GPHash: An Efficient Hash Index for GPU with Byte-Granularity Persistent Memory

Menglei Chen, Yu Hua*, Zhangyu Chen, Ming Zhang, Gen Dong
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

GPU with persistent memory (GPM) enables GPU-powered applications to directly manage the data in persistent memory at the byte granularity. Hash indexes have been widely used to achieve efficient data management. However, conventional hash indexes become inefficient for GPM systems due to warp-agnostic execution manner, high-overhead consistency guarantee, and significant bandwidth gap between PM and GPU. In this paper, we propose GPHash, an efficient hash index for GPM systems with high performance and consistency guarantee. To fully exploit the parallelism of GPU, GPHash executes all index operations in a lock-free and warp-cooperative manner. Moreover, by using CAS primitive and slot states, GPHash ensures consistency guarantee with low overhead. To further bridge the bandwidth gap between PM and GPU, GPHash caches hot items in GPU memory while minimizing the overhead for cache management. Extensive evaluations on YCSB and real-world workloads show that GPHash outperforms state-of-the-art CPU-assisted data management approaches and GPM hash indexes by up to 27.62×.

1 Introduction

With dramatic increases in computing throughput and memory bandwidth, GPUs have been widely used in various applications to accelerate computation, e.g., deep neural networks training/inference [24, 42], databases [18, 19, 53], and scientific computing [35, 58, 60, 76]. These GPU-powered applications are driven by large-scale data. For instance, by storing TB-scale embedding vectors, deep recommendation systems achieve higher recommendation accuracy and richer feature representations [2, 73]. Constrained by limited memory capacity and the volatile property, it is not feasible for GPU applications to store large-scale essential data in GPU memory. To accommodate data of ever-increasing scale while ensuring data reliability, GPU-based applications store the data in large-capacity persistent storage devices (e.g., SSD) [21] and rely on the CPU to manage the data in the storage (e.g., storing/searching a given key).

However, *CPU-assisted data management* involves time-consuming data transmission between GPU and CPU, while introducing extra CPU consumption and interference with other CPU applications. To alleviate data transmission overhead and eliminate extra CPU consumption, GPUDirect Storage (GDS) technology [48] provides a direct path between GPU memory and block storage devices. While GDS enables fast transmission between GPU memory and storage devices for large files, unfortunately, it only provides *block-level* interfaces, which do not meet the programming and high-performance requirements that necessitate *byte-granularity* access for indexes [36, 44, 51, 63, 77].

In order to support fine-grained direct access to storage devices from GPU kernels, the GPU with persistent memory (GPM) model [51, 52] exploits the byte-addressability of persistent memory (PM) [14, 59]. GPM can be implemented by leveraging the unified virtual addressing (UVA) technology [46] to map the PM onto the GPU's virtual address space. GPM enables applications to directly manage data on PM at the byte granularity without CPU involvement, thus decreasing the costs of data management. Meanwhile, such *GPM-enabled data management* can also benefit from the high parallelism of GPU.

Hash indexes support parallel accesses to the fine-grained data with constant-scale single-point query performance, which hence are widely used to achieve efficient data management [3, 12, 44, 50, 63, 77]. GPM-enabled hash indexes become promising to boost data management performance in GPU-powered applications. For example, deep recommendation systems benefit from GPM hash indexes to search target embedding vectors during each inference (e.g., *embedding_lookup* in Microsoft recommenders [41] and Facebook DLRM [20]). However, to the best of our knowledge, there is no existing hash index designed for GPM. In practice, simply porting existing hash indexes to GPM systems unfortunately suffers from the following challenges.

1) Performance degradation due to warp-agnostic execution manner. The basic unit of scheduling in the GPU is *warp*, which consists of 32 GPU threads [45]. When threads within the same warp take different execution paths due to conditional branching, they would be serialized and only a small portion can execute instructions, called warp divergence [56, 66]. When the threads in a warp access the same cache line, the GPU hardware coalesces these accesses into a single cache access, which delivers high throughput [3, 34]. However, hash indexes typically make each thread independently perform index operations without the awareness of the warp-based execution features. Such warp-agnostic execution manner suffers from severe warp divergence and uncoalesced memory accesses, thus leading to severe performance degradation. Even worse, some hash indexes adopt lock-based designs, which exacerbate the contentions among thousands of concurrent GPU threads.

2) High overheads for crash consistency guarantee. It is important to guarantee crash consistency for the PM data managed in the GPM hash indexes. However, it is non-trivial to achieve this, since the unit of atomic memory write on PM (e.g., 8 bytes for 64-bit CPUs) is limited by the width of memory bus [49, 77]. Hence, when writing data whose size exceeds this atomic unit, a system crash may result in partial updates and cause data inconsistency, thus leading to unexpected errors and incorrectness in GPU applications. Although logging [22] and copy-on-write (CoW) [25] techniques can be used to guarantee crash consistency for data, these techniques introduce extra write overhead to PM.

3) Huge bandwidth gap between PM and GPU. While the read bandwidth of GPU memory in NVIDIA V100 can reach 900 GB/s [47], the read bandwidth of PM is only 39.6 GB/s in six Intel Optane DC PMMs [29, 69]. When performing thousands of concurrent index operations, bandwidth-hungry GPU kernels suffer from PM's limited bandwidth [29, 47, 52]. As a result, the naive GPM indexes fail to fully leverage the GPU to boost the performance due to massive bandwidth-limited PM accesses from GPU kernels when concurrently performing index operations.

In order to efficiently address the mentioned problems, we propose GPHash[1], an efficient hash index for GPM systems. GPHash fully leverages the high parallelism of GPU and provides a low-overhead consistency guarantee while further increasing the performance with a frozen-based bucket cache. Specifically, this paper makes the following contributions:

• ***GPU-conscious and PM-friendly hash table structure.*** To achieve high performance and efficiency of GPHash, the hash table structure of GPHash is designed to be GPU-conscious and PM-friendly. GPHash features *one-shot warp access*, i.e., using 32 threads in a warp to concurrently access the buckets to find the target key, which exploits the parallelism of a warp and thus delivers high performance. Moreover, GPHash facilitates coalesced memory accesses by directly placing keys in the hash table slots. Furthermore, GPHash achieves high memory efficiency by employing slot associativity, inter-level sharing, and multi-hash locations.

• ***Lock-free concurrency control with crash consistency guarantee.*** GPHash employs lock-free index operations to mitigate the contentions among threads. These operations are executed in a warp-cooperative manner via warp-level instructions to alleviate warp divergence. Moreover, by using the compare-and-swap (CAS) primitive and the slot state (indicating whether an index slot is under insertion), the index operations are implemented as log-free, which further reduces the overheads for crash consistency guarantee.

• ***Frozen-based bucket cache.*** To bridge the bandwidth gap between PM and GPU, we aim to reduce PM access by caching hot items in GPU memory. However, traditional list-based cache management causes high contentions among GPU threads when massive threads concurrently query and update the lists. To address this challenge, we propose a frozen-based bucket cache design, called BktCache, to minimize the overhead of cache management. Since the items in the same bucket are concurrently accessed, BktCache caches items at the bucket granularity. BktCache periodically identifies and fetches hot buckets (i.e., loading phase) while keeping the membership of the cached buckets unchanged (i.e., frozen phase) between the loading phases. We employ a concurrent loading scheme to reduce the overhead of loading BktCache.

• ***Real implementation and extensive evaluation.*** We have implemented and evaluated GPHash using YCSB and real-world workloads. Extensive experimental results show that GPHash outperforms state-of-the-art CPU-assisted data management approaches and GPM hash indexes by up to 27.62× and 17.42×, respectively.

It is worth noting that our GPHash can be efficiently adopted in GPM systems that are implemented via Compute Express Link (CXL) technology [16, 59], as long as these GPM systems provide direct accesses to PM from GPU kernels using `load/store` instructions. Implementing hash indexes on these GPM systems also requires efficient orchestration of GPU thread execution, low-overhead crash consistency guarantee, and mitigation of the bandwidth gap between GPU memory and CXL. GPHash has addressed these challenges via GPM-friendly hash table design, lock-free and log-free operations, and frozen-based cache. Hence, GPHash is still efficient for CXL-based GPM systems.

2 Background and Motivation

2.1 GPU with Persistent Memory

2.1.1 Persistent Memory

Persistent memory (PM) provides persistence, large capacity, byte-addressability, and near-DRAM performance. The access latency of PM is 3-10 times of DRAM, and the bandwidth of PM is 1/6 of DRAM [29, 69]. Existing schemes mainly focus on re-configuring architectures or re-designing system software for the PM to gain full benefits [10, 11, 27, 37, 62, 64, 65, 68, 69]. The data persistence is guaranteed via executing flush and fence instructions. Although Intel would discontinue Optane PM [23], the features and properties of persistent memory can be achieved via other memory technologies and

[1] We have released the open-source codes for public use in `https://github.com/LighT-chenml/GPHash`.

techniques [15, 57, 59], e.g., Samsung memory-semantic CXL SSD [59]. By leveraging CXL interconnect technology [16] and a built-in DRAM cache, the memory-semantic SSD can effectively act as persistent memory to build GPM systems.

2.1.2 The Parallelism of GPUs

GPUs achieve high parallelism by using thousands of GPU threads [47]. A modern GPU contains hundreds of streaming multiprocessors (SMs). Each SM contains many SIMD units. An SIMD unit consists of several lanes, which concurrently execute the same instruction on different data items. The smallest execution unit is the GPU thread, which is mapped to an SIMD lane. GPU programming languages (e.g., CUDA) require programmers to arrange threads in the GPU hardware hierarchy. In CUDA, 32 threads form a *warp*, which is the smallest scheduled unit in the GPU [46]. However, the threads in the same warp will be serialized if executing different instructions due to branch instructions, called *warp divergence*. When the warp divergence occurs, only a small portion of threads in the warp is executing instructions, thus limiting the parallelism of the warp. When a warp loads/stores data that fall in the same GPU cache line (i.e., 128 bytes), the GPU hardware coalesces them into a single cache access, called *coalesced memory access*. In contrast, uncoalesced memory accesses lead to multiple cache accesses, causing performance degradation. Several warps form a thread block, which is scheduled to run on an SM. The grid is the largest execution unit, which consists of several thread blocks that execute a GPU kernel.

2.1.3 Efficient Data Management for GPU with PM

After over a decade of phenomenal growth in computing throughput and memory bandwidth of GPUs, the GPU-powered applications have become prevalent in various domains [24, 42, 58]. These applications are always in pursuit of handling larger amounts of data for better efficiency. For example, recent studies show that the embedding vectors in deep recommendation systems will scale to dozens of TBs for higher recommendation accuracy and richer feature representations [2, 73]. Such an unprecedented data scale requires highly efficient data management.

To meet the demands of ever-increasing data scale and reliability, applications typically store the data in large-capacity persistent storage devices (e.g. SSD and PM). In general, GPU applications rely on the CPU to manage the data in the storage, called *CPU-assisted data management*. The blue line in Figure 1 shows the data path of the CPU-assisted data management. Specifically, the GPU applications need to first transfer data from GPU memory to CPU memory and then rely on the CPU to write (read) the data into (from) storage devices, e.g., updating the value of a given key. The CPU-assisted approach suffers from time-consuming data transmission between GPU and CPU, while introducing extra CPU consumption and interference with other CPU applications.

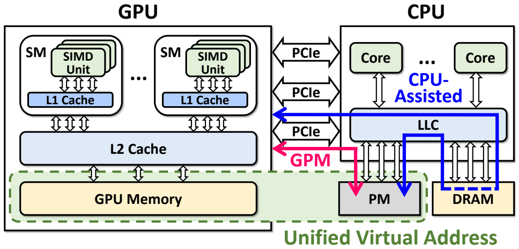


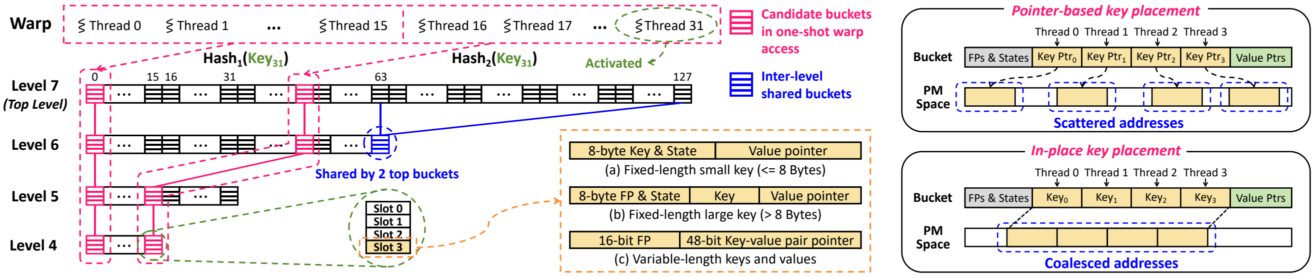
Figure 1: The overview of GPM system model.

To mitigate data transmission overhead and eliminate extra CPU consumption, NVIDIA GPUDirect Storage (GDS) technology [48] provides a direct path between GPU memory and block storage devices via direct memory access (DMA). Unfortunately, while GDS shows great capability in transmitting large files between GPU memory and storage devices, it only provides *block-level* interfaces, which mismatches the *byte-granularity* requirement for data structures such hash indexes, thus incurring high overheads [51]. Recent works propose a system model, called GPM (i.e., GPU with Persistent Memory) [51, 52], which exploits the properties of the PM [14, 59]. GPM can be implemented by: (1) mapping the PM onto the GPU's virtual address space via unified virtual addressing (UVA) technology [46]; (2) ordering data using system-scoped threadfence (e.g., `threadfence_system()` in CUDA); (3) turning DDIO off to bypass the last level cache (LLC) to guarantee persistence. The red line in Figure 1 shows the data path of the GPM-based data management. By leveraging the byte-addressability of PM, GPM enables GPU-based applications to directly access PM at the byte granularity without CPU involvement, thus decreasing the costs of data management. GPM-enabled data management also benefits from the high parallelism of GPU.

2.2 Hash Indexes on GPM Systems

Hash indexes have been widely used in many data management systems [3, 12, 44, 63, 77]. GPU-powered applications can benefit from GPM-enabled hash indexes to achieve efficient data management [32,67,72]. Porting existing PM-based and GPU-based hash indexes to GPM systems is an intuitive way to implement GPM hash indexes. Unfortunately, these indexes suffer from the following challenges.

Challenge 1: *Warp-agnostic execution manner causes severe performance degradation.* In general, hash indexes leverage multiple threads to concurrently perform index operations for high throughput, where each thread independently performs index operations. However, such warp-agnostic execution manner suffers from severe warp divergence and uncoalesced memory accesses in the context of GPU. Specifically, when the threads in the same warp perform different operations without intra-warp communications, they passively encounter warp divergence due to branch instructions. Moreover, most hash indexes only store the pointers of keys (or key-value pairs) to reduce the storage overhead [12,36,63,78]. However, when the threads in a warp access keys in parallel, the addresses of these keys are scattered, hence leading to uncoalesced memory accesses. In addition, some hash indexes



(a) GPU-conscious and PM-friendly hash table

(b) In-place key placement

Figure 2: The hash table structure of GPHash *(using 2 hash locations and 4-level buckets with 4-way associativity as an example)*.

use lock-based designs, which exacerbate the contentions among thousands of concurrent GPU threads and even cause dead-lock when the threads in a warp acquire the same lock.

Challenge 2: ***Guaranteeing data consistency in the presence of crashes introduces high overheads.*** Since GPM hash indexes directly manage data in PM, it is important to guarantee data consistency in the presence of crashes, which however is non-trivial. The size of atomic memory write of PM is limited by the memory bus width (e.g., 8 bytes for 64-bit CPUs) [49, 70, 77]. As a result, if a system failure occurs before completing writing the data whose size is larger than 8 bytes, the data will be corrupted. Logging and copy-on-write (CoW) techniques are widely used to guarantee crash consistency for data larger than 8 bytes [33, 75]. When using logging, indexes store the old data (undo logging) or new data (redo logging) into logs and then write the new data in place. When using CoW, we copy the old data to a newly created space and perform the updates on the copy, and then atomically modify the 8-byte pointer to point to the new data. However, these techniques introduce high write overhead.

Challenge 3: ***Huge bandwidth gap between PM and GPU limits the utilization of GPU's high parallelism.*** There is a significant bandwidth gap between PM and GPU memory. For example, while the bandwidth of GPU memory in NVIDIA V100 can reach 900 GB/s [47], the read and write bandwidths of PM are only 39.6 GB/s and 13.8 GB/s for six Intel Optane DC PMMs, respectively [29, 69]. Hence, when performing massive concurrent index operations, the bandwidth-limited PM fails to efficiently handle massive concurrent accesses from the bandwidth-hungry GPU kernels [29, 47, 52]. Such a huge bandwidth gap between PM and GPU hinders fully utilizing the high parallelism of GPU. Moreover, some schemes, such as cuckoo hashing [34] and linked-list-based hashing [3], incur extra PM accesses to handle hash collisions, which further deteriorates the bandwidth problem.

3 The GPHash Design

We propose GPHash, an efficient hash index for GPU with persistent memory.

3.1 The Hash Table Structure of GPHash

In the context of GPM systems, an efficient hash table needs to fully utilize the high parallelism of GPU and minimize extra PM writes for handling hash collisions. To this end, we propose a GPU-conscious and PM-friendly hash table that supports one-shot warp access with minimal uncoalesced memory accesses while keeping high memory efficiency. As shown in Figure 2(a), GPHash uses a level-based hash table that consists of multiple levels, where level i contains 2^i buckets. GPHash embraces the following design decisions.

1) Slot associativity. Like many hash schemes, in our GPHash, each bucket contains multiple slots, each of which stores a key-value item. Figure 2(a) shows an example of GPHash with 4-way slot associativity, where each bucket has 4 slots. By leveraging slot associativity, each bucket can handle multiple hash collisions without any movements and extra PM writes. Moreover, slot associativity is friendly for exploiting GPU's parallelism since multiple slots can be concurrently accessed.

2) Inter-level sharing. GPHash leverages inter-level sharing [12, 77] to handle more hash collisions for higher memory efficiency. Only the buckets in the top level can be addressed by hash functions. The buckets in the other levels are shared by several buckets in the top level and each bucket in the top level has multiple sharing buckets. For example, in Figure 2(a), each bucket in level 5 is shared by 4 buckets in level 7 (i.e., the top level), while each bucket in level 7 has 3 sharing buckets (in levels 4, 5, 6 respectively). We insert a new item into the less-loaded bucket among the addressed bucket and its sharing buckets (e.g., 4 sharing buckets in total for GPHash with 4 levels). With the aid of inter-level sharing, GPHash can handle more hash collisions and improve the efficiency of load balance, thus enabling higher memory efficiency.

3) Multiple hash locations. Prior study [43] reveals that enabling each key to have multiple choices for its storage locations leads to exponential improvements in memory efficiency over one choice. Based on this observation, GPHash uses several hash functions to compute multiple hash locations for each key. As shown in Figure 2(a), by using 2 hash functions, there are 8 candidate buckets for each new item to insert, which further improves memory efficiency.

4) One-shot warp access. By leveraging the parallelism of the warp, GPHash can access all slots of candidate buckets for a given key at one time. As mentioned above, GPHash leverages slot associativity, inter-level sharing, and multiple hash locations, to achieve higher memory efficiency. However, an insertion operation needs to probe all candidate buckets for better load balance while a search operation also has to access all candidate buckets in the worst case. Therefore, for CPU-based hash schemes, there is a trade-off between high memory efficiency and high performance. However, for GPM systems, hash indexes can exploit the high parallelism of GPU to achieve both high memory efficiency and high performance. By using appropriate configurations, we probe all slots of candidate buckets with *one-shot warp access*. As shown in Figure 2(a), 32 threads in a warp can concurrently access all 32 slots of candidate buckets for a given key. Note that the one-shot warp access feature is based on the warp-cooperative execution manner (§3.2.1).

5) In-place key placement. While many hash schemes store the pointers of keys to reduce storage overheads for empty slots, these hash schemes exhibit uncoalesced memory accesses when GPU threads in a warp concurrently access keys, as shown in Figure 2(b). To address this problem and facilitate coalesced memory accesses, GPHash leverages in-place key placement, which directly stores the keys in slots. The keys of the same buckets hence can be stored in continuous coalesced addresses. When the threads in a warp access keys in the same buckets, these accesses can be coalesced. Since the values are typically accessed by a dedicated thread in a warp, we still store the pointer of the value.

Specifically, for the item with the fixed-length key, we store the key and the pointer of the value. For the item with the variable-length key, we store the pointer of the key-value pair, which is consistent with prior schemes [12, 78]. Furthermore, by employing fingerprints (FP, i.e., a part of hash value) [36, 49], GPHash can avoid the unnecessary reads for full keys if the fingerprints of keys are different. To support lock-free and log-free operations on the fixed-length keys, two values in the key/fingerprint value ranges are reserved as the slot states, which are used to indicate whether the slot is empty or under insertion (detailed in §3.2.3). The concern about in-place key placement is extra storage overheads for empty slots. Prior works [4,5,71,74] have shown that key-value items whose sizes are smaller than 128 bytes dominate in large-scale key-value stores and GPU-specific workloads. Therefore, the storage overheads of empty slots are limited since GPHash also provides high load factors.

Put them all together, the structure of GPHash is shown in Figure 2(a), which contains multi-level buckets with K-way slot associativity. The blue buckets indicate the shared buckets while the pink buckets show the candidate buckets in a warp access. The structure of GPHash is simple and efficient on GPM systems, exhibiting the following strengths:

- ***GPU-friendly.*** GPHash probes all slots of candidate buckets within one-shot warp access, which is beneficial for utilizing GPU's parallelism. Meanwhile, GPHash facilitates the coalesced memory accesses with in-place key placement.
- ***Write-optimized.*** Each insertion operation in GPHash only involves a constant number of buckets without any data movement from/to other buckets or linking new buckets. In tandem with coalesced memory accesses, GPHash minimizes PM writes.
- ***Memory-efficient.*** By embracing slot associativity, inter-level sharing, and multiple hash locations, GPHash can tolerate more hash collisions with better load balance. GPHash provides a high load factor that is up to 92% (§4.2)

3.2 Lock-Free Concurrency Control with Crash Consistency Guarantee

As discussed in §2.2, lock-based designs suffer from severe contentions among thousands of concurrent GPU threads and even cause dead-locks. Hence, it is critical for GPM hash indexes to achieve lock-free concurrency control. Moreover, GPM hash indexes need to minimize overheads for the crash consistency guarantee. Since compare-and-swap (CAS) primitive (e.g., `atomicCAS()` in CUDA [46]) can atomically update an 8-byte content, GPHash leverages its atomicity in tandem with slot states to enable lock-free concurrency control with crash consistency guarantee. Besides, to mitigate warp divergence, GPHash executes index operations in a warp-cooperative manner.

3.2.1 Warp-cooperative Execution Manner

In order to mitigate the warp divergence, GPHash performs index operations at the *warp* granularity. As shown in Figure 2(a), the index operation assigned to thread 31 is *activated*, and all 32 threads in the warp cooperate to complete this activated index operation. We use CUDA's warp-level instructions [45] to perform intra-warp communication among threads. Specifically, the `ballot` instruction is used to find the threads whose assigned index operations have not been completed. Among these threads, we then select the thread with minimal thread number and activate its assigned index operation, and the thread is called the *activated thread*. Moreover, we use the `shfl` instruction to broadcast a variable to all threads in the warp.

When cooperating to complete the activated operation, the threads in a warp concurrently perform processes that can be performed in parallel. For instance, to find the activated key, the activated thread first broadcasts the activated key via the `shfl` instruction, and then all 32 threads in the warp access the candidate slots in parallel. For other processes that should be performed in sequence (e.g., writing the key after locating the target slot), a dedicated thread in the warp (e.g., the activated thread) is responsible for performing these processes. Unlike the warp-agnostic execution manner, such a warp-cooperative execution manner actively controls warp divergence with intra-warp synchronization and efficiently exploits the parallelism of warps.

3.2.2 Correctness Challenges

Duplicate items. In rare cases, when concurrently performing insertion operations with the same key in a lock-free manner, threads may insert the key into different slots, thus leading to duplicate items. To tolerate duplicate items, akin to the prior work [78], GPHash determines the *valid* item of a key. Given multiple items of the same key, the valid item is the one having the maximal level number, the minimal bucket number, and the minimal slot number. When finding duplicates, GPHash keeps the valid item and deletes other duplicates.

Concurrency correctness. When threads concurrently perform the search and the IDU (i.e., insertion/deletion/update) operations with the same key, the readers may return the partial-updated value, which violates the concurrency correctness. To ensure concurrency correctness while providing high performance, GPHash follows the *"no lost key"* concurrent correctness condition akin to prior schemes [38, 78]. Specifically, when threads concurrently perform the search and the update operations, the search operations return either the old or the new values instead of partial-updated values. When a search and a deletion run in parallel, the search operation returns either the value or no-key statement.

Crash consistency guarantee. When directly managing data in persistent memory, a crash would interrupt the ongoing index operations, which can lead to persistent partial updates for keys and values. Such data inconsistency causes data loss and unpredictable errors. To guarantee data consistency in the presence of crashes, GPHash uses CAS primitive and the slot state to achieve log-free operations with negligible overhead.

3.2.3 Lock-Free and Log-Free Operations

We introduce the details of lock-free and log-free operations. Here, we focus on operations of the fixed-length large keys whose sizes are larger than 8 bytes, while the operations of fixed-length small keys (i.e., $\leq$ 8 bytes) and variable-length keys can be implemented in a similar way using the CAS primitive. We use system-scoped threadfence [46] to order the persists for the correct consistency guarantee.

Insertion. Figure 3 illustrates the lock-free and log-free insertions. First, GPHash obtains the fingerprints and the keys of all candidate slots of the activated key with one-shot warp access. GPHash then checks if the key exists by comparing these keys with the activated key, while leveraging the fingerprints for fast comparison. If the activated key does not exist, GPHash finds the empty slots, i.e., the slots whose states are `EMPTY`[2]. If there are several empty slots, GPHash inserts the activated key into the slot belonging to the less-loaded bucket. After deciding the target slot for insertion, the activated thread uses CAS primitive to atomically change the slot state (i.e., fingerprint region) from `EMPTY` to `INSERT`. If the CAS fails, meaning that the slot is changed by another thread, GPHash re-executes the insertion from the beginning. If CAS succeeds, the activated thread writes the item into the target slot. Finally, the activated thread sets the fingerprint region of the target slot to the hash value of the activated key.

The insertion can easily recover from crashes. There are two cases of a slot after crashes. (1) The slot state is `INSERT`,

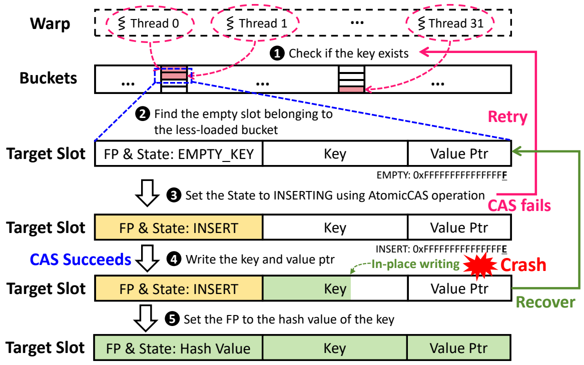


Figure 3: The illustration of lock-free and log-free insertion *(using the logical structure of a slot for easy understanding).*

indicating that the slot is under insertion (i.e., writing a new item) before crashes. In this case, the slot may be broken, and thus we need to clear the slot and set the slot state to `EMPTY`. (2) The slot state is not `INSERT`, meaning that the slot is empty or contains an unbroken item. In this case, we do not need to do anything since the slot is already in a valid state.

Deletion. For deletion operation, GPHash first locates the target items whose keys are equal to the activated key, including duplicate items. Similar to insertion, the activated thread is responsible for atomically deleting all these items by using the CAS primitive to set the slot states to `EMPTY`. Thanks to the atomicity of the CAS primitive, the deletion does not introduce any invalid slot state in the presence of crashes.

Update. For the update operation in GPHash, after locating the target slot and deleting the other duplicates, the activated thread atomically changes the value pointer to point to the new value via the CAS primitive. GPHash writes the new value to the pre-allocated space before updating the value pointer. After crashes, the value pointer either points to the old value or the new one, both of which are unbroken.

Search. Since GPHash takes advantage of the atomicity of the CAS primitive to perform the IDU operations, the lock-free search operation can be easily implemented. After locating all slots whose keys are equal to the activated key, the activated thread reads the value that is pointed by the value pointer of the valid slot. If the activated key does not exist, the thread returns a no-key statement. Based on the above introduction to other operations, the search operation can be proved to meet the *"no lost key"* concurrent correctness condition.

Resizing. As the load factor increases, more hash collisions will occur in hash indexes, which results in performance degradation and insertion failure. Thanks to the one-shot warp access, GPHash does not suffer from performance degradation caused by more hash collisions. However, GPHash still needs to handle insertion failure to avoid item loss. If failing to find an empty slot to insert a new item, GPHash has to resize. Specifically, GPHash first allocates a new level as the new top one. GPHash then leverages thousands of GPU threads to scan the bottom level in parallel and rehashes the items. Each rehashing operation consists of reading the item in the

[2] We reserve two 8-byte values in the fingerprint value range, i.e., `EMPTY` and `INSERT`, to indicate the slot is empty or under insertion.

bottom level, inserting the item into the other levels (including the new level), and deleting the item from the bottom level. GPHash also performs rehashing in a warp-cooperative execution manner to mitigate warp divergence.

It is worth noting that there is no rehashing failure (meaning that the insertion of a rehashed item fails) in GPHash. Because the new level can store more items than the bottom level, and there are fewer hash collisions in the new level than in the bottom level. Moreover, by leveraging the atomicity of insertion and deletion operations, the resizing can tolerate crashes. There are three cases of rehashed items after crashes. (1) The item has not been inserted into the other levels. (2) The item has been inserted into the other levels and has not been deleted from the bottom level. (3) The item has finished rehashing and has been deleted from the bottom level. While we do not need to do anything for items in case (3), the rehashing of items in cases (1) and (2) continues after recovery. We observe that GPHash can directly continue to rehash items in the bottom level, i.e., items in cases (1) and (2), since re-inserting the items in case (2) will simply return the key-existing statement.

Recovery. To recover from normal shutdowns or system crashes, GPHash first initializes the GPM system, i.e., mapping the PM file onto the GPU's virtual address space. Then, GPHash checks the slot states and clears the slots that were under insertion. Besides, if GPHash was performing resizing before crashes (indicated by a flag `is_resizing`), GPHash continues to rehash the items in the bottom level. In our implementation, we carefully consider all cases during resizing operation including updating metadata in the presence of crashes, which are omitted here due to space limits.

3.3 Frozen-Based Bucket Cache

To bridge the bandwidth gap between PM and GPU, we aim to reduce PM accesses. Previous studies show that real-world workloads often feature Zipfian popularity distribution [4, 8, 9, 28, 71]. Under such skewed workloads, hot items receive extremely frequent accesses. Based on this observation, GPHash reduces PM accesses by caching hot items in GPU memory. Traditional caching schemes fetch items in case of encountering a cache miss (i.e., the accessed item is not in the cache) and evict items to make room for these newly fetched items. Unfortunately, in the context of GPM systems, these caching schemes suffer from high overheads for cache management. For example, most caching schemes, such as LRU, use linked-lists to achieve the $O(1)$ time efficiency for cache management [17, 39]. However, when massive GPU threads frequently query and update the lists, such list-based implementations cause high contentions among GPU threads, which makes these caching schemes inefficient. To address this problem, we propose BktCache, a frozen-based bucket cache that minimizes the overhead of cache management.

Bucket granularity. While traditional caches often cache hot items at the *single-item* granularity, our BktCache caches items at the *bucket* granularity. This is based on the observation that the items in the same bucket are *always* accessed concurrently due to the one-shot warp access feature of GPHash. Besides, BktCache identifies hot items at the bucket granularity, which reduces storage overhead for the metadata. For example, when implementing the LFU caching algorithm [39], we only record the accessed frequencies for buckets. We further record mapping relationships between buckets in GPHash and buckets in BktCache, defined as (1) $F(x) = y$, which indicates that the bucket x in GPHash is cached into the bucket y in BktCache, and (2) $G(y) = x$, which indicates that the bucket y in BktCache is cached from the bucket x in GPHash.

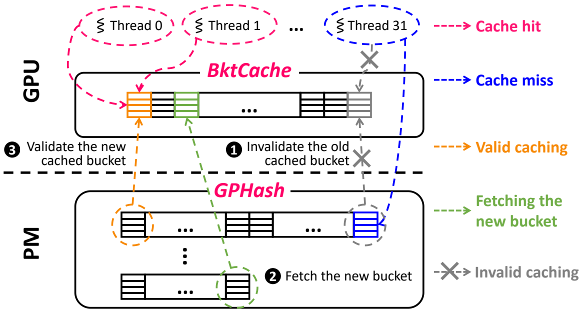
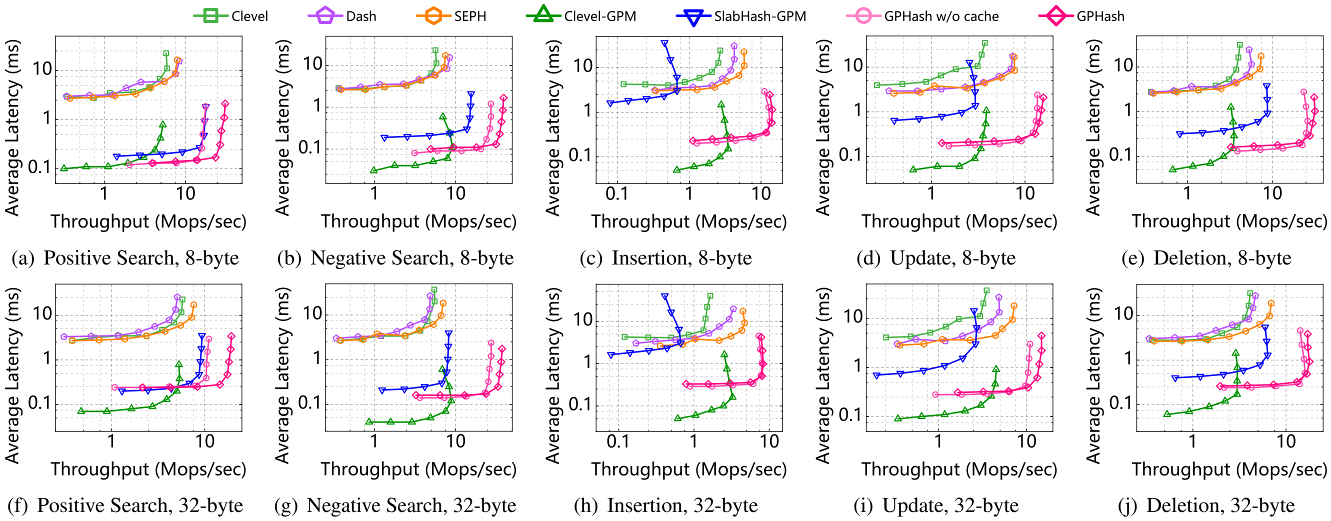


Figure 4: The overview of concurrent fetching mechanism.

Frozen-based caching. Since there are thousands of concurrent GPU threads, it is important for BktCache to avoid contentions among threads. Inspired by FrozenHot [54], we adopt the frozen-based design in BktCache. We periodically load the BktCache, which includes identifying the hot buckets via caching algorithms (e.g., LFU [39]) and fetching these buckets into the BktCache. The membership of the cached buckets is unchanged between the two adjacent loading phases. There are two benefits of the frozen-based caching design. (1) It significantly decreases the overhead of cache management. (2) It does not suffer from performance degradation caused by cache thrashing [54]. The concern about adopting the frozen-based design comes from the decrease in the hit rate. Since the real-world workloads often exhibit the scan or repeated access patterns [55, 71], shuffling cache contents leads to marginal profits on hit rates when the scan size is larger than the cache size. In GPHash, the overhead of dynamically evicting and fetching buckets in BktCache overwhelms the performance gain of the limited increases in hit rates, which causes severe performance degradation by several orders of magnitude. In contrast, the frozen-based cache design minimizes the management cost while achieving comparable hit rates, hence enabling higher performance.

Concurrent loading scheme. To mitigate the interference of the cache loading to ongoing index operations, BktCache concurrently loads the BktCache. While the concurrent identification of hot buckets can be easily implemented, it is non-trivial to realize concurrent fetching. Figure 4 shows the overview of concurrent fetching mechanism. To fetch a bucket x to the bucket y in the BktCache, we first invalidate the old mapping by setting the $F(G(y))$ to `NOT_CACHED`. To avoid inconsis-



(a) Positive Search, 8-byte (b) Negative Search, 8-byte (c) Insertion, 8-byte (d) Update, 8-byte (e) Deletion, 8-byte

(f) Positive Search, 32-byte (g) Negative Search, 32-byte (h) Insertion, 32-byte (i) Update, 32-byte (j) Deletion, 32-byte

Figure 5: The throughputs and latencies of different index operations under various configurations.

tency in the ongoing operations on the bucket y, we wait for completions of these operations (i.e., waiting for the reference count of bucket y to become 0) before updating the bucket y. We then copy the content of the bucket x to the bucket y. Finally, we validate the new mapping by setting the $F(x)$ to y and the $G(y)$ to x. The mapping relationships between buckets are atomically updated by using the CAS primitive, hence ensuring the concurrency correctness. We implement the concurrent fetching mechanism by using another GPU stream to fetch different buckets in parallel.

Operations with caching. For all index operations, GPHash prioritizes reading keys from the BktCache, which accelerates accessing the buckets. After locating the target bucket, for search operations, if the target buckets are cached, GPHash directly reads the values via the pointers in the cached buckets. For IDU operations, if the target buckets are cached, GPHash needs to modify the corresponding buckets in the BktCache after completing modifications to the buckets in GPHash.

4 Performance Evaluation

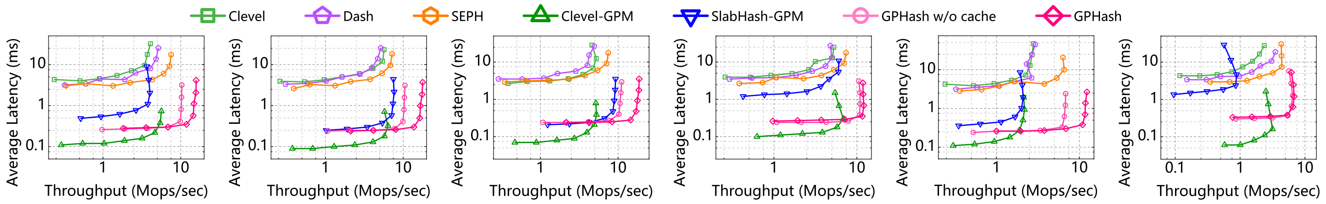
4.1 Experimental Setup

Platform. Our experiments are conducted on a Linux server equipped with two Intel 26-core Xeon Gold 6230R CPUs, one NVIDIA Tesla V100 GPU, 192 GB DDR4 DRAM, and 768 GB Intel Optane DC PMM (6 × 128 GB PMMs). The Optane DC PMMs, as a case in point, are configured in the App Direct mode and mounted with the ext4-DAX file system. The server is installed with Ubuntu 18.04. The CUDA version is 11.4. To avoid the impact of NUMA architectures, we conduct all the experiments on one CPU socket by pinning threads to one NUMA node consistent with prior works [12, 33].

Comparisons. We evaluate the following data management approaches using configurations suggested by original papers. We evaluate the CPU-assisted data management approaches. For fair evaluation, we also use PM to store the data and leverage the three representative PM hash indexes to manage the data in PM, including *Clevel* [12], *Dash* [36], and *SEPH* [63]. Moreover, we evaluate the GPM-enabled data management approaches that leverage GPM hash indexes to directly manage the data stored in PM, including naive GPM hash indexes and *GPHash*. To the best of our knowledge, there is no existing hash index tailored for GPM, and hence we implement and evaluate two naive GPM hash indexes by porting Clevel (which is the closest to GPHash) and a list-based GPU hash index SlabHash [3] to GPM systems, i.e., *Clevel-GPM* and *SlabHash-GPM*. In Clevel-GPM, each thread independently performs index operations without intra-warp communication. For SlabHash-GPM, we use logging to ensure crash consistency.

Workloads. We leverage widely used YCSB [13] benchmark and multiple real-world workloads to evaluate the performance of GPHash and the compared schemes.

- ***YCSB workloads.*** We generate a micro-benchmark to evaluate the performance of different index operations, which contains the following types of workloads: *Positive search* (the queried keys exist), *Negative search* (the queried keys do not exist), *Insertion*, *Update*, and *Deletion*. Besides, we further generate a macro-benchmark, which contains the 5 YCSB core workloads: *A* (50% read, 50% update), *B* (95% read, 5% update), *C* (100% read), *D* (read-the-latest, 95% read, 5% insertion), *F* (50% read, 50% read-modify-write) and *LOAD* (100% insertion). Since none of the hash indexes optimizes for the range query, we do not evaluate the YCSB E workload. The workloads are generated in the Zipfian distribution with the default skewness ($\theta = 0.99$). The experiment on YCSB workloads consists of load and run phases. In the load phase, we initialize the hash indexes with 16 million key-value pairs for micro- and macro-benchmarks. In the run phase, each hash index performs 16 million and 64 million operations in the micro- and macro-benchmarks, respectively. For most experiments, we use 8-byte and 32-byte keys that



(a) YCSB A (b) YCSB B (c) YCSB C (d) YCSB D (e) YCSB F (f) YCSB LOAD

Figure 6: The throughputs and latencies under YCSB workloads using 32-byte keys.

are representative of real large-scale key-value store workloads [4,71], while the length of values is set to 128 bytes (i.e., the size of 32-dimension vector stored in FP32 type).

• ***Real-world workloads.*** We leverage the following real-world workloads for extensive evaluation. (1) *DLRM*. This workload is generated by running the deep learning recommendation model [20] using the Criteo-Kaggle CTR dataset [30]. The Criteo-Kaggle CTR dataset contains more than 45 million click feedbacks. (2) *PageRank.* This workload is generated by running the PageRank algorithm [7] using Twitter social graph [61]. The Twitter social graph contains 41.7 million user profiles and 1.5 billion social relations.

Default configurations. Unless otherwise stated, we evaluate CPU-assisted approaches using 52 threads. For both CPU-assisted and GPM-enabled approaches, we perform operations in a batched manner using a default batch size 2^{12} (i.e., the number of batched operations). By default, GPHash uses 2 hash locations and 2-level buckets with 8-way associativity, and the size of BktCache (i.e., the ratio of the number of cached buckets to the number of total buckets) is configured to 20% while using the LFU algorithm to identify hot buckets.

4.2 Overall Performance

Figures 5 and 6 present the throughput-latency curves of different data management approaches using 8-byte and 32-byte keys, respectively. To plot a throughput-latency curve, we record the throughput and latency of an approach using various batch sizes.

