GDH+: A GPGPU-Empowered Gradient Data Hierarchy and Key-Value Separation for Optimizing LSM-Tree-Based KV Stores

HUI SUN, School of Computer Science and Technology, Anhui University, Hefei, China
XIANGXIANG JIANG, School of Computer Science and Technology, Anhui University, Hefei, China
JINFENG XU, School of Computer Science and Technology, Anhui University, Hefei, China
ENHUI WANG, School of Computer Science and Technology, Anhui University, Hefei, China
ZHENG FANG, School of Computer Science and Technology, Anhui University, Hefei, China
YI ZOU, