frontiers | Frontiers in Computer Science

TYPE Technology and Code
PUBLISHED 19 June 2024
DOI 10.3389/fcomp.2024.1407365

OPEN ACCESS

EDITED BY
Huai Liu,
Swinburne University of Technology, Australia

REVIEWED BY
Yongquan Fu,
National University of Defense Technology, China
Luca Deri,
University of Pisa, Italy

*CORRESPONDENCE
Konstantinos Papadakis
✉ konstantinos.papadakis@helsinki.fi

RECEIVED 26 March 2024
ACCEPTED 31 May 2024
PUBLISHED 19 June 2024

CITATION
Papadakis K, Battarbee M, Ganse U, Pfau-Kempf Y and Palmroth M (2024) Hashinator: a portable hybrid hashmap designed for heterogeneous high performance computing.
Front. Comput. Sci. 6:1407365.
doi: 10.3389/fcomp.2024.1407365

COPYRIGHT
© 2024 Papadakis, Battarbee, Ganse, Pfau-Kempf and Palmroth. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

Hashinator: a portable hybrid hashmap designed for heterogeneous high performance computing

Konstantinos Papadakis [1]*, Markus Battarbee [1], Urs Ganse [1], Yann Pfau-Kempf [1] and Minna Palmroth [1,2]

[1]Department of Physics, University of Helsinki, Helsinki, Finland, [2]Space and Earth Observation Centre, Finnish Meteorological Institute, Helsinki, Finland

Scientific computing has become increasingly parallel and heterogeneous with the proliferation of graphics processing unit (GPU) use in data centers, allowing for thousands of simultaneous calculations accessing high-bandwidth memory. Adoption of these resources may require re-design of scientific software. Hashmaps are a widely used data structure linking unsorted unique keys with values for fast data retrieval and storage. Several parallel libraries exist for performing hashmap operations utilizing GPU hardware, but none have yet supported GPUs and CPUs interchangeably. We introduce Hashinator, a novel portable hashmap designed to operate efficiently on both CPUs and GPUs using CUDA or HIP/ROCm Unified Memory, offering host access methods, in-kernel access methods, and efficient GPU offloading capability on both NVIDIA and AMD hardware. Hashinator utilizes open addressing with Fibonacci hashing and power-of-two capacity. By comparing against existing implementations, we showcase the excellent performance and flexibility of Hashinator, making it easier to port scientific codes that rely heavily on the use of hashmaps to heterogeneous architectures.

KEYWORDS

hashmaps, hashtable, GPU, heterogeneous computing, CUDA, HIP, HPC

1 Introduction

Hashmaps serve as a crucial component in computer science, offering an efficient mechanism for mapping unordered keys to values. They are commonly used in applications including database management and data compression as well as in scientific computing. Hashmaps operate by using a hash function that maps input keys to indices in an array. This specific index links to the location of the related value in the array, enabling quick access to the desired data. The size of this array is typically associated with the maximum number of keys that the hashmap can store. To prevent collisions, where multiple keys lead to the same index, hashmaps use collision resolution techniques such as open and closed addressing (Liu and Xu, 2015), double hashing (Cormen et al., 2001a) and perfect hashing (Cormen et al., 2001b). This allows multiple values to be stored at the same index without data loss. However, central processing unit (CPU) based hashmap implementations suffer from low throughput usually caused by irregular memory access patterns in their probing mechanism. This has been mitigated through hashmaps which utilize graphics processing units (GPUs) Lessley and Childs (2020); Awad et al. (2023) which are capable of leveraging their massive parallelism to obscure memory latency and

efficiently manage very high hashing throughput. These hashmaps operate on a GPU and are tailored to handle large parallel data processing tasks. With the increasing need for efficient data processing solutions (Freiberger, 2012) and scientific codes turning to hybrid computing (Burau et al., 2010), GPU hashmaps have gained popularity in high-performance computing (HPC) in recent years. The majority of early implementations of GPU hashmaps were static (Awad et al., 2021) and exclusively provided read functionality on the GPU. Other implementations could only be operated on device code and not on the host.

One example of the current state of the art in GPU hashmaps is Warpcore (Jünger et al., 2020). Warpcore is a library that provides optimized hashmaps for GPUs, including both single and multi-value hashmaps. In their Single Value HashTable, a novel probing scheme introduced in WarpDrive (Jünger et al., 2018) is utilized, which employs CUDA's Cooperative Group mechanism to efficiently traverse the probing chain and achieve very high insertion (1.6 billion key-values per second) and retrieval (4.2 billion key-values per second) rates. Although Warpcore has demonstrated significant advancements and performance, it has a limitation in the flexibility it provides. Warpcore's interface consists of host-side (CPU) methods which perform the necessary hashmap procedures using highly optimized device kernels. Direct access to hashmap entries from host is not supported. In short, in Warpcore a user is unable to insert elements into an already existing hashmap from host code. The insertion of elements into a hashmap in Warpcore must be carried out by calling device code, and by providing the respective elements as inputs together with the respective CG to use. While this may not pose an issue for certain workflows, other workflows may require a more versatile approach for managing their hashmaps, allowing for their operation from both host and device code (GPU).

In the context of the Exascale era and in the constantly evolving landscape of high-performance computing (HPC), the porting of scientific codes from CPU to GPU architectures has emerged as a crucial aspect in achieving optimal performance. In this work, we introduce Hashinator (Papadakis et al., 2024), a novel portable hashmap implementation designed for scientific codes utilizing hashmaps on heterogeneous HPC environments. Hashinator simplifies the utilization of hashmaps by enabling their operations interchangeably across CPUs and GPUs. Hashinator can act as replacement for `std::unordered_map` for host code and that can seamlessly expose its data and functionalities on the GPU side of the codebase achieving very high throughput thus streamlining the porting process. In the rest of this manuscript, the terms "host" and "device" are adopted to distinguish between code executed on the CPU (host) and code executed on the GPU (device). Hashinator exploits the CUDA/HIP Unified Memory model, ensuring map data remains always valid whether it is accessed or edited from either host or device. Section 2 presents an in-depth examination of the design considerations in the development of Hashinator. A comprehensive analysis of the host and device implementations is provided, including a description of their operating principles. The probing methods utilized by Hashinator and its ability to perform operations such as inserting, deleting, and retrieving elements are demonstrated and discussed. In Chapter 3, a thorough evaluation of the performance of Hashinator for both the host and device Application Programming Interfaces (APIs) is conducted and compared to industry-standard hashmap implementations.

2 Method

The Hashinator library consists of the Hashinator hashmap itself, and an auxiliary vector implementation called SplitVector.

2.1 SplitVector

SplitVector is a vector library written in C++ that leverages CUDA/HIP Unified Memory (Li et al., 2015) to store its data and acts as a replacement for `std::vector`, whilst exposing its data to both the GPU and the CPU. SplitVector is designed as a flexible, header-only library and includes a comprehensive set of tools to allow for seamless data access and manipulation in both host and device code. To maintain portability SplitVector's codebase is architecture agnostic and can be compiled with CUDA and HIP compilers without modifications. Since SplitVector uses Unified Memory it provides the user with prefetching methods both to and from the device to enable robust memory handling and to avoid page faults resulting from on-demand data migration. The utilization of advanced functionalities and algorithms that come with SplitVector, including stream compaction and prefix scan routines, significantly enhances the performance of data processing in device code. Additionally, the availability of most host member functions in device code allows for seamless integration and efficient development. However, operations that modify the size, such as resizing, reallocating, and reserving, can only utilize storage up to the maximum capacity allocated via the host. As a result, these operations, when called from device, may fail if the user requests more space than what is currently allocated.

2.2 Hashinator: general implementation overview

Hashinator is a portable hashmap implementation designed for efficient lookup, insertion, and deletion of key-value pairs. The key-value pairs are stored in buckets, which are in turn stored in a contiguous memory region, which improves cache efficiency by making it more likely that adjacent buckets will be loaded into the cache together. This memory region is managed using SplitVector, which was introduced in Section 2.1. In particular, Hashinator uses an open addressing scheme (Liu and Xu, 2015), which means that when a collision occurs (i.e., two keys hash to the same bucket), it probes for the next available bucket in the table and places the item there (as opposed to a "closed bucket", chaining approach like a linked list implementation for resolving collisions). Hashinator employs a power-of-2 size for its hashmap, which means that the number of buckets in the table is always a power of 2 (e.g., 16, 32, 64, etc.). This makes it faster to compute the bucket for a given key using bitwise operations instead of using the modulo operator.

Choosing an appropriate hash function to map incoming keys to the underlying hashmap is critical for minimizing collisions.

Listing 1 32-bit Fibonacci multiplicative hash function. Here sizePower is the exponent of the capacity of the hashtable bucket array

```
uint32_t fibonacci(uint32_t key,const int
    sizePower){
   key ^= key >> (32 - sizePower);
   uint32_t retval = (uint64_t)(key * 2654435769
       ul) >> (32 - sizePower);
   return retval;
}
```

The hash function must be both fast to compute and sufficiently dispersive. In Hashinator, we use the Fibonacci multiplicative hash function (Knuth, 1998; Chen et al., 2013), which takes the form:

$$h(x) = \frac{M}{W}(Ax \bmod W), \quad (1)$$

where A is a predefined constant, M is the capacity of the hashtable bucket array and W represents the size (in bits) of the key being hashed. Defining $\phi = \frac{1}{2}(1+\sqrt{5})$ as the golden ratio we can select A to be:

$$A = \phi^{-1} W, \quad (2)$$

and we arrive at the Fibonacci multiplicative hash function which is characterized by having very few collisions in a given range while maintaining fast hashing rates. This makes it an excellent candidate for use in applications that require efficient hashing with minimal collisions. Furthermore, this choice of hash function enables Hashinator to maintain a low memory footprint using powers of 2 bucket sizes. The impact of this memory footprint improvement, compared against Warpcore, is exemplified in Section 2.5. In Listing 1 we demonstrate the 32-bit Fibonacci hash function used in Hashinator. In Hashinator, keys are restricted to 32 or 64-bit integer values, while the bucket values can be more complex objects, provided they are trivially copyable.

Hashinator employs a linear probing scheme to handle collisions, the specifics of which are detailed for each environment in the following sections. Upon deletion of an element from the hashmap, Hashinator employs a strategy to replace the associated key with a tombstone (Purcell and Harris, 2005), a sentinel marker indicating that the element has been deleted. This approach allows Hashinator's methods to skip over any tombstones during probing queries. It is important to note that the use of tombstones comes at the cost of filling the hashmap with residual data, which can potentially degrade performance. There are alternative approaches to handle collisions, such as the back-substitution method (Barnat and Ročkai, 2008), but they are not suited for parallel processing which is the target environment for Hashinator.

Hashinator offers three distinct interfaces that can be easily deployed during software development. In "host-only" mode, Hashinator performs all its operations on the CPU and the hashmap behaves like a replacement for the standard `std::unordered_map`. This mode is not thread-safe, and the hashmap should be used serially, similar to other standard implementations.

In "device-only" mode, Hashinator is passed into device code, and elements are inserted, queried, and deleted on the GPU. Access to Hashinator's device pointer is either gained through the `upload()` method, which returns a device pointer pointing to Hashinator or by the user using dynamic allocation for initializing Hashinator in Unified Memory.

Finally, Hashinator can be used in an "accelerated" mode, where all operations are launched on host code but can be offloaded to the GPU. This mode is intended for performance-critical applications where the goal is to achieve the highest possible throughput. We go into more details about the principles of operation and the performance of these three modes in the following paragraphs.

In "host-only" mode, the bucket overflow, that is the number of positions considered for insertion beyond the one determined by the hash function, is restricted to a predetermined limit, known as "bucket overflow limit" O_{lim}. Once the limit is exceeded, the contents of Hashinator are rehashed in a larger container with a capacity corresponding to the next power of 2. The bucket overflow limit is determined by the user. However, in "device-only" and "accelerated" modes, Hashinator allows the buckets to overflow up to the capacity of the SplitVector holding its buckets, tracked as "current overflow" O_{curr}. This allows Hashinator to continue its operations on device code even if the load factor, defined as the ratio of the number of occupied elements stored in the hashmap to its capacity, approaches unity. Following any host-only operation or if triggered by the user via a host-only method, Hashinator rehashes its contents if O_{curr} has surpassed O_{lim} and essentially reduces the load factor.

An important feature of both Hashinator and SplitVector is their portability, which allows them to be used even on platforms that do not support GPUs. Hashinator and SplitVector are designed to be compiled using a standard C++ compiler and only expose their GPU functionalities when compiled with a CUDA/HIP compiler. The codebase strictly maintains an architecture agnostic approach to provide portability between different systems. This design approach enables the development process to employ Hashinator and SplitVector in a variety of computing environments and provides a convenient option for transitioning to GPU-accelerated computing incrementally.

2.3 Hashinator: host only interface

Hashinator's interfaces are built around three fundamental operations: inserting new keys into the hashmap (or replacing a value associated with a key with a new one), deleting unwanted keys, and retrieving existing keys. Our goal in designing Hashinator was to provide functionality similar to that of standard implementations, so that it can serve as a near drop-in replacement. In the following section, we will detail the mechanism underlying these three key operations. We wish to note here that while thread-parallel accesses on the device are possible and optimized, the host API of Hashinator is not thread-safe so all the operations regarding insertion and deletion are undefined when operated in parallel.

2.3.1 Insertion

In Hashinator, new keys are first hashed using the Fibonacci hash function, as demonstrated in Listing 1, and then mapped

to a bucket in the hashmap. If the bucket is empty, the key is inserted directly. If the bucket is already occupied, we examine the key residing at the current bucket. If the key is the same as the candidate key, its value is updated with the candidate's value. If the key is different, we iterate linearly over the subsequent buckets and examine their keys until an empty bucket or a matching key is found. After inserting the new key-value pair, the value of O_{curr} is updated. However, if O_{lim} is reached during this probing sequence, we stop and reinsert all the currently existing elements into a new bucket storage array (constructed as a SplitVector), which has double the capacity of the previous one. After that, we re-attempt inserting the candidate key into the newly resized hashmap. This resizing is repeated until the process succeeds. The rehashing operation to double the capacity of the underlying bucket container is computationally expensive, alleviated by the capability of Hashinator being able to perform this operation in parallel on the GPU.

2.3.2 Retrieval

To retrieve a key's value from the hashmap, the key is first hashed using the same method as during insertion. This generates an index that points to a bucket location in the underlying buckets. The bucket is then examined, and if the key residing there matches the candidate key, its value is returned. If the key is not found at the first bucket, the process is repeated by examining the subsequent buckets until an empty bucket is encountered or O_{curr} is reached. If no matching key has been found, the candidate key does not exist in the hashmap. Depending on the specific querying method being used, Hashinator will either return an iterator pointing to the end of the hashmap or abort the execution.

2.3.3 Deletion

Deleting keys from Hashinator involves first hashing the key to generate an index in the underlying buckets. Similar to the retrieval process, the key is then checked to determine whether it exists in the hashmap or not. If the key is not found during probing or an empty key is found, Hashinator returns and does nothing. If the key is found during probing, it is replaced by a tombstone which is ignored during the probing sequence.

2.4 Hashinator: device only interface

Hashinator offers a device interface that allows for the implementation of various hashmap functionalities within device code. Specifically, users can access a device pointer by calling the `upload()` member function, which enables the insertion, querying, and deletion of keys from the hashmap in kernel code. When the `upload()` method is called, Hashinator performs an asynchronous prefetch of its data and bookkeeping information to the GPU so that operations launched later do not suffer page fault performance penalties. Hashinator can be also used in device code if allocated manually but in that case it is upon the user to execute proper data prefetching by calling the supplied methods of `optimizeGPU()` and `optimizeCPU()` to avoid any potential page faults. The device pointer provided by `upload()` is automatically deallocated at the end of lifetime of a Hashinator object.

2.4.1 Insertion

Device insertion generally follows the same approach as the host insertion method, with the same hash function and collision avoidance protocol being utilized to maintain interoperability. Yet, device code poses unique challenges in that numerous threads may attempt to write to the hashmap concurrently, which can result in data races. To address this issue, Hashinator employs CUDA/HIP atomic operations. When attempting to insert keys, threads follow the probing sequence and use atomic Compare and Swap (`atomicCAS`) operations to insert keys and atomic exchange operations (`atomicExch`) to update their values in a thread-safe manner. In situations where a candidate key cannot be inserted into a hashmap due to its probing chain being completely occupied (i.e. more collisions than O_{lim} have happened), the hashmap is allowed to overflow further. We keep probing for empty buckets to insert the candidate key and if successful we atomically update O_{curr} to match the current overflow. This makes each probing operation more expensive, however, it is a design decision we opt to make since otherwise an overflow situation would require termination of the kernel execution and specific handling of this situation on the host side.

2.4.2 Retrieval

Device retrieval does not carry the memory concurrency issues that trouble insertion, since the hashmap's state does not change when elements are retrieved. Threads can concurrently query keys from the hashmap safely, using the same approach as the host interface for querying keys. The query probing process may continue further than the O_{lim}, up to O_{curr}, or until an empty bucket is encountered.

2.4.3 Deletion

Deletion of elements on device code is following the same procedure as the deletion of elements on host code. The elements queued for deletion are atomically substituted with tombstones, which are ignored during the probing traversal.

2.5 Hashinator: accelerated interface

Apart from the host and device interfaces, Hashinator is equipped with an interface that can be operated by host code and use device kernels for performing its three basic operations, namely insertion, retrieval and deletion. The "accelerated" interface is designed to deal with bulk insertion, retrieval and deletion of key-value pairs from Hashinator. Crucially, this interface is the most efficient and results in very high performance. The methodology used is inspired by Warpcore (Jünger et al., 2020) and detailed in the following sections. The bucket overflow limit (O_{lim}) value significantly impacts the probing sequence of the accelerated interface. For the purposes of this discussion, we will assume an O_{lim} of 32, which aligns with the number of threads within a

CUDA warp unit. On AMD hardware, we would similarly use an O_{lim} of 64.

2.5.1 Insertion

To insert N key-value pairs into Hashinator using the accelerated interface, N CUDA warps must be launched, with each warp handling one of the N elements. The threads in each warp operate under SIMT (Single Instruction Multiple Threads) architecture, and our aim is to avoid branch divergence between the threads within each warp but to also make the most of the SIMT environment. Each key to be inserted into the map is hashed, providing the optimal bucket index. Each thread in the warp then accesses a subsequent bucket, offset by its warp thread index.

First, the threads perform an intra-warp vote using the `_ballot_sync()` (or the HIP equivalent `_ballot()`) function to determine if the candidate key already exists in the probing chain. The voting operation returns a 32-bit (AMD: 64-bit) mask whose bits are raised if the corresponding thread is evaluated as true. If the key already exists, the team of threads determines the valid thread by finding the first significant bit in the voting mask using the `_ffs()` (find first bit set) function. The thread with the first significant bit set is called the winner, and it uses atomic operations to overwrite the value of the existing key with the new provided value. Another voting operation signals the warp to stop probing and exit. The process of detecting if the key already exists repeats only until either the key or an empty bucket is encountered. In the case where O_{curr} is no greater than O_{lim}, i.e. aligns with the number of threads in a CUDA warp or AMD wavefront, each probe only performs one pass, resulting in a highly efficient probing algorithm.

In the event that no matching key is found during probing, the candidate key is inserted into the hashmap. The threads follow the same probing mechanism as before, this time voting for an empty bucket in their respective probing buckets. When an empty bucket is found, the winning thread is determined and it attempts to insert the key into the underlying bucket. If this operation succeeds, an atomic operation is used to replace the existing (unused) value with the candidate value. If the atomic compare-and-swap operation fails, it means another parallel access already added a key to that bucket. The winner's bit is unset in the voting mask and the process is repeated by the next winner. This process continues until either one thread manages to perform a successful insertion or until there are no winners left in the voting mask. In the latter case, the entire warp shifts to the right by a full 32 buckets, and the process is repeated. This design enables the hashmap to overflow, as with the "device-only" insertion interface. If the hashmap tries to overflow beyond the capacity of the underlying bucket container (i.e., the SplitVector holding the data), the program execution is aborted. The insertion mechanism as described above is illustrated in Figure 1.

2.5.2 Retrieval and deletion

The retrieval and deletion operations in the accelerated interface follow the same probing scheme as the insertion method described above. For retrieval the candidate key is probed for and if found its key is returned. If the key is not found during probing or an empty key is found the operation exits without returning a value. For deletion, if the key to be deleted is found during probing it is replaced atomically with a tombstone. If the key is not found during probing or an empty key is met the deletion algorithm returns. This process ensures that the underlying hashmap structure remains intact and that the program execution does not crash due to improper manipulation of the hashmap.

2.5.3 Tombstone cleaning

Cleaning up tombstones using traditional methods involves a complete rehash of the entire hashmap, which can be a computationally expensive operation. To address this issue, we introduce a new tombstone cleaning method in Hashinator that can efficiently remove all existing tombstones while also potentially reducing O_{curr} faster than a complete rehash of the hashmap's contents. This new method leverages the massive parallelism offered by heterogeneous architectures to perform tombstone cleanup in two steps as illustrated in Figure 2. First, we utilize a parallel stream compaction provided by SplitVector (inspired by Billeter et al., 2009) to extract all elements that have overflown beyond their nominal bucket position. This stream compaction kernel also resets all tombstones to empty elements. Then, we use the insertion mechanism as illustrated in Figure 1 to re-insert all extracted elements back into the hashmap. If there are no elements extracted by the stream compaction algorithm, the hashmap is already in a valid state and the process can return. In the performance evaluation section of this work we evaluate the efficiency of our tombstone cleaning method against more commonly found approaches.

3 Results

The computational performance of any tool or software that uses hashing techniques is critical in many applications that require fast data retrieval and storage. In this chapter we evaluate Hashinator's performance by analyzing the insertion, deletion, and retrieval execution times for all three different interfaces as described above. Moreover, we show comparisons against other commonly used containers. For the host and device only interfaces we compare against the standard library version 12.2 `std::unordered_map` and for the accelerated interface we compare against Warpcore's Single Value Hashtable. We perform all of our following NVIDIA tests on an HPC node equipped with an Intel®Xeon ®Gold 6226R CPU,[1] an NVIDIA® A100® 80GB GPU,[2] and 512,GB of RAM. We use CUDA 12.1 and NVCC to compile all code-bases. Specifically since Hashinator is portable between NVIDIA and AMD, we also demonstrate the performance of the accelerated interface on AMD hardware. Those tests are performed on an HPC node equipped with an AMD®EPYC® Trento®7A53 64-Core Processor, an AMD®Instinct®MI250x

1 https://www.intel.com/content/www/us/en/products/sku/199347/intel-xeon-gold-6226r-processor-22m-cache-2-90-ghz/specifications.html

2 https://www.nvidia.com/en-us/data-center/a100/

Candidate Key
H=hash(key)
H+=S
Load buckets[H:H+S]
Team of threads 1 … S
Ballot Key Exists
Overwrite
Return
Ballot Empty Key
Insert Key
Return
Key entry
Empty bucket
Tombstone
Overflown entry by N

FIGURE 1
The accelerated insertion mechanism used in Hashinator. A team of S threads attempts to insert a new key-value pair using voting instructions. The threads first check if the key already exists and, if it does, overwrite its value. If the key is not present in the buckets, the team inserts it into an empty bucket using a voting-based mechanism.

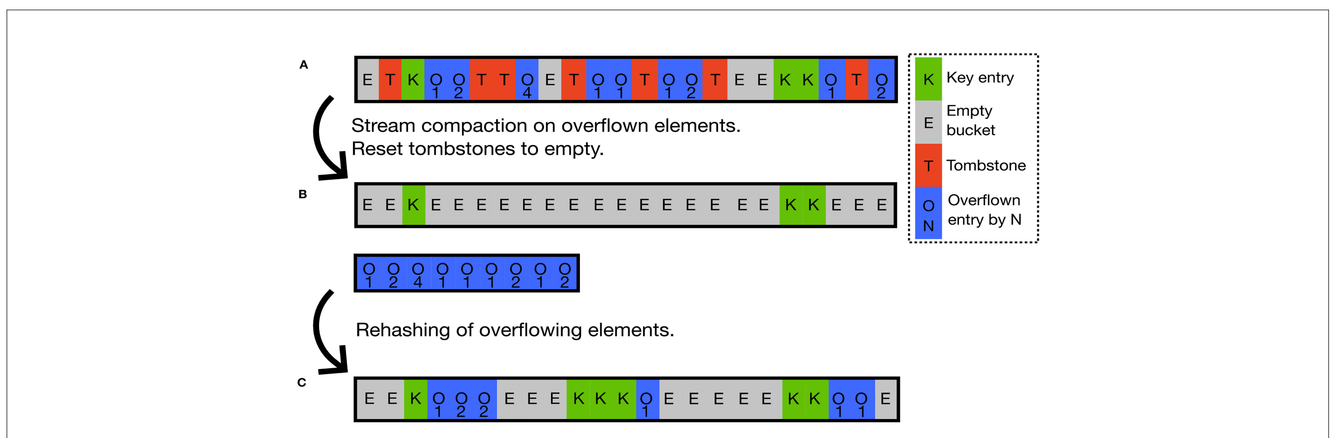

FIGURE 2
Parallel tombstone cleaning in Hashinator. **(A)** Initial state of the hashmap with tombstones marking deleted elements. **(B)** Step 1: A parallel stream compaction extracts all elements that have overflowed beyond the zeroth bucket in their probing chain and resets tombstones to empty elements. **(C)** Step 2: Using the insertion mechanism we reinsert all extracted elements back into the hashmap. This new tombstone cleaning mechanism in Hashinator utilizes parallelism to optimize the tombstone removal without using a full rehash.

GPU[3] and 512 GB of RAM. We use ROCm version 5.3.3 and HIP to compile hashinator on AMD. All timings illustrated in our figures are averaged across 10 consecutive executions, unless specifically stated otherwise.