Search-only workloads *(Pos./Neg. Search, YCSB C).* For the search-only workloads, the results show that directly managing data via GPM hash indexes enables higher throughput and lower latency than CPU-assisted approaches by up to 13.84× and 27.62× respectively. This is because these GPM-enabled approaches eliminate the time-consuming transmission between GPU and CPU, while leveraging the high parallelism of GPU. Among three GPM hash indexes, Clevel-GPM obtains lower latency when batch size is small due to no overheads for intra-warp communication. However, when the batch size increases, Clevel-GPM suffers from severe warp divergence due to its unawareness of GPU's warp-based execution manner, thus failing to achieve high throughput. In contrast, GPHash and SlabHash-GPM can offer high throughput by employing the warp-cooperative execution manner. Moreover, by leveraging BktCache, GPHash enables up to 4.28× higher throughput than SlabHash-GPM.

Insertion workloads *(Insertion, YCSB D, LOAD).* For the Insertion and YCSB LOAD workloads, GPHash consistently outperforms other approaches by 5.79×. We attribute these improvements to the one-shot warp access feature of GPHash, which exploits the parallelism of a warp and does not introduce any extra data movement. Unlike GPHash, SlabHash-GPM needs to probe all linked-list nodes of the target bucket for an insertion, which is time-consuming. Moreover, such a list-based design fails to achieve efficient load balance, thus leading to more contentions among threads and performance degradation. For the YCSB D workload, GPHash provides 2.16× higher throughput than the other two GPM hash indexes due to its warp-oriented optimization.

Update/deletion workloads *(Update, Deletion, YCSB A, B, F).* Compared with other approaches, GPHash gains improvements in terms of throughput by up to 9.23× and 10.37× under the Update and the Deletion workloads, respectively. The improvements stem from the log-free and lock-free operations of GPHash as discussed in §3.2.3. In contrast, the logging overheads deteriorate the performance of SlabHash-GPM. The trends under the YCSB A and B workloads are similar, where GPHash outperforms other approaches by 4.75×.

Load factor. To evaluate the memory efficiency of different hash schemes, we record the load factors (i.e., the number of the inserted keys divided by the capacity of the hash index) of each scheme after every 16K (i.e., 2^{14}) insertions during the load phase. As shown in Figure 7, by adopting effective techniques for load factor improvement, Dash, Clevel (and Clevel-GPM), and SEPH can achieve high load factors of up to 85%. SlabHash-GPM resizes when the number of inserted keys approaches the current capacity of the hash table (e.g., 80%). As a result, the load factor of SlabHash-GPM is up to 82%. GPHash employs slot associativity, inter-level sharing, and multiple hash locations to handle more hash collisions and achieve efficient load balance between buckets, hence offering high load factors of up to 92%.

4.3 Sources of Improvements

Figure 8 presents the latency breakdowns of different schemes. The results show that the data transmission between GPU and CPU accounts for over 30.1% end-to-end latency in CPU-assisted data management approaches. In contrast, GPM-enabled approaches eliminate the transmission overheads by leveraging GPM hash indexes to directly manage the data, thus delivering lower latency. However, Clevel-GPM pas-

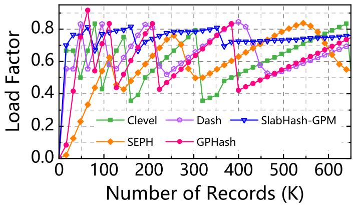


Figure 7: The load factors of different hash schemes.

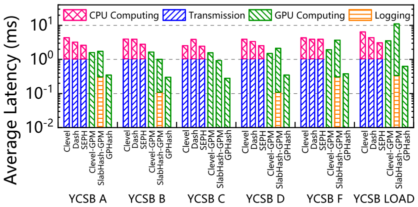


Figure 8: The latency breakdowns of different hash schemes.

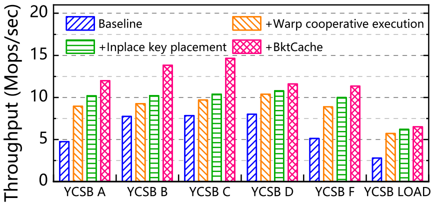


Figure 9: The factor analysis of GPHash design.

sively handles warp divergence due to its warp-agnostic execution manner, failing to fully exploit the high parallelism of GPU. Moreover, SlabHash-GPM uses logging for the crash consistency guarantee, which introduces up to 15.2% overhead and hinders SlabHash-GPM from achieving high performance. By embracing warp-cooperative execution manner and log-free operations, GPHash avoids the drawbacks of Clevel-GPM and SlabHash-GPM and hence outperforms other approaches.

4.4 Factor Analysis

To better understand the impacts of the proposed techniques in GPHash, we evaluate and analyze the performance contributions of these techniques. Figure 9 presents the factor analysis of GPHash. We start with *Baseline* that does not adopt any mentioned techniques. We then apply each proposed technique one by one.

+ Warp-cooperative execution manner. Executing index operation at the warp granularity mitigates the warp divergence problem and thus improves the performance. The warp-cooperative execution manner contributes to up to 104.1% improvement in throughput. Warp cooperation is especially important for workloads that consist of mixed operations, e.g., YCSB A and F. Without warp cooperation, when threads in a warp concurrently perform different operations, the *Baseline* suffers from severe warp divergence. Note that although YCSB LOAD only contains insertion operations, it also benefits a lot from the warp-cooperative execution manner. This is because, for insertion operations, GPHash needs to check whether the activated key exists and re-insert the key if the CAS primitive fails, which also introduces massive warp divergence among concurrent insertion operations.

+ In-place key placement. The in-place key placement contributes to all workloads by facilitating coalesced memory accesses. By adopting the in-place key placement, when comparing the keys of the slots and the activated key, the accesses of the keys can be coalesced, thus further increasing the throughput by up to 13.7%.

+ BktCache. Caching hot items in BktCache enables higher performance in searching and locating a specific key. For skewed workloads such as YCSB A, B, and C, BktCache brings improvements in throughput by up to 40.9%. On the other hand, for balanced workloads such as YCSB D and LOAD, BktCache only achieves 7.6% improvements. Besides, for workloads that contain IDU operations, GPHash needs to modify the corresponding buckets in the BktCache, which weakens the performance benefits brought by BktCache. Specifically, with the same skewness, BktCache achieves 17.6% improvements for YCSB A while offering 35.5% improvements for YCSB B because YCSB B contains fewer update operations.

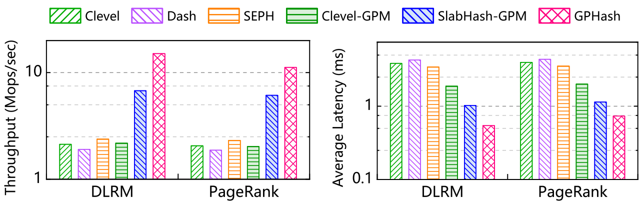
4.5 Real-World Workloads

To demonstrate the generality of GPHash, we further use real-world workloads to evaluate GPHash and the compared schemes. Figure 10 shows the throughputs and latencies of different schemes under the *DLRM* and *PageRank* workloads. Despite various workloads, GPHash consistently provides up to 7.09× higher throughput and up to 7.91× lower latency than other schemes, respectively. The results demonstrate the efficiency of GPHash across different workloads. For other approaches except GPHash, their throughputs show trivial differences across workloads. This is because the access pattern (i.e., workload skewness) has little impact on their performance due to the constant-scale query performance of hash indexes. In contrast, GPHash provides higher throughput on workloads that exhibit higher skewness, since BktCache can accelerate more index operations upon these workloads.

4.6 Sensitivity Analysis

In this section, we investigate how caching algorithms, cache sizes, key size, workload skewness, and configurations of GPHash affect the performance of GPHash.

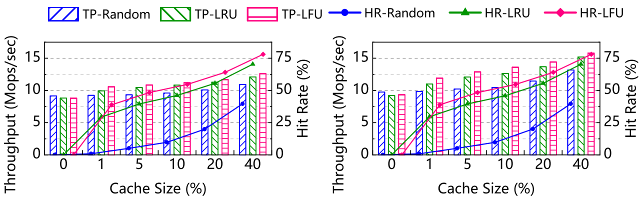
Caching algorithm and cache size. To demonstrate the impacts of caching algorithms and cache sizes on GPHash's performance, we evaluate the performance of GPHash using different caching algorithms and cache sizes. As shown in Figure 11, for both the YCSB A and C workloads, the LFU and LRU algorithms outperform the Random algorithm for most cache sizes, while LFU provides slightly higher hit rates and throughput. However, when the cache size is 0, the Random algorithm delivers higher throughput since it does not need to record any extra bucket information for deciding which buckets to cache. On the other hand, the LFU and LRU algorithms need to record the number and the latest time of access, which introduces some overheads of recording. For the YCSB A workload, since GPHash needs to update the cached buckets in the BktCache for update operations, the BktCache provides less performance improvement under YCSB A compared to under YCSB C. Moreover, it is worth noting that the benefits



(a) Throughput

(b) Latency

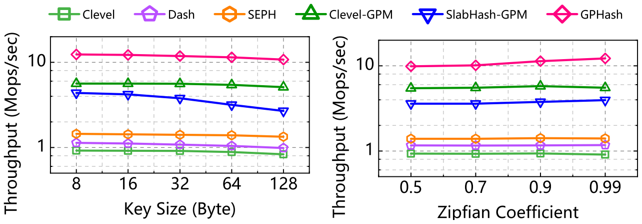
Figure 10: The throughputs and latencies of different schemes under real-world workloads.



