al., 2006). In strategies one and two, the maximum distance in the Y-plane was set to 10% of the footprint radius, following the disc-like morphology from the axonal projections of the relay cells and the dendritic trees of reticular cells (Murray Sherman and Guillery, 2001; Lam et al., 2006). The third strategy was a divergence rule, with the number of projections from and to HO nuclei having a fixed value and being distributed without spatial constraints. This divergence value was adjusted so that the HO dynamics resembled that of the FO nuclei. This allowed us to replicate a column-like topological organization in FO nuclei using single-compartment cells (Lam and Sherman, 2007), and

[3] https://bbp.epfl.ch/nexus/cell-atlas/

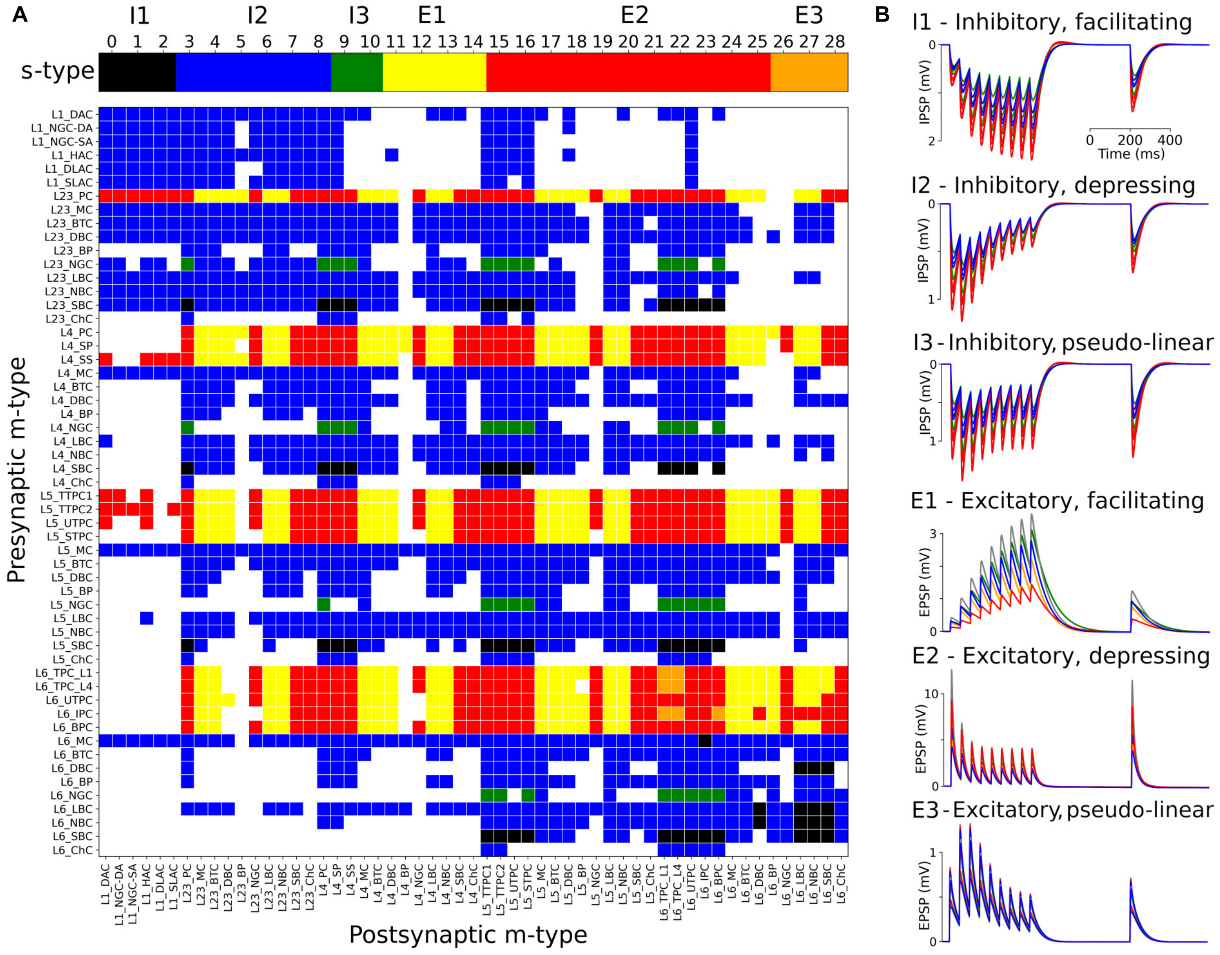


FIGURE 3

Matrix of s-types for each of the 1,941 pathways and simulated PSPs examples for each s-type in NetPyNE. **(A)** Color-coded s-types for each class of connections (top) and for each of the 1,941 pathways (bottom). Note that for pathways with multiple s-types only either I2 or E2 is shown. **(B)** Example simulations of post-synaptic potentials to illustrate each of the six s-types. Each example shows the results of 20 simulations with five different post-synaptic cells (different colors) of the same me-type, and 4 random synaptic distributions. Inhibitory s-types were simulated using pathway L23_SBC:L23_PC, which included different s-types depending on pre-synaptic e-type: I1 for e-type cAC, I2 for e-type dNAC, and I3 for e-type bNAC (as shown in **Table 1**). Excitatory s-types E1 and E2 were simulated using pathway L23_PC:L23_LBC, e-types cAC, and dNAC, respectively; and s-type E3 was simulated using pathway L6_TPC_L4:L6_TPC_L4.

distribute the HO connections to behave in a functionally similar fashion (**Figure 4B**).

All excitatory thalamic nuclei were indirectly interconnected through their RTN projections, which was divided into three sectors, in line with reports of preferred innervation zones by each of the thalamic nuclei (Lam and Sherman, 2011). Synapses within RTN were mediated by GABA$_a$, those from RTN to the excitatory nuclei by a combination of GABAa and GABAb with equal weight, and those from the excitatory nuclei to RTN and cortex by AMPA (Destexhe et al., 1996a). The probability and weight of connections were the targets of parameter optimization. The matrix with the convergence of intra-thalamic connections is shown in **Figure 4E**.

Feedback corticothalamic connectivity originated from S1 m-types L5_TTPC2 and L6_TPC_L4 (O'Reilly et al., 2021). Similar to the topological rules described above, we implemented connectivity with convergence of 30 (i.e., number of pre-synaptic cells projecting to each post-synaptic cell), but only if the horizontal distance between the pre- and post-synaptic neurons was lower than 50.0 μm (**Figure 4**).

TC connectivity from VPL, POm and VPM to S1 was implemented using convergence values estimated from previous studies (Meyer et al., 2010; **Figure 4C**). Convergence values for each of the 55 m-types were calculated based on the weighted average of the populations in each layer. The convergence values for inhibitory populations were multiplied by a scaling factor derived from the original mode (~0.595). This thalamic convergence factor for inhibitory cells was estimated by dividing the IE ratio of VPM thalamic innervation (83/775 = 0.107) by the average IE population ratio (4,779/26,567 = 0.18). The resulting

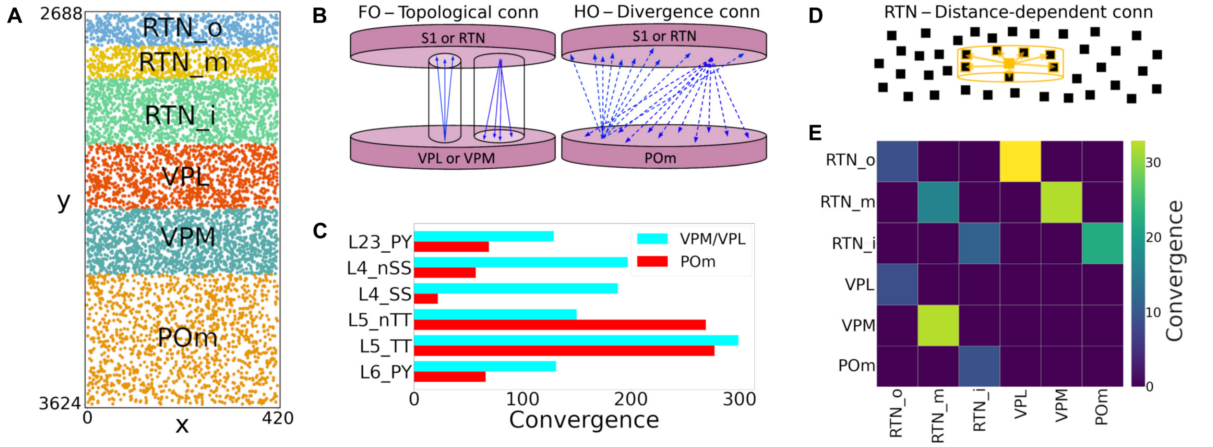


FIGURE 4

NetPyNE model of somatosensory thalamic populations and connectivity extending the original S1 model. **(A)** Distribution of neurons across the six different thalamic populations, roughly mimicking the thalamus anatomy (x and y axes in μm). **(B)** Schematic of bidirectional connectivity between thalamic regions and cortex. Bidirectional connections between S1/RTN and first order (FO) regions VPL and VPM are topological, whereas those with high order (HO) region POm are non-topological and implemented using divergence rules. **(C)** Convergence connectivity between thalamic regions and S1 **(D)** RTN cells have sector-specific distance-dependent connectivity **(E)** convergence connectivity matrix across all thalamic populations.

S1 column received approximately 4.95 M synapses from VPM, 4.95 M from VPL, and 3.1 M from POm. This is consistent with values that can be derived from experimental studies (Meyer et al., 2010), with 4.27 M synapses from VPM and, and 2.66 M from POm. We approximated TC synaptic physiology using the parameters of model #25 in **Table 1**, and using 9 synapses per connection, following the Markram et al. (2015) characterization of TC synapses as excitatory depressing (E2).

Background inputs

Each cell in the S1 circuit received 10 synaptic inputs from Poisson-distributed spike generators (NetStims) to represent the global effect of spontaneous synapses, background, and other noise sources from non-modeled brain regions projecting to S1. These stimuli were randomly distributed remove across all sections. The quantal synaptic conductance was calculated based on the average quantal conductance for excitatory and inhibitory synapses. We tuned the excitatory and inhibitory stimuli rates using grid search parameter exploration to obtain average excitatory firing rates of ∼1 Hz and physiological firing rates for most S1 populations.

Model building

We used the NetPyNE modeling tool (Dura-Bernal et al., 2019) to build, manage simulations, and analyze results of the S1 and thalamic circuit model. NetPyNE employs NEURON (Carnevale and Hines, 2006; Migliore et al., 2006; Lytton et al., 2016) as backend simulation engine, with either the standard or CoreNEURON libraries (Kumbhar et al., 2019). The high-level Python-based declarative language provided by NetPyNE facilitated the development of this highly complex and extensive circuit model. This language enabled us to easily import existing morphological and biophysical parameters of different cell types, and define complex connectivity and stimulation rules. We used NetPyNE to explore and optimize the model parameters through automated submission and managing of simulations on supercomputers. We also employed NetPyNE's built-in analysis functions to plot 2D representations of cell locations, connectivity matrices, voltage traces, raster plots, local field potentials (LFPs), 3D synapses representations, and firing rate statistics. NetPyNE can also be used to export the model into the NeuroML (Gleeson et al., 2010) and SONATA (Dai et al., 2020) standard formats.

Model parameters are based on experimental data and the original model (Markram et al., 2015). Nonetheless, parameter optimization was necessary to ensure the model reproduces experimental measures such as population firing rates and PSP. The parameters optimized for the S1 TC circuit were the background rate for excitatory and inhibitory connections. For the intrathalamic projections, we optimized connection weight (range 0–2 mV), connection probability (range 0–1), y-axis connection radius (1%, 2%, 5%, or 10%) and connectivity divergence of the HO populations (5, 10, 20, or 40 cells). For the TC and corticothalamic projections we optimized connection weight (range 0–2 mV) and connection probability (range 0–1). More details about model parameter optimization/exploration are described in section 2 of the **Supplementary material**.

Simulation of local field potentials

We simulated LFP extracellular recordings using the "line source approximation" (Buzsáki et al., 2012; Łęski et al., 2013; Parasuram et al., 2016), which is based on the sum of the transmembrane currents generated by each segment of each neuron, divided by the distance between the segment and the electrode. This method assumes that the electric conductivity (sigma = 0.3 mS/mm) and permittivity of the extracellular medium are constant everywhere and do not depend on frequency. LFP calculation, analysis and visualization was performed using NetPyNE.

Given the computational cost and memory requirements of simulating the full S1 model with morphologically-detailed neurons while recording LFPs, we calculated transmembrane currents only for the most central cells within an 84 μm (20% of 420 μm) diameter cylinder. This means that only 4.4% of the neurons were simulated in full detail, i.e., using full morphological reconstructions and with all synapses from the full model. The dynamics of the remaining cells (∼96% S1 and thalamus) were simulated using spike generators (VecStims in NEURON) using the spiking activity previously recorded in full simulations. That is, the inputs and activity of the 4% of fully detailed neurons from which the LFPs were calculated were identical to those of the full network simulations (when 100% of the neurons are simulated in detail).

Results

Reproduction of cell morphologies, physiological responses, spatial distribution and connectivity

Cells imported into NePyNE using the files from The Neocortical Microcircuit Collaboration NMCP (Ramaswamy et al., 2015), reproduced the morphological and electrophysiological characteristics of the original model (**Figure 1**): Mean firing rate and time to the first spike after a current clamp stimulation were fitted for all the 1,035 cell types. Firing dynamic differences were observed in cells with the stochastic K channel (StochKv), but their firing irregularity was partly preserved (**Supplementary Figure 1**) and, given their low proportion (3.63%), the average firing rates of all m-type populations closely matched those in the original model (**Figure 1H**).

We were able to recreate the general characteristics across the 7 BBP S1 microcircuit instances: the 31,346 cells were distributed randomly by layer, and probabilistic connections were generated for each of the 1,941 pathways (**Figure 2**). Here, we replaced the original connectivity method, based on the overlap between axonal and dendritic fields, with one based on connection probability based on cell type, layer, inter-cell distance, and dendritic pattern of post-synaptic locations. This network parameterization allowed us to rescale the microcolumn and generate different instances by changing the random number generator seed. Our probabilistic rules best reproduced the original number of connections using a Gaussian fit in most projection pathways (1,303 of 1,941) and an exponential fit plus a linear saturation in the remaining 638 cases (**Figure 2**).

Extension to include detailed thalamic circuits

We extended the model to include the somatosensory thalamic populations with projections to S1: RTN, POm, VPL, and VPM. The number of thalamic cells was adapted to fit a cylindrical column with the same radius as the S1 column. This facilitated the inclusion of topological connectivity rules between the two regions. We reproduced the firing dynamics of the different thalamic cell types using a single compartment neuron model (Moreira et al., 2021). The connections from TC cells to S1 were based on convergence rules derived from experimental data (Meyer et al., 2010), and synaptic physiological mechanisms were generalized from the BBP VPM projections to S1 layers 4 and 5 (Markram et al., 2015). Feedback connections originated from S1 cell types L5_TTPC2 and L6_TPC_L4 and targeted VPL and VPM following a topological organization, and POm in a following a non-topological broader distribution (**Figure 4**). The parameters of the thalamic circuit were adjusted to reproduce a stable self-sustained activity with rhythmic bursting and spindle oscillations (Destexhe and Contreras, 2011), as well as a shift in dynamics following localized excitatory input in the relay cells (Bonjean et al., 2012; Moreira et al., 2021).

Cortical and thalamic circuits independent response to background inputs (no thalamocortical connections)

We first evaluated the response to background inputs of the S1 cortical circuit and the thalamus circuit independently, i.e., without any connections between cortex and thalamus. When driven with background inputs, the S1 model generated spontaneous activity with most populations (48 out of 55) firing within physiological rates (**Figure 5**). To achieve this, excitatory and inhibitory background inputs were tuned via grid search parameter optimization (see section Materials and methods). **Figure 5** illustrates the S1 spontaneous activity results, including a spiking raster plot of all 31,346 cortical

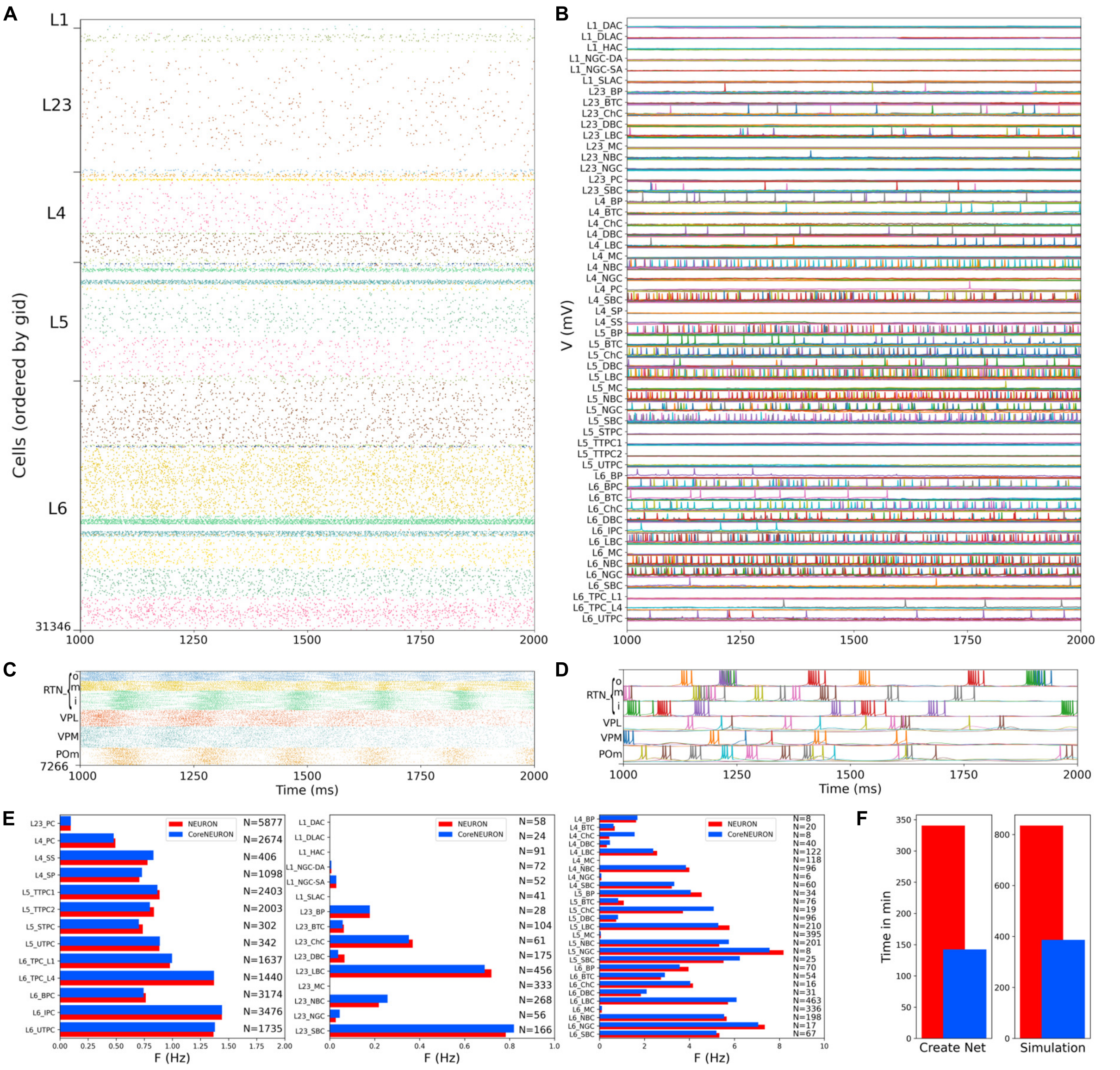


FIGURE 5

NetPyNE S1 and thalamus circuit response to background inputs (spontaneous activity). **(A)** Spiking raster plot of the 31,346 cells in the S1 column during 1 s (the first second was omitted to allow the network to reach a steady state). **(B)** Example voltage traces for each of the 207 me-types grouped by rows into their respective 55 m-types (same time period as raster plot). **(C)** Spiking raster plot of the 7,266 thalamic neurons during 1 s showing intrinsic oscillations. **(D)** Example voltage traces for each of the 6 thalamic populations. **(E)** Mean firing rates of each of the 55 m-types for NEURON (red) vs. CoreNEURON (blue). **(F)** Comparison of the time required to create the network and run the simulation on a 40-core Google Cloud virtual machine using NEURON (red) or CoreNEURON (blue).

cells, examples of voltage traces for each of the 207 me-type population, and the average firing rates for each of the 55 m-type populations. The thalamic populations, disconnected from S1 and driven by background inputs, exhibited stable self-sustained activity with rhythmic bursting at theta ∼6 Hz (Kim and McCormick, 1998; **Figures 5C,D**). These oscillations, which were most prominent in the RTNi and POm populations, emerged despite the lack of rhythmicity in the background inputs. The thalamic circuit oscillatory dynamics are consistent with the recurrent interactions between thalamic relay and reticular neurons described in previous studies (Destexhe et al., 1996a).

Simulations were run using NetPyNE and NEURON on a Google Cloud virtual machine with 40 cores. We compared the S1 results using the standard NEURON simulation engine vs. CoreNEURON, a state-of-the-art solver optimized for large scale parallel simulations on both CPUs and GPUs (Kumbhar et al., 2019). Both simulation engines produced very similar firing rates for each population (**Figure 5E**), with excitatory and L1-L3 inhibitory cells showing overall lower firing rates than L4-L6 inhibitory cells. The overall average firing rate across the 2 simulated seconds was 0.95 Hz in both cases (NEURON: 59,779 spikes; CoreNEURON: 59,749 spikes). This demonstrates the consistency of results obtained from both simulation engines, making CoreNEURON a viable alternative to study the S1 network. CoreNEURON was 2.4x faster to create the network and 2.2x faster to run the simulation (**Figure 5F**).

Somatosensory cortex circuit response to background inputs with short term plasticity (no thalamocortical connections)

We simulated the response of the S1 cortical circuit to background inputs but including short term plasticity (STP) in its local synaptic connections (**Figure 6A**). Adding STP resulted in the emergence of synchronous bursting within the S1 cortical column at approximately 1 Hz frequency (compare S1 raster in **Figures 5A**, **6A**). The spontaneous synchronous bursts first appeared in L5, and then spread to all S1 cells within 100 ms. **Figure 6B** shows an amplified raster plot of L4-L6 with 70 ms of activity at the time when spontaneous synchronous bursts started. **Figure 6C** shows example voltage traces of cortical and thalamic neurons, illustrating the spike synchrony of S1 and the thalamic bursts. These results are comparable to the simulations presented in Figures 11B,C of the original publication (Markram et al., 2015).

Somatosensory cortex and thalamic circuit response with bidirectional thalamic connectivity and cortical short term plasticity

We then simulated the full circuit with bidirectional connections between S1 and thalamus and STP in the thalamus to S1 connections (**Figure 7A**). The full cortico-thalamo-cortical circuit exhibited overall increased activity with S1 oscillations around 6 Hz frequency, and strong thalamic oscillatory activity at the same frequency. Oscillations were now synchronized across all S1 and thalamic populations. **Figure 7B** shows the voltage traces of several cortical and thalamic neurons, illustrating the spike synchrony of S1 and thalamic populations. Finally, in **Figure 7C** we compare the mean firing rate for all S1 and thalamic populations with (red bars) and without (blue bars) bidirectional TC connectivity. All 55 model populations now exhibited physiological firing rates. Adding bidirectional TC connectivity resulted in a modest increase of the overall mean firing rate, from 4.96 to 5.29 Hz, with more pronounced increases in the average firing rates of L1 and L2/3 inhibitory populations. These results do not have a direct correspondence to any in the original BBP publication, since the original model did not include thalamic populations bidirectionally connected to cortex.

Somatosensory cortex and thalamic circuit response after reducing the extracellular calcium concentration to reproduce asynchronous *in vivo*-like state

Experimental evidence shows that extracellular calcium concentration ($[Ca^{2+}]_o$) *in vivo* is lower than *in vitro*, and, as a consequence, PSP amplitudes are also lower (Borst, 2010). Markram et al. (2015) divided the dependency of PSPs on $[Ca^{2+}]_o$ into three classes for specific connection types: steep, intermediate, and shallow. Here, the PSP amplitudes were set to have steep dependence for connections between PC-PC and PC-distal targeting cell types (DBC, BTC, MC, BP) and a shallow dependence for connections between PC-proximal targeting (LBCs, NBCs, SBCs, ChC). An intermediate level of dependence was assumed for other connections. To simulate reduced $[Ca^{2+}]_o$ in the NetPyNE implementation we decreased the *cao* parameter from 2.0 to 1.2 in all cells, and modified the use parameter of synaptic transmission (U) adding a factor to multiply its value from 0.25 to 0.75. This resulted in a transition from synchrony (*in vitro*-like, **Figures 6**, **7**) to asynchrony (*in vivo*-like, **Figure 8**) network states, as in Markram et al. (2015). The increased asynchrony happened both for the S1-TH disconnected (**Figure 8A**) and the S1-TH connected (**Figure 8B**) cases. Decreasing extracellular calcium concentration resulted in decreased firing rates for most populations (compare **Figures 7C**, **8C**). In both the *in vitro* and *in vivo* conditions, cortical firing rates were generally slightly higher for the S1-TH connected case. However, bidirectional thalamic connectivity (S1-TH connected) resulted in increased thalamic population firing rates *in vitro*, whereas under *in vivo* conditions (low $[Ca^{2+}]_o$), it lowered thalamic firing rates and decreased synchrony.

Local field potentials recorded from the *in vivo*-like somatosensory cortex circuit

We simulated extracellular LFP recordings at multiple depths and horizontal distances in the S1 cortical column during the *in vivo*-like state (**Figure 9**). The LFP calculation was based on the transmembrane currents across all segments of neurons. To reduce the computational cost of the calculation,

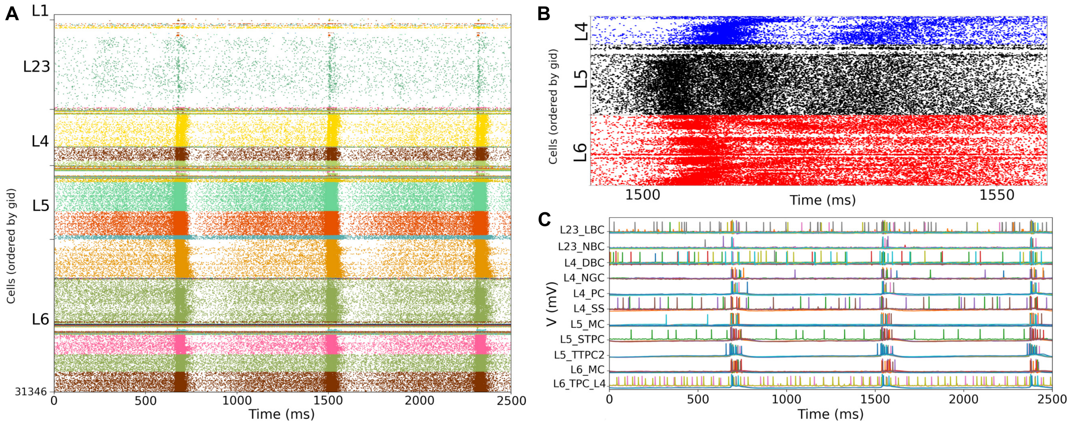


FIGURE 6

NetPyNE S1 circuit response to background inputs with short-term plasticity (STP). **(A)** Spiking raster plot of S1 with STP. S1 and thalamus were not interconnected; only intracortical connections were included. **(B)** Amplified spiking raster plot **(A)** showing the 70 ms around the time when synchronous bursts first occur in L5 (black) and then propagate to L6 (red) and L4 (blue). **(C)** Example traces from **(A)** showing spike synchrony across cortical populations. Rasters in A show 2.5 s after steady state was reached.

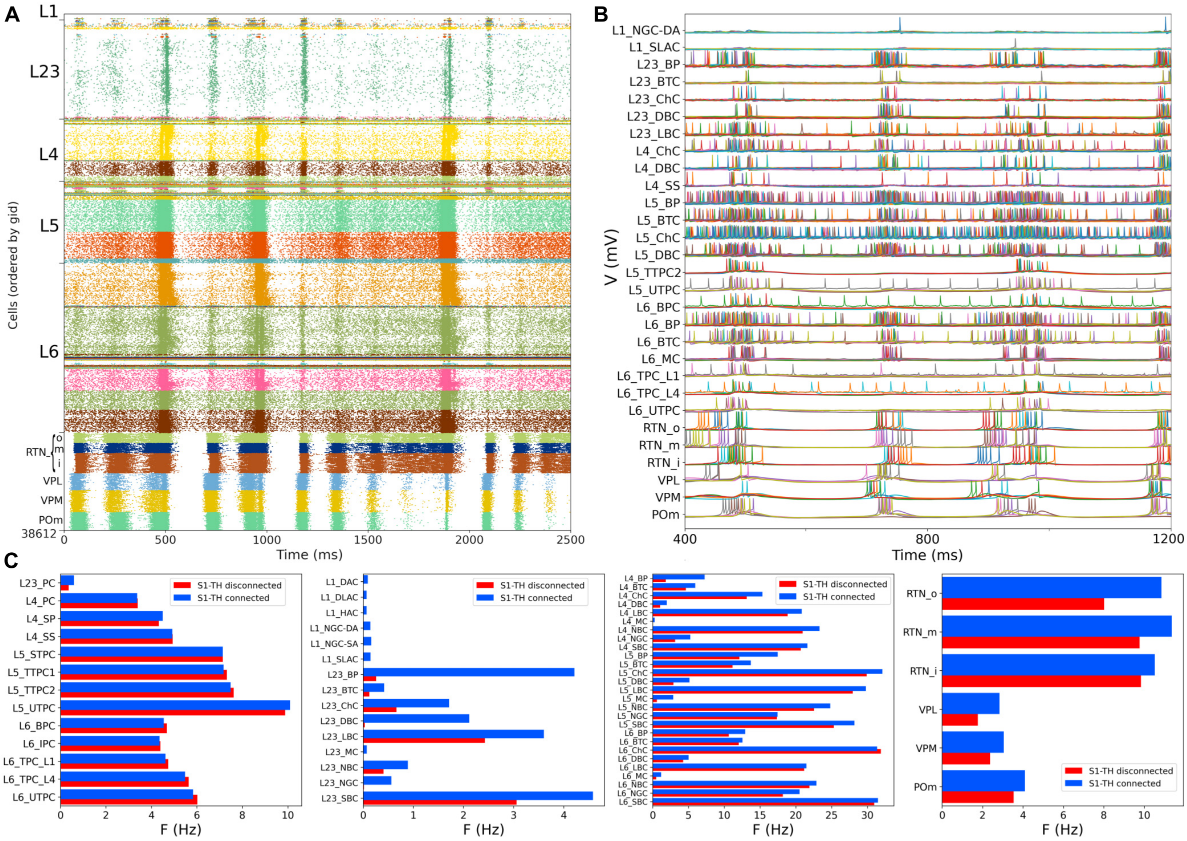


FIGURE 7

NetPyNE S1 and thalamic circuit response with bidirectional thalamic connectivity and cortical STP. **(A)** Spiking raster plot of the fully connected circuit model, including bidirectional connections between S1 and thalamus (shows 2.5 s of simulation after steady state was reached). Oscillations at ~6 Hz were now synchronized across all S1 and thalamic populations. **(B)** Example traces from **(A)** during 800 ms showing spike synchrony across cortical and thalamic populations. **(C)** Comparison of mean firing rates of each of the 55 S1 and 6 thalamic m-types with (*S1-TH connected*) and without (*S1-TH disconnected*) bidirectional thalamocortical connectivity (compare rasters in panel **A** and **Figure 6A**, respectively).

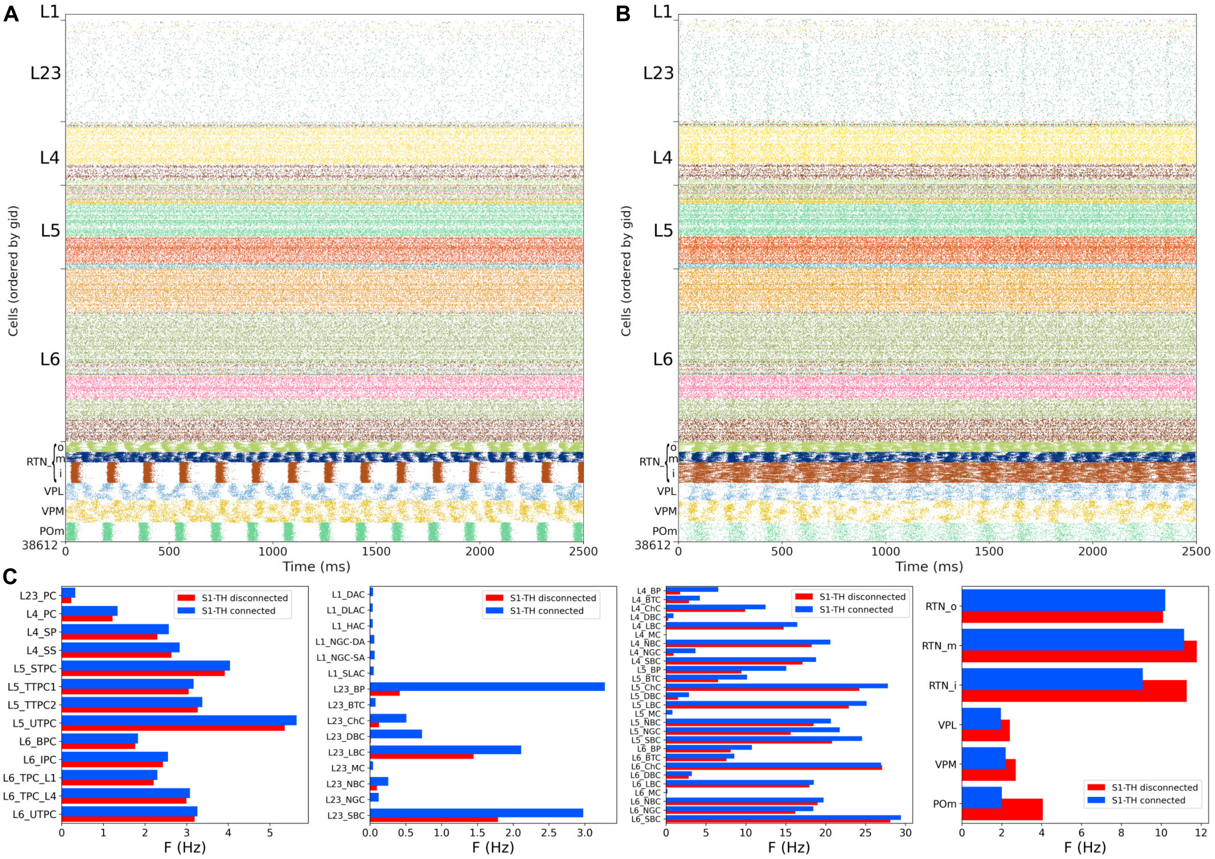


FIGURE 8

NetPyNE S1 and thalamic circuit response with low extracellular calcium (*in vivo*-like asynchronous states). **(A)** Spiking raster plot of spontaneous activity with only intracortical and intrathalamic connections (*S1-TH disconnected*). **(B)** Spiking raster plot of the fully connected circuit model, including bidirectional connections between S1 and thalamus (*S1-TH connected*). **(C)** Comparison of mean firing rates of each of the 55 S1 and 6 thalamic m-types without (*S1-TH disconnected*) and with (*S1-TH connected*) bidirectional thalamocortical connectivity.

we included only the 1,376 morphologically detailed neurons (4.4% of the total neurons) within a central cylinder of 84 μm diameter (**Figures 9A,B**). The remaining 29,970 S1 and 7,266 thalamic neurons were simulated using artificial spike generators (VecStims) to ensure the dynamics of the detailed neurons were identical as in the full scale simulation (**Figure 9D**). The simulated morphologically-detailed neurons therefore included the same 2,702,107 synapses with STP as those in the full *in vivo* simulation. We inserted recording electrodes at 4 different cortical depths (500, 1000, 1500 and 2000 um) and 2 radial (x-z plane) distances (0 and 297 μm) from the cylinder center (**Figure 9C**). Recorded LFP amplitudes were in the order 1–1,000 μV consistent with the experimental literature (Reimann et al., 2013; Hagen et al., 2018; **Figures 9E,F**). The amplitudes of LFPs recorded further away from the cylinder were attenuated ∼10 to 20x compared to those closer to the cylinder center, for example, the peak amplitudes for electrodes 1 and 5 were 401 and 24 μV, respectively. This is consistent with LFP amplitude being inversely proportional to the squared distance between electrode and current sources. When compared to the *in vitro* recorded LFPs (see **Supplementary Figures 2, 3**), which exhibited stronger slow frequency oscillations, the attenuation measured at the distant electrodes was only 5x. This is consistent with the observed frequency-dependent attenuation phenomenon, where high frequency signals are attenuated more than low frequency oscillations (Buzsáki et al., 2012; Reimann et al., 2013). The LFP power spectral densities generally depict an inverse relationship between power and frequency, which is typically described in animal LFP recordings. Overall, these preliminary results demonstrate the model can be used to simulate and capture several physiological features of extracellular LFPs.

Discussion

We provide here the first large-scale S1 model that is accessible to the wider community, building on the details of the prior state-of-the-art BBP S1 model. The model closely reproduced the original cell morphologies

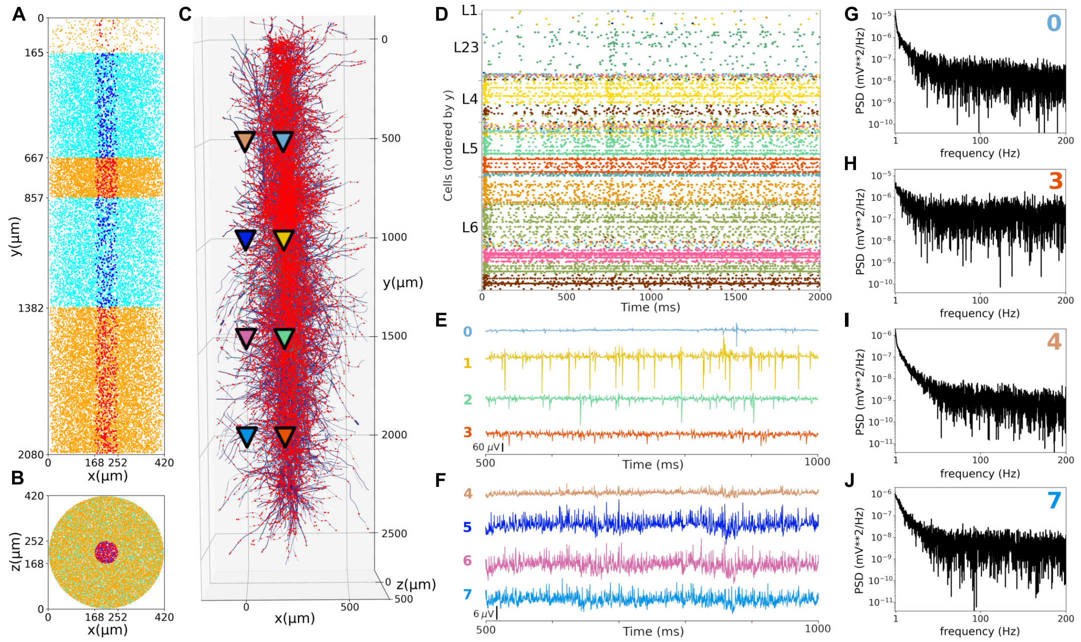


FIGURE 9
Local field potentials (LFPs) recorded from the *in vivo*-like S1 circuit. **(A,B)** Lateral and top-down 2D representation of the location of 1,376 morphological cells within a cylinder with 2,082 µm height and 84 µm radius. Morphologically-detailed neurons are shown in red (L1, L4, and L6) and blue (L23 and L5); whereas the 29,970 simplified cells (spike generators) are shown in orange (L1, L4, and L6) and cyan (L23 and L5) circles. **(C)** 3D representation of the morphologically-detailed neurons with the location of all synapses (red dots), and the location of the 8 LFP recording electrodes at 4 different depths and 2 radial distances (color triangles). **(D)** Spiking raster plot of the morphologically-detailed neurons used to calculate the LFP. **(E)** LFP signals recorded at the 4 electrodes in the center of the cylinder (colors correspond to triangles in panel **C**). Electrodes numbered 0, 1, 2, and 3 correspond with cortical depths (y) 500, 1,000, 1,500, and 2,000 µm, respectively. **(F)** Same as E, but for the LFPs recorded at a radial distance of 297 µm; electrodes 4–7. **(G–J)** Power spectral densities (PSDs) for electrodes 0, 3, 4, and 7 calculated over a 10-s simulation (initial transient period was not included). PSDs exhibit an inverse relationship between power and frequency.

and electrophysiological responses for the 207 morpho-electrical (me) cell types, with 5 examples for each, totaling 1,035 cell models (Figure 1); the spatial distribution of these cells across layers; and the connectivity properties of the 1,941 pathways, including synaptic dynamics and short-term plasticity (Figures 2, 3). After tuning, the simulations produced reasonable dynamics with rates and activity patterns corresponding to *in vivo* measures of cortical activity (Figures 5, 6). There was no direct comparison to the full network dynamics of the original BBP model since original simulation data was not available. However, firing rates and overall 1 Hz underlying oscillation when using STP was comparable to that seen in the original model version paper (Markram et al., 2015; Figure 11). We also extended the model by adding thalamic circuits, including 6 distinct thalamic populations that reproduced cell and circuit-level dynamics, and with intrathalamic, TC and corticothalamic connectivity derived from experimental data (Figure 4). The addition of the thalamic circuit resulted in distinct activity patterns and synchronous activity across cortical and thalamic populations (Figure 7). Finally, we decreased the extracellular calcium concentration ($[\mathrm{Ca}^{2+}]_o$) to simulate *in vivo*-like states with asynchronous activity (Figure 8). LFPs recorded at multiple cortical depths and horizontal distances exhibited realistic oscillatory patterns and power spectra, including the experimentally observed distance- and frequency-dependent attenuation (Figure 9).

The S1 model now joins other NetPyNE cortical simulations: generic cortical circuits (Romaro et al., 2021), auditory and motor TC circuits (Sivagnanam et al., 2020; Dura-Bernal et al., 2022a,b), as well as simulations of thalamus (Moreira et al., 2021), dorsal horn of spinal cord (Sekiguchi et al., 2021), Parkinson's disease (Ranieri et al., 2021) and schizophrenia (Metzner et al., 2020). These large cortical simulations can be extremely computer-intensive, which is a major motivation for NetPyNE's facilities that allow one to readily simplify the network by swapping in integrate-and-fire or small-compartmental cell models, or by down-scaling to more manageable sizes. CoreNEURON is a state-of-the-art solver optimized for large scale parallel simulations, now included as part of the official NEURON package. The optimization on CPUs and the ability to run across GPUs in CoreNEURON is another key NetPyNE feature enhancing runnability. In the present case, the original S1 model is largely inaccessible, despite the cooperation of its designers, since it requires specialized tools, workflows, and training.

Nonetheless, most of the data required to replicate it is available via the NMCP, which we ourselves used to implement the NetPyNE version.

We were able to get substantial speedup (> 2x) for both model setup and run using CoreNEURON despite only using CPUs with no GPU at this time. We note that using CPU cycles/timestep would provide a more direct measure than the total simulation time, which may be affected by other factors such as background processes (Girardi-Schappo et al., 2017). Nonetheless, the speedup obtained is consistent with the 2–7x speedups recently reported when using CoreNEURON on CPUs to simulate large-scale models (Kumbhar et al., 2019; Awile et al., 2022). For example, the NetPyNE-based motor cortex model exhibited a speedup of 3.5x on Google Cloud. When using GPUs, speedups of up to 40x were reported. The differences in firing activity seen with NEURON vs. CoreNEURON are expected due to vectorization of the compute kernels in CoreNEURON and potential differences due to different solvers when using NMODL with sympy. Further differences are to be expected once this is extended to GPUs (Jézéquel et al., 2015; Kumbhar et al., 2019).

We made 2 significant changes in our port to NetPyNE. First, we did not replicate the stochastic K channels that appear in 3.6% of the neurons, making our port somewhat simpler than the original. This channel required writing custom code and made simulations slower, but it could be added to the model in a future iteration. Second, we have not utilized the original cell-to-cell connection mappings that were obtained by BBP from direct microscopic observations of overlap between pre-synaptic axonal fields and post-synaptic dendritic fields (so-called Peter's principle). In the original BBP S1 model, the use of cell-to-cell connections necessarily limited the simulation to use precisely the original model's cell morphologies, cell positions and scales. It also required storing and loading large files of connection data. We therefore replaced this connection framework with one based on connection probability based on cell type (including layer), inter-cell distance, and dendritic pattern of post-synaptic locations. Although saving somewhat on space, there is a time-space tradeoff since this requires further calculations on start-up. Despite these limitations, we had excellent agreement with both cell model matching and connection density matching.

Our implementation also incorporates a novel model of thalamic circuitry that recapitulates multiple experimental findings at the single neuron and circuit levels. Thalamic and reticular cell models were adjusted to reproduce the reported resting membrane potential, approximately −60 and −80 mV, respectively (Jahnsen and Llinás, 1984; Destexhe et al., 1996b; Sherman and Guillery, 2009). In the networks, TC cells fired at low frequencies (2–4 Hz), while reticular cells fired at higher rates (6–14 Hz), consistent with values previously reported in the literature (Kim and McCormick, 1998). Thalamic simulations also showed rhythmic rebound bursting when hyperpolarized and regular spiking activity at depolarized potentials (Destexhe and Contreras, 2011). The thalamic network exhibited synchronous activity within and across several populations, as well as synchronous firing with cortical populations, particularly in the *in vitro* condition. These synchronous patterns likely emerged as a consequence of the implemented intrathalamic and TC connectivity, including the topological organization based on axonal footprints (Lam et al., 2006; Lam and Sherman, 2007, 2011). Taken together, these results make the thalamic circuit a valuable extension to the S1 model, by providing a more realistic input source to the cortical circuit and enabling the study of TC interactions.

Recording the intracellular potential of multiple neurons *in vivo* requires an elaborate set up and is generally challenging. Extracellular recordings are more accessible and therefore more commonly used in experimental studies. Extracellular potentials are generated by transmembrane currents resulting from neuronal activity. Evidence suggests the main contributor to extracellular signals are synaptic currents (Buzsáki et al., 2012; Reimann et al., 2013). Computational modeling coupled with recordings of field activity in animals can provide insights into the cooperative behavior of neurons and increase our understanding of how these processes contribute to the extracellular signal (Buzsáki et al., 2012; Reimann et al., 2013). The simulated LFPs exhibited a similar range of amplitudes as those recorded experimentally, and reproduced several features of LFPs, including the distance-dependent and frequency-dependent attenuation. This opens the door to future validation of the model by comparing LFPs to those recorded experimentally under different conditions, and to future studies of the biophysical sources of LFPs and the exact contribution of different network populations (Hagen et al., 2018).

To place our model in the context of recent literature, we follow the classification proposed by a recent review of data-driven models structural connectivity at the microcircuit level (Shimoura et al., 2021). Our model can be classified as using conductance-based, morphologically-detailed neurons, with a network size of 38,612 neurons, synaptic plasticity and network spatiality (e.g., distance-based connectivity). Our NetPyNE implementation, together with the original BBP implementation (Markram et al., 2015), constitute the only conductance-based, morphologically-detailed models of S1. These contrast with previous models of S1 (Huang et al., 2022) or of generic sensory cortex (Potjans and Diesmann, 2014) that employ simpler neuron models (leaky integrate and fire point neurons). Models with detailed conductance-based and morphologically-detailed neurons have been developed for other cortical regions, including V1 (Arkhipov et al., 2018; Billeh et al., 2020), M1 (Dura-Bernal et al., 2022b), A1 (Dura-Bernal et al., 2022a), and CA1 (Bezaire et al., 2016; Ecker et al., 2020). Our model is also unique in incorporating thalamic neurons and TC bidirectional topological connectivity. Previous TC circuit models included less biophysically-detailed neuron models and simpler connectivity (Izhikevich and Edelman, 2008), or focused on single cell (Iavarone et al., 2019) or small circuit models (Destexhe et al., 1996a). An impressively detailed model of the thalamoreticular microcircuit has recently been developed, although this is limited to the VPL and RTN somatosensory thalamus regions (Iavarone et al., 2022).

As outlined above, the level of biophysical, morphological and connectivity detail in the model is very high compared to most existing models. Although this makes it harder to simulate and tune, it also enables exploration of a unique set of scientific questions that simpler models cannot address, or at least not with the same level of realism. Here we included two results that require and justify the level of detail of the model. First, we simulated a network state with lower extracellular calcium concentration that more closely resembles the *in vivo* conditions (**Figure 8**). Secondly, we calculated realistic LFPs, which critically depend on the sum of transmembrane currents along detailed neuronal morphologies (**Figure 9**). We also describe the methodology for future model parameter explorations, and provide a basic code set up example to explore the effects of inhibitory GABAergic connections on network dynamics. Examples of parameter explorations in NetPyNE-based biophysically detailed circuit models can be found in our related publications on motor and auditory cortex models (Sivagnanam et al., 2020; Dura-Bernal et al., 2022a,b), including an exploration of the effects of long-range and neuromodulatory inputs.

Consequently, our port of the S1 model provides a quantitative framework that can be used in several ways. First, it can be used to perform *in silico* experiments to explore sensory processing under the assumption of various coding paradigms or brain disease, including the representation of whisker motion (Bosman et al., 2011; Huang et al., 2022), maximization of sensory dynamic range (Gautam et al., 2015), response to unexpected sensory inputs (Amsalem et al., 2020) schizophrenia (Metzner et al., 2020) and Parkinson's disease (Ranieri et al., 2021). Second, drug effects can be directly tested in the simulation (Neymotin et al., 2016)—this is an advantage of a multiscale model with scales from molecule to network, which is not available in simpler models that elide these details. Third, the model constitutes a unified multiscale framework for organizing our knowledge of S1 which serves as a dynamical database to which new physiological, transcriptomic, proteomic, and anatomical data can be added. This framework can then be utilized as a community tool for researchers in the field to test hypotheses and guide the design of new experiments.

Data availability statement

The model and data for this study can be found in the following repositories and online platforms: GitHub (https://github.com/suny-downstate-medical-center/S1_Thal_NetPyNE_Frontiers_2022), ModelDB (https://senselab.med.yale.edu/ModelDB/), Open Source Brain (https://www.opensourcebrain.org/), and EBRAINS Model Catalog (https://ebrains.eu/).

Author contributions

FB, JM, WL, and SD-B conceived and designed research and drafted the manuscript. FB, JM, LT, and SD-B implemented and optimized the simulation code. FB and JM prepared the figures. All authors contributed to the manuscript revision, read, and approved the submitted version.

Funding

This work was funded by the following grants: NIH NIBIB U24EB028998, NSF 1904444-1042C, NYS SCIRB DOH01-C32250GG-3450000, and NIH NIDCD R01DC012947. This research was funded in part by the Aligning Science Across Parkinson's (ASAP-020572) through the Michael J. Fox Foundation for Parkinson's Research (MJFF).

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2022.884245/full#supplementary-material

References

Amsalem, O., King, J., Reimann, M., Ramaswamy, S., Muller, E., Markram, H., et al. (2020). Dense computer replica of cortical microcircuits unravels cellular underpinnings of auditory surprise response. *BioRxiv* [Preprint]. doi: 10.1101/2020.05.31.126466

Arkhipov, A., Gouwens, N. W., Billeh, Y. N., Gratiy, S., Iyer, R., Wei, Z., et al. (2018). Visual physiology of the layer 4 cortical circuit in silico. *PLoS Comput. Biol.* 14:e1006535. doi: 10.1371/journal.pcbi.1006535

Awile, O., Kumbhar, P., Cornu, N., Dura-Bernal, S., King, J. G., Lupton, O., et al. (2022). Modernizing the NEURON simulator for sustainability, portability, and performance. *Front. Neuroinform.* 16:884046. doi: 10.3389/fninf.2022.884046

Azarfar, A., Calcini, N., Huang, C., Zeldenrust, F., and Celikel, T. (2018). Neural coding: A single neuron's perspective. *Neurosci. Biobehav. Rev.* 94, 238–247. doi: 10.1016/j.neubiorev.2018.09.007

Barthas, F., and Kwan, A. C. (2017). "Secondary motor cortex: Where 'sensory' meets 'motor' in the rodent frontal cortex. *Trends Neurosci.* 40, 181–193. doi: 10.1016/j.tins.2016.11.006

Bezaire, M. J., Raikov, I., Burk, K., Vyas, D., and Soltesz, I. (2016). Interneuronal mechanisms of hippocampal theta oscillations in a full-scale model of the rodent CA1 circuit. *Elife* 5:e18566. doi: 10.7554/eLife.18566

Billeh, Y. N., Cai, B., Gratiy, S. L., Dai, K., Iyer, R., Gouwens, N. W., et al. (2020). Systematic integration of structural and functional data into multi-scale models of mouse primary visual cortex. *Neuron* 106, 388.e–403.e. doi: 10.1016/j.neuron.2020.01.040

Bonjean, M., Baker, T., Bazhenov, M., Cash, S., Halgren, E., and Sejnowski, T. (2012). Interactions between core and matrix thalamocortical projections in human sleep spindle synchronization. *J. Neurosci.* 32, 5250–5263. doi: 10.1523/JNEUROSCI.6141-11.2012

Borst, J. G. (2010). The low synaptic release probability in vivo. *Trends Neurosci.* 33, 259–266. doi: 10.1016/j.tins.2010.03.003

Bosman, L. W., Houweling, A. R., Owens, C. B., Tanke, N., Shevchouk, O. T., Rahmati, N., et al. (2011). Anatomical pathways involved in generating and sensing rhythmic whisker movements. *Front. Integr. Neurosci.* 5:53. doi: 10.3389/fnint.2011.00053

Buzsáki, G., Anastassiou, C. A., and Koch, C. (2012). The origin of extracellular fields and currents — EEG, ECoG, LFP and spikes. *Nat. Rev. Neurosci.* 13, 407–420. doi: 10.1038/nrn3241

Carnevale, N. T., and Hines, M. L. (2006). *The neuron book*. New York, NY: Cambridge University Press.

Dai, K., Hernando, J., Billeh, Y. N., Gratiy, S. L., Planas, J., Davison, A. P., et al. (2020). The sonata data format for efficient description of large-scale network models. *PLoS Comput. Biol.* 16:e1007696. doi: 10.1371/journal.pcbi.1007696

Destexhe, A., and Contreras, D. (2011). "The fine structure of slow-wave sleep oscillations: From single neurons to large networks," in *Sleep and anesthesia: Neural correlates in theory and experiment*, ed. A. Hutt (New York, NY: Springer New York), 69–105. doi: 10.1007/978-1-4614-0173-5_4

Destexhe, A., Bal, T., McCormick, D. A., and Sejnowski, T. J. (1996a). Ionic Mechanisms underlying synchronized oscillations and propagating waves in a model of ferret thalamic slices. *J. Neurophysiol.* 76, 2049–2070. doi: 10.1152/jn.1996.76.3.2049

Destexhe, A., Contreras, D., Steriade, M., Sejnowski, T. J., and Huguenard, J. R. (1996b). In vivo, in vitro, and computational analysis of dendritic calcium currents in thalamic reticular neurons. *J. Neurosci.* 16, 169–185. doi: 10.1523/JNEUROSCI.16-01-00169.1996

Dura-Bernal, S., Neymotin, S. A., Suter, B. A., Dacre, J., Schiemann, J., Duguid, I., et al. (2022b). Multiscale model of primary motor cortex circuits reproduces in vivo cell type-specific dynamics associated with behavior. *bioRxiv* [Preprint]. doi: 10.1101/2022.02.03.479040

Dura-Bernal, S., Griffith, E. Y., Barczak, A., O'Connell, M. N., McGinnis, T., Schroeder, C. E., et al. (2022a). Data-driven multiscale model of macaque auditory thalamocortical circuits reproduces in vivo dynamics. *bioRxiv* [Preprint]. doi: 10.1101/2022.02.03.479036

Dura-Bernal, S., Suter, B. A., Gleeson, P., Cantarelli, M., Quintana, A., Rodriguez, F., et al. (2019). NetPyNE, a tool for data-driven multiscale modeling of brain circuits. *Elife* 8:e44494. doi: 10.7554/eLife.44494

Ecker, A., Romani, A., Sáray, S., Káli, S., Migliore, M., Falck, J., et al. (2020). Data-driven integration of hippocampal ca1 synaptic physiology in silico. *Hippocampus* 30, 1129–1145. doi: 10.1002/hipo.23220

Erö, C., Gewaltig, M. O., Keller, D., and Markram, H. (2018). A cell atlas for the mouse brain. *Front. Neuroinform.* 12:84. doi: 10.3389/fninf.2018.00084

Fuhrmann, G., Segev, I., Markram, H., and Tsodyks, M. (2002). Coding of temporal information by activity-dependent synapses. *J. Neurophysiol.* 87, 140–148. doi: 10.1152/jn.00258.2001

Gal, E., London, M., Globerson, A., Ramaswamy, S., Reimann, M. W., Muller, E., et al. (2017). Rich Cell-type-specific network topology in neocortical microcircuitry. *Nat. Neurosci.* 20, 1004–1013. doi: 10.1038/nn.4576

Gautam, S. H., Hoang, T. T., McClanahan, K., Grady, S. K., and Shew, W. L. (2015). Maximizing sensory dynamic range by tuning the cortical state to criticality. *PLoS Comput. Biol.* 11:e1004576. doi: 10.1371/journal.pcbi.1004576

Girardi-Schappo, M., Bortolotto, G. S., Stenzinger, R. V., Gonsalves, J. J., and Tragtenberg, M. H. (2017). Phase diagrams and dynamics of a computationally efficient map-based neuron model. *PLoS One* 12:e0174621. doi: 10.1371/journal.pone.0174621

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., et al. (2010). NeuroML: A language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comput. Biol.* 6:e1000815. doi: 10.1371/journal.pcbi.1000815

Gleeson, P., Cantarelli, M., Marin, B., Quintana, A., Earnshaw, M., Sadeh, S., et al. (2019). Open source brain: A collaborative resource for visualizing, analyzing, simulating, and developing standardized models of neurons and circuits. *Neuron* 103, 395–411.e5. doi: 10.1016/j.neuron.2019.05.019

Guo, K., Yamawaki, N., Barrett, J. M., Tapies, M., and Shepherd, G. M. G. (2020). Cortico-Thalamo-cortical circuits of mouse forelimb S1 are organized primarily as recurrent loops. *J. Neurosci.* 40, 2849–2858. doi: 10.1523/JNEUROSCI.2277-19.2020

Hagen, E., Næss, S., Ness, T. V., and Einevoll, G. T. (2018). Multimodal Modeling of neural network activity: Computing LFP, ECoG, EEG, and MEG signals with LFPy 2.0. *Front. Neuroinform.* 12:92. doi: 10.3389/fninf.2018.00092

Hennig, M. H. (2013). Theoretical models of synaptic short term plasticity. *Front. Comput. Neurosci.* 7:154. doi: 10.3389/fncom.2013.00154

Hill, S., and Tononi, G. (2005). Modeling sleep and wakefulness in the thalamocortical system. *J. Neurophysiol.* 93, 1671–1698. doi: 10.1152/jn.00915.2004

Huang, C., Zeldenrust, F., and Celikel, T. (2022). Cortical representation of touch in silico. *Neuroinformatics* doi: 10.1007/s12021-022-09576-5 [Epub ahead of print].