3.1 Host and device interfaces

In this subsection, we investigate the performance of Hashinator's host and device interfaces by comparing them to the performance of `std::unordered_map`. We conduct four distinct tests and present our findings here. Firstly, we evaluate the performance of raw insertion for an increasing number of 32-bit unique key-value pairs into an empty hashmap of capacity resulting in a final load factor of 0.5. We measure the throughput of Hashinator and compare it with `std::unordered_map` for inserting these elements. Next, we conduct a raw retrieval test where we insert the same unique key-value pairs into each hashmap and measure the throughput of retrieving all of them. Third, we conduct a deletion benchmark where all elements that were previously inserted in the hashmap are now deleted. For Hashinator the key-value pairs are allocated using Unified Memory and are appropriately prefetched to host or device depending on the test case. The prefetching overhead is not included in the depicted throughput measurements.

Deletion can have a significant impact on performance. This is because hashmaps rely on a hashing function to quickly locate the position of an element within the data structure. When a key-value pair is deleted, the hashmap needs to adjust its internal state to maintain proper indexing in case of hash collisions, which can result in a performance degradation. Additionally, some implementations of hashmaps may also require rehashing or resizing of the data structure after a certain number of deletions, further impacting performance. Thus, we also set up a fourth, more realistic test scenario where we insert all the 32-bit unique key-value pairs into each hashmap, retrieve them, and then delete half of the elements. Finally, we immediately re-insert all the elements into the hashmaps and retrieve all the elements again. We measure the execution times for both Hashinator and `std::unordered_map` for this scenario, and evaluate the throughput. We present the results of these three tests in Figure 3. For the device interface we launch CUDA/HIP kernels with maximal launch parameters. Specifically, we utilize 1024 threads per block and launch as many grids as necessary to cover the input size. To provide a baseline for comparison, we also evaluate the device performance using a single thread, which executes tasks in a sequential fashion. In Figure 3 the performance of Warp Accessor methods is also illustrated. Warp Accessor methods can be called from within device-code and make use of the probing scheme used in the "accelerated" interface which is explained in depth in Section 2.

As shown in Figure 3, our device-serial test exhibits lower performance compared to other tests. This can be attributed to the comparatively slower clock speeds of GPUs when compared to CPUs, as well as the inherent inefficiencies of GPUs in executing single-threaded code. However, we observe that the use of a maximum launch configuration in the device test leads to a significant improvement in performance. With this configuration, insertion, retrieval, and deletion operations are executed in parallel by multiple CUDA/HIP threads. The observed performance gap in device mode between the two distinct architectures (NVIDIA and AMD) as illustrated in Figure 3 can be attributed to the utilization of atomic operations, which are necessary for ensuring thread safety during hashmap operations on device. The atomic operations employed by Hashinator appear to incur greater performance penalty on AMD hardware.

3.2 Accelerated interface

In this subsection we benchmark the performance of Hashinator's accelerated interface. We conduct the same four tests as in the previous subsection, this time comparing our results against Warpcore Jünger et al. (2018). Our benchmarks test for insertion, deletion and retrieval performance as well as the performance in a more realistic test case scenario. We use NVIDIA's `Nsight Compute` to measure the execution times of the kernels launched by each hashmap. For the performance metrics of Hashinator on AMD hardware as shown below, we use AMD's `rocProf` profiler. We demonstrate these results in Figures 4–8.

In Figures 4, 5, 6, we present a performance analysis of inserting, deleting and retrieving 32-bit elements at a target load factor of 0.5. This approach allows us to compare the performance of the two hashmaps under similar capacities. We examine the capacity of each hashmap and report their respective memory footprints using shaded bars in both figures. Notably our results demonstrate that Hashinator manages to outperform Warpcore in terms of insertion and retrieval as illustrated in Figure 4, 6 while maintaining a smaller memory footprint, particularly up to the 1 million mark (2^{20}). Warpcore seems to outperform Hashinator for deletion operations of relatively small datasets as illustrated in Figure 5. It is important to note here that our measurements illustrated in Figures 4–8 capture kernel times only, excluding any overhead that may be caused by kernel launches.

As previously noted, both implementations utilize tombstones to manage key deletions. However, it is important to consider that the performance of the hashmaps may be impacted as the tombstone ratio increases. Therefore, we conduct a more realistic test case scenario to further examine the performance of both implementations. To evaluate the performance of the hashmaps, we (similar to the host and device interface tests) populate each implementation with a unique set of random 32-bit key-value pairs until reaching a target load factor of 0.5. Subsequently, we retrieve all keys and immediately delete half of them. Finally, we re-insert all key-value pairs and perform a final retrieval of the data. For Hashinator we perform tombstone cleaning before the second insertion. For Warpcore, rehashes happen whenever deemed necessary. We measure the timings of all the kernels executed by each implementation. We present the resulting throughput values in Figure 7. The illustrated throughput rates

3 https://www.amd.com/en/products/accelerators/instinct/mi200/mi250x.html

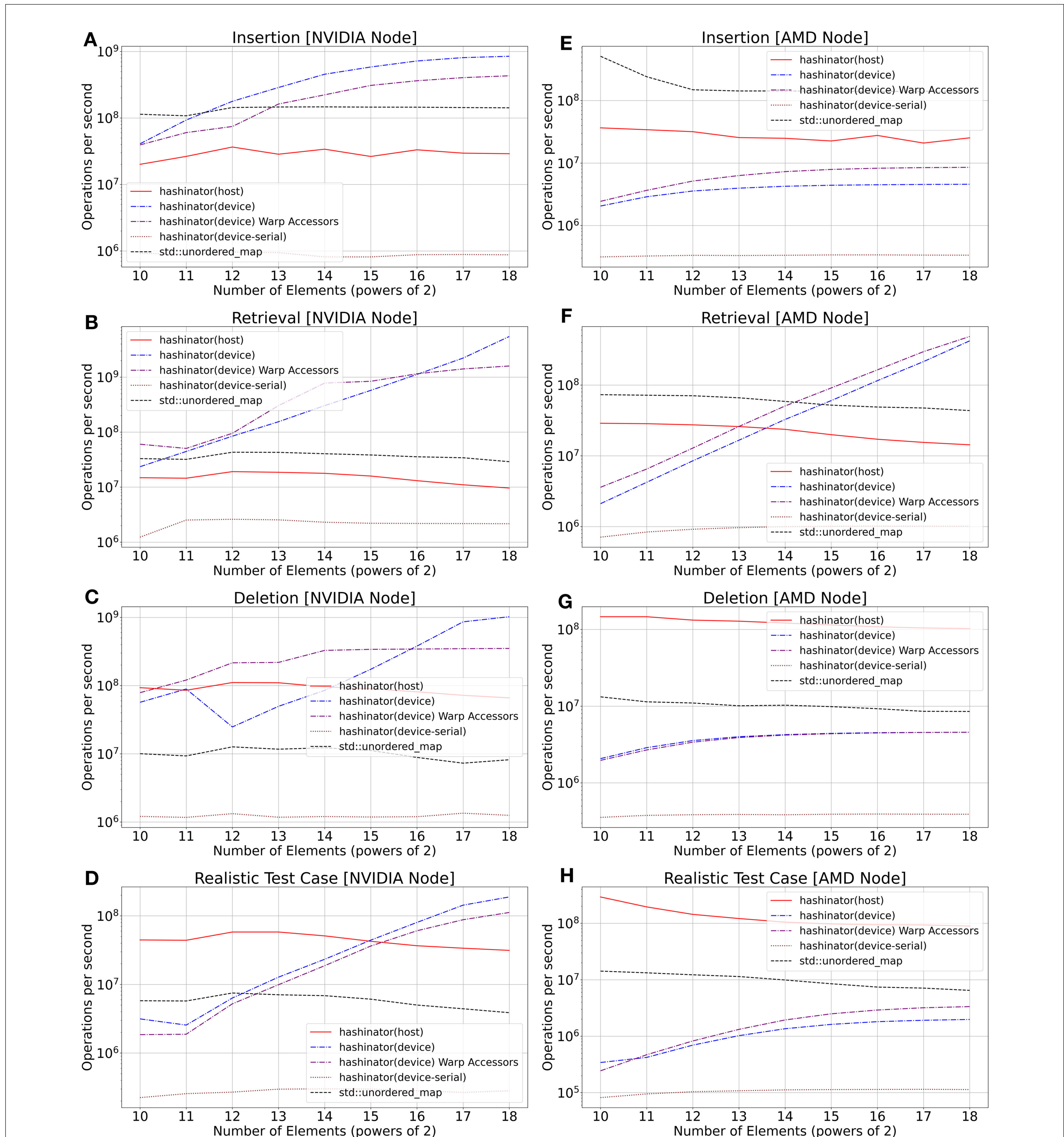

FIGURE 3
Comparison of Hashinator's host and device interfaces against `std::unordered_map`. **(A–D)** illustrate throughput on NVIDIA hardware while **(E–H)** illustrate throughput on AMD hardware. **(A, E)**: Insertion performance. **(B, F)**: Retrieval performance. **(C, G)**: Deletion performance. **(D, H)**: Performance for a more realistic test case where elements are inserted, retrieved, deleted, reinserted and finally retrieved again. The device serial throughput provides insight into the performance of a CUDA/HIP kernel operating in a serialized manner with a single thread. The depicteded results are the median value obtained from a series of 10 consecutive runs for each test.

indicate that Hashinator matches the performance of Warpcore for problem sizes up to 1 million elements. However, for larger problem sizes, Hashinator takes the lead, demonstrating superior performance in the realistic test case. Again, Hashinator maintains a smaller memory footprint for the smaller problem sizes, as illustrated by the shaded bars. This means that Warpcore effectively operates at a much lower load factor for the left-most part of Figure 7.