(b) YCSB A

(c) YCSB C

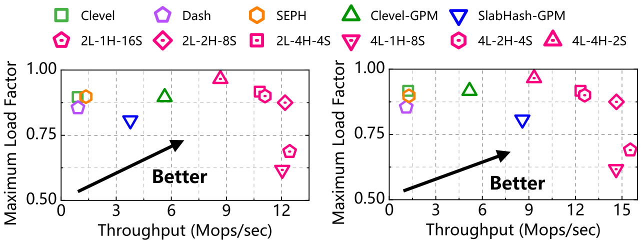
Figure 11: The throughputs (TP) and hit rates (HR) of different caching algorithms with various cache sizes.



(a) Key Size

(b) Skewness

Figure 12: The throughputs under different experimental configurations.



(a) YCSB A

(b) YCSB C

Figure 13: A spectrum of GPHash allow users to tradeoff between load factor and throughput. *(2L-2H-8S indicates that GPHash uses 2 hash locations and 2-level buckets with 8-way associativity.)*

of BktCache become marginal as the cache size increases. For example, the LFU-based BktCache brings 45.7% improvements on YCSB C when the cache size increases from 0 to 10%. However, it only provides 16.0% higher throughput when the cache size increases from 10% to 40%. Therefore, it would be better to use a suitable cache size (e.g., 20%) to achieve the sweet point between high performance and low GPU memory footprint.

Key size. Figure 12(a) shows the throughputs of different approaches using various key sizes under the YCSB C workload. When the key size increases from 8 bytes to 128 bytes, most approaches only exhibit slight performance degradation (e.g., 13.1% for GPHash), while SlabHash-GPM experiences a rapid decline in performance by 38.6%. This is because most approaches employ fingerprinting or similar technologies to avoid reading the full keys in most cases, while SlabHash-GPM does not.

Skewness of workloads. Figure 12(b) shows how the skewness of workloads affects the performance of GPHash under the YCSB C workload. For other approaches except GPHash, the skewness of workloads has a negligible impact on their performance. For GPHash, with increasing skewness, BktCache can absorb more index operations, thus increasing the performance.

Configuration of GPHash. Figure 13 presents the throughputs and the maximum load factors of GPHash under different configurations of GPHash. To avoid potential impacts on other design components, we only consider configurations that enable the one-shot warp access feature. For clarity, we do not plot the configurations that neither provide higher maximum load factors nor offer higher throughput. The results show that using multiple hash locations and adopting inter-level sharing can achieve higher load factors. However, while using large numbers of hash functions and levels enables extremely high load factor, the number of slots is limited, which causes uncoalesced memory accesses and performance degradation. In GPHash, we use *2L-2H-8S* as the default configuration since it offers high performance while providing a decent load factor. In practice, users can choose which configurations to use, depending on the specific requirements of the deployed scenarios.

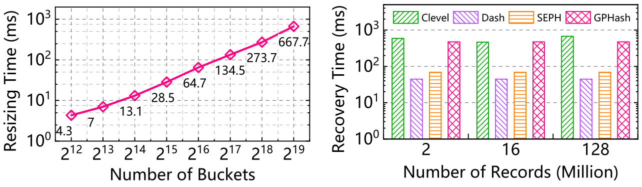
4.7 Overheads of GPHash

Resizing. Figure 14(a) shows the resizing time with different numbers of buckets in GPHash. Since GPHash performs rehash operations in parallel, the resizing can be completed within hundreds of milliseconds.

Recovery. As shown in Figure 14(b), GPHash achieves instant recovery and provides recovery time comparable to PM hash indexes (within hundreds of milliseconds). In fact, GPM initialization (i.e., mapping PM space onto the GPU's virtual address space) consumes the most time (> 99%). Since GPHash leverages thousands of threads to check the states of buckets, the time consumption of checking is trivial.

Loading BktCache. Figure 15 presents the overheads of loading BktCache with different cache sizes. Without the concurrent fetching mechanism, the throughput rapidly drops to 0 since the operations stall until the loading is completed. By adopting concurrent fetching mechanism, the throughput does not significantly decrease. Although the concurrent fetching slightly increases the loading time, loading BktCache only consumes hundreds of milliseconds.

Metadata in BktCache. The storage overheads of metadata



(a) Resizing Time (b) Recovery Time

Figure 14: The time consumptions of resizing and recovery.

in BktCache are limited. When using the LFU algorithm, the per-bucket counter only introduces a storage overhead of 4 bytes for a bucket, which is negligible compared with the bucket size (i.e., 384 bytes under default configurations).

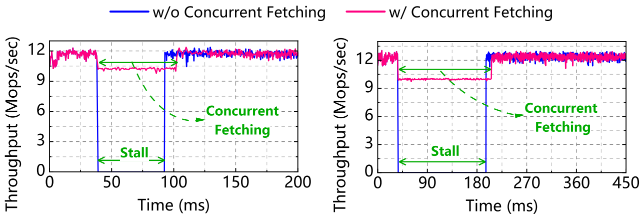
5 Discussions

Variable-length key-value pairs. GPHash supports variable-length key-value pairs by storing the pointers to keys and values. However, to efficiently support variable-length KVs, the GPU allocator needs to achieve fast concurrent variable-length allocation without incurring high memory fragmentation, which is not the main design goal of this work and is left to our future work. In fact, supporting variable-length key-value pairs is not a necessity in real-world GPM-enabled applications. For example, in deep recommendation systems, since the keys (e.g., product IDs) and values (i.e., embedding vectors) of embedding vector lookups are fixed-length, the hash indexes that support fixed-length key-value pairs are enough to boost the performance in such a scenario.

Concurrent resizing. It is challenging for GPM hash schemes to support concurrent resizing. During the rehashing operation, the hash index needs to concurrently perform the ongoing index operations and insertions for rehashing. As a result, the insertions for rehashing will interfere with the ongoing index operations, which decreases the performance. In GPHash, it is harder to support concurrent resizing since we need to guarantee the consistency of BktCache during resizing. Moreover, concurrent resizing weakens the performance gain of one-shot warp access since GPHash needs to also probe the new level. However, since the number and the overhead of resizing are limited in GPHash, it is well-recognized in the community to only support static resizing [3, 33, 34, 77].

6 Related Work

Hash indexes for PM. There are many hash indexes [6, 12, 26, 33, 36, 40, 44, 63, 77] tailored for PM to achieve high performance. Level hashing [77] proposes a two-level hash table to achieve cost-efficient resizing and constant-scale time complexity with limited extra PM writes. Based on the level hashing, Clevel hashing [12] proposes a lock-free multi-level scheme that supports asynchronous resizing, thus further improving the performance. Dash [36] employs several techniques including balanced insert, displacement, and stashing to delay segment splitting, to reduce the cache misses and PM accesses. SEPH [63] introduces a level segment structure as a key to break the dilemma between efficiency and predictability. Some designs such as RECIPE [33] and Pronto [40] consider the general-purpose conversion methods that convert volatile DRAM indexes into persistent counterparts for PM. However, the above hash indexes are agnostic to the GPU architecture, which hence become inefficient for GPM systems. Unlike these schemes, our GPHash can efficiently leverage the high parallelism of GPU to achieve high performance.



(b) Small Size (5%) (c) Large Size (20%)

Figure 15: The overheads of loading BktCache.

GPU hash tables. Existing GPU hash tables focus on exploiting the high parallelism of GPU [1, 3, 31, 34, 74]. Stash [31] uses a compact data structure to support out-of-core GPU parallel hashing. Mega-KV [74] is an efficient design of GPU-based cuckoo hashing, which boosts overall performance. However, these hash tables are designed for the static case where the data size for insertions is known in advance, and thus cannot support dynamic workloads. SlabHash [3] is a dynamic hash table on GPUs, which uses an efficient GPU allocator to handle concurrent allocations for the hash table. However, the above GPU hash tables are designed for volatile GPU memory, hence failing to efficiently guarantee crash consistency in GPM systems. Unlike these schemes, our GPHash can ensure crash consistency with proper overheads by leveraging log-free operations.