Iavarone, E., Simko, J., Shi, Y., Bertschy, M., García-Amado, M., Litvak, P., et al. (2022). Thalamic control of sensory enhancement and sleep spindle properties in a biophysical model of thalamoreticular microcircuitry. *bioRxiv* [Preprint]. doi: 10.1101/2022.02.28.482273

Iavarone, E., Yi, J., Shi, Y., Zandt, B. J., O'Reilly, C., Van Geit, W., et al. (2019). Experimentally-constrained biophysical models of tonic and burst firing modes in thalamocortical neurons. *PLoS Comput. Biol.* 15:e1006753. doi: 10.1371/journal.pcbi.1006753

Izhikevich, E. M., and Edelman, G. M. (2008). Large-scale model of mammalian thalamocortical systems. *Proc. Natl. Acad. Sci. US.A.* 105, 3593–3598. doi: 10.1073/pnas.0712231105

Jahnsen, H., and Llinás, R. (1984). Ionic basis for the electro-responsiveness and oscillatory properties of guinea-pig thalamic neurones in vitro. *J. Physiol.* 349, 227–247. doi: 10.1113/jphysiol.1984.sp015154

Jézéquel, F., Lamotte, J.-L., and Saïd, I. (2015). "Estimation of numerical reproducibility on CPU and GPU," in *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, (Piscataway, NJ: IEEE), 675–680. doi: 10.15439/2015f29

Kim, U., and McCormick, D. A. (1998). The functional influence of burst and tonic firing mode on synaptic interactions in the thalamus. *J. Neurosci.* 18, 9500–9516. doi: 10.1523/JNEUROSCI.18-22-09500.1998

Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., et al. (2019). CoreNEURON : An optimized compute engine for the NEURON Simulator. *Front. Neuroinform.* 13:63. doi: 10.3389/fninf.2019.00063

Lam, Y. W., and Sherman, S. M. (2007). Different topography of the reticulothalmic inputs to first- and higher-order somatosensory thalamic relays revealed using photostimulation. *J. Neurophysiol.* 98, 2903–2909. doi: 10.1152/jn.00782.2007

Lam, Y. W., and Sherman, S. M. (2011). Functional organization of the thalamic input to the thalamic reticular nucleus. *J. Neurosci.* 31, 6791–6799. doi: 10.1523/JNEUROSCI.3073-10.2011

Lam, Y. W., Nelson, C. S., and Sherman, S. M. (2006). Mapping of the functional interconnections between thalamic reticular neurons using photostimulation. *J. Neurophysiol.* 96, 2593–2600. doi: 10.1152/jn.00555.2006

Łęski, S., Lindén, H., Tetzlaff, T., Pettersen, K. H., and Einevoll, G. T. (2013). Frequency Dependence of signal power and spatial reach of the local field potential. *PLoS Comput. Biol.* 9:e1003137. doi: 10.1371/journal.pcbi.1003137

Luczyńska, A., Dziewiatkowski, J., Jagalska-Majewska, H., Kowiański, P., Wójcik, S., Labuda, C., et al. (2003). Qualitative and quantitative analysis of the postnatal development of the ventroposterolateral nucleus of the thalamus in rat and rabbits. *Folia Morphol.* 62, 75–87.

Lytton, W. W., Seidenstein, A. H., Dura-Bernal, S., McDougal, R. A., Schürmann, F., and Hines, M. L. (2016). Simulation neurotechnologies for advancing brain research: Parallelizing large networks in NEURON. *Neural Comput.* 28, 2063–2090. doi: 10.1162/NECO_a_00876

Ma, P. M. (1991). The barrelettes–architectonic vibrissal representations in the brainstem trigeminal complex of the mouse. I. Normal structural organization. *J. Comp. Neurol.* 309, 161–199. doi: 10.1002/cne.903090202

Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

McDougal, R. A., Bulanova, A. S., and Lytton, W. W. (2016). Reproducibility in Computational neuroscience models and simulations. *IEEE Trans. Biomed. Eng.* 63, 2021–2035. doi: 10.1109/TBME.2016.2539602

Metzner, C., Mäki-Marttunen, T., Karni, G., McMahon-Cole, H., and Steuber, V. (2020). The effect of alterations of schizophrenia-associated genes on gamma band oscillations. *bioRxiv* [Preprint]. doi: 10.1101/2020.09.28.316737

Meyer, H. S., Wimmer, V. C., Hemberger, M., Bruno, R. M., de Kock, C. P., Frick, A., et al. (2010). Cell type-specific thalamic innervation in a column of rat vibrissal cortex. *Cereb. Cortex* 20, 2287–2303. doi: 10.1093/cercor/bhq069

Migliore, M., Cannia, C., Lytton, W. W., Markram, H., and Hines, M. L. (2006). Parallel network simulations with NEURON. *J. Comput. Neurosci.* 21, 119–129.

Moreira, J. V. S., Borges, F. S., Doherty, D., Lytton, W. W., and Dura-Bernal, S. (2021). *Topographically detailed computational model of the motor and somatosensory thalamic circuits*. Available online at: https://www.abstractsonline.com/pp8/#!/10485/presentation/16321 (accessed February 25, 2022).

Murray Sherman, S., and Guillery, R. W. (2001). "Chapter II – the nerve cells of the thalamus," in *Exploring the thalamus*, eds S. Murray Sherman and R. W. Guillery (San Diego, CA: Academic Press), 19–58. doi: 10.4324/9781315152837-8

Murray, J. D., and Anticevic, A. (2017). Toward understanding thalamocortical dysfunction in schizophrenia through computational models of neural circuit dynamics. *Schizophr. Res.* 180, 70–77. doi: 10.1016/j.schres.2016.10.021

Neymotin, S. A., Dura-Bernal, S., Moreno, H., and Lytton, W. W. (2016). Computer modeling for pharmacological treatments for dystonia. *Drug Discov. Today Dis. Models* 19, 51–57.

Ohno, S., Kuramoto, E., Furuta, T., Hioki, H., Tanaka, Y. R., Fujiyama, F., et al. (2012). A morphological analysis of thalamocortical axon fibers of rat posterior thalamic nuclei: A single neuron tracing study with viral vectors. *Cereb. Cortex* 22, 2840–2857. doi: 10.1093/cercor/bhr356

O'Reilly, C., Iavarone, E., Yi, J., and Hill, S. L. (2021). Rodent Somatosensory thalamocortical circuitry: Neurons, synapses, and connectivity. *Neurosci. Biobehav. Rev.* 126, 213–235.

Parasuram, H., Nair, B., D'Angelo, E., Hines, M., Naldi, G., and Diwakar, S. (2016). Computational modeling of single neuron extracellular electric potentials and network local field potentials using LFPsim. *Front. Comput. Neurosci.* 10:65. doi: 10.3389/fncom.2016.00065

Peña-Rangel, T. M., Lugo-Picos, P. I., Báez-Cordero, A. S., Hidalgo-Balbuena, A. E., Luma, A. Y., Pimentel-Farfan, A. K., et al. (2021). Altered sensory representations in parkinsonian cortical and basal ganglia networks. *Neuroscience* 466, 10–25.

Petrof, I., Viaene, A. N., and Sherman, S. M. (2015). Properties of the Primary somatosensory cortex projection to the primary motor cortex in the mouse. *J. Neurophysiol.* 113, 2400–2407. doi: 10.1152/jn.00949.2014

Potjans, T. C., and Diesmann, M. (2014). The Cell-type specific cortical microcircuit: Relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Ramaswamy, S., Courcol, J. D., Abdellah, M., Adaszewski, S. R., Antille, N., Arsever, S., et al. (2015). The neocortical microcircuit collaboration portal: A resource for rat somatosensory cortex. *Front. Neural Circuits.* 9:44. doi: 10.3389/fncir.2015.00044

Ranieri, C. M., Montino Pimentel, J., Romano, M. R., Elias, L. A., Romero, R. A. F., Lones, M. A., et al. (2021). A data-driven biophysical computational model of Parkinson's disease based on marmoset monkeys. *IEEE Access* 9, 122548–122567.

Reimann, M. W., Anastassiou, C. A., Perin, R., Hill, S. L., Markram, H., and Koch, C. (2013). A biophysically detailed model of neocortical local field potentials predicts the critical role of active membrane currents. *Neuron* 79, 375–390. doi: 10.1016/j.neuron.2013.05.023

Reimann, M. W., King, J. G., Muller, E. B., Ramaswamy, S., and Markram, H. (2015). An algorithm to predict the connectome of neural microcircuits. *Front. Comput. Neurosci.* 9:120. doi: 10.3389/fncom.2015.00120

Reimann, M. W., Horlemann, A., Ramaswamy, S., Muller, E., and Markram, H. (2017b). Morphological diversity strongly constrains synaptic connectivity and plasticity. *Cereb. Cortex* 27, 4570–4585. doi: 10.1093/cercor/bhx150

Reimann, M. W., Nolte, M., Scolamiero, M., Turner, K., Perin, R., Chindemi, G., et al. (2017a). Cliques of neurons bound into cavities provide a missing link between structure and function. *Front. Comput. Neurosci.* 11:48. doi: 10.3389/fncom.2017.00048

Romaro, C., Najman, F. A., Lytton, W. W., Roque, A. C., and Dura-Bernal, S. (2021). NetPyNE Implementation and rescaling of the potjans-diesmann cortical microcircuit model. *Neural Comput.* 33, 1993–2032. doi: 10.1162/neco_a_01400

Sekiguchi, K., Medlock, L., Dura-Bernal, S., Prescott, S. A., and Lytton, W. W. (2021). Multiscale computer model of the spinal dorsal horn reveals changes in network processing associated with chronic pain. *bioRxiv* [Preprint]. doi: 10.1101/2021.06.09.447785

Shepherd, G. M. G., and Yamawaki, N. (2021). Untangling the cortico-thalamo-cortical loop: Cellular pieces of a knotty circuit puzzle. *Nat. Rev. Neurosci.* 22, 389–406. doi: 10.1038/s41583-021-00459-3

Sherman, S. M., and Guillery, R. W. (2009). *Exploring the thalamus and its role in cortical function*, 2nd Edn. Cambridge, MA: Mit Press.

Shimoura, R. O., Pena, R. F. O., Lima, V., Kamiji, N. L., Girardi-Schappo, M., and Roque, A. C. (2021). Building a model of the brain: From detailed connectivity maps to network organization. *Eur. Phys. J.* 230, 2887–2909.

Sivagnanam, S., Gorman, W., Doherty, D., Neymotin, S. A., Fang, S., Hovhannisyan, H., et al. (2020). Simulating large-scale models of brain neuronal circuits using google cloud platform. *PEARC20 (2020)* 2020, 505–509. doi: 10.1145/3311790.3399621

Sugitani, M., Yano, J., Sugai, T., and Ooyama, H. (1990). Somatotopic organization and columnar structure of vibrissae representation in the rat ventrobasal complex. *Exp. Brain Res.* 81, 346–352. doi: 10.1007/BF00228125

Vázquez, Y., Salinas, E., and Romo, R. (2013). Transformation of the neural code for tactile detection from thalamus to cortex. *Proc. Natl. Acad. Sci. U.S.A.* 110, E2635–E2644. doi: 10.1073/pnas.1309728110

frontiers | Frontiers in Neuroscience

TYPE Original Research
PUBLISHED 29 September 2022
DOI 10.3389/fnins.2022.944262

OPEN ACCESS

EDITED BY
Markus Diesmann,
Helmholtz Association of German Research Centres (HZ), Germany

REVIEWED BY
Cyrille Mascart,
Cold Spring Harbor Laboratory, United States
Mingyuan Meng,
The University of Sydney, Australia

*CORRESPONDENCE
Lingfei Mo
lfmo@seu.edu.cn

SPECIALTY SECTION
This article was submitted to Neuromorphic Engineering, a section of the journal Frontiers in Neuroscience

RECEIVED 15 May 2022
ACCEPTED 30 August 2022
PUBLISHED 29 September 2022

CITATION
Mo L and Tao Z (2022) EvtSNN: Event-driven SNN simulator optimized by population and pre-filtering. *Front. Neurosci.* 16:944262. doi: 10.3389/fnins.2022.944262

EvtSNN: Event-driven SNN simulator optimized by population and pre-filtering

Lingfei Mo* and Zhihan Tao

FutureX Lab, School of Instrument Science and Engineering, Southeast University, Nanjing, China

Recently, spiking neural networks (SNNs) have been widely studied by researchers due to their biological interpretability and potential application of low power consumption. However, the traditional clock-driven simulators have the problem that the accuracy is limited by the time-step and the lateral inhibition failure. To address this issue, we introduce EvtSNN (Event SNN), a faster SNN event-driven simulator inspired by EDHA (Event-Driven High Accuracy). Two innovations are proposed to accelerate the calculation of event-driven neurons. Firstly, the intermediate results can be reused in population computing without repeated calculations. Secondly, unnecessary peak calculations will be skipped according to a condition. In the MNIST classification task, EvtSNN took 56 s to complete one epoch of unsupervised training and achieved 89.56% accuracy, while EDHA takes 642 s. In the benchmark experiments, the simulation speed of EvtSNN is 2.9–14.0 times that of EDHA under different network scales.

KEYWORDS

spiking neural network (SNN), event-driven, acceleration, simulator, unsupervised learning

1. Introduction

Spiking neural networks (SNNs) (Maass, 1997) have attracted increasing attention because of their characteristics, including preferable biological interpretability and low-power processing potential (Akopyan et al., 2015; Shen et al., 2016; Davies et al., 2018; Moradi et al., 2018; Pei et al., 2019; Li et al., 2021; Pham et al., 2021). Compared to traditional artificial neural networks (ANNs), SNNs increase the time dimension so that they naturally support information processing in the temporal domain. To introduce the extra time dimension into the calculation, two methods are usually adopted: clock-driven and event-driven. The idea of clock-driven is to discretize the time and update the state of all neurons in each timestamp. The clock-driven method is widely used in the existing SNN frameworks (simulators) (Goodman and Brette, 2008; Hazan et al., 2018; Stimberg et al., 2019) because it is simulated by the iterative method which can be compatible with the differential equations of most neuron models. However, this method has two problems that cannot be ignored. Firstly, there is a conflict between simulation accuracy and calculation speed. The smaller the time step, the higher the simulation accuracy and the larger the calculation amount. Secondly, lateral inhibition cannot be effective on other neurons that fire lately in the same time slice.

In the event-driven method, the state of neurons is updated when spikes are received, which means that the sparsity of spikes can be fully utilized to reduce computations. The realization of event-driven simulation on hardware (Davies et al., 2018; Li et al., 2021) has the ability of parallel computing and the potential of low-power processing, but it is costly and less flexible than software. Our team previously proposed an event-driven software simulation framework EDHA (Event-Driven High Accuracy), whose core task is to maintain the pulse priority queue (Mo et al., 2021). During the simulation, the earliest spike is popped from the queue, and then postsynaptic neurons are updated independently. However, the high complexity of its single update limits the overall simulation speed.

In this paper, an event-driven software simulator named EvtSNN (Event SNN) is introduced, which includes two contributions. To begin with, neurons are clustered into populations, which means that intermediate results can be reused. In addition, pre-filtering is adopted to avoid unnecessary calculations according to the condition. After rewriting the framework code with the C++ programming language and combining these two innovations, the simulation speed of EvtSNN has been greatly improved. In the ablation experiment task, the processing capacity of EvtSNN(C++) reached 117.8 M spikes × fan-outs/s, which was 13 times that of EDHA(java). In the unsupervised training task of MNIST, the network (784–400) took 56 s to train one epoch with an accuracy of 89.59%, which is 11.4 times faster than EDHA.

In Section 2, we describe the related work, including unsupervised learning and supervised learning, as well as clock-driven and event-driven simulation. In Section 3, the principle of EDHA is reviewed, and two innovations are proposed to accelerate the event-driven simulation. Section 4 contains several comparison experiments and results. And the discussion is in Section 5. Finally, Section 6 summarizes the current work.

2. Related works

The learning methods of SNN mainly include supervised learning and unsupervised learning. Supervised learning is similar to the traditional ANN, which can be trained by the gradient back-propagation (BP) method. However, the spike is non-differentiable, so ANN to SNN (Sengupta et al., 2019; Deng and Gu, 2021) or surrogate gradient (Neftci et al., 2019) is often used to handle this problem. Unsupervised learning refers to the learning style of neurons in biology, which has better biological interpretability. Similar to EDHA, EvtSNN pays more attention to biological interpretability, and currently mainly supports unsupervised learning rules, such as the spike time-dependent plasticity (STDP) (Masquelier and Kheradpisheh, 2018) learning rule.

Most SNN simulators need to deal with the temporal dimension. Clock-driven and event-driven approaches are often used to deal with this problem. Owing to high model compatibility, the clock-driven method is mostly used in software SNN simulation frameworks, such as Brian (Goodman and Brette, 2008), Brian2 (Stimberg et al., 2019), and BindsNET (Hazan et al., 2018), etc. Brian and Brian2 were frameworks based on code generation, which can specify the neuron dynamic equation and synapse update rule to generate corresponding codes. On the other hand, BindsNET was based on PyTorch (Paszke et al., 2019) and implemented the behaviors of neurons and synapses by writing code. Its advantage is that Graphics Processing Unit (GPU) acceleration can be conveniently carried out based on PyTorch. Although with high model compatibility, the clock-driven method also has some problems. Firstly, the existence of time-slice limits the simulation accuracy, and the calculations increase inversely with the decrease of time-step. Secondly, when using lateral inhibition, multiple neurons in the same layer can activate in a time step, leading to multiple winning neurons that are unfavorable for training.

The event-driven method is mostly used in Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC). FPGA is an expensive and flexible chip with limited resources, which is suited for laboratory prototype validation and not for actual deployment (Pham et al., 2021). Some researchers have made ASIC for simulating SNN, such as DYNAPs (Moradi et al., 2018), TrueNorth (Akopyan et al., 2015), and Loihi (Davies et al., 2018), etc. The power consumption of these is at the milliwatt level, but they have expensive costs for design, and it is not convenient to add new functions.

Event-driven simulation can take advantage of the sparsity of spike events and neural connections. EDHA is an event-driven framework proposed by our team earlier, whose core task is to maintain the spike priority queue. Without the concept of time-slice, it solves the problems of lateral suppression failure and the conflict between accuracy and speed in the clock-driven method. However, there are large calculations in the update of neurons in EDHA, which limits the overall simulation speed. Therefore, two innovations have been proposed to solve this problem.

3. Methods

3.1. Workflow of EDHA framework

The SNN simulation framework needs to deal with additional time dimensions. The clock-driven method is similar to polling, while the event-driven method calculates only when pulse events are received.

As an event-driven simulation framework, the core task of EDHA is to maintain the spike priority queue. The specific steps are as follows: (1) take out the earliest spike event that should be issued from the priority queue; (2) update the neurons which are connected behind the fired neuron (i.e., post-synaptic neurons); (3) adjust the priority queue elements according to the predicted spike information. Figure 1 is a flow chart of the above processing details.

3.2. Neuron and synapse model

The leaky integrate-and-fire (LIF) model (Gerstner et al., 2014; Mo et al., 2021) was adopted in this paper, which is represented by Equations (1) and (2). $\delta(t)$ in the formula is the impulse function, and other parameters are explained in Table 1. It should be noted that there is no dependence on membrane potential in Equation (2), which facilitates the derivation of other formulas.

$$\begin{cases} \frac{dv}{dt} = -\frac{v}{\tau_v} + g_E & v < v_{th} \\ v = v_{reset} & v \geq v_{th} \end{cases} \tag{1}$$

$$\begin{cases} \frac{dg_E}{dt} = -\frac{g_E}{\tau_g} + C\sum_{i,j} w_i \delta(t - t_{i,j}) & v < v_{th} \\ g_E = 0 & v \geq v_{th} \end{cases} \tag{2}$$

The adaptive threshold method is often used to avoid a few neurons firing too frequently. It is increased by the fixed item (θ) at each fire, and otherwise exponentially decays toward θ_0 with a time constant (τ_θ). It is more difficult to generate new spikes after increasing the threshold, which is beneficial for other neurons' learning (Masquelier and Kheradpisheh, 2018). Owing to large τ_θ, the decay of v_{th} in a short time can be ignored in practice, which simplifies some calculations.

The plasticity synaptic model is an important part of unsupervised learning. Like EDHA, the STDP learning rule was employed in this paper, see formula (3) (Mo et al., 2021). In which σ_+ and σ_- are the amplitude constants of LTP and LTD, τ_+ and τ_- are the time constant, and t_{pre} and t_{post} are the firing time of the pre-/post-synaptic neuron.

$$\Delta w = \begin{cases} \sigma_+ \cdot e^{-\frac{t_{post} - t_{pre}}{\tau_+}} & t_{pre} < t_{post} \\ \sigma_- \cdot e^{-\frac{t_{pre} - t_{post}}{\tau_-}} & t_{pre} > t_{post} \end{cases} \tag{3}$$

3.3. Update steps of neuron

In step (2) of the event-driven framework workflow, there are three sub-steps: (a) calculate the current state according to the update interval, (b) handle the currently input spike event, and (c) estimate the potential pulse based on the current state. Figure 2 shows the diagram of the above three sub-steps, and the formula of each step is related to the neuron model. It is important to note that the neuron's membrane potential may continue to increase for some time after receiving a single spike, so sub-step (c) is required to estimate the potential pulse of neurons.

Unless received spike or activated, v and g_E will not change instantaneously, and Equations (4) and (5) are obtained. T represents the time of the currently received spike. At any time from T to the next pulse received, the state can be solved by the above two equations, which is the computing formula in sub-step (a).

$$v(T + \Delta t) = v(T) \cdot e^{-\frac{\Delta t}{\tau_v}} + g_E(T) \cdot \frac{\tau_g \tau_v}{\tau_g - \tau_v}(e^{-\frac{\Delta t}{\tau_g}} - e^{-\frac{\Delta t}{\tau_v}}) \tag{4}$$

$$g_E(T + \Delta t) = g_E(T) \cdot e^{-\frac{\Delta t}{\tau_g}} \tag{5}$$

In sub-step (b), the influence of the pulse on the current state is calculated. According to Equation (1) and (2), v is unchanged and g_E increases by $Cw_{i,j}$ instantaneously when received spike.

In sub-step (c), the neuron needs to estimate whether it can generate a new spike according to current states. After receiving the spike, the conductance of the neuron increases, which indirectly promotes the increase of the membrane potential, and it is difficult to predict the precise timing of the pulse firing. Instead, the method of calculating the peak value of voltage was adopted. If the peak exceeds the threshold, combined Equation (4), the dichotomy method was employed to calculate the exact spike time (Mo et al., 2021).

Here are the details of solving peak membrane potential. Equation (1) shows that the membrane potential is affected by the self-attenuation term and conductance. The voltage will increase when the conductance is large and plays a leading role. In the case no subsequent spikes are received, the conductance g_E decays exponentially to 0. In other words, if the conductance is large enough, the voltage will rise first and then fall, and reach the peak during this period. In other cases, potential decreases monotonically under the influence of attenuation term. According to Equation (4), the extreme point $t^{'}$ and the corresponding peak membrane potential were obtained, i.e., Equations (6) and (7).

$$t^{'} = \frac{\tau_g \tau_v}{\tau_g - \tau_v} \cdot [\ln(\frac{\tau_g}{\tau_v}) + \ln(1 - \frac{\tau_g - \tau_v}{\tau_g \tau_v} \cdot \frac{v(T)}{g_E(T)})] \tag{6}$$

$$v_{max} = \begin{cases} g_E(T)\tau_v[\frac{\tau_g}{\tau_v}(1 - \frac{\tau_g - \tau_v}{\tau_v \tau_g} \cdot \frac{v(T)}{g_E(T)})]^{-\frac{\tau_v}{\tau_g - \tau_v}} & t^{'} > 0 \\ v(T) & \text{else} \end{cases} \tag{7}$$

In the above formula, there is a large computational complexity in sub-step (a) and (c), which is the bottleneck in the whole simulate computation. However, it is found that two points can be optimized in the actual calculation. First, there are many same Δt of postsynaptic neurons (fan-outs), which means the decay factors can be reused. Second, the complicated

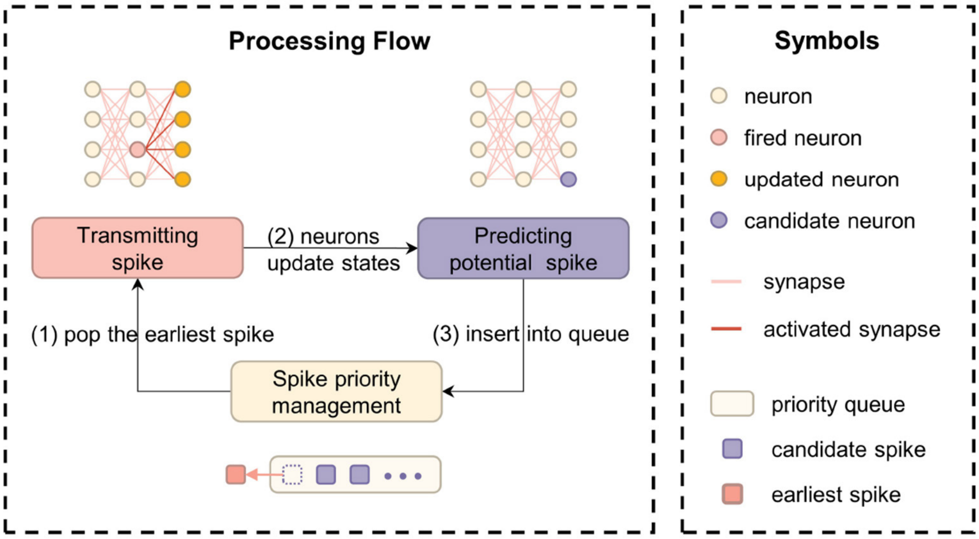


FIGURE 1
The computational flow chart of EDHA and EvtSNN, which core task is to maintain the spike priority queue.

TABLE 1 Explanation and experimental value of the parameters in the LIF neuron model.

Parameter	Explanation	Section 4.1	Section 4.2	Section 4.3
v	Membrane potential			
g_E	Excitatory conductance			
τ_v	The leaky time constant of voltage	20	20	50
τ_g	The leaky time constant of conductance	1	5	5
C	Conductivity gain coefficient	1	1,000/($N{\cdot}Fr$)	Variable
w_i	Weight connect to presynaptic neuron i	Rand (0, 0.3)	1	Rand (0.45, 0.55)
$t_{i,j}$	The time of jth spike of presynaptic neuron i			
v_{th}	Activation threshold of neuron			
v_{reset}	Reset potential after fire	0	0	0
θ_0	Adaptive threshold baseline	40	100	75
θ	Increment of adaptive threshold	0.5	0.5	2.5
τ_θ	Time constant of adaptive threshold	1e7	1e7	1e7

calculation in sub-step (c) can be omitted if the neuron will not be activated in the current state. Therefore, in the following sections, the two innovations of population computing and pre-filtering were proposed, which avoid repeated calculation and unnecessary calculation, respectively.

3.4. Population computing

The supported connection style in EDHA was the fine-grained (one-to-one) connection between neurons, and it is extremely flexible. However, the coarse-grained (layer-to-layer) connection is usually adopted due to its convenience. Furthermore, all neurons in the population will be updated when they receive a spike so that they have the same update interval. According to Equations (4) and (5), the decay factors of neurons in the population can be reused due to the same update interval.

To cooperate with the population, the local priority queue was introduced, which sorted the pulses in the population and provided the earliest spike to the global queue. It slightly increases the complexity of the program and reduces some flexibility. However, using the concept of the population to

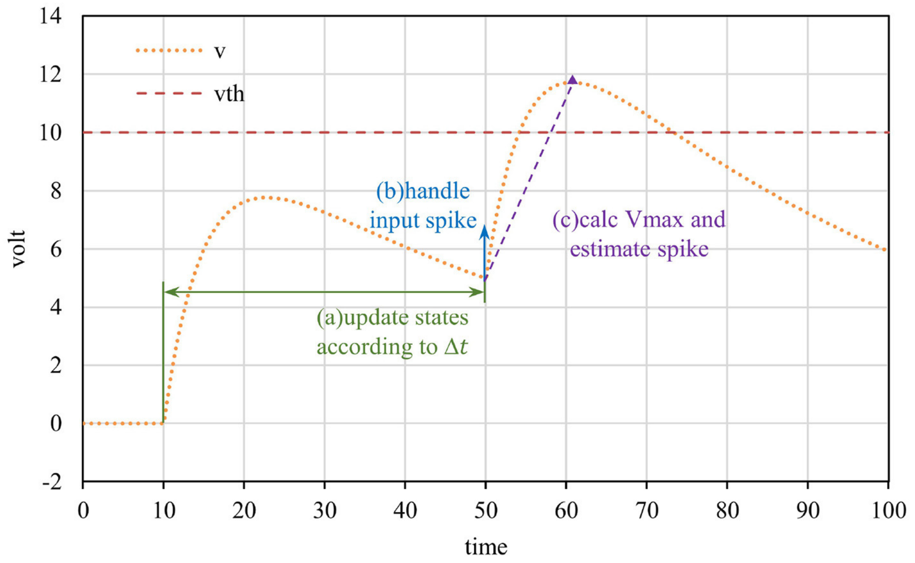


FIGURE 2
Event-driven neuron updated diagram. The neuron is updated when a spike is received, which consists of three sub-steps: updating states, processing the input spike, and predicting the potential spike.

manage multiple neurons achieves high cohesion and low coupling, which is beneficial to follow-up work.

In addition, due to the introduction of the concept of population, the simulation could support more operations, such as the delayed update of the adaptive threshold. In EDHA, neurons update the threshold when receiving a spike, but usually, the threshold change is very small. The delay update threshold has little effect on the simulation accuracy while avoiding a lot of exponential calculations.

3.5. Pre-filtering

In sub-step (c), the peak potential is calculated according to the current state. In most cases, the peak potential of the neuron will not exceed the threshold after receiving the spike, which means that many peak calculations are unnecessary. Therefore, the judgment condition of pre-screening was proposed to filter out unnecessary calculations at a low cost.

At time t_f, the membrane potential increases to v_{th}, and inequality (8) can be derived. This inequality is a necessary condition for activation. If the current conductance does not meet this condition, it means that the neuron will not generate a spike in the future unless more input events are received.

$$v^{'}(t_f) = -\frac{v(t_f)}{\tau_v} + g_E(t_f) \geq 0 \tag{8}$$

Inequality (8) can be used as the condition of pre-filtering, but it does not involve other information. For example, when the difference between v and v_{th} is large, a greater g_E is required to boost v. When voltage attenuation is neglected, it is easy to get the contribution of conductance to the potential which is shown in inequation (9). After integration and arranging, inequality (10) is obtained, which reflects the contribution of conductance to membrane potential. Combining inequality (8), the judgment condition (11) can be obtained finally.

$$\frac{dv}{dt} = -\frac{v}{\tau_v} + g_E \leq g_E \tag{9}$$

$$\Delta v \leq \int g_E \, dt = \int -\tau_g \frac{dg_E}{dt} \, dt = -\tau_g \Delta g_E \tag{10}$$

$$g_E(T) \geq g_E(t_f) + \frac{v(t_f) - v(T)}{\tau_g} \geq \frac{v_{th}}{\tau_v} + \frac{v_{th} - v(T)}{\tau_g} \tag{11}$$

Inequation (11) is also an essential condition for the neuron to activate and then emit a spike. If the current state does not meet this condition, no pulse will be generated in the future (unless more input events are received), which means that complicated peak calculations can be ignored. This inequality makes use of most information, such as voltage, threshold, and two time-constants (the "leak" terms τ_v and τ_g), and it is a good pre-filtering condition. Figure 3 shows the pre-filtering effect, and only a few cases (the purple solid dots) need to calculate the peak value, thus avoiding unnecessary computational overhead.

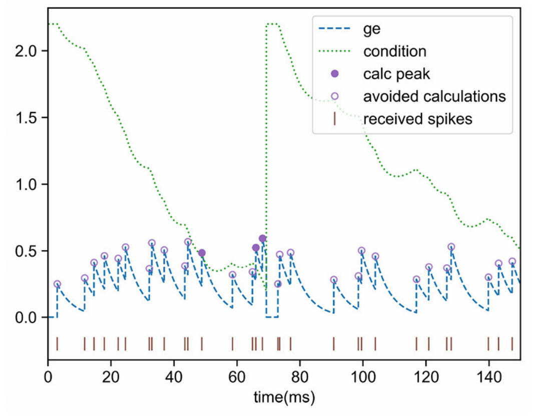


FIGURE 3
Effect of conductance pre-filtering. The peak value needs to be calculated only when the updated conductance is greater than the condition.

3.6. Method summary

In this section, the principle of the event-driven simulator EDHA is reviewed, and two innovations are proposed to optimize the problem of a large amount of calculation in a single update, which is shown in Figure 4. Based on EDHA and the above two optimizations, we propose the new simulation algorithm, named EvtSNN.

4. Experiments and results

In this section, Brian2 (python), BindsNET (python), and EDHA (java) were selected as comparison frameworks. To be fair, each framework was tested on the central processing unit (CPU) platform with one thread. The test platform is a workstation with Intel Xeon Silver 4215R@3.2 GHz CPU and the operating system is Ubuntu20.04. The parameters used in the experiments are shown in Table 1. The test programs of this section are shown in http://www.snnhub.com/FutureX-Lab/EvtSNN-exe.

In Brian2 simulator, one can specify different device as the backend, including runtime (cython or numpy), cpp_standalone, genn and so on. For the cpp_standalone and genn backends, they can avoid repeating compilation by setting *build_on_run* to False, but this approach cannot adjust the parameters of the next run according to the simulation results of this run. The runtime (cython) is the relatively fastest backend when the simulation time is short, and it is also selected as the default backend of Brian2 in the follow-up experiments. Furthermore, without changing the network structure and parameters, an empty function could be used to mask *before_run* after *run(0ms)*, which can avoid generating duplicate code, named Brian2*.

4.1. Performance test

The network structure used in the test included an input neuron layer (200 neurons), an output neuron layer (200 neurons), and fully connected synapses. With a 10 Hz average fire rate and 10,000 ms simulation time, there were 20 k spikes in total. For testing, we used the algorithm described in Bautembach et al. (2020) for generating the input spikes to the networks.

Table 2 shows the performance test results of the above four simulators. It can be seen that the simulation time of the clock-driven framework changes inversely with dt. When dt was reduced from 1.0 to 0.01 ms, the simulation accuracy was improved, but the simulation time was increased to nearly 100 times. Thanks to no time-step limitation, the event-driven simulator has extremely high simulation accuracy, which was almost the same as the output spikes of the clock-driven simulator with dt = 0.01 ms (∼511). And the computation of event-driven simulator is only affected by the number of spikes and neuron fan-outs. According to Table 2, the processing capacity of EvtSNN reached 20 k×200/0.033958 s = 117.8 M spikes×fan-outs/second. Compared with EDHA, the performance of EvtSNN has greatly improved without loss of simulation accuracy.

In addition, we compare the results of Brian2 and BindsNET using GPU acceleration. Among them, Brian2GeNN (GPU) takes a long time to compile, and when dt is reduced from 1.0 to 0.01 ms the time-consuming increment of Brian2GeNN is similar to that of Brian2 (CPU), which means that the processing capacity of Brian2GeNN has not been significantly improved. At the same time, the speed of BindsNET-GPU is not as fast as that of BindsNET (CPU). It can be seen that in some cases, such as small networks and sparse pulses, using the GPU does not make the SNN simulation faster.

4.2. Benchmarking

The amount of computation of event-driven simulation is related to the number of spikes and neuron fan-outs. Theoretically, when the number of neurons in each layer increases and the firing rate of the input layer remains, the time-consuming increases in square trend. In this section, the simulation experiments of different scale networks were designed. The time step (dt) of Brian2 and BindsNET was set to 1.0 ms.

The structure of the network was two neuron layers, which were fully connected. We marked the number of neurons in both layers as *N* and the firing frequency of the input layer

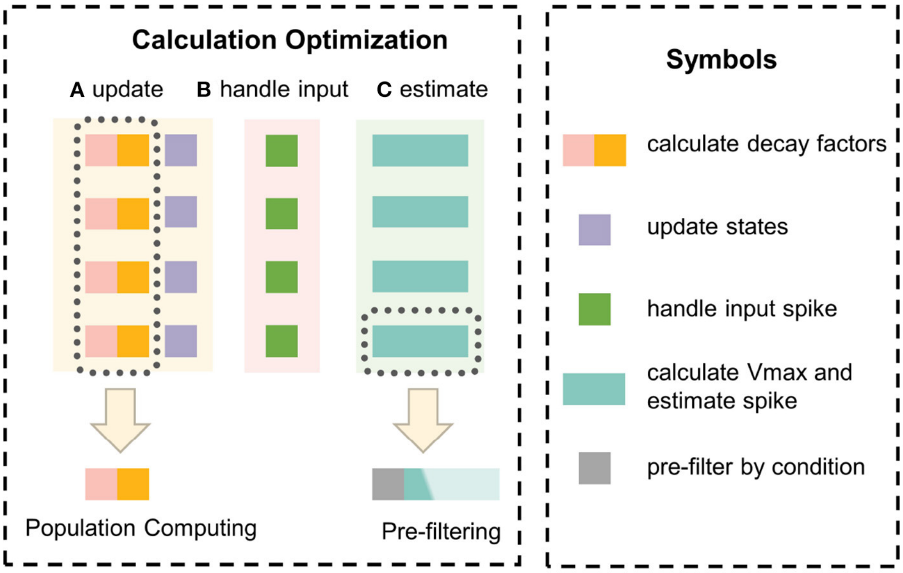


FIGURE 4
The optimization principle of innovation points. Population computing uses the hidden information of the same update interval to reuse decay factors to avoid repeated calculations. Pre-filtering adopts a low-cost judgment condition to filter out unnecessary calculations to reduce the average time-consuming.

as *Fr*. Under different *N* and *Fr*, we recorded the execution time of simulating the network for 1,000 ms. To avoid spike number drastic change, the impact factor of weight (i.e., *C*) changed inversely according to *N* and *Fr*, thus stabilizing the firing frequency of the output layer.

The experimental results are shown in Figure 5. The horizontal and vertical coordinates of the chart were logarithmic coordinates, which could easily see the time-consuming growth trend. As predicted by the previous theory, the time consumption of the four simulators increased approximately square with the increase of the number of neurons (purple dotted baseline). Without the learning rule, in the case of small network scale (e.g., hundreds of neurons), the fastest simulator was EvtSNN, otherwise, it was Brian2*. When *Fr* increased (e.g., 20 Hz), compared with the clock-driven simulator, the performance of EvtSNN and EDHA decreased relatively. EvtSNN was the fastest framework at different *N* and *Fr* when using the STDP learning rule. The speed of EvtSNN is 2.9–14.0 times that of EDHA under the same calculation flow.

Brian2* performs better when it comes to large-scale network simulation, but the following points should be noted. Firstly, the statistical time taken by Brain2* does not include the time taken to generate the code, so it appears to be faster. In many cases it is necessary to change the run parameters, at which point Brian2 needs to recompile the code, so the extra time taken to compile cannot be ignored. Secondly, the input layer in the benchmark experiments had the same frequency of pulse delivery for each neuron, and there was no channel sparsity, so the event-driven framework, EvtSNN, did not take full advantage of sparse computation. Finally, the time step(dt) of the clock-driven framework in the benchmark experiments

TABLE 2 Performance test results of simulators.

Simulator	Method	dt (ms)	Time-consuming (s)	Output spikes
BindsNET	Clock-driven	1.0	2.353	499
		0.1	22.486	508
		0.01	231.169	510
BindsNET-GPU	Clock-driven	1.0	4.068	499
		0.1	38.851	508
		0.01	383.747	510
Brian2	Clock-driven	1.0	0.667	587
		0.1	3.169	519
		0.01	30.284	511
Brian2*	Clock-driven	1.0	0.312	587
		0.1	3.035	519
		0.01	29.915	511
Brian2GeNN	Clock-driven	1.0	19.630	587
		0.1	21.787	519
		0.01	49.443	511
EDHA	Event-driven	None	0.439	510
EvtSNN(ours)	Event-driven	None	0.034	511

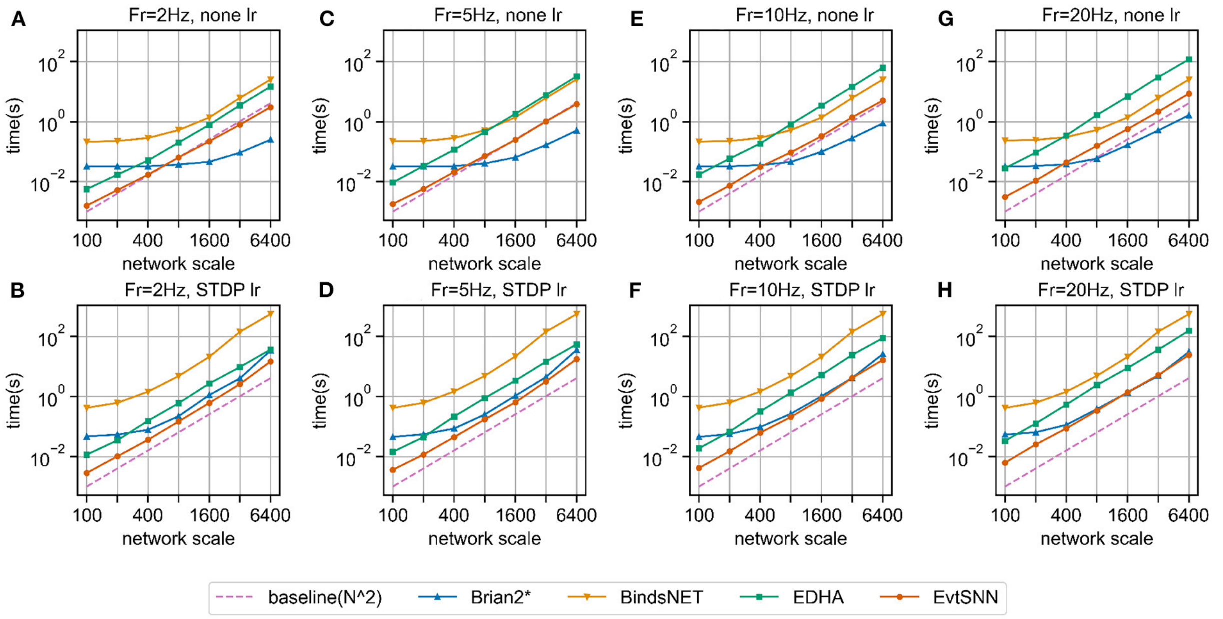


FIGURE 5
Benchmark simulation results. **(A–D)** No learning rule, the firing frequency of input layer neurons is 2, 5, 10, and 20 Hz, respectively. **(E–H)** Using the STDP learning rule, the firing frequency is 2, 5, 10, and 20 Hz, respectively.

is 1.0 ms, and decreasing dt for higher simulation accuracy will increase the time consumption inversely; whereas EvtSNN has no time step limitation and has a fixed time consumption and high simulation accuracy.

4.3. Unsupervised training task on MNIST dataset

In this section, the performances of several simulation frameworks were compared on the unsupervised training task of MNIST. The network mainly included the input neuron layer, output neuron layer, feature synapses layer, and suppression layer, which was inspired by Diehl's paper (Diehl and Cook, 2015). In EDHA and EvtSNN, the suppression layer was replaced by direct lateral inhibition, because it can be efficiently realized by the event-driven framework, and the network structure after modification is shown in Figure 6. The training code for Brian2 and BindsNET come from https://github.com/zxzhijia/Brian2STDPMNIST and https://github.com/BindsNET/bindsnet, respectively.

The MNIST dataset is image style data with 28 × 28 input pixels and 60,000 training samples (LeCun et al., 1998). Before being fed to the spike neural network, the image data should be encoded in spikes format. In EDHA, to reduce the number of input spikes and the amount of event-driven calculation, the time encoding method was adopted, and each pixel was coded into at most one pulse. The average pulse number of samples encoded in time encoding is far less than that of frequency coding, which can greatly reduce the calculation and speed up the simulation of the event-driven framework.

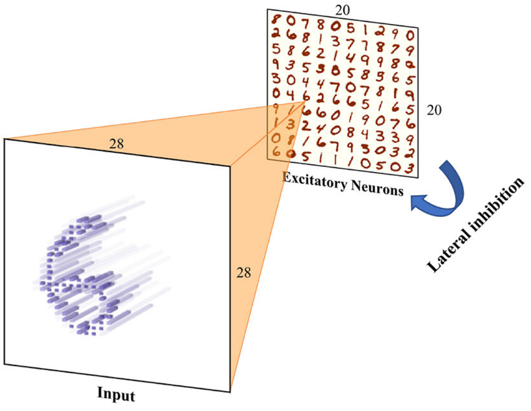


FIGURE 6
Unsupervised training network structure. Input images are encoded into spikes and then fed into the spiking neural network. The network includes an input layer and an output layer (excitation layer), there are synapses with learnable weights between them. Any neuron in the output layer has inhibitory connections with fixed weights to the others to achieve lateral inhibition.

Unsupervised training configuration, simulation time, and training results are shown in Table 3. The number of neurons in the input layer is the same as the number of image pixels,

TABLE 3 Comparison of simulators performance in task of MNIST unsupervised training (1 epoch).

Simulator	dt(ms)	Encode method	Sample duration (ms)	Average spikes	Accuracy (%)	Training time (s)
Brian2	0.5	Frequency	350	2285.64	87.90	1.538E+05
Brian2*	0.5	Frequency	350	2285.64	87.90	1.479E+04
BindsNET	0.5	Frequency	250	1632.60	90.10	8.274E+04
BindsNET-GPU	0.5	Frequency	250	1632.60	88.58	3.288E+04
EDHA	None	Frequency	350	2285.64	89.71	9.128E+03
EDHA	None	Time	100	136.54	88.86	6.418E+02
EvtSNN(ours)	None	Frequency	350	2285.64	89.19	4.790E+02
EvtSNN(ours)	None	Time	100	136.54	89.59	5.637E+01

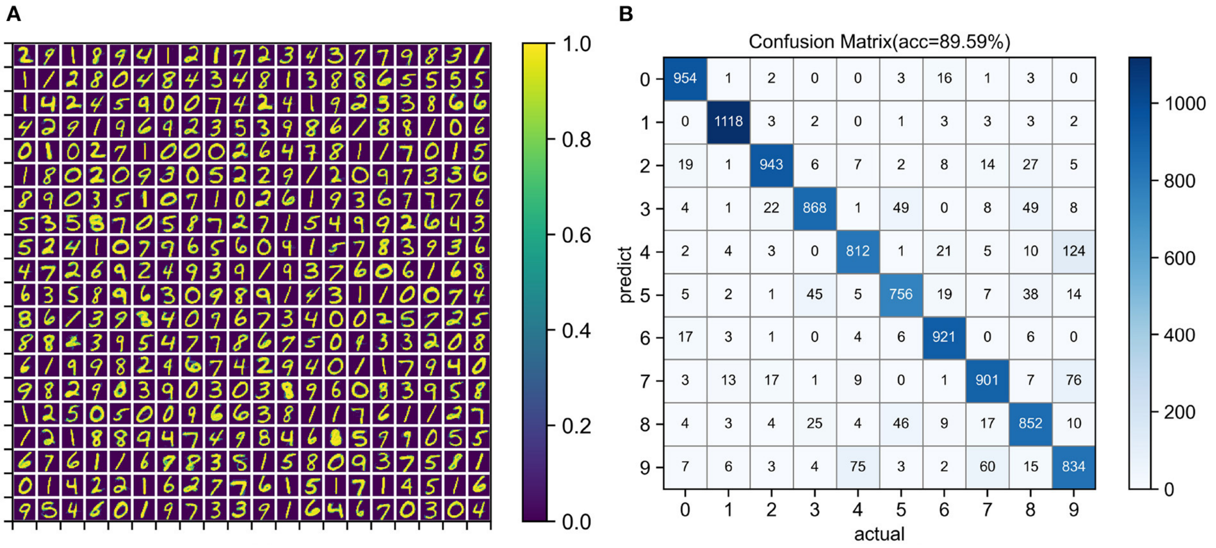


FIGURE 7
(A) Weight visualization after 1 epoch of unsupervised learning on the MNIST training set. **(B)** Confusion matrix for classification on the MNIST test set. The network size used for training and testing is 784–400.

which is 784. The more neurons in the output layer, the better the expressive ability of the network. When the output layer has 400 neurons, the network training speed is fast and the classification accuracy is acceptable. After one epoch of training, the test accuracy of four simulators was similar. Training on the EvtSNN framework took only 56 s, which was much faster than other frameworks. In this task, EDHA and EvtSNN use the same parameters, the latter is 11.4 times faster (with time encoding) and 19.1 times faster (with frequency encoding) than the former. Figure 7A is the weight visualization after 1 epoch of training. It can be seen that the features of the image have been learned by the network and stored in the weights. Figure 7B is the confusion matrix on the test set, with an accuracy of 89.59%.

After turning on the GPU, with the support of PyTorch for GPU, the speed of BindsNET-GPU is 2.5 times faster than that of BindsNET. In this experiment, it is necessary to judge whether the network has pulse output after inputting the sample, so Brian2GeNN needs to compile and run to obtain the simulation results. However, each compilation takes tens of seconds, which is a considerable overhead, so no relevant experiments have been done.

As shown in Figure 8, when the number of neurons in the output layer is increased, the classification accuracy of the network will be improved. Similar to the results of Diehl and Cook (2015), our network can achieve 95.16% accuracy when using 6,400 neurons, which demonstrates the feasibility of event-driven simulation combined with time coding.

5. Discussion

5.1. Simulation accuracy

In the experiment of Section 4.1, the membrane potential of neurons during the simulation of Brian2 and EvtSNN were

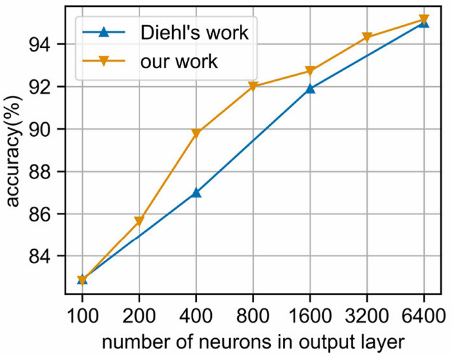


FIGURE 8
Classification accuracy of MNIST under different output layer scales. We use the EvtSNN framework combined with time coding for simulation, and the accuracy is no less than the Brian framework and frequency coding in Diehl's paper.

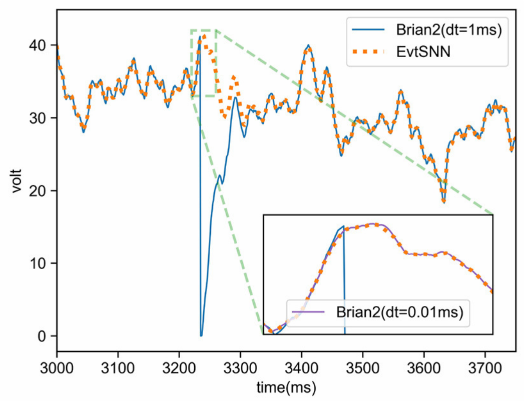


FIGURE 9
Voltage curves of neurons in Brian2 and EvtSNN simulations. The clock-driven framework Brian2 has higher simulation accuracy when using smaller time steps (e.g., dt = 0.01 ms), and its voltage is close to the result of the event-driven framework EvtSNN. When using a larger time step (e.g., dt = 1 ms), the simulation accuracy of Brian2 decreases and leads to changes in the firing pattern.

recorded and plotted, as shown in Figure 9. It can be seen that the voltage variation trend in the two simulations is basically the same, but sometimes there are small errors that affect spike delivery. Interestingly, even if the pulses do not match, the voltage will tend to be consistent after a while. This may be due to the decay of membrane potential. As time passes, the subsequent state is mainly affected by the input pattern rather than the initial state.

The simulation accuracy of the clock-driven method is limited by the time step, which may cause a small number of spike mismatches. However, in most cases, there are similar voltage curves in clock-driven (Brian2) and event-driven (EvtSNN) simulation, and the overall spike pattern is not much different, which means that the error of clock-driven simulation could be ignored many times. To sum up, there are some simulation errors in the clock-driven method, which can be ignored in most cases; the precision of the event-driven method can be very high, and it can use network sparsity to reduce the amount of calculation, which has a higher potential.

5.2. Quantitative analysis of sub-steps acceleration

In this section, a dynamic code analysis tool (Clion Profiler) was employed to count the time-consuming of each part. With the 10 kHz sampling frequency and 100 times repeated tasks (same as Section 4.1), the simulation time-consuming composition is shown in Figure 10. Population computing reduces the calculation time of sub-step (a) from 77.75 to 34.34 ms, while pre-filtering reduces the time consumption of sub-step (c) from 76.3 to 0.86 ms. It can be seen that the accelerating effect of pre-filtering is commendable so that the time consumption of sub-step (c) can be neglected. Even poor filtering in extreme cases does not slow down the overall simulation because its computational overhead is negligible.

5.3. Acceleration ability in multi-scale network

To measure the contribution of each innovation in different network scales, we combined the benchmark and ablation experiment, and the results are shown in Figure 11. Firstly, the acceleration effect of population calculation and pre-filtering under different network scales is relatively stable, reducing the time consumption by about 25 and 35%, respectively compared with EvtSNN (base). Secondly, when the average input frequency (*Fr*) of the neuron group is higher than the output frequency, the delayed update term can have some acceleration effect, otherwise, it will have a negative effect. In addition, after code optimization and rewriting, the speed of EvtSNN (base) is 2.6–4.5 times that of EDHA under the same calculation flow. Finally, EvtSNN using all optimization items is 2.9–14.0 times faster than EDHA.

5.4. Limitations

Of course, there are some limitations to our framework. First, as an event-driven framework, EvtSNN has poor model compatibility, requiring the derivation of time-domain

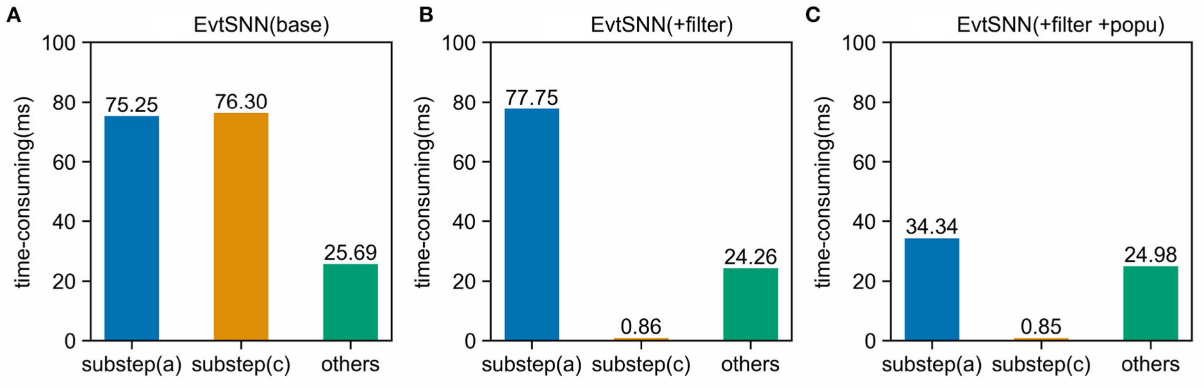


FIGURE 10
Time-consuming components in simulation. Using Pre-filtering (+filter) avoids unnecessary computation and can significantly reduce the time-consuming of sub-step (c). Enabling population computing (+popu) avoids repeated calculations and speeds up sub-step (a). **(A)** EvtSNN (base), **(B)** EvtSNN (+filter), **(C)** EvtSNN (+filter+popo).

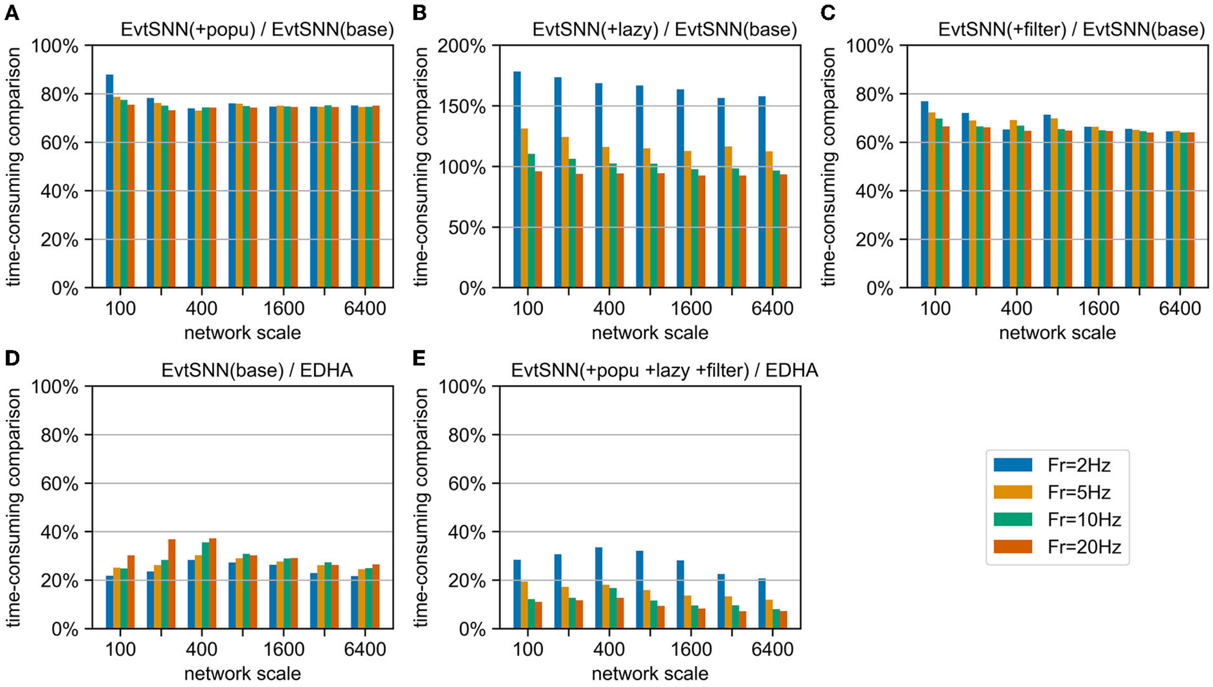


FIGURE 11
Time-consuming comparison chart. **(A–C)** EvtSNN time-consuming reduction after enabling popu, lazy, and filter, respectively. **(D,E)** Time-consuming comparison of EvtSNN and EDHA with none/all optimizations.

equations and the solution of spike firing time. Secondly, the pre-filtering formulation is only used for the neuronal model used. However, pre-filtering formulas for other neuronal models can draw on the derivation process in this paper (3.5).

6. Conclusion and future work

Based on the SNN event-driven framework EDHA, a simulator named EvtSNN is introduced. In this paper, two innovations are proposed to speed up the simulation without any accuracy loss. Firstly, repeated calculations are avoided according to the hidden information of the population. Secondly, the unnecessary calculation is filtered by the conditions derived from differential inequality. In the benchmark experiment, without the learning rule, the EvtSNN was the fastest in small network scale simulation (hundreds of neurons). EvtSNN always kept the lead when using the STDP learning rule. In the unsupervised training task of MNIST, EvtSNN only took 56 s to complete one epoch and reached 89.59% accuracy, which is 11.4 times faster than EDHA.

Our work can be further improved. Firstly, large-scale network simulation can be optimized in combination with the clock-driven method. Secondly, multithreading acceleration and parallel computing can be used with the single-layer parallel structure of Inception (Szegedy et al., 2015; Meng et al., 2021) for population-level concurrent acceleration.

Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author/s.

Author contributions

LM proposed the idea of EvtSNN. ZT implemented the code of the EvtSNN framework and performed the experiments. Both authors participated in the writing of the manuscript.