To fully assess the performance of both implementations, we conduct a test to evaluate their behavior under high load

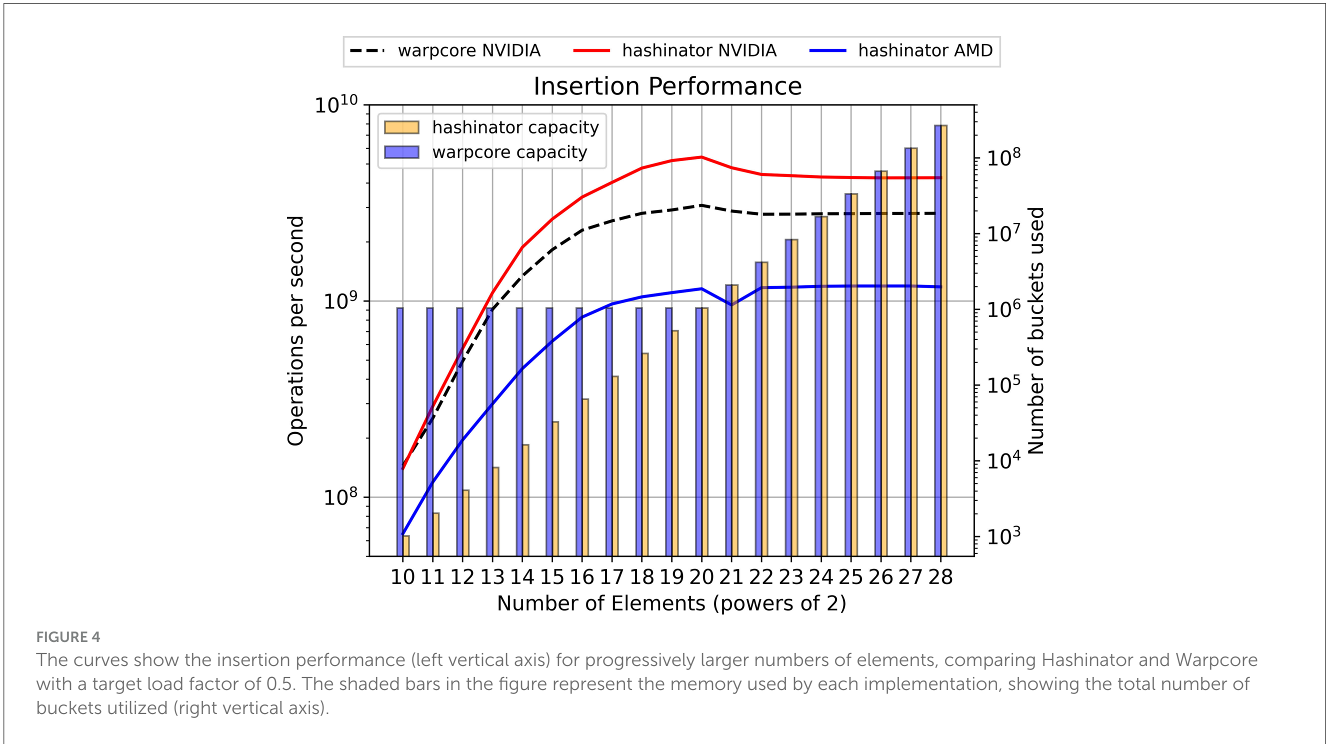

FIGURE 4
The curves show the insertion performance (left vertical axis) for progressively larger numbers of elements, comparing Hashinator and Warpcore with a target load factor of 0.5. The shaded bars in the figure represent the memory used by each implementation, showing the total number of buckets utilized (right vertical axis).

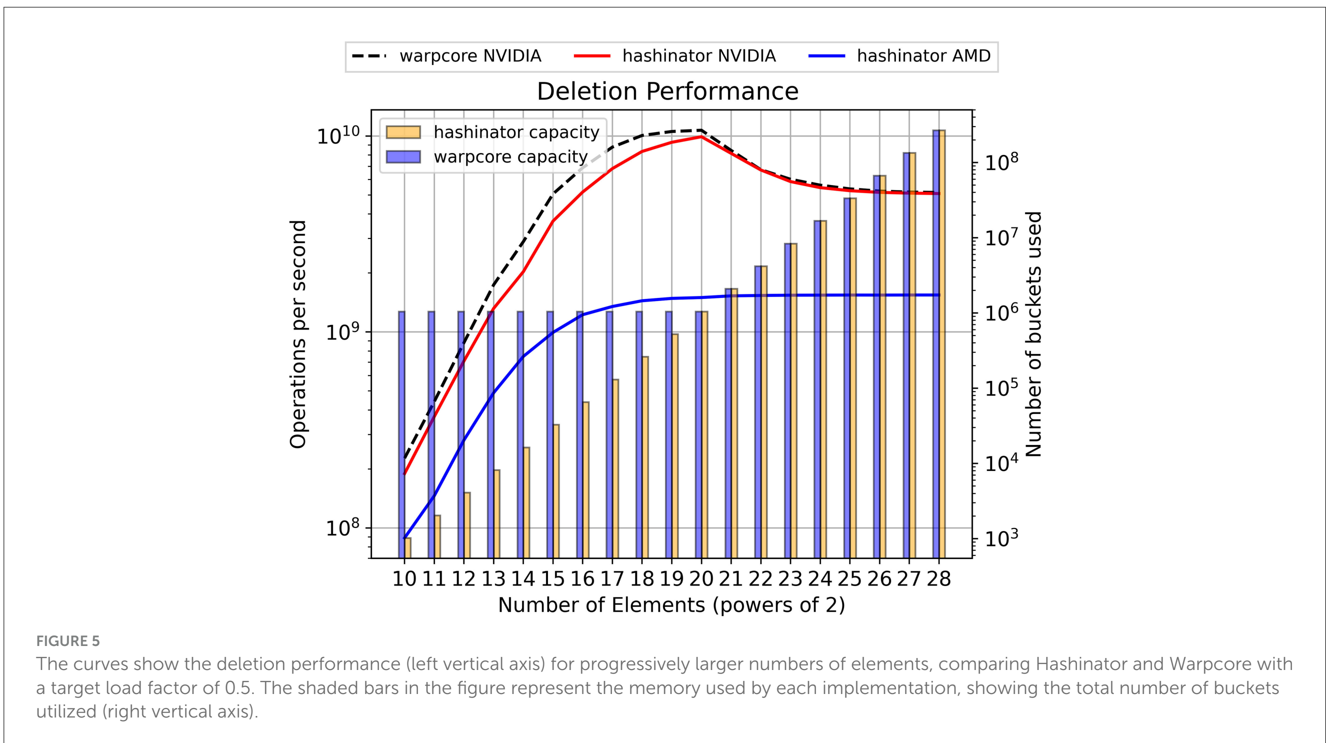

FIGURE 5
The curves show the deletion performance (left vertical axis) for progressively larger numbers of elements, comparing Hashinator and Warpcore with a target load factor of 0.5. The shaded bars in the figure represent the memory used by each implementation, showing the total number of buckets utilized (right vertical axis).

factors. This is particularly crucial for hashmaps because a high load factor can lead to increased collision rates and decreased performance in both insertion and retrieval operations. To evaluate the performance of the hashmaps, we construct the hashmaps with a fixed capacity of 2^{25} elements and we populate each implementation with an increasing amount of random 32-bit key-value pairs until reaching a target load factor of up to 0.95, after which we retrieve all the elements. We report our throughput evaluation in Figure 8. We note that Hashinator manages to outperform Warpcore for load factors up to 90% but incurs more penalty than Warpcore for very high load factors. Warpcore's double hashing scheme outperforms Hashinator's Fibonacci hash function in collision avoidance, but it comes with added instruction overhead.