7 Conclusion

In this paper, we have designed, implemented, and evaluated GPHash, which is an efficient hash scheme for GPM systems. The hash table structure is designed to be GPU-conscious and PM-friendly, which enables high performance and high memory efficiency. GPHash adopts a warp-cooperative execution manner to mitigate warp divergence and support one-shot warp access. GPHash further leverages lock-free and log-free operations to achieve lock-free concurrency control with the crash consistency guarantee. Moreover, GPHash reduces PM accesses by caching hot buckets in BktCache while minimizing cache management overheads. Evaluation shows that GPHash outperforms state-of-the-art CPU-assisted data management approaches and GPM hash indexes by several times.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to our shepherd, Yi Xu, and anonymous reviewers for their comments and suggestions.

References

[1] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics*, 28(5):154, 2009

[2] Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Jens Axboe, Valmiki Rampersad, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Dheevatsa Mudigere, Shishir Juluri, Akshat Nanda, Manoj Wodekar, Krishnakumar Nair, Maxim Naumov, Chris Petersen, Mikhail Smelyanskiy, and Vijay Rao. Supporting massive DLRM inference through software defined memory. In *Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems (ICDCS'22)*, pages 302–312. 2022

[3] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. A dynamic hash table for the GPU. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS'18)*, pages 419–429. 2018

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, pages 53–64. 2012

[5] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martin Farach-Colton, and John D. Owens. Engineering a high-performance GPU b-tree. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*, pages 145–157. 2019

[6] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment (PVLDB)*, 14(9):1544–1556, 2021

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 209–223. 2020

[9] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 239–252. 2020

[10] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment (PVLDB)*, 13(12):2634–2648, 2020

[11] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, pages 1077–1091. 2020

[12] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*, pages 799–812. 2020

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 143–154. 2010

[14] Intel Corporation. Intel Optane DC persistent memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, 2019.

[15] Kioxia Corporation. XL-FLASH Storage Class Memory Solution. https://www.kioxia.com/en-jp/business/news/2022/20220802-1.html, 2022.

[16] CXL. Compute Express Link Specification. https://computeexpresslink.org/, 2024.

[17] Asit Dan and Donald F. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems (SIGMETRICS'90)*, pages 143–152. 1990

[18] Harish Doraiswamy and Juliana Freire. A gpu-friendly geometric data model and algebra for spatial queries. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD'20)*, pages 1875–1885. 2020

[19] Harish Doraiswamy and Juliana Freire. SPADE: gpu-powered spatial database engine for commodity hardware. In *Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE'22)*, pages 2669–2681. 2022

[20] Facebook. Deep Learning Recommendation Model for Personalization and Recommendation Systems. https://github.com/facebookresearch/dlrm, 2024.

[21] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC'24)*, pages 111–126. 2024

[22] Robert B. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating System Principles (SOSP'87)*, pages 155–162. 1987

[23] Tom's Hardware. Intel Kills Optane Memory Business. https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good, 2022.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, pages 770–778. 2016

[25] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference (USENIX Winter'94)*, pages 235–246. 1994

[26] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. Halo: A hybrid pmem-dram persistent hash index with fast'recovery. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD'22)*, pages 1049–1063. 2022

[27] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. Persistent memory hash indexes: An experimental evaluation. *Proceedings of the VLDB Endowment (PVLDB)*, 14(5):785–798, 2021

[28] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets'14)*, pages 8:1–8:7. 2014

[29] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019

[30] Kaggle. Display Advertising Challenge. https://www.kaggle.com/c/criteo-display-ad-challenge, 2014.

[31] Farzad Khorasani, Mehmet E. Belviranli, Rajiv Gupta, and Laxmi N. Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. In *Proceedings of the 2015 International Conference on Parallel Architectures and Compilation (PACT'15)*, pages 63–74. 2015

[32] Daniar Heri Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. Evstore: Storage and caching capabilities for scaling embedding tables in deep recommendation systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 281–294. 2023

[33] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, pages 462–477. 2019

[34] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. Dycuckoo: Dynamic hash tables on gpus. In *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE'21)*, pages 744–755. 2021

[35] Xiaocheng Liu, Ziming Zhong, and Kai Xu. A hybrid solution method for CFD applications on gpu-accelerated hybrid HPC platforms. *Future Generation Computer Systems*, 56:759–765, 2016

[36] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *Proceedings of the VLDB Endowment (PVLDB)*, 13(8):1147–1161, 2020

[37] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: range-query optimized persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 1–16. 2021

[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the Seventh European Conference on Computer Systems (EuroSys'12)*, pages 183–196. 2012

[39] Dhruv Mátáni, Ketan Shah, and Anirban Mitra. An O(1) algorithm for implementing the LFU cache eviction scheme. *CoRR*, abs/2110.11602, 2021

[40] Amir Saman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast'persistence for volatile data structures. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, pages 789–806. 2020

[41] Microsoft. Best Practices on Recommendation Systems. `https://github.com/recommenders-team/recommenders`, 2024.

[42] Ben Mildenhall, Peter Hedman, Ricardo Martin-Brualla, Pratul P. Srinivasan, and Jonathan T. Barron. Nerf in the dark: High dynamic range view synthesis from noisy raw images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'22)*, pages 16169–16178. 2022

[43] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001

[44] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, pages 31–44. 2019

[45] NVIDIA. Using CUDA Warp-Level Primitives. `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`, 2018.

[46] NVIDIA. CUDA C++ Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2022.

[47] NVIDIA. GPU Performance Background User's Guide. `https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html`, 2023.

[48] NVIDIA. Getting Started with NVIDIA GPUDirect Storage. `https://docs.nvidia.com/gpudirect-storage/getting-started/index.html`, 2024.

[49] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, pages 371–386. 2016

[50] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluç, Jiejun Xu, and Hang Liu. H-INDEX: hash-indexing for parallel triangle counting on gpus. In *Proceedings of the 2019 IEEE High Performance Extreme Computing Conference (HPEC'19)*, pages 1–7. 2019

[51] Shweta Pandey, Aditya K. Kamath, and Arkaprava Basu. GPM: leveraging persistent memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, pages 142–156. 2022

[52] Shweta Pandey, Aditya K. Kamath, and Arkaprava Basu. Scoped buffered persistency model for gpus. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 688–701. 2023

[53] Shujian Qian and Ashvin Goel. Massively parallel multi-versioned transaction processing. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 765–781. 2024

[54] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys'23)*, pages 557–573. 2023

[55] Liana V. Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. 2021

[56] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*, pages 99–110. 2013

[57] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. Persistent memory disaggregation for cloud-native relational databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 498–512. 2023

[58] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, pages 622–636. 2018

[59] Samsung. Samsung Memory-Semantic CXL SSD. https://semiconductor.samsung.com/us/news-events/tech-blog/samsung-cxl-solutions-cmm-h/, 2024.

[60] Stefan Seritan, Keiran Thompson, and Todd J. Martínez. Terachem cloud: A high-performance computing service for scalable distributed gpu-accelerated electronic structure calculations. *Journal of Chemical Information and Modeling*, 60(4):2126–2137, 2020

[61] Twitter. Twitter Social Graph. https://anlab-kaist.github.io/traces/WWW2010, 2010.

[62] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. Plush: A write-optimized persistent log-structured hash-table. *Proceedings of the VLDB Endowment (PVLDB)*, 15(11):2895–2907, 2022

[63] Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. SEPH: scalable, efficient, and predictable hashing on persistent memory. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 479–495. 2023

[64] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and optimizing remote persistent memory with RDMA and NVM. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC'21)*, pages 523–536. 2021

[65] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Nyxcache: Flexible and efficient multi-tenant persistent memory caching. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, pages 1–16. 2022

[66] Ping Xiang, Yi Yang, and Huiyang Zhou. Warp-level divergence in gpus: Characterization, impact, and mitigation. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*, pages 284–295. 2014

[67] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: an efficient GPU embedding cache for personalized recommendations. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys'22)*, pages 402–416. 2022

[68] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, pages 346–359. 2021

[69] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 169–182. 2020

[70] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 167–181. 2015

[71] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 191–208. 2020

[72] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex M. Bronstein, Ronald G. Dreslinski, Trevor N. Mudge, and Nishil Talati. GRACE: A scalable graph-based approach to accelerating recommendation model inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 282–301. 2023

[73] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: an fpga-accelerated embedding-based retrieval system. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, pages 841–856. 2022

[74] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment (PVLDB)*, 8(11):1226–1237, 2015

[75] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In Dahlia Malkhi and Dan Tsafrir, editors, *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*, pages 897–912. 2019

[76] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. Digraph: An efficient path-based iterative directed graph processing system on multiple gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 601–614. 2019

[77] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 461–476. 2018

[78] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC'21)*, pages 15–29. 2021