Funding

This work was sponsored by the Blue Project for the University of Jiangsu Province 2021.

Acknowledgments

We would like to thank all the members of FutureX LAB of Southeast University for their help and support.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., et al. (2015). TrueNorth: design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.* 34, 1537–1557. doi: 10.1109/TCAD.2015.2474396

Bautembach, D., Oikonomidis, I., Kyriazis, N., and Argyros, A. (2020). "Faster and simpler SNN simulation with work queues," in *Proceedings of the International Joint Conference on Neural Networks* (Glasgow: Institute of Electrical and Electronics Engineers Inc.). doi: 10.1109/IJCNN48605.2020.9206752

Davies, M., Srinivasa, N., Lin, T. H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Deng, S., and Gu, S. (2021). *Optimal Conversion of Conventional Artificial Neural Networks to Spiking Neural Networks*. Available online at: http://arxiv.org/abs/2103.00476

Diehl, P. U., and Cook, M. (2015). Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Front. Comput. Neurosci.* 9:99. doi: 10.3389/fncom.2015.00099

Gerstner, W., Kistler, W. M., Naud, R., and Paninski, L. (2014). *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge: Cambridge University Press. doi: 10.1017/CBO9781107447615

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Front. Neuroinformatics* 2:5. doi: 10.3389/neuro.11.005.2008

Hazan, H., Saunders, D. J., Khan, H., Patel, D., Sanghavi, D. T., Siegelmann, H. T., et al. (2018). BindsNET: a machine learning-oriented spiking neural networks library in python. *Front. Neuroinformatics* 12:89. doi: 10.3389/fninf.2018.00089

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 2278–2323. doi: 10.1109/5.726791

Li, S., Zhang, Z., Mao, R., Xiao, J., Chang, L., and Zhou, J. (2021). A fast and energy-efficient SNN processor with adaptive clock/event-driven computation scheme and online learning. *IEEE Trans. Circuits Syst. I* 68, 1543–1552. doi: 10.1109/TCSI.2021.3052885

Maass, W. (1997). Networks of spiking neurons: the third generation of neural network models. *Neural Netw.* 10, 1659–1671. doi: 10.1016/S0893-6080(97)00011-7

Masquelier, T., and Kheradpisheh, S. R. (2018). Optimal localist and distributed coding of spatiotemporal spike patterns through STDP and coincidence detection. *Front. Comput. Neurosci.* 12:74. doi: 10.3389/fncom.2018.00074

Meng, M., Yang, X., Bi, L., Kim, J., Xiao, S., and Yu, Z. (2021). High-parallelism Inception-like spiking neural networks for unsupervised feature learning. *Neurocomputing* 441, 92–104. doi: 10.1016/j.neucom.2021.02.027

Mo, L., Chen, X., and Wang, G. (2021). Edha: Event-driven high accurate simulator for spike neural networks. *Electronics* 10:2281. doi: 10.3390/electronics10182281

Moradi, S., Qiao, N., Stefanini, F., and Indiveri, G. (2018). A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Trans. Biomed. Circuits Syst.* 12, 106-122. doi: 10.1109/TBCAS.2017.2759700

Neftci, E. O., Mostafa, H., and Zenke, F. (2019). Surrogate gradient learning in spiking neural networks: bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Process. Mag.* 36, 51–63. doi: 10.1109/MSP.2019.2931595

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). "PyTorch: an imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems, Vol. 32* (Vancouver, BC).

Pei, J., Deng, L., Song, S., Zhao, M., Zhang, Y., Wu, S., et al. (2019). Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* 572, 106–111. doi: 10.1038/s41586-019-1424-8

Pham, Q. T., Nguyen, T. Q., Hoang, P. C., Dang, Q. H., Nguyen, D. M., and Nguyen, H. H. (2021). "A review of SNN implementation on FPGA," in *2021 International Conferenceon Multimedia Analysis and Pattern Recognition, MAPR 2021 - Proceedings* (Hanoi: Institute of Electrical and Electronics Engineers Inc.). doi: 10.1109/MAPR53640.2021.9585245

Sengupta, A., Ye, Y., Wang, R., Liu, C., and Roy, K. (2019). Going deeper in spiking neural networks: VGG and residual architectures. *Front. Neurosci.* 13:95. doi: 10.3389/fnins.2019.00095

Shen, J., Ma, D., Gu, Z., Zhang, M., Zhu, X., Xu, X., et al. (2016). Darwin: a neuromorphic hardware co-processor based on Spiking Neural Networks. *Sci. China Inform. Sci.* 59, 1–5. doi: 10.1007/s11432-015-5511-7

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8:e47314. doi: 10.7554/eLife.47314.028

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., et al. (2015). "Going deeper with convolutions," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Boston, MA: IEEE Computer Society), 1–9. doi: 10.1109/CVPR.2015.7298594

frontiers | Frontiers in Neuroinformatics

TYPE Original Research
PUBLISHED 26 October 2022
DOI 10.3389/fninf.2022.883742

OPEN ACCESS

EDITED BY
Sharon Crook,
Arizona State University, United States

REVIEWED BY
Patrick K. Quoika,
Technical University of Munich,
Germany
Hans Ekkehard Plesser,
Norwegian University of Life Sciences,
Norway
Neslihan Serap Sengor,
Istanbul Technical University, Turkey

*CORRESPONDENCE
Erik De Schutter
erik@oist.jp

†These authors share first authorship
‡These authors share senior authorship

RECEIVED 25 February 2022
ACCEPTED 30 September 2022
PUBLISHED 26 October 2022

CITATION
Chen W, Carel T, Awile O, Cantarutti N, Castiglioni G, Cattabiani A, Del Marmol B, Hepburn I, King JG, Kotsalos C, Kumbhar P, Lallouette J, Melchior S, Schürmann F and De Schutter E (2022) STEPS 4.0: Fast and memory-efficient molecular simulations of neurons at the nanoscale.
Front. Neuroinform. 16:883742.
doi: 10.3389/fninf.2022.883742

STEPS 4.0: Fast and memory-efficient molecular simulations of neurons at the nanoscale

Weiliang Chen[1†], Tristan Carel[2†], Omar Awile[2], Nicola Cantarutti[2], Giacomo Castiglioni[2], Alessandro Cattabiani[2], Baudouin Del Marmol[2], Iain Hepburn[1], James G. King[2], Christos Kotsalos[2], Pramod Kumbhar[2], Jules Lallouette[1], Samuel Melchior[2], Felix Schürmann[2‡] and Erik De Schutter[1*‡]

[1]Okinawa Institute of Science and Technology Graduate University (OIST), Okinawa, Japan, [2]Blue Brain Project, École Polytechnique Fédérale de Lausanne (EPFL), Geneva, Switzerland

Recent advances in computational neuroscience have demonstrated the usefulness and importance of stochastic, spatial reaction-diffusion simulations. However, ever increasing model complexity renders traditional serial solvers, as well as naive parallel implementations, inadequate. This paper introduces a new generation of the STochastic Engine for Pathway Simulation (STEPS) project (http://steps.sourceforge.net/), denominated STEPS 4.0, and its core components which have been designed for improved scalability, performance, and memory efficiency. STEPS 4.0 aims to enable novel scientific studies of macroscopic systems such as whole cells while capturing their nanoscale details. This class of models is out of reach for serial solvers due to the vast quantity of computation in such detailed models, and also out of reach for naive parallel solvers due to the large memory footprint. Based on a distributed mesh solution, we introduce a new parallel stochastic reaction-diffusion solver and a deterministic membrane potential solver in STEPS 4.0. The distributed mesh, together with improved data layout and algorithm designs, significantly reduces the memory footprint of parallel simulations in STEPS 4.0. This enables massively parallel simulations on modern HPC clusters and overcomes the limitations of the previous parallel STEPS implementation. Current and future improvements to the solver are not sustainable without following proper software engineering principles. For this reason, we also give an overview of how the STEPS codebase and the development environment have been updated to follow modern software development practices. We benchmark performance improvement and memory footprint on three published models with different complexities, from a simple spatial stochastic reaction-diffusion model, to a more complex one that is coupled to a deterministic membrane potential solver to simulate the calcium burst activity of a Purkinje neuron. Simulation results of these models suggest that the new solution dramatically reduces the per-core memory consumption by more than a factor of 30, while maintaining similar or better performance and scalability.

KEYWORDS

STEPS, stochastic simulation, molecular neuroscience, HPC, supercomputing

1. Introduction

For several decades computational modeling has progressively proven its importance in neuroscience research, covering a wide range of research domains and disciplines: from sub-cellular molecular reaction-diffusion dynamics to whole-brain neural network simulations. Breakthroughs in experimental methods and community-driven data sharing portals have significantly increased the amount of available experimental data, enabling the advance of complex data-driven modeling and analysis. These efforts are further enhanced by large collaborative projects such as the US BRAIN initiative (Insel et al., 2013), and the EU Human Brain Project (Markram et al., 2011; Amunts et al., 2016, 2019), where complex computational modeling plays an essential role. The rapid progress of neuroscience modeling brings critical advances to our understanding of neuronal systems, yet unprecedented challenges to simulator software development have emerged from two primary directions: first, the need to simulate neuronal functionalities across multiple spatio-temporal scales, and second, the requirement of simulating such systems with extraordinary efficiency.

1.1. The STEPS project and its applications

The STochastic Engine for Pathway Simulation (STEPS) project has evolved following the above trends over the years. The STEPS project started as a mesoscopic scale stochastic reaction-diffusion solution (Hepburn et al., 2012) driven by a spatial variant of the well-known Gillespie Stochastic Simulation Algorithm (SSA) method (Gillespie, 1977). Over the years, serial STEPS has contributed to a wide range of research domains, such as studies on long-term depression in cerebellar Purkinje cells (Antunes and De Schutter, 2012; Zamora Chimal and De Schutter, 2018), viral RNA degradation and diffusion (Schelker et al., 2016), longitudinal anomalous diffusion in neuron dendrites (Mohapatra et al., 2016), and calcium signaling in astrocytes (Denizot et al., 2019). We gradually expanded STEPS to support electrical potential calculation on tetrahedral meshes with the EField solver (Hepburn et al., 2013), allowing combined simulations of reaction-diffusion and membrane potential dynamics on a single mesh reconstruction of neuronal morphology. This solution was important for research that showed that stochastic activation of ion channels, in particular calcium-activated potassium channels, produces significant variability in Purkinje cell dendritic calcium spike shape (Anwar et al., 2013). However, it was soon clear to us that the serial nature of STEPS was the major bottleneck for simulating such complicated models; even a sub-branch of a Purkinje neuron often took weeks to complete one realization of 500 ms biological time. This issue was partially addressed in STEPS 3.0 by introducing the parallel operator splitting method to the reaction-diffusion solution (Hepburn et al., 2016; Chen and De Schutter, 2017), which aided research such as platform development for automatic cancer treatment discovery (Stillman et al., 2021). A parallel EField implementation supported by the PETSc library (Abhyankar et al., 2018) was added to STEPS 3.1. The parallel solution dramatically improved performance by thousand folds compared to the serial counterpart, making it possible to model a complete neuron with detailed morphology and channel mechanisms (Chen et al., 2022).

1.2. The need of a new parallel solver

Moving to parallel STEPS has greatly improved performance compared to the serial solution. However, as the hardware and software of high-performance computing have advanced in recent years, noticeable bottlenecks have been observed in modeling applications with STEPS. The main objective of this article is to identify these bottlenecks and address them with a new parallel implementation.

For many scientific applications, the memory capacity of High-Performance Computing (HPC) systems is one of the main constraints for running simulations at scale. A large number of today's HPC systems have about 2~3GB of main memory per core (Zivanovic et al., 2017). This is an improvement compared to previous BlueGene-like systems where memory capacity is typically ~1GB per core. Current systems are increasingly heterogeneous with the use of accelerators such as GPUs. The memory capacity of such a system is significantly lower compared to what is commonly available on host CPUs. In the case of Intel Knights Landing processors (Sodani et al., 2016), the total capacity is approximately 0.2GB per core. The next generation of processors such as Intel Sapphire Rapids will most likely have a per-core memory capacity similar to the current generation. This poses a significant challenge to application developers: on the one hand the raw computing power is significantly increasing with architectures like GPUs, while on the other hand maintaining a low memory footprint becomes increasingly important to achieve better performance.

One major limitation of the existing parallel implementation in STEPS comes from the mesh data architecture inherited from the serial solution. While bridging the gap between serial and parallel STEPS and making many non-parallel components reusable, the serial nature of the design requires the complete data of the whole mesh and the molecule state of each mesh element to be stored in every computing core. This poses a hard limit on the maximum model size determined by the per-core memory availability, the model complexity, and the mesh size. Thanks to support from the parallel solver, realistic simulations with a large number of chemical reactions for a great period of biological time can now be accomplished in a reasonable computing time. However, this in turn raises research interests in even more complicated models and more realistic morphologies, reaching the limits of the implementation. The memory constraints in modern HPC systems further amplify such limitations.

The solution to this issue is a new parallel implementation constructed on the foundation of a sophisticated distributed mesh library, Omega_h (Ibanez and Roberts, 2018). Thanks to the distributed nature of the mesh library and the redesigns of other STEPS components, we are able to dramatically reduce the memory footprint of the simulation while maintaining similar or better performance and scalability.

1.3. Other solutions for spatial reaction-diffusion simulations

Traditionally, spatial reaction-diffusion simulation solutions are divided into two major categories, voxel-based and off-voxel particle-based. Voxel-based simulators divide the geometry into small voxels, where the Reaction-Diffusion Master Equation is solved by variants of the Gillespie SSA method (Gillespie, 1977). Example simulators in this category include STEPS (Hepburn et al., 2012), MesoRD (Hattne et al., 2005), and NeuroRD (Oliveira et al., 2010). Off-voxel particle-based solutions represent each molecule in the system individually as sphere-like physical entities, track the Brownian motion of each molecule in a continuum space, and simulate molecular reactions caused by collisions. Example simulators of this category include Smoldyn (Andrews and Bray, 2004), MCell (Kerr et al., 2008), and ReaDDy (Schöneberg and Noé, 2013). Solutions between these two major categories also exist, for instance, Spatiocyte (Arjunan and Tomita, 2010), which simulates individual molecule particle movement with reactions on a hexagonal close-packed lattice.

Some early attempts of parallel spatial reaction-diffusion simulation solutions have been reviewed in Chen and De Schutter (2017). Here we report the latest developments in the field since then. In the voxel-based simulator domain, apart from STEPS 3.x in our previous report, Patoary et al. (2019) further optimized the multi-threading Neuron Time Warp solution, and achieved 5.5x speedup with 7 logical processors, comparing to the single logical processor simulation. In the off-voxel particle-based domain, the ReaDDy 2 simulator reported an approximately sixfold speedup with 11 threads, using single thread simulation as the baseline (Hoffmann et al., 2019). The parallel implementation of Spatiocyte, pSpatiocyte (Arjunan et al., 2020), reported a 7,686x speedup with 663,552 cores on the RIKEN K computer, compared to the 64 core baseline simulation. It is worth noting that direct performance comparisons of these simulators are often challenging, as different theoretical solutions and model abstractions are applied in the implementations.

1.4. Naming conventions and the structure of the article

To avoid confusion, we will hereby call the non-parallel, spatial STEPS solver "serial STEPS," the existing parallel implementation reported in Chen and De Schutter (2017) "STEPS 3," and the new parallel implementation supported by Omega_h that we introduce in this paper "STEPS 4." Note that serial STEPS, STEPS 3 and STEPS 4 are all integrated solutions of the STEPS 4.0 release, and the users are free to choose any of them for their simulations based on the research requirements.

In Section 2, we first describe our design principles and the implementation details of STEPS 4, and then introduce some software engineering techniques applied to the overall STEPS project for maintainability and efficiency improvements. In Section 3, we present the validations of the implementation with a series of well-established models, followed by performance and scalability analysis of results. In Section 4, we further discuss the achievements, limitations and potential solutions of this study, as well as the future development plans for STEPS 4 and the STEPS project in general.

2. Methods

The STEPS development project follows three major methodological principles. First, it aims toward the researchers. STEPS attempts to provide a user-friendly modeling interface, and to progressively reduce the need for manual coding efforts with implementations of auxiliary supports. Second, we focus on improving its performance, as this determines if the simulations can be completed within the expected research time frame.

Third, it aims to be future-proof. Since the first public release, the STEPS project has more than 10 years of history. Over the years, many new standards and solutions in programming and software engineering have been established and become the new standard in software development. Some of them have been adopted in previous STEPS development, but more work is still required to ensure that the software development infrastructure is ready for future project expansions. The following sections detail how these principles are practically applied in the project.

2.1. Code modernization and future-proofing

Although STEPS introduced many new features and additions in the following years since its first release in 2012 the core coding components and style remained relatively unchanged. With this in mind, in this work we have implemented various changes in STEPS in general and adopted modern software design principles to STEPS 4 in particular. All these changes have the aim to reduce bugs, improve maintainability and usability of the code and increase the performance of time-critical data structures and routines.

First, we have adopted the C++17 standard for STEPS. This allowed us to take advantage of modern programming language features, increasing code expressiveness and compactness through meta-programming techniques such as SFINAE (Substitution Failure Is Not An Error). We have also removed raw pointers in favor of references and other safer data-passing and access strategies provided by the C++ standard and the Guidelines Support Library[1]. Second, we have reduced code branching and indirections using meta-programming techniques, which streamline code execution. Third, when choosing container data structures we avoid C++ Standard Template Library (STL) associative containers, which are known to be very inefficient in terms of memory management and performance. The intrinsic arborescent memory layout of `std::map` brings very poor data locality that makes it unusable in computational kernels. Using `std::unordered_map` is a better choice since it uses a contiguous arrays to store the hash values, but its implementation relies on `std::list` to store the values for backward compatibility reasons of the API, which brings back a data locality issue. Because the dataspace of the keys in STEPS 4 is made contiguous, the best data structure based on the STL is `std::vector<std::vector<>>` because the data access is O(1) and data locality, though still flawed, is a bit better than `std::unordered_map` since the values of a key are stored contiguously. Instead, we have designed a new optimized data structure to maximize both access and data locality, the `flat-multimap`.

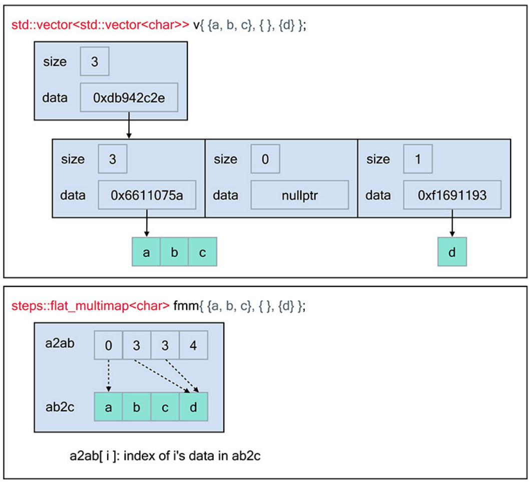

std::vector<std::vector<char>> v{ {a, b, c}, { }, {d} };
size
3
data
0xdb942c2e
size
3
data
0x6611075a
size
0
data
nullptr
size
1
data
0xf1691193
a
b
c
d
steps::flat_multimap<char> fmm{ {a, b, c}, { }, {d} };
a2ab
0
3
3
4
ab2c
a
b
c
d
a2ab[i]: index of i's data in ab2c

FIGURE 1
Memory layout of the `flat-multimap` container in comparison with a `vector of vectors` container constructed using the Standard Template Library to store the following key-values: `0 → [a, b, c], 2 → [d]`. The `flat-multimap` class relies on 2 member variables, `a2ab` and `ab2c`. *a* is the top element index, *ab* is an index to retrieve the data of *a* in `ab2c`. Data are stored contiguously in `flat-multimap` to reduce heap fragmentation and increase data locality. In contrast, data stored in the STL container are more fragmented. With `flat-multimap`, the values of key a are stored in the range `ab2c[a2ab[a]]` and `ab2c[a2ab[a + 1]] - 1`. In this example, values of key 0 are in ab2c[0, 2] i.e [a, b ,c], key 1 has no value since a2ab[1] == a2ab[2], finally values of key 3 are in ab2c[3, 3] i.e [d].

Figure 1 illustrates the memory layout of the `flat-multimap` container in comparison with a naive STL implementation by employing a vector of vectors data structure. The STL implementation exhibits poor data locality as the number of heap allocations required is O(n) whereas `flat-multimap` is O(1) as it always requires 2 allocations. This gives `flat-multimap` several advantages over the STL counterpart. First, it reduces heap fragmentation in the memory. In addition, as the data are stored contiguously in `flat-multimap`, data locality is greatly improved and the solution is more cache-friendly. In exchange, the `flat-multimap` container requires a fixed size and shape upon creation, which can not be changed throughout the simulation. However, this restriction is mostly irrelevant to STEPS 4, as the sizes of the majority of data are determined and fixed by the model.

With the increased complexity of a software, there is a growing concern about introducing bugs in the code that remain undetected. In the best case, these bugs will lead to crashes during runtime. In the worst case, they may silently introduce

[1] https://github.com/microsoft/GSL

erroneous results and non-reproducible behavior. Although STEPS runs an extensive validation set to try and ensure this doesn't happen, it is difficult to make sure that every base is covered by such efforts. In an attempt to address this issue at least partially, we introduced C++ vocabulary types meant to indicate to the compiler the different entities used in a STEPS simulation (e.g., `species`, `membrane`, `channel`, `patch`, etc., but also `tetrahedron`, `triangle`, etc.).

A vocabulary type is a type whose name carries a specific meaning in addition to its data. For example, an instance of a class `Width` made of a floating-point value carries both the value and the nature of this value, in opposition to fundamental types like integers or floating-points. Usually fundamental types don't tell much about the meaning of their instances. Vocabulary types can be used to create interfaces comprehensible, expressive, and robust. For instance, vocabulary types can improve functions like below:

```
void process_local_tetrahedron(int index);
```

In the signature of this function, most of the information about the parameter is carried by the variable name and function name, which the compiler cannot use. For the compiler, `process_local_tetrahedron` is only a function that takes a 32 bits integer in parameter. For the developer, this integer is an index of a tetrahedron, local to the current process. Vocabulary types allow us to transfer information traditionally held by the name of the symbols to the typing system by rewriting the signature of the function like this:

```
struct local_tetrahedron_id {
    int value{};
};
void process(local_tetrahedron_id entity);
```

Thus, the compiler is now able to report an issue when the index of one type is erroneously passed to a function expecting another type.

Furthermore, by ensuring the code compiles with GCC, clang, AppleClang and Intel OneAPI, we ensure that language and system compatibility is maintained, further increasing code safety. Numerous compilation flags have been added into our build system, which allow us to spot and fix potential issues in the code early in the development process. We have also moved to a more modular build design where features can be enabled *via* build configuration flags, which also benefits overall software architecture.

Finally, we have tackled software sustainability beyond code modernization. To improve developer confidence and bug detection we have added continuous integration (CI) pipelines into the review process. Proposed patches are automatically built and tested before they can be merged into the development trunk. We have also created a STEPS package for the Spack (Gamblin et al., 2015) package manager. This not only adds a software distribution channel for HPC systems but also provides the developers with a comprehensive build environment that allows them to conveniently test STEPS with various dependency versions and build options. The choice of the underlying libraries (see Section 2.2.2) plays an important role in ensuring that STEPS remains well maintainable, and easily extensible toward new features and use-cases while continuing to support the latest hardware architectures and parallel programming paradigms.

2.2. Implementing a parallel solver with distributed mesh backend

2.2.1. Implementation criteria

To be able to make informed choices about the STEPS 4 implementation, we set early in the development a number of criteria by which to make decisions. Clearly the first and most important criterion is simulation runtime. The goal of the STEPS 4 implementation is to develop a new efficient solution for large-scale modeling with complex geometries. From a user's perspective, the most straightforward and important concern is time-to-solution, how fast a simulation reaches a desired stopping time. For parallel simulations, another important concern is scalability. In high performance computing, parallel scalability is commonly described by two notions, strong scaling and weak scaling. The former describes runtime performance at increasing number of cores and a fixed problem size, while the latter scales the problem size with the number of cores. In practice, the problem size of a STEPS production simulation is often determined by the source materials. Thus, we focus on strong scaling as our parallel performance criterion. STEPS 4 is designed mainly for simulations that run on high performance computing clusters. As mentioned previously, one key characteristic of modern clusters is the large amount of computing cores together with the limited amount of per-core memory, thus memory footprint management is essential to support large scale simulations. We regard it as our third implementation criterion.

These criteria often affect each other in a simulation. For instance, the reduction of memory footprint could substantially improve the efficiency of memory caching, and further improve scalability. Therefore, we do not focus on an individual criterion, but consider them as a whole when making implementation decisions.

2.2.2. Prototyping STEPS 4

Choosing the distributed mesh library with the most suitable abstractions and best performance properties is vital for the success of STEPS 4. This library is the backbone of the whole implementation, providing fundamental data layout and access functionalities, which tightly associates with the criteria discussed above. Besides performance considerations, from a developer's perspective, the mesh library should also provide a rich and extendable API that can be connected with other STEPS components with ease. Furthermore, while STEPS 4 mainly targets CPUs, the algorithms themselves could in principle be implemented on other hardware architectures and the right abstraction layer should allow a relatively smooth transition toward supporting shared-memory parallelism or GPUs.

To investigate the advantages and drawbacks of different distributed mesh libraries, we used them to implement a series of stand-alone mini-applications to cover a wide range of STEPS functionalities, from simple mesh importing and exporting in a distributed manner, to a functional reaction-diffusion solution integrated with various validations and use case models. These mini-applications were gathered in a library named Zee. Using the Zee library we were able to investigate how different components of STEPS, for example, the operator splitting method, can be implemented on top of different distributed mesh libraries, and to investigate the coding flexibility as well as the performance of our implementations. These investigations provided us essential insight for the choice of a suitable distributed mesh library for STEPS 4, and prototypes for the actual implementation.

We put our evaluation focus on two distributed mesh library candidates, Omega_h (Ibanez and Roberts, 2018) and the DMPlex module from the PETSc library (Abhyankar et al., 2018). Both libraries provide very well-suited features and showed promising performance. The choice of library, however, depends on factors beyond pure technical considerations. On the one hand, PETSc seemed a natural choice since STEPS 3's parallel EField solver already uses PETSc as a backend. Choosing PETSc's DMPlex would eliminate the need for an extra library, as well as the associated data conversions and transfers between libraries. Additionally, PETSc is an extremely well-known and supported library with a large active community. On the other hand, DMPlex is a minor component in the PETSc framework, supported only by few developers and with a small user community. Since the Zee mini-applications revealed that not all functionalities required in STEPS 4 are currently present in DMPlex, and some of which have considerably low priority on the PETSc development roadmap, our choice had to fall on Omega_h.

Omega_h is a C++14 library providing highly-scalable distributed adaptive meshing primitives. Distributed-memory parallelism is natively supported through Message Passing Interface (MPI), while on-node shared-memory parallelism is supported *via* Kokkos (Trott et al., 2022), a C++ library that provides abstractions for parallel execution with OpenMP on CPU and CUDA on GPU. Omega_h ensures a fully deterministic execution. Given the same mesh, global numbering and size field, mesh operations produce the exact same results regardless of parallel partitioning and ordering. This does, however, not extend to changing compilers or hardware. Omega_h is being actively developed and is used for a number of ongoing projects. Moreover, its codebase being much smaller than PETSc, it allowed us to have a comprehensive overview of its capabilities. Despite the lack of documentation, the source code is concise and self-explanatory. Contributing to Omega_h has been much easier than it would have been with PETSc. We were for instance able to add support to the MSH multi-part file format version 4 into Omega_h quite easily.

Omega_h's modern C++ interface was a significant advantage over PETSc as its ease of use allowed us to implement compact yet expressive mini-applications very quickly. We found that the C-oriented API of PETSc makes the library hard to comprehend and is much more error prone than Omega_h's. Additionally, the data management policies of DMPlex are quite complex and require a deep knowledge of PETSc internals as entity data is not directly exposed to the user as it is in Omega_h. This leads to the code being more cluttered and difficult to maintain.

2.2.3. Solver components and the simulation core loop

Fundamentally, STEPS 4 adopts the same operator splitting solution for reaction-diffusion simulation as in STEPS 3 (Hepburn et al., 2016), but with significant differences in the implementation details due to its distributed nature and other optimization goals.

In STEPS 3, the data and operators are intermixed in the solver, and data that are associated may be stored sparsely due to the data structures inherited from previous STEPS implementations. For instance, the molecule state of a tetrahedron and the states of its neighboring tetrahedrons may be stored far away from each other in memory. This is because the molecule state is stored sparsely in individual tetrahedrons together with other data such as mesh connectivity and kinetic processes. This means operator visits to the molecule state often require significant address jumps across memory, decreasing cache efficiency. The bundle of operators and data also make their optimization cumbersome, as new operator solutions or new data structures can not be implemented directly as independent alternatives.

In STEPS 4 one critical implementation change is the separation and encapsulation of different solver components. The two major components are: SimulationData, the data that represents the current state of the simulation, and the operator collection, which are applied to the data so that the simulation evolves to the next state. The simulation state consists of the molecule state M, where the distribution of molecule species is stored and updated, the kinetic process state K, which stores and maintains all kinetic processes such as reactions and surface reactions in the simulation and the information of each kinetic process, including the propensity and update dependencies, and finally the voltage state V of the mesh if voltage-dependent surface reactions and channels are expressed in the model. The voltage state contains the electrical potential at each vertex of the mesh, as described in Hepburn et al. (2013). The operator collection consists of the operators needed for each step of the simulation core loop, mainly, the reaction SSA operator, the diffusion operator and the PETSc EField operator. As the state data is encapsulated and accessed by operators *via* a unified interface, new operators can be easily developed and provided to the solver as alternative solutions. The encapsulation of simulation states M, K and V also allows the state data to be stored contiguously in memory space, thus improving caching efficiency of the solution.

As mentioned in Section 1, while the kinetic processes and their dependency graphs are partitioned and distributed among computing cores in STEPS 3, all mesh elements and their molecule states are duplicated, leading to high memory consumption and communication overhead when dealing with large scale models. In STEPS 4, the mesh itself is partitioned and distributed, thus each computing core only operates on the data for the sub-domain problem of its associated partition. Ghost layers were implemented for partition boundaries so that simulation states of the boundaries can be synchronized through regular data exchange. This solution ensures a relatively consistent memory footprint for any given sub-domain problem with a fixed partition size, regardless of the size of the overall problem.

Figure 2 schematically illustrates the simulation core loop. When the simulation enters the core loop that advances the simulation state from time T_{start} to $T_{end} = T_{start} + \Delta T$, the simulation period is divided into multiple time windows, whose period is either determined by a user-defined EField period ΔT_{EField} if the EField operator is involved, or equals ΔT otherwise. We call this the EField time window. Each EField time window is then further subdivided by a period of ΔT_{RD}, where ΔT_{RD} is determined by the mesh and the diffusion constants of the simulated model. This is the reaction-diffusion (RD) time window.

At the beginning of each RD time window, the reaction SSA operator is applied to the simulation data repeatedly. Each time, the SSA operator first randomly selects a kinetic process event *kp* from the kinetic process state K and the event time Δt according to the SSA solution described by the operator and the propensities of the kinetic processes. It then applies the molecule changes caused by the event to the molecule state M, updates the propensities of all kinetic processes that depend on *kp* in K, and advances the simulation state time for Δt. The SSA iteration stops when the state time reaches the end of the RD time window. As explained in Hepburn et al. (2016) and Chen and De Schutter (2017), the SSA operator is executed independently by each MPI rank without the need for any communication.

At the end of the RD time window, the diffusion operator computes the number of molecules that should diffuse out of each tetrahedron for the time window period ΔT_{RD}. For this calculation the diffusion rates of each diffusive molecule species must be taken into account. The operator then removes them from their original tetrahedrons and redistributes them to their target tetrahedrons. The redistribution is stored in a delta molecule state ΔM, which is then synchronized by Omega_h across all simulation ranks. After the synchronization, each rank applies the changes in ΔM to M for the tetrahedrons it owns, and updates the propensities that are affected by the changes. This completes the operations in a single RD time window.

The solver then repeats this process until the state time reaches the end of the EField time window, at which point the EField operator evolves the voltage state V for the period of ΔT_{EField}, based on the electric currents computed from M and K. This concludes the operations in a EField time window.

If $\Delta T_{EField} < \Delta T$ the EField time window process is repeated, otherwise the simulation core loop is completed and the user regains the simulation control for data inquiry.

2.2.4. Optimization on kinetic process dependency graph

A kinetic process dependency graph describes the update dependency of each kinetic process in the system. Technically, it returns a list of kinetic processes whose propensities must be updated when a certain kinetic process is selected and applied by the SSA operator. Under the operator splitting framework, the reactions in each tetrahedron are independent until the diffusion operator is applied. Therefore, it is possible to divide the dependency graph into independent subgraphs and apply the SSA operator to them separately. This independent graph optimization further compresses the targeting domain of the SSA operator, providing potentially substantial gains in simulation performance.

An example of the optimization for a small model is depicted in Figure 3. This model has two tetrahedrons, each with three volume reactions. One tetrahedron also contains four surface reactions. Each colored node in the figure represents a kinetic process. An arrow goes from one node to the other if the occurrence of an event of the first entails a change in propensity of the second. The whole dependency graph of the model can therefore be subdivided into two independent subgraphs, in red and blue as shown in the figure. Each subgraph can be evolved freely by a SSA operator without the other's interference in a RD time window period.

Note that this optimization heavily relies on the hit rate of drawing SSA events that take place within the time window in each subgraph. Its advantage diminishes and eventually becomes a burden if most of the drawn events happen beyond the time window and are discarded. This hit rate positively correlates to the duration of the time window, the molecule concentrations and the reaction rates. Therefore, this optimization favors simulations with a large RD time window, high molecule concentrations and highly active reactions, but

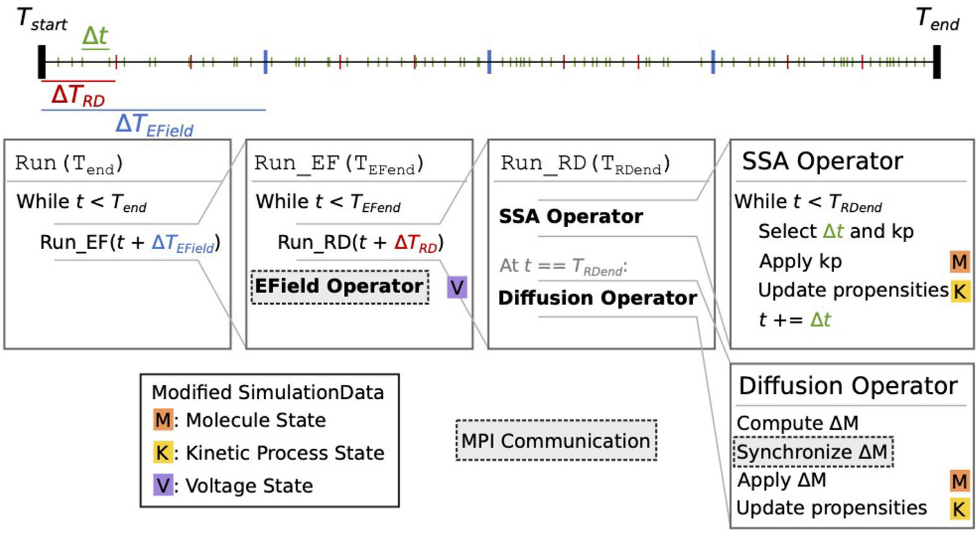


FIGURE 2
Schematic representation of the STEPS 4 simulation core loop. In this example, when running the simulation from T_{start} to T_{end}, the simulation time is first split into ΔT_{EField} time windows (blue ticks). Each ΔT_{EField} time window is further subdivided into ΔT_{RD} time windows (red ticks). Kinetic process events are represented as green ticks, their number in each time window depends on the propensities of the reactions. Current state time is denoted t. The leftmost `Run(T`$_{end}$`)` box is the entry point into the core loop, it splits the time in ΔT_{EField} time windows. The second box (`Run_EF(T`$_{EFend}$`)`) runs a full EField time step until T_{EFend}, the end-time that was passed from the first loop. It first subdivides the EField time window in ΔT_{RD} time windows and calls `Run_RD` for each one. When state time t reaches T_{EFend}, it runs the EField operator. The third box (`Run_RD(T`$_{RDend}$`)`) represents one RD time window, it is composed of the SSA operator and the diffusion operator. The SSA operator first selects and applies kinetic processes until the state time reaches T_{RDend}; it's in this loop that the state time is updated. The diffusion operator is then applied: it computes the changes ΔM to the molecule state and applies them. Each of these steps can involve the modification of the simulation data. When it does, a letter with a colored background is present to its right. The letter M with an orange background signifies that this operation modifies the molecule state; the letter K with a yellow background signifies that it modifies the kinetic process state; and the letter V on a purple background signifies that it modifies the voltage state. Finally, steps with a darker background and a dashed outline involve MPI communication between processes.

disfavors simulations with a small time window, low molecule concentrations and less active reactions.

In the STEPS 4 parallel scheme, a simulation core loop is completed after every MPI process finishes its operations, therefore the overall performance of the solution is determined by the slowest computing core. Due to concentration gradients as well as spatial variations of channel density in the model, large scale simulations with complex morphology may exhibit high variability of event drawing hit rate among computing cores. In this case, switching off the independent graph optimization is preferred.

2.2.5. EField solver improvements

Generally, in order to obtain the most accurate results and the best performance, the solver and preconditioner in PETSc need to be tailored to the particular simulation. Previously STEPS 3 used by default the Conjugate Gradient iterative solver (CG) and the Geometric Algebraic Multigrid (AMG) preconditioner. However, performance tests have consistently shown that they do not scale well for large problems. Thus, for STEPS 4 we replaced solver and preconditioner with the widely used Pipelined Conjugate Gradient method (KSPPIPECG) and the Point Block Jacobi preconditioner (PCPBJACOBI), respectively. The same configuration was also applied to STEPS 3 as the new default option. We have not performed a thorough investigation on solvers and preconditioners as it was out of the scope of the present paper.

Another improvement is the distribution of PETSc vectors and matrices for the EField computation. STEPS 3 distributes them equally among computing cores without considering if the mesh elements represented by the matrix partition are owned by the same core. This causes owner mismatches between the EField solution data and the reaction-diffusion solution data, which need to be resolved by expensive cross process data exchanges. In order to avoid this issue, STEPS 4 assembles the vectors and matrices so that each processor only takes care of the degrees of freedom corresponding to the sub-part of the mesh that is owned locally on this processor. This greatly increases data locality and performances since reaction-diffusion and the EField solvers exchange data only locally.

2.2.6. Coupling with other STEPS components

Setting up a simulation in STEPS 4 is mostly done in the same way as in STEPS 3: it involves the declaration of

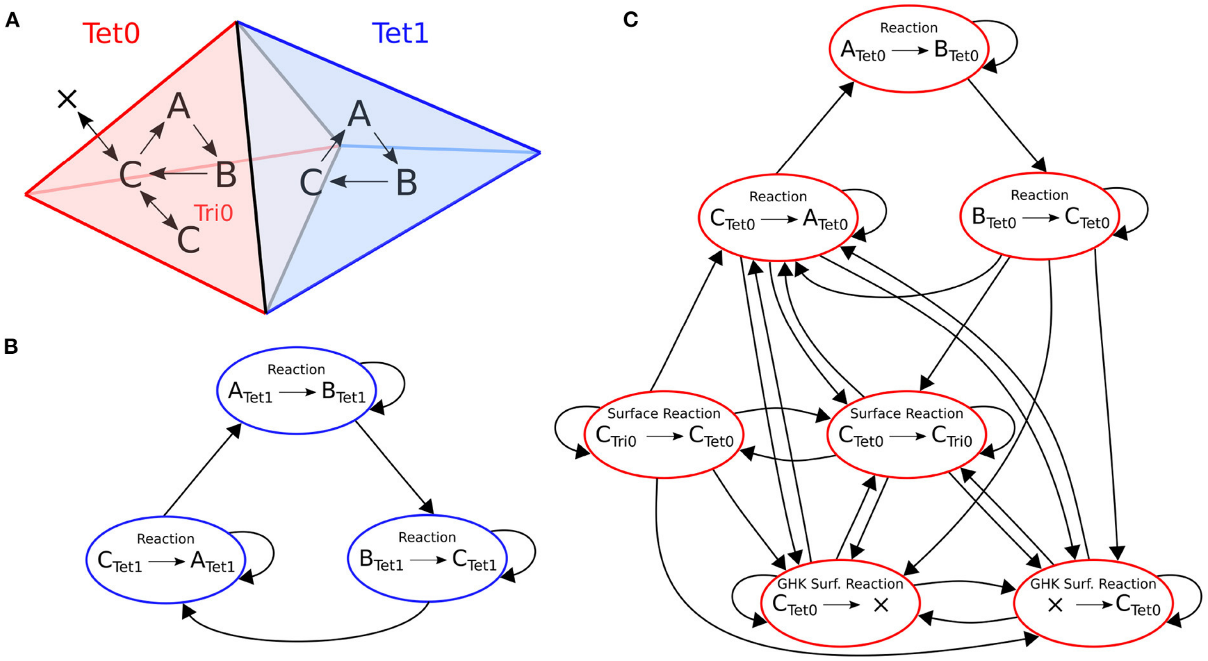

A
Tet0
Tet1
A
B
C
C
Tri0
A
B
C
B
Reaction
$A_{Tet1} \rightarrow B_{Tet1}$
Reaction
$C_{Tet1} \rightarrow A_{Tet1}$
Reaction
$B_{Tet1} \rightarrow C_{Tet1}$
C
Reaction
$A_{Tet0} \rightarrow B_{Tet0}$
Reaction
$C_{Tet0} \rightarrow A_{Tet0}$
Reaction
$B_{Tet0} \rightarrow C_{Tet0}$
Surface Reaction
$C_{Tri0} \rightarrow C_{Tet0}$
Surface Reaction
$C_{Tet0} \rightarrow C_{Tri0}$
GHK Surf. Reaction
$C_{Tet0} \rightarrow$ ×
GHK Surf. Reaction
× $\rightarrow C_{Tet0}$

FIGURE 3
Structure of the reaction dependencies graph on a mesh with two connected tetrahedrons labeled 0 and 1. **(A)** The two tetrahedrons and the reactions they contain. Both tetrahedrons contain reactions $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$. Tetrahedron 0 also contains four surface reactions: Species C can be transferred back and forth to a triangle ($C_{tet} \rightarrow C_{tri}$ and $C_{tri} \rightarrow C_{tet}$); and species C can cross the membrane back and forth as a GHK current (since the amount of *C* outside of tetrahedron 0 is not modeled, it is equivalent to creating and removing species *C*). **(B,C)** The corresponding reaction dependencies graphs. Each colored node represents a kinetic process. An arrow goes from one node to the other if the occurrence of an event of the first entails a change in propensity of the second. In blue and red are the extracted connected components of the graph.

a biochemical model and a description of the geometry in which the model will be simulated. The biochemical model is composed of species, channels, reactions, diffusion rules and currents that are grouped by volume or surface systems. Although most of the biochemical modeling features available in STEPS 3 are also available in STEPS 4, surface diffusion rules are not yet supported. Internally, the same classes are used for declaring a biochemical model in STEPS 3 and in STEPS 4. While in STEPS 3 tetrahedral meshes were managed with the `TetMesh` class, a different class (`DistMesh`) was added for distributed meshes in STEPS 4. This class inherits from the same `Geom` base class as `TetMesh` but acts as a wrapper around the `Omega_h::Mesh` distributed mesh class. Classes related to the declaration of compartments (`DistComp`), patches (`DistPatch`) and membranes (`DistMemb`) in a distributed mesh are also different from the ones used in STEPS 3. Most notably, as explained in the previous section, while tetrahedral compartments in STEPS 3 are usually built from a list of tetrahedron identifiers, STEPS 4 makes use of physical tags in distributed meshes to create distributed compartment and distributed patches. On solver creation, the `DistTetOpSplit` distributed solver class in STEPS 4 initializes the relevant data structures from the biochemical model and geometry description classes. Although this type of initialization through the python API corresponds to the most frequent use case, the distributed solver can also be used and initialized directly in C++, without requiring the creation of STEPS biochemical model and geometry classes.

2.3. Validation strategy

In order to ensure accurate results, STEPS 4 is validated on a series of published models. We extend the validation pack described in Hepburn et al. (2012, 2016) to validate the reaction-diffusion solver and the basic functionalities of other data structures introduced in the new implementation. The faster validations are integrated into the STEPS release and used in continuous integration while the others are available in the STEPS validation repository[2].

The package also contains fast validations with the EField solution. However, since these models are stochastic models designed to run in a reasonable amount of time, they each contain a small tolerance that could mask minor numerical

[2] https://github.com/CNS-OIST/STEPS_Validation

inaccuracies. So as to rigorously test our new methods and implementations in STEPS 4 and ensure even no small loss of numerical accuracy, we go further in this study and investigate STEPS 4 in a series of models, comparing either to STEPS 3 results or analytical solutions to a high degree of accuracy.

Validating stochastic simulation solutions presents several challenges. Often, analytical solutions exist only for a few trivial problems and, even in those cases, the stochastic nature of the simulator makes results fluctuate around the analytical solution depending on the particular seed provided to the random number generator (RNG). Unfortunately, fixing the seeds and numerically comparing STEPS 3 and STEPS 4 results is not a meaningful strategy since the two simulators use RNG streams in different ways. Thus, we validate STEPS 4 in a statistical sense.

2.3.1. Statistical analysis

We extract meaningful statistical data from multiple realizations with different RNG seeds and compare either with STEPS 3 results or the analytical solution when available.

The general steps are:

- Record relevant trace results such as the voltage traces in a particular location in the mesh from multiple realizations of STEPS 3 and STEPS 4 simulations.
- Refine traces to extract key features of the simulation, e.g., the frequency of a spike train.
- Collect refined features among the various simulation runs and statistically compare STEPS 3 and STEPS 4.

The choice of what must be recorded and what are the relevant features depends on the particular model at hand.

In literature, many goodness of fit tests exist. One of the most used is the Kolmogorov-Smirnov test (KS test) (Massey Jr, 1951). It is demonstrated that it produces conservative results in case of discrete distributions (Noether, 1963). Since our analysis also consider peak time stamps which are inherently discretized, we decide to use the Cramér-von Mises test (CVM test) (Cramér, 1928; Von Mises, 1928) for our statistical comparisons between STEPS 3 and STEPS 4, utilizing the Scientific Python (SciPy) library. The null hypothesis is that the two samples come from the same distribution. Perhaps a common misconception is that a *p*-value below a chosen level such as 0.01 means that the null hypothesis must be rejected and, therefore, the distributions are different. In fact, when comparing two identical distributions the *p*-value is expected to be uniformly distributed on [0,1], and so if this test is repeated many times one would expect to see a *p*-value below 0.01 1% of the time. In our tests, where multiple distributions are compared within one model, we reject the null hypothesis only if there is strong evidence that *p*-values are consistently low, evidenced by significantly more than 1% of the *p*-values generated being below the 0.01 level.

Conversely, when traces are relatively smooth and the features are few, we study directly the confidence intervals at a 99% confidence level. In this case, we reject the null hypothesis if the mean of the STEPS 3 traces does not lie in the confidence interval of the STEPS 4 traces or vice versa.

3. Results

3.1. Validations

As STEPS 4 contains multiple operator components targeting different sub-systems, such as molecular reaction-diffusion and EField, we carefully select the models and independently validate each component before testing the whole implementation on a complex, real case scenario.

3.1.1. Validations of the reaction-diffusion solver

As mentioned in Section 2.3, the reaction-diffusion validations have been discussed in previous publications and are included in the STEPS validation package. STEPS 4 passes all the validations in the package. For the sake of brevity we do not provide detailed analysis of these validations here.

3.1.2. Validations of the EField solver

To validate the EField solver we use the Rallpack models described in Bhalla et al. (1992), focusing on Rallpack 1 as a basic validation of our solution, and we introduce a new statistical analysis of a stochastic implementation of Rallpack 3.

- Rallpack 1 simulates a simple uniform unbranched passive cable. No randomness is involved in this validation and STEPS 4 results are compared directly to the analytic solution.
- Rallpack 2 model solution is equivalent to Rallpack 1 but based on branching morphology. This mathematical morphological description is in practice very difficult to capture realistically in a mesh (Hepburn et al., 2013), and since Rallpack 1 already provides a basic passive validation we do not provide a Rallpack 2 solution here.
- Rallpack 3 examines the interaction between the EField system and the stochastic channel activities of the well-known Hodgkin-Huxley model (Hodgkin and Huxley, 1952). No analytical solution is available for this test, thus we compare STEPS 3 and STEPS 4 solutions using the statistical validation framework illustrated in Section 2.3.1.

3.1.2.1. Rallpack 1

Rallpack 1 (Bhalla et al., 1992) focuses on the validation of the EField solver in a passive model, with no active properties. It consists of a leaking, sealed straight cable with a current injection

TABLE 1 Parameters for rallpack 1.

Parameters	Value
Leak conductance	0.25 S/m^2
Reversal potential	−65 mV
Resistance	1 Ω
Current	0.1 nA
Cable length	1 mm
Membrane capacitance	0.01 F/m^2
EField time step (ΔT_{EField})	5 μs
Number of tetrahedrons	1,135

(J) at z_{min}. Rallpack 1 setup is depicted in Figure 4A. Current is injected in a leaking cable with sealed ends. Table 1 provides the parameters. A leak channel is introduced on every surface triangle. This is slightly different from the analytic solution setup where the leak is uniformly distributed along the cable. However, the effects should be negligible if the mesh is sufficiently refined.

Without loss of generality, we can focus on the voltage traces at the extremes of the cable, where the voltage taps (V taps) are located. This is because the equations are linear and all the intermediate solutions are super-positions of the results at the extremities.

Figure 4 visually compares STEPS 4 results with the analytic solution. As expected, there is close agreement with mean square errors (mse) $mse_{V_{z_{min}}} = 0.069\text{mV}^2$ and $mse_{V_{z_{max}}} = 0.019\text{mV}^2$. STEPS 3 presents almost exactly the same results. When comparing STEPS 3 with STEPS 4 on the same mesh, the mse is $< 10^{-15}\text{mV}^2$ for both $V_{z_{min}}$ and $V_{z_{max}}$ and is due to numerical precision (results not shown).

Convergence to the analytical solution through mesh refinement proceeds as expected with an initial steep drop followed by a plateau at numerical precision (Supplementary Section S2.1).

3.1.2.2. Rallpack 3

Rallpack 3 is an active model that builds on Rallpack 1 by adding Hodgkin-Huxley sodium and potassium channels, and is simulated on the same simple, uniform, unbranched cable geometry. The model tests ion channel activation as well as spike propagation. Rallpack 3, when run stochastically, presents sources of randomness and the problem cannot be solved analytically. A statistical analysis is employed to study this simulation and validate the code.

The degree of randomness strongly depends on the single-channel conductance and resulting density of channels, which are parameters that must be introduced when running the model stochastically. Using biologically-plausible values for single-channel conductance, with 20 pS the Rallpack 3 model demonstrates a significant number of failed spikes as illustrated in Figure 5A. Even if this behavior is an interesting stochastic effect, it strongly hinders statistical analysis. For this reason, we chose single-channel conductance of both sodium and potassium channels to be 4 pS. This almost entirely extinguishes failed spikes whilst maintaining biological plausibility. Figures 5B,C and the additional studies in Supplementary Section S2.1.1 were produced using single-channel conductance of 4 pS.

The two sample sets consist of 10,000 simulation runs each performed with STEPS 3 and STEPS 4 respectively. As for Rallpack 1, we record voltages at the extremes of the cable ($V_{z_{min}}$ $V_{z_{max}}$) (the raw traces).

The voltage trace at z_{min} presents a high peak of ~40 mV followed by a regular spike train with peaks just surpassing 20 mV. The spike train at z_{max} has no bigger spike at the beginning and spike peaks are above 40 mV. For both traces valleys are at ~-65 mV and frequencies are ~69 Hz. The simulated time span is 250 ms.

Given that traces are spike trains with, possibly, a single greater initial peak, the key features extracted and statistically analyzed are:

- peak heights;
- peak timestamps.

The null hypothesis is that STEPS 3 and STEPS 4 simulation results come from the same population, in other words, the simulations are identical. We use the CVM test to refute it with a 99% confidence level. In order to study uncorrelated events we divide the two sample sets into 100 batches each with 100 samples and we compare each STEPS 3 batch with each STEPS 4 batch, producing a set of p-values. Thus, for each key feature (e.g., time stamp of peak number 3) we obtain 10,000 p-values. If the two initial samples are taken from the same population, p-value distributions are expected to be uniform (Murdoch et al., 2008). If the number of p-values below 0.01 is higher than would be expected from a uniform distribution, we refute the null hypothesis. Figures 5B,C present the p-value distributions for peak heights and time stamps as boxplots. For the sake of clarity and brevity here we show only the results for the traces at z_{max}. At z_{min} the results are qualitatively identical. We briefly recall here that the boxplot of a uniform distribution of p-values is centered around 0.5, the median is at 0.5, min and max are at 0 and 1 and Q1 and Q3 quartiles are at 0.25 and 0.75, respectively. All the boxplots follow this trend.

For these reasons, we cannot refute the null hypothesis and we accept that the two samples are taken from the same population.

Supplementary Section S2.1.1 offers a thorough overview of the peak statistics (distributions, means, and standard deviations) while Supplementary Section S2.2 reports all the p-value distributions in detail.

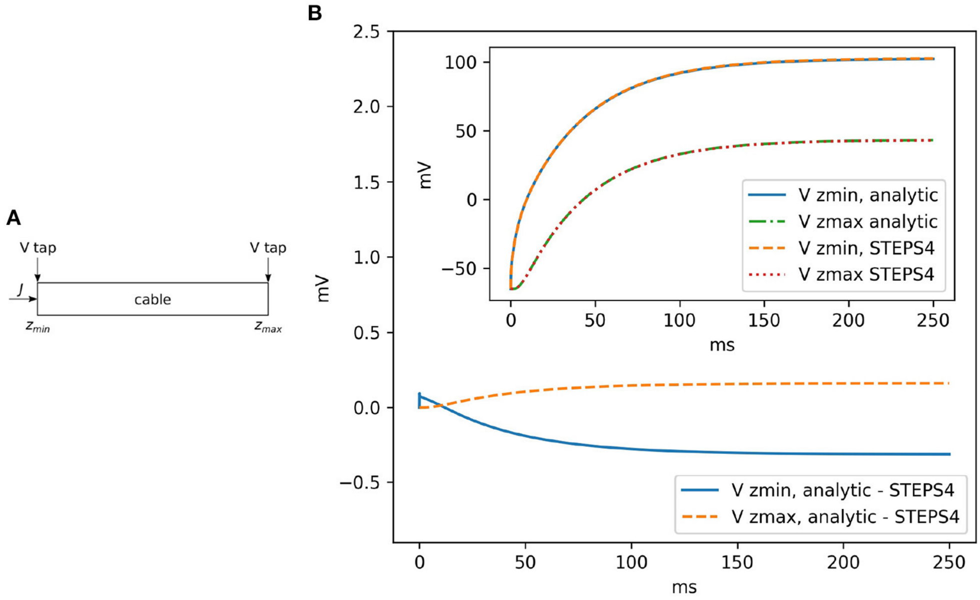


FIGURE 4
(A) The general setup for Rallpack 1. Current is introduced into a leaking cable with sealed ends. Voltage is recorded at the extremities (V taps). **(B)** The voltage difference between the analytic solution and STEPS 4 at z_{min} and z_{max}. The inset shows the overlapping curves.

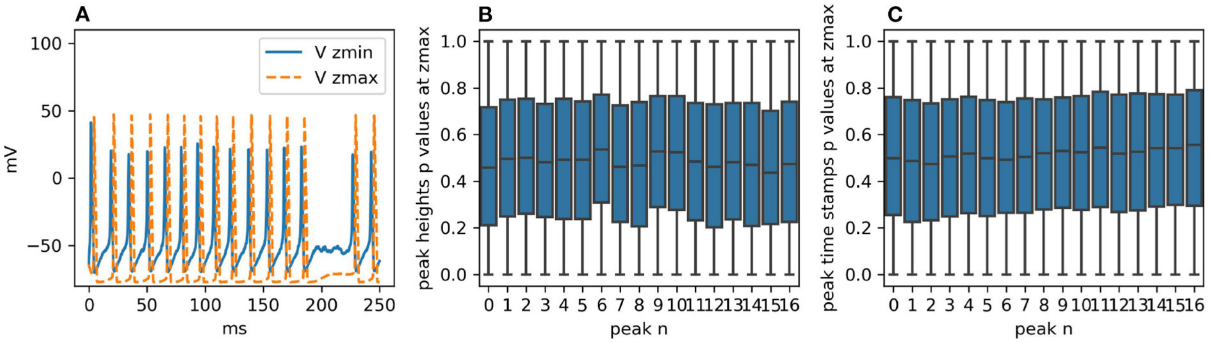


FIGURE 5
(A) Voltage traces at z_{min} and z_{max} for one realization of the simulation with STEPS 4. Failed spikes can occur with single-channel sodium and potassium conductances of 20 pS, as shown, but are eliminated with 4pS. **(B)** With single-channel conductance of 4pS, boxplots for each peak height at z_{max} of the p-values generated by dividing the two samples in 100 batches of 100 runs and then comparing them with the CVM test. As expected, the distributions are uniform. **(C)** The same analysis for the peak heights. Their intrinsic discretization does not affect the p-value distributions.

3.1.3. Validation of the reaction-diffusion and EField combined solution

Finally, we validate all components of the STEPS 4 simulator together by combining reaction-diffusion and EField features and their possible interactions.

3.1.3.1. The calcium burst model

A previously published calcium burst model (Anwar et al., 2013) is selected for the full validation. It contains most of the modeling features supported by STEPS 4, such as regular molecule reaction-diffusion events, ligand-based channel activation and electric potential dynamics. Thus, it contains all the mechanics required to validate STEPS 4 as a whole. Minor modifications are applied to the original model in Anwar et al. (2013) in order to run on a full dendritic mesh, as opposed to the sub-branch mesh used in previous studies. Figure 6A illustrates the full dendritic morphology. The full dendritic mesh was created from reconstruction retrieved from

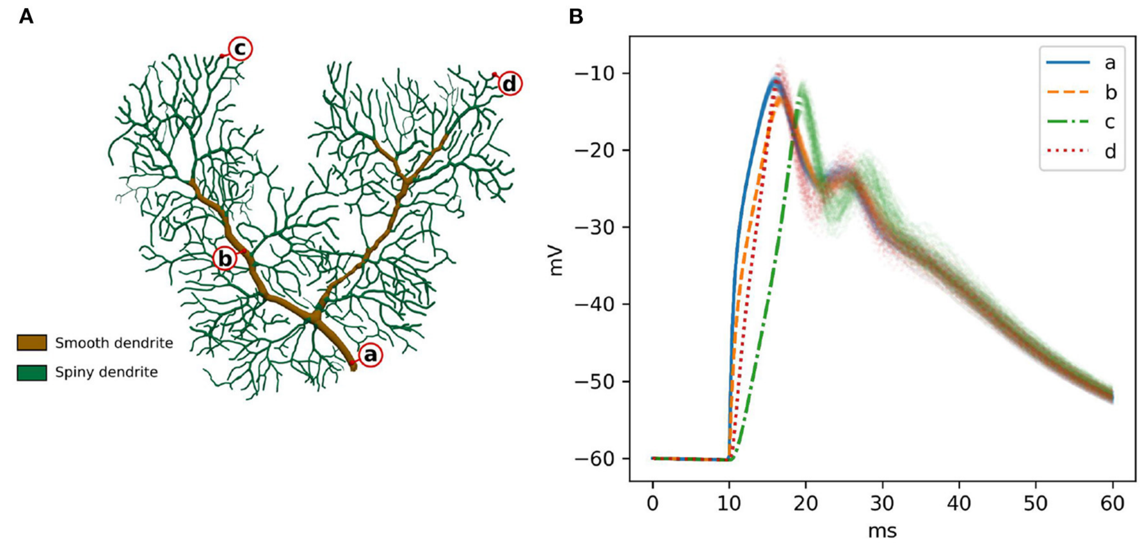


FIGURE 6
(A) The Purkinje dendrite mesh reconstruction for both calcium burst models. The mesh consists of 853,193 tetrahedrons. Dendrite elements are further classified and annotated into two components, representing smooth dendrite and spiny dendrite. **(B)** Raw voltage traces for 100 runs of STEPS 4 at the four different spatial locations indicated by a–d in **(A)**. After the first depolarization to ~18 ms the systems starts to behave stochastically coinciding with calcium-activation of potassium channels.

