Check for updates

Exploring code portability solutions for HEP with a particle tracking test code

Hammad Ather[1], Sophie Berkman[2], Giuseppe Cerati[3]*,
Matti J. Kortelainen[3], Ka Hei Martin Kwok[3], Steven Lantz[4],
Seyong Lee[5], Boyana Norris[1], Michael Reid[4],
Allison Reinsvold Hall[6]*, Daniel Riley[4], Alexei Strelchenko[3] and
Cong Wang[3,7]

[1]Department of Computer and Information Science, University of Oregon, Eugene, OR, United States,
[2]Department of Physics and Astronomy, Michigan State University, East Lansing, MI, United States,
[3]Computational Science and AI Directorate, Fermi National Accelerator Laboratory, Batavia, IL,
United States, [4]Department of Physics, Cornell University, Ithaca, NY, United States, [5]Computer
Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, United States,
[6]Physics Department, United States Naval Academy, Annapolis, MD, United States, [7]School of
Computing, Clemson University, Clemson, SC, United States

Traditionally, high energy physics (HEP) experiments have relied on x86 CPUs for
the majority of their significant computing needs. As the field looks ahead to the
next generation of experiments such as DUNE and the High-Luminosity LHC,
the computing demands are expected to increase dramatically. To cope with
this increase, it will be necessary to take advantage of all available computing
resources, including GPUs from different vendors. A broad landscape of code
portability tools—including compiler pragma-based approaches, abstraction
libraries, and other tools—allow the same source code to run efficiently on
multiple architectures. In this paper, we use a test code taken from a HEP
tracking algorithm to compare the performance and experience of implementing
different portability solutions. While in several cases portable implementations
perform close to the reference code version, we find that the performance varies
significantly depending on the details of the implementation. Achieving optimal
performance is not easy, even for relatively simple applications such as the test
codes considered in this work. Several factors can affect the performance, such
as the choice of the memory layout, the memory pinning strategy, and the
compiler used. The compilers and tools are being actively developed, so future
developments may be critical for their deployment in HEP experiments.

1 Introduction

Modern high energy physics (HEP) experiments have to process enormous volumes of
data in their search to probe extremely rare interactions between fundamental particles.
The Compact Muon Solenoid (CMS) experiment (CMS Collaboration, 2008) at the
CERN Large Hadron Collider (LHC), for example, processed hundreds of petabytes of
detector data and Monte Carlo (MC) simulations during Run 2 (2015–2018) of the
LHC (CMS Offline Software and Computing, 2021, 2022). Within the next decade,
HEP experiments such as the High-Luminosity LHC (HL-LHC) (Apollinari et al.,
2015) at CERN and the Deep Underground Neutrino Experiment (DUNE) (DUNE
Collaboration, 2020) at Fermilab will pose significant additional computing challenges.
The event rate at the LHC is expected to increase by a factor of 7.5, and the data

volumes will grow to exabyte scale. Likewise, the expected data rate of a DUNE far-detector module is 9.4 PB per year, and so the total tape volume is expected to exceed the exabyte scale by 2040 (DUNE Collaboration, 2022). To handle these data volumes without sacrificing the physics potential of each experiment, significant R&D–and accompanying shifts in traditional HEP computing paradigms–are required.

One paradigm shift that will help HEP experiments prepare for upcoming computing challenges is the ability to utilize parallel heterogeneous computing resources. Historically, the LHC experiments have relied on traditional x86 CPUs for the vast majority of offline computing needs. The majority of the data processing capabilities for the LHC experiments are provided by the Worldwide LHC Computing Grid (WLCG) (Bird, 2011), which connects 170 computing centers in over 40 countries. Increasingly, however, experiments are adapting their software frameworks to take advantage of computing resources at High Performance Computing (HPC) centers (Megino et al., 2023; DUNE Collaboration, 2022). All planned exascale platforms rely heavily on GPUs to achieve their anticipated compute performance, and HEP workflows will need to run on GPUs in order to efficiently utilize these resources (Albrecht and et al., 2019).

Adapting HEP algorithms to run on GPUs is not a trivial task. For example, CMSSW (Jones et al., 2006), the CMS software framework, includes almost 8 million lines of code (CMS Offline Software and Computing, 2021) and was written by hundreds of scientists with varying software backgrounds over the course of decades. Additionally, it is not clear what compute architectures will be prevalent in HPC centers or international scientific computing grids in a decade or two, when the HL-LHC and DUNE experiments are collecting and analyzing data. Even the planned exascale machines in the US use a variety of architectures: Aurora at Argonne National Laboratory uses CPUs and GPUs from Intel, while Frontier at Oak Ridge National Laboratory and El Capitan at Lawrence Livermore Laboratory rely on CPUs and GPUs from AMD. Initial attempts to port HEP algorithms to GPUs typically involved rewriting the original C++ code using CUDA. This process is labor-intensive and only enables offloading to NVIDIA GPUs. Moreover, significant efforts are required to optimize the performance of the initial implementations. The HIP programming language is very similar to CUDA and supports both NVIDIA and AMD GPUs (with very early support for Intel GPUs as well), but neither HIP nor CUDA support CPU architectures directly. Writing and maintaining different implementations for every individual computing platform would take much more expertise and personpower than any HEP experiment can provide.

This is a widely recognized challenge in scientific computing, and there is a broad, rapidly changing landscape of portability solutions that allow a single source code to be compiled and run on a variety of computing backends. The available portability solutions vary widely in terms of overall approach, performance, maturity, and support for different backends or compilers. It is clear, however, that taking advantage of these portability tools will be an essential part of modernizing HEP software. Ideally, a portability solution would achieve two important goals. First, the portability solution should enable straightforward adaptations of existing HEP algorithms, with minimal rewriting and optimization required. Second, the tool should enable algorithms to run efficiently on a variety of different computing architectures, including both CPU and GPU platforms from different manufacturers. The performance on different architectures needs to be reasonable, on the same order of magnitude, although it is unlikely to match that of a fully optimized native implementation. In this paper, we used a standalone benchmark algorithm to test different code portability solutions and evaluated each in terms of its computational performance and subjective ease of use.

Programming models and C++ libraries such as Kokkos (Edwards et al., 2014; Trott et al., 2022) and Alpaka (Matthes et al., 2017) provide high level data structures and parallel execution options that can be adapted to produce optimized code for a variety of backends, including CPUs and GPUs from NVIDIA, AMD, or Intel (preliminary). Another portability solution is the std::execution::par (stdpar) interface, which has been included in the C++ standard since C++ 17. The application programming interface (API) allows for a high level description of concurrent loops, but does not allow for low level optimizations that can be used to enhance performance in native CUDA or HIP. Various C++ compilers and associated libraries—such as the oneAPI DPC++/C++ Compiler (dpcpp) from Intel, and nvc++ from NVIDIA—provide support for offloading loops to GPUs, but these compilers are still relatively new. Similarly, SYCL is a programming model based on the ISO C++17 standard that enables host and kernel (device) code to be included in the same source file. Finally, there is a category of directive-based portability solutions, which includes OpenMP and OpenACC: through the use of pragmas, developers can specify high level parallelization and memory management behaviors, with the compilers managing the low level optimizations. It should be noted that all these portability solutions, though they are based on open specifications and open-source libraries, generally rely on proprietary, vendor-supplied software stacks (and often compilers) in order to run on particular GPUs.

This paper is organized as follows: In Section 2, we describe the motivation and context for this work. In Section 3, the benchmark algorithm is described in more detail. The different implementations are covered in Section 4, including technical details and a subjective discussion of the experience porting the algorithm to each tool. Compute performance results are shown in Section 5, and Section 6 provides an overall discussion of our experience and lessons learned.

2 Background and related work

There are several processing steps involved in analyzing data from a HEP experiment. For example, analyzing a time window at the LHC that contains at least one proton-proton collision (referred to as an "event"), includes the initial data acquisition, "reconstruction" of the raw detector data into higher level information including what particles were observed and their energies, and the final data analysis and statistical interpretations. This process is similar for other HEP experiments. In CMS and ATLAS (ATLAS Collaboration, 2008), the most computationally expensive reconstruction step is track finding, which is the combinatorial process of reconstructing the trajectories of charged particles from the energy deposits ("hits") they leave in different layers of the detector (see CMS Collaboration, 2014 for a full description from CMS). The benchmarks used for the results in

this paper represent the propagation and Kalman update steps (described below) of a traditional Kalman Filter (KF) tracking algorithm (Fruhwirth, 1987). There are two test codes, referred to as the "propagate to z" or "propagate to r" benchmarks, denoted by `p2z` and `p2r`, respectively. These compact programs ("mini apps"), although including functions extracted from the CMS tracking code, are standalone applications that can be more easily replicated, profiled, and optimized for the purpose of this work.