Finally we demonstrate the performance of Hashinator across NVIDIA and AMD GPUs by normalizing the insertion and

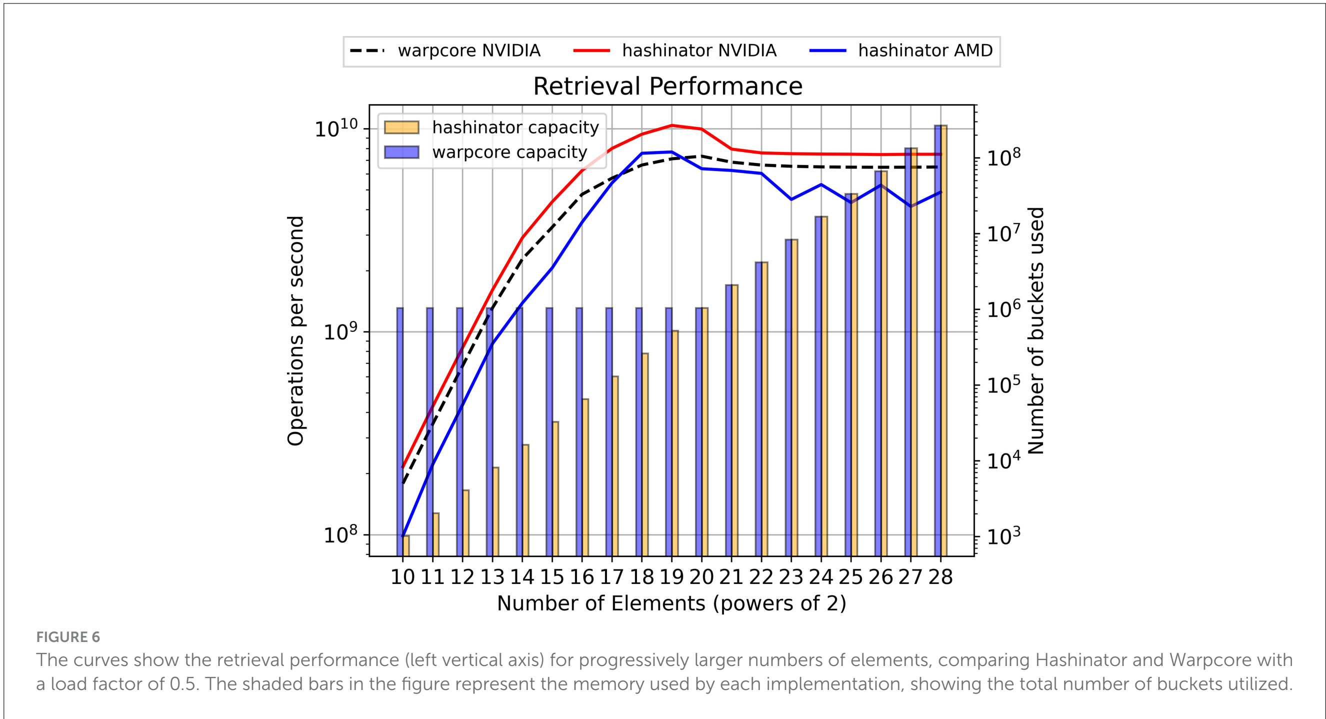

FIGURE 6
The curves show the retrieval performance (left vertical axis) for progressively larger numbers of elements, comparing Hashinator and Warpcore with a load factor of 0.5. The shaded bars in the figure represent the memory used by each implementation, showing the total number of buckets utilized.

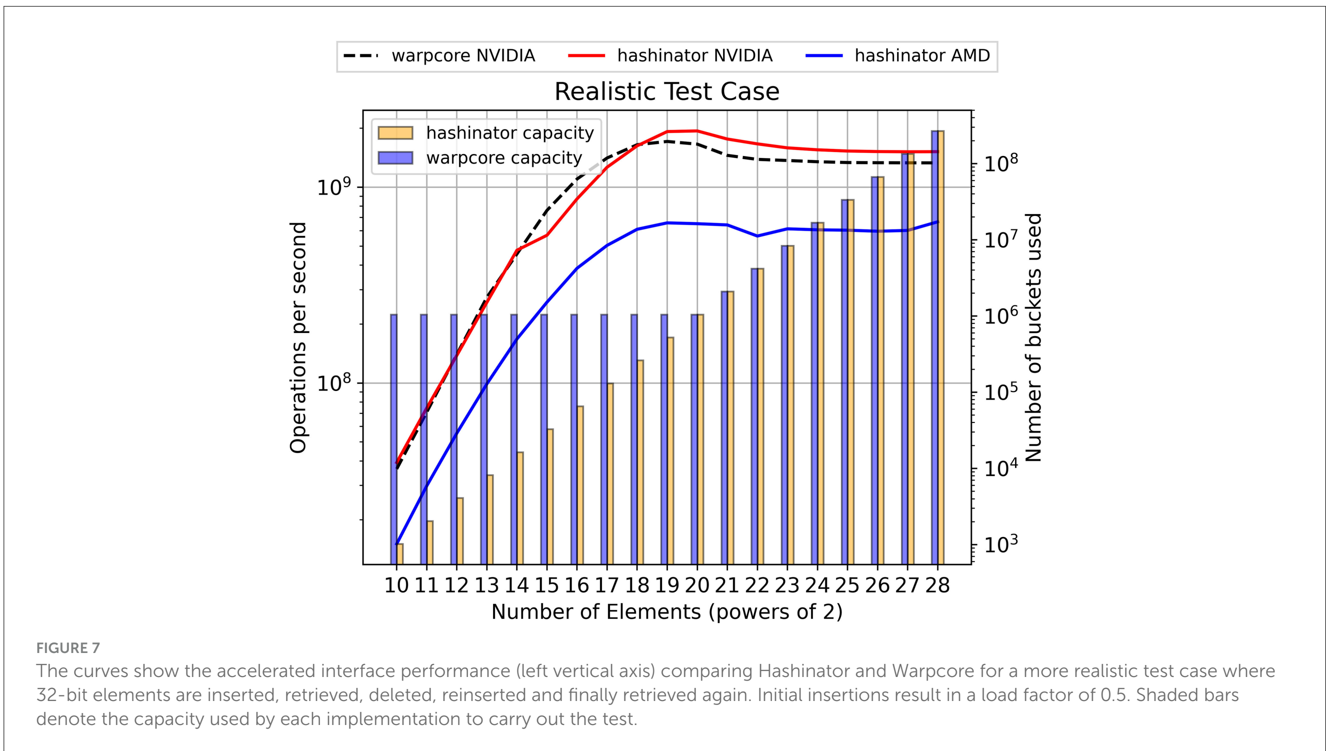

FIGURE 7
The curves show the accelerated interface performance (left vertical axis) comparing Hashinator and Warpcore for a more realistic test case where 32-bit elements are inserted, retrieved, deleted, reinserted and finally retrieved again. Initial insertions result in a load factor of 0.5. Shaded bars denote the capacity used by each implementation to carry out the test.

retrieval throughput, measured in operations per second, by the maximum theoretical memory bandwidth, measured in GB/s, of the respective hardware. The resulting metric is termed the "Figure of Merit" and is illustrated in Figure 9.

We observe that Hashinator exhibits higher efficiency on NVIDIA hardware. This can be attributed to the parallel probing mechanism discussed in Section 2, which both employs independent thread synchronization intrinsics unsupported by AMD hardware and necessary atomic operations to maintain thread safety. As a consequence, unnecessary warp synchronization occurs within the parallel probing algorithm on AMD hardware, leading to decreased efficiency. Insertions incur more penalty on AMD hardware compared to retrievals due to the extra and unavoidable atomic operations required to emplace the new values along with their associated keys in the hashmap in a thread safe manner.