NeuroMorpho.Org (Ascoli et al., 2007), data ID: NMO_35058 (Anwar et al., 2014)[3]. The calcium burst model is also used to analyze the performance of the implementation in Section 3.2.

In order to sample a good representation of the dendritic tree we recorded voltage at the four disparate points shown in Figure 6A. Two points (a and b) were recorded from the smooth part of the dendrite, characterized by high diameter and low capacitance, and two (c and d) in separate regions of the spiny dendrite, characterized by low diameter and high capacitance. Figure 6B presents these traces for 100 runs of STEPS 3. In brief and as described in Anwar et al. (2013), AMPAR channel activation by a simulated glutamate burst beginning at 10 ms gives a strong depolarization, and corresponding activation of Ca$_V$2.1 P-type calcium channels gives rise to a peaks at ~18 ms and ~28 ms. Calcium activity activates mslo BK and SK2 calcium-activated potassium channels, producing the repolarization.

As for Section 3.1.2.2, our null hypothesis is that the two simulators run the same simulation and results are picked from the same population. We try to refute this statement, computing the confidence intervals of the averages of the traces at 99% probability. By definition, the confidence intervals mark a region where the trace average lies with 99% probability. Thus, if the average of the traces of the STEPS 3 set does not lie in between the confidence intervals of the STEPS 4 set or vice versa we reject the hypothesis. Figure 7 presents average and confidence intervals for all the four traces. Since confidence bands are extremely narrow, each picture is also shown with the average of the averages removed. This greatly enhances the small differences that exist between the two simulation results. STEPS 4 averages almost always lie in the confidence intervals of the STEPS 3 simulation set and vice versa. For these reasons we cannot reject the null hypothesis and we consider STEPS 4 validated even in this complex scenario.

3.2. Performance

We evaluated the performance of the implementation using three models with gradually increased complexity to cover the use cases from a wide range of research interests. The first one is a simple reaction-diffusion model on a simple cuboid mesh (we term the "simple" model). In the second model, we simulate the background activities of the calcium burst model to investigate the performance of the reaction-diffusion solution on complex Purkinje cell morphology with resting calcium activity (the "background" model). Finally, in the third model we simulate the complete calcium burst model by adding calcium channels, potassium channels and AMPAR activation (see Figure 6) to study how the combined solution performs with a real world model (the "complete" model). The simple model and the background model have previously been used to study performance and scalability of the reaction-diffusion operator splitting solution in STEPS 3

[3] http://neuromorpho.org/neuron_info.jsp?neuron_name=10-2012-02-09-001

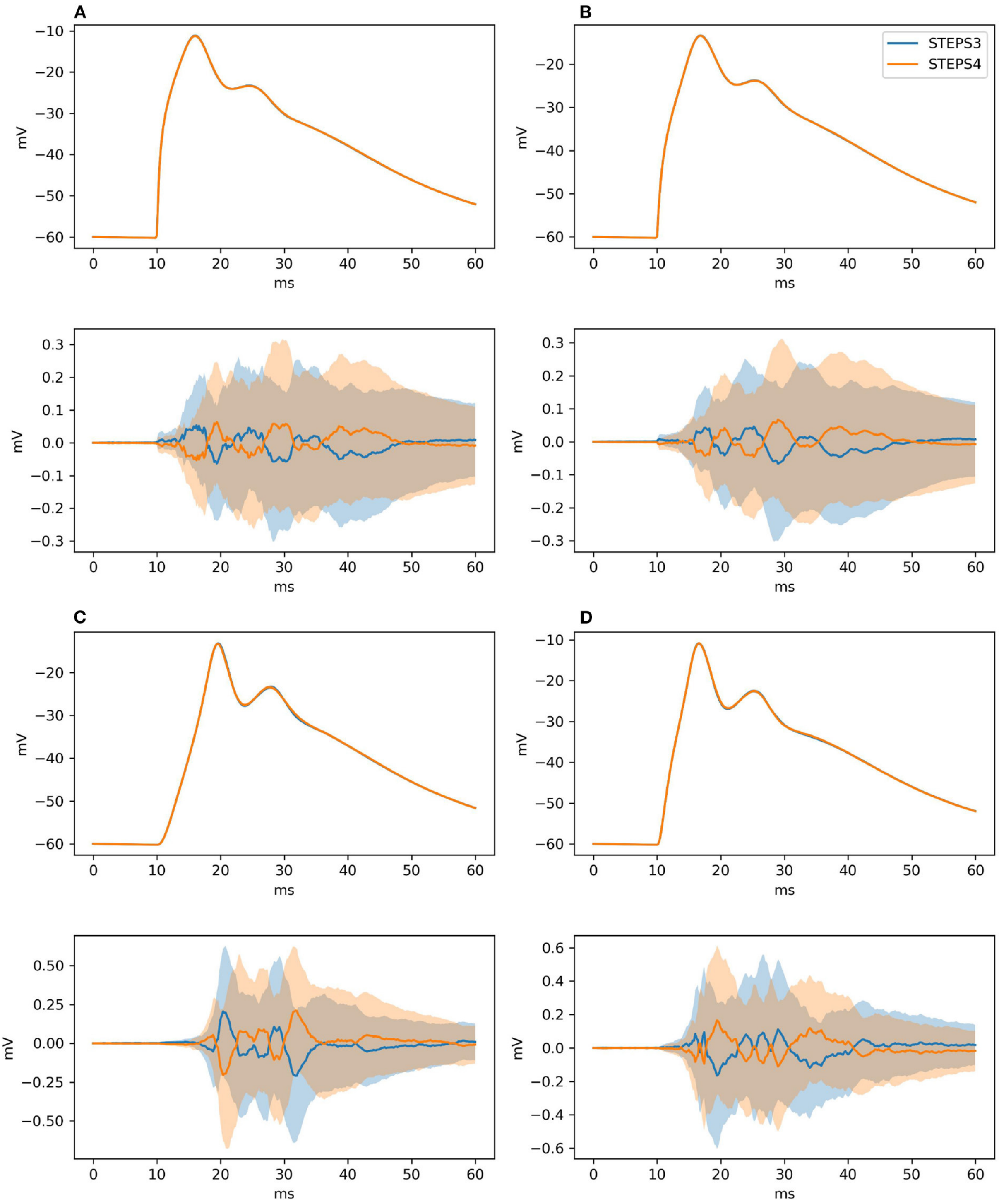


FIGURE 7
The panels illustrate average and confidence intervals (at 99% probability) of STEPS 3 and STEPS 4 simulation sets for the voltages measured at: **(A)** root and **(B)** middle point on the spiny membrane, and on **(C)** left and **(D)** right tip on the smooth membrane. Since confidence intervals are extremely narrow, the lower subplot in each panel presents the results relative to the average of all traces so that the confidence intervals can be seen clearly. Each sample consists of 100 runs. Since averages lie everywhere in each other confidence intervals we cannot refute the null hypothesis.

TABLE 2 Hardware and software configurations of the Blue Brain 5 (Phase 2) supercomputer.

	System	**HPE SGI 8600**
Hardware	Compute Node (880×)	2× Intel Xeon Gold 6248 Cascadelake @2.5*GHz* (20 physical cores per CPU)
	Memory	384*GB* of main memory (12 × 32*GB* DDR4-2933 DIMMS)
	Network	InfiniBand EDR 100*Gbps* / Fat-tree topology
	Accelerator	GPFS/ IBM Spectrum Scale Filsystem (6.2*PB*)
Software	Compiler	GCC C++ compiler 9.3.0
	Operating System	Red Hat Enterprise Linux Server 7.9
	MPI	HPE MPI (SGI MPT) 2.25
	Python	3.8.3
	Linked Libraries	PETSc 3.14.1, Omega_h 9.34.6, Intel MKL 2018.3, Eigen 3.3.8, SUNDIALS 2.7.0, mpi4py 3.1.3, NumPy 1.21.4, GMSH 4.9.0

Supercomputer.

(Chen and De Schutter, 2017). As the implementation has been improved since the initial implementation, and the hardware used in the previous research is now outdated, new simulation series of these two models are performed to acquire up-to-date results for comparison. The parallel performance of the combined solution with the complete calcium burst model has not been reported previously. We also investigate the effect of the independent graph optimization on the simple model with different molecule concentration setups. We disable this optimization for the calcium burst background and complete models as the complex morphology of Purkinje cell could lead to poor SSA event hit rates in some partitions, and worsens the overall performance of these simulations if this optimization is enabled.

3.2.1. Benchmarking setup

All simulation benchmarks were run on the Blue Brain 5 (BB5) supercomputer hosted at the Swiss National Computing Center (CSCS) in Lugano, Switzerland. A complete description of the hardware and software configuration details of the BB5 system are provided in Table 2. All benchmarks were executed in pure MPI mode by pinning one MPI rank per core. As the number of cores used for simulation needs to be a power of 2 (see Supplementary Section S1.1), for each series of benchmark we first choose an initial core count as a baseline and then double the core count.

The code instrumentation for the performance measurement in STEPS is performed through an *Instrumentor* interface. This is a light wrapper that allows for marking/profiling code regions of interests either by calling a start/stop method or by *C*++ Resource Acquisition Is Initialization (RAII) style. Various backends are used by this interface, in particular in this work we use Caliper 2.6 (Boehme et al., 2016), and LIKWID 5.2.0 (Treibig et al., 2010).

For each benchmark configuration, we repeat the simulation 30 times, and show the average results in the figures. The standard deviations of the results are reported as the error bars for each data point in the figures. Per-core memory consumption of each simulation is also measured using the *psutil* Python module (Rodola, 2020) and reported. The comparisons are mainly conducted between STEPS 3 and STEPS 4. For the scalability studies, we also compare the results with the theoretical ideal speedup scenarios. We further investigate the contribution and scaling properties of operator components in STEPS 4, namely, the SSA operator, the diffusion operator and the EField operator, by measuring their individual speedup as well as the proportion in the overall simulation time cost.

3.2.2. The simple model

We reuse the simple model in Chen and De Schutter (2017) which consists of 10 diffusing species with different initial molecule counts within simple cuboid geometry with 13,009 tetrahedrons. These species interact with each other through 4 different reversible reactions with different rate constants. The details of the model can be found in Table 3. We choose 2 cores as the performance baseline and increase the core count to $2^{11} = 2,048$ as the maximum. Note each core has less than 10 tetrahedrons with this maximum, at which point it is unlikely that the simulations remain scalable. However, the result is still interesting as it illustrates the behavior of our solution under extreme scaling scenarios.

The effect of the independent graph optimization is also investigated using the simple model with different initial molecule counts. We first simulate the model in Table 3 without the optimization and use it as the baseline configuration. We then modify the baseline model with four new settings, the first two reduce the initial count of each molecular species by 10x and 100x, and the other two increase molecule counts by 10x and 100x. We name these simulation series "0.01x," "0.1x,"

TABLE 3 Species and reactions as well as the initial configuration of the simple model.

Species	Diffusion coefficient (μm^2 / s)	Initial count
A	100	1,000
B	90	2,000
C	80	3,000
D	70	4,000
E	60	5,000
F	50	6,000
G	40	7,000
H	30	8,000
I	20	9,000
J	10	10,000
Reaction	**Rate Constant**	
$A + B \rightleftharpoons C$	$k_f : 1,000(\mu M \cdot s)^{-1}, k_b : 100s^{-1}$	
$C + D \rightleftharpoons E$	$k_f : 100(\mu M \cdot s)^{-1}, k_b : 10s^{-1}$	
$F + G \rightleftharpoons H$	$k_f : 10(\mu M \cdot s)^{-1}, k_b : 1s^{-1}$	
$H + I \rightleftharpoons J$	$k_f : 1(\mu M \cdot s)^{-1}, k_b : 1s^{-1}$	

"1x," "10x," and "100x" respectively. We also repeat these series with independent graph optimization enabled and record the results for comparison. As this optimization solely targets the SSA operator, a single core is used to run the simulation series, and the time cost of the SSA operator instead of the overall simulation time cost is measured.

Simulation results of the simple model are summarized in Figure 8. Both STEPS 3 and STEPS 4 implementations demonstrate a steady decrease of simulation time early on until $2^6 = 64$ cores, and maintain roughly the same time cost for the rest of the configurations. The memory footprint improvement from STEPS 4 is significant. In the baseline simulations, STEPS 4 consumes 45.6MB of memory per core, about 60% of the required memory for STEPS 3. When simulating the model with thousands of cores, the memory consumption of STEPS 4 further decreases to about 4.5MB per core, 10% of the baseline simulation consumption, thanks to the completely distributed nature of the solution. While the memory footprint of STEPS 3 simulations also decreases with high core counts, the number stabilizes at 16MB, 2.6 times more than STEPS 4 requires. The strong scaling speedup for both STEPS 3 and STEPS 4 in Figure 8C suggests that the STEPS 4 achieves close-to-ideal speedup until $2^6 = 64$ cores, reflecting the time cost result in Figure 8A. In fact, the SSA component further maintains a linear speedup until $2^9 = 512$ cores according to the component scalability analysis in Figure 8D. However, due to the high scalability, its proportion in the overall time cost reduces significantly in high core count simulations. For these simulations, the diffusion operator and other background maintenance routines become the two major proportions of the simulation time cost.

The performance difference caused by the independent graph optimization is illustrated in Figure 8F by the ratio between enabling and disabling the optimization. In the baseline 1x simulations and other series with reduced molecule counts, enabling the optimization results in a slight performance decrease as the SSA time cost ratios in these series are all above 1.0, ranging from 1.15 in the 0.01x series, to 1.09 in the 1x series. The benefit of the optimization is noticeable in the 10x series with a ratio of 0.97, and becomes significant in the 100x case, which shortens more than half of the simulation time. These results agree with our analysis in Section 2.2.4.

3.2.3. The calcium burst background model

We extend our investigation on the reaction-diffusion component with the calcium burst background model with complex Purkinje cell morphology as described in Section 3.1.3.1. There is no voltage component nor any ion channels in this model, only background buffering reaction and diffusion. In total, the model consists of 15 molecule species, 8 of which are diffusive, and 22 reactions. The simulated mesh consists of 853,193 tetrahedrons. To eliminate any difference caused by partitioning, we pre-partition the mesh in Gmsh then import the partitioned mesh to the simulations, therefore the partitioning is always the same for each benchmark configuration. We start the simulation series from $2^5 = 32$ cores within a single node, then double the core count each time until the maximum of 512 nodes with $2^{14} = 16,384$ cores is reached.

Figure 9 presents the key results of the simulation series. In general, STEPS 4 performs slightly worse than STEPS 3 in low core count configurations, but eventually achieves similar performance as the core count increases. This is because currently STEPS 4 implements the widely accepted Gibson and Bruck (Gibson and Bruck, 2000) next reaction method as the default SSA operator. This method provides logarithmic computational complexity with simple data structures that we find suitable for the distributed solution. On the other hand, STEPS 3 inherits the serial implementation of the Composition and Rejection method (Slepoy et al., 2008), which requires a more complex data structure but takes advantage of its constant time complexity, particularly when dealing with large number of reactions in low core count simulations. It is worth noting that the compartmental design in STEPS 4 supports multiple operator implementations, therefore more efficient operators can be easily integrated to the solution in the future.

Dramatic improvement in memory consumption can be observed for STEPS 4 in Figure 9B. All STEPS 3 simulations require no less than 2GB of memory per core; on the other hand, the highest per-core memory footprint for STEPS 4 is about 630 MB with $2^5 = 32$ cores, and drops down to about 67MB with $2^{10} = 1,024$ cores and above, roughly 3% of what is required by STEPS 3.

Both STEPS 4 and STEPS 3 demonstrate linear to super-linear speedup until $2^{12} = 4,096$ cores in Figure 9C. Component scaling analysis in Figure 9D suggests that both

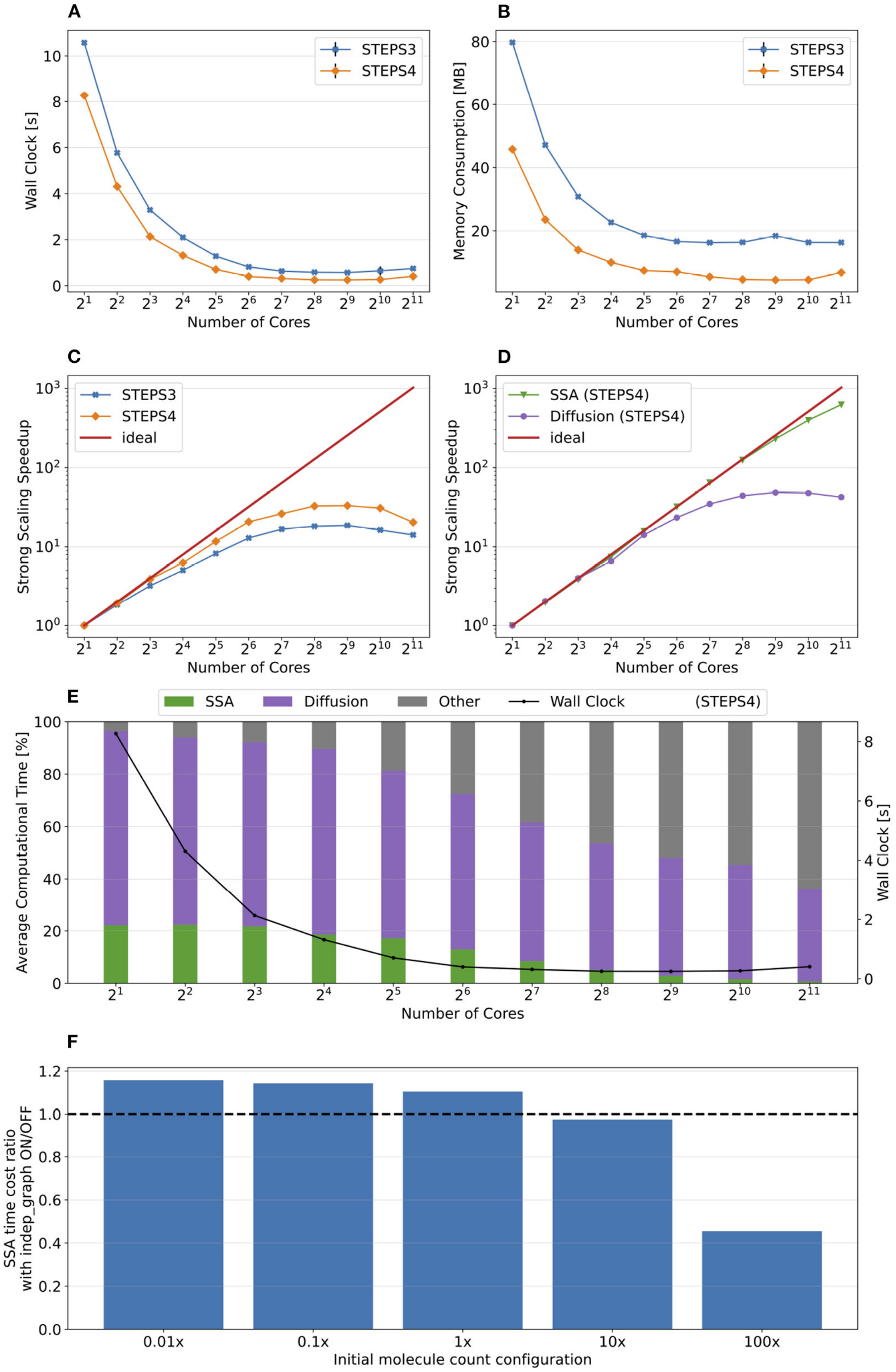


FIGURE 8
The performance results and scalability of the simple model. **(A)** Both STEPS 3 and STEPS 4 implementations demonstrate a steady decrease of time cost early on, then maintain similar time cost beyond $2^6 = 64$ cores. **(B)** STEPS 4 consumes significantly less per-core memory than STEPS 3, ranging from 60% in the baseline simulation, to approximately 30% in high core count simulations. **(C)** STEPS 4 achieves close-to-ideal

(Continued)

FIGURE 8 (Continued)
speedup until $2^6 = 64$ cores, but has poor scalability afterward. Similar but slightly worse scalability can be observed for STEPS 3. **(D)** Component scalability analysis of STEPS 4. The SSA operator shows a linear speedup until $2^9 = 512$ cores. **(E)** Component proportion analysis of STEPS 4. Due to the highly scalable SSA operator, the time cost of high core-count simulations is dominated by the diffusion operator and other non-scalable bookkeeping routines, resulting in poor scalability in high core count scenarios. **(F)** The SSA operator time cost ratio with and without independent graph optimization in different initial molecule count setups. Enabling the optimization results in performance decrease in low molecule density simulations, but provides significant speedup in simulations with high molecule density.

the SSA and the diffusion operators contribute to this result. The diffusion operator maintains close-to-linear speedup until $2^{12} = 4,096$, while the SSA operator demonstrates super-linear speedup throughout the series. We investigated this scaling behavior and further profiling on the SSA operator indicates that the super-linear speedup mainly comes from the update routine of the operator, including the propensity calculations and the priority queue updates. This suggests that the improvement on memory caching may play an important role here.

The diffusion operator is the dominating component in this series, as shown in Figure 9E. Its proportion in the overall time cost increases from 65 to 95%. The proportion of other non-scaling routines also rises but is still less than 10% with the maximum $2^{14} = 16,384$ cores. Overall the performance profile of the background model is very similar to the simple model profile. This is not surprising as they both involve the same operators but the background model has more tetrahedrons and reactions per core compared to the simple model.

3.2.4. Complete calcium burst model

The complete calcium burst model as described in Section 3.1.3.1 extends the background model by coupling molecular reaction-diffusion updates with voltage-dependent channel activation as well as membrane potential changes. Different channel density parameters are assigned to the smooth and the spiny sections of the mesh to approximate the effect caused by regional spine density difference. The model consists of 15 regular species, 8 of which are diffusive, 5 types of channels with in total 27 different channel states, 59 regular reactions and 16 voltage-dependent reactions. Compared to the previous two models, the complete model produces a simulation with extremely complex dynamics and imbalanced computational load, both spatially and temporally. We consider it as an excellent demonstration of STEPS 4 performance in realistic research projects.

Figure 10 summarizes the key results of the simulation series. While STEPS 4 performs slightly worse than STEPS 3 initially, it reaches similar performance with $2^9 = 512$ cores, and outperforms STEPS 3 for the rest of the series. As expected, STEPS 4 continues its advantage on per-core memory footprint management, starting from 1.5GB for $2^5 = 32$ core simulations, to approximately 500MB for $2^9 = 512$ cores and above. The minimum memory requirement for STEPS 3 is 5GB, 10 times what is needed with STEPS 4. While the BB5 cluster has high memory capacity per compute node and is able to provide 12GB of memory per core for simulations (given the 32 active processes per node), many HPC clusters commonly have the memory capacity restriction of about 4GB per core (Zivanovic et al., 2017), therefore only STEPS 4 simulations can be run on those clusters.

Overall, STEPS 4 achieves a better scalability compared to STEPS 3, with linear speedup from the diffusion operator, and the super-linear speedup from the SSA operator. However, the EField operator has limited scalability, reaching maximum 10x speedup relative to the baseline. This results in a great increase of EField operator time cost in proportion to the total computation time, from 10% in the baseline simulations to 76% in the $2^{14} = 16,384$ core simulations, making it the major performance bottleneck of the series, as shown in Figure 10E.

3.2.5. Memory footprint with refined mesh

As shown in the above results, the significantly reduced memory footprint is one of the major advantages of STEPS 4. To further investigate the memory consumption difference between STEPS 4 and STEPS 3, we refine the Purkinje cell mesh and rerun both the calcium burst background model and the complete calcium burst model with the new mesh. The refined mesh consists of 3,176,768 tetrahedrons. For simplicity, we name the original mesh as the "1M" mesh, and the refined mesh as the "3M" mesh accordingly. As the 3M simulations exhibit similar performance profiles as the 1M versions, we focus on the memory footprint of the simulations. Performance and scalability results of the 3M simulations can be found in the Supplementary Section S2.3.

Figures 11A,B provide an overall view of the results. For all simulation series, the baseline configuration, i.e., the one with the lowest core count, has the highest memory footprint, then progressively reduces to a consistent minimum. This is essential as any cluster with per-core memory capacity below the minimum can not execute the simulation regardless how many cores are available. Thus, we hereby use the minimum memory consumption from each series for comparison. Figure 11A presents the memory consumption of the background model, for both STEPS 3 and STEPS 4, and for both the 1M and 3M meshes. For the 1M mesh simulations, STEPS 4 requires 67MB memory per core, while STEPS 3 requires approximately 2GB, 30x of the STEPS 4 requirement. For the 3M mesh simulations,

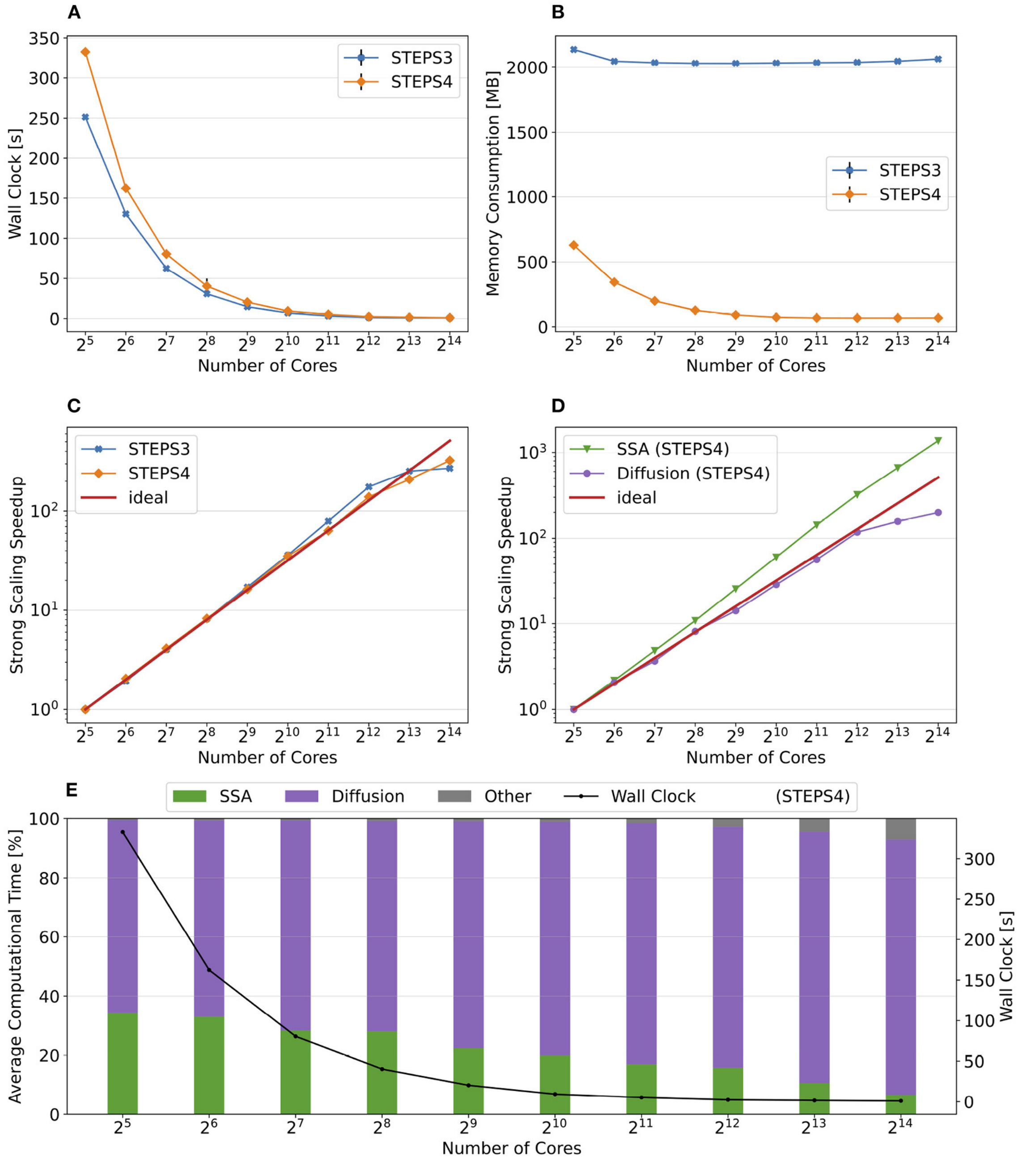


FIGURE 9
The performance results and scalability of the calcium burst background model. **(A)** Steady decrease of simulation time cost can be observed in both STEPS 4 and STEPS 3 simulations. STEPS 4 performs slightly worse than STEPS 3 in low core count simulations, but both eventually achieve similar performance as core count increases. **(B)** The memory footprint of STEPS 4 is superior compared to the STEPS 3 counterparts, requiring about 630MB for $2^5 = 32$ core simulations, and 67MB for $2^{10} = 1,024$ core and above simulations. STEPS 3 consumes more than 2GB of memory per core for the whole series. **(C)** Both STEPS 4 and STEPS 3 demonstrate linear to super-linear scaling speedup. **(D)** Component scalability analysis of STEPS 4 suggests that the diffusion operator in STEPS 4 exhibits linear speedup until $2^{10} = 1,024$ cores, while the SSA operator shows a remarkable super-linear speedup throughout the series. **(E)** Component proportion analysis of STEPS 4. The diffusion operator is the dominating component, taking from 65 to 95% of the overall computational time.

200MB memory per core is required by STEPS 4, while 6.6GB is required by STEPS 3, about 33x of the STEPS 4 requirement. Results of the complete model are shown in Figure 11B. For the 1M series, STEPS 4 requires about 500MB of memory, while STEPS 3 requires approximately 5.1GB, resulting in a 10x difference. For the 3M series, the memory footprint of

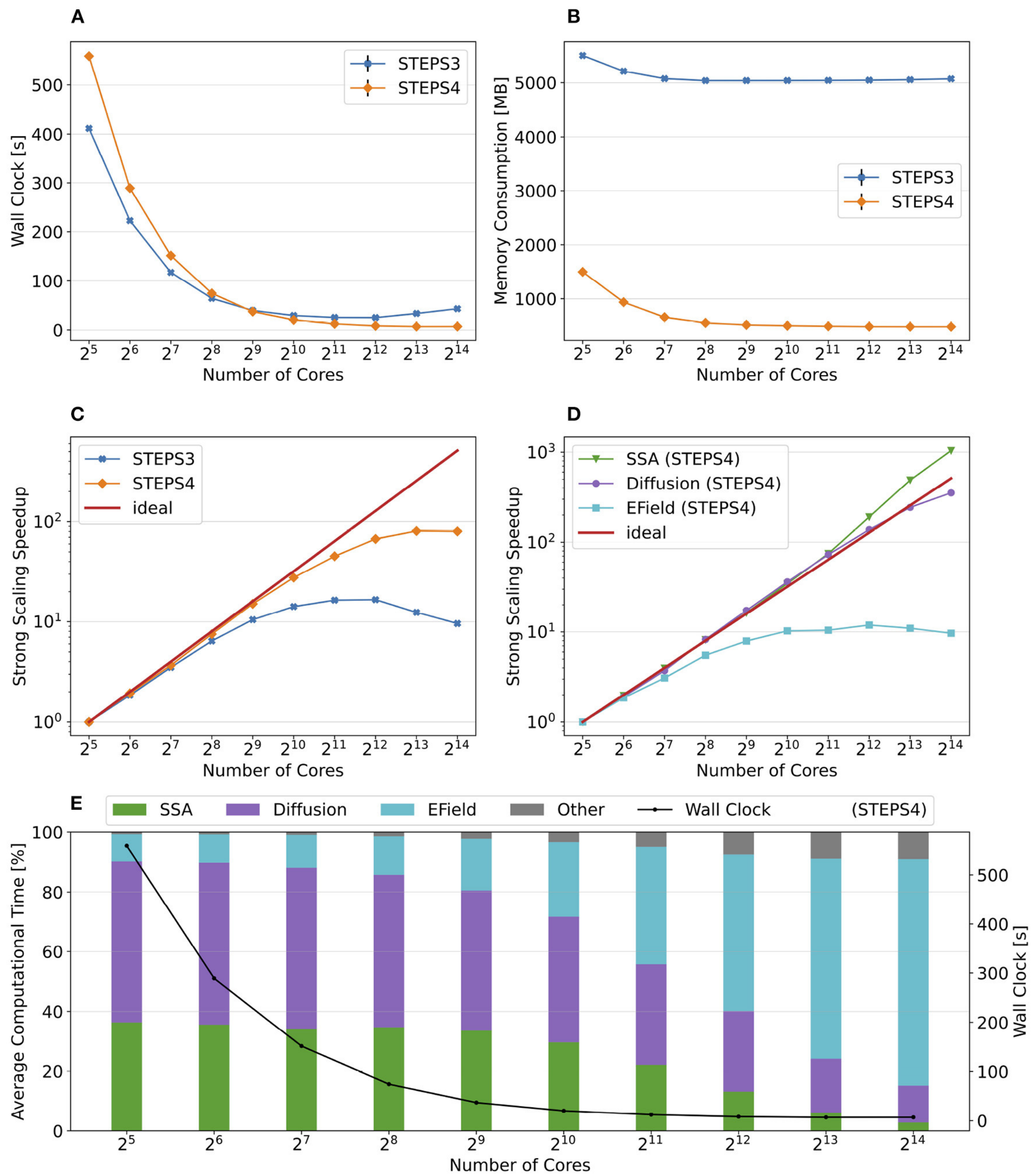


FIGURE 10
The performance results and scalability of the calcium burst complete model. **(A)** STEPS 4 performs slightly worse than STEPS 3 in low core count simulations, but reaches similar performance with $2^9 = 512$ cores, and outperforms STEPS 3 afterward. **(B)** STEPS 4 requires about 1.5GB for the $2^5 = 32$ core baseline simulations. Its memory footprint quickly decreases to 500MB for $2^9 = 512$ cores and above. STEPS 3 consumes more than 5GB of memory per core for the whole series. **(C)** STEPS 4 achieves a better scalability compared to STEPS 3. **(D)** Component scalability analysis of STEPS 4. The SSA operator shows super-linear speedup throughout the series. The diffusion operator also exhibits linear speedup until $2^{13} = 8,192$ cores. However, the EField operator shows limited scalability with maximum 10x speedup with $2^{10} = 1,024$ cores and above. **(E)** Component proportion analysis of STEPS 4. The EField operator progressively dominates the computational time, from 10% in the baseline simulations to 76% in the $2^{14} = 16,384$ core simulations, due to its limited scalability compared to the other operator components.

STEPS 4 increases to 770MB. We are unable to simulate the 3M complete model in STEPS 3 with 12GB of memory per core.

To further explore the capability of STEPS 4 in supporting super-large scale models, we refine the Purkinje cell mesh using Gmsh, then simulate the complete model with the refined

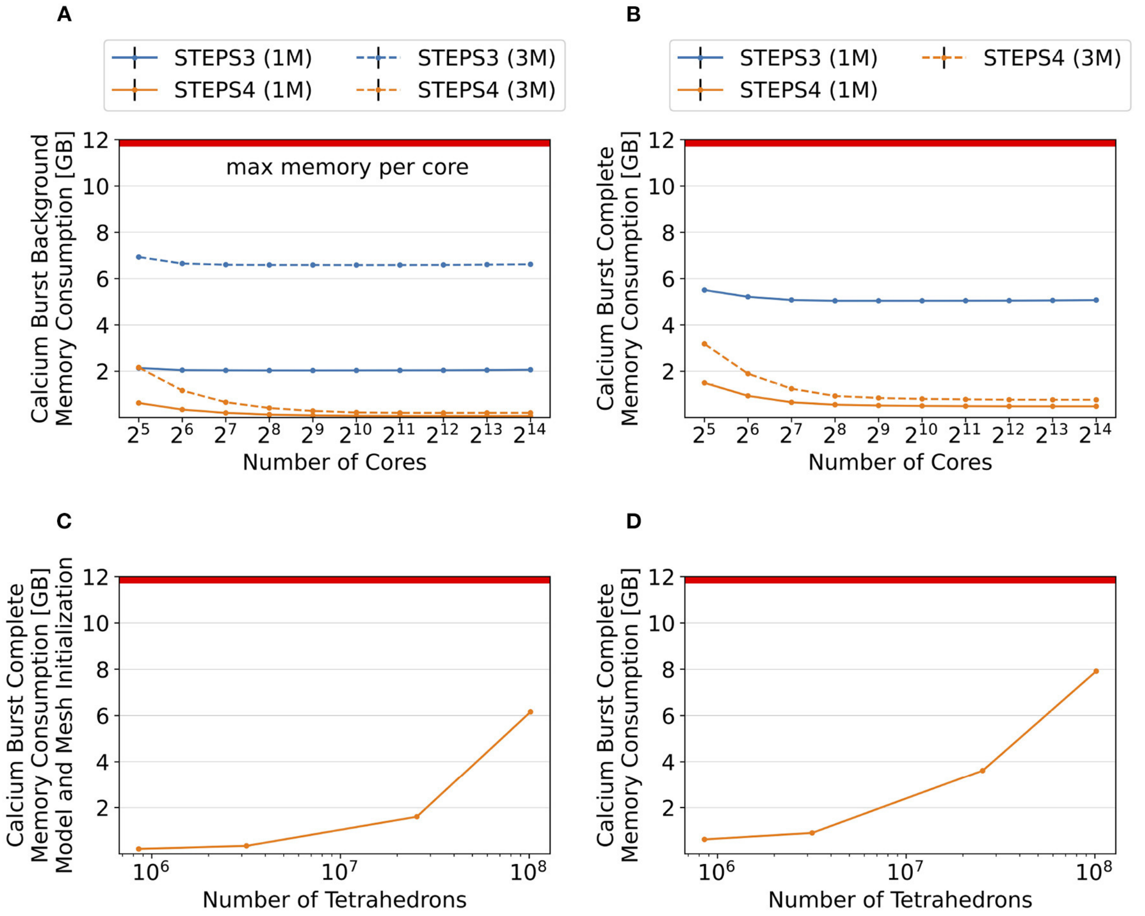


FIGURE 11
Memory footprint analysis and exploration of super-large scale models. In general, per-core memory consumption decreases as the core count increases, until a stabilized minimum consumption is reached. **(A)** Results of the background model simulations. The memory required per core hardly changes for STEPS 3 from 2.1GB to 2.0GB, while it declines rapidly for STEPS 4 from 626MB to 67MB. Similar results can be observed in the 3M series, where per-core memory consumption declines from 6.9GB to 6.6GB for STEPS 3, and from 2.1GB to 200MB for STEPS 4. **(B)** Results of the complete model simulations. In the 1M series results, memory consumption decreases from 5.5GB to 5.1GB for STEPS 3, and from 1.5GB to 500MB for STEPS 4. STEPS 4 in the 3M series consumes 3.2GB to 770MB of memory per core as core count increases. The 12GB memory capacity of the cluster per core is inadequate for the 3M complete model simulations with STEPS 3. **(C)** Memory consumption in GB at the initialization stage for the 1, 3, 25, and 100M meshes. **(D)** Total memory consumption in GB of STEPS 4 for the 1, 3, 25, and 100M mesh models. From our estimation, the 200M mesh requires a little over the 12GB memory capacity per core in the current setup using 16,384 MPI tasks.

meshes on $2^{14} = 16,384$ cores, and record the memory consumption at both the initialization and execution stages. The refined meshes have 25.4 million, 101.6 million and 203.3 million tetrahedrons, and are named “25M”, “100M”, and “200M” meshes respectively. Due to the large scale and consequently long execution time of these models, we do not run the full simulations but stop them after the first time point when memory consumption is stabilized. As shown in Figure 11C, memory consumption at the initialization stage increases from 213MB for the 1M mesh to 6.16GB for the 100M mesh. Slightly more memory is required for the simulation stage (Figure 11D), varying from 480MB for the 1M mesh, to 7.77GB for the 100M mesh. We are unable to initialize and execute simulations with the 200M mesh as the 12GB memory capacity is reached. From our estimation based on curve fitting of the results, a successful execution of the 200M mesh simulation would require approximately 13GB of memory on each core.

3.2.6. Single node roofline analysis of STEPS 4

In general, STEPS 4 demonstrates similar or better performance compared to STEPS 3 in high core count simulations, but has lower performance in small core count simulations. As discussed previously, one of the reasons is the different SSA operator implementations, but other factors may also be involved. As the performance with small core count simulations is also important for STEPS 4 usage, a detailed performance analysis of current simulations is necessary to determine the direction of future optimizations. We choose the complete model as the profiling target since all major operators are included in the simulation. Note that in low core count configuration, the SSA and the diffusion operators are the dominating components in the simulation, thus they are the main focus of the analysis here. This is different from the optimization of high core count simulations, where the EField operator dominates the computation.

The analysis is based on the Roofline model (Williams et al., 2009), evaluating the scaling trajectory (Ibrahim et al., 2018) of the most computationally expensive routines, in our case, the SSA reaction operator, the Diffusion operator, and the EField operator. The Roofline model is one of the simplest tools to apply hardware/software co-design, enabling investigation on the interaction between hardware characteristics like memory bandwidth and peak performance, and the software characteristics such as memory locality and arithmetic intensity. Thus, it provides essential information on whether the investigated components are memory bandwidth or compute bound, and consequently vital suggestions on optimization strategies.

The Roofline model shown in Figure 12 for the Cascade Lake node on BB5 is constructed from a measured memory bandwidth ($\approx 197 GB/s$) and a measured peak core performance ($\approx 78 Gflop/s$, where *flop* stands for floating-point operations). Both metrics are measured with the likwid-bench utility (Treibig et al., 2010). In the Roofline graph, the x-axis is the arithmetic (or computational) intensity, computed as the ratio of floating point operations to transferred bytes from the main memory (DRAM traffic), and the y-axis is the observed performance. To obtain a scaling trajectory (Ibrahim et al., 2018) the measures are taken for varying core counts. Additionally, we run simulations with hyper-threading ($2^6 = 64$ processes) in order to utilize maximum resources.

For the measurements of the routine with LIKWID, a MPI synchronization barrier is added before and after each measured kernel. This is done to ensure that the measured metrics (e.g., hardware counters) indeed belong to the respective routines.

From Figure 12, it can be seen that all routines have a low arithmetic intensity. Each routine is represented by a different symbol and each data point is labeled by the number of processes. As described in Ibrahim et al. (2018), for ideal scaling a doubling of concurrency corresponds to a change in Δy (observed Flop/s) of $\approx 2\times$ without a corresponding change in Δx (arithmetic intensity), a behavior observed in our experiments. The SSA kernel is the one with the lowest arithmetic intensity and it is well into the arithmetic intensity regime where we expect the kernel to be memory bound. The Diffusion kernel presents similar behavior but with higher arithmetic intensity. Both kernels reach a saturation point as they approach the peak memory bandwidth. This observation suggests that there would be little to no gain to be had by vectorizing these kernels, instead possible improvements would have to come from algorithmic changes and/or cache blocking strategies in order to either increase the arithmetic intensity or fit the working memory set into the last level cache (LLC). For the EField kernel, we observe both $\Delta y > 0$ and $\Delta x > 0$ as we perform the strong scaling. This transition indicates that the number of floating-point operations has remained constant, so data movement must have decreased (Ibrahim et al., 2018). Finally, for all the computational kernels hyper-threading does not lead to any substantial performance increase.

To reach the maximum performance of a compute node, we need to efficiently utilize the cache memory hierarchy. In the Roofline graph, the higher the cache efficiency the higher the computational intensity. In our case, the low arithmetic intensity could be explained by the use of data structures that do not favor data locality (e.g., maps/dictionaries over vectors). Thus, a substantial improvement in the computational intensity of STEPS 4 can be achieved by favoring data locality and thus higher cache utilization, such as by a more extensive use of the flat-multimap.

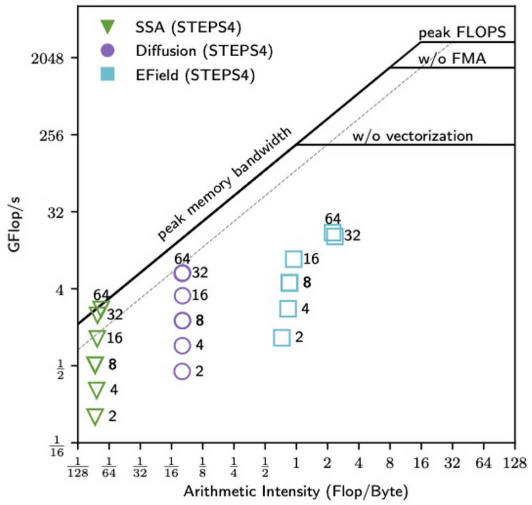


FIGURE 12
Roofline single-node scaling trajectories. The solid black lines are the full node hardware limits and the dashed gray line is the peak memory bandwidth for one socket. Each data point is labeled by the number of processes. All the computational kernels present a low arithmetic intensity mainly due to not ideal data locality (not optimal cache utilization). Nevertheless, the scaling is close to ideal (especially for the SSA and Diffusion operators) given that the doubling of concurrency leads to a corresponding $\Delta y > 0$. Hyper-threading at 64 cores does not give any substantial performance increase.

4. Discussion

4.1. Achievements

With this continuous development and modernization of STEPS, we achieved several major goals:

- We modernized the existing code base of the entire framework adopting modern programming standards and practices such as C++17 and continuous integration. Particular care was posed on safety features such as vocabulary types. These improvements provided a solid modern foundation for STEPS 4 development.
- We developed a distributed solution that addressed the bottlenecks of STEPS 3.

STEPS 4 achieves similar performance and scalability as STEPS 3 while dramatically reducing the memory footprint. This is a key feature for future realistic modeling using STEPS. The Purkinje dendrite morphology simulated in the calcium burst model was reconstructed from light microscopic imaging. The spines were ignored and only the skeleton of the dendrite was preserved. It is possible to reconstruct a highly realistic Purkinje neuron containing all visible spines from high resolution electron microscopic imaging, however, the mesh generated from such morphology is expected to have 10 to 100 times more tetrahedrons than those used in current simulations. Such large models are completely out-of-reach for STEPS 3 since even the relatively small 3M calcium burst model already exceeds the 12GB per-core memory capacity on a state-of-the-art cluster like BB5. Conversely, STEPS 4 showed its potential on supporting simulations with such scale in the refined mesh simulations.

4.2. Limitations and solutions

STEPS 4 is not a complete replacement for STEPS 3. It is a highly specialized version of the operator-splitting solution specifically tailored for cluster-based, super-large scale simulations. Thus, we paid particular attention to performance optimizations whilst maintaining accuracy.

Even if STEPS 4 covers most features available in STEPS 3, some remain missing. For instance, the diffusion of species on surfaces (i.e., between patch triangles), and the associated surface diffusion boundaries, are not yet available. Patches between compartments are in principle supported but meshes have to be partitioned in such a way that tetrahedrons on both sides of patch triangles are owned by the same process. In STEPS 3, this constraint is enforced by *ad-hoc* partitioning adjustment; in STEPS 4 since the mesh is handled by Omega_h, this constraint is not enforced and it is up to the modelers to generate suitable partitioned meshes for their simulations, which is a limitation of this approach. We plan to support automatic partitioning adjustment with constraints in STEPS 4 in the future, however this requires further collaboration with the Gmsh and Omega_h developers as these libraries need further development to support such functionality. Finally, some auxiliary features in STEPS 3 such as the Region of Interest (ROI) functionality and visualization are not yet supported as implementations of new STEPS modeling toolkits are required to adapt the new distributed mesh formats and protocols.

4.3. Potential enhancements for STEPS 4

The distributed mesh backend of STEPS 4, Omega_h, not only supports traditional MPI based distributed-memory parallelism, but also shared-memory parallelism through OpenMP, and GPU parallelization *via* the CUDA framework. It also provides unique features such as mesh adaptation suitable for GPUs using flat array data structures and bulk transformations. These advanced features are currently not utilized in STEPS 4 as it relies solely on CPU based MPI parallelism. With the importance of GPU based fat compute nodes in modern HPC clusters, such features will play important roles when STEPS 4 is transitioned to other parallelism schemes.

In addition, the scalability analysis in Section 3.2 suggests two major axes for future development. Firstly, the EField operator is shown to be the major bottleneck in high core count simulations due to its poor scalability. Detailed profiling is required in the future to investigate the fundamental cause of this bottleneck, and to address it. Secondly, the Roofline analysis shows low computational intensity for all major kernels (SSA, Diffusion, EField). This behavior points to unsatisfactory use of cache memory, mainly caused by containers/data structures with poor data locality. A more extensive use of the `flat-multimap` could greatly improve cache utilization and increase the arithmetic intensity of these computational kernels.

4.4. Choosing between STEPS 3 and STEPS 4 in research projects

It is difficult to provide a solid guideline for choosing between STEPS 3 and STEPS 4 in a research project as different factors need to be considered. At the current stage, because not all the features in STEPS 3 are supported by STEPS 4, we recommend the researcher to firstly check if the features required in the model are supported by STEPS 4. If the model is supported by both implementations, then the researcher needs to consider what platform the model will be simulated on. Due to the efficiency difference of the current SSA operator, simulations on multi-core desktop workstation may be in favor of STEPS 3, while simulations on large scale clusters with limited memory resource may prefer STEPS 4 thanks to its memory footprint optimization. It is also worth mentioning that converting a

STEPS 3 model to STEPS 4 is a relatively trivial task, often only involving several lines of code changes in the modeling script. Therefore, the researcher can conduct a pilot benchmark with both solutions, then choose the suitable one for later simulation tasks based on the benchmark results.

4.5. Other current developments and future directions

4.5.1. Vesicle modeling

Currently STEPS, as all SSA methods in general, models molecules as points that do not occupy a significant volume of the space in which they reside. This is an obvious limitation if one wants to model certain types of structures in the cell such as vesicles. Vesicles are relatively large structures (∼40 nm diameter in the case of synaptic vesicles for example) that play many important roles in biology, and their complex structure and diverse functionality mean they cannot be realistically simulated by the point-molecule approach. Vesicles undergo processes such as endocytosis and exocytosis, interact with cytosolic and surface-bound molecules, and can be spatially organized into clusters such as in the presynaptic readily-retrievable pool. In an upcoming release, STEPS aims to support all of these features in an initial parallel implementation.

While the vesicle modeling development has been a separate project from STEPS 4, one tantalizing prospect is to marry many of the novel features of STEPS 4 with the vesicle modeling to allow bigger, more detailed simulations that can be run for longer biological times. This will, however, require substantial development on STEPS 4.

4.5.2. Coupling of STEPS with other simulator software

As part of the BBP mission to create a large scale reconstruction of brain tissue, a multi-scale approach for simulation is deemed necessary to capture elements at various temporal and spatial scales: one time scale for rapidly changing neuron voltages, a different, slower time scale for changing ion concentrations. Likewise, neuron morphologies can be distributed among computing ranks irrespective of geometric boundaries whereas bulk ion concentrations and metabolism use a coarse grain division of the spatial scale. For this purpose different simulators are used to leverage their specialized capabilities. NEURON (Carnevale and Hines, 2009) is used to solve relevant equations for membrane voltage and communication between neurons in addition to calcium in astrocyte morphologies. Meanwhile STEPS is employed to compute concentrations of diffusing ions in the extracellular space. A more memory efficient STEPS enables better sharing of computing resources between the two simulators.

5. Conclusion

The STEPS 4.0 project development reported in this article addresses several issues in previous STEPS releases, improving the user modeling experience, as well as modernizing the existing code base in order to aid future development. The main contribution of this research is a new parallel stochastic reaction-diffusion solver supported by a sophisticated distributed mesh library. While maintaining similar performance and scalability, the new solver dramatically reduces the memory footprint of simulations, resolving the major bottleneck in previous solutions. This breakthrough empowers future neuroscience research by enabling super-large scale molecular reaction-diffusion simulations with biologically realistic models.

Data availability statement

The STEPS simulator is available at http://steps.sourceforge.net/. Models for validation and performance investigation, as well as simulation data presented in this publication are available at https://github.com/CNS-OIST/STEPS4ModelRelease/tree/Frontiers2022.

Author contributions

ED and FS conceptualized and led this study. TC and WC led the overall software development of STEPS 4. SM and TC added in Omega_h the support to Gmsh file format 4 and features required in STEPS 4. BD, SM, TC, and WC contributed to the Zee library development and evaluations. AC, BD, CK, GC, JL, SM, TC, and WC contributed to the software development of STEPS 4. AC, CK, IH, JL, TC, and WC contributed to the pre-release testing, debugging and optimization of STEPS 4. JL contributed to the python interface development for STEPS 4 and model conversions from STEPS 3 to STEPS 4. AC and IH designed and conducted the validation benchmarks. CK, GC, and WC designed and conducted the performance benchmarks. NC checked statistical soundness of the tests and contributed in CI. OA, JK, and PK contributed to technical discussions and supervised the BBP team. WC coordinated the writing of the paper. All authors gave feedback and contributed to the article and approved the submitted version.

Funding

Research reported in this publication was supported by the Okinawa Institute of Science and Technology Graduate

University (OIST) and funding to the Blue Brain Project, a research center of the École polytechnique fédérale de Lausanne (EPFL), from the Swiss government's ETH Board of the Swiss Federal Institutes of Technology and the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement No. 785907 (Human Brain Project SGA2).

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2022.883742/full#supplementary-material

References

Abhyankar, S., Brown, J., Constantinescu, E. M., Ghosh, D., Smith, B. F., and Zhang, H. (2018). Petsc/ts: a modern scalable ode/dae solver library. *arXiv preprint arXiv*:1806.01437. doi: 10.48550/arXiv.1806.01437

Amunts, K., Ebell, C., Muller, J., Telefont, M., Knoll, A., and Lippert, T. (2016). The human brain project: creating a european research infrastructure to decode the human brain. *Neuron* 92, 574–581. doi: 10.1016/j.neuron.2016.10.046

Amunts, K., Knoll, A. C., Lippert, T., Pennartz, C. M., Ryvlin, P., Destexhe, A., et al. (2019). The human brain project–synergy between neuroscience, computing, informatics, and brain-inspired technologies. *PLoS Biol.* 17, e3000344. doi: 10.1371/journal.pbio.3000344

Andrews, S. S., and Bray, D. (2004). Stochastic simulation of chemical reactions with spatial resolution and single molecule detail. *Phys. Biol.* 1, 137–151. doi: 10.1088/1478-3967/1/3/001

Antunes, G., and De Schutter, E. (2012). A stochastic signaling network mediates the probabilistic induction of cerebellar long-term depression. *J. Neurosci.* 32, 9288–9300. doi: 10.1523/JNEUROSCI.5976-11.2012

Anwar, H., Hepburn, I., Nedelescu, H., Chen, W., and De Schutter, E. (2013). Stochastic calcium mechanisms cause dendritic calcium spike variability. *J. Neurosci.* 33, 15848–15867. doi: 10.1523/JNEUROSCI.1722-13.2013

Anwar, H., Roome, C. J., Nedelescu, H., Chen, W., Kuhn, B., and De Schutter, E. (2014). Dendritic diameters affect the spatial variability of intracellular calcium dynamics in computer models. *Front. Cell Neurosci.* 8, 168. doi: 10.3389/fncel.2014.00168

Arjunan, S., and Tomita, M. (2010). A new multicompartmental reaction-diffusion modeling method links transient membrane attachment of *E. coli* MinE to E-ring formation. *Syst. Synth. Biol.* 4, 35–53. doi: 10.1007/s11693-009-9047-2

Arjunan, S. N., Miyauchi, A., Iwamoto, K., and Takahashi, K. (2020). pspatiocyte: a high-performance simulator for intracellular reaction-diffusion systems. *BMC Bioinform.* 21, 33. doi: 10.1186/s12859-019-3338-8

Ascoli, G. A., Donohue, D. E., and Halavi, M. (2007). NeuroMorpho.Org: a central resource for neuronal morphologies. *J. Neurosci.* 27, 9247–9251. doi: 10.1523/JNEUROSCI.2055-07.2007

Bhalla, U., Bilitch, D., and Bower, J. (1992). Rallpacks: a set of benchmarks for neuronal simulators. *Trends Neurosci.* 15, 453–458. doi: 10.1016/0166-2236(92)90009-W

Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., et al. (2016). "Caliper: performance introspection for HPC software stacks," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, UT: IEEE), 550–560.

Carnevale, N. T., and Hines, M. L. (2009). *The NEURON Book, 1st Edn.* Cambridge: Cambridge University Press.

Chen, W., and De Schutter, E. (2017). Parallel STEPS: Large scale stochastic spatial reaction-diffusion simulation with high performance computers. *Front. Neuroinform.* 11, 13. doi: 10.3389/fninf.2017.00013

Chen, W., Hepburn, I., Martyushev, A., and De Schutter, E. (2022). "Modeling neurons in 3d at the nanoscale," in *Computational Modelling of the Brain* (Cham: Springer), 3–24.

Cramér, H. (1928). On the composition of elementary errors: II, Statistical applications. *Scand. Actuar. J.* 11, 141–180. doi: 10.1080/03461238.1928.10416872

Denizot, A., Arizono, M., Nägerl, U. V., Soula, H., and Berry, H. (2019). Simulation of calcium signaling in fine astrocytic processes: effect of spatial properties on spontaneous activity. *PLoS Comput. Biol.* 15, e1006795-e1006795. doi: 10.1371/journal.pcbi.1006795

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., et al. (2015). "The spack package manager: bringing order to HPC software chaos," in *Supercomputing 2015 (SC'15).* (Austin, TX: LLNL-CONF-669890).

Gibson, M. A., and Bruck, J. (2000). Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A* 9, 104. doi: 10.1021/jp993732q

Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.* 81, 2340–2361. doi: 10.1021/j100540a008

Hattne, J., Fange, D., and Elf, J. (2005). Stochastic reaction-diffusion simulation with MesoRD. *Bioinformatics* 21, 2923–2924. doi: 10.1093/bioinformatics/bti431

Hepburn, I., Cannon, R., and De Schutter, E. (2013). Efficient calculation of the quasi-static electrical potential on a tetrahedral mesh and its implementation in steps. *Front. Comput. Neurosci.* 7, 129. doi: 10.3389/fncom.2013.00129

Hepburn, I., Chen, W., and De Schutter, E. (2016). Accurate reaction-diffusion operator splitting on tetrahedral meshes for parallel stochastic molecular simulations. *J. Chem. Phys.* 145, 054118. doi: 10.1063/1.4960034

Hepburn, I., Chen, W., Wils, S., and De Schutter, E. (2012). STEPS: efficient simulation of stochastic reaction-diffusion models in realistic morphologies. *BMC Syst. Biol.* 6, 36. doi: 10.1186/1752-0509-6-36

Hodgkin, A. L., and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* 117, 500–544. doi: 10.1113/jphysiol.1952.sp004764

Hoffmann, M., Fröhner, C., and No,é, F. (2019). Readdy 2: fast and flexible software framework for interacting-particle reaction dynamics. *PLoS Comput. Biol.* 15, e1006830. doi: 10.1371/journal.pcbi.1006830

Ibanez, D., and Roberts, N. (2018). *Omega_h.* [*Software*] Available online at: https://github.com/sandialabs/omega_h.