The work described in this paper builds off efforts by several other groups working to modernize HEP reconstruction algorithms. The `p2z` and `p2r` benchmarks are part of a larger algorithm development effort known as MKFIT (Lantz et al., 2020). The goal of the MKFIT project is to rewrite the traditional KF tracking algorithms used by most major HEP experiments and develop a new CPU implementation that is efficient, vectorized, and multithreaded. Depending on the compiler, the MKFIT algorithm achieves up to a factor of six speedup compared to previous KF tracking implementations, and it is now the default algorithm used to reconstruct the majority of tracks in the CMS experiment (Cerati et al., 2023). The key insight of the MKFIT project is that the KF calculations can be parallelized over the thousands of tracks that may be present within a single detector event. Moreover, if the small matrices and vectors holding the data for each track are arranged in memory so that matching elements from different tracks are stored in adjacent locations, then vector or SIMD (Single Instruction, Multiple Data) operations can be used to perform the KF calculations. Similar efforts have also been effective at speeding up code for Liquid Argon Time Projection Chamber (LArTPC) neutrino experiments (Berkman et al., 2022).

The MKFIT effort has so far targeted optimizations for Intel multicore CPU architectures such as the Intel Xeon and Intel Xeon Phi processors and coprocessors, but efficient implementations for other architectures will become increasingly important, especially during the HL-LHC era. Given that MKFIT was explicitly designed to create opportunities for vector or SIMD operations, it seems that GPUs should also make a suitable target platform for the MKFIT approach to parallelizing Kalman filtering. However, initial attempts to port MKFIT to the NVIDIA Kepler GPU (K40) using CUDA were not very encouraging, both in terms of difficulty and in terms of observed performance [for a full discussion, see Section 4 of Cerati et al. (2017)]. The irregular patterns of memory access that are occasionally needed in order to reorganize the data coming from different tracks turned out to be particularly challenging to manage on GPUs. Even with well-structured data, however, translating standard C++ code to be compatible with NVIDIA CUDA required significant low level re-coding effort to achieve acceptable performance for the basic KF operations. Since it is not feasible to rewrite MKFIT for every possible architecture, the `p2z` project was started to explore code portability tools in the context of charged particle tracking.

A broader effort with similar motivation is the HEP Computational Center for Excellence (HEP-CCE) collaboration's Portable Parallelization Strategies (PPS) activity (Bhattacharya et al., 2022; Atif et al., 2023). The HEP-CCE PPS project is exploring portability solutions using representative reconstruction algorithms from CMS and ATLAS as well as LArTPC neutrino experiments such as DUNE. Collaborators from the HEP-CCE PPS

activity became involved in this project and have used the `p2r` mini app to evaluate GPU offloading via different technologies, such as CUDA, OpenACC, and stdpar, as described in detail below.

Having two teams working simultaneously on these two complementary mini apps has proven to be important to the project's success. For many of the implementations described in Section 4, we found that it was relatively straightforward to do an initial porting of the algorithm but fairly difficult to have a fully optimized version. Different initial strategies in porting the `p2z` or `p2r` benchmark meant that multiple approaches could be simultaneously developed and tested. In several cases, an issue was identified in a specific `p2z` or `p2r` implementation and the solution was propagated to both mini apps. Having two different teams with unique expertise also expanded the number of portability technologies we could test. Finally, as explained in Section 5, slightly different approaches were taken to measure the final results, giving additional insight into the performance of each tool.

3 Description of algorithm

Track finding (also known as track building) is the process of reconstructing a particle's trajectory by identifying which hits in an event likely came from the same particle. It requires testing many potential combinations of hits to find a set that is consistent with the expected helical trajectory of a charged particle in a magnetic field. Track fitting, on the other hand, is the process of taking a pre-determined set of hits and determining the final parameters of the track. The `p2z` and `p2r` benchmarks include everything that would be needed for a realistic track *fitting* algorithm, but do not include the combinatorial selection required for track finding.

The CMS and ATLAS detectors are divided into two main sections: the cylindrical region coaxial with the beam pipe, known as the barrel, and a disk region on either end of the barrel, known as the endcaps. To first approximation, the individual tracker layers can be approximated as being located at constant radius r or constant z position for the barrel and endcap layers, respectively. Charged particles in a constant magnetic field will travel in a helix, so if the position and momentum are known on layer N, then the expected position can be calculated for layer $N + 1$.

The mini apps used in this analysis perform two key steps of KF tracking:

1. **Track propagation:** Propagate the track state—including the track's momentum and position vectors and associated uncertainties in the form of a covariance matrix—at layer N to a prediction at layer $N + 1$, which is specified by either a z coordinate or a radius r for the `p2z` and `p2r` benchmarks, respectively.
2. **Kalman update:** Update the track state on layer $N + 1$ by combining information about the propagated track state and the coordinates of a compatible hit on that layer.

These two steps are the most arithmetically intensive steps of both track finding and track fitting. They are relatively simple algorithms but have the biggest impact on the overall execution time, making them suitable for a standalone test code.

3.1 Input data

The starting point for a KF track finding algorithm is a track "seed", an initial guess at the track state. For the full MKFIT algorithm, the input track seeds are built in an upstream algorithm using three or four hits from the innermost layers of the CMS detector. For simplicity, the p2z and p2r benchmarks use an artificial standalone input consisting of a single track seed. The parameters of the initial seed are smeared according to a Gaussian distribution, in order to prevent the algorithm from performing identical numerical operations for each track.

The tracks are built by first propagating the initial seed parameters to the next layer. In the full combinatorial MKFIT application, the next step is to search for compatible hits on that layer. In the p2z and p2r benchmarks, only one hit per layer is considered, similar to what is required for track fitting. The hit parameters—the hit locations and uncertainties on each layer—are smeared per track according to the same procedure as the input track parameters.

The total number of tracks generated with this procedure is split into "events" with a fixed number of tracks per event.
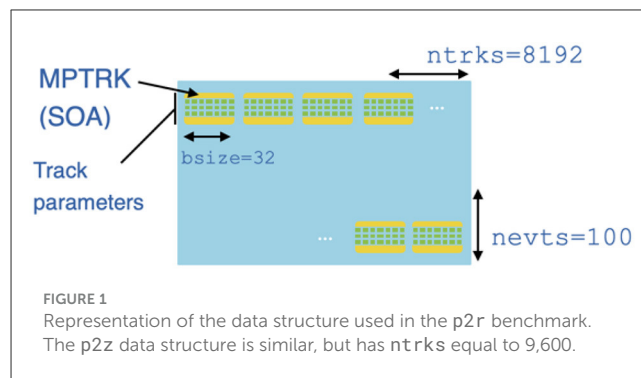
For the GPU implementations of the benchmark, the track propagation and Kalman update steps are run as a single GPU kernel. The data for all tracks and hits are prepared on the CPU, transferred to the GPU for the computations in two bulk transfers, and the output data are transferred back to the CPU.

3.2 Computations and data structure

The "propagation to z" or p2z benchmark uses the expected helical trajectory of the charged particle to calculate the track parameters and the covariance matrix on endcap layer $N + 1$. In contrast, the "propagation to r" or p2r benchmark uses an iterative approach to propagation, advancing the track state from the initial barrel radius to the final radius in discrete steps. In practice, both approaches involve a series of operations involving sin() and cos() functions, as well as matrix multiplication of up to 6x6 matrices. The two benchmarks are expected to be similar in terms of arithmetic intensity.

The second task in both benchmarks is the Kalman update step, which "updates" the track state using the parameters of a hit on that layer. The hit parameters include three coordinates for the hit position and a 3x3 covariance matrix. Similar to the propagation step, the update step involves small matrix multiplication and matrix inverse operations as well as trigonometric functions.

Both benchmarks employ an Array-Of-Structures-Of-Arrays (AOSOA) data structure, as illustrated in Figure 1. Each benchmark runs over a fixed number of events (nevts) and processes a fixed number of tracks in each event (ntrks). Within one event, tracks are grouped into batches of bsize tracks, and each batch of tracks is put into a Structure-Of-Arrays construct referred to as an MPTRK. Here, as in the full MKFIT algorithm, the goal of organizing tracks into different batches is to enable SIMD operations across batch elements. The value of bsize can be optimized for different platforms; for example, on a GPU it might be the NVIDIA warp size of 32, while on a CPU it might be a multiple of the AVX-512 vector width of 16; for consistency we use 32 everywhere.

FIGURE 1
Representation of the data structure used in the p2r benchmark. The p2z data structure is similar, but has ntrks equal to 9,600.