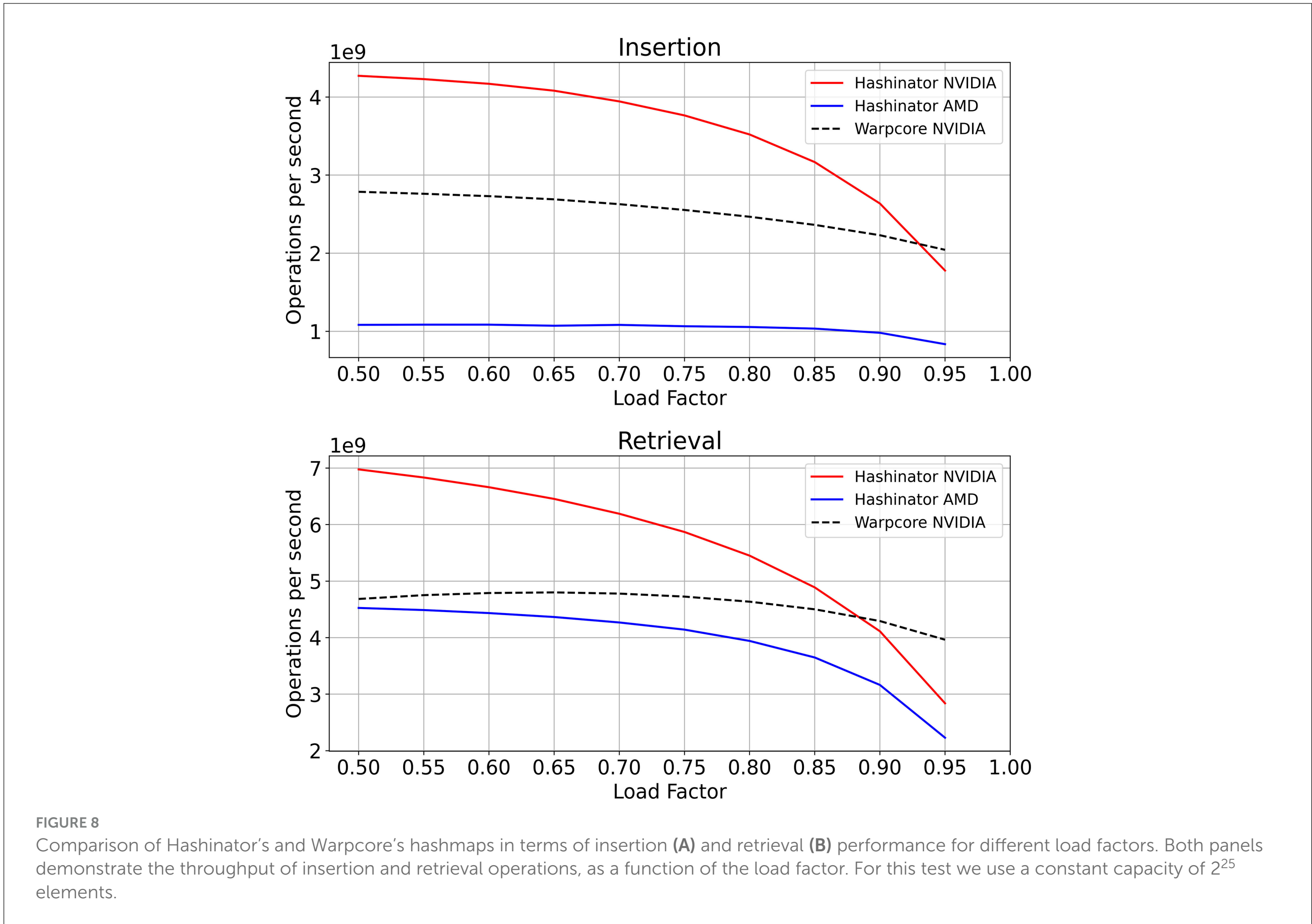

FIGURE 8
Comparison of Hashinator's and Warpcore's hashmaps in terms of insertion **(A)** and retrieval **(B)** performance for different load factors. Both panels demonstrate the throughput of insertion and retrieval operations, as a function of the load factor. For this test we use a constant capacity of 2^{25} elements.

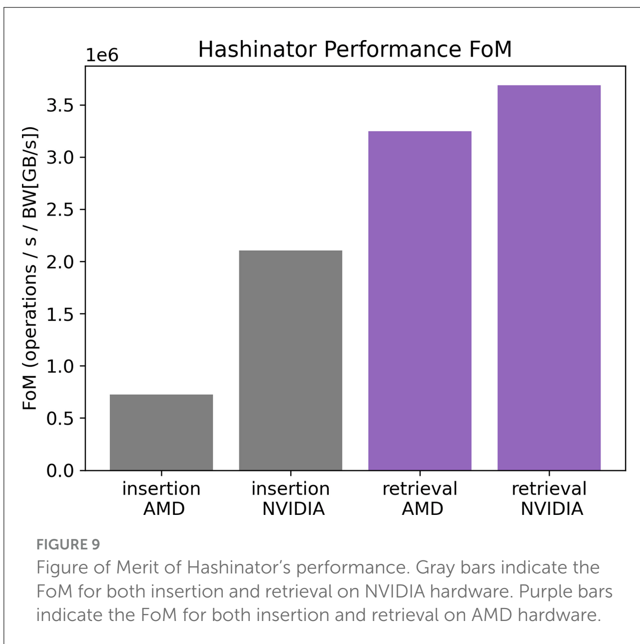

FIGURE 9
Figure of Merit of Hashinator's performance. Gray bars indicate the FoM for both insertion and retrieval on NVIDIA hardware. Purple bars indicate the FoM for both insertion and retrieval on AMD hardware.

3.3 Tombstone cleaning performance

Hashinator is able to efficiently clean up tombstones resulting from multiple key deletions, which significantly improves the probing traversal and speeds up insert and retrieval operations. The problem with tombstones arises when they are not properly cleaned up, as they can lead to a build-up of deleted elements in the hashmap. This can cause performance issues such as slower lookup times and increased memory usage. Traditionally, cleaning up tombstones in a hashmap requires a complete rehash of the hashmap contents, processing only valid keys while ignoring any tombstones. This process is often costly. To assess the efficiency of the tombstone cleaning mechanism in Hashinator, we profile key insertions at high load factors under the presence of tombstones. To prepare the test, we populate the hashmap with an increasing amount of random 32-bit key-value pairs until reaching a load factor close to 90% after rounding down to the nearest integer count of elements. We then delete 20% of the inserted elements. We then measure the time to reinsert half of the deleted elements (which corresponds to 10% of the original dataset) and evaluate the throughput. The last insertion leaves the hashmap with only 1% of vacant buckets as all other buckets are either occupied or tombstones. We perform the test for three different approaches: when tombstones are not cleaned up, when contents are rehashed using a CUDA kernel, and when our tombstone cleaning mechanism is employed. The timings include the time spent in rehashing or tombstone cleaning along with the time spent in the final insertion. Each test is repeated 10 times with the mean timing used for throughput evaluation. This allows us to simulate the presence of tombstones in real-world scenarios and to analyze the effectiveness of Hashinator's

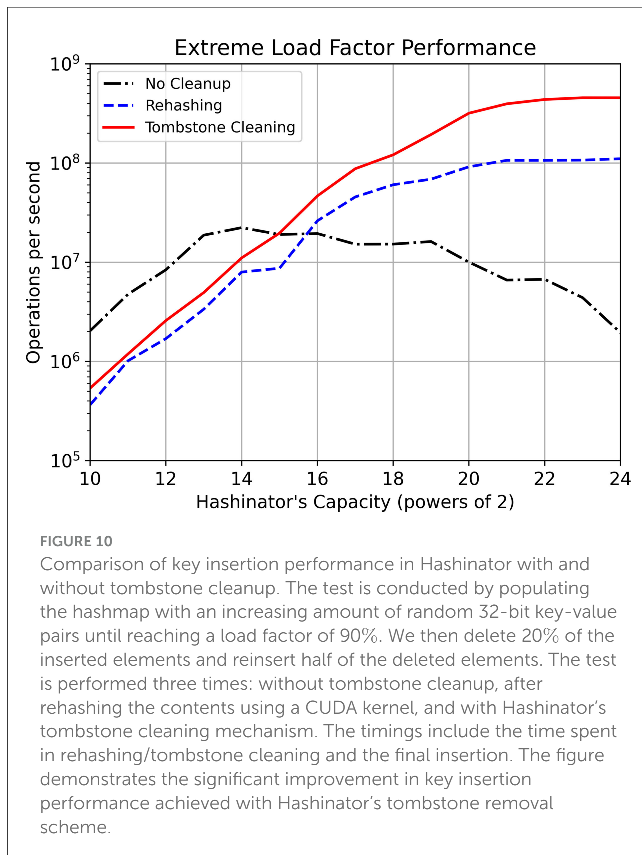

FIGURE 10
Comparison of key insertion performance in Hashinator with and without tombstone cleanup. The test is conducted by populating the hashmap with an increasing amount of random 32-bit key-value pairs until reaching a load factor of 90%. We then delete 20% of the inserted elements and reinsert half of the deleted elements. The test is performed three times: without tombstone cleanup, after rehashing the contents using a CUDA kernel, and with Hashinator's tombstone cleaning mechanism. The timings include the time spent in rehashing/tombstone cleaning and the final insertion. The figure demonstrates the significant improvement in key insertion performance achieved with Hashinator's tombstone removal scheme.

tombstone removal mechanism against other common approaches. The resulting throughput values are presented in Figure 10. We note that the tombstone cleaning method visibly outperforms all the other examined methods in terms of throughput performance, except for very small hashmap sizes.