Ibrahim, K., Williams, S., and Oliker, L. (2018). "Roofline scaling trajectories: a method for parallel application and architectural performance analysis," in *2018 International Conference on High Performance Computing and Simulation (HPCS)* (Orleans: IEEE), 350–358.

Insel, T. R., Landis, S. C., and Collins, F. S. (2013). The nih brain initiative. *Science* 340, 687–688. doi: 10.1126/science.1239276

Kerr, R. A., Bartol, T. M., Kaminsky, B., Dittrich, M., Chang, J. C., Baden, S. B., et al. (2008). Fast monte carlo simulation methods for biological reaction-diffusion systems in solution and on surfaces. *SIAM J. Sci. Comput*. 30, 3126. doi: 10.1137/070692017

Markram, H., Meier, K., Lippert, T., Grillner, S., Frackowiak, R., Dehaene, S., et al. (2011). Introducing the human brain project. *Procedia Comput. Sci*. 7, 39–42. doi: 10.1016/j.procs.2011.12.015

Massey Jr, F. J. (1951). The kolmogorov-smirnov test for goodness of fit. *J. Am. Stat. Assoc*. 46, 68–78. doi: 10.1080/01621459.1951.10500769

Mohapatra, N., Tønnesen, J., Vlachos, A., Kuner, T., Deller, T., Nägerl, U. V., et al. (2016). Spines slow down dendritic chloride diffusion and affect short-term ionic plasticity of gabaergic inhibition. *Scientific Rep*. 6, 23196–23196. doi: 10.1038/srep23196

Murdoch, D. J., Tsai, Y.-L., and Adcock, J. (2008). *P*-values are random variables. *Am. Stat*. 62, 242–245. doi: 10.1198/000313008X332421

Noether, G. E. (1963). Note on the kolmogorov statistic in the discrete case. *Metrika* 7, 115–116. doi: 10.1007/BF02613966

Oliveira, R., Terrin, A., di benedetto, G., Cannon, R., Koh, W., Kim, M., et al. (2010). The role of type 4 phosphodiesterases in generating microdomains of camp: large scale stochastic simulations. *PLoS ONE* 5, e11725. doi: 10.1371/journal.pone.0011725

Patoary, M. N. I., Tropper, C., McDougal, R. A., Lin, Z., and Lytton, W. W. (2019). Parallel stochastic discrete event simulation of calcium dynamics in neuron. *IEEE/ACM Trans. Comput. Biol. Bioinform*. 16, 1007–1019. doi: 10.1109/TCBB.2017.2756930

Rodola, G. (2020). *psutil*. Available online at: https://github.com/giampaolo/psutil.v5.8.0.

Schelker, M., Mair, C. M., Jolmes, F., Welke, R.-W., Klipp, E., Herrmann, A., et al. (2016). Viral rna degradation and diffusion act as a bottleneck for the influenza a virus infection efficiency. *PLoS Comput. Biol*. 12, e1005075. doi: 10.1371/journal.pcbi.1005075

Schöneberg, J., and Noé, F. (2013). Readdy-a software for particle-based reaction-diffusion dynamics in crowded cellular environments. *PLoS ONE* 8, e74261. doi: 10.1371/journal.pone.0074261

Slepoy, A., Thompson, A. P., and Plimpton, S. J. (2008). A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks. *J. Chem. Phys*. 128, 205101. doi: 10.1063/1.2919546

Sodani, A., Gramunt, R., Corbal, J., Kim, H.-S., Vinod, K., Chinthamani, S., et al. (2016). Knights landing: second-generation intel xeon phi product. *IEEE Micro* 36, 34–46. doi: 10.1109/MM.2016.25

Stillman, N. R., Balaz, I., Tsompanas, M.-A., Kovacevic, M., Azimi, S., Lafond, S., et al. (2021). Evolutionary computational platform for the automatic discovery of nanocarriers for cancer treatment. *NPJ Comput. Mater*. 7, 1–12. doi: 10.1038/s41524-021-00614-5

Treibig, J., Hager, G., and Wellein, G. (2010). "LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures* (San Diego, CA).

Trott, C. R., Lebrun-Grandi,é, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., et al. (2022). Kokkos 3: programming model extensions for the exascale era. *IEEE Trans. Parallel Distribut. Syst*. 33, 805–817. doi: 10.1109/TPDS.2021.3097283

Von Mises, R. (1928). *Wahrscheinlichkeit Statistik und Wahrheit*. (Berlin:Springer).

Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: an insightful visual performance model for floating-point programs and multicore architectures. *ACM Commun*. 52, 65–76 doi: 10.1145/1498765.1498785

Zamora Chimal, C. G., and De Schutter, E. (2018). Ca2+ requirements for long-term depression are frequency sensitive in purkinje cells. *Front. Mol. Neurosci*. 11, 438. doi: 10.3389/fnmol.2018.00438

Zivanovic, D., Pavlovic, M., Radulovic, M., Shin, H., Son, J., Mckee, S. A., et al. (2017). Main memory in hpc: do we need more or could we live with less? *ACM Trans. Archit. Code Optim*. 14, 1–26. doi: 10.1145/3023362

TECHNOLOGY AND CODE
published: 31 October 2022
doi: 10.3389/fninf.2022.883700

Brian2CUDA: Flexible and Efficient Simulation of Spiking Neural Network Models on GPUs

Denis Alevi[1,2]*, *Marcel Stimberg*[3], *Henning Sprekeler*[1,2], *Klaus Obermayer*[2,4] and *Moritz Augustin*[2,4]

[1] Technische Universität Berlin, Chair of Modelling of Cognitive Processes, Berlin, Germany, [2] Bernstein Center for Computational Neuroscience Berlin, Berlin, Germany, [3] Sorbonne Université, INSERM, CNRS, Institut de la Vision, Paris, France, [4] Technische Universität Berlin, Chair of Neural Information Processing, Berlin, Germany

Graphics processing units (GPUs) are widely available and have been used with great success to accelerate scientific computing in the last decade. These advances, however, are often not available to researchers interested in simulating spiking neural networks, but lacking the technical knowledge to write the necessary low-level code. Writing low-level code is not necessary when using the popular Brian simulator, which provides a framework to generate efficient CPU code from high-level model definitions in Python. Here, we present Brian2CUDA, an open-source software that extends the Brian simulator with a GPU backend. Our implementation generates efficient code for the numerical integration of neuronal states and for the propagation of synaptic events on GPUs, making use of their massively parallel arithmetic capabilities. We benchmark the performance improvements of our software for several model types and find that it can accelerate simulations by up to three orders of magnitude compared to Brian's CPU backend. Currently, Brian2CUDA is the only package that supports Brian's full feature set on GPUs, including arbitrary neuron and synapse models, plasticity rules, and heterogeneous delays. When comparing its performance with Brian2GeNN, another GPU-based backend for the Brian simulator with fewer features, we find that Brian2CUDA gives comparable speedups, while being typically slower for small and faster for large networks. By combining the flexibility of the Brian simulator with the simulation speed of GPUs, Brian2CUDA enables researchers to efficiently simulate spiking neural networks with minimal effort and thereby makes the advancements of GPU computing available to a larger audience of neuroscientists.

Keywords: spiking neural networks, simulator, GPU, CUDA, Python, software, open-source, parallel algorithm

Edited by:
James Courtney Knight,
University of Sussex, United Kingdom

Reviewed by:
Jochen Martin Eppler,
Helmholtz Association of German Research Centres (HZ), Germany
Felix Benjamin Kern,
International Research Center for Neurointelligence (IRCN), Japan

***Correspondence:**
Denis Alevi
denis.alevi@tu-berlin.de

Received: *25 February 2022*
Accepted: *09 May 2022*
Published: *31 October 2022*

Citation:
Alevi D, Stimberg M, Sprekeler H, Obermayer K and Augustin M (2022) Brian2CUDA: Flexible and Efficient Simulation of Spiking Neural Network Models on GPUs.
Front. Neuroinform. 16:883700.
doi: 10.3389/fninf.2022.883700

1. INTRODUCTION

In computational neuroscience, there is high demand for computationally efficient simulations allowing for realtime applications or exhaustive parameter explorations. Efficient simulations require both optimized simulation software and powerful hardware. In practice, there is always a trade-off between the performance of the hardware and its price and accessibility. A promising technology with a very beneficial performance–cost trade-off are graphics processing units (GPUs) with their massively parallel arithmetic capabilities. While they were initially designed for computer graphics, they have since become commonly used for general-purpose computing, leading to their designation as general-purpose graphics processing units (GPGPUs). The most popular framework is the Compute Unified Device Architecture (CUDA; NVIDIA Corporation, 2007–2022) which allows users to write parallel code for GPUs in an extension of the C/C++ programming languages. To make efficient use of GPUs, simulation code has to perform computations in a highly parallel way. This parallelization is rather straightforward to implement for some aspects of neuronal models, e.g., the numerical integration of neuronal state variables over a simulation time step, but is non-trivial for other aspects, e.g., spike propagation with synaptic delays (cf. Brette and Goodman, 2012).

The earliest attempts at using the GPU (e.g., Bernhard and Keriven, 2006; Nageswaran et al., 2009) explored the general feasibility of accelerating simulations of spiking neural networks, and described many of the challenges that are still relevant today. To benefit from the capabilities of a GPU, a simulation needs to be parallelized efficiently, parallel memory access to shared memory has to be handled carefully, and synaptic connections have to be stored in sparse data structures to fit into the limited memory of GPUs (Nageswaran et al., 2009). The earliest implementations were typically technology demonstrations, but not released as software packages to be used by other researchers. This changed in the following years, as a number of general-purpose simulators such as NEMO (Fidjeland et al., 2009; Fidjeland and Shanahan, 2010), CNS (Mutch et al., 2010), CARLsim (Richert et al., 2011; Chou et al., 2018), and NCS6 (Hoang et al., 2013) were released. While these simulators could be adapted to a researcher's needs, they typically only supported specific neuron models or network structures: The NEMO, CARLsim, and NCS6 simulators were built to simulate networks of leaky integrate-and-fire or quadratic integrate-and-fire model (Izhikevich, 2003) neurons, and the CNS simulator was built to simulate networks structured in cortical layers. Extending these simulators to other models requires a researcher to write CUDA code and is therefore not accessible to many researchers without the necessary technical background.

Most recent simulators (e.g., Abi Akar et al., 2019; Panagiotou et al., 2021; Ben-Shalom et al., 2022) do not come with predefined neuron models, but instead translate neuron model definitions created for the NEURON simulator (Carnevale and Hines, 2006) or model definitions exported to NeuroML (Cannon et al., 2014) by a compatible simulator. An advantage of this approach is that it makes it possible to immediately reuse a large number of existing neuron models. On the other hand, this workflow is not ideal for researchers that want to adapt and change existing models, or introduce completely new ones. For these use cases, the fact that the model description and its simulation require more than one software package can be a major hurdle.

A number of simulators addressed this issue by using code generation (Goodman, 2010; Blundell et al., 2018). In such a framework, the model description in a convenient high-level or domain-specific language is an integral part of the simulator itself. When starting a simulation, these model descriptions are translated into efficient low-level code, compiled, and executed. Two simulators that have used this approach to generate code for GPUs are ANNarchy (Vitay et al., 2015), which specializes in networks that mix rate and spike-based elements, and GeNN (Yavuz et al., 2016), where model descriptions have to be specified in a variant of C++ (but note that the main simulation code can be written as a Python script *via* the PyGeNN interface; Knight et al., 2021b).

The Brian[1] simulator (Stimberg et al., 2019a) is a widely used neural simulator that provides a user-friendly system for model descriptions based on mathematical equations, as well as an extensible code generation framework. So far, this framework was only capable of generating C++ code for multithreaded execution on central processing units (CPUs). Recently, Brian's framework has been extended to generate code for the GeNN simulator (Brian2GeNN; Stimberg et al., 2020b), making it finally possible to run Brian simulations on the GPU. However, this approach limits simulations to the common feature set provided by Brian, GeNN, and the Brian2GeNN interface: some of Brian's features (e.g., multicompartmental models) are not supported by GeNN at all, and the support for other features (e.g., heterogeneous synaptic delays) was added after the creation of the Brian2GeNN interface, and they are therefore also unsupported at this time.

Here, we present a new approach to GPU code generation with the Brian simulator. This interface, named Brian2CUDA, directly generates CUDA code for the GPU, and supports the full set of features that the Brian simulator offers. It can therefore be used as a drop-in replacement in all situations where multithreaded CPU code generation was used previously, including simulations of detailed network models of neurons, synapses, and glia cells (Stimberg et al., 2019b), or when optimizing neuronal models with the brian2modelfitting toolbox (Teska et al., 2020).

We describe how our approach exploits non-trivially parallelizable simulation parts, in particular the data structures and algorithms for the propagation of neuronal spikes through a network taking into account – potentially hetereogeneous – synaptic delays. For several relevant generic model classes, we compare the performance of Brian2CUDA with Brian's built-in multithreaded execution on CPUs and – where possible – with the Brian2GeNN interface. The results show that Brian2CUDA strongly outperforms the multithreaded execution on CPUs, sometimes by orders of magnitudes. Its performance is comparable to the performance of Brian2GeNN. For large networks, Brian2CUDA is faster, while for smaller networks slightly slower.

Our code is available as open source software under a free license at GitHub: https://github.com/brian-team/brian2cuda.

2. METHOD

Brian2CUDA implements a new Brian backend, which runs spiking neural network simulations on NVIDIA graphics processing units (GPUs). It makes use of Brian's code generation system to generate C++/CUDA code based on a user's model definition in Python.

[1] Note that while its Python package is named "brian2," we will use the name "Brian" in this paper for simplicity.

In the following, we provide in Section 2.1 background on the Brian simulator and describe how our proposed CUDA backend can be used. In Section 2.2, we outline GPU programming essentials. Section 2.3 contains the algorithms implemented in Brian2CUDA including neuronal state updates, spike propagation, and synaptic effect application. Section 2.4 summarizes the alternative CUDA-based simulator Brian2GeNN and in Section 2.5 we specify the benchmark models and experimental procedure.

2.1. Brian Simulation and Code Generation

Brian is a simulator for spiking neural networks written in Python (Stimberg et al., 2014, 2019a). It is designed to be highly flexible and easy to use by using its own domain language to define models. This allows users to define arbitrary differential equations in Python strings. As an example, consider the model (Brunel and Hakim, 1999) depicted in **Figure 1A**. It consists of a population of N leaky integrate-and-fire (LIF) neurons with sparse random recurrent inhibitory connections, which are driven by Gaussian white noise. This model can be described by the differential equation depicted in **Figure 1B**. Since the inhibitory feedback is strong enough, the model exhibits fast global oscillations in the population firing rate while the single neuron firing rates remain small (see **Figure 1C**). A Python script that implements this model in Brian is shown in **Figure 1D**. By changing two lines of code, the simulation can be switched from Brian's C++ backend to our new Brian2CUDA backend.

Both backends generate simulation code in their target language (C++ or CUDA) which is then compiled and executed. The generated code implements the simulation loop, memory management, and all computations; it can be executed independently of Python. **Figure 1E** illustrates the Brian C++ backend, showing a simplified example of the generated C++ code for updating all neuronal states at a single time step of the simulation. In our example, one central processing unit (CPU) thread sequentially updates the membrane voltage V_i for each neuron i. To speed up simulations, the Brian C++ backend can be configured to use OpenMP to parallelize computations over multiple CPU threads (not shown here). Our Brian2CUDA backend extends the C++ backend to generate C++/CUDA code. The simulation loop and memory management are implemented in C++ and executed on a single CPU thread, while most computations are implemented in CUDA and are parallelized on the GPU. **Figure 1F** shows the same neuronal state updates as before, but now implemented in CUDA. The voltages of all neurons are updated in parallel by all available threads on a GPU.

2.2. GPU Programming With CUDA

To implement software that runs on NVIDIA GPUs, the Compute Unified Device Architecture (CUDA) programming model is used. CUDA works with multiple programming languages, and here we use the CUDA API implemented in C++.

2.2.1. CUDA Programming Logic

2.2.1.1. Thread Hierarchy

A typical C++/CUDA program is executed on a single CPU thread, which calls special GPU functions that are executed on the GPU (see **Figure 2A**). These functions are called CUDA *kernels*. When called, kernels execute their code in parallel by multiple CUDA *threads* (see **Figure 2B**), which are grouped into CUDA *blocks* (see **Figure 2C**). The number of threads per block N_{threads} and the number of blocks N_{blocks} in this thread hierarchy is set when calling the kernel (see **Figures 2A,D**).

2.2.1.2. Memory Hierarchy

Each GPU has its own memory, which is separate from the CPU's memory. GPU memory is split into different types, which are hierarchically organized (see **Figures 2B–D**). *Global memory* is large (several gigabytes depending on specific hardware) and accessible by all threads, but memory access is very slow. *Shared memory* is accessible by all threads within the same thread block. This memory is much faster to access, but limited in size (up to a few megabytes split across all blocks). And finally, each thread has its own *registers* with the fastest access time, but which are also limited in number (up to a few megabytes split across all threads). Threads use registers to store the intermediate results during their computations, shared memory to communicate intermediate results during kernel execution between threads of the same block and global memory to communicate between threads in different blocks and to store results between kernel calls.

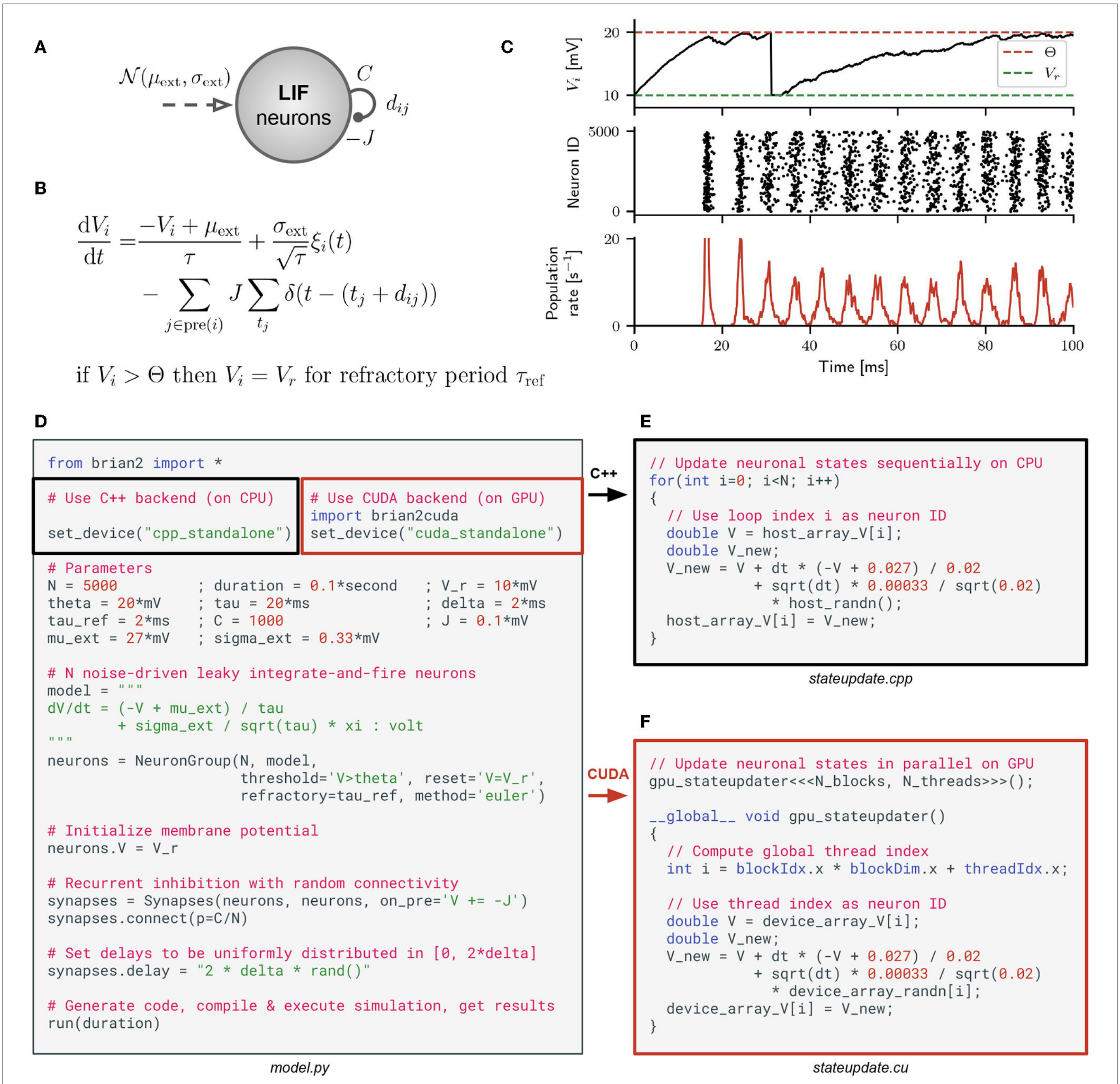
2.2.2. Execution Control Logic

On the hardware level, NVIDIA GPUs consist of multiple streaming multiprocessors (SMs). During the execution of CUDA kernels, thread blocks are assigned to streaming multiprocessors (SMs) (see **Figure 2**). Each SM can execute a limited number of blocks concurrently, which are referred to as *active* blocks. All remaining thread blocks are queued for execution on the next available slot on any of the SMs. The maximal number of active blocks per SM depends on the resource requirements of the executed kernel and resource limits per SM (e.g., how many registers are required vs. available). When a block is executed on an SM, its threads are executed in groups of 32 threads, which are called *warps*. Each thread of a warp executes the same instructions at each clock cycle, which implements the single instruction multiple threads (SIMT) paradigm.

2.2.3. Performance Considerations

2.2.3.1. Occupancy

Occupancy per SM is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. Given the number of threads and blocks of a kernel and its resource requirements, an upper occupancy limit can be determined, the *theoretical occupancy*. There are multiple hardware limits that determine how well a kernel can be parallelized on the GPU. Here, we will only introduce a few of them which are relevant for our algorithms. Each SM has a limit on the number of threads in all active blocks, a limit on registers available to all threads in all active blocks, and a general limit on the number of active blocks. If any of these limits is exceeded, the number of active blocks per SM is automatically reduced such that the



$$\frac{\mathrm{d}V_i}{\mathrm{d}t} = \frac{-V_i + \mu_\text{ext}}{\tau} + \frac{\sigma_\text{ext}}{\sqrt{\tau}}\xi_i(t) - \sum_{j \in \text{pre}(i)} J \sum_{t_j} \delta(t - (t_j + d_{ij}))$$

if $V_i > \Theta$ then $V_i = V_r$ for refractory period τ_ref

```
from brian2 import *

# Use C++ backend (on CPU)

set_device("cpp_standalone")
```

```
# Use CUDA backend (on GPU)
import brian2cuda
set_device("cuda_standalone")
```

```
# Parameters
N = 5000          ; duration = 0.1*second    ; V_r = 10*mV
theta = 20*mV     ; tau = 20*ms              ; delta = 2*ms
tau_ref = 2*ms    ; C = 1000                 ; J = 0.1*mV
mu_ext = 27*mV    ; sigma_ext = 0.33*mV

# N noise-driven leaky integrate-and-fire neurons
model = """
dV/dt = (-V + mu_ext) / tau
        + sigma_ext / sqrt(tau) * xi : volt
"""
neurons = NeuronGroup(N, model,
                      threshold='V>theta', reset='V=V_r',
                      refractory=tau_ref, method='euler')

# Initialize membrane potential
neurons.V = V_r

# Recurrent inhibition with random connectivity
synapses = Synapses(neurons, neurons, on_pre='V += -J')
synapses.connect(p=C/N)

# Set delays to be uniformly distributed in [0, 2*delta]
synapses.delay = "2 * delta * rand()"

# Generate code, compile & execute simulation, get results
run(duration)
```

model.py

```
// Update neuronal states sequentially on CPU
for(int i=0; i<N; i++)
{
  // Use loop index i as neuron ID
  double V = host_array_V[i];
  double V_new;
  V_new = V + dt * (-V + 0.027) / 0.02
            + sqrt(dt) * 0.00033 / sqrt(0.02)
              * host_randn();
  host_array_V[i] = V_new;
}
```

stateupdate.cpp

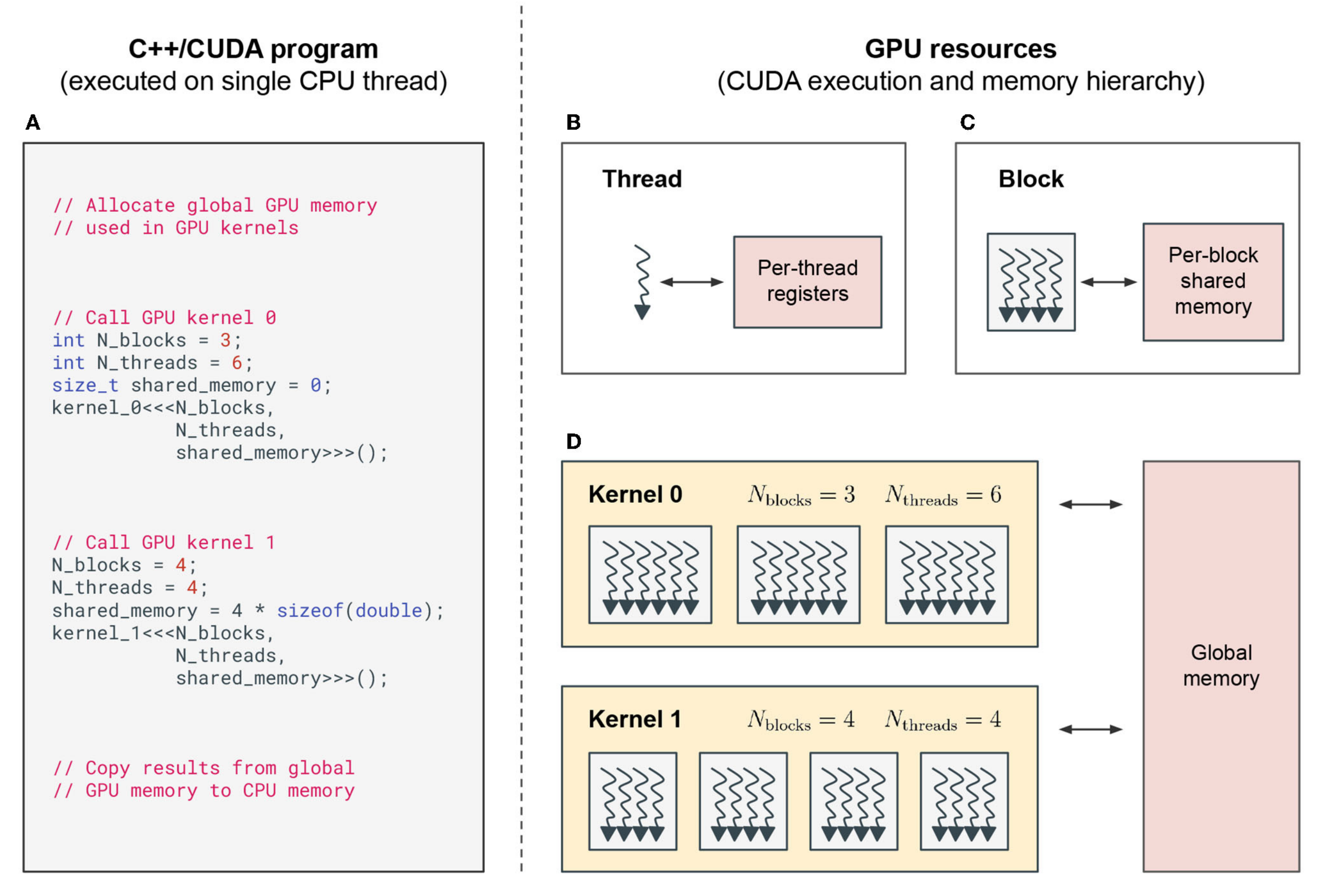
```
// Update neuronal states in parallel on GPU
gpu_stateupdater<<<N_blocks, N_threads>>>();

__global__ void gpu_stateupdater()
{
  // Compute global thread index
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  // Use thread index as neuron ID
  double V = device_array_V[i];
  double V_new;
  V_new = V + dt * (-V + 0.027) / 0.02
            + sqrt(dt) * 0.00033 / sqrt(0.02)
              * device_array_randn[i];
  device_array_V[i] = V_new;
}
```

stateupdate.cu

FIGURE 1 | Brian model definition and C++/CUDA code generation. **(A)** Population of leaky integrate-and-fire (LIF) neurons with recurrent inhibitory coupling J, average number of random synapses per neuron C and synaptic transmission delays d_{ij} between neurons j and i. The neurons are driven by external Gaussian white noise with mean μ_ext and standard deviation σ_ext (model from Brunel and Hakim, 1999). **(B)** Corresponding stochastic differential equation defining the dynamics of the membrane potential V_i of a single LIF neuron i [with $i = 1, \ldots, N$; membrane time constant τ; unit Gaussian white noise process $\xi_i(t)$ that is uncorrelated across neurons; $j \in \text{pre}(i)$ runs over all neurons j that are presynaptic to neuron i; t_j are all spike times of neuron j; Dirac delta function $\delta(x)$]. When the voltage V_i crosses threshold Θ, the neuron spikes and is set to the reset voltage V_r for a refractory period τ_ref. **(C)** Network dynamics from simulating the model with $N = 5000$ LIF neurons in Brian. Top panel: voltage trace for one exemplary neuron i. Middle panel: raster plot of the spike times for all neurons in the network. Bottom panel: instantaneous mean firing rate across all neurons. **(D)** A Python script implementing the model in Brian, either with its C++ backend (black blox) or with Brian2CUDA's CUDA backend (red box). In this example, the synaptic transmission delays are independently sampled from a uniform distribution $d_{ij} \sim \mathcal{U}(0, 4)$ ms. **(E)** Simplified version of generated C++ code to update all neuronal states defined by the voltages V_i when using the C++ backend in Brian. **(F)** The same for the CUDA backend in Brian2CUDA. Here the CUDA kernel `gpu_stateupdater` is launched with $N_\text{blocks} \times N_\text{threads}$ parallel threads.


C++/CUDA program
(executed on single CPU thread)
A
GPU resources
(CUDA execution and memory hierarchy)
B
Thread
Per-thread registers
C
Block
Per-block shared memory
D
Kernel 0
$N_{\mathrm{blocks}} = 3$
$N_{\mathrm{threads}} = 6$
Kernel 1
$N_{\mathrm{blocks}} = 4$
$N_{\mathrm{threads}} = 4$
Global memory

```
// Allocate global GPU memory
// used in GPU kernels


// Call GPU kernel 0
int N_blocks = 3;
int N_threads = 6;
size_t shared_memory = 0;
kernel_0<<<N_blocks,
           N_threads,
           shared_memory>>>();


// Call GPU kernel 1
N_blocks = 4;
N_threads = 4;
shared_memory = 4 * sizeof(double);
kernel_1<<<N_blocks,
           N_threads,
           shared_memory>>>();


// Copy results from global
// GPU memory to CPU memory
```

FIGURE 2 | CUDA programming model. **(A)** A simplified, exemplary C++/CUDA program that is executed on a single CPU thread. The CPU manages memory on the GPU and calls CUDA kernels that are executed on the GPU. **(B–D)** GPU resources and CUDA execution and memory hierarchy when running CUDA kernels on the GPU. **(B)** Each CUDA thread on a GPU has access to its own memory registers. **(C)** Each CUDA block groups together multiple CUDA threads. All threads of the same block have access to the same shared memory. **(D)** CUDA kernels can be executed with different numbers of blocks N_{blocks}, threads per block N_{threads} and shared memory per block. The kernels called in (A) are executed sequentially on the GPU, while multiple CUDA blocks are executed in parallel on the streaming multiprocessors (SMs) of the GPU. The example program in (A) calls kernel 0 without any shared memory and kernel 1 with enough shared memory to store one floating point number per thread. This memory could e.g., be used to calculate the sum of a variable over all threads in a block, using fast shared memory instead of slow global memory.

limits are fulfilled, reducing the theoretical occupancy of the kernel. **Table 1** lists these limits for the GPUs used in this work.

2.2.3.2. Coalesced Memory Access

Memory accesses are issued in warps or half-warps (depending on the GPU and the memory request). When accessing global memory, always chunks of 32, 64 or 128 bytes are transferred – even if less memory was requested. That means, if a single thread wants to read 4 byte from global memory, a 32 byte transfer will be issued (the smallest transfer possible). If multiple threads read 4 byte from different non-contiguous memory addresses in global memory, there will be one 32 byte transfer per thread. But if all threads in a warp request 4 byte memory from contiguous memory addresses, a single $32 \times 4\,\mathrm{byte} = 128\,\mathrm{byte}$ transfer will be issued to transfer all memory requested by all threads. This is called a *coalesced* memory access, which reduces latencies (i.e., waiting time) for global memory accesses significantly. Hence, it is crucial to layout data structures such that as many memory accesses as possible are coalesced.

2.3. Brian2CUDA Algorithms

Brian is a clock-driven simulator, which performs the same set of computations after each discrete time step Δt of a simulation. In this section, we will explain the algorithms and data structures used in Brian2CUDA by going through the different simulation steps necessary to simulate one time step of the recurrent LIF network from **Figure 1**. All data structures introduced in the following reside in global GPU memory and all kernels introduced are executed sequentially on the GPU.

TABLE 1 | Hardware limits relevant to determine theoretical occupancy of GPU kernels.

GPU (cc)	Active threads per SM	Threads per block	Active blocks per SM	Registers per SM	Registers per thread for 100% occupancy
A100 (8.0)	2,048	1,024	32	65,536	32
RTX2080 Ti (7.5)	1,024	1,024	16	65,536	64

The values are the same for all GPUs with the same compute capability (cc) and represent upper limits. To get the maximal registers per thread for 100 % occupancy, we divided the maximal registers per SM by the maximally active threads per SM. Only GPUs used in our experiments are listed.

2.3.1. Neurons

In Brian, neurons are defined in populations, where each neuron is described by the same set of dynamical equations and hence the same set of state variables (e.g., V_i in **Figure 1B**). In Brian2CUDA, for each neuronal population typically three separate CUDA kernels are defined: one for integrating the neuronal states (see **Figure 1F**), one for detecting spikes and one for resetting state variables of spiking neurons[2]. Since these computations are independent for all neurons, parallelization on the GPU is trivial: Each thread performs all computations for a single neuron. Nevertheless, it is important to coalesce global memory access (see Section 2.2.3.2). This is ensured in the integration kernel by storing neuronal state variables in contiguous global memory arrays (one entry per neuron) and accessing those such that consecutive threads access consecutive entries of the state variable arrays. In the spike detection kernel, the threshold crossing (typically of the membrane voltage) can be detected efficiently in parallel in the same way as in the integration kernel. The challenge here is to select and count the spiking neurons of the current timestep which naturally involves serialization. The implemented solution relies on a method from the CUDA programming API: threads that detect a spike perform an atomic increment of a population spike counter and use the counter value to store their neuron ID in a spiking neuron array. An atomic operation is an operation on a single variable that can be safely called by multiple threads, guaranteeing that all updates get applied correctly. These atomic increments limit the parallelization in the spike detection kernel when multiple threads try to increment the counter at the same time, and the writing of spiking neuron IDs into the spiking neuron array is generally not coalesced. The reset kernel is parallelized over spiking neurons and therefore the reading of spiking neuron IDs is coalesced, but the reset updates of the neuronal state variables are generally not. Since for the majority of models in computational neuroscience, the number of spikes per time step is much lower than the number of neurons per population, the potentially inefficient computations in the spike detection and reset kernels often contribute only little to the total computation time[3].

2.3.2. Synapses

A population of synapses in Brian is defined between a pre- and a postsynaptic population of neurons (for the recurrent synapses defined in **Figure 1D**, pre- and postsynaptic populations are the same). The simulation of synapses can generally be separated into synapse generation, synaptic state updates, spike propagation and synaptic effect application. The synapse generation in Brian2CUDA is performed on the CPU, using the same algorithm as Brian's C++ backend and thereby supporting all of Brian's connection methods. Synaptic state updates in Brian can be *clock-driven* or *event-driven*. Clock-driven updates are performed at every time step and are implemented in Brian2CUDA in a separate kernel in the same way as neuronal state updates. Event-driven updates are performed only when the pre- or postsynaptic neuron of a synapse spikes. These are performed during the synaptic effect application of the corresponding spike. With spike propagation, we refer to the processing of synaptic delays, which can be either *homogeneous* (the same for all synapses) or *heterogeneous* (varying across synapses). With synaptic effect applications, we refer to the modifications of synaptic target variables based on spikes (e.g., the reduction of the postsynaptic voltage potential by J for each presynaptic spike in the model from **Figure 1**). In Brian, both the pre- and postsynaptic spikes can have synaptic effects on pre- and postsynaptic neurons and the synapse itself. In the following, we will illustrate how Brian2CUDA implements spike propagation and synaptic effect application for different delay types and for the case of presynaptic neurons that modify postsynaptic variables, but the algorithms generalize to all other synaptic effect types. For both, spike propagation and effect application, kernels are parallelized over synapses in Brian2CUDA.

2.3.2.1. Connectivity Information

Consider the example connectivity for our recurrent LIF network shown in **Figure 3A**, where synaptic effects are triggered by presynaptic spikes. The (sparse) connectivity matrix of synapse IDs sorted by presynaptic neuron ID is stored in YALE format (**Figure 3B**; Eisenstat et al. 1982). This connectivity matrix can optionally (via a Brian2CUDA preference) be split into multiple partitions of postsynaptic neurons, in which case synapses per presynaptic neuron are sorted by partition (**Figure 3C**). This creates *synapse groups* defined by presynaptic neuron and postsynaptic partition (different colors in **Figure 3C**). If synaptic effects are triggered by postsynaptic spikes, e.g., for models with spike-timing dependent plasticity (STDP), a separate connectivity matrix is created, sorted by postsynaptic neurons

[2] Note that using separate kernels allows us to support Brian's flexible execution scheduling, e.g., synaptic effect application between threshold detection and reset.
[3] In a population of neurons firing at $1\,\mathrm{s}^{-1}$ and a simulation time step of $\Delta t = 0.1$ ms, on average only 0.01 % of the neurons spike at each time step.

and partitioned by presynaptic neurons (not shown here). To access the neurons connected by a synapse, two additional arrays store the pre- and postsynaptic neuron IDs for all synapses, sorted by synapse ID (**Figure 3D**).

2.3.2.2. Synapses Without Delays

When synapses have no transmission delay, there is no need for a separate spike propagation phase, and synaptic effects can be applied directly after spike detection. Effect application is parallelized over all synapses of all spiking neurons, where each CUDA block processes the synapses of one synapse group (**Figure 3E**). Each thread reads one synapse ID from the connectivity matrix, uses it as an index to read the postsynaptic neuron ID of that synapse, which is then used to index the postsynaptic membrane voltage. The synaptic effect (decreasing V_{post} by J in our inhibitory LIF network example) is performed using atomic operations to avoid race conditions from multiple threads writing to the same memory location, cf. **Figure 3E**. Reading synapse IDs from the connectivity matrix is coalesced, while reading the corresponding postsynaptic neuron IDs and membrane voltages is generally not. Partitioning the connectivity matrix can increase occupancy for networks with homogeneous delays if the overall number of spikes per time step is small enough (less than the limit on maximally active CUDA blocks; see **Table 1**) and the number of synapses per neuron is large enough (more than there are threads in each CUDA block). Under such conditions and without partitioning, there are too few active blocks with too many threads to parallelize all synaptic effects. Partitioning the connectivity matrix then moves threads from too full blocks into new active blocks, increasing parallelization.

2.3.2.3. Synapses With Homogeneous Delays

When synapses have homogeneous delays $d = k\Delta t$, the spiking neuron array is stored for k time steps before the synaptic effects are applied. This results in a circular list of $k + 1$ spiking neuron arrays. **Figure 3F** shows an example for $k = 2$. The synaptic effect application algorithm is the same as for the no delay case (see **Figure 3E**), but using the neurons that spiked k time steps ago. Spike propagation for networks with homogeneous delays amounts to incrementing the circular list index referencing the spiking neuron array that is due for synaptic effect application. Therefore, adding homogeneous delays to a network comes at close to no computational cost at each time step, but increases memory requirements for storing multiple spiking neuron arrays.

2.3.2.4. Synapses With Heterogeneous Delays

In networks with heterogeneous synaptic delays, synapses connected to spiking neurons are sorted into *spike queues* based on their synaptic delay. Analogously to the spiking neuron arrays used in the homogeneous delays case (cf. **Figure 3F**), $k + 1$ spike queues are created, which are arranged in a circular list and where k is the number of time steps in the highest delay in the network $\max(d_{ij}) = k\Delta t$. As before, the connectivity matrix can be partitioned by postsynaptic neurons, in which case each partition gets its own spike queues. To reduce the number of elements that need to be inserted into spike queues, the synapses with the same delay that would be propagated together, are grouped into *synapse bundles* and those bundles are inserted into the spike queues instead of synapses. **Figure 4A** shows the example network from **Figure 3**, but now with heterogeneous delays and additionally with synapse bundle IDs (instead of just synapse IDs). **Figure 4B** shows how the spike propagation algorithm sorts bundle IDs into spike queues. Since the maximal number of synapse bundles that will be stored in any of the spike queues is generally not known before a simulation, a custom dynamic vector implementation is used, which allows increasing spike queue sizes in GPU kernels on demand. This resizing requires reallocating spike queue contents in global GPU memory. While this is generally very expensive, it only happens at the beginning of a simulation until the spike queues are large enough and hence has an overall negligible effect on performance.

During spike propagation, parallelization is over synapse bundles, where each CUDA block operates on a different *bundle group* (different colors in **Figure 4B**; analogous to the synapse groups in **Figure 3C**). All CUDA blocks for the same postsynaptic partition and for different spiking neurons collect synapse bundles in the same spike queues. To avoid race conditions from potential memory reallocation, a critical section code allows only one CUDA block per partition to add bundle IDs into the spike queues at any time. All threads of this block can parallelize the pushing of synapse bundle IDs into the spike queues over threads, since each bundle with a different delay will be added to a different queue. Note that CUDA blocks from different postsynaptic partitions operate on different spike queues and can be executed concurrently. Therefore, increasing the number of partitions decreases the amount of serialization during spike propagation of heterogeneous delays. This can lead to better performance as long as the additional CUDA blocks don't exceed the maximal number of active CUDA blocks on the GPU (see **Table 1**).

During synaptic effect application, the synaptic effects of all synapses in the bundles in the $0\Delta t$ spike queue are applied. In our toy example, where we only consider a single time step, these are all synapses without delays (**Figure 4C**). In general, multiple different synapses from neurons that spiked at different times are collected in each spike queue. **Figure 4D** shows how the synaptic effect application parallelizes over synapses. The number of CUDA blocks during effect application equals the number of partitions. A fixed number of CUDA threads per synapse bundle performs the effect application for all synapses of each bundle. In the present work, the largest bundle size is used as the number of threads per bundle, but this can be set by the user. Bundle sizes depend on the delay distribution and number of synapses per neuron in the network. If all bundles have the same size, each thread applies the synaptic effects of one synapse. The more bundle sizes vary, the less efficient is the parallelization given a fixed number of threads per bundle. In general, spike propagation performance benefits from partitioning the connectivity matrix as long as the typical size of the spike queue is larger than the number of threads per CUDA block.

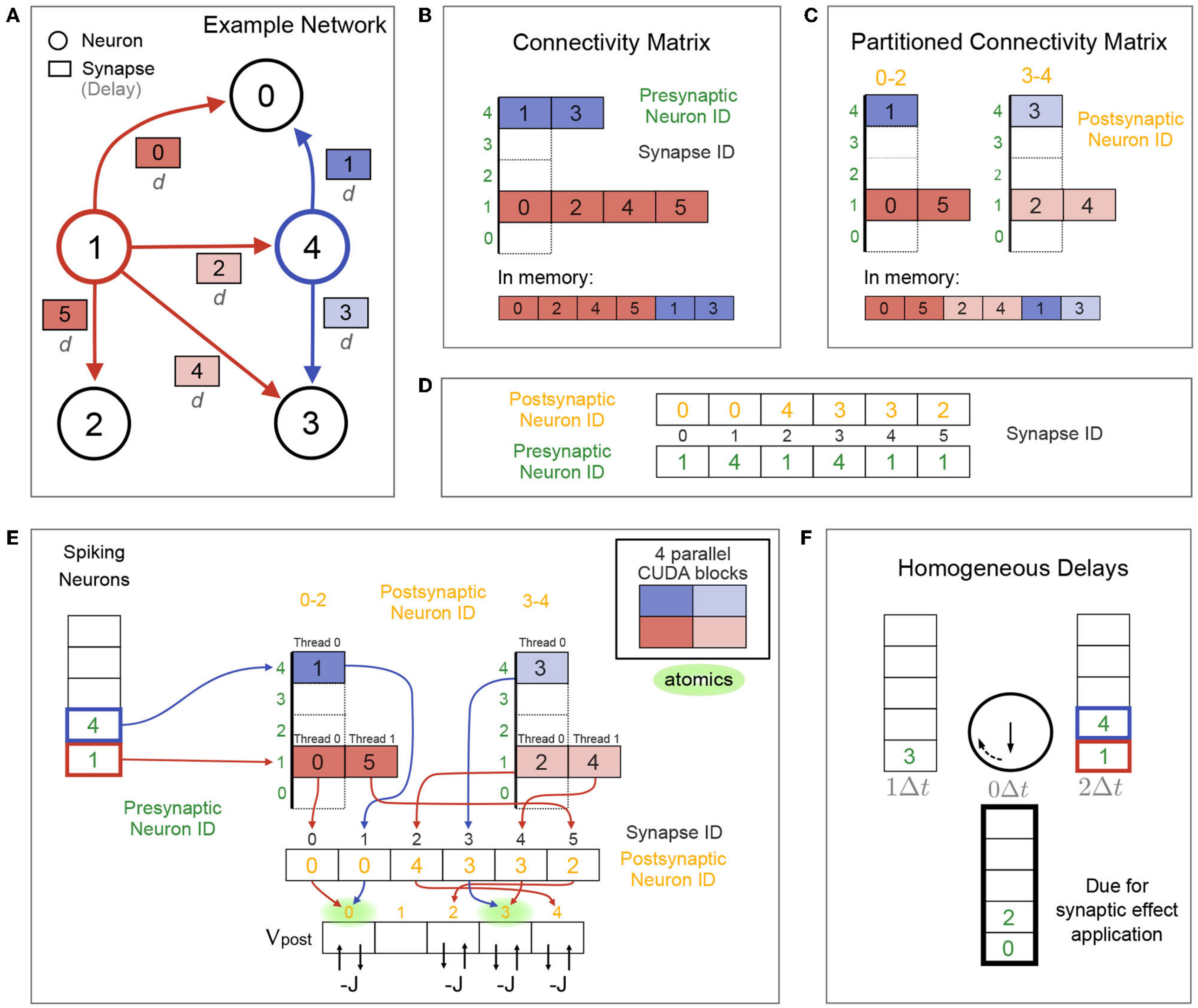


FIGURE 3 | Synaptic algorithm for networks with no or homogeneous delays. **(A)** Example connectivity for the recurrent network from **Figure 1**, restricted to homogeneous synaptic transmission delays d and $N = 5$ neurons. Colored neurons 1 and 4 are spiking in the current time step. Color of their synapse IDs correspond to the parallelization over CUDA blocks in (E). **(B)** Connectivity stored in compressed form (YALE format) in global GPU memory as one concatenated array of synapse IDs sorted by presynaptic neuron ID (bottom view). Top view shows this array split by presynaptic neurons for visualization. Two additional arrays (not shown) store the start indices and number of synapses in the synapse array for each presynaptic neuron. Coloring correspond to the parallelization over CUDA blocks in (E). **(C)** Connectivity matrix for two postsynaptic neuron partitions, visualized as in (B). Each color shows one synapse group, defined by presynaptic neuron (red or blue) and postsynaptic partition (bright or dark). The synapse array is sorted in memory first by presynaptic neuron ID and then by partition (bottom view). **(D)** Pre- and postsynaptic neuron IDs for all synapses are stored in two arrays, sorted by synapses IDs. **(E)** Fully parallelized synaptic effect application for the network from (A) without delays ($d = 0\Delta t$) and with the partitioned connectivity matrix from (C). Each of the 4 CUDA blocks (cf. colors) applies synaptic effects for all synapses of its respective synapse group. Membrane voltage updates are performed using CUDA's atomic operations to avoid race conditions. Potential atomic conflicts at the same memory location are marked in green. Without connectivity matrix partioning (B), only two CUDA blocks (one per spiking neuron) would process the synapses (not shown). **(F)** Circular list of spiking neuron arrays for the network from (A) with homogeneous delays $d = 2\Delta t$. Spiking neuron arrays are labeled with the time in which their synaptic effects are due for application. Spiking neurons of the current time step are stored in the array labeled with $d = 2\Delta t$. Synaptic effects are applied for the neurons in the array labeled with $0\Delta t$. After each time step, all array labels are rotated clockwise and the applied spiking neuron array will be overwritten by the new spikes of the next time step.

2.4. CUDA Code Generation With Brian2GeNN

Our benchmarks compare Brian2CUDA's performance with the performance obtained when using the Brian2GeNN interface (Stimberg et al., 2020b). Since both are implemented as backends for the Brian 2 simulator, the exact same models can be run and easily compared. Note that the Brian2GeNN interface does not support synaptic connections with heterogeneous

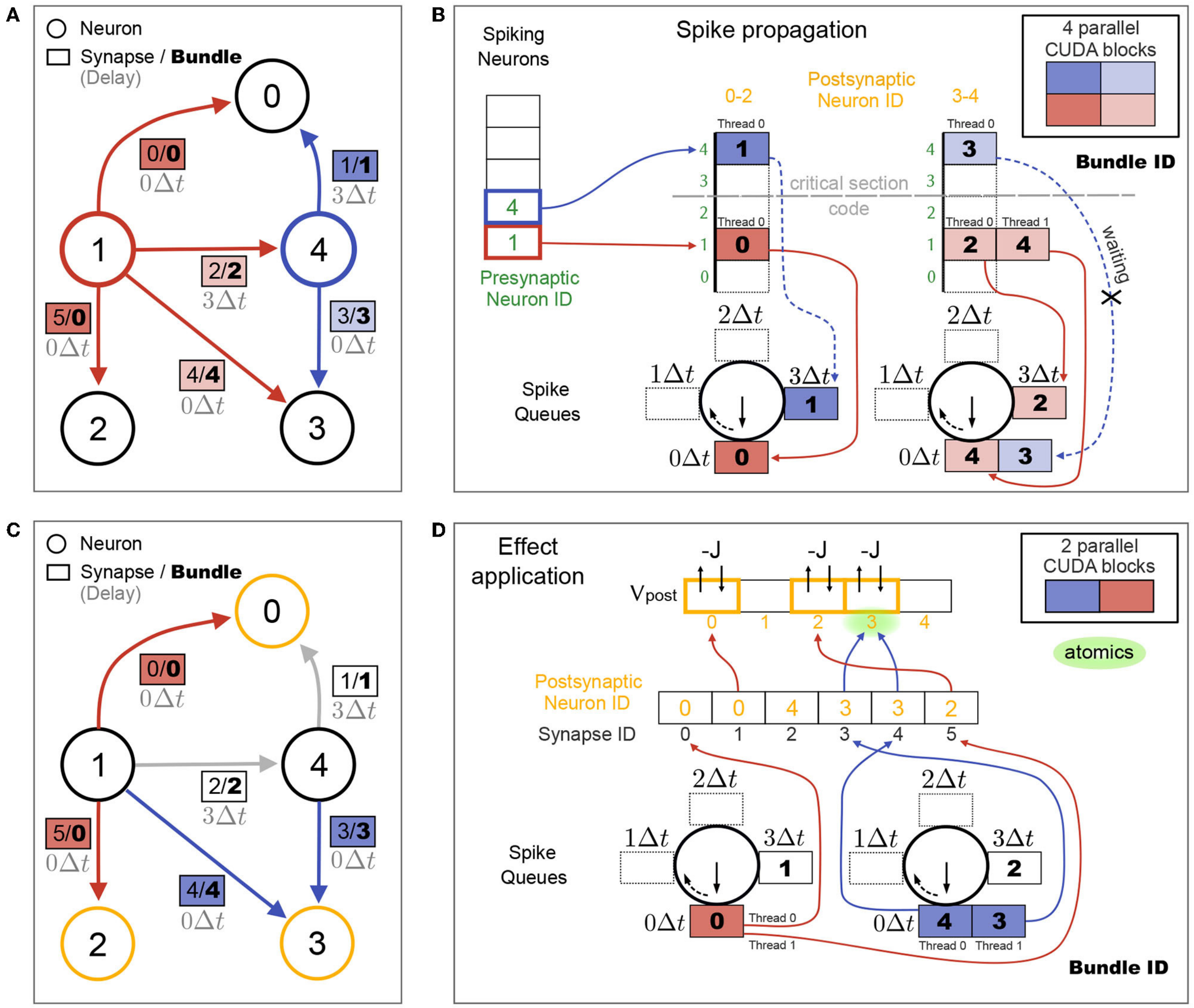


FIGURE 4 | Spike propagation and synaptic effect application for synapses with heterogeneous delays. **(A)** Same connectivity as shown in **Figure 3A**, but with heterogeneous delays d_{ij} from neuron j to i. Neurons spiking in the current time step and their outgoing synapses are colored. Colors of synapse labels correspond to the parallelization over CUDA blocks during spike propagation in (B). Bundles group together synapses with same presynaptic neuron, postsynaptic partition as shown in (B) and delay value. All bundles for the same presynaptic neuron and postsynaptic partition define a bundle group (same color), each with a different delay. In this toy example, only bundle 0 has two synapses (0 and 5), the other bundles contain only one synapse. Additionally only the bright red bundle group consists of two bundles (2 and 4), while the other bundle groups contain only one bundle. **(B)** Spike propagation step. Bundles for all spiking neurons are sorted into spike queues based on their delay value and postsynaptic partition. Maximal delay in (A) is $d_{0,4} = 3\Delta t$, requiring 4 spike queues per partition. Each of the 4 CUDA blocks propagates all bundles of its respective bundle group. A critical section code ensures that only one CUDA block per partition (red or blue) has access to the spike queues of its partition at any time. Two CUDA blocks of different partitions (dark and light) can operate concurrently on separate spike queues. For each partition, delay queues are constructed as a circular list of arrays and their labels are rotated at the end of each time step [after (D)] analogously to the circular list of spiking neuron arrays in the case of homogeneous delays in **Figure 3F**. **(C)** Synaptic effect application. Same connectivity as in (A). Colors now indicate neurons receiving synaptic effects in the current time step (yellow) and their incoming synapses (red and blue, these are the synapses from (A) without delay). Colors of synapse labels correspond to the parallelization over CUDA blocks during effect application in (D). **(D)** Effect application step. Synaptic effects of all synapses in all bundles in the $0\Delta t$ spike queues are applied to their targets. One CUDA block per partition processes all bundles of its partition. Bundles are unpacked and each thread applies the effect on one synapse (e.g, two threads are processing the two synapses in bundle 0).

delays, therefore the corresponding benchmarks only compare Brian2CUDA to CPU performance.

The Brian2GeNN interface uses Brian's C++ framework to generate synaptic connections, initialize variables, and to generate the numerical update steps based on the given model equations. In the next step, the interface converts synaptic data structures and model descriptions to the GeNN format, and runs GeNN's own code generation process. Finally, the generated code gets integrated into a run loop running on the CPU that also takes care of exchanging memory between CPU and GPU when necessary (for details see Stimberg et al., 2020b). The internally used data structures and algorithms are identical to running a simulation with the GeNN simulator (for details see Yavuz et al., 2016). GeNN allows the user to choose data structures and algorithms most adapted to their model, and many of these choices are exposed in the Brian2GeNN interface. All benchmarks presented in this paper use GeNN's sparse connectivity method, and chose the—for the respective model configuration—faster of its two parallelization modes: *pre mode*, i.e., parallelization over pre-synaptic sources and sequential loops over post-synaptic targets, or *post mode*, i.e., parallelization over post-synaptic targets and sequential loops over pre-synaptic sources.

2.5. Benchmarks

2.5.1. Benchmark Models

To assess the runtime performance of Brian2CUDA in comparison to Brian2 on CPU and Brian2GeNN we use as benchmarks different models that cover popular types used in computational neuroscience. Here, we give an overview of the model characteristics and behaviors. The simulation code with all model implementations, parameters and benchmark procedures that were used to generate the results of this paper are available in our Brian2CUDA GitHub repository[4] and archived as Alevi et al. (2022).

2.5.1.1. HH Benchmark: Hodgkin-Huxley Type Neurons With Static Synapses

For the first benchmark, we use a model of excitatory and inhibitory conductance-based Hodgkin-Huxley (HH) type neurons (also used in Brette et al. 2007; Stimberg et al. 2020b and based on Traub and Miles 1991). This neuron model consists of six coupled ordinary differential equations describing the dynamics of the membrane voltage, three gating variables, and excitatory and inhibitory synaptic conductances. We initialized membrane voltages and synaptic conductances independently from Gaussian distributions, such that all neurons had slightly different initial conditions (for details see Stimberg et al., 2020b). We simulated populations of N neurons (80% excitatory and 20% inhibitory) with random recurrent synapses. Synapses from spiking presynaptic excitatory and inhibitory neurons modify postsynaptic excitatory and inhibitory conductances based on their synaptic weights w_E and w_I, respectively. Connectivity was randomly Bernoulli-sampled for each pair of neurons (including self-connections) with fixed probability $p = \frac{C}{N}$, where $C =$ 1,000 is the average number of synapses per neuron. For $N <$ 1,000, all neuron pairs were connected. The model is identical to the *COBAHH benchmark* in Stimberg et al. (2020b), where a mathematical description of the model and a list of parameters can be found.

For **Figure 5A**, we simulated N neurons without any synapses, i.e., an uncoupled HH-type population. For an example of the activity in this network, see **Supplementary Figure S2**. For **Figure 5B**, we simulated the model with an average of $C =$ 1,000 synapses per neuron and with random synaptic weights uniformly sampled from $w_\mathrm{E}, w_\mathrm{I} \sim \mathcal{U}(0, w_\mathrm{max})$ with $w_\mathrm{max} = 10^{-18}\,\mathrm{S}$. The weights are chosen small enough to have no substantial effect on postsynaptic conductances such that the network activity does not change when increasing the population size, but synaptic propagation and effect application is still performed during the simulation (same procedure as in Stimberg et al., 2020b).

2.5.1.2. LIF Benchmark: Noisy Integrate-and-Fire Neurons With Synaptic Transmission Delays

The LIF benchmark consists of a population of N noise-driven LIF neurons with recurrent inhibitory connections (based on Brunel and Hakim, 1999). This is the same model we introduced in **Figure 1**. The dynamics of each neuron are described by a single ordinary differential equation for the membrane voltage shown in **Figure 1B**. For all benchmark results, we simulated the model with spike threshold $\Theta = 20\,\mathrm{mV}$, reset potential $V_r = 10\,\mathrm{mV}$, membrane time constant $\tau = 20\,\mathrm{ms}$ and inhibitory coupling $J = 0.1\,\mathrm{mV}$. Neurons have a refractory period of $\tau_\mathrm{ref} =$ 2 ms. Recurrent random connectivity is implemented in the same way as in the HH benchmark, with connection probability $p = \frac{C}{N}$ with the same average number of synapses per neuron $C =$ 1,000. Synapses from spiking presynaptic neurons modify postsynaptic membrane voltages.