To maximize the opportunities for SIMD operations, the storage order in the AOSOA follows the same general scheme as in MKFIT. Within an MPTRK, the first elements of bsize different vectors (or matrices) get stored in adjacent locations, followed by the second elements of bsize different vectors (or matrices), and so on, until the full structure representing a batch of bsize tracks is completely populated. Then the next MPTRK structure is written into memory, and the next, until all ntrks tracks (8192 for p2r and 9600 for p2z) for the first event are present in memory. This first event corresponds to the first row of SOAs in Figure 1. It is followed by a second row of SOAs for the second event, etc. A similar memory layout applies to the hit data.

4 Implementations

The state-of-the-art of portability tools is a moving target, as many tools are undergoing very active development, with additional features, backends, and compiler support being added on a regular basis ($\mathcal{O}$(monthly)). In total, we tested nine different parallelization tools on four different architectures, but testing the full matrix of possibilities was beyond the scope of this paper. The final set of p2z implementations is shown in Table 1, including which backends and compilers were used to test each implementation, and the full set of p2r implementations is shown in Table 2. Note that this does not include the full set of backends or compilers that each tool is capable of supporting.

For both test codes, the original CPU implementation was multithreaded using the oneAPI Threading Building Blocks (TBB) library and compiled with gcc, since this is the combination that most closely matches what the MKFIT project uses for its highly optimized implementation. The reference GPU implementation is the one written using CUDA and compiled with nvcc. Below we describe each tool, its corresponding p2r and p2z implementations, and our subjective experience porting these benchmarks using the different portability solutions.

4.1 TBB

Our reference CPU implementation is based on the oneAPI Threading Building Blocks (or oneTBB, or just TBB) library. TBB is a template library originated by Intel for parallel programming on multi-core processors that simplifies the definition and

optimization of threaded applications. For a given executable, TBB collects all tasks that are specified to run in parallel and the library manages and schedules threads to execute them. TBB is used as the CPU thread manager in the software framework of the CMS experiment.

In our code, nested TBB `parallel_for` loops are over events and batches of tracks. Each batch of tracks is then vectorized, so that tracks in the batch are processed in a SIMD fashion. Vectorization is implemented following the approach used in MKFIT, where groups of matrices are processed concurrently and loops over the matrix index in the group are decorated with `omp`

`simd` pragmas. These pragmas were activated by the `-fopenmp` compiler option.

4.2 CUDA and HIP

Our reference GPU implementation is based on the CUDA programming model, which is a multi-threaded SIMD model for general purpose GPU programming, introduced by NVIDIA. The CUDA implementation is ported from the TBB version, which shares the same AOSOA data structure for input data. The main difference in CUDA is that each `MPTRK` is processed by a block of GPU threads, and each thread processes the computation for one track in the `MPTRK`. Since the computation of each track is independent of the others, we find keeping the intermediate results in the local registers to have the most efficient memory access. We explored using shared memory to store the intermediate results within an `MPTRK` for all the threads in the block, but it was shown to have significantly lower memory throughput in a detailed profiling study.

In relation to other portability technologies, CUDA provides a level of abstraction similar to the general accelerator execution model and memory model that is also employed by general GPU programming models such as OpenCL and SYCL. As a proprietary NVIDIA GPU programming model, however, it exposes several NVIDIA-GPU-specific features, which allows available architecture-specific features to be fully exploited, but it also means that the code is not portable across heterogeneous accelerators.

HIP is the vendor-supported, native programming model for AMD GPUs and is designed to be portable across NVIDIA and AMD GPUs. It is also designed to be syntactically similar to CUDA so that most API calls can be simple translations of names. In the case of `p2z` and `p2r`, the kernels only rely on

TABLE 1 Summary of `p2z` implementations used for the results shown in Section 5.

	NVIDIA GPU V100	x86 CPU Xeon gold
TBB (oneTBB/2021.10.0)	-	gcc/12.3.0
CUDA	cuda/11.2 nvcc/11.0.221	-
Alpaka (v0.8.0)	cuda/11.2 nvcc/11.0.221	Alpaka v0.9.0 oneTBB/2021.10.20 gcc/12.3.0
Kokkos (v4.0)	cuda/11.2 nvcc/11.0.221	gcc/12.3.0
stdpar	nvc++/24.5	nvc++/24.5
OpenMPv4	OpenARC/0.76 (Lee and Vetter, 2014) nvcc/11.0.221	gcc/12.3.0
OpenACC	OpenARC/0.76 (Lee and Vetter, 2014) nvcc/11.0.221	-
Results shown in:	Figure 2, top	Figure 7, top

The table lists the compiler versions used for each implementation.

TABLE 2 Summary of `p2r` implementations used for the results shown in Section 5.

	NVIDIA GPU A100	AMD GPU MI100	Intel GPU A770	x86 CPU Xeon Gold
TBB (oneTBB/2021.10.0)	-	-	-	gcc/12.3.0
CUDA	cuda/11.6.2 nvcc/11.6.124	-	-	-
HIP	-	rocm/5.2.0	-	-
Alpaka (v0.9.0)	cuda/11.6.2 nvcc/11.6.124	rocm/5.1.3 hipcc/3.5.0	-	oneTBB/2021.10.0 gcc/12.3.0
Kokkos (v3.6.1)	cuda/11.6.2 nvcc/11.6.124	rocm/5.1.3 hipcc/3.5.0	Kokkos 4.0 dpcpp/2023.0.0	Kokkos 4.0.0 gcc/12.3.0
SYCL	cuda/11.6.2 intel/llvm-sycl	rocm/5.1.3 intel/llvm-sycl	dpcpp/2023.0.0	dpcpp/2023.1.0
stdpar	nvc++/22.7	-	dpcpp/2023.0.0 dpl/2022.0.0	nvc++/24.5
Results shown in:	Figure 2, bottom	Figure 6, top	Figure 6, bottom	Figure 7, bottom

The table lists the compiler versions used for each implementation. For SYCL, the intel/llvm-sycl repository is used to compile dpc++ tool chains to be compatible with NVIDIA/AMD GPUs. This is not the full list of all possible combinations of tools and backends. For example, HIP could be compiled for NVIDIA GPUs as well, but that was not tested in this work. Similarly, we note that the oneAPI toolkit recently included a plugin that enables stdpar to be used on AMD GPUs, and Alpaka has recently introduced experimental support for Intel GPUs in v0.9.0, but neither was tested here.

the core functionalities of CUDA, such as memory allocation and kernel dispatches, which are supported in HIP. This leads to a straightforward port to the HIP version starting from the CUDA version.

4.3 Directive-based solutions, OpenMP and OpenACC

Directive-based, high-level programming models such as OpenMP and OpenACC use a set of directives (a special type of comments that a compiler can understand) that allow a programmer to provide the compilers with important characteristics of an application, such as available parallelism and data sharing/mapping rules, so that much of the low-level programming and optimization burdens are automatically handled by the directive compilers.

The biggest advantage of the directive programming model is that it allows incremental porting of existing sequential CPU applications written in standard programming languages such as C, C++, and Fortran to parallel versions that can offload work to heterogeneous accelerators, without requiring major changes in the existing code structures. The initial OpenMP version was created by converting the reference TBB CPU implementation into an OpenMP CPU implementation, which is relatively straightforward due to the similarity between the TBB parallel syntax (using lambdas) and OpenMP parallel constructs. Then, the GPU offloading version was created by extending the OpenMP CPU implementation with OpenMP target-offloading directives and data mapping directives. The CPU version and the GPU version have different parallelism and data mapping strategies. For instance, on GPUs, the OpenMP target and data mapping directives are essential, but when targeting CPUs, they are unnecessary since the original host data and corresponding device data will share storage; in the latter case, how the OpenMP compiler implements those unnecessary directives on the CPUs is implementation-defined. Another issue is that on CPUs, team-level parallelism may or may not be ignored depending on the compiler implementations, possibly resulting in additional overheads due to loop serialization.

The initial conversion from the OpenMP GPU implementation into the OpenACC implementation was straightforward since both OpenMP and OpenACC are directive-based accelerator programming models and provide very similar execution and memory models. However, the main issue in converting between OpenMP and OpenACC was that different OpenMP/OpenACC compilers may choose different parallelism mapping strategies and vary in terms of maturity and supported features (see Section 5.1.1 for results and a more detailed discussion).

4.4 Alpaka

Alpaka (Matthes et al., 2017) is a single-source, header-only C++ parallelization library. The API level is similar to CUDA, with an abstraction layer added between the application and the vendor-specific programming models to achieve portability.

For example, the kernel functions in Alpaka are templated with an accelerator type, which is resolved at compile time for different execution backends of the kernel. One difference between Alpaka and CUDA is that Alpaka has an additional abstraction level called *elements* in the parallel hierarchy model, where multiple elements can be processed in a thread. Having an explicit level allows compilers to take advantage of the SIMD vector registers when compiling for the CPU backends. Each SOA (MPTRK) is processed by a block, and the thread/element level is mapped differently between CPU and GPU backends to take full advantage of the parallel hierarchy. For GPU backends, blocks of bsize threads are assigned to process each MPTRK, whereas one thread with bsize elements is assigned to process each MPTRK for CPU backends. In each case, enough threads are defined so that all ntrks tracks are processed for an event. Profiling results confirm the use of vectorized instructions in the regions where the original MKFIT implementation was also able to vectorize.

The overall conversion from CUDA to Alpaka is relatively smooth due to the similarity between the two programming models, except that the heavy use of templating often leads to a more verbose coding style and convoluted error messages during debugging. Nevertheless, Alpaka versions are often able to produce close-to-native performance after some effort of optimization. A particular relevant note for HEP experiment is that CMS has chosen to use Alpaka as its supported portability solution for the GPU usage in LHC Run 3 (Kortelainen et al., 2021; Bocci et al., 2023).

4.5 Kokkos

Similar to Alpaka, Kokkos (Edwards et al., 2014; Trott et al., 2022) serves as a single-source C++ template metaprogramming (TMP) library, intended to achieve architecture agnosticism and alleviate programmers from the complexities of vendor- or target-specific programming paradigms and heterogeneous hardware architectures. By embracing TMP methodologies, Kokkos facilitates device-specific code generation and optimizations via template specialization. To cater to diverse computing environments, Kokkos offers multiple device-specific backends, implemented as template libraries atop various HPC programming models like CUDA, HIP, OpenMP, HPX, SYCL, and OpenACC. These backends are tailored to adhere to advancements in the C++ standard, ensuring compatibility and efficacy.

A notable departure from Alpaka is Kokkos' emphasis on descriptive rather than prescriptive parallelism. Kokkos prompts developers to articulate algorithms in general parallel programming concepts, which are subsequently mapped to hardware by the Kokkos framework. The Kokkos programming model revolves around two fundamental abstractions: the first being the user data abstraction (Kokkos::View), a template library facilitating the representation of multidimensional arrays while managing efficient data layout for both CPU and GPU. The second abstraction revolves around parallel execution patterns (parallel_for, parallel_reduce, and parallel_scan), which can be executed under three distinct execution policies: RangePolicy for mapping single parallel loops, MDRangePolicy for mapping

directly nested parallel loops, and `TeamPolicy` for hierarchical mapping of multiple nested parallel loops.

In this study, Kokkos versions were developed by translating CUDA code to Kokkos using the Kokkos `View` and parallel dispatch abstractions. The initial translation process was relatively straightforward, owing to the striking similarities between the Kokkos and CUDA execution models and memory models, except for the added complexity due to Kokkos-specific restrictions on C++ template programming. Efficient execution on both CPU and GPU took further optimization and was achieved by configuring the `teamSize` and `vectorSize` in the `TeamPolicy` used in the `parallel_for` execution pattern, as the defaults were found not to be optimal.

4.6 Standard parallelization using stdpar in C++

The C++ programming language is often the preferred choice for implementing high performance scientific applications. The recent revisions of the ISO C++ standard introduced a suite of algorithms capable of being executed on accelerators. Although this approach may not yield the best performance, it can present a viable balance between code productivity and computational efficiency. Numerous production-grade compilers are available, such as clang and its variants from various providers, or the recently released NVHPC from NVIDIA.

With the introduction of the C++17 standard, the C++ Standard Template Library (STL) underwent a substantial overhaul of its suite of algorithms, now updated with execution policies to adapt across various computing architectures, including multi-core x86 systems and GPUs. These parallel algorithms extended most of the existing STL algorithms with an additional argument, which is an execution policy. The policy enables programmers to specify the intended parallelism of an algorithm, which can result in performance improvements for the computational task. In particular, the execution policies in C++17 include:

- `std::execution::seq`
- `std::execution::unseq`
- `std::execution::par`
- `std::execution::par_unseq`

The first option forces the algorithm to run sequentially, while the remaining three options allow the algorithm to be vectorized or run in parallel (with additional vectorization). Both the `p2z` and `p2r` stdpar implementations use the `std::execution::par_unseq` execution policy. Currently, only the nvc++ compiler offers support for stdpar algorithms to be offloaded on NVIDIA GPUs. It leverages CUDA Unified Memory to handle automatic data movement between CPU and GPU. On the systems that do not support Heterogeneous Memory Management, only data that is dynamically allocated in CPU code compiled by nvc++ is automatically managed, whereas memory allocated in GPU code is exclusively for GPU use and remains unmanaged. Thus, on such systems, CPU and GPU stack memory, along with global objects, are outside nvc++'s automatic

management scope. Even data allocated on the CPU heap outside units compiled by nvc++ is not managed. When dealing with parallel algorithm invocations, pointers and objects must refer to data on the managed CPU heap to avoid errors. Any dereferencing of pointers to CPU stack or global objects in GPU code can lead to memory violations. These aspects encapsulate the nvc++ compiler's precise approach to memory management across CPU and GPU, emphasizing careful allocation and reference handling to ensure efficient operations. In our experience, developing code for this application is largely similar to standard C++ programming, with the primary distinction being the need to consider the previously mentioned limitations.

4.7 SYCL

SYCL represents a cross-platform abstraction layer that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++. This approach aims to enhance the efficiency and accessibility of programming for a variety of compute architectures. While the SYCL programming language was promoted by the Khronos Group, it is predominantly advocated for by Intel, so the primary focus is on the optimization for Intel GPUs.

One of the key advantages of SYCL is the ability to handle regular C++ code for the host CPU and a subset of C++ for the device code within the same source file. This ability paves the way for an integrated and simplified development process. It also enables parallelism and the usage of memory hierarchies through a class template library. This effectively allows the expression of parallel STL, thus further integrating SYCL with standard C++ features. Since it was designed to be fully compatible with standard C++, it allows developers to utilize any C++ library within a SYCL application. This compatibility with standard C++ makes SYCL a versatile tool for developers. Moreover, SYCL's design prioritizes performance portability, aiming to provide high performance across a wide range of hardware architectures. Its abstractions are constructed to allow optimization but do not require a particular architecture or kernel language.

For the benchmarks described in this paper, the programming approach for SYCL is nearly a direct replication of the CUDA approach. In our implementation, we utilized SYCL's Unified Shared Memory feature for data management. To compile SYCL for NVIDIA and AMD GPUs, we compiled the dpc++ tool chain following the instructions in Intel's open-source llvm-sycl repository.[1]

5 Results

The most important computing metric, from the point of view of HEP computing as a whole, is the algorithm's throughput, defined here as the number of tracks that can be processed per second. To measure the throughput, all versions processed approximately 800k tracks in a single kernel. The kernel, together

---

1 Intel   llvm/sycl   branch,   https://github.com/intel/llvm/tree/70c2dc6dcf73f645248aa7c70c8cefdabf37e9b7.
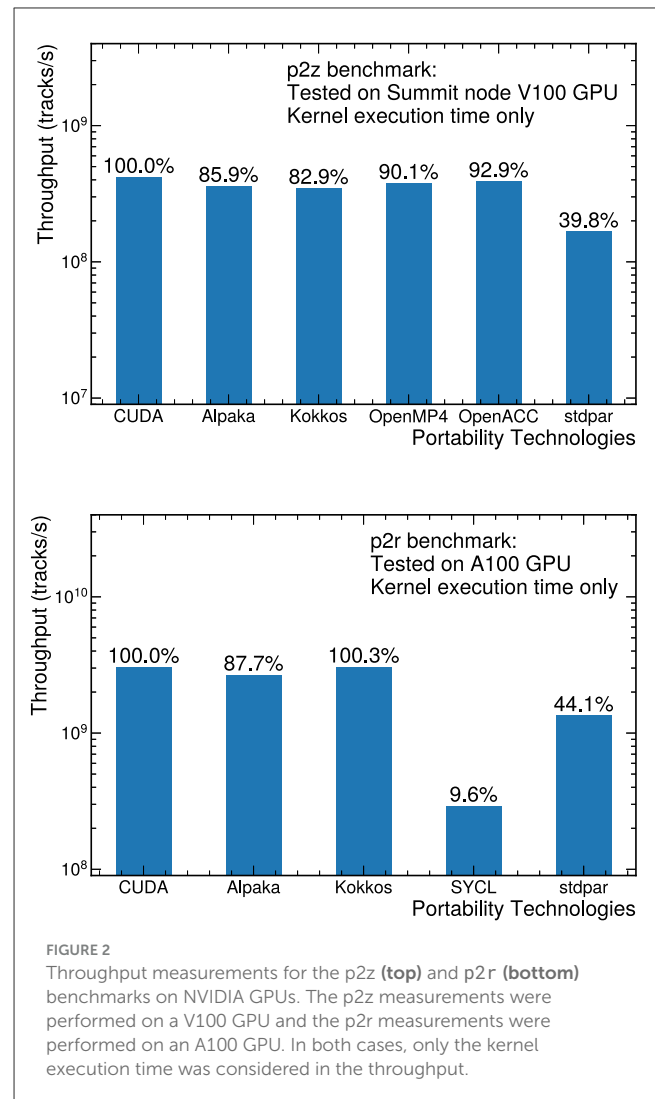
with the data movement operations, are then iterated a fixed number of times to sustain a total program run time of around one minute. The measurements were repeated multiple times to ensure stability of results. We tested the performance of different tools on a number of different hardware systems, including NVIDIA GPUs (Section 5.1), Intel and AMD GPUs (Section 5.2), and Intel CPUs (Section 5.3).