3.4 Bucket overflow limit parameterization

As previously mentioned, Hashinator uses a parallel warp voting scheme for its probing operations. Since the O_{lim} and hardware warp size are closely intertwined, a natural choice for O_{lim} is 32, the number of threads in a CUDA warp, or 64, the number of threads in an AMD wavefront. However, Hashinator can decouple these concepts by utilizing sub-masking to form teams of threads within a single warp/wavefront, allowing for smaller O_{lim} values. These teams, known as Virtual Warps in Hashinator's terminology, enable independent probing and processing of several elements by a single hardware warp. They closely resemble CUDA's cooperative groups and are used in Hashinator to maintain portability on AMD hardware. The benefit of concurrent processing of several elements outweighs the penalty of higher warp divergence. Thus, finding the optimal balance between these two concepts is crucial for achieving the best performance. Additionally, restricting O_{lim} to 32 would pose a significant penalty for the "host-only" mode, as it would make the "host-only" probing more expensive on average. On the architectures used for this work, we determined that Hashinator performs optimally with 4 Virtual Warps and an O_{lim} of 8. Therefore, for our tests and timings presented in this work, we utilize an O_{lim} of 8 and 4 Virtual Warps, enabling a single hardware warp to process 4 key-value pairs concurrently. We have determined that on AMD hardware hashinator performs optimally with an O_{lim} of 8 and 8 Virtual Warps.

4 Discussion

In this work, we have introduced Hashinator, a novel heterogeneous and portable hashmap that is designed to facilitate the porting of scientific codes to GPU platforms. Hashinator employs a parallel probing scheme inspired by Warpcore (Jünger et al., 2020), and it utilizes the CUDA/HIP Unified Memory model, allowing it to expose its data and functionalities to both host and device code seamlessly. This is unlike the previous state of the art, which allowed hashmaps to operate only on either host or device code exclusively. Furthermore, Hashinator uniquely supports both AMD and NVIDIA hardware, with the HIP/ROCm and CUDA interfaces, and is a lightweight header-only library, easily included in existing software projects. Hashinator offers three distinct modes: the "host-only" mode, the "device-only" mode and the "accelerated mode". Hashinator enables operations to be performed on both host and device code, avoiding unnecessary data transfers, and using internal prefetching methods to avoid page faults. Moreover, Hashinator provides robust portability by maintaining an architecture agnostic codebase which compiles for both NVIDIA and AMD GPUs and for CPU-only use. This feature makes it an ideal choice for developers seeking to accelerate scientific codes on heterogeneous architectures while also simplifying the development process.

Our results demonstrate that Hashinator performs well compared to industry standard implementations in terms of retrieval and insertion throughput when operated in "host-only" and "device-only" modes, as illustrated in Figure 3. Additionally, we conduct a comprehensive set of performance tests to assess the GPU performance of Hashinator's "accelerated" mode. Our throughput analyses of bulk insertion and retrieval rates of Hashinator, depicted in Figures 4, 5, reveal that Hashinator achieves better performance results against other cutting-edge implementations. Furthermore, our results indicate that Hashinator maintains a smaller memory footprint compared to other implementations, particularly for smaller datasets, as illustrated by the capacity bars in Figures 4–7.

Importantly, our results demonstrate that Hashinator incurs little throughput penalty for medium to high load factors, as shown Figure 8 and only incurs performance degradation at very values. This behavior is attributed to the favorable properties of the Fibonacci hash function used by Hashinator, which minimizes the number of collisions that occur. Other hashing schemes such as double hashing, can outperform the Fibonacci hash function in terms of collision avoidance but come with added instruction overhead.

In this work, we also introduced a novel tombstone cleaning mechanism in Hashinator, which leverages SplitVector's stream compaction routines to maintain excellent throughput even in situations where a large number of elements have been deleted, leading to a hashmap overloaded with tombstones. The

effectiveness of our approach is demonstrated in Figure 10, where we compare Hashinator's tombstone cleaning method against other commonly used techniques, showing the considerable speedup achieved by Hashinator's approach. This highlights Hashinator's potential to address common challenges associated with managing tombstones in hashmaps and leading to improved performance and scalability.

In summary, Hashinator provides a valuable tool for heterogeneous high-performance computing by offering an efficient and flexible hashmap implementation that can seamlessly operate on both CPUs and GPUs while at the same time providing cutting edge performance. By facilitating the porting of scientific codes between CPU and GPU architectures, Hashinator enables faster and more efficient computation across a wide range of applications, and fulfills a need previously unanswered. Overall, Hashinator represents a significant advancement in the field of hashmap implementations, with the potential to drive innovation in heterogeneous high-performance computing.

Data availability statement

The source code of Hashinator (Papadakis et al., 2024) is publicly hosted on GitHub at https://github.com/kstppd/hashinator. Further inquiries can be directed to the corresponding author.

Author contributions

KP: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing—original draft, Writing—review & editing. MB: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Writing—original draft, Writing—review & editing, Supervision. UG: Conceptualization, Methodology, Software, Supervision, Writing—review & editing. YP-K: Writing—review & editing, Supervision. MP: Funding acquisition, Project administration, Writing—review & editing, Supervision.

Funding

The author(s) declare that financial support was received for the research, authorship, and/or publication of this article. This work was related to the European High Performance Computing Joint Undertaking (JU) under grant agreement No 101093261 (Plasma-PEPSC). The Academy of Finland (grant nos. 336805, 339756, 339327, 347795, 345701 and in particular for MB's work, 335554) is acknowledged.

Acknowledgments

The work presented in this paper would not have been possible without the high-performance computing resources provided by the Finnish IT Center for Science (CSC). The verification of Hashinator was conducted on Puhti and Mahti supercomputers and the performance tests for AMD hardware presented in this paper were run on the LUMI supercomputer. The authors also wish to acknowledge the Oregon Advanced Computing Institute for Science and Society (OACISS). The performance tests for NVIDIA hardware were run on a Voltar supercomputing node.

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

Awad, M. A., Ashkiani, S., Porumbescu, S. D., Farach-Colton, M., and Owens, J. D. (2022). Better GPU hash tables. *arXiv [Preprint]*. arXiv: 2108.07232. doi: 10.48550/arXiv.2108.07232

Awad, M. A., Ashkiani, S., Porumbescu, S. D., Farach-Colton, M., and Owens, J. D. (2023). "Analyzing and implementing GPU hash tables," in *SIAM Symposium on Algorithmic Principles of Computer Systems* (Florence: APOCS23), 33–50.

Barnat, J., and Ročskai, P. (2008). Shared hash tables in parallel model checking. *Elect. Notes Theoret. Comp. Sci.* 198, 79–91. doi: 10.1016/j.entcs.2007.10.021

Billeter, M., Olsson, O., and Assarsson, U. (2009). "Efficient stream compaction on wide SIMD many-core architectures," in *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY: ACM).

Burau, H., Widera, R., Honig, W., Juckeland, G., Debus, A., Kluge, T., et al. (2010). PIConGPU: A fully relativistic particle-in-cell code for a GPU cluster. *IEEE Trans. Plasma Sci.* 38, 2831–2839. doi: 10.1109/TPS.2010.2064310

Chen, M., Xiao, Q., Matsumoto, K., Yoshida, M., Luo, X., and Kita, K. (2013). "A fast retrieval algorithm based on fibonacci hashing for audio fingerprinting systems," in *Advances in Intelligent Systems Research* (Amsterdam: Atlantis Press).

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001a). *Introduction to Algorithms*. London: MIT Press, 221–252.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001b). *Introduction to Algorithms*. London: MIT Press, 277–282.

Freiberger, M. (2012). "The agile library for image reconstruction in biomedical sciences using graphics card hardware acceleration," in *Technical report, Karl-Franzens Universität Graz, Technische Universität Graz* (Graz: Medizinische Universität Graz).

Jünger, D., Hundt, C., and Schmidt, B. (2018). "WarpDrive: Massively parallel hashing on multi-GPU nodes," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Vancouver, BC: IEEE).

Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., and Schmidt, B. (2020). *Warpcore: A Library for Fast Hash Tables on GPUS.*

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3, Sorting and Searching*. Boston: Addison Wesley.

Lessley, B., and Childs, H. (2020). Data-parallel hashing techniques for GPU architectures. *IEEE Trans. Parallel Distrib. Syst.* 31, 237–250. doi: 10.1109/TPDS.2019.2929768

Li, W., Jin, G., Cui, X., and See, S. (2015). "An evaluation of unified memory technology on NVIDIA GPUs," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (Shenzhen: IEEE).

Liu, D., and Xu, S. (2015). Comparison of hash table performance with open addressing and closed addressing: an empirical study. *IJNDC* 3:60. doi: 10.2991/ijndc.2015.3.1.7

Papadakis, K., Battarbee, M., and Widera, R. (2024). *fmihpc/hashinator: v1.0.1 Hashinator stable*. Zenodo. doi: 10.5281/zenodo.11396297

Purcell, C., and Harris, T. (2005). "Non-blocking hashtables with open addressing," in *Lecture Notes in Computer Science*. Berlin: Springer Berlin Heidelberg, 108–121.