For the benchmark version with homogeneous delays (**Figures 5C**, **7A**), the synaptic transmission delay was $d_{ij} = 2\,\mathrm{ms}$ for each synapse from neuron j to neuron i. The parameters of the external drive (Gaussian white noise) were chosen as $\mu_\mathrm{ext} = 25\,\mathrm{mV}$ and $\sigma_\mathrm{ext} = 1\,\mathrm{mV}$. For the benchmark version with heterogeneous delays (**Figures 6A**, **7C**), the synaptic transmission delays were uniformly sampled from a uniform distribution $d_{ij} \sim \mathcal{U}(0, 4)\,\mathrm{ms}$. Resolved on the integration time grid with $\Delta t = 0.1\,\mathrm{ms}$, this resulted in up to 41 different delay values. The external drive parameters were chosen as $\mu_\mathrm{ext} =$ 27 mV and $\sigma_\mathrm{ext} = 0.33\,\mathrm{mV}$. These parameters ensured that both benchmark versions had the same mean synaptic delay and that network activities showed qualitatively similar slow global oscillations (see Brunel and Hakim 1999; example activity for the heterogeneous version with $N =$ 5,000 shown in **Figure 1C**).

2.5.1.3. STDP Benchmark: Dynamic Synapses With Spike-Timing Dependent Plasticity

The spike-timing dependent plasticity (STDP) benchmark consists of N Poisson generators with dynamic feedforward synapses to a population of $\frac{N}{1{,}000}$ LIF neurons (for an example of the activity in the network, see **Supplementary Figure S3**). The Poisson generators have no dynamics that need to be integrated, but produce random Poisson spike trains with a mean firing rate of $15\,\mathrm{s}^{-1}$ (each generator performs one independent Bernoulli trial per time step). The dynamics of the LIF neurons are described by two differential equations, one for the membrane voltage and one for an excitatory synaptic conductance. The connection probability is $p = \frac{C}{N}$, where $C =$ 1,000 is the average number of incoming synapses per LIF neuron (while each Poisson generator has on average only one outgoing synapse). Each synapse has a dynamic weight, which determines the

[4] https://github.com/brian-team/brian2cuda/tree/paper2022/brian2cuda/tools/benchmarking.

synaptic effect that a presynaptic spike has on the postsynaptic neuron. The dynamics of the weights implement an all-to-all, additive STDP rule (Song et al., 2000; Morrison et al., 2008). Each pre- and postsynaptic spike increases a corresponding trace variable, stored for each synapse (necessary to support heterogeneous delays). In the absence of spikes, these trace variables decay exponentially, which is implemented through an event-driven update (see Section 2.3.2), and is therefore only calculated when necessary. When a presynaptic spike arrives, the current postsynaptic trace is used to decrease the synaptic weight, and conversely a postsynaptic spike triggers an increase in the synaptic weight based on the current presynaptic trace. Together, these changes implement the observed asymmetry of the STDP rule, where a presynaptic spike followed by a postsynaptic spike leads to synaptic facilitation, and the inverted sequence leads to synaptic depression (Bi and Poo, 2001). The technically challenging aspect of this model is that there are multiple synaptic effects triggered by pre- and postsynaptic spikes: synaptic trace variables are bidirectionally affected and additionally presynaptic spikes influence postsynaptic neurons *via* increasing the excitatory synaptic conductance. In Brian2CUDA, for this model, two separate connectivity matrices are generated, one for pre- and one for postsynaptically triggered synaptic effects. Both matrices are sorted differently, the former one by pre- and the latter one by postsynaptic neurons (see Section 2.3.2.1).

We use two versions of this model for benchmarking: with homogeneous delays (**Figures 5D**, **7B**) and with heterogeneous delays (**Figures 6B**, **7D**). The delays are the same as in the LIF benchmarks with the corresponding delay type. Note that the transmission delays are implemented as axonal delays, i.e., they only apply to synaptic effects triggered by the presynaptic population, while the synaptic effects from the postsynaptic population have no delays.

2.5.1.4. Mushroom Body Benchmark: Complex Model With Multiple Neuronal Populations, Spike-Timing Dependent Plasticity and Noise

As the final benchmark (for **Figure 5E**), we consider a more "realistic," complex model with multiple neuronal populations and synapse types, that combines several of the features of the previous benchmarks. For an example of the activity in the network, see **Supplementary Figure S4**. This model is inspired by the mushroom body of insects, based on the model by Nowotny et al. (2005), and used as a benchmark in earlier studies (Yavuz et al., 2016; Stimberg et al., 2020b). Briefly, this model consists of three populations: the first population consists of 100 pattern generators (i.e., does not simulate any dynamics but replays a pre-defined spike pattern), connecting to N HH-type neurons in the second population with a connection probability of $p = 0.15$ for each possible connection (Bernoulli sample). These connections are modeled as static, excitatory synapses. The neurons of the second population are modeled with the same equations (but different parameters) as in the HH benchmark presented earlier, except that they have no inhibitory conductance, which is not required without inhibitory synapses. This second population connects further to a third population of 100 HH-type neurons, with a connection probability of $p = \frac{10{,}000}{N}$ (with all-to-all connectivity for $N < 10{,}000$). These connections are plastic, following the STDP rule presented in the STDP benchmark. Finally, the third population has recurrent synapses to itself with all-to-all connectivity and static inhibitory synapses. For more details and parameters of this model, see Yavuz et al. (2016) and Stimberg et al. (2020b).

2.5.2. Benchmark Procedure

All our benchmarks running on GPUs were executed on a single A100 data-center GPU (40 GB global memory), except for some results in **Figure 9**, which were executed on a single consumer-level GeForce RTX2080 Ti GPU (11 GB global memory). Brian's C++ backend was executed on an Intel Xeon Gold 6226R CPU with 16 physical cores, using 16 threads. Benchmarks were run on Brian2CUDA commit-tag `paper2022`[5] (Alevi et al., 2022), Brian version 2.4.2 (Stimberg et al., 2020a), GeNN version 4.5.1 (Knight et al., 2021a) and Brian2GeNN commit `5f844d0` (based on version 1.6; Stimberg et al., 2021). We modified the Brian and Brian2GeNN versions with custom patches to execute our benchmarks and to get more detailed profiling information than available in the original implementations. Note that we ensured that these modifications had no significant impact on the runtime durations. The correct versions of these packages are stored as Git submodules in our GitHub repository, together with the necessary patch files and instructions on how to apply them. C++ code was compiled with gcc version 9.3.0 and CUDA code was compiled with nvcc version 11.2 based on CUDA toolkit version 11.2. The operating system on the computers with the A100 GPUs and Intel Xeon Gold 6226R CPUs was CentOS Linux release 7.4.1708 and on the computers with RTX2080 GPUs it was Ubuntu Linux version 20.04.3 LTS.

For all benchmarks, we first recorded network activities for different network sizes and inspected that network activities were as expected. Additionally, we compared the results of Brian's C++ backend with the results of the Brian2CUDA backend for validation. For the final computation time measurements, we disabled the recording of any network activities. All benchmarks were simulated once for 10 s biological time (except for **Figure 8**) with a simulation time step of $\Delta t = 0.1$ ms, and the computation times were divided by 10 to produce computation times relative to biological time (referred to as *Time [comp / bio]* in our figures). Simulations that exceeded 1,000 s of total computation time were interrupted before the end of the simulation (except for **Figure 8**). The computation time for the entire simulation was then linearly extrapolated based on the fraction of biological time that was simulated (data points marked in all figures). All figures except for **Figure 8** show the computation time only for the main simulation loop, which consists of all simulation kernels that are executed at each time step of a simulation. This time does therefore not include compilation, network initialization, synapse generation, or result storage. For the Brian2CUDA profiling simulations in **Figure 7**, the CPU and GPU were synchronized after each kernel launch (forcing the CPU to wait for the kernel to terminate before continuing execution, which results in increased computation

[5] https://github.com/brian-team/brian2cuda/tree/paper2022

time) and kernel times were measured using timing functions in C++ code. For the Brian2GeNN profiling experiments, Brian2GeNN's own kernel timing preference was enabled, which records kernel times with CUDA events and without additional CPU/GPU synchronization. All Brian2CUDA simulations of benchmarks with no or homogeneous delays were executed without partitioning the connectivity matrix. For benchmarks with heterogeneous delays, the number of connectivity matrix partitions is shown in the figure labels or captions. In all benchmarks with heterogeneous delays, synapse bundles are used (and not individual synapses as Brian2CUDA can be configured to do). For **Figure 8**, the STDP benchmark with homogeneous delays was simulated for the biological times and network sizes shown in the figure legends and simulations were not interrupted after 1,000 s computation time. Code generation and compilation times were recorded from within the Brian package. To measure the initialization and finalization times, we computed the difference between the time spent within the main loop of the generated code and the total execution time of the compiled binary.

3. RESULTS

To illustrate how different model features affect simulation performance on GPUs in Brian2CUDA and what speedup levels are typical, we consider multiple benchmark models covering various model types often used in computational neuroscience. In Sections 3.1–3.4, we focus on the computation time needed for the main simulation loop, which is the part of the simulation that is executed at every simulation time step. In particular, we summarize simulation performance for models without synaptic delays or with homogeneous delays in Section 3.1 and for models with heterogeneous delays in Section 3.2. In Section 3.3, we analyze the contributions of different algorithm parts to the runtime and in Section 3.4, we illustrate how recording of network activity and state variables influences runtimes. Section 3.5 then quantifies the overhead of preparing a simulation in terms of compilation and synapse initialization runtime beyond the main simulation loop. Finally, we show in Section 3.6 how the performance depends on the choice of floating point precision (single vs. double) and specific GPU hardware.

3.1. Benchmark Models Without Delays or With Homogeneous Synaptic Delays

3.1.1. Hodgkin-Huxley Benchmark

To make efficient use of GPUs, simulation code has to perform highly parallel computations. The independent integration of neuronal state variables performed at each simulation time step for all neurons is trivial to parallelize on a GPU. In Brian2CUDA, each GPU thread computes the full state update for a single neuron (see Section 2.3.1). For a network of Hodgkin-Huxley (HH) type neurons without synapses, Brian2CUDA achieves a speedup of 3 orders of magnitude compared to Brian's single-threaded C++ backend for large enough network sizes ($N > 10^5$; **Figure 5A**). The speedup of the GPU backend implemented in Brian2GeNN is comparable to the speedup of Brian2CUDA. With single-precision floats as shown here, Brian2CUDA performs slightly better than Brian2GeNN for large network sizes (**Figure 5A**), while for double-precision floats this difference is negligible (shown in **Supplementary Figure S1F**). Both backends also have comparable memory requirements, but Brian2GeNN is slightly more efficient. For example, on an RTX2080 Ti with 11 GB memory, Brian2GeNN can simulate a network that has about 1.4 times the size of the biggest network that can be simulated with Brian2CUDA (about $2.8 \cdot 10^8$ vs. $2.0 \cdot 10^8$ neurons).

Next we turn to networks with synapses, where the application of postsynaptic effects is less trivial to parallelize, since the effects of multiple spikes at the same target neuron cannot be applied at the same time in GPU memory (see Section 2.3.2.2). We therefore extend our benchmark model to a network of recurrently coupled HH-type neurons with conductance-based synapses without transmission delays (**Figure 5B**). In this model, each neuron has on average 1,000 synapses. To analyze the particular effect of the added recurrent synapses, we ensured that they do not change the network activity. In this benchmark, Brian2CUDA still achieves a speedup of 3 orders of magnitude compared to Brian's single-threaded C++ backend for large enough network sizes ($N > 10^5$; **Figure 5B**). Notably, Brian2CUDA performs roughly 5 times faster than Brian2GeNN for the largest investigated network size ($N = 10^6$), while being 2–3 times slower for smaller network sizes ($N < 10^4$). The performance differences for small networks can be explained by the sequential execution from multiple small kernels in Brian2CUDA compared to the execution of fewer merged kernels in Brian2GeNN, see Section 4 for more details. In comparison to the network without synapses, the speedups gained through parallel computations in the multithreaded C++ backend and the GPU backends are reduced by a factor of 2–5 when including synapses (see **Figures 5A** vs. **5B**). This illustrates that synaptic computations are generally less parallelizable than neuronal computations.

3.1.2. Leaky Integrate-and-Fire Benchmark

The speedups of the GPU backends for the HH benchmarks demonstrate that neuronal computations benefit much more from parallelizations on the GPU than synaptic computations. Consequently, models for which the single-threaded C++ backend spends relatively less time for neuronal computations, should benefit less from computations on the GPU. To illustrate this effect, we next consider a population of noise-driven recurrently connected leaky integrate-and-fire (LIF) neurons with homogeneous synaptic transmission delays (based on Brunel and Hakim, 1999). This benchmark has the same number of synapses per neuron as the HH benchmark, but its neurons are described by only one dynamic state variable, compared to six state variables in the HH neuron model. Therefore, the single-threaded C++ backend spends relatively less of the overall computation time for neuronal computations when using LIF neurons. While Brian2CUDA still achieves a speedup of almost 3 orders of magnitude compared to Brian's single-threaded C++ backend for large enough network sizes ($N > 10^5$; **Figure 5C**),

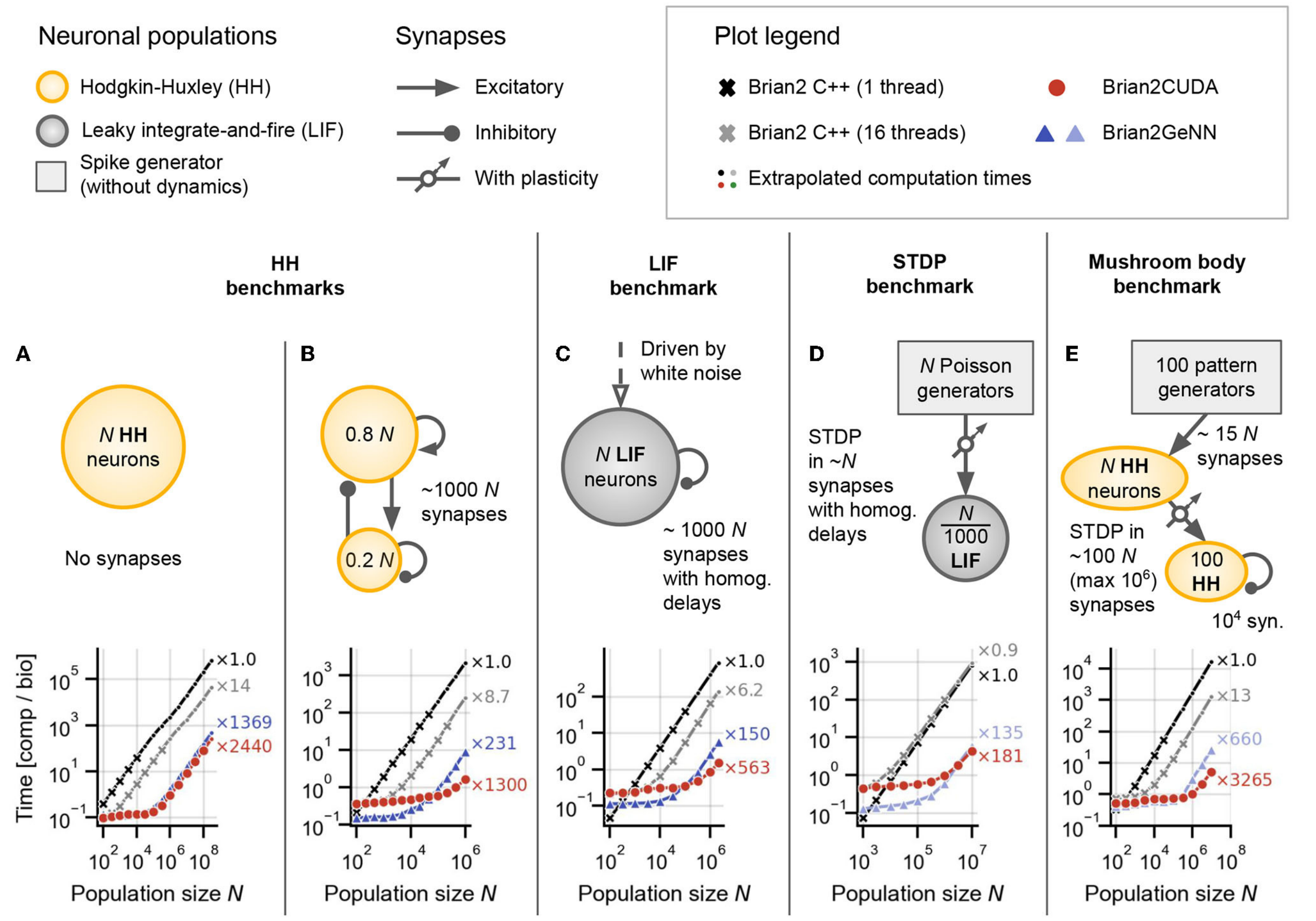


FIGURE 5 | Benchmark results for networks without delays or with homogeneous delays. **(A)** Hodgkin-Huxley (HH) population without synapses. **(B)** Sparsely coupled recurrent HH network with 80% excitatory and 20% inhibitory neurons, without synaptic delays. **(C)** Leaky integrate-and-fire (LIF) network with sparse random connectivity and homogeneous synaptic delays $d_{ij} \equiv d = 2$ ms for all synapses. **(D)** Spike-timing dependent plasticity (STDP) benchmark with homogeneous delays $d_{ij} \equiv d = 2$ ms. **(E)** Mushroom body benchmark with non-plastic synapses in the first layer and synapses with STDP in the second layer, both randomly connected and without delays. For all panels: The text annotations on the right of the axes show the factor by which each simulation was faster than Brian's single-threaded C++ backend (i.e., obtained speedup) at the largest displayed *N*. Brian2CUDA was simulated without partitioning the connectivity matrix in all simulations (corresponding to **Figure 3B**). Brian2GeNN was simulated using its *post* parallelization strategy for **(A–C)** (dark blue) and *pre* parallelization strategy for **(D,E)** (light blue), which was the respective faster simulation mode compared to the other (not shown). All simulations were run once for 10 s biological time. All times shown are computation times for the main time loop (i.e., without code generation, compilation, synapse generation or network initialization/finalization) and are relative to the simulated biological time. All simulations were interrupted if the main time loop took longer than 1,000s and the total computation time was extrapolated based on the fraction of biological time that was simulated (simulations for which this was done are indicated by small circular markers). All simulations were performed with Brian's single-precision preference for floating point numbers, i.e., 32-bit arithmetic, on A100 GPUs.

the speedup is approximately halved compared to that of the recurrent HH benchmark (**Figures 5B** vs. **5C**). Note that the addition of homogeneous synaptic transmission delays comes at almost no additional computational cost in Brian2CUDA (see Section 2.3.2.3). In relation to Brian2GeNN, Brian2CUDA performs 3 − 4 times better for the largest network sizes ($N \geq 10^6$), while being 2 − 3 times slower for smaller network sizes ($N < 10^4$). As observed previously, Brian2GeNN is more memory-efficient than Brian2CUDA. It is able to simulate this benchmark on an RTX2080 Ti for a network with more than $2.0 \cdot 10^6$ neurons, about 2.3 times the size supported by Brian2CUDA (about $8.6 \cdot 10^5$ neurons).

3.1.3. Spike-Timing Dependent Plasticity Benchmark

The benchmarks presented so far are based on static synapses, which do not change over the course of the simulation. However, an important subfield of computational neuroscience is interested in synaptic plasticity, where synaptic weights continuously adapt. Of particular interest in spiking neural networks are spike-timing dependent plasticity (STDP) rules, where the change in synaptic weight depends on the precise timing of pre- and post-synaptic spikes (Bi and Poo, 2001). Such plasticity rules present particular challenges for GPU acceleration, since they require more complex memory access patterns during the spike effect application phase than common static synapse models (cf. Brette and Goodman, 2012). To investigate the acceleration of models with STDP, we next examine a network with dynamic feedforward synapses from a large population of N Poisson generators to a much smaller population of $\frac{N}{1,000}$ LIF neurons (**Figure 5D**). The synapses in this network have (again) homogeneous transmission delays. Brian2CUDA achieves here 2 orders of magnitude speedup compared to Brian's single-threaded C++ backend for large enough network sizes ($N > 10^6$), but the speedup is reduced by a factor of 3 compared to the LIF benchmark. This is due to the increased relative computation time required for synaptic computations from the STDP learning rule (as will be shown in more detail below). Compared to Brian2GeNN, Brian2CUDA is again slower for small network sizes ($N \leq 10^6$) while being slightly faster for the largest network size ($N = 10^7$).

3.1.4. Mushroom Body Benchmark

In the final benchmark on simulations without or with homogeneous delays, we consider a model of an insect mushroom body based on Nowotny et al. (2005), in an implementation already used in Stimberg et al. (2020b). It is a three-layer network with HH-type neurons and STDP in some of its synapses (**Figure 5E**). Since this model includes HH-type neurons with relatively few synapses, most of the computational effort is spent on the integration of the neuronal state variables. More precisely, there are two operational regimes: For smaller network sizes ($N \leq 10^4$), the number of synapses is 2 orders of magnitude higher than the number of neurons in the network and the performance is comparable to the HH benchmark with synapses (cf. **Figure 5B**). For larger network sizes ($N \geq 10^5$), the number of synapses is only 1 order of magnitude higher than the number of neurons and the performance is closer to that of the HH benchmark without synapses (cf. **Figure 5A**). Surprisingly, Brian2CUDA's speedup for large network sizes in the mushroom body model is even larger than for the HH benchmark without synapses. This behavior is probably due to an extra dynamic inhibitory conductance variable in the neuron model of the HH benchmark, which requires additional registers during the neuronal integration. Due to hardware limits of available registers on the GPU, this decreases the maximal theoretical occupancy of the neuronal integration kernel to 62.5 % for the HH benchmark compared to 100 % in the mushroom body benchmark (48 registers per thread vs. 32 registers per thread; cf. Section 2.2.3.1 and **Table 1**). Note that the number of registers needed by a kernel is not easily predictable, and does not directly reflect the number of state variables. The number of intermediate computation steps in the chosen integration method, additional temporary variables introduced by Brian's code generation process, but also the CUDA compute capability of the GPU and even seemingly irrelevant details such as the order of variable declarations, all affect register usage. This demonstrates that minor differences in a model can have large effects on performance.

Note that for Brian2GeNN, which performs about 5x slower for the largest network size, we again show the performance of its *pre* parallelization mode, for which the performance is better at larger network sizes. For smaller network sizes, Brian2GeNN in *post* parallelization mode performs slightly better (see **Supplementary Figure S1E**).

All results in **Figure 5** are from simulations with single-precision floats and with preferences that gave the best performance. Results for additional preferences and simulations with double-precision floats are shown in **Supplementary Figure S1**.

3.2. Benchmark Models With Heterogeneous Synaptic Delays

Brian supports the simulation of networks with heterogeneously distributed synaptic delays. To simulate such networks, presynaptic spikes have to be sorted by delay and stored before their synaptic effects are applied. This spike propagation is challenging to parallelize efficiently on GPUs and additionally influences the parallelization of the synaptic effect application (Brette and Goodman 2012; Section 2.3.2.4). To evaluate the performance of Brian2CUDA's spike propagation and effect application algorithms, we include heterogeneously distributed synaptic delays in our LIF and STDP benchmarks, without qualitatively changing their network dynamics. We further evaluate Brian2CUDA's performance when partitioning its connectivity matrix (see Section 2.3.2.1).

Note that while the GeNN simulator has recently added support for heterogeneously distributed synaptic delays, this feature is currently not available in the Brian2GeNN interface. We therefore compare Brian2CUDA's performance only to Brian's C++ backends.

Figures 6A,B show the results for the LIF and STDP benchmarks, respectively. The performance of Brian's single-threaded C++ backend is not significantly affected by the presence of heterogeneous delays, while Brian2CUDA's performance drops by an order of magnitude for the LIF benchmark and between one and three orders of magnitude for the STDP benchmark (cf. **Figures 5C,D**), depending on the partitioning of the connectivity matrix. Note that Brian's multithreaded C++ backend does not efficiently parallelize spike propagation or the computations for Poisson generators, and hence performs similarly as its single-threaded backend in the STDP benchmark. Partitioning the connectivity matrix has little effect on overall runtime for the LIF benchmark, but increases performance by up to two orders of magnitude in the STDP benchmark. This strongly depends on the number of partitions and best performance was reached for 64 partitions (**Figure 6C**). To understand the effects of partitioning the connectivity matrix, we next consider profiling experiments to analyze the contribution of different parts of the simulation to the overall runtime.

3.3. Runtime Decomposition Into Different Algorithm Parts

We examine the contributions of the different algorithm parts by profiling the simulation. The following individual runtimes are available: the computation times for (1) performing neuron related computations (integration of dynamics and spike detection), (2) spike propagation and (3) synaptic effect

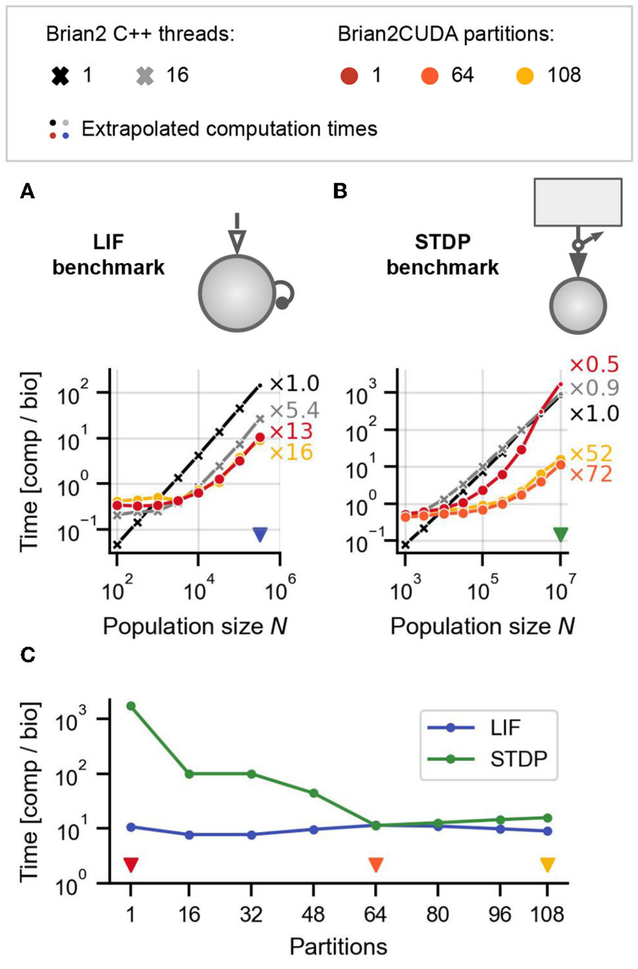


FIGURE 6 | Benchmark results for networks with heterogeneous delays. **(A)** LIF benchmark model from **Figure 5C** but with heterogeneous delays. **(B)** STDP benchmark from **Figure 5D** but with heterogeneous delays. Delays for all synapses in both models **(A,B)** were uniformly sampled $d_{ij} \sim \mathcal{U}(0, 4)\,ms$. The external drive for the LIF benchmark was additionally modified to maintain the same regime of network activity as in the case of homogeneous delays (see Section 2.5.1.2). **(C)** Computation times for LIF (blue line) and STDP (green line) benchmarks for different numbers of connectivity matrix partitions in Brian2CUDA, for the maximal network sizes from **(A,B)** [indicated with blue and red triangular markers in **(A,B)**]. Triangular markers in **(C)** indicate the number of partitions plotted in **(A,B)** in the corresponding colors. All simulations were performed as described in **Figure 5**.

application (including the event-driven integration of synaptic dynamics in the STDP benchmark). The decomposed runtimes for the LIF and STDP benchmarks with homogeneous delays are shown in **Figures 7A,B**, those for heterogeneous delays are contained in **Figures 7C,D**.

Brian's multithreaded C++ backend spends around half the computation time for spike propagation and synaptic effect application in the LIF benchmarks (**Figures 7A,C** yellow), while spending almost all time in the neuronal state updates and Poisson spike generation in the STDP benchmarks (**Figures 7B,D** blue). This is because the ratio of synapses to neurons (including Poisson generators) is much lower in the STDP benchmark compared to the LIF benchmark. The speedup of the GPU backends compared to the multithreaded C++ backend comes mostly from parallelizing the neuronal state updates and Poisson spike generation (including random number generation) on the GPU. When comparing Brian2CUDA and Brian2GeNN, both require similar times for the neuronal state updates and Poisson spike generation, but their efficiency for the synaptic effect applications differs for both benchmarks with homogeneous delays (**Figures 7A,B**). For the LIF benchmark, Brian2CUDA's synaptic effect application is more efficient compared to Brian2GeNN since the former parallelizes CUDA threads over all synapses while the latter parallelizes over postsynaptic neurons, requiring sequential looping over presynaptic spikes (using the *post* parallelization strategy of Brian2GeNN). In the STDP benchmark on the other hand, Brian2CUDA is only slightly more efficient in the synaptic effect application because Brian2GeNN's *pre* parallelization strategy is particularly suited to the case of many spiking neurons and few postsynaptic partners as explained above.

For heterogeneous delays, Brian2CUDA spends most of the computation time on spike propagation and synaptic effect application relative to neuronal state updates (**Figures 7C,D**). For the LIF benchmark with heterogeneous delays, increasing the number of partitions increases spike propagation times but decreases synaptic effect application times (**Figure 7C**). Each neuron in this benchmark has on average 1,000 synapses grouped into 41 synapse bundles per partition (see Section 2.3.2.4). Without partitioning the connectivity matrix, each CUDA block sorts all synapse bundles of one spiking neuron into spike queues, using one CUDA thread per bundle. This results in small CUDA blocks with only 41 active threads during spike propagation. For the large network size here, the number of spikes per time step is of the same order as the maximal number of active CUDA blocks on the GPU (see **Table 1**). Partitioning the connectivity matrix under these conditions reduces the size and increases the number of the already small CUDA blocks without being able to execute them concurrently. Consequently, spike propagation times increase with partition number (**Figure 7C**, red). On the other hand, the synaptic effect application profits from partitioning the connectivity matrix. Without partitioning, only a single CUDA block applies all synaptic effects of the spike queue for the current time step. For large networks with large spike queues, partitioning distributes synapses across multiple CUDA blocks, significantly increasing effect application performance (**Figure 7C**, yellow). Note that partitioning also has a small impact on the memory usage: on an RTX2080 Ti, Brian2CUDA can simulate a LIF network with heterogeneous synapses with around $5.6 \cdot 10^5$ neurons when using a single partition, but with only about 0.77 times the size (around $4.3 \cdot 10^5$ neurons) when using 68 partitions.

For the STDP benchmark with heterogeneous delays, increasing the number of partitions decreases both, the spike propagation and the effect application times up to an optimal number of 64 partitions (**Figure 7D**). Without partitioning, the spike propagation is so inefficient that the total runtime exceeds that of the single-threaded C++ simulation (cf. **Figure 6B**). For

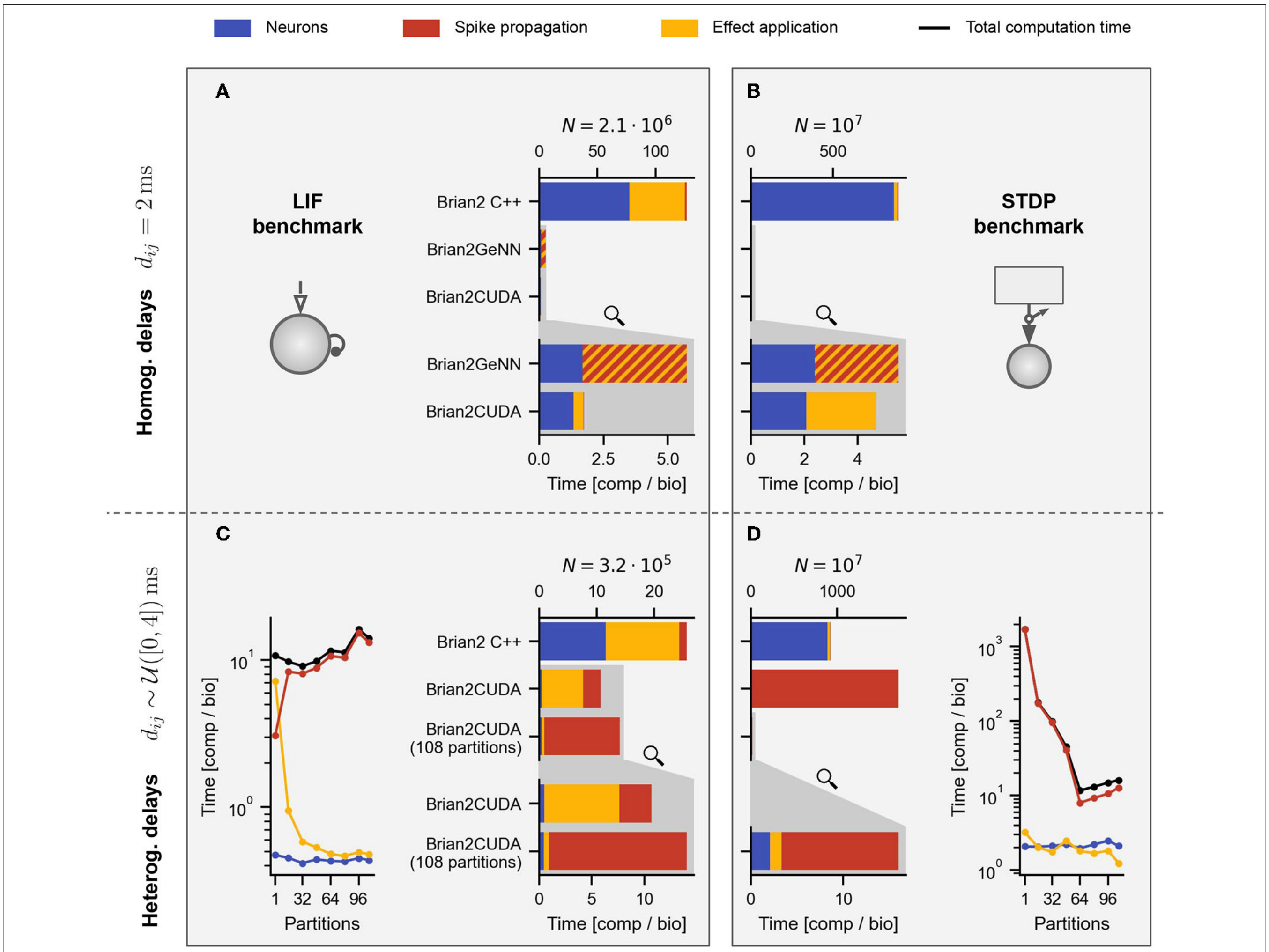


FIGURE 7 | Profiling results for benchmarks with homogeneous and heterogeneous delays. **(A,B)** Profiling results for LIF **(A)** and STDP **(B)** benchmarks with homogeneous delays for the respectively largest population of **Figures 5C,D**. **(C,D)** Profiling results for LIF **(C)** and STDP **(D)** benchmarks with heterogeneous delays for the largest population size of **Figures 6A,B**. For the STDP benchmarks in **(B,D)**, the Poisson spike generators are included in the neuronal computation times (blue). The gray shaded areas in the lower part of **(A–D)** contain zooms of the respective GPU simulations in the middle (indicated by the magnifying glass symbol). (**C**, left) and (**D**, right) show profiling results for different numbers of partitions of the connectivity matrix in Brian2CUDA. Black lines are the total computation times for the main time loop (cf. **Figure 6C**). In all panels, Brian C++ was simulated with 16 threads, Brian2CUDA was simulated without connectivity matrix partitioning if not stated otherwise and Brian2GeNN was simulated in *post* mode for **(A)** and in *pre* mode for **(B)**. For Brian2GeNN, only the combined time of spike propagation and effect application (striped bars) was recorded since both are combined into a single CUDA kernel. All simulations were performed as described in **Figure 5**, but with enabled profiling measurements leading to slightly higher total computation times.

this benchmark, every Poisson neuron has on average only 1 synapse. This is the worst-case scenario for Brian2CUDA's spike propagation algorithm, since all CUDA blocks have only a single thread and the hardware limit on maximally active blocks per SM strongly limits the number of synapse bundles that can be added to the spike queues concurrently. Additionally, the resulting small workload per SM leads to a low parallelization across active CUDA blocks since only one CUDA block can access all spike queues at any time. Increasing the number of partitions also partitions the spike queues. Since concurrent access of different CUDA blocks to different spike queues is possible, this increases spike propagation performance. This benchmark shows that for large neuronal populations with extremely sparse synapses (here only 1 synapse per neuron), Brian2CUDA's connectivity matrix partitioning can have drastic benefits on performance.

3.4. Runtime Contribution of Network Activity and State Variable Recordings

To analyze a spiking network model, Brian allows recording spike times, state variables and population firing rates. In Brian2CUDA, recorded variables are stored in GPU memory during a simulation and are transferred to CPU memory and written to disk at the end of a simulation. The contribution to overall computation time of such recordings strongly depends on

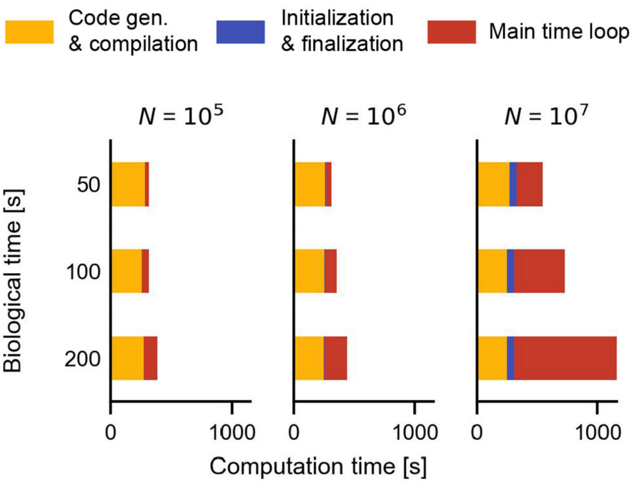


FIGURE 8 | Additional time required during a simulation with Brian2CUDA for the STDP benchmark with homogeneous delays (**Figure 5D**). Code generation and compilation times (yellow) are independent of network size and biological time. Network initialization and finalization (blue) depend on network size but not on biological time. Simulation of the main time loop (red) scales with both, biological time (linearly) and with network size. Compilation was performed in parallel on 16 CPU threads.

the details of a model (e.g., neuronal firing rates) and the number of recorded variables.

Consider for example **Figure 1C**, which shows the results for a simulation of the LIF benchmark with heterogeneous delays. To record this data, a spike recorder records all spikes in the network, a state variable recorder records the voltage of a single neuron for all time steps of the simulation and a population rate recorder records the fraction of spiking neurons at each time step. When adding these recorders to the largest LIF network with heterogeneous delays shown in **Figure 6A** ($N = 3.2 \cdot 10^5$), they require around 6 % of the computation time in the main simulation loop. Half of this additional time is spent on the spike recordings. For networks with overall less computation time per recorded unit, the contribution of recordings to total computation time naturally increases. For the extreme case of the HH benchmark without synapses (**Figure 5A**; recorded data shown in **Supplementary Figure S2**), the same recordings as above require around 40 % of the computation time for $N = 10^6$ neurons. Of this additional time around 2/3 is spent on spike recordings.

While Brian2CUDA stores recordings in GPU memory until the end of a simulation, Brian2GeNN copies them at each time step from GPU to CPU memory. Therefore, GPU memory can be a limiting factor for recordings in Brian2CUDA, whereas Brian2GeNN requires very little GPU memory. For the HH benchmark above, spike and population rate recordings in Brian2GeNN perform similarly to those in Brian2CUDA, while state variable recordings perform significantly worse. This is because of an inefficient implementation of the state variable recorder in Brian2GeNN, which copies at each time step all state variables of all neurons to the CPU, independent of the number of recorded neurons. This results in up to 2 orders of magnitude longer computation times when recording a single voltage trace in the HH benchmark example above.

Compared to Brian's C++ backends, absolute network recording runtimes in Brian2CUDA are comparable for large recordings (e.g., for the HH benchmark example), and can be slower for smaller recordings (around 5 times slower for LIF benchmark example). This is because memory copies in GPU memory are slow, and Brian2CUDA benefits more from parallelizing the copy process for larger recordings. Given the large speedups for other computations in Brian2CUDA, network activity and state variable recordings contribute relatively much more to total computation times than in Brian's C++ backends.

3.5. Additional Computation Time Factors: Code Generation, Compilation, Initialization, and Finalization

So far we have been analyzing the computation time needed for the main simulation loop, i.e., only that part of the simulation that is executed at every simulation time step. For long running simulations of large networks or for real-time applications, this is the most relevant performance measure. But in order to get from a Brian model script to the results, the Python code needs to be translated into the target language, which needs to be compiled and executed and finally, the results need to be transferred back into the Python environment. Furthermore, at the beginning of the simulation, the model needs to be initialized, which includes generating synapses, setting up connectivity matrices in the necessary format and for GPU backends, transferring data to GPU memory. **Figure 8** shows how compilation and network initialization contribute to the overall execution time for the STDP benchmark with homogeneous delays simulated with Brian2CUDA (cf. **Figure 5D**). The time spent in the simulation loop is proportional to the simulated biological time and also depends on the population size N of the network. The initialization time during the simulation is independent of the simulated biological time and increases with network size. And finally, the compilation time is independent of both, the simulated biological time and the network size.

Generally, for smaller networks (here $N \leq 10^6$) with shorter biological times (here $T < 200$ s), most of the computation time is spent on compilation, while this becomes negligible for larger networks simulating longer biological times. The compilation time for Brian's C++ backends Brian2GeNN and Brian2CUDA is mostly comparable, but can differ for some models. Brian's C++ backends and Brian2CUDA generate separate (CUDA C++) source files for each neuronal population object or synapse object. This can increase compilation times for networks with many objects, in particular for Brian2CUDA, which suffers from slower CUDA code compilation. The mushroom body benchmark for example requires almost twice the time for compilation as the STDP benchmark shown here because it consists of twice as many neuronal populations and synapse groups. While Brian2GeNN requires additional time for first generating GeNN code from the Brian model, it can compile the final CUDA code faster because

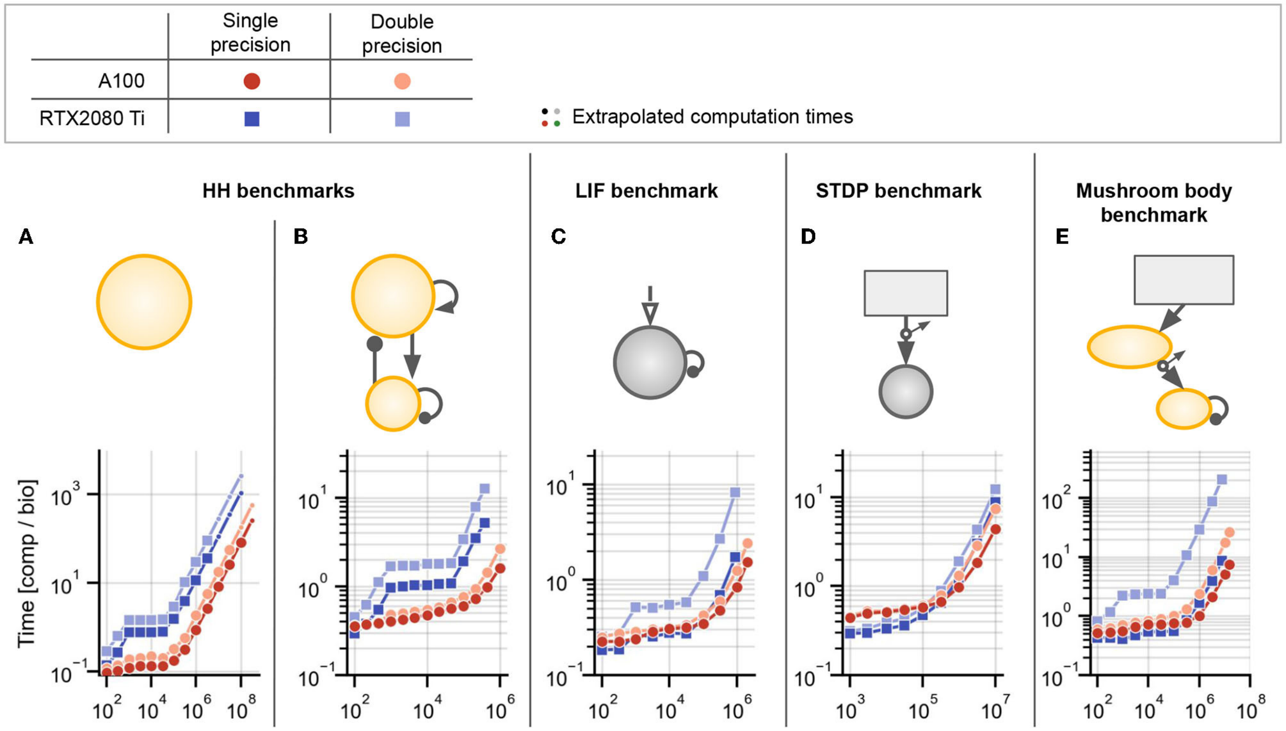


FIGURE 9 | Benchmark results for single- vs. double precision on consumer-grade vs. data-center GPUs. **(A–E)** Same benchmarks as shown in **Figure 5**. Simulated without connectivity matrix partitioning in Brian2CUDA, with single-precision floats (dark colors) and double-precision floats (bright colors) and on A100 data-center GPUs (red colors) and GeForce RTX2080 Ti consumer-grade GPUs (blue colors). Dark red lines show same data as in **Figure 5**.

it merges multiple computations into fewer CUDA kernels and source files.

3.6. Dependence on Floating Point Number Precision and GPU Hardware Choice

The results above stem from simulations using single-precision floating point arithmetics on A100 data center GPUs. Here we compare those results with Brian2CUDA's performance for simulations with double-precision floats on A100 GPUs and using a more affordable hardware GeForce RTX2080 Ti consumer-grade GPU (**Figure 9**). Consumer-grade GPUs are typically optimized for single-precision arithmetic operations and often have very low processing power on double-precision floats. The processing power of the RTX2080 Ti for double-precision floats is ∼32 times lower than for single-precision floats, while for the A100 it is only ∼2 times lower. The processing power of single-precision floats on the A100 is ∼1.66 times larger than on the RTX2080 Ti. However, the performance differences between GPUs and between single-precision and double-precision simulations don't necessarily reflect the difference in processing power. Additional factors play a role, such as hardware limits on memory per SM or available data transfer bandwidths. Specifically, the hardware limits can have double effect when comparing single- to double precision simulations. For the mushroom body benchmark, the speedup from double- to single-precision simulations on the RTX2080 Ti (**Figure 9E**, bright blue vs. dark blue) is much higher than in the other benchmarks. This is not only because of the increased processing power, but also because for double-precision simulation the extended memory requirements reach the hardware limits on available registers (see **Table 1**), forcing the simulation to run with less active threads. With single-precision floats, the reduced memory requirements allow higher GPU occupancy on top of the higher processing power for single-precision floats. Additionally, only computation bound simulations will show strong performance differences between floating point precisions and GPUs (e.g., in the HH benchmarks; **Figures 9A,B**). For simulations which are bound by communication tasks such as spike propagation, the performance differences are much lower (e.g., in the STDP benchmark; **Figure 9D**).

In summary, these results show that one does not need extremely expensive data-center GPUs to benefit from GPU computations in spiking neural networks, since much cheaper consumer-grade GPUs can perform comparably for many model types—at least for simulations with single-precision floats.

4. DISCUSSION

Building on the user-friendly simulator Brian and its code generation framework, the Brian2CUDA package presented here allows users with little technical expertise to simulate arbitrary neural and synaptic models on GPUs. As we have shown, this can lead to an important acceleration of a wide range of model simulations. The achievable speedup depends on the details of the model and the size of the network. For a small network, or a model with challenging features for parallelization such as heterogeneous transmission delays, only a several-fold increase in simulation speed might be possible. On the other hand, for models that are more favorable to parallelization, such as unconnected networks or networks with homogeneous delays and complex neuron models, the simulation speed can increase dramatically by several orders of magnitude. Our detailed benchmarking has shown a number of possible routes to further optimize the simulation speed for the challenging situations, which we will discuss in the following section.

4.1. Limitations and Future Work

Efficiently simulating different types of synaptic models on GPUs is challenging because there is no single algorithm that is best for all situations (Brette and Goodman, 2012; Kasap and van Opstal, 2018). Through partitioning the connectivity matrix, Brian2CUDA can counteract performance degradation for some cases where the default parallelization strategy would be inefficient. For models with homogeneous transmission delays and without partitioning the connectivity matrix, the effect application of individual spiking neurons is parallelized over CUDA blocks. Partitioning the connectivity matrix distributes the synapses for each spiking neuron over additional CUDA blocks. However, this increases performance only when the number of synapses per spiking neuron is larger than the maximal number of threads per CUDA block (1,024; cf. **Table 1**). For all benchmark models here, the average number of synapses per neuron is ≤1,000, for which partitioning does not increase parallelization. For models with more synapses, however, partitioning is expected to be beneficial as long as the number of spiking neurons per time step is small enough in order to keep the total number of CUDA blocks below the hardware limit on active blocks on all SMs of a GPU. For models with heterogeneous delays, partitioning the connectivity matrix has a non-trivial effect on spike propagation and synaptic effect application algorithms (see **Figure 7**). For example, without partitioning, spike propagation is very efficient while effect application is inefficient due to only one CUDA block applying all synaptic effects. Future work can further accelerate the simulation of models with heterogeneous delays by parallelizing the effect application over more CUDA blocks instead of using only one CUDA block per partition.

The current implementation of Brian2CUDA is optimized for large networks, where its speedups compared to Brian's C++ backends are the largest and where it outperforms Brian2GeNN in simulating the benchmark models employed here. For smaller network sizes, however, Brian2CUDA is often outperformed by Brian2GeNN (see for example **Figures 5B–D**). This can be at least partly explained by Brian's modular approach, inherited by Brian2CUDA. Each individual model component—e.g., the numerical integration, the thresholding, the resetting (cf. Section 2.3)—is contained in an individual kernel, and all kernels are executed sequentially. For kernels that don't utilize all resources (e.g., small populations of synapses/neurons), this leads to performance degradation. In contrast, Brian2GeNN merges all calculations related to neurons into one kernel and all updates of synapses in another kernel. We are currently working on two features that are promising to increase performance for smaller networks: (1) Using CUDA's concurrent kernel execution capabilities, kernels for separate neuron and synapse objects can be executed in parallel while keeping Brian's modular approach. (2) Convenience functions in Brian's Python interface can be implemented that allow users to easily merge multiple versions of the same (potentially small) model into a single large model. This would not only allow much easier parameter explorations of networks on a single GPU but also benefit from Brian2CUDA's optimizations for large networks.

Brian2CUDA's main focus is on optimizing the simulation phase, since this typically dominates the overall time for larger networks. To run smaller networks or simpler simulations, however, the long code generation and compilation phase in the beginning (cf. **Figure 8**), can be a major inconvenience. The long compilation times partly stem from Brian's modular approach mentioned above. Each component of the simulation is contained in a separate code file that needs to be compiled individually. To reduce compile times, multiple code files could be combined during code generation. It should be noted, however, that the reported compilation times are the full compilation times for a new simulation. If a user re-runs an existing simulation and only changes some aspects of it, only the changed source code will be re-compiled.

Another major future optimization for network simulations that run for a short biological time, is the synapse generation in the initialization phase of the simulation. At the moment, synapse generation uses Brian's C++ mechanism and therefore does not benefit from the GPU at all.

Current data structures and algorithms for simulating synapses are designed to handle all synaptic models and connection structures supported by Brian. But they perform better on some model types than on others. For example, for homogeneous delays, our synaptic effect application algorithm performs best when the number of connections is equally distributed across neurons. For structured connectivity, variable synapse group sizes can lead to unbalanced workloads across CUDA blocks during effect application (cf. **Figure 3E**), which can affect performance. Similarly, for heterogeneous delays, our synaptic effect application algorithm for synapse bundles performs best when bundle sizes are uniformly distributed within each synapse group because the same number of threads is assigned to each bundle (cf. **Figure 4D**). Strong variability across bundle sizes would lead to unbalanced workloads across groups of threads processing synapses of different bundles. In order to avoid these unbalanced workloads, one future direction could be to optimize our connectivity matrix scheme based on connectivity details. This could allow distributing workloads more evenly or exploiting local connectivity structures in our algorithms (as has been done before by Fidjeland et al., 2009; Fidjeland and Shanahan, 2010).

The presented work mostly focussed on optimizing simulation performance and less on memory usage. However, available memory can be a major constraint, in particular on consumer-grade GPUs. This is especially true when recording spike times and state variables from many neurons or synapses, which Brian2CUDA stores in GPU memory (see Section 3.4). In future versions, we plan a recorder implementation that allows transferring data in regular intervals from GPU to CPU memory, and we will focus on optimizing further unnecessary redundancies and memory inefficiencies. This should close the gap to Brian2GeNN, which is currently more memory-efficient.

Brian2CUDA is designed to support all features of Brian2 that are currently supported by its C++ backend and builds on the same code generation framework. It is therefore considerably less limited than the Brian2GeNN backend (discussed below), and supports a large variety of models. We have focused the development on spiking networks of single-compartment models, since they are most likely to benefit from GPU acceleration. Nevertheless, Brian2CUDA has support for other types of models supported by Brian, such as multi-compartment models, or rate-based models. This support is preliminary, though, and using Brian2CUDA might not give any performance benefits prior to improving the respective parallel algorithms.

As a general note on the limitations above, we would like to again emphasize that due to Brian2CUDA's implementation as a backend for the Brian simulator, a researcher does not need to invest any additional time or effort to port a model to Brian2CUDA. In contrast, porting a model to a simulator that only targets the GPU carries the risk that the effort is not worth the benefit. Due to the backend approach, researchers can also easily switch between the CPU and GPU-based approaches during development of a new model. For example, a researcher can do the initial development and testing on a small-scale model with the CPU, without having to pay the additional cost for the CUDA compilation, and then switch to the GPU for the final model, where the slower compilation is more than compensated by the faster computation time.

The Brian2CUDA backend is currently only supported for Linux operating systems (in contrast to Brian which supports Windows, Linux, and OS X), but this limitation will be removed in the future.

4.2. Comparison to Existing Approaches

Accelerating neural network simulations with the parallelization capabilities of GPUs has been a promising approach for more than a decade. The Brian2CUDA simulator presented here, builds on the foundations laid by these earlier simulators. For example, Brian2CUDA's spike propagation algorithm groups synapses based on their pre- and postsynaptic targets, as well as their delays into synapse *bundles*, similar to the approach of the NEMO simulator (Fidjeland et al., 2009; Fidjeland and Shanahan, 2010); in the case of homogeneous delays, Brian2CUDA's postsynaptic update algorithm results in a similar parallelization over synapses as in the dynamic parallelism approach described in Kasap and van Opstal (2018).

In recent years, several new, general-purpose simulators have seen the light of day, with each of them making different tradeoffs between the requirements of ease-of-use, flexibility and performance. To give a few recent examples: the Spike simulator (Ahmad et al., 2018) has been optimized for speed, but is implemented as a C++ library and therefore not easily useable for many researchers; the EDEN simulator (Panagiotou et al., 2021) runs arbitrary NeuroML v2 models (Cannon et al., 2014), which means it inherits NeuroML's focus on multi-compartmental models but also its limitations with regard to networks of spiking neurons; the NeuronGPU simulator (Golosio et al., 2021) comes with a convenient Python interface, but implementing new models requires editing the C++ source code of the simulator.

The Brian2CUDA interface and its general approach is comparable to the ANNarchy simulator (Vitay et al., 2015) and the GeNN simulator (Yavuz et al., 2016) when used together with its PyGeNN interface (Knight and Nowotny, 2021). By being a fully-featured backend for the Brian simulator, however, Brian2CUDA provides additional benefits for researchers that other simulators lack, such as a system of physical units, support for multi-compartmental models, and the possibility to precisely customize execution schedules. As we have shown in this article, Brian2CUDA not only provides flexibility and convenience, but also shows competitive performance for a wide range of network models.

The most similar approach to the Brian2CUDA package presented here is obviously the Brian2GeNN package, which is also implemented as a backend for the Brian simulator. Instead of generating CUDA code directly, the Brian2GeNN backend generates code for the GeNN simulator, which then in turn generates CUDA code. This approach has its advantages—e.g., Brian2GeNN will automatically benefit from performance optimizations in the GeNN package—but it also leads to a much more restricted set of Brian features that are supported. While the GeNN simulator provides a large amount of flexibility, it does not go as a far as Brian and Brian2CUDA, for example it does not allow for a customized execution order for all the elements of a simulation. The Brian2GeNN interface adds a number of additional restrictions. As a result, less common synapse implementations, in particular those that need access to and change variables both on the pre- and post-synaptic side, might not be supported. Brian2GeNN is also behind in enabling features added in newer versions of GeNN. Most importantly, GeNN added support for heterogeneous synaptic delays with its version 3.2, but this support is not yet available *via* the Brian2GeNN interface. The benchmark results for Brian2GeNN presented in this study should therefore be interpreted with caution and not necessarily be taken as indicative of GeNN's performance. For example, it appears as if the Brian2GeNN performance does not improve as much as expected when switching from double to single precision floats (**Supplementary Figure S1**), but that might be due to a suboptimal conversion of the model by Brian2GeNN.

5. CONCLUSION

By combining the flexibility of the Brian simulator with the simulation speed of GPUs, Brian2CUDA enables researchers to efficiently simulate spiking neural networks with minimal effort and thereby makes the advancements of GPU computing available to a larger audience of neuroscientists.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. The Brian2CUDA software package is publicly available on GitHub: https://github.com/brian-team/brian2cuda. The software version to reproduce the simulations in this study can be found at https://github.com/brian-team/brian2cuda/tree/paper2022 and instructions on how to run them can be found at https://github.com/brian-team/brian2cuda/tree/paper2022/brian2cuda/tools/benchmarking.

AUTHOR CONTRIBUTIONS

MA conceptualized and supervised the project. DA and MA designed the Brian2CUDA algorithms and designed the benchmarks. DA developed the Brian2CUDA software and performed and analyzed the benchmarks, and wrote the initial draft of the manuscript. MS supervised the integration with Brian and contributed necessary features to Brian itself. DA, MS, and MA revised the manuscript. All authors contributed to the article and approved the submitted version.

FUNDING

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) in the framework of collaborative research centers SFB910 and SFB1315 (project number 327654276), the Open Access Publication Fund of TU Berlin and by Programme Investissements d'Avenir IHU FOReSIGHT (ANR-18-IAHU-01).

ACKNOWLEDGMENTS

We thank Sudeshna Bora for assistance during the preparation of the first Brian2CUDA release, with benchmark simulations and analysis, as well as with manuscript formatting and visualization. We additionally thank Laura Naumann and Gregory Knoll for helping with visualizations and Konrad Wartke for an early implementation of some of the Brian2CUDA algorithms.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2022.883700/full#supplementary-material

REFERENCES

Abi Akar, N., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor- A morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Pavia, 274–282.