The code used for the results in this paper is directly extracted from the MKFIT application. In order to reduce overheads, it does not have its own setup for validating the results in terms of physics output. For a discussion of the MKFIT physics performance results, see the MKFIT papers (Cerati et al., 2017; Lantz et al., 2020). Numerical reproducibility and consistency of results across different p2z and p2r versions are verified with summary printouts. Since the artificial sample is produced by smearing the inputs from a single track, printouts report the mean and RMS values of the track parameters computed over the full sample. Output results from different versions are numerically identical when the same compiler is used. Compilers may introduce differences in the average parameter values at or below the ppm level. Such differences are due to numerical precision in floating point operations and the different levels of optimization used by default by different compilers. The level of agreement in the output guarantees that all versions perform the same operations, making the comparisons in terms of computing performance meaningful.
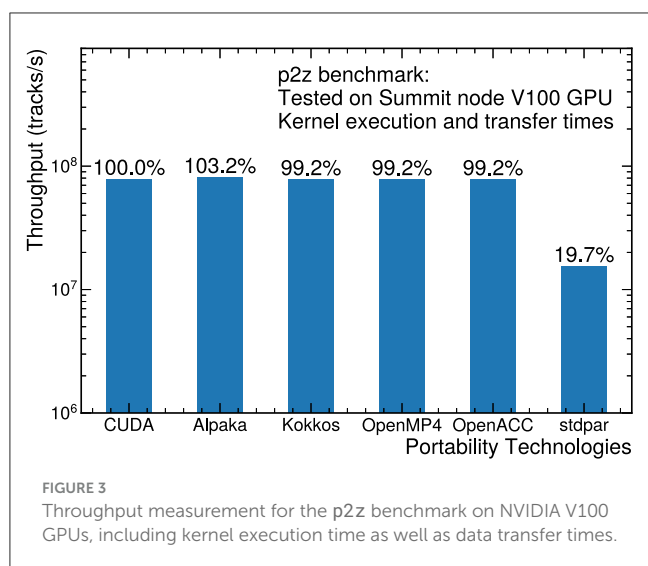
5.1 NVIDIA GPU results

The throughput of the p2z and p2r benchmarks were measured using two different systems with two different NVIDIA GPUs. To test the p2r implementations, we used the Joint Laboratory for System Evaluation (JLSE), a collection of HPC testbeds hosted at Argonne National Lab. The NVIDIA GPU that was used for testing is an A100 GPU with an AMD 7532 CPU as the host machine. For the p2z implementations, the measurements were performed on a test node for the Summit HPC system at Oak Ridge National Laboratory. Each Summit node includes six NVIDIA V100 GPUs and two IBM Power9 CPUs, although only one GPU was utilized in our tests.

The throughput measurements on NVIDIA GPUs are shown in Figure 2 for both the p2z and p2r benchmarks. In both cases, the native CUDA implementation is used as the reference implementation, and the measurement time includes the kernel execution only. Several different compilers are used for the different implementations: the CUDA, Alpaka, and Kokkos implementations are compiled with the nvcc compiler; OpenMP and OpenACC are compiled with the OpenARC (Lee and Vetter, 2014) compiler; the stdpar versions are compiled with nvc++; and for SYCL, the intel/llvm-sycl repository is used to compile dpc++ tool chains to be compatible with NVIDIA/AMD GPUs. For evaluation, whenever possible, we use the same launch parameters, including number of blocks and number of threads per block. In the OpenACC and OpenMP versions, different compilers varied in how they enforced the user-specified configurations (see Section 5.1.1). The launch parameters need to be manually specified for Alpaka and Kokkos, otherwise the libraries choose suboptimal

FIGURE 2
Throughput measurements for the p2z **(top)** and p2r **(bottom)** benchmarks on NVIDIA GPUs. The p2z measurements were performed on a V100 GPU and the p2r measurements were performed on an A100 GPU. In both cases, only the kernel execution time was considered in the throughput.

values and the performance is about 30% worse, taking those versions even further from the native CUDA version. Setting the number of registers per thread is another approximately 10% effect. These parameters cannot be manually specified in the stdpar implementation. In the SYCL implementations, we specified the execution volume manually but relied on the defaults for the other launch parameters.

In general, most of the different portability solutions managed to produce close-to-native performance. The stdpar implementation did not perform well for either benchmark, mainly because the stdpar implemented in the nvc++ compiler relies on CUDA Unified Memory for all data movements between CPU and GPU memory, which fetches data needed by the GPU kernel on demand, exposing memory transfer overheads to the kernel execution. CUDA Unified Memory provides APIs to prefetch or migrate data between CPU and GPU memory to hide or reduce the memory transfer overheads, but the current stdpar does not include such functionalities. In order to mitigate the effects of data transfers while measuring the kernel execution time, we introduced implicit prefetching using the parallel std::copy algorithm. For the p2r benchmark, the worst performing version

FIGURE 3
Throughput measurement for the p2z benchmark on NVIDIA V100 GPUs, including kernel execution time as well as data transfer times.

is the SYCL implementation. Detailed profiling using NVIDIA NSight Compute shows significant branching when using SYCL, but preliminary investigations have not revealed an obvious explanation for the branching.

For the p2z benchmark, most of the different implementations achieved similar performance except for the stdpar version. While similar to other p2z implementations, the relative performance of p2z Kokkos version is lower than that of the p2r Kokkos version, when compared to their corresponding CUDA versions. Detailed profiling shows that the Kokkos version of p2z uses more registers than the CUDA version, while the Kokkos version of p2r uses a similar number of registers compared to the CUDA version. Further investigations of the branching of the p2r SYCL version and of the difference in register usage in the p2z and p2r Kokkos version are beyond the scope of the present paper.

Figure 3 shows the throughput of the p2z implementations on the NVIDIA V100 GPUs, this time including both the kernel execution time and the memory transfer times. The transfer times are generally 2 to 5 times larger then the kernel execution times, which means that much of the variability between implementations is concealed. With the exception of stdpar, all implementations have close to identical performance because they provide memory management features to facilitate explicit data transfers, and in some cases memory prepinning as discussed in Section 5.1.2.

5.1.1 Compiler dependence

Different compilers can yield very different timing results, especially for the directive-based portability solutions. Figure 4 shows the throughput performance for the OpenACC and OpenMP p2z versions on NVIDIA V100 GPUs, including both kernel execution time and data transfer times. The left bars in Figure 4 show that the OpenARC-compiled OpenMP version performs better than the versions compiled with llvm, gcc, or IBM. Detailed profiling shows that the llvm, gcc, and IBM compiled versions use different launch parameters (the number of threads in a thread block and the number of thread blocks)
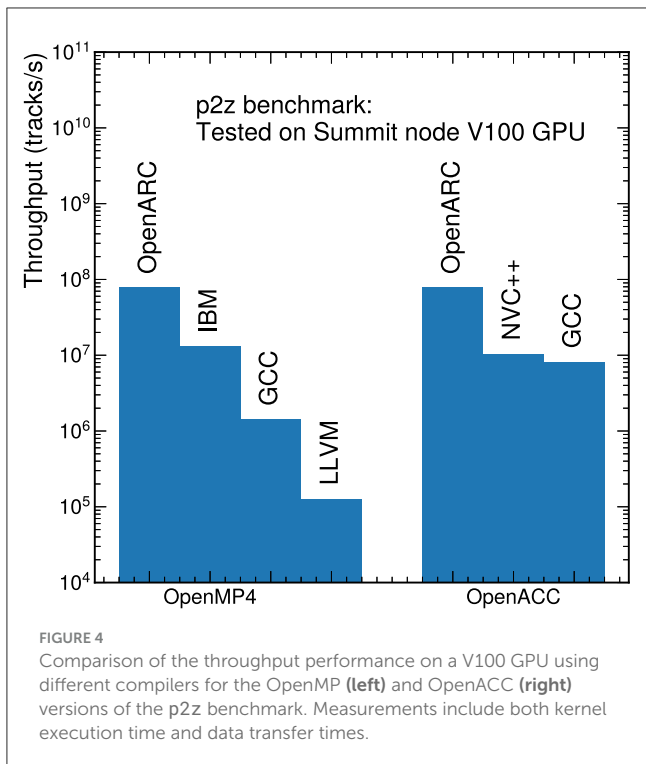
than those specified in the OpenMP program, while the OpenARC-compiled version literally follows the user-specified configuration. Different launch parameters in the llvm/gcc/IBM compiled versions adversely affected the concurrency. The OpenMP version of p2z allocates temporary user data in the team-private memory: the OpenARC-compiled version allocates the team-private data in the CUDA shared memory, but the llvm/gcc/IBM compiled versions only partially use the CUDA shared memory, which incurs more device global memory accesses than the OpenARC-generated version. Lower concurrency and more global memory accesses seem to be the main reasons for the lower performance of the llvm/gcc/IBM compiled OpenMP versions.

In the OpenACC version of p2z (the right bars in Figure 4), both the nvc++ and OpenARC compiled versions achieved similar kernel computation times, but the OpenARC-compiled version had better memory transfer performance than the version compiled with nvc++. OpenARC literally translates the OpenACC data clauses into corresponding memory transfer APIs (one transfer API call per list item in a memory transfer data clause), but the nvc++ compiler automatically splits memory transfers into multiple, small asynchronous memory transfer calls. Splitting memory transfers may expose more communication and computation overlapping opportunities, but in the p2z case, too many small asynchronous memory transfers in the nvc++ version perform worse than the simple memory transfers implemented in OpenARC, which is the main reason for the better performance of the OpenARC version than the nvc++ version. Like OpenARC, gcc also generates one memory transfer API call per list item in a memory transfer data clause, but the gcc version achieves lower memory transfer throughput than the OpenARC version, which may be caused by the host memory pre-pinning optimization, which is supported by OpenARC but not by gcc, as shown in the next section.

5.1.2 Effect of memory prepinning

For GPU versions, implementing host memory pinning was found to greatly improve the performance, which is shown in Figure 5 for the Kokkos and OpenACC p2z versions, including both kernel execution time and data transfer times. In the system with NVIDIA GPUs, prepinning the host memory enables direct-memory access (DMA) transfers, which achieve better memory transfer bandwidth than non-DMA transfers.

We compared the performance with and without explicit prepinning for three OpenACC versions (Figure 5, left). The first uses batched shared memory and synchronous transfers. The second implementation uses batched shared memory and asynchronous transfers, and the final version uses thread-private data on local memory with asynchronous transfers. Like the CUDA and HIP implementations (see Section 4.2), the results show that keeping the intermediate results in the local register (thread-private data version) performs better than using the shared memory. When host memory prepinning is on, all of the host data appearing in OpenACC clauses are prepinned. In all cases, prepinning is observed to improve the performance. The asynchronous versions have on-demand host-memory pre-pinning, which means the host memory is prepinned before each asynchronous memory transfer if it is not already prepinned. Therefore, a smaller impact from

FIGURE 4
Comparison of the throughput performance on a V100 GPU using different compilers for the OpenMP **(left)** and OpenACC **(right)** versions of the p2z benchmark. Measurements include both kernel execution time and data transfer times.
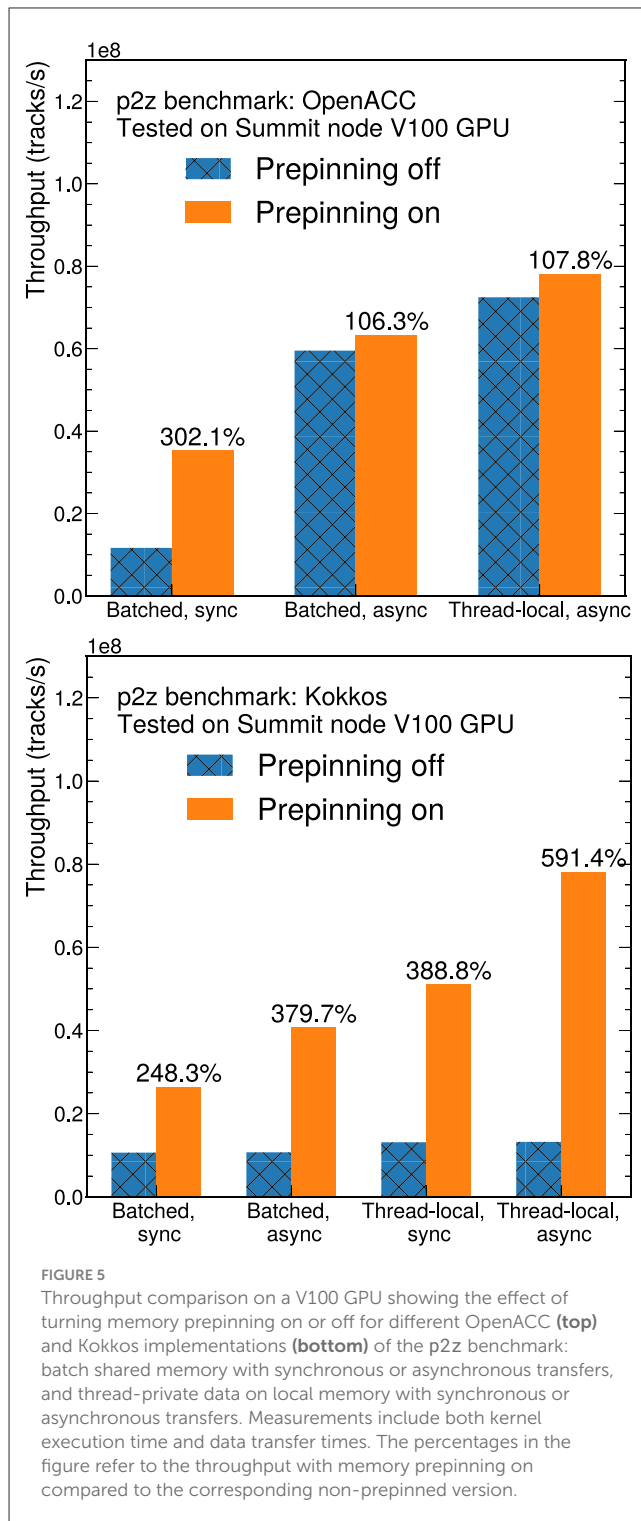
explicit prepinning is observed than for the synchronous transfer version. The performance improvement from the host memory prepinning is more pronounced in the Kokkos results (Figure 6, right), because the internal implementation of the Kokkos library requires additional memory transfers, such as copying the functor objects, while the OpenACC/OpenMP versions do not.

5.2 AMD and Intel GPU results

Support for other GPU architectures is in general less mature than the support for NVIDIA GPUs, although this is an area of rapid expansion. We explored the preliminary performance of the p2r benchmark on both AMD and Intel GPUs, and the kernel-only throughput measurements are shown in Figure 6. This is an out-of-the-box comparison of each tool's portability; no dedicated effort was made to optimize for AMD or Intel GPU architectures. For example, the HIP implementation for the AMD GPU is a carbon copy of the native CUDA version. For both Alpaka and Kokkos, switching backends is relatively seamless and does not require any code changes.

The results on the AMD GPU are shown in the left plot of Figure 6. The AMD GPU tests were performed on the JLSE testbed, which includes two AMD EPYC 7543 32c (Milan) CPUs and four AMD MI100 32GB GPUs. Only one GPU was used to perform the measurements. Both Kokkos and Alpaka include HIP backends which achieve reasonable performance: Alpaka actually outperforms the HIP version that was ported from CUDA, and Kokkos is within a factor of about 2. The same launch parameters as the CUDA implementation are used for the native HIP, Alpaka:HIP, and Kokkos:HIP measurements.

FIGURE 5
Throughput comparison on a V100 GPU showing the effect of turning memory prepinning on or off for different OpenACC **(top)** and Kokkos implementations **(bottom)** of the p2z benchmark: batch shared memory with synchronous or asynchronous transfers, and thread-private data on local memory with synchronous or asynchronous transfers. Measurements include both kernel execution time and data transfer times. The percentages in the figure refer to the throughput with memory prepinning on compared to the corresponding non-prepinned version.