Ahmad, N., Isbister, J. B., Smithe, T. S. C., and Stringer, S. M. (2018). Spike: a GPU optimised spiking neural network simulator. *bioRxiv*, 461160. doi: 10.1101/461160

Alevi, D., Augustin, M., and Stimberg, M. (2022). Brian2CUDA (Version paper2022). *Zenodo*. doi: 10.5281/zenodo.6406656

Ben-Shalom, R., Ladd, A., Artherya, N. S., Cross, C., Kim, K. G., Sanghevi, H., et al. (2022). NeuroGPU: accelerating multi-compartment, biophysically detailed neuron simulations on GPUs. *J. Neurosci. Methods* 366, 109400. doi: 10.1016/j.jneumeth.2021.109400

Bernhard, F., and Keriven, R. (2006). "Spiking neurons on GPUs," in *Computational Science - ICCS 2006, Vol. 3994*, eds D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra (Berlin; Heidelberg: Springer Berlin Heidelberg), 236–243.

Bi, G.-,q. and Poo, M.-,m. (2001). Synaptic modification by correlated activity: Hebb's postulate revisited. *Annu. Rev. Neurosci.* 24, 139–166. doi: 10.1146/annurev.neuro.24.1.139

Blundell, I., Brette, R., Cleland, T. A., Close, T. G., Coca, D., Davison, A. P., et al. (2018). Code generation in computational neuroscience: a review of tools and techniques. *Front. Neuroinform.* 12, 68. doi: 10.3389/fninf.2018.00068

Brette, R., and Goodman, D. F. M. (2012). Simulating spiking neural networks on GPU. *Network* 23, 167–182. doi: 10.3109/0954898X.2012.730170

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398. doi: 10.1007/s10827-007-0038-6

Brunel, N., and Hakim, V. (1999). Fast global oscillations in networks of integrate-and-fire neurons with low firing rates. *Neural Comput*. 11, 1621–1671. doi: 10.1162/089976699300016179

Cannon, R. C., Gleeson, P., Crook, S., Ganapathy, G., Marin, B., Piasini, E., et al. (2014). LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Front Neuroinform*. 8, 79. doi: 10.3389/fninf.2014.00079

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge: Cambridge University Press.

Chou, T.-S., Kashyap, H. J., Xing, J., Listopad, S., Rounds, E. L., Beyeler, M., et al. (2018). "CARLsim 4: an open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters," in *2018 International Joint Conference on Neural Networks (IJCNN)* (Rio de Janeiro: IEEE), 1–8.

Eisenstat, S. C., Gursky, M. C., Schultz, M. H., and Sherman, A. H. (1982). Yale sparse matrix package I: The symmetric codes. *Int J Numer Methods Eng*. 18, 1145–1151. doi: 10.1002/nme.1620180804

Fidjeland, A. K., Roesch, E. B., Shanahan, M. P., and Luk, W. (2009). "NeMo: a platform for neural modelling of spiking neurons using GPUs," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors* (Boston, MA: IEEE), 137–144.

Fidjeland, A. K., and Shanahan, M. P. (2010). "Accelerated simulation of spiking neural networks using GPUs," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, Barcelona, 1–8.

Golosio, B., Tiddia, G., De Luca, C., Pastorelli, E., Simula, F., and Paolucci, P. S. (2021). Fast Simulations of highly-connected spiking cortical models using GPUs. *Front. Comput. Neurosci*. 15, 627620. doi: 10.3389/fncom.2021.627620

Goodman, D. F. M. (2010). Code generation: a strategy for neural network simulators. *Neuroinform* 8, 183–196. doi: 10.1007/s12021-010-9082-x

Hoang, R. V., Tanna, D., Jayet Bray, L. C., Dascalu, S. M., and Harris, F. C. (2013). A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. *Front. Neuroinform*. 7, 19. doi: 10.3389/fninf.2013.00019

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Trans. Neural Netw*. 14, 1569–1572. doi: 10.1109/TNN.2003.820440

Kasap, B., and van Opstal, A. J. (2018). Dynamic parallelism for synaptic updating in GPU-accelerated spiking neural network simulations. *Neurocomputing* 302, 55–65. doi: 10.1016/j.neucom.2018.04.007

Knight, J., Nowotny, T., Turner, J. P., Yavuz, E., Ali, F., Zhang, M., et al. (2021a). GeNN 4.5.1. (4.5.1) [Computer software]. *Zenodo*. doi: 10.5281/zenodo.5121623

Knight, J. C., Komissarov, A., and Nowotny, T. (2021b). PyGeNN: a python library for GPU-enhanced neural networks. *Front. Neuroinform*. 15, 10. doi: 10.3389/fninf.2021.659005

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Computat. Sci*. 1, 136–142. doi: 10.1038/s43588-020-00022-7

Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biol. Cybern*. 98, 459–478. doi: 10.1007/s00422-008-0233-1

Mutch, J., Knoblich, U., and Poggio, T. (2010). *CNS : a GPU-based framework for simulating cortically-organized networks CNS : a GPU-based framework for simulating cortically-organized networks*. Computer Science and Artificial Intelligence Laboratory Technical Report.

Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., and Veidenbaum, A. (2009). "Efficient simulation of large-scale spiking neural networks using CUDA graphics processors," in *2009 International Joint Conference on Neural Networks*, Atlanta, GA, 2145–2152.

Nowotny, T., Huerta, R., Abarbanel, H. D. I., and Rabinovich, M. I. (2005). Self-organization in the olfactory system: one shot odor recognition in insects. *Biol. Cybern*. 93, 436–446. doi: 10.1007/s00422-005-0019-7

NVIDIA Corporation (2007–2022). *CUDA^{TM}*, https://developer.nvidia.com/cuda-zone

Panagiotou, S., Sidiropoulos, H., Negrello, M., Soudris, D., and Strydis, C. (2021). EDEN: a high-performance, general-purpose, NeuroML-based neural simulator. *arXiv:2106.06752 [q-bio]*. *arXiv: 2106.06752*. doi: 10.48550/arXiv.2106.06752

Richert, M., Nageswaran, J. M., Dutt, N., and Krichmar, J. L. (2011). An efficient simulation environment for modeling large-scale cortical processing. *Front. Neuroinform*. 5, 19. doi: 10.3389/fninf.2011.00019

Song, S., Miller, K. D., and Abbott, L. F. (2000). Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nat. Neurosci*. 3, 919–926. doi: 10.1038/78829

Stimberg, M., Brette, R., and Goodman, D. F. (2019a). Brian 2, an intuitive and efficient neural simulator. *Elife* 8, e47314. doi: 10.7554/eLife.47314

Stimberg, M., Goodman, D. F. M., Benichoux, V., and Brette, R. (2014). Equation-oriented specification of neural models for simulations. *Front. Neuroinform*. 8, 6. doi: 10.3389/fninf.2014.00006

Stimberg, M., Goodman, D. F. M., Brette, R., and Brian contributors. (2020a). Brian 2 (2.4.2). *Zenodo*. 6226753.

Stimberg, M., Goodman, D. F. M., Brette, R., and Pitt,à, M. D. (2019b). "Modeling neuron-glia interactions with the brian 2 simulator," in *Computational Glioscience, Springer Series in Computational Neuroscience*, eds M. De Pittà and H. Berry (Cham: Springer International Publishing), 471–505.

Stimberg, M., Goodman, D. F. M., and Nowotny, T. (2020b). Brian2GeNN: accelerating spiking neural network simulations with graphics hardware. *Scientific Rep*. 10, 1–12. doi: 10.1038/s41598-019-54957-7

Stimberg, M., Nowotny, T., Goodman, D. F. M., and Brian2GeNN contributors. (2021). Brian2GeNN (1.6). *Zenodo*. doi: 10.5281/zenodo.1464116

Teska, A., Stimberg, M., and Brette, R. (2020). brian2modelfitting (0.4). *Zenodo*. doi: 10.5281/zenodo.4601961

Traub, R. D., and Miles, R. (1991). *Neuronal Networks of the Hippocampus*. Cambridge: Cambridge University Press.

Vitay, J., Dinkelbach, H., Ü., and Hamker, F. H. (2015). ANNarchy: a code generation approach to neural simulations on parallel hardware. *Front. Neuroinform*. 9, 19. doi: 10.3389/fninf.2015.00019

Yavuz, E., Turner, J., and Nowotny, T. (2016). GeNN: a code generation framework for accelerated brain simulations. *Sci. Rep*. 6, 18854. doi: 10.1038/srep18854

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

frontiers | Frontiers in Neuroinformatics

TYPE Original Research
PUBLISHED 10 February 2023
DOI 10.3389/fninf.2023.941696

Efficient parameter calibration and real-time simulation of large-scale spiking neural networks with GeNN and NEST

Felix Johannes Schmitt, Vahid Rostami and Martin Paul Nawrot*

Computational Systems Neuroscience, Institute of Zoology, University of Cologne, Cologne, Germany

Check for updates

OPEN ACCESS

EDITED BY
Thomas Nowotny,
University of Sussex, United Kingdom

REVIEWED BY
Marcel Stimberg,
Sorbonne Université, INSERM, CNRS, Institut de la Vision, France
Johanna Senk,
Julich Research Center (HZ), Germany

*CORRESPONDENCE
Martin Paul Nawrot
✉ martin.nawrot@uni-koeln.de

RECEIVED 23 May 2022
ACCEPTED 16 January 2023
PUBLISHED 10 February 2023

CITATION
Schmitt FJ, Rostami V and Nawrot MP (2023) Efficient parameter calibration and real-time simulation of large-scale spiking neural networks with GeNN and NEST.
Front. Neuroinform. 17:941696.
doi: 10.3389/fninf.2023.941696

Spiking neural networks (SNNs) represent the state-of-the-art approach to the biologically realistic modeling of nervous system function. The systematic calibration for multiple free model parameters is necessary to achieve robust network function and demands high computing power and large memory resources. Special requirements arise from closed-loop model simulation in virtual environments and from real-time simulation in robotic application. Here, we compare two complementary approaches to efficient large-scale and real-time SNN simulation. The widely used NEural Simulation Tool (NEST) parallelizes simulation across multiple CPU cores. The GPU-enhanced Neural Network (GeNN) simulator uses the highly parallel GPU-based architecture to gain simulation speed. We quantify fixed and variable simulation costs on single machines with different hardware configurations. As a benchmark model, we use a spiking cortical attractor network with a topology of densely connected excitatory and inhibitory neuron clusters with homogeneous or distributed synaptic time constants and in comparison to the random balanced network. We show that simulation time scales linearly with the simulated biological model time and, for large networks, approximately linearly with the model size as dominated by the number of synaptic connections. Additional fixed costs with GeNN are almost independent of model size, while fixed costs with NEST increase linearly with model size. We demonstrate how GeNN can be used for simulating networks with up to $3.5 \cdot 10^6$ neurons ($> 3 \cdot 10^{12}$ synapses) on a high-end GPU, and up to $250,000$ neurons ($25 \cdot 10^9$ synapses) on a low-cost GPU. Real-time simulation was achieved for networks with $100,000$ neurons. Network calibration and parameter grid search can be efficiently achieved using batch processing. We discuss the advantages and disadvantages of both approaches for different use cases.

KEYWORDS

computational neuroscience, attractor neural network, metastability, real-time simulation, computational neuroethology, spiking neural network (SNN)

Introduction

Information processing in animal nervous systems is highly efficient and robust. The vast majority of nerve cells in invertebrates and vertebrates are action potential generating (aka spiking) neurons. It is thus widely accepted that neural computation with action potentials in recurrent networks forms the basis for sensory processing, sensory-to-motor transformations, and higher brain function (Abeles, 1991; Singer and Gray, 1995). The availability of increasingly detailed anatomical, morphological, and physiological data allows for well-defined functional SNNs of increasing complexity that are able to generate testable experimental predictions at physiological and behavioral levels. SNNs have thus become a frequent tool in basic

(Van Vreeswijk and Sompolinsky, 1996; Brunel, 2000), translational (McIntyre and Hahn, 2010; Eliasmith et al., 2012), and clinical (Hammond et al., 2007; Kasabov and Capecci, 2015) neuroscience research. In the applied sciences, brain-inspired SNNs have the potential to shape future solutions for intelligent systems (Neftci et al., 2013; Chicca et al., 2014; Schuman et al., 2022). This specifically includes spike-based approaches to machine learning (Gütig and Sompolinsky, 2006; Indiveri et al., 2010; Schmuker et al., 2014; Gütig, 2016; Pfeiffer and Pfeil, 2018; Zenke and Ganguli, 2018; Tavanaei et al., 2019; Rapp et al., 2020), reservoir computing (Büsing et al., 2010; Tanaka et al., 2019), and brain-inspired control architectures for artificial agents and robots (Helgadóttir et al., 2013; Rapp and Nawrot, 2020; Sakagiannis et al., 2021; Bartolozzi et al., 2022; Feldotto et al., 2022). Computation with attractor networks has been hypothesized as one hallmark of brain-inspired computation (Hopfield, 1982; Amit and Brunel, 1997) and, with increasing evidence, has been implicated in decision-making (Finkelstein et al., 2021), working memory (Sakai and Miyashita, 1991; Inagaki et al., 2019), and sensory-motor transformation (Mazzucato et al., 2019; Wyrick and Mazzucato, 2021; Mazzucato, 2022; Rostami et al., 2022).

The conventional tool for the simulation of SNNs are CPU-based simulation environments. Several well-adopted simulators are in community use (Brette et al., 2007; Tikidji-Hamburyan et al., 2017), each of which has typically been optimized for specific purposes such as the simulation of complex neuron models with extended geometry and detailed biophysics (Hines and Carnevale, 2001), the convenient implementation of neuron dynamics by means of coupled differential equations (Stimberg et al., 2019), or the implementation of the Neural Engineering Framework (NEF) (Eliasmith and Anderson, 2003; Bekolay et al., 2014). The NEural Simulation Tool (NEST, https://www.nest-simulator.org/, Gewaltig and Diesmann, 2007) that we consider here was designed for the parallelized simulation of large and densely connected recurrent networks of point neurons. It has been under continuous development since its invention under the name of SYNOD (Diesmann et al., 1995, 1999; Rotter and Diesmann, 1999; Morrison et al., 2005, 2007; Jordan et al., 2018) and enjoys a stable developer and a large user community. More recently, new initiatives have formed to harness GPU-based simulation speed for the modeling of SNNs (Fidjeland et al., 2009; Nageswaran et al., 2009; Mutch et al., 2010; Brette and Goodman, 2012; Florimbi et al., 2021; Golosio et al., 2021; Ben-Shalom et al., 2022). The GPU-enhanced Neural Network (GeNN) simulation environment (https://genn-team.github.io/genn/) developed by Thomas Nowotny and colleagues (Yavuz et al., 2016; Knight and Nowotny, 2021; Knight et al., 2021) is a code generation framework (Blundell et al., 2018) for SNNs and their use in computational neuroscience and for machine learning (Knight and Nowotny, 2022). Neuromorphic hardware (Ivanov et al., 2022; Javanshir et al., 2022) provides an alternative substrate for the simulation of SNNs and is not considered here.

The present study aims to evaluate the use of a GPU-based simulation technique (GeNN) in comparison with a CPU-based simulation technique (NEST) with respect to simulation speed independent of network size and in the context of efficient parameter search. We restricted our benchmark approach to simulations on single machines with multiple CPU cores. These machines can be considered standard equipment in a computational lab. In addition we compare simulation performance on a high-end GPU with an affordable low-cost GPU that can be used, e.g., for teaching purposes. Based on our experience, we provide practical advice in the Supplementary material along with documented code.

Results

Spiking neural attractor network as benchmark model

We performed simulations of the spiking cortical attractor network model established by Rostami et al. (2022). This network inherits the overall network structure of the random balanced network (RBN, Van Vreeswijk and Sompolinsky 1996; Brunel 2000) with random recurrent connections (drawn from the Bernoulli distribution) among excitatory and inhibitory neurons (Figure 1A) but introduces a topology of strongly interconnected pairs of excitatory and inhibitory neuron populations (E/I clusters, Figure 1B) by increasing the intra-cluster synaptic weights (see Section Materials and methods). This E/I clustered network exhibits a complex pattern of spontaneous network activity, where each cluster can dynamically switch between a state of low (baseline) activity and states of increased activity (Figure 1). This network behavior marks the desired feature of metastability (Rost et al., 2018; Mazzucato et al., 2019; Rostami et al., 2022) where the network as a whole cycles through different attractors (or network-wide states) that are defined by the possible cluster activation patterns.

The pairwise Bernoulli connectivity scheme with a connection probability p between any pair of neurons implies that the number of synapses M scales quadratically with the number of neurons N as $M = pN^2$. For the chosen network parameters, we obtain an overall connectivity parameter of $p \approx 0.3$ (see Section Materials and methods). The clustered network topology in our benchmark model results from stronger synaptic excitatory and inhibitory weights within each E/I cluster than between different E/I clusters (Figure 1B). This compartmentalized architecture suits well our benchmarking purpose because it is reminiscent for whole-system or multi-area modeling in large-scale models that involve several neuropiles or brain areas (Schmidt et al., 2018; Rapp and Nawrot, 2020). We kept the number of clusters fixed to $N_Q = 20$.

Benchmark approach and quantification of simulation costs

We benchmark performance by measuring the wall-clock time of the simulation. We differentiate fixed costs T_{fix} that are independent of the biological model time to be simulated, and variable costs T_{var} determined by the simulation speed after model generation (see Section Materials and methods). We used two different hardware configurations for CPU-based simulation with NEST (servers S2 and S3 in Table 1) and two hardware configurations for GPU-based simulation with GeNN (Table 1) comparing a low-cost GPU (S1) with a state-of-the-art high-end GPU (S3). With GeNN, we tested two different approaches to store the connectivity matrix of the model (Knight and Nowotny, 2021). The SPARSE connectivity format (Sp) stores the matrix in a sparse representation. The PROCEDURAL

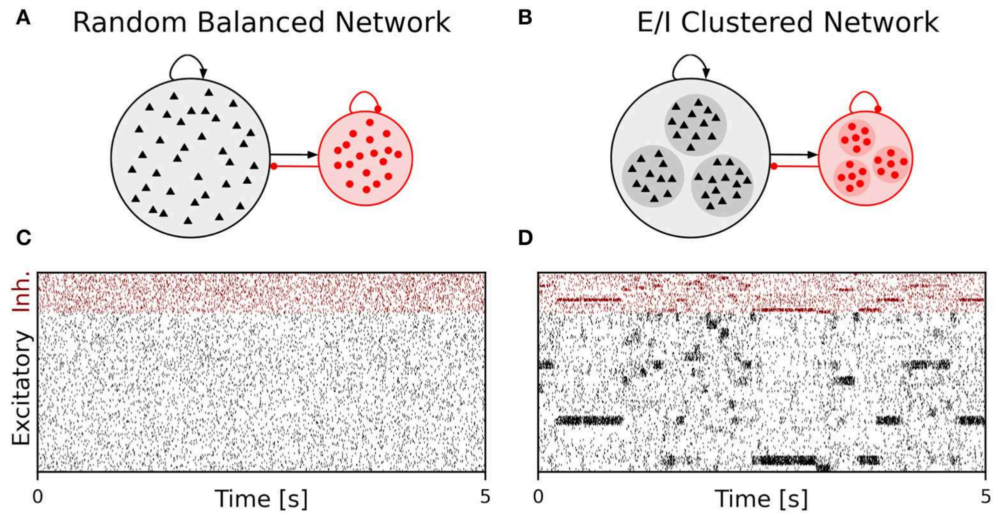


FIGURE 1
Metastable network activity emerges by introducing excitatory-inhibitory clusters in the random balanced network. **(A)** Sketch of RBN architecture with one excitatory neuron pool (gray shaded circle, 80% of all neurons) and one inhibitory neuron pool (red shaded circle, 20% of all neurons). Excitatory neurons (black triangles) and inhibitory neurons (red circles) make random connections within and across pools, respectively. The respective connection strengths are tuned such that for each neuron, on average, the total synaptic input current balances excitatory and inhibitory input currents. **(B)** Sketch of excitatory-inhibitory (E/I) cluster topology. Both, excitatory and inhibitory neuron pools are tiled into clusters (small shaded circles) of strongly interconnected neurons (indicated by darker shading). In addition, each excitatory cluster is strongly and reciprocally connected to one corresponding inhibitory cluster such that the balance of excitatory and inhibitory synaptic input is retained for all neurons. In our network definition, a single parameter J_{E+} determines the cluster strength in terms of synaptic weights and allows to move from the RBN ($J_{E+} = 1$) to increasingly strong clusters by increasing $J_{E+} > 1$. The model uses exponential leaky Integrate-and-Fire (I&F) neurons, and all neurons receive weak constant input current. **(C, D)** Raster plot of excitatory (black) and inhibitory (red) spiking activity in a network of $N = 25,000$ neurons during 5 s of spontaneous activity after an initial warm-up time of 1 s was discarded. Shown are 8% of the total neuron population. The spike raster plots are generated from GeNN simulations. **(C)** The RBN ($J_{E+} = 1.0$, $I_{thE} = 2.6$, $I_{thI} = 1.9$) exhibits irregular spiking of excitatory and inhibitory neurons with constant firing rates that are similar for all excitatory and inhibitory neurons, respectively. **(D)** The E/I clustered network ($J_{E+} = 2.75$, $I_{thE} = 1.6$, $I_{thI} = 0.9$) shows a metastable behavior, where different individual E/I clusters can spontaneously assume states of high activity and fall back to spontaneous activity levels. The overall firing rates are higher than in the RBN.

TABLE 1 Hardware configurations and benchmark setups.

CPU			GPU			
No. of cores	Clock speed [GHz]	Memory [GB]	Architecture	No. of CUDA-cores	Memory [GB]	Performance (single) [TFLOPS]
Server 1 (S1) Ubuntu 16.04 LTS						
Dual AMD Opteron 6380			GeForce GTX 970			
2 × 16	2.5	128	Maxwell	1,664	4	3.5
Server 2 (S2) Ubuntu 16.04 LTS						
Dual Intel Xeon E5-2630 v4			-			
2 × 10	2.2	192	-	-	-	-
Server 3 (S3) Ubuntu 20.04 LTS						
Intel Xeon Gold 6248R			Quadro RTX A6000			
24	3.0	128	Ampere	10,752	48	38.7

connectivity (Pr) regenerates the connectivity on demand, i.e., after a spike has occurred.

Fixed costs for GeNN are high but independent of network size

We find that for NEST, the overall fixed costs scale approximately linearly with the network connectivity as expressed in the total number of connections $M \propto N^2$ (Figure 2), while the overall fixed costs stay approximately constant for GeNN and essentially over the complete range of tested network sizes. The fixed costs add up different contributions as shown in Figure 2A and in Supplementary Figure S5. These are model definition, building of the model, and loading of the model for GeNN and node creation and creation of connections for NEST (see Section Materials and methods). For NEST, compilation of the model (Build phase of GeNN) is not needed because it uses pre-compiled neuron and synapse models (Diesmann and Gewaltig, 2002) in combination

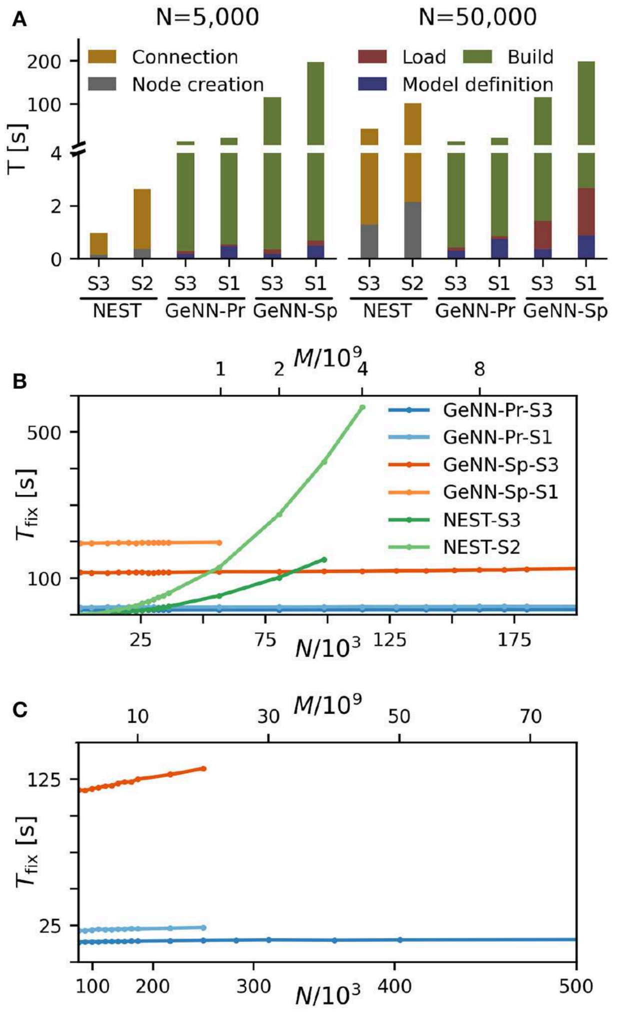


FIGURE 2
Fixed costs of simulation. **(A)** Individual costs for execution phases of NEST and GeNN for two network sizes of $N = 5,000$ and $N = 50,000$ neurons. The GPU-based simulation requires expensive model compilation (Build). NEST uses pre-compiled neuron models. Note that the y-axis has different linear scales for low (≤ 4) and high (≥ 4) values of fixed costs. **(B, C)** Total fixed costs in seconds over network size for the different simulators and hardware configurations as indicated. Total fixed costs are approximately constant across network size for the GPU-based simulation with the PROCEDURAL connectivity, while they have a small slope for the SPARSE connectivity. **(C)** Extends **(B)** for larger network sizes. Note that the x-axis in **(B)** is linear in N, while the x-axis in **(C)** is linear in M.

with exact integration (Rotter and Diesmann, 1999). The fixed costs of GeNN are dominated by the wall-clock time required for building the model and these appear to be essentially independent of model size. The costs of model definition and loading the model increases with model size, but make only a negligible contribution to the overall fixed costs. Thus, for a small network size of $N = 5,000$ neurons, the overall fixed costs amount to ≈ 30 s and ≈ 3 min for the PROCEDURAL and SPARSE connectivity, respectively, compared to only 3 s with NEST. This picture changes for a 10 times larger network with $N = 50,000$ neurons. Now, the wall-clock time for setting up the model with NEST is more costly than building with GeNN (PROCEDURAL connectivity) on our hardware configurations. The fixed costs for NEST increase quadratically in N (linear in M) on both hardware configurations and eventually exceed fixed costs with GeNN (Figure 2B).

TABLE 2 Maximum network size N_{RT} within real-time limit.

E/I-model	N_{RT} [s]	$M_{\mathrm{RT}}/10^6$
GeNN-Pr-S3	20,500	129
GeNN-Pr-S1	6,400	13
GeNN-Sp-S3	102,000	3,204
GeNN-Sp-S1	26,900	223
NEST-S3	15,000	69
NEST-S2	7,900	19
RBN	N_{RT} [s]	$M_{\mathrm{RT}}/10^6$
GeNN-Pr-S3	24,300	182
GeNN-Sp-S3	160,000	7,885
NEST-S3	27,500	233
E/I-model, $\tau_{\mathrm{syn}} \in \mathcal{U}$	N_{RT} [s]	$M_{\mathrm{RT}}/10^6$
GeNN-Pr-S3	17,200	91
GeNN-Sp-S3	47,400	692
NEST-S3	15,100	70

The real-time limit for the E/I-model was determined in simulation steps of 500 neurons. The real-time limits of the RBN and E/I-model with heterogeneous synaptic time constants were estimated with a linear interpolation between the nearest data points (cf. markers in Figure 4C). Simulations comprised 10 s of biological model time with $\Delta t = 0.1$ ms.

The maximal network size that we were able to simulate is indicated by the end points in the benchmark graphs in Figures 2B, C. It is bound by the available memory, which is required for storing the network model and the data recorded during simulation. In the case of simulation with NEST, this bound is determined by the RAM configuration (Table 2). The larger RAM size of 192 GB on S2 allowed for a maximum network connectivity of $M = 4 \cdot 10^9$ synapses and $N \approx 114,000$ neurons, while on the faster server configuration S3 with 128 GB RAM, the limit was reached earlier (Figure 2B). With GeNN, the limiting factor for the network size and connectivity is the hardware memory on the GPU itself. The PROCEDURAL connectivity allows for a more efficient usage of the GPU memory (Supplementary Figure S3) at the expense of simulation speed and allowed for a network size of $> 3.5 \cdot 10^6$ neurons and $> 3,000 \cdot 10^9$ synapses on the high-end GPU (S3) and a respectable size of $N \approx 250,000$ neurons ($M \approx 20 \cdot 10^9$ synapses) on the low-cost GPU (S1).

Variable costs scale linearly with biological model time and approximately linearly with network connectivity

We first quantify wall-clock time T_{var} in dependence on the simulated biological model time T_{bio} for a fixed network size of $N = 50,000$ (Figure 3A). As to be expected, simulation time grows approximately linearly with the number of simulated time steps and thus, for a pre-defined simulation

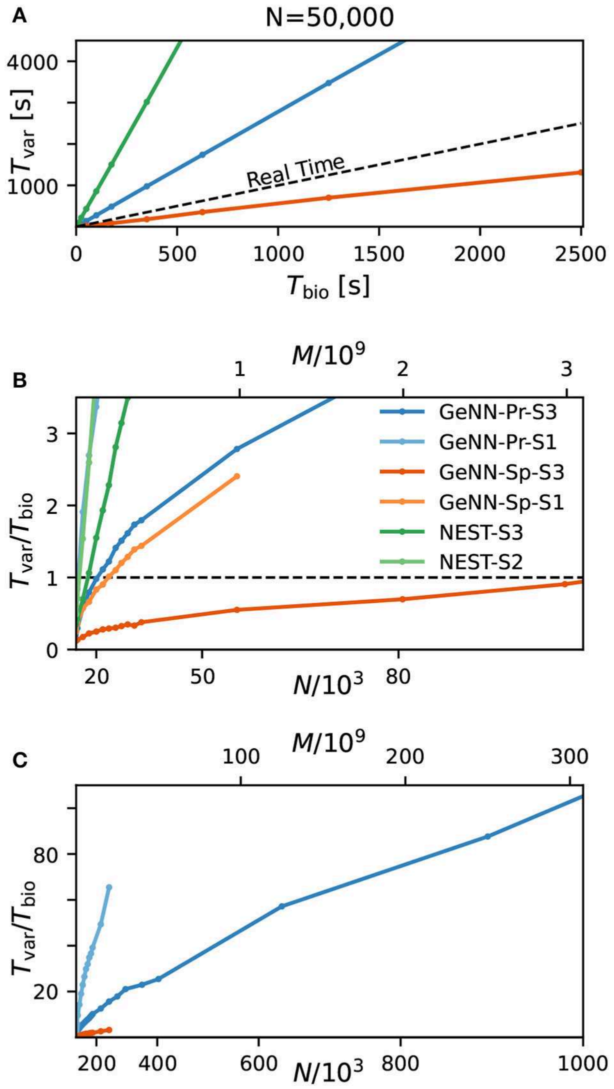


FIGURE 3
Variable costs of simulation and real-time limitation. **(A)** Wall-clock time T_{var} as a function of simulated biological model time T_{bio}. **(B)** Real-time factor $RT = T_{\mathrm{var}}/T_{\mathrm{bio}}$ as a function of network size. Real-time capability requires a variable cost factor that remains below $RT = 1$ (dashed line). On the high-end GPU, simulation faster than real-time can be achieved for a network size of $N \approx 100,000$ neurons and $M \approx 3.1 \cdot 10^9$ synapses. With the best performing NEST configuration, a network of $N \approx 15,000$ neurons and $M \approx 0.1 \cdot 10^9$ synapses can be simulated in real-time. GeNN-Pr-S1 is congruent with NEST-S2 in this view. **(C)** As in **(B)** for larger network size. Wall-clock time scales almost linearly with the total number of synapses M (and thus with N^2) for large network sizes. All simulations have been conducted with a time resolution of $\Delta t = 0.1$ ms in biological model time. **(B, C)** The x-axis is linear in M (top axis).

time constant $1/\Delta t$, is proportional to the simulated time with

$$T_{\mathrm{var}} \propto \frac{1}{\Delta t} \cdot T_{\mathrm{bio}}$$

for all tested hardware platforms. We see considerably faster simulations with the SPARSE connectivity compared to the PROCEDURAL connectivity in our network with a total connectivity of $p \approx 0.308$. Interestingly, a previous publication by Knight and Nowotny (2021) considering the RBN with a lower connectivity density of $p = 0.1$ found that PROCEDURAL connectivity performs equally fast or even faster, which could be due to their lower connectivity density.

Next, we analyzed the relation between wall-clock time and network size. As shown in Figures 3B, C, the proportionality factor $T_{\mathrm{var}}/T_{\mathrm{bio}}$ for NEST shows an approximate linear dependence on the total number of synapses M for large network sizes and the variable costs are thus proportional to the squared number of neurons $T_{\mathrm{var}} \propto pN^2 \cdot T_{\mathrm{bio}}$ with linear scale factor p, denoting network-specific connectivity.

For GeNN, the graph shows a convex relation between $T_{\mathrm{var}}/T_{\mathrm{bio}}$ and lower range network sizes. For increasing network size, this relation becomes increasingly linear in N^2. Additional model-related factors may contribute, in particular, the average spiking activity of neurons and the resulting spike traffic (see Section Discussion).

Real-time simulation is defined as $T_{\mathrm{sim}} = T_{\mathrm{bio}}$. For applications in neurorobotics or when using SNNs for real-world machine learning application, we may require the simulation to run equally fast or faster than real time. For our attractor network model, we determined a maximum network size of $N_{\mathrm{RT}} = 102,000$ that fulfills the real-time requirement using GeNN on the high-end GPU (Figure 3B and Table 2).

The RBN is a standard model in computational neuroscience and is used widely for the simulation of cortical activity. We therefore repeated our calibrations for the RBN in direct comparison to the E/I clustered network using the fastest hardware configuration (S3). As shown in the Supplementary Figure S1, the fixed costs are identical for both network types. This was to be expected as the overall network connectivity is identical in both cases. The variable costs show the same general dependence on network size (Supplementary Figure S1C) but are smaller for the RBN mainly due to the overall lower firing rates (Figure 1).

Simulation costs with heterogeneous synaptic time constants

Thus far, all neuron and synapse parameters were identical across the network with fixed synaptic weight and time constant for excitatory and inhibitory synapses, respectively. We now introduce heterogeneity of the excitatory and inhibitory synaptic time constants using uniform distributions with the means corresponding with the parameter values used earlier and with a standard deviation of ±5%. Using the same aforementioned neuron model in NEST, the heterogeneity applies across postsynaptic neurons, while for each neuron, all incoming synapses have identical time constants for excitatory and inhibitory synapses, respectively (see Section Materials and methods). In GeNN, we defined the neuron model and synapse model independently and synaptic time constants are heterogeneously distributed across all excitatory and inhibitory synapses individually, independent of postsynaptic neuron identity (see Section Discussion).

As a first result, we observe that the E/I clustered network retains the desired metastable network dynamics with distributed synaptic time constants as shown in Figure 4A. When comparing the simulation costs to the homogeneous case on the fastest hardware

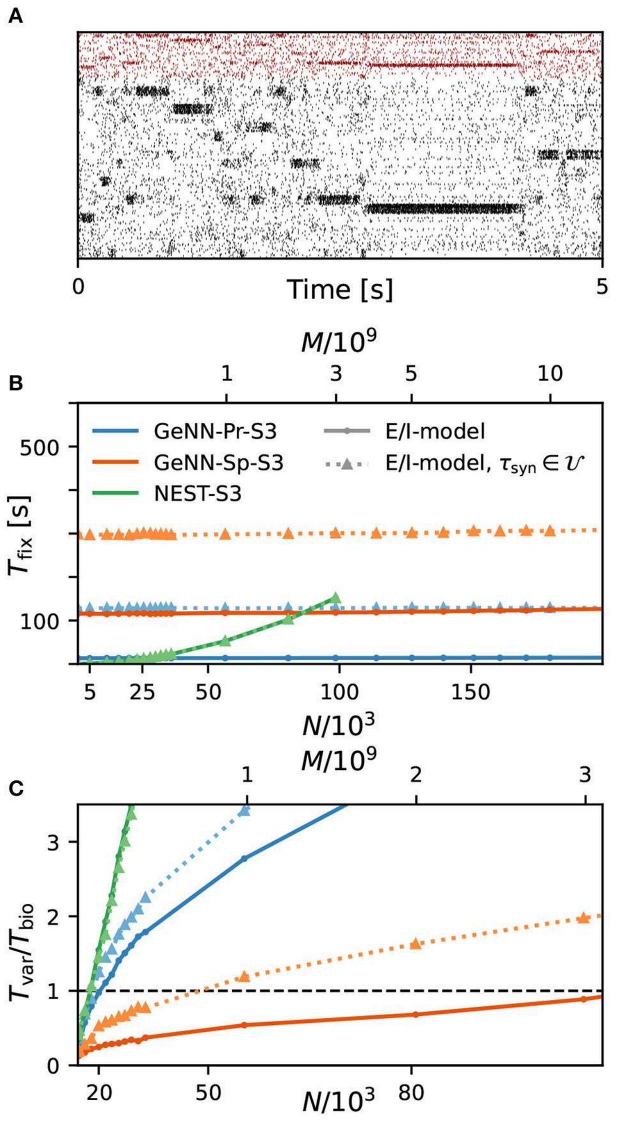


FIGURE 4
Metastability and simulation costs for networks with heterogeneous synaptic time constants. **(A)** Raster plot of excitatory (black) and inhibitory (red) spiking activity in a network of $N = 25,000$ neurons with heterogeneous synaptic time constants during 5 s of spontaneous activity shows the desired metastable behavior where different individual clusters can spontaneously assume states of high and low activities. A portion of 8% of neurons from each of the 20 clusters is shown. Network parameters are identical to Figure 1D: $J_{E+} = 2.75$, $I_{\text{thE}} = 1.6$, and $I_{\text{thI}} = 0.9$. The spike raster plot is generated from a GeNN simulation. **(B)** Total fixed costs in dependence of network size for the E/I model with (dotted lines) and without (solid lines) distributed synaptic time constants when simulated with NEST (green) or GeNN. For NEST, the fixed costs are indistinguishable for both model variants. For GeNN, the fixed costs are independent of network size but considerably higher with distributed synaptic time constants. **(C)** Real-time factor in dependence of network size. For NEST, the variable costs are the same in the case of homogeneous and heterogeneous synaptic time constants, while GeNN has increased variable costs in the heterogeneous case. Note that the x-axis in **(B)** is linear in N, while the x-axis in **(C)** is linear in M.

TABLE 3 Fixed costs (T_{fix}) and variable costs (T_{var}) for simulating networks of size $N = 50.000$ during 10 s of biological model time.

E/I-model	T_{fix} [s]	T_{var} [s]
GeNN-Pr-S3	13.6	24.9
GeNN-Pr-S1	21.6	80.3
GeNN-Sp-S3	117.3	5.2
GeNN-Sp-S1	202.8	19.8
NEST-S3	41.9	80.8
NEST-S2	104.7	280.3
RBN	T_{fix} [s]	T_{var} [s]
GeNN-Pr-S3	13.4	16.0
GeNN-Sp-S3	116.6	3.3
NEST-S3	41.7	26.4
E/I-model, $\tau_{\text{syn}} \in \mathcal{U}$	T_{fix} [s]	T_{var} [s]
GeNN-Pr-S3	128.2	30.2
GeNN-Sp-S3	298.9	10.3
NEST-S3	41.9	78.5

Top: Standard benchmark model with clustered connectivity ($J_{E+} = 10$) and homogeneous synaptic time constants (E/I-model). Middle: Random balanced network (RBN) without clustering ($J_{E+} = 1$) and homogeneous synaptic time constants. Bottom: Clustered network model with ($J_{E+} = 10$) and with heterogeneous synaptic time constants ($\tau_{\text{syn}} \in \mathcal{U}$).

configurations (S3), we find that fixed costs did not increase with NEST and show an indistinguishable dependence on network size (Figure 4B). However, with GeNN, fixed costs remain independent of network size but were strongly increased in comparison to the homogeneous case, for both the SPARSE and the PROCEDURAL approaches.

For the variable costs, there is again no increase with NEST with the same linear dependence on network size as in the homogeneous case (Figure 4C). This was to be expected, as NEST, by default, stores one propagator for the excitatory and one for the inhibitory input per neuron. Thus, the per neuron integration of the postsynaptic current is performed identically to the homogeneous case. In GeNN, on the other hand, we use independent synapse models where each individual synapse has a different time constant. This requires to perform the integration over time independently. Hence, we observe a considerable increase in variable costs that follows the same convex dependency on network size as in the homogeneous case (Figure 4C). For the duration of 10 s of biological model time used here and for a network size of 50.000 neurons, the total costs with GeNN are higher than that of NEST (Table 3, see Section Discussion).

Efficient approach to parameter grid search

Achieving robust model performance requires the vital and computationally demanding step of model calibration with respect to independent model parameters (see Section Discussion). Generally, the total costs for a parameter optimization directly scale with the number of samples tested for the considered parameter combinations. In our spiking attractor benchmark model, we have 22 independent parameters (cf. Tables 6, 7). To this end, we perform a 2D grid search investigating the average firing rate across the entire population of excitatory neurons in dependence on two parameters: the constant background stimulation currents $I_{x\text{E}}$ and $I_{x\text{I}}$ measured in multiples of the rheobase current $I_{x\text{X}} = I_{\text{thX}} \cdot I_{\text{rheoX}}$ (Figure 5).

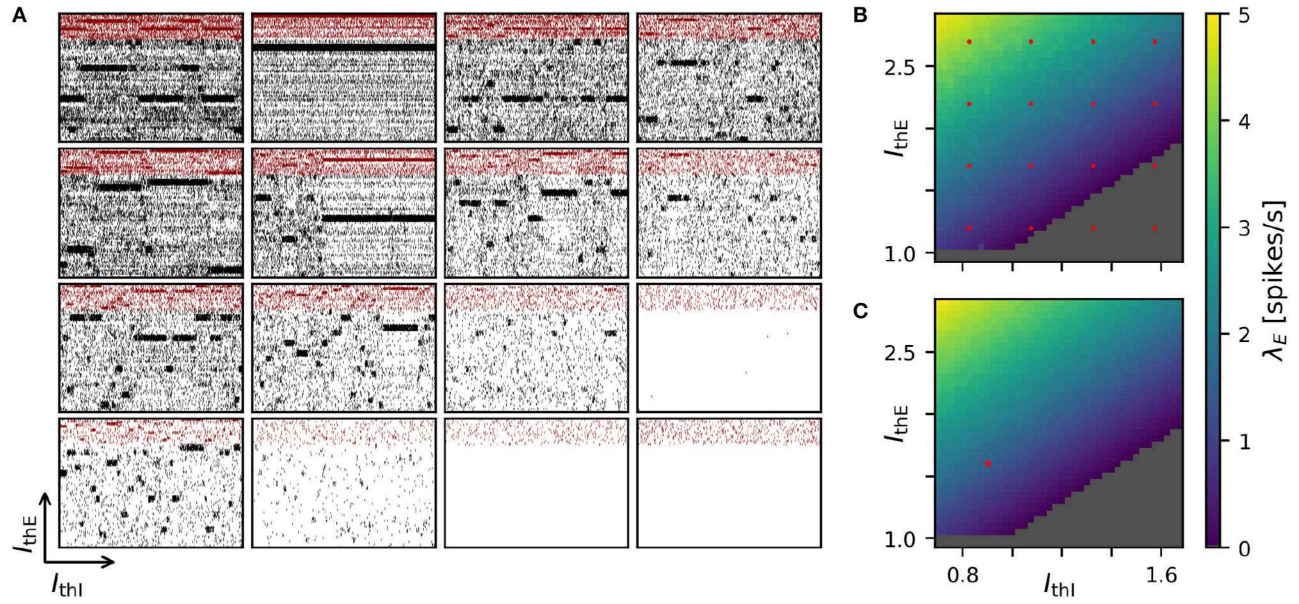


FIGURE 5
Two-dimensional parameter grid search with GeNN and NEST. **(A)** Shown are a subset of 4 × 4 spike raster plots as generated from simulations with NEST. The respective parameter values are indicated by the red point markers in **(B)**. In each raster, 8% of neurons from each of the 20 clusters are displayed. The network size was $N = 25,000$ and the clustering parameter was $J_{\mathrm{E}+} = 2.75$ as in Figures 1D, 4. **(B)** Average spontaneous firing rates of all excitatory neurons in dependence of I_{thE} and I_{thI} across the complete grid of 40 × 40 parameter pairs when simulated with NEST. The run-time for completing the grid search was 29,692 s (≈8 h: 15 min). **(C)** Same as in **(B)** for the simulation with GeNN in batch mode (batch size 40). The red point marker indicates the parameter combination used for the spike raster plot in Figure 1D. The required run-time was 2,809 s (≈47 min). The dependence of wall-clock time on the average firing rate is analyzed for NEST and GeNN in Supplementary Figure S2.

We sampled a grid of 40 × 40 parameter values, and for each sample point, our benchmark model is simulated for 10 s of biological model time. This results in a total simulated biological model time of 16,000 s for a network size of $N = 25,000$ neurons and $M = 193 \cdot 10^6$ connections. We obtain plausible spike raster plots for different combinations of the background stimulation of excitatory and inhibitory neuron populations, and metastability emerges in a large parameter regime (Figure 5A). The activity of the excitatory populations increases along the I_{thE}-axis and decreases with stronger stimulation of the inhibitory neurons. The comparison of average firing rates across the excitatory population in simulations with NEST (Figure 5B) and GeNN (Figure 5C) shows only negligible differences due to the random network structure.

We first compared this grid search for simulation with NEST on different servers and with different parallelization schemes (Table 4). For each parameter combination, a new network instance was generated ensuring independent samples. The fastest grid search is achieved using server S3 (single CPU socket) with one worker that uses all available cores as threads. This parallelization scheme reduces the run-time by 40% in comparison to a scheme with six simulations run in parallel with four threads each. The observed advantage of using a single worker is in line with the results of Kurth et al. (2022). The results are different on S2 with its two CPU sockets where five parallel simulations, each with four threads, resulted in a slightly improved performance compared to the case of a single simulation on all cores. Performing the same grid search with GeNN using independent network instances for each parameter combination required a total of 4.500 s and was thus 6.6 times faster than the independent grid search with NEST.

TABLE 4 Run-time of 40 × 40 grid search with GeNN and NEST with different hardware configurations and parallelization schemes.

GeNN		
Server	Batch size	Run-time [s]
S3	40	2, 809
S3	1	4, 500
S1	4	13, 080
S1	1	14, 491
NEST		
Server	n_W / n_T	Run-time [s]
S3*	1 / 24	18, 028
S3	1 / 24	29, 692
S3	6 / 4	49, 403
S2	1 / 20	83, 848
S2	5 / 4	81, 631

n_w denotes the number of workers and n_T the number of threads used for CPU simulation on multiple cores. S3* denotes the NEST simulation approach where a single network instance is set up once and reused for all samples in the grid.

To maximize GPU utilization and to save fixed costs, we here propose an alternative batch mode for the parameter search with GeNN. It uses the same network connectivity for all instances in a batch and thus reduces memory consumption while using all cores of the GPU. To this end, we distributed the instances of a single batch pseudorandomly across the entire grid. The identical connectivity introduces correlations across all network instances within a single batch (see Section Discussion), while a batch size of 1 results in fully independent networks. The shortest run-time was achieved for a batch size of 40 (Table 4) and resulted in a significant speed-up factor of 1.6 when compared to a batch size of one.

To match the batch mode in GeNN, we tested an alternative approach with NEST in which we generate the network model only once and then re-initiate this same network for all parameter combinations in the grid. Now, network connectivity is identical across the complete parameter grid and thus not independent (see Section Discussion). In this approach, the reduced fixed costs for model setup (combined phases of node creation and connection) considerably reduced the overall wall-clock time for completing the grid search by almost 40 % (S3* in Table 4), resulting in a speed-up factor of 1.7 compared to the simulation of independent network instances. The batch-mode approach with GeNN was thus 6.4 times faster than the single network instance approach with NEST (S3*).

We observed a considerable dependence of wall-clock time on the average firing rate for the grid search in NEST simulations. This dependence is comparably weak in the currently tested SPARSE mode with GeNN (Supplementary Figure S2).

Discussion

Limitations of the present study

We here restricted our benchmark simulations with NEST and GeNN to single machines with multiple CPU cores equipped with either a high-end or low-cost GPU (Table 1). We may consider this type of hardware configuration as standard equipment in computational labs. In our simulations with NEST on a single-processor machine (S3), we found that matching the number of threads to the number of cores was most efficient, in line with the results reported by Kurth et al. (2022), while on a dual-processor machine (S1), matching the number of threads and cores resulted in a small loss of speed compared to multiple simulations run in parallel (Table 4). We did not attempt to use the message passing interface (MPI) for distributing processes across the available cores. As pointed out by a previous study (Ippen et al., 2017), this can increase simulation performance with respect to simulation speed but at the same time considerably increases memory consumption, which would further limit the achievable network size.

NEST is optimized for distributed simulation by means of efficient spike communication across machines and processes (Kunkel et al., 2012, 2014; Ippen et al., 2017; Jordan et al., 2018; Pronold et al., 2022). This allows for scaling from single machines to multiple machines and allowed for the simulation of very large SNNs on supercomputers with thousands of compute nodes (Kunkel et al., 2014; Jordan et al., 2018; Schmidt et al., 2018). For the E/I cluster topology of our benchmark model, however, we do not expect a good scaling behavior of simulation speed in distributed environments for two reasons. One limiting factor for simulation speed is the communication of spikes between machines and the spike delivery on each machine. In NEST, communication between machines is optimized by communicating packages of sequential spikes, which requires a sufficiently large minimal synaptic delay (Morrison et al., 2005), and thus, spike delivery on each machine dominates the cost of communication (Jordan et al., 2018; Pronold et al., 2022). Our current model implementation uses the minimum synaptic delay of only a single time step (0.1 ms). Second, the E/I cluster model shares the structural connectivity of the RBN (Figure 1), where the topology of excitatory and inhibitory clusters is defined through connection strengths while connectivity is unaffected and comparably high with a pairwise connection probability of $p \approx 0.3$. In future work, we will consider an alternative structural definition of the cluster topology where the number of connections between E/I clusters is reduced, while it remains high within clusters. Simulating one or several E/I clusters on a single machine could then benefit distributed simulation due to a reduced spike communication between machines. A cluster topology defined by connectivity also opens the possibility to form clusters by means of structural plasticity (Gallinaro et al., 2022). We note that, in our current network definition and for large network size, the number of synapses per neuron exceeds biological realistic numbers in the order of 10,000 synapses per neuron reported in the primate neocortex (Boucsein et al., 2011; Sherwood et al., 2020). However, here we deliberately used a fixed connectivity scheme across the complete investigated range of network sizes. The large number of synapses creates a high computational load for the spike propagation.

A number of GPU-based simulators are currently in use for SNN simulation, such as ANNarchy (Vitay et al., 2015), CARLsim (Niedermeier et al., 2022), BINDSnet (Hazan et al., 2018), GeNN, and NEST GPU (Golosio et al., 2021). These use different design principles (Brette and Goodman, 2012; Vlag et al., 2019) that are optimal for specific use cases. NEST GPU, for example, follows the design principle of NEST allowing for the distributed simulation of very large networks on multiple GPUs (on multiple machines) using MPI. Simulation of the multi-area multi-layered cortical network model as defined in Schmidt et al. (2018) with a size of $N = 4.13 \cdot 10^6$ neurons has recently been benchmarked on different systems. Knight and Nowotny (2021) found that NEST simulation on the JURECA system (Thörnig, 2021) at the Jülich Supercomputing Center was ≈ 15 times faster than simulation with GeNN on a single GPU (NVIDIA TITAN RTX). The recent work by Tiddia et al. (2022) found that NEST GPU (parallel simulation on 32 GPUs, NVIDIA V100 GPU with 16 GB HBM2e) outperformed NEST (simulated on JUSUF HPC cluster, Von St. Vieth, 2021, and JURECA) by at least a factor of two.

Kurth et al. (2022) reported the real-time factor for the multi-layered model of a single cortical column introduced by Potjans and Diesmann (2014) with about $N = 80,000$ neurons and $0,3 \cdot 10^9$ synapses for a simulation with GeNN as $RT = 0.7$ (NVIDIA Titan RTX) and with NEST as $RT = 0,56$ (cluster with two dual-processor machines with 128 cores each). For networks with approximately the same number of neurons and a higher number of connections, we here report similar real-time factors. With GeNN-Sp-S3, we achieved $RT = 0,7$ (cf. Figure 3) for the E/I cluster network with $N = 80,500$ neurons and $2 \cdot 10^9$ synapses, and $RT = 0,56$ (cf. Supplementary Figure S1) for the corresponding RBN. The faster simulation of the RBN results from a lower average firing rate of ≈ 0.9 spikes/s as compared to ≈ 8.5 spikes/s in the E/I cluster network.

Providing comparable benchmarks for the simulation of SNNs across different simulation environments and different hardware systems is generally hampered by two factors. First, different simulators use different design principles. To fully exploit their capabilities in a benchmark comparison, one needs to optimize for each simulator and use case. Second, the community has not agreed on standardized benchmark models (Kulkarni et al., 2021; Steffen et al., 2021; Albers et al., 2022; Ostrau et al., 2022). The RBN is widely used in computational neuroscience (cf. Supplementary Figure S1). However, its definition varies considerably across studies, e.g., with respect to connection probabilities, fixed vs. distributed in-degrees of synaptic connections, neuron and synapse models, or the background stimulation by constant or noise current input. Second, SNN simulation environments are subject to continuous development affecting optimization for speed and memory consumption, which complicates comparability across different versions. We thus did not attempt to directly compare the run-time performance obtained for the model simulations considered in the present study to the performances reported in previous studies.

NEST provides a high degree of functionality, good documentation, and many implemented neuron and synapse models. This results in a high degree of flexibility allowing, for instance, to introduce distributed parameters by using a NEST function that passes the respective distribution parameters as arguments when initializing the model, which is then set up from scratch. One obvious limitation of the flexibility of GeNN is the high fixed costs for model definition and building the model on the CPU and for loading the model on the GPU before it can be initialized. This limits its flexible use in cases where non-global parameters of a model change, which cannot be changed on the GPU. We would thus like to encourage the development of a method that automatically translates selected model parameters into GeNN variables for a given model definition, allowing to change those model parameters without recompilation between simulations. This functionality would allow to fully exploit the simulation speed on the GPU and benefit the time-to-solution while reducing the likelihood of implementation errors.

Efficient long-duration and real-time simulation on the GPU

Our results show that GPU-based simulation can support the efficient simulation over long biological model times (Figure 3). This is desirable, e.g., in spiking models that employ structural (Deger et al., 2012; Gallinaro et al., 2022) or synaptic (Vogels et al., 2011; Sacramento et al., 2018; Zenke and Ganguli, 2018; Illing et al., 2021; Asabuki et al., 2022) plasticity to support continual learning and the formation and recall of short-term (on the time scale of minutes), middle-term (hours), or long-term (days) associative memories. Similarly, simulating nervous system control of behaving agents in approaches to computational neuroethology may require biological model time scales of minutes to hours or days. The low variable simulation costs achieved with GeNN can also benefit real-time simulation of SNNs, e.g., in robotic application.

We here considered spiking networks in the approximate size range from one thousand to a few million neurons. This range covers the complete nervous system of most invertebrate and of small vertebrate species as shown in Figure 6 and Table 5, including, for instance, the adult fruit fly *Drosophila melanogaster* with $N \approx 100,000$ neurons in the central brain (Raji and Potter, 2021), the European honeybee *Apis mellifera* with $N \approx 900,000$ (Witthöft, 1967; Menzel, 2012; Godfrey et al., 2021), and the zebrafish *Danio rerio* with $N \approx 10 \cdot 10^6$. In mammals, it covers a range of subsystems from a single cortical column with approximately $30,000 - 80,000$ neurons (Boucsein et al., 2011; Potjans and Diesmann, 2014; Markram et al., 2015) to the complete neocortex of the mouse ($N \approx 5 \cdot 10^6$ neurons) (Herculano-Houzel et al., 2013). Models exceeding this scale are currently still an exception (Eliasmith and Trujillo, 2014; Kunkel et al., 2014; Van Albada et al., 2015; Jordan et al., 2018; Igarashi et al., 2019; Yamaura et al., 2020) and typically require the use of a supercomputer.

Benchmarking with grid search

Spiking neural networks have a large set of parameters. These include connectivity parameters and parameters of the individual neuron and synapse models. Thus, both in scientific projects and for the development of real-world applications of SNNs, model tuning through parameter search typically creates the highest demand in computing time. Currently available methods (Feurer and Hutter, 2019) such as grid search or random search (LaValle et al., 2004; Bergstra and Bengio, 2012), Bayesian optimization (Parsa et al., 2019), and machine learning approaches (Carlson et al., 2014; Yegenoglu et al., 2022) require extensive sampling of the parameter space. We therefore suggest to include the parameter search in benchmarking approaches to the efficient simulation of large-scale SNNs.

To this end, we exploited two features of GeNN: batch processing and the on-GPU initialization of the model. Batch processing was originally introduced to benefit the execution of machine learning tasks with SNNs. It enables the parallel computation of multiple model instances within a batch. In our example with a network size of $N = 25,000$ and for simulating a biological model time of 10 s, we obtained a speed-up factor of 1.6 for a batch size of 40 compared to a single model instance per run. The current version of GeNN requires that all model instances of a batch use identical model connectivity. Thus, quantitative results, e.g., of the average firing rates (Figure 5) are correlated across all samples within one batch (for batch size > 1). We distributed the 40 instances of one batch pseudorandomly across the grid such that correlations are not systematically introduced among neighboring samples in the grid. An important future improvement of the batch processing that will allow for different connectivity matrices within a batch and thus for independent model connectivities is scheduled for the release of GeNN 5.0 (https://github.com/genn-team/genn/issues/463). The possibility to initialize and re-initialize a once defined model and connectivity on the GPU (Knight and Nowotny, 2018) uses the flexibility of the code generation framework. This allows to define, build, and load the model to the GPU once and to repeatedly initialize the model on the GPU with a new connectivity matrix (per batch). It also allows for the variable initialization of, e.g., the initial conditions of model variables such as the neurons' membrane potentials. In addition, global parameters can be changed during run-time. After initialization of a model this allows, for example, to impose arbitrary network input as pre-defined in an experimental protocol.