The throughput measurements on an Intel A770 GPU are shown in the right plot of Figure 6. Since the A770 is not an HPC-class GPU, all calculations were converted to single-precision operations. Relying on double-precision emulation results in performance that is 3 to 30 times slower, depending on the implementation. The SYCL backend for Alpaka has only experimental support (introduced in v0.9.0) and was not tested

FIGURE 6
Throughput measurements for the `p2r` benchmark on an AMD MI-100 GPU **(top)** and on an Intel A770 GPU **(bottom)**. The performance of each implementation is compared to the performance of the native version (HIP for the AMD GPU and SYCL for the Intel GPU). Only the kernel execution time was included in the measurements.
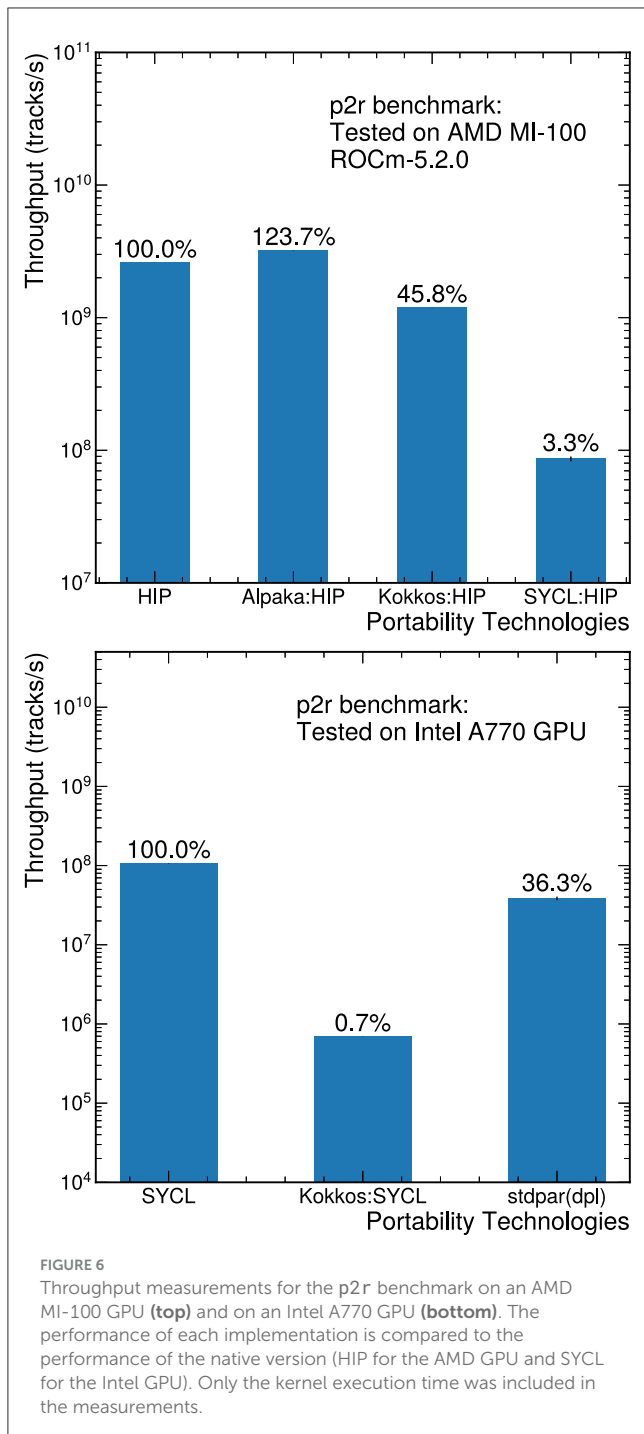
here. The Kokkos SYCL backend is still under active development: we observed a factor of 2 improvement in the throughput of the Kokkos:SYCL implementation when we updated from Kokkos 3.6.1 to Kokkos 4.1.0.

5.3 CPU results

The original MKFIT application is parallelized using the Threading Building Blocks (TBB) library from Intel, so we also

used TBB as the reference native implementation on the CPU. This implementation is multithreaded and vectorized. It is worth noting that the original version was initially developed based on the "classic" Intel C++ Compiler (icc version 19), which led to improved vectorization performance compared to more recent compilers, resulting in `p2z` execution times that were approximately 2.7x faster. However, since this version is not supported anymore, we choose not to include it in our main results. Figure 7 shows the throughput performance of the `p2z` and `p2r` benchmarks on a two-socket system equipped with Intel Xeon Gold 6248 CPUs. All implementations were compiled with gcc except stdpar, which was compiled with nvc++, and SYCL, which was compiled with dpcpp. Nonstandard options such as `-ffast-math` are not included in the compilation since, while they may help vectorizing loops including trigonometric functions, they do not guarantee numerical reproducibility.

Using the portability layers, we are able to achieve throughput equal to or better than 70% of the original, native performance on the CPU for most implementations. The Alpaka implementation of the `p2z` benchmark actually outperformed the TBB reference implementation. The reason for the better performance of `p2z` Alpaka is not currently known, and the results might be platform dependent. Note that the `p2z` Alpaka implementation uses the OpenMP execution backend, while the `p2r` Alpaka implementation relies on the TBB execution backend. The `p2r` SYCL implementation only achieves 27% of the reference implementation, but unlike many of the other implementations, SYCL is a language extension and depends heavily on compiler optimizations rather than a performance-tuned library. When optimizing these implementations, considerations included deciding on an optimal data layout that works for both CPUs and GPUs and ensuring that the loops are properly vectorized when run on the CPU.

6 Summary

In the project described in this paper, we ported two benchmark applications for charged-particle track reconstruction using state-of-the-art portability tools and compared the results to the native implementations. These benchmarks could form the backbone of a realistic track fitting algorithm. We have tested our ports on Intel CPUs and on GPUs from different vendors. In developing and testing these benchmarks, we found that the performance can vary significantly depending on the details of the implementation. Achieving optimal performance was not easy, even for relatively simple applications such as these. Each implementation took several iterations of profiling and development to achieve the results shown here. Ranking the difficulty of each porting effort would not be straightforward, since the lessons learned when implementing one solution would often apply to subsequent solutions, without the need for rediscovery. Even so, the steps that were taken to improve performance on one type of accelerator (NVIDIA GPUs, for example) did not necessarily translate into analogous gains on other types of GPUs or CPUs.

Several factors were found to have large effects on the final performance. We found that optimizing the memory layout and enabling explicit memory prepinning (in the case of

FIGURE 7
Throughput measurements for p2z **(top)** and p2r benchmarks **(bottom)** on a two-socket system equipped with Intel Xeon Gold 6248 CPUs. The performance of each implementation is compared to the performance of the native TBB version.
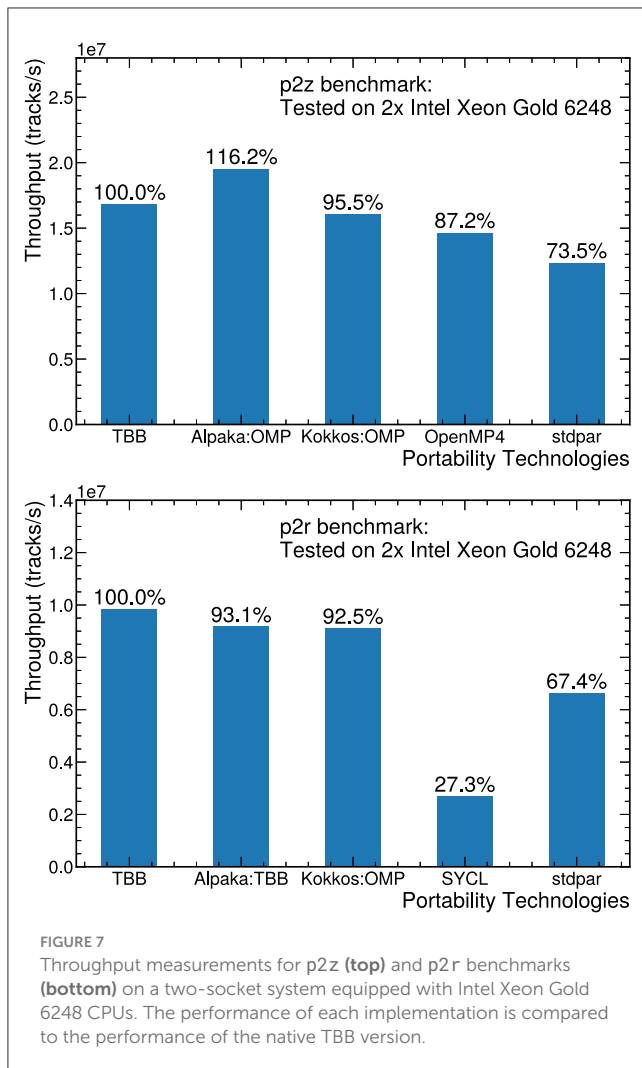
NVIDIA GPUs) led to big improvements in the performance of each implementation, up to a factor of six speedup. The choice of compiler also changed the throughput performance on NVIDIA GPUs by an order of magnitude or more for OpenMP and OpenACC implementations. Because these compilers are undergoing very active development, regularly checking performance with the latest versions is important. The same is true for the tools themselves, especially with respect to support for AMD and Intel GPUs. For example, we observed a factor of two speedup on an Intel GPU when updating to a newer version of the Kokkos library.

In summary, we explored major portability solutions using two benchmark applications from HEP, including implementations using Alpaka, Kokkos, SYCL, stdpar, OpenMP, and OpenACC. Most of these solutions can give reasonable performance, on the same order of magnitude, on different architectures, but most cases required significant optimization. The ability to run algorithms on GPUs from different vendors will allow HEP experiments to take advantage of a variety of computing resources, including current, planned, and future HPCs. This paper demonstrates that while tools exist to effectively port existing CPU algorithms to GPUs,

reaching the desired performance is not yet straightforward. Future development of these portability solutions and their application, or alternative methods, will be necessary for the successful operation and data analysis of these experiments.

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/cerati/p2z-tests/releases/tag/v1.0; https://github.com/cerati/p2r-tests/releases/tag/v1.0.

Author contributions

HA: Investigation, Software, Writing – review & editing. SB: Investigation, Writing – review & editing. GC: Investigation, Project administration, Software, Writing – original draft, Writing – review & editing. MK: Investigation, Writing – original draft, Writing – review & editing, Software. KK: Investigation, Software, Writing – original draft, Writing – review & editing. SLa: Investigation, Writing – original draft, Writing – review & editing. SLe: Investigation, Software, Writing – original draft, Writing – review & editing. BN: Investigation, Software, Writing – review & editing. MR: Investigation, Software, Writing – review & editing. AR: Investigation, Project administration, Visualization, Writing – original draft, Writing – review & editing. DR: Investigation, Writing – review & editing. AS: Investigation, Software, Writing – original draft, Writing – review & editing. CW: Investigation, Software, Writing – review & editing.

Funding

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

Albrecht, J., Alves, A. A., Amadio, G., Andronico, G., Anh-Ky, N., Aphecetche, L., et al. (2019). A roadmap for HEP software and computing R&D for the 2020s. *Comp. Softw. Big Sci.* 3:8. doi: 10.1007/s41781-018-0018-8

Apollinari, G., Brüning, O., Nakamoto, T., and Rossi, L. (2015). *High Luminosity Large Hadron Collider HL-LH. Technical Report FERMILAB-PUB-15-699-TD*, CERN Yellow Rep. Available at: https://cds.cern.ch/record/2116337

Atif, M., Battacharya, M., Calafiura, P., Childers, T., Dewing, M., Dong, Z., et al. (2023). *Evaluating Portable Parallelization Strategies for Heterogeneous Architectures in High Energy Physics*. Available at: https://inspirehep.net/literature/2672556

ATLAS Collaboration (2008). The ATLAS experiment at the CERN Large Hadron Collider. *JINST* 3:S08003. doi: 10.1088/1748-0221/3/08/S08003

Berkman, S., Cerati, G., Knoepfel, K., Mengel, M., Reinsvold Hall, A., Wang, M., et al. (2022). Optimizing the hit finding algorithm for liquid argon TPC neutrino detectors using parallel architectures. *JINST* 17:P01026. doi: 10.1088/1748-0221/17/01/P01026

Bhattacharya, M., Calafiura, P., Childers, T., Dewing, M., Dong, Z., Gutsche, O., et al. (2022). "Portability: a necessary approach for future scientific software," in *Proceedings of the US Community Study on the Future of Particle Physics (Snowmass 2021)*. Available at: https://inspirehep.net/literature/2054702

Bird, I. (2011). Computing for the large Hadron Collider. *Ann. Rev. Nucl. Part. Sc.* 61, 99–118. doi: 10.1146/annurev-nucl-102010-130059

Bocci, A., Czirkos, A., Pilato, A. D., Pantaleo, F., Hugo, G., Kortelainen, M., et al. (2023). Performance portability for the CMS reconstruction with Alpaka. *J. Phys. Conf. Ser.* 2438:012058. doi: 10.1088/1742-6596/2438/1/012058

Cerati, G., Elmer, P., Gartung, P., Giannini, L., Kortelainen, M., Krutelyov, V., et al. (2023). "Generalizing mkFit and its application to HL-LHC," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP23)*. Available at: https://www.epj-conferences.org/articles/epjconf/abs/2024/05/epjconf_chep2024_03019/epjconf_chep2024_03019.html

Cerati, G., Elmer, P., Krutelyov, S., Lantz, S., Lefebvre, M., Masciovecchio, M., et al. (2017). "Parallelized Kalman-filter-based reconstruction of particle tracks on many-core processors and GPUs," in *Proceedings of Connecting The Dots/Intelligent Trackers 2017 (CTD/WIT 2017)* (Les Ulis: EDP Sciences), 00006.

CMS Collaboration (2008). The CMS Experiment at the CERN LHC. *JINST* 3:S08004. doi: 10.1088/1748-0221/3/08/S08004

CMS Collaboration (2014). Description and performance of track and primary-vertex reconstruction with the CMS tracker. *JINST* 9:P10009. doi: 10.1088/1748-0221/9/10/P10009

CMS Offline Software, and Computing (2021). "Evolution of the CMS computing model towards Phase-2," in *Technical Report CMS-NOTE-2021-001, CERN-CMS-NOTE-2021-001* (Geneva: CERN).

CMS Offline Software and Computing (2022). "CMS Phase-2 computing model: update document," in *Technical Report CMS-NOTE-2022-008, CERN-CMS-NOTE-2022-008* (Geneva: CERN).

DUNE Collaboration (2020). Deep underground neutrino experiment (DUNE), far detector technical design report, volume I introduction to DUNE. *JINST* 15:T08008. doi: 10.1088/1748-0221/15/08/T08008

DUNE Collaboration (2022). "DUNE offline computing conceptual design report," in *Technical Report FERMILAB-DESIGN-2022-01, Fermilab*. Available at: https://inspirehep.net/literature/2171912

Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *J. Parall. Distrib. Comp.* 74:3202–3216. doi: 10.1016/j.jpdc.2014.07.003

Fruhwirth, R. (1987). Application of Kalman filtering to track and vertex fitting. *Nucl. Instrum. Meth.* A262:444–450. doi: 10.1016/0168-9002(87)90887-4

Jones, C. D., Paterno, M., Kowalkowski, J., Sexton-Kennedy, L., and Tanenbaum, W. (2006). "The new CMS event data model and framework," in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP06)*. Available at: https://indico.cern.ch/event/408139/contributions/979800/attachments/815724/1117731/FrameworkPaper.pdf

Kortelainen, M. J., Kwok, M., Childers, T., Strelchenko, A., and Wang, Y. (2021). "Porting CMS heterogeneous pixel reconstruction to Kokkos," in *25th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2021)*, 03034. Available at: https://www.epj-conferences.org/articles/epjconf/abs/2021/05/epjconf_chep2021_03034/epjconf_chep2021_03034.html

Lantz, S., McDermott, K., Reid, M., Riley, D., Wittich, P., Berkman, S., et al. (2020). Speeding up particle track reconstruction using a parallel Kalman filter algorithm. *J. Instrument.* 15, P09030–P09030. doi: 10.1088/1748-0221/15/09/P09030

Lee, S., and Vetter, J. (2014). "OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *HPDC'14: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper* (New York, NY: Association for Computing Machinery). doi: 10.1145/2600212.2600704

Matthes, A., Widera, R., Zenker, E., Worpitz, B., Huebl, A., and Bussmann, M. (2017). "Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library," in *High Performance Computing. ISC High Performance 2017. Lecture Notes in Computer Science, 10524* (Cham: Springer). doi: 10.1007/978-3-319-67630-2_36

Megino, F. B., Bryant, L., Hufnagel, D., and Anampa, K. H. (2023). *US ATLAS and US CMS HPC and Cloud Blueprint*. Available at: https://inspirehep.net/literature/2651770

Trott, C. R., Lebrun-Grandi, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., et al. (2022). Kokkos 3: programming model extensions for the exascale era. *IEEE Transactions on Parall. Distrib. Syst.* 33, 805–817. doi: 10.1109/TPDS.2021.3097283