We here propose that the batch processing with GeNN can be efficiently used not only to perform parameter search but also to perform batch simulations of the identical model with identical connectivity in parallel. This can be beneficial, e.g., to generate

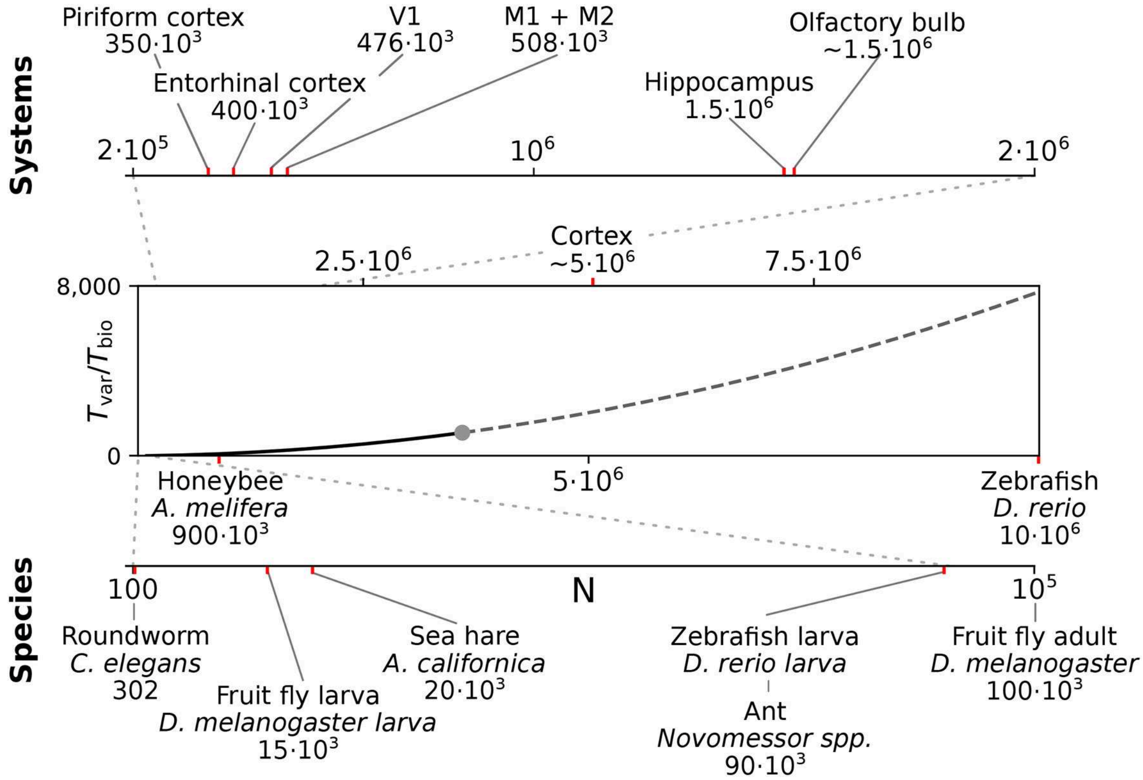


FIGURE 6
Full system modeling with spiking neurons. Wall-clock time relative to simulated biological model time $T_{\text{var}} / T_{\text{bio}}$ over a realistic range of network sizes for spiking network simulation. Network size is projected to the total number of neurons per hemisphere for subsystems in the mouse **(top)** and in the CNS of invertebrate and small vertebrate model species **(bottom)**; (see also Table 5). The graph sketches the quadratic dependence on network size N for large N fitted for the PROCEDURAL connectivity simulation method with GeNN (Figure 3, see Section Materials and methods). The solid line shows the range of network sizes covered in our simulations (cf. Figure 2) using a high-end GPU with 48 GB RAM (S3, Table 2). The dashed line extrapolates to a maximum size of 10 million neurons assuming larger GPU RAM that would allow to cover the neocortex of the mouse and the complete CNS of the zebrafish.

multiple simulation trials for a given stimulation protocol allowing for across-trial statistics, or to efficiently generate responses of the same model to different stimulation protocols within a single batch. In NEST, this can be efficiently achieved by re-initialization of the identical network (Table 4). We note that we have tested one additional alternative approach to the grid search with NEST where we set up the model connectivity once and, afterwards, for each initialization reconstructed the network from the stored connectivity. This leads to a significant reduction in performance for large networks (data not shown).

Networks with heterogeneous neuron and synapse parameters

SNNs are typically simulated with homogeneous parameters across all neuron and synapse elements of a certain type. Using heterogeneous parameters that follow experimentally observed parameter distributions increases the biological realism of a model and has been argued to benefit model robustness and neuronal population coding (Mejias and Longtin, 2012, 2014; Lengler et al., 2013; Tripathy et al., 2013; Gjorgjieva et al., 2016; Litwin-Kumar et al., 2016). In the present study, we performed benchmark simulations with heterogeneity in the single parameter of synaptic time constant to quantify its effect on simulation costs. The efficient solution provided by NEST for the specific neuron model used here does neither increase fixed costs nor variable costs (Figure 4) in line with the results of Stimberg et al. (2019), albeit with the limitation to one single time constant per synapse type (excitatory and inhibitory) for each postsynaptic neuron. NEST offers an alternative neuron model that allows the definition of an arbitrary time constant for each synapse (see Section Materials and methods) that was not tested in the present study. In GeNN, we had deliberately defined our neuron and synapse models separately (see Section Materials and methods), because in future work, we aim at introducing stochasticity of synaptic transmission to capture the inevitable variability of synaptic transmission in biology (Nawrot et al., 2009; Boucsein et al., 2011) that has been argued to support efficient population coding (Lengler et al., 2013).

Metastability emerges robustly in attractor networks with large E/I clusters and heterogeneous synaptic time constants

With respect to attractor network computation, an important question is whether the functionally desired metastability can be reliably achieved in large networks and for large population sizes of neuron clusters. In our previous work, we had limited our study of attractor networks to a maximum network size of 5,000 neurons

TABLE 5 Number of neurons of selected model organisms and subsystems.

	Neurons $[\cdot 10^3]$	References
Invertebrate		
Round worm (*Caenorhabditis elegans*)	0.302	White et al., 1986
Fruit fly larva (*Drosophila melanogaster*)	15	Eschbach and Zlatic, 2020
California sea hare (*Aplysia californica*)	20	Zhao and Wang, 2009
Fruit fly adult (*Drosophila melanogaster*)	70–200	Godfrey et al., 2021 Raji and Potter, 2021
Ant (*Novomessor spp.*)	90	Godfrey et al., 2021
European honeybee, worker (*Apis mellifera*)	600–900	Witthöft, 1967 Godfrey et al., 2021
Vertebrate		
Human: Medial Superior olive	15.5	Kulesza, 2007
Zebrafish larva (*Danio rerio*)	90	Bruzzone et al., 2021
Sprague Dawley rat: Basal ganglia	2,900	Oorschot, 1996
Zebrafish adult (*Danio rerio*)	10,000	Hinsch and Zupanc, 2007
Smoky shrew (*Sorex fumeus*)	39,490	Sarko et al., 2009
Mouse (C57BL/6J) per hemisphere		
Piriform cortex	350	Herculano-Houzel et al., 2013
Entorhinal cortex	400	Herculano-Houzel et al., 2013
Visual cortex V1	476	Herculano-Houzel et al., 2013
Motor cortex M1+M2	508	Herculano-Houzel et al., 2013
Hippocampus	1,500	Herculano-Houzel et al., 2013
Olfactory bulb	1,520	Parrish-Aungst et al., 2007
Cortex	5,049	Herculano-Houzel et al., 2013

and could show that the topology of excitatory-inhibitory clusters benefit metastability for a varying number and size of clusters while pure excitatory clustering failed to support metastability for larger cluster size (Rost et al., 2018; Rostami et al., 2022). In our calibration approach of Figure 5, we simulated networks of $N = 25,000$ with a cluster size of 1,000 excitatory and 250 inhibitory neurons. This results in the robust emergence of metastable activity for a reasonable regime of excitatory and inhibitory background currents. Metastability was retained when introducing distributed excitatory and inhibitory time constants (Figure 4). We hypothesize that, due to its local excitation-inhibition balance (Rostami et al., 2022), the E/I cluster topology affords metastability for very large network and cluster sizes.

Materials and methods

Hardware configurations

We perform benchmark simulations on hardware systems that can be considered as standard equipment in a computational research lab. We did not attempt to use high performance computing facilities that, for most users, are available only for highly limited computing time and require an overhead in scheduling simulation jobs. We employ three computer server systems specified in Table 1, which were acquired between 2016 and 2022. The amount of investment at the time of purchase has been fairly stable on the order of \$7,000 — \$10,000 depending on whether a state-of-the-art GPU was included. Servers S1 and S3 are equipped with GPUs. The GeForce GTX 970 (S1; NVIDIA, Santa Clara, USA) can now be considered a low-cost GPU in the price range of \$300. The Quadro RTX A6000 (S3; NVIDIA, Santa Clara, USA) is one example of current state-of-the-art high-end GPUs, for which prices vary in the range of \$3,000—\$5,000. We use the job scheduler HTCondor (Thain et al., 2005) on all servers independently, with one job scheduler per server. We ensure acquisition of all cores and the complete GPU to a running job and prevent other jobs from execution till the running job is finished.

Simulators

We benchmark with the Neural Simulation Tool (NEST) (Gewaltig and Diesmann, 2007) and the GPU-enhanced Neuronal Networks (GeNN) framework (Yavuz et al., 2016) that follow different design principles and target different hardware platforms. The Neural Simulation Tool (NEST) (Gewaltig and Diesmann, 2007) is targeted toward computational neuroscience research and provides various biologically realistic neuron and synapse models. Since the introduction of NESTML (Plotnikov et al., 2016), it also allows custom model definitions for non-expert programmers. NEST uses the so-called simulation language interpreter (SLI) to orchestrate the simulation during run-time as a stack machine. This allows a modular design of the whole simulator and thus the usage of pre-compiled models. NEST supports parallelization across multiple threads via Open Multi-Processing (OpenMP) as well as multiple processes, which can be distributed across multiple machines *via* the message passing interface (MPI). It is suitable for the whole range of desktop workstations to multi-node high-performance clusters. We use NEST version 3.1 (Deepu et al., 2021) (in its standard cmake setting) with the Python interface PyNEST (Eppler et al., 2009) and Python version 3.8 to define our model and control the simulation.

GeNN is a C++ library to generate code from a high-level description of the model and simulation procedure. It employs methods to map the description of the neuron models, the network topology, as well as the design of the experiment to plain C++ code, which then is built for a specific target. GeNN supports single-threaded CPUs or a GPU with CPU as host. The scope of GeNN is broader than that of NEST. With features like the batch-mode, which allows for inference of multiple inputs, GeNN becomes especially useful for machine learning tasks with SNNs as well as for general research in computational neuroscience. GeNN is more rigid in the network topology and in its parameters after the code generation is finished. It does not support general reconfiguration of the network during the simulation. If GeNN is used on a GPU, the CPU is used to generate and build the simulation code and orchestrate the simulation. All other time-consuming processes such as initialization of the connectivity and variables of the model, update of state variables during simulation, and spike propagation are performed by the GPU. The model construction in the GPU memory is run by loading the model or can be rerun by reinitializing the model, which affects the connectivity and state variables, as well as the spike buffers. The code generation framework in GeNN allows for a heavily optimized code depending on the use case. One of these optimization possibilities is the choice of connectivity matrices. Here, we utilize only the SPARSE and the PROCEDURAL connectivity as explained below.

SPARSE connectivity. The connectivity matrix is generated on the GPU during the loading of the model and if the model is reinitialized. It is persistent during the simulation. No additional computational load is generated during the simulation.

PROCEDURAL connectivity. Single elements of the connectivity matrix are generated on demand during simulation for the spike propagation. Only fixed random seeds are saved during the loading of the model and if the model is reinitialized. An additional computational load is caused during the simulation, but the memory consumption is low.

Furthermore, we use global synapse models for all simulations except for simulations with distributed synapse parameters. In simulations with distributed synaptic time constants, we use synapse models with individual postsynaptic model variables. We use the released GeNN version 4.6.0 and a development branch of version 4.7.0, which is now merged into the main and released (commit ba197f24f), with its Python interface PyGeNN (Knight et al., 2021) and Python version 3.6.

Neuron models and network architectures

We use the spiking neural network described by Rostami et al. (2022) as benchmarking model. This model uses leaky integrate-and-fire neurons with exponentially shaped postsynaptic currents. The subthreshold dynamics evolves according to

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \frac{-(V-E_L)}{\tau_m} + \frac{I_{\text{syn}}+I_x}{C_m}$$

and the synaptic current to a neuron i evolves according to

$$\begin{aligned}\tau_{\text{syn}}^{ij}\frac{\mathrm{d}I_{\text{syn}}^{ij}}{\mathrm{d}t} &= -I_{\text{syn}}^{ij} + J_{ij}\sum_k \delta\left(t-t_k^j\right)\\ I_{\text{syn}}^{i} &= \sum_j I_{\text{syn}}^{ij},\end{aligned}$$

where t_k^j is the arrival of the k^{th} spike of the presynaptic neuron j and δ is the Dirac delta function.

We use the NEST model *iaf_psc_exp*, which employs an exact integration scheme as introduced by Rotter and Diesmann (1999) to solve the above equations efficiently for two different synaptic input ports. The input ports can have different time constants. We implement the same integration scheme in GeNN but modify it to fit only one synaptic input port while treated as piece-wise linear to combine different synapse types in terms of their time constant.

We follow the widely used assumption that 80% of neurons in the neocortex are excitatory while 20% are inhibitory to build the network model, which is based on the statistical work in the mouse neocortex by Braitenberg and Schüz (1998). Connections between neurons are established with the probabilities $p_{\text{EE}} = 0.2$, $p_{\text{EI}} = p_{\text{IE}} = p_{\text{II}} = 0.5$. Autapses and multapses are excluded. Excitatory and inhibitory populations are divided into N_Q clusters by increasing the weights of intra-cluster connections while decreasing the weights of inter-cluster connections. Weights are calculated to ensure a local balance between excitation and inhibition, as introduced before for binary networks (Rost et al., 2018). Parameters are given in Tables 6, 7. For comparability, we matched our parameters to those used in previous related works (Litwin-Kumar and Doiron, 2012; Mazzucato et al., 2015; Rost, 2016; Rostami et al., 2022). Synaptic weights J_{XY} as well as the background stimulation currents are scaled by the membrane capacitance C. By this scaling, the capacitance has no influence on the dynamics of the network but only on the magnitude of PSCs and the background stimulation currents. Following previous studies, we set the value of capacitance to $C= 1pF$.

In our model, each neuron can be presynaptic and postsynaptic partner to all other neurons. As a result, the number of synapses M scales quadratically with the neuron number N. We calculate the expectation of the number of synapses M by using the assumption of portioning into 80% excitatory and 20% inhibitory neurons and calculating the expected number of synapses between the combinations of both by using the connection probabilities. Due to our exclusion of autapses, we have to reduce the number of postsynaptic possibilities by one for the connections among excitatory neurons as well as among inhibitory neurons. The overall network connectivity $p = 0.308$ is determined by

$$\begin{aligned}E(M) &= (N\cdot 0.8)\cdot(p_{\text{EE}}\cdot(N\cdot 0.8-1)+p_{\text{IE}}\cdot(N\cdot 0.2))\\ &\quad +(N\cdot 0.2)\cdot(p_{\text{EI}}\cdot(N\cdot 0.8)+p_{\text{II}}\cdot(N\cdot 0.2-1)) \qquad (1)\\ &= 0.308\cdot N^2 - 0.25\cdot N \approx 0.308\cdot N^2.\end{aligned}$$

In addition, we implement a model with excitatory and inhibitory synaptic time constants drawn from a uniform distribution with the same means as provided in Table 6 and a standard deviation of 5% of the mean. In NEST, this is achieved by using the nest.random.uniform function as argument for the parameters *tau_syn_ex* and *tau_syn_in* of the neuron model. This results in distributed synaptic time constants across neurons. Thus, all synaptic inputs of one input type (excitatory and inhibitory) to a postsynaptic neuron have the same time constant. Replacing the neuron model by a multisynapse neuron model (e.g., *iaf_psc_exp_multisynapse*) would enable to use different time constants for subsets of the synaptic inputs of a single neuron. In GeNN, we implement the distribution of synaptic time constants by re-implementing the postsynaptic model of an exponential PSC. We provide the decay factor $\exp(-\frac{\Delta t}{\tau_{\text{syn}}})$ as variable to minimize the calculations during the simulation. GeNN only allows random number initialization for variables and not for parameters. Parameters within a group (neurons as well as synapses are generated as groups) have to be the same in GeNN. To initialize the decay factors corresponding to the uniform distribution of synaptic time constant, we define a custom variable initialization method. The generated network model does not enforce the same synaptic time constants for all inputs of a neuron. Deviating from the NEST model, the synaptic time constants are thus distributed across all connections rather than across neurons. A limited distribution across neurons as in NEST could be implemented in GeNN by implementing the synapse dynamics in the neuron model.

TABLE 6 Constant model parameters for all simulation modalities.

Parameter		Unit	Value
Simulation resolution	Δt	ms	0.1
Resting potential	E_L	mV	0
Threshold voltage	V_{th}	mV	20
Reset voltage	V_r	mV	0
Membrane capacitance	C	pF	1
Membrane time constant (exc. neurons)	${\tau_m}^{\mathrm{E}}$	ms	20
Membrane time constant (inh. neurons)	${\tau_m}^{\mathrm{I}}$	ms	10
Synaptic time constant (exc. synapses)	${\tau_{\mathrm{syn}}}^{\mathrm{E}}$	ms	3
Synaptic time constant (inh. synapses)	${\tau_{\mathrm{syn}}}^{\mathrm{I}}$	ms	2
Synaptic delay	D_{syn}	ms	0.1
Absolute refractory period	τ_r	ms	5
Connectivity probability EE	p_{EE}	-	0.2
Connectivity probability others	$p_{\mathrm{EI}}, p_{\mathrm{IE}}, p_{\mathrm{II}}$	-	0.5
Relative strength of inhibition	g	-	1.2
Number of clusters	N_Q	-	20
Proportionality factor inhibitory to excitatory clustering	R_J	-	3/4

TABLE 7 Variable model parameters for different simulation modalities.

Parameter		Unit	Value			
			5,000	50,000	Var N	Grid
Number exc. neurons	N_{E}	-	4,000	40,000	$0.8 \cdot N$	20,000
Number inh. neurons	N_{I}	-	1,000	10,000	$0.2 \cdot N$	5,000
Current stimulation exc. neurons	$I_{x\mathrm{E}}$	pA	2.13	2.13	2.13	0.95–2.9
Current stimulation inh. neurons	$I_{x\mathrm{I}}$	pA	1.24	1.24	1.24	0.7–1.675
Clustering strength exc. neurons	$J_{\mathrm{E}+}$	-	10.0	10.0	10.0	2.75
Synaptic weight EE (RBN)	J_{EE}	pA	0.329	0.104	$f(N)$	0.147
Synaptic weight IE (RBN)	J_{IE}	pA	0.250	0.079	$f(N)$	0.112
Synaptic weight EI (RBN)	J_{EI}	pA	–0.877	–0.277	$f(N)$	–0.392
Synaptic weight II (RBN)	J_{II}	pA	–1.337	–0.423	$f(N)$	–0.598

$f(N)$ denotes the dependency of the synaptic weights (J_{XY}) on the network size. $J_{\mathrm{E}+}$ is the clustering strength of the excitatory neurons and influences the inhibitory clustering strength *via* the proportionality factor R_J. Implementation details are described in Rostami et al. (2022).

Simulations

We perform two different sets of simulations with all tested combinations of a server and a simulator. We enforce a maximum run-time of 5 h per simulation. The first set contains simulations of two networks with the sizes of 5,000 and 50,000 neurons and simulation times of 0, 1, 5, 25, 50, 100, 175, 350, 625, 1,250, and 2,500 s of biological model time (0 s is used to determine the fixed costs for simulation preparation). We discard 10% as presimulation time from our analyses of network activity. We execute each simulation five times with different seeds of the random number generator. All simulations in the second set are 10 s long (biological model time; 1 s presimulation time and 9 s simulation time), and we used network sizes between 500 neurons and $\approx 3,6$ million neurons. We execute each simulation 10 times with different seeds of the random number generator. Based on the recorded wall-clock times, we determine the maximum network size for each configuration that fulfills real-time requirements.

For GeNN, we use the spike recorder, which saves the spikes during the simulation in the RAM of the GPU and allows a block transfer of all spikes for a given number of simulation steps. The consumption of GPU-RAM is composed of the model with all its state variables and connectivity matrix, and of the memory for the spike recorder. The memory consumption of the model is independent of the simulated biological model time, but heavily dependent on the network size and the choice of the connectivity matrix type. The size of the memory of the spike recorder depends on the network size and the number of simulation steps, and thus on the simulated biological model time. The dependence of the memory consumption on network size and biological model time to be simulated is analyzed in Supplementary Figure S3. We implement a partitioning of simulations into sub-simulations, if the GPU-RAM is too small to fit the model and the spike recorder in one simulation. We determine the maximum number of simulation steps for a large network that fits in the GPU-RAM and use this number as a heuristic. If the product of network size and simulation steps exceeds the heuristic, we iteratively divide the simulation into two until the product of both is smaller than the heuristic. As a fallback mechanism, we include an error handling to divide the simulation further, if the model and the spike recorder do not fit the GPU-RAM. We transfer the spikes from the GPU to the host after each sub-simulation and process them into one list containing the neuron-ID and the spike time by a thread pool with a size of 16 workers. A greater number of workers exceeded the available RAM on our servers for large networks. All simulations are performed once with the SPARSE connectivity format and once with the PROCEDURAL connectivity. We delete the folder with the generated and compiled code before we submit the first job to the scheduler to prevent the use of code from previous code generation runs. We run one complete sequence of all different biological model times before switching to the other matrix type. This ensures that the code is generated five and 10 times, respectively, for the two different simulation sets.

We use the same order of execution for our simulations in NEST. We match the number of OpenMP threads of the simulations in NEST to the number of cores available $n_T = n_{\text{cores}}$ on the server. We do not utilize parallelization with MPI to minimize memory consumption (Ippen et al., 2017) and despite possible advantages in speed.

We additionally simulate the E/I cluster network model with distributed synaptic time constants with both simulators on S3 as described earlier. The network with $J_{E+} = 1$ represents the random balanced network (RBN) with constant background current stimulation. For this, we use the same code and parameters as for the clustered network model for all calibrations.

Grid search

We implement a general framework to perform a grid search. The framework takes multiple parameters and generates a regular mesh grid of the parameter values. It simulates each point in the grid, analyzes the network activity in the simulation, and saves the result together with its position in the grid to a pickle file. Pickle is a module from the Python Standard Library, which can serialize and de-serialize Python objects and save them to binary files. Pickle allows saving multiple objects in one file. Simulations are performed with the same network definition as used for the simulations before. All simulations of individual samples are 10 s long (biological model time; 1 s presimulation time and 9 s simulation time). We simulate a 2D grid with 40 × 40 samples.

We implement the grid search in GeNN by utilizing the batch-mode with n_{Batch} network instances in each run. We add global parameters to the model description. Global parameters are not translated into constants during compilation, but can be set independently for each instance in a batch and can be modified during run-time. The instances share all other parameters and the specific connectivity matrices. We use the SPARSE connectivity format and generate the simulation code only one time and then reinitialize the network after each batch and set the global parameters to their respective values. The reinitialization regenerates the connectivity matrix and resets all state variables as well as the spike buffer. The parameters as well as the batch size n_{batch} can only be changed by recompilation. Each time a number of samples equal to n_{batch} is drawn without replacement from the grid until all points in the grid are simulated. If fewer samples as n_{batch} are left, the free slots are filled by setting the global parameters to 0; the results of these slots are ignored. After a simulation, the spike times are transferred in a block transfer and are processed by $n_W = 24$ workers into a matrix with the size $2 \times n_{\text{spikes}} \times n_{\text{batch}}$, where the first row contains the spike times, the second row the ID of the neuron, which emitted the spike, and the third dimension corresponds to the different network instances in the batch. We use the same representation of the spike times and the senders for GeNN and NEST. The processing takes a reasonable amount of computation time as GeNN implements the spike recorders in the neuron populations and returns no global IDs but instead IDs for the neurons in the specific population. Afterward, we analyze the spike times serially for each instance with the specific analyses, which are the firing rates of the excitatory and inhibitory populations in this grid search. The process writes the results afterward to the pickle file. We test the grid search with different batch sizes on S1 and S3 (see Table 4).

We implement the grid search in NEST by creating a list of all parameters in the grid and then parallelizing the simulations with Pathos by n_W workers. The workers import NEST independently, thus each simulation is completely independent. Each worker utilizes n_T threads. We match the number of cores available on the system: $n_{\text{cores}} = n_W \cdot n_T$. The workers write the results to one pickle file, which is protected by a lock to ensure data integrity by only allowing one worker to write at a time. This pickle file does not contain by default the spike times but instead only the result of the specific analyses applied. We test different combinations of n_W and n_T on S2 and S3 (see Table 4).

We extend the NEST implementation for grid search by allowing the re-use of a once set up network. Before each run, the spike recorder is reset to zero events, the membrane voltages are reinitialized, and the parameters are set for the current run. All runs thus use the same network with fixed connectivity.

Definition of fixed and variable simulation costs

We divide the simulation cost into fixed and variable costs:

$$T_{\text{sim}} = T_{\text{fix}} + T_{\text{var}}.$$

The fixed costs involve all steps necessary to set up and prepare the model before the first simulation step. They are independent of the biological model time to be simulated. The variable costs involve the actual propagation of the network state during each time step and the overall wall-clock time used for data collection during a simulation. We include timestamps in the simulation scripts to access the run-time of different execution phases. Due to the different design principles, not all phases of GeNN can be mapped to NEST. Table 8 shows the defined phases and the commands,

TABLE 8 Execution phases of simulation.

	GeNN		NEST	
Fixed	Model definition	*model.add_neuron_population()*	Node creation	*nest.Create()*
		model.add_synapse_population()	Connection	*nest.Connect()*
	Build	*model.build()*		*nest.Prepare()*
	Load	*model.load()*		
Variable	Simulation	*model.step_time()*	Simulation	*nest.Run()*

	Download	*model.pull_recording_buffers_from_device()*
		+ conversion to spike representation
			Download	*nest.GetStatus(spike_recorder)*
				+ conversion to spike representation
				nest.Cleanup()

which are executed in each phase. We included in the model definition phase of GeNN all steps necessary to set up the model description and the simulation itself. This contains the definition of neurons, their connectivity, the setup of the spike recorders, the stimulation of neurons, and the definition of the neuron parameters, as well as the parameters of the simulation itself, such as the time resolution. During the build phase in GeNN, the code generation is executed based on the defined model and the code is compiled for its target, which in our case is the specific server with a GPU. This process is not necessary for NEST because of its design using an interpreter to orchestrate the simulation by using pre-compiled models. The load phase in GeNN contains the construction of the model and all needed procedures to simulate it in the host memory and in the GPU memory and the initialization of the model. This includes the connectivity, variables of the model, and variables of the general simulation. The initialization of the model can be also re-run manually to generate a fresh model based on the currently loaded code, as used for the grid search with GeNN. The model setup in NEST comprises the node creation and the connection phase. The former creates the structures of neurons, spike recorders and stimulation devices, and the general simulation parameters such as the parallelization scheme. In the latter, the structures of the connectivity of nodes is created. Due to the ability of NEST to use distributed environments, presynaptic connectivity structures can only be created after the calibration of the system. This calibration contains the determination of delays between MPI processes if used, the allocation of buffers, and calibrating nodes. The simulation phase contains for both simulators the propagation of the state and the delivery of spikes. In the case of GeNN, each call of the simulate function only simulates one time step and thus the call has to be issued as many times as needed to complete the simulation. In the case of NEST, this is done automatically. The download phase contains the query of the recorded spikes and the generation of our used format of representation as described for the grid search. In the case of GeNN, the conversion is proceeded by the transfer of the recorded spikes from the GPU to the host.

We report the median values of fixed and variable simulation costs across repeated simulations. Supplementary Figure S4 additionally provides the mean and standard deviation for the calibrations shown in Figure 3.

Extrapolation to large network sizes

We can approximate the parabolic dependence of the proportionality factor $\mathrm{RT} = T_{\mathrm{var}}/T_{\mathrm{bio}}$ on the number of neurons N for the PROCEDURAL simulation approach with GeNN (Figures 3B, C) by the polynomial function

$$\begin{aligned}\varphi(N,\boldsymbol{\alpha}) &= \alpha_2 \cdot 0.308N^2 + \alpha_1 \cdot N + \alpha_0 \\ \ell &= \sum_{i=1}^{n} \left(\frac{\lambda}{F(N)} + (1-\lambda) \right) \left(\frac{\varphi(N,a)_i - \mathrm{RT}_i}{\mathrm{RT}_i} \right)^2 \\ \boldsymbol{\alpha} &= \underset{\boldsymbol{\alpha}}{argmin}\,(\ell)\,. \end{aligned} \tag{2}$$

To this end, we used a weighted least squares fit of Equation 2 with three modifications: (*i*) We scale the quadratic term by the connection density $p = 0.308$ to relate this term to the number of synapses M, (*ii*) we use the relative error, and (*iii*) we weigh the samples by the inverse of the density along the independent variable N. This balances the influence of the large network with a smaller number of samples and the larger number of samples for small networks with the factor $\lambda = 0.75$. We estimate the density $F(N)$ by a kernel density estimation using a Gaussian kernel ($\sigma = 10,000$ neurons). The resulting fit is used for the extrapolation of $T_{\mathrm{var}}/T_{\mathrm{bio}}$ to larger network sizes as shown in Figure 6. Simulation of larger networks will require larger GPU RAM.

Data availability statement

The original contributions presented in the study are publicly available. This data can be found here: https://github.com/nawrotlab/SNN_GeNN_Nest.

Author contributions

FS and MN designed the research and wrote the manuscript. FS carried out simulations and analysis of results. VR and MN supervised project. All authors contributed to the article and approved the submitted version.

Funding

This research was supported by the German Research Foundation through the Collaborative Research Center Key Mechanisms of Motor Control in Health and Disease (DFG-CRC 1451, grant no. 431549029 to MN, https://www.crc1451.uni-koeln.de/). FS was funded through the DFG Research Training Group Neural Circuit Analysis (DFG-RTG 1960, grant no. 233886668).

Acknowledgments

We are grateful to James Knight for his support with GeNN.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fninf.2023.941696/full#supplementary-material

References

Abeles, M. (1991). *Corticonics: Neural Circuits of the Cerebral Cortex*. Cambridge: Cambridge University Press.

Albers, J., Pronold, J., Kurth, A. C., Vennemo, S. B., Haghighi Mood, K., Patronis, A., et al. (2022). A modular workflow for performance benchmarking of neuronal network simulations. *Front. Neuroinform.* 16, 837549. doi: 10.3389/fninf.2022.837549

Amit, D. J., and Brunel, N. (1997). Model of global spontaneous activity and local structured activity during delay periods in the cerebral cortex. *Cereb. Cortex* 7, 237–252. doi: 10.1093/cercor/7.3.237

Asabuki, T., Kokate, P., and Fukai, T. (2022). Neural circuit mechanisms of hierarchical sequence learning tested on large-scale recording data. *PLoS Comput. Biol.* 18, 1–25. doi: 10.1371/journal.pcbi.1010214

Bartolozzi, C., Indiveri, G., and Donati, E. (2022). Embodied neuromorphic intelligence. *Nat. Commun.* 13, 1024. doi: 10.1038/s41467-022-28487-2

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T., Rasmussen, D., et al. (2014). Nengo: a Python tool for building large-scale functional brain models. *Front. Neuroinform.* 7, 48. doi: 10.3389/fninf.2013.00048

Ben-Shalom, R., Ladd, A., Artherya, N. S., Cross, C., Kim, K. G., Sanghevi, H., et al. (2022). NeuroGPU: Accelerating multi-compartment, biophysically detailed neuron simulations on GPUs. *J. Neurosci. Methods* 366, 109400. doi: 10.1016/j.jneumeth.2021.109400

Bergstra, J., and Bengio, Y. (2012). Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13, 281–305.

Blundell, I., Brette, R., Cleland, T. A., Close, T. G., Coca, D., Davison, A. P., et al. (2018). Code generation in computational neuroscience: a review of tools and techniques. *Front. Neuroinform.* 12, 68. doi: 10.3389/fninf.2018.00068

Boucsein, C., Nawrot, M. P., Schnepel, P., and Aertsen, A. (2011). Beyond the cortical column: abundance and physiology of horizontal connections imply a strong role for inputs from the surround. *Front. Neurosci.* 5, 32. doi: 10.3389/fnins.2011.00032

Braitenberg, V., and Schüz, A. (1998). *Cortex: Statistics and Geometry of Neuronal Connectivity*. Berlin; Heidelberg: Springer Berlin Heidelberg.

Brette, R., and Goodman, D. F. (2012). Simulating spiking neural networks on GPU. *Network Comput. Neural Syst.* 23, 167–182. doi: 10.3109/0954898X.2012.730170

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398. doi: 10.1007/s10827-007-0038-6

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 183–208. doi: 10.1023/A:1008925309027

Bruzzone, M., Chiarello, E., Albanesi, M., Miletto Petrazzini, M. E., Megighian, A., Lodovichi, C., et al. (2021). Whole brain functional recordings at cellular resolution in zebrafish larvae with 3d scanning multiphoton microscopy. *Sci. Rep.* 11, 11048. doi: 10.1038/s41598-021-90335-y

Büsing, L., Schrauwen, B., and Legenstein, R. (2010). Connectivity, dynamics, and memory in reservoir computing with binary and analog neurons. *Neural Comput.* 22, 1272–1311. doi: 10.1162/neco.2009.01-09-947

Carlson, K. D., Nageswaran, J. M., Dutt, N., and Krichmar, J. L. (2014). An efficient automated parameter tuning framework for spiking neural networks. *Front. Neurosci.* 8, 10. doi: 10.3389/fnins.2014.00010

Chicca, E., Stefanini, F., Bartolozzi, C., and Indiveri, G. (2014). Neuromorphic electronic circuits for building autonomous cognitive systems. *Proc. IEEE* 102, 1367–1388. doi: 10.1109/JPROC.2014.2313954

Deepu, R., Spreizer, S., Trensch, G., Terhorst, D., Vennemo, S. B., Mitchell, J., et al. (2021). *Nest 3.1*. Zenodo. doi: 10.5281/zenodo.5508805

Deger, M., Helias, M., Rotter, S., and Diesmann, M. (2012). Spike-timing dependence of structural plasticity explains cooperative synapse formation in the neocortex. *PLoS Comput. Biol.* 8, 1–13. doi: 10.1371/journal.pcbi.1002689

Diesmann, M., and Gewaltig, M.-O. (2002). "NEST: an environment for neural systems simulations," in *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001, GWDG-Bericht* (Göttingen: Ges. für Wiss. Datenverarbeitung). Available online at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=74eebbc23056acb83796673c6bb51dd41deb21db

Diesmann, M., Gewaltig, M.-O., and Aertsen, A. (1995). *Synod: An environment for neural systems simulations language interface and tutorial*. Technical report, The Weizmann Institute of Science, 76100 Rehovot.

Diesmann, M., Gewaltig, M.-O., and Aertsen, A. (1999). Stable propagation of synchronous spiking in cortical neural networks. *Nature* 402, 529–533. doi: 10.1038/990101

Eliasmith, C., and Anderson, C. H. (2003). *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. Cambridge: MIT Press.

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, Y., et al. (2012). A large-scale model of the functioning brain. *Science* 338, 1202–1205. doi: 10.1126/science.1225266

Eliasmith, C., and Trujillo, O. (2014). The use and abuse of large-scale brain models. *Curr. Opin. Neurobiol.* 25, 1–6. doi: 10.1016/j.conb.2013.09.009

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2009). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2, 2008. doi: 10.3389/neuro.11.012.2008

Eschbach, C., and Zlatic, M. (2020). Useful road maps: studying Drosophila larva's central nervous system with the help of connectomics. *Curr. Opin. Neurobiol.* 65, 129–137. doi: 10.1016/j.conb.2020.09.008

Feldotto, B., Eppler, J. M., Jimenez-Romero, C., Bignamini, C., Gutierrez, C. E., Albanese, U., et al. (2022). Deploying and optimizing embodied simulations of large-scale spiking neural networks on HPC infrastructure. *Front. Neuroinform.* 16, 884180. doi: 10.3389/fninf.2022.884180

Feurer, M., and Hutter, F. (2019). *Hyperparameter Optimization*, Cham: Springer International Publishing.

Fidjeland, A. K., Roesch, E. B., Shanahan, M. P., and Luk, W. (2009). "NeMo: a platform for neural modelling of spiking neurons using GPUs," in *2009 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors* (Boston, MA: IEEE), 137–144.

Finkelstein, A., Fontolan, L., Economo, M. N., Li, N., Romani, S., and Svoboda, K. (2021). Attractor dynamics gate cortical information flow during decision-making. *Nat. Neurosci.* 24, 843–850. doi: 10.1038/s41593-021-00840-6

Florimbi, G., Torti, E., Masoli, S., D'Angelo, E., and Leporati, F. (2021). Granular layEr simulator: design and multi-gpu simulation of the cerebellar granular layer. *Front. Comput. Neurosci.* 15, 630795. doi: 10.3389/fncom.2021.630795

Gallinaro, J. V., Gašparovi,ć, N., and Rotter, S. (2022). Homeostatic control of synaptic rewiring in recurrent networks induces the formation of stable memory engrams. *PLoS Comput. Biol.* 18, e1009836 doi: 10.1371/journal.pcbi.1009836

Gewaltig, M.-O., and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia* 2, 1430. doi: 10.4249/scholarpedia.1430

Gjorgjieva, J., Drion, G., and Marder, E. (2016). Computational implications of biophysical diversity and multiple timescales in neurons and synapses for circuit performance. *Curr. Opin. Neurobiol.* 37, 44–52. doi: 10.1016/j.conb.2015.12.008

Godfrey, R. K., Swartzlander, M., and Gronenberg, W. (2021). Allometric analysis of brain cell number in Hymenoptera suggests ant brains diverge from general trends. *Proc. R. Soc. B Biol. Sci.* 288, 20210199. doi: 10.1098/rspb.2021.0199

Golosio, B., Tiddia, G., De Luca, C., Pastorelli, E., Simula, F., and Paolucci, P. S. (2021). Fast simulations of highly-connected spiking cortical models using gpus. *Front. Comput. Neurosci.* 15, 627620. doi: 10.3389/fncom.2021.627620

Gütig, R. (2016). Spiking neurons can discover predictive features by aggregate-label learning. *Science* 351, aab4113. doi: 10.1126/science.aab4113

Gütig, R., and Sompolinsky, H. (2006). The tempotron: a neuron that learns spike timing-based decisions. *Nat. Neurosci.* 9, 420–428. doi: 10.1038/nn1643

Hammond, C., Bergman, H., and Brown, P. (2007). Pathological synchronization in Parkinson's disease: networks, models and treatments. *Trends Neurosci.* 30, 357–364. doi: 10.1016/j.tins.2007.05.004

Hazan, H., Saunders, D. J., Khan, H., Patel, D., Sanghavi, D. T., Siegelmann, H. T., et al. (2018). BindsNET: a machine learning-oriented spiking neural networks library in Python. *Front. Neuroinform.* 12, 89. doi: 10.3389/fninf.2018.00089

Helgadóttir, L. I., Haenicke, J., Landgraf, T., Rojas, R., and Nawrot, M. P. (2013). "Conditioned behavior in a robot controlled by a spiking neural network," in *2013 6th International IEEE/EMBS Conference on Neural Engineering (NER)* (San Diego, CA: IEEE), 891–894.

Herculano-Houzel, S., Watson, C., and Paxinos, G. (2013). Distribution of neurons in functional areas of the mouse cerebral cortex reveals quantitatively different cortical zones. *Front. Neuroanat.* 7, 35. doi: 10.3389/fnana.2013.00035

Hines, M. L., and Carnevale, N. T. (2001). NEURON: a tool for neuroscientists. *Neuroscientist* 7, 123–135. doi: 10.1177/107385840100700207

Hinsch, K., and Zupanc, G. (2007). Generation and long-term persistence of new neurons in the adult zebrafish brain: a quantitative analysis. *Neuroscience* 146, 679–696. doi: 10.1016/j.neuroscience.2007.01.071

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci. U.S.A.* 79, 2554–2558. doi: 10.1073/pnas.79.8.2554

Igarashi, J., Yamaura, H., and Yamazaki, T. (2019). Large-scale simulation of a layered cortical sheet of spiking network model using a tile partitioning method. *Front. Neuroinform.* 13, 71. doi: 10.3389/fninf.2019.00071

Illing, B., Ventura, J., Bellec, G., and Gerstner, W. (2021). "Local plasticity rules can learn deep representations using self-supervised contrastive predictions," in *Advances in Neural Information Processing Systems, Vol. 34*, eds M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan (Curran Associates, Inc.), 30365–30379.

Inagaki, H. K., Fontolan, L., Romani, S., and Svoboda, K. (2019). Discrete attractor dynamics underlies persistent activity in the frontal cortex. *Nature* 566, 212–217. doi: 10.1038/s41586-019-0919-7

Indiveri, G., Stefanini, F., and Chicca, E. (2010). "Spike-based learning with a generalized integrate and fire silicon neuron," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems* (Paris: IEEE), 1951–1954.

Ippen, T., Eppler, J. M., Plesser, H. E., and Diesmann, M. (2017). Constructing neuronal network models in massively parallel environments. *Front. Neuroinform.* 11, 30. doi: 10.3389/fninf.2017.00030

Ivanov, D., Chezhegov, A., Grunin, A., Kiselev, M., and Larionov, D. (2022). Neuromorphic artificial intelligence systems. *Front. Neurosci.* 16, 95626. doi: 10.3389/fnins.2022.959626

Javanshir, A., Nguyen, T. T., Mahmud, M. A. P., and Kouzani, A. Z. (2022). Advancements in algorithms and neuromorphic hardware for spiking neural networks. *Neural Comput.* 34, 1289–1328. doi: 10.1162/neco_a_01499

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Front. Neuroinform.* 12, 2. doi: 10.3389/fninf.2018.00002

Kasabov, N., and Capecci, E. (2015). Spiking neural network methodology for modelling, classification and understanding of EEG spatio-temporal data measuring cognitive processes. *Inf. Sci.* 294, 565–575. doi: 10.1016/j.ins.2014.06.028

Knight, J. C., Komissarov, A., and Nowotny, T. (2021). PyGeNN: a Python library for GPU-enhanced neural networks. *Front. Neuroinform.* 15, 659005. doi: 10.3389/fninf.2021.659005

Knight, J. C., and Nowotny, T. (2018). GPUs Outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Front. Neurosci.* 12, 941. doi: 10.3389/fnins.2018.00941

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci.* 1, 136–142. doi: 10.1038/s43588-020-00022-7

Knight, J. C., and Nowotny, T. (2022). "Efficient GPU training of LSNNs using EProp," in *Neuro-Inspired Computational Elements Conference, NICE 2022* (New York, NY:. Association for Computing Machinery), 8–10.

Kulesza, R. J. (2007). Cytoarchitecture of the human superior olivary complex: medial and lateral superior olive. *Hear Res.* 225, 80–90. doi: 10.1016/j.heares.2006.12.006

Kulkarni, S. R., Parsa, M., Mitchell, J. P., and Schuman, C. D. (2021). Benchmarking the performance of neuromorphic and spiking neural network simulators. *Neurocomputing* 447, 145–160. doi: 10.1016/j.neucom.2021.03.028

Kunkel, S., Potjans, T., Eppler, J., Plesser, H. E., Morrison, A., and Diesmann, M. (2012). Meeting the memory challenges of brain-scale network simulation. *Front. Neuroinform.* 5, 35. doi: 10.3389/fninf.2011.00035

Kunkel, S., Schmidt, M., Eppler, J. M., Plesser, H. E., Masumoto, G., Igarashi, J., et al. (2014). Spiking network simulation code for petascale computers. *Front. Neuroinform.* 8, 78. doi: 10.3389/fninf.2014.00078

Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2022). Sub-realtime simulation of a neuronal network of natural density. *Neuromorphic Comput. Eng.* 2, 021001. doi: 10.1088/2634-4386/ac55fc

LaValle, S. M., Branicky, M. S., and Lindemann, S. R. (2004). On the relationship between classical grid search and probabilistic roadmaps. *Int. J. Rob. Res.* 23, 673–692. doi: 10.1177/0278364904045481

Lengler, J., Jug, F., and Steger, A. (2013). Reliable neuronal systems: the importance of heterogeneity. *PLoS ONE* 8, e80694. doi: 10.1371/journal.pone.0080694

Litwin-Kumar, A., and Doiron, B. (2012). Slow dynamics and high variability in balanced cortical networks with clustered connections. *Nat. Neurosci.* 15, 1498–1505. doi: 10.1038/nn.3220

Litwin-Kumar, A., Rosenbaum, R., and Doiron, B. (2016). Inhibitory stabilization and visual coding in cortical circuits with multiple interneuron subtypes. *J. Neurophysiol.* 115, 1399–1409. doi: 10.1152/jn.00732.2015

Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., et al. (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 456–492. doi: 10.1016/j.cell.2015.09.029

Mazzucato, L. (2022). Neural mechanisms underlying the temporal organization of naturalistic animal behavior. *arXiv.* doi: 10.7554/eLife.76577

Mazzucato, L., Fontanini, A., and La Camera, G. (2015). Dynamics of multistable states during ongoing and evoked cortical activity. *J. Neurosci.* 35, 8214–8231. doi: 10.1523/JNEUROSCI.4819-14.2015

Mazzucato, L., La Camera, G., and Fontanini, A. (2019). Expectation-induced modulation of metastable activity underlies faster coding of sensory stimuli. *Nat. Neurosci.* 22, 787–796. doi: 10.1038/s41593-019-0364-9

McIntyre, C. C., and Hahn, P. J. (2010). Network perspectives on the mechanisms of deep brain stimulation. *Neurobiol. Dis.* 38, 329–337. doi: 10.1016/j.nbd.2009.09.022

Mejias, J. F., and Longtin, A. (2012). Optimal heterogeneity for coding in spiking neural networks. *Phys. Rev. Lett.* 108, 228102. doi: 10.1103/PhysRevLett.108.228102

Mejias, J. F., and Longtin, A. (2014). Differential effects of excitatory and inhibitory heterogeneity on the gain and asynchronous state of sparse cortical networks. *Front. Comput. Neurosci.* 8, 107. doi: 10.3389/fncom.2014.00107

Menzel, R. (2012). The honeybee as a model for understanding the basis of cognition. *Nat. Rev. Neurosci.* 13, 758–768. doi: 10.1038/nrn3357

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801. doi: 10.1162/0899766054026648

Morrison, A., Straube, S., Plesser, H. E., and Diesmann, M. (2007). Exact subthreshold integration with continuous spike times in discrete-time neural network simulations. *Neural Comput.* 19, 47–79. doi: 10.1162/neco.2007.19.1.47

Mutch, J., Knoblich, U., and Poggio, T. (2010). *CNS: A GPU-Based Framework for Simulating Cortically-Organized Networks.* MIT CSAIL.

Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., and Veidenbaum, A. V. (2009). A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Netw.* 22, 791–800. doi: 10.1016/j.neunet.2009.06.028

Nawrot, M. P., Schnepel, P., Aertsen, A., and Boucsein, C. (2009). Precisely timed signal transmission in neocortical networks with reliable intermediate-range projections. *Front. Neural Circ.* 3, 2009. doi: 10.3389/neuro.04.001.2009

Neftci, E., Binas, J., Rutishauser, U., Chicca, E., Indiveri, G., and Douglas, R. J. (2013). Synthesizing cognition in neuromorphic electronic systems. *Proc. Natl. Acad. Sci. U.S.A.* 110, E3468-E3476. doi: 10.1073/pnas.1212083110

Niedermeier, L., Chen, K., Xing, J., Das, A., Kopsick, J., Scott, E., et al. (2022). "CARLsim 6: an open source library for large-scale, biologically detailed spiking neural network simulation," in *2022 International Joint Conference on Neural Networks (IJCNN)* (Padua: IEEE), 1–10.

Oorschot, D. E. (1996). Total number of neurons in the neostriatal, pallidal, subthalamic, and substantia nigral nuclei of the rat basal ganglia: a stereological study using the cavalieri and optical disector methods. *J. Compar. Neurol.* 366, 580–599. doi: 10.1002/(SICI)1096-9861(19960318)366:4andlt;580::AID-CNE3andgt;3.0.CO;2-0

Ostrau, C., Klarhorst, C., Thies, M., and Rückert, U. (2022). Benchmarking neuromorphic hardware and its energy expenditure. *Front. Neurosci.* 16, 873935. doi: 10.3389/fnins.2022.873935

Parrish-Aungst, S., Shipley, M., Erdelyi, F., Szabó, G., and Puche, A. (2007). Quantitative analysis of neuronal diversity in the mouse olfactory bulb. *J. Comp. Neurol.* 501, 825–836. doi: 10.1002/cne.21205

Parsa, M., Mitchell, J. P., Schuman, C. D., Patton, R. M., Potok, T. E., and Roy, K. (2019). "Bayesian-based hyperparameter optimization for spiking neuromorphic systems," in *2019 IEEE International Conference on Big Data (Big Data)* (Los Angeles, CA: IEEE), 4472–4478.

Pfeiffer, M., and Pfeil, T. (2018). Deep learning with spiking neurons: opportunities and challenges. *Front. Neurosci.* 12, 774. doi: 10.3389/fnins.2018.00774

Plotnikov, D., Rumpe, B., Blundell, I., Ippen, T., Eppler, J. M., and Morrison, A. (2016). "NESTML: a modeling language for spiking neurons," in *Modellierung 2016*, eds S. Betz and U. Reimer (Karlsruhe).

Potjans, T. C., and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Pronold, J., Jordan, J., Wylie, B. J. N., Kitayama, I., Diesmann, M., and Kunkel, S. (2022). Routing brain traffic through the von Neumann bottleneck: efficient cache usage in spiking neural network simulation code on general purpose computers. *Parallel Comput.* 113, 102952. doi: 10.1016/j.parco.2022.102952

Raji, J. I., and Potter, C. J. (2021). The number of neurons in Drosophila and mosquito brains. *PLoS ONE* 16, 1–11. doi: 10.1371/journal.pone.0250381

Rapp, H., and Nawrot, M. P. (2020). A spiking neural program for sensorimotor control during foraging in flying insects. *Proc. Natl. Acad. Sci. U.S.A.* 117, 28412–28421. doi: 10.1073/pnas.2009821117

Rapp, H., Nawrot, M. P., and Stern, M. (2020). Numerical cognition based on precise counting with a single spiking neuron. *iScience* 23, 100852. doi: 10.1016/j.isci.2020.101283

Rost, T. (2016). *Modelling Cortical Variability Dynamics*. (Ph.D. thesis). Freie Universität Berlin.

Rost, T., Deger, M., and Nawrot, M. P. (2018). Winnerless competition in clustered balanced networks: inhibitory assemblies do the trick. *Biol. Cybern.* 112, 81–98. doi: 10.1007/s00422-017-0737-7

Rostami, V., Rost, T., Riehle, A., van Albada, S. J., and Nawrot, M. P. (2022). Excitatory and inhibitory motor cortical clusters account for balance, variability, and task performance. *bioRxiv*. doi: 10.1101/2020.02.27.968339

Rotter, S., and Diesmann, M. (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biol. Cybern.* 81, 381–402. doi: 10.1007/s004220050570

Sacramento, J. A., Ponte Costa, R., Bengio, Y., and Senn, W. (2018). "Dendritic cortical microcircuits approximate the backpropagation algorithm," in *Advances in Neural Information Processing Systems, Vol. 31*, eds S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Montréal, QC: Curran Associates), 8721–8732. Available online at: https://proceedings.neurips.cc/paper/2018/file/1dc3a89d0d440ba31729b0ba74b93a33-Paper.pdf

Sakagiannis, P., Jürgensen, A.-M., and Nawrot, M. P. (2021). A realistic locomotory model of drosophila larva for behavioral simulations. *bioRxiv*. doi: 10.1101/2021.07.07.451470

Sakai, K., and Miyashita, Y. (1991). Neural organization for the long-term memory of paired associates. *Nature* 354, 152–155. doi: 10.1038/354152a0

Sarko, D. K., Catania, K., Leitch, D. B., Kaas, J. H., and Herculano-Houzel, S. (2009). Cellular scaling rules of insectivore brains. *Front. Neuroanat.* 3, 2009. doi: 10.3389/neuro.05.008.2009

Schmidt, M., Bakker, R., Shen, K., Bezgin, G., Diesmann, M., and van Albada, S. J. (2018). A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas. *PLoS Comput. Biol.* 14, e1006359. doi: 10.1371/journal.pcbi.1006359

Schmuker, M., Pfeil, T., and Nawrot, M. P. (2014). A neuromorphic network for generic multivariate data classification. *Proc. Natl. Acad. Sci. U.S.A.* 111, 2081–2086. doi: 10.1073/pnas.1303053111

Schuman, C. D., Kulkarni, S. R., Parsa, M., Mitchell, J. P., Date, P., and Kay, B. (2022). Opportunities for neuromorphic computing algorithms and applications. *Nat. Comput. Sci.* 2, 10–19. doi: 10.1038/s43588-021-00184-y

Sherwood, C. C., Miller, S. B., Karl, M., Stimpson, C. D., Phillips, K. A., Jacobs, B., et al. (2020). Invariant synapse density and neuronal connectivity scaling in primate neocortical evolution. *Cereb. Cortex* 30, 5604–5615. doi: 10.1093/cercor/bhaa149

Singer, W., and Gray, C. M. (1995). Visual feature integration and the temporal correlation hypothesis. *Annu. Rev. Neurosci.* 18, 555–586. doi: 10.1146/annurev.ne.18.030195.003011

Steffen, L., Koch, R., Ulbrich, S., Nitzsche, S., Roennau, A., and Dillmann, R. (2021). Benchmarking highly parallel hardware for spiking neural networks in robotics. *Front. Neurosci.* 15, 667011. doi: 10.3389/fnins.2021.667011

Stimberg, M., Brette, R., and Goodman, D. F. M. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8, e47314. doi: 10.7554/eLife.47314.028

Tanaka, G., Yamane, T., Héroux, J. B., Nakane, R., Kanazawa, N., Takeda, S., et al. (2019). Recent advances in physical reservoir computing: a review. *Neural Networks* 115, 100–123. doi: 10.1016/j.neunet.2019.03.005

Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., and Maida, A. (2019). Deep learning in spiking neural networks. *Neural Networks* 111, 47–63. doi: 10.1016/j.neunet.2018.12.002

Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed computing in practice: the Condor experience. *Concurrency Pract. Exp.* 17, 323–356. doi: 10.1002/cpe.938

Thörnig, P. (2021). JURECA: data centric and booster modules implementing the modular supercomputing architecture at jülich supercomputing centre. *J. Largescale Res. Facilit.* 7, A182. doi: 10.17815/jlsrf-7-182

Tiddia, G., Golosio, B., Albers, J., Senk, J., Simula, F., Pronold, J., et al. (2022). Fast simulation of a multi-area spiking network model of macaque cortex on an MPI-GPU cluster. *Front. Neuroinform.* 16, 883333. doi: 10.3389/fninf.2022.883333

Tikidji-Hamburyan, R. A., Narayana, V., Bozkus, Z., and El-Ghazawi, T. A. (2017). Software for brain network simulations: a comparative study. *Front. Neuroinform.* 11, 46. doi: 10.3389/fninf.2017.00046

Tripathy, S. J., Padmanabhan, K., Gerkin, R. C., and Urban, N. N. (2013). Intermediate intrinsic diversity enhances neural population coding. *Proc. Natl. Acad. Sci. U.S.A.* 110, 8248–8253. doi: 10.1073/pnas.1221214110

Van Albada, S. J., Helias, M., and Diesmann, M. (2015). Scalability of asynchronous networks is limited by one-to-one mapping between effective connectivity and correlations. *PLoS Comput. Biol.* 11, e1004490. doi: 10.1371/journal.pcbi.1004490

Van Vreeswijk, C., and Sompolinsky, H. (1996). Chaos in neuronal networks with balanced excitatory and inhibitory activity. *Science* 274, 1724–1726. doi: 10.1126/science.274.5293.1724

Vitay, J., Dinkelbach, H. O., and Hamker, F. H. (2015). ANNarchy: a code generation approach to neural simulations on parallel hardware. *Front. Neuroinform.* 9, 19. doi: 10.3389/fninf.2015.00019

Vlag, M. A., v. d., Smaragdos, G., Al-Ars, Z., and Strydis, C. (2019). Exploring complex brain-simulation workloads on multi-GPU deployments. *ACM Trans. Arch. Code Optim.* 16, 1–25. doi: 10.1145/3371235

Vogels, T. P., Sprekeler, H., Zenke, F., Clopath, C., and Gerstner, W. (2011). Inhibitory plasticity balances excitation and inhibition in sensory pathways and memory networks. *Science* 334, 1569–1573. doi: 10.1126/science.1211095

Von, S.t., and Vieth, B. (2021). JUSUF: Modular Tier-2 supercomputing and cloud infrastructure at jülich supercomputing centre. *J. Largescale Res. Facilit.* 7, A179. doi: 10.17815/jlsrf-7-179

White, J. G., Southgate, E., Thomson, J. N., and Brenner, S. (1986). The structure of the nervous system of the nematode Caenorhabditis elegans. *Philos. Trans. R. Soc. Londo. B Biol. Sci.* 314, 1–340. doi: 10.1098/rstb.1986.0056

Witthöft, W. (1967). Absolute Anzahl und Verteilung der Zellen im Hirn der Honigbiene. *Zeitschrift für Morphologie der Tiere* 61, 160–184. doi: 10.1007/BF00298776

Wyrick, D., and Mazzucato, L. (2021). State-dependent regulation of cortical processing speed *via* gain modulation. *J. Neurosci.* 41, 3988–4005. doi: 10.1523/JNEUROSCI.1895-20.2021

Yamaura, H., Igarashi, J., and Yamazaki, T. (2020). Simulation of a human-scale cerebellar network model on the K computer. *Front. Neuroinform.* 14, 16. doi: 10.3389/fninf.2020.00016

Yavuz, E., Turner, J., and Nowotny, T. (2016). GeNN: a code generation framework for accelerated brain simulations. *Sci. Rep.* 6, 1–14. doi: 10.1038/srep18854

Yegenoglu, A., Subramoney, A., Hater, T., Jimenez-Romero, C., Klijn, W., Pérez Martín, A., et al. (2022). Exploring parameter and hyper-parameter spaces of neuroscience models on high performance computers with learning to learn. *Front. Comput. Neurosci.* 16, 885207. doi: 10.3389/fncom.2022.885207

Zenke, F., and Ganguli, S. (2018). SuperSpike: supervised learning in multilayer spiking neural networks. *Neural Comput.* 30, 1514–1541. doi: 10.1162/neco_a_01086

Zhao, Y., Wang, D. O., and Martin, K. C. (2009). Preparation of aplysia sensory-motor neuronal cell cultures. *J. Vis. Exp.* 8, 1355. doi: 10.3791/1355-v

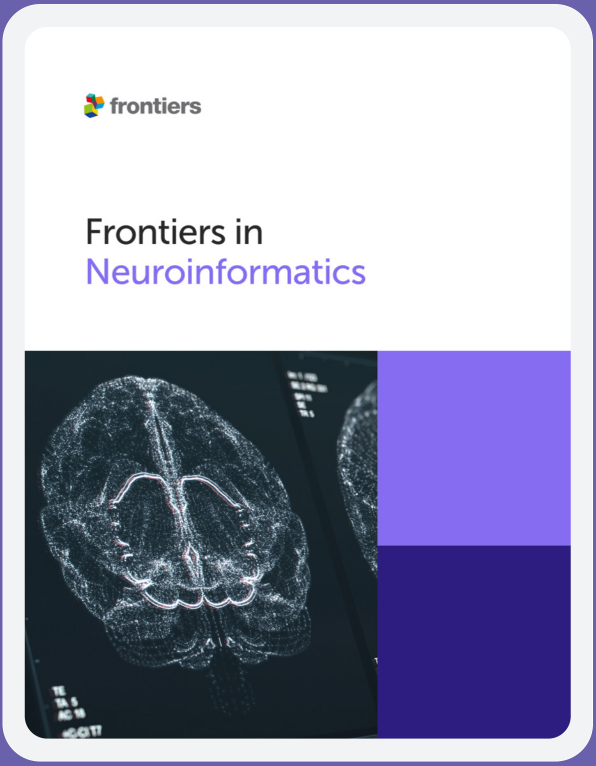
Frontiers in Neuroinformatics

Leading journal supporting neuroscience in the information age

Part of the most cited neuroscience journal series, developing computational models and analytical tools used to share, integrate and analyze experimental data about the nervous system functions.

Discover the latest Research Topics

See more →



frontiers

Frontiers in Neuroinformatics

Frontiers

Avenue du Tribunal-Fédéral 34
1005 Lausanne, Switzerland
frontiersin.org

Contact us

+41 (0)21 510 17 00
frontiersin.org/about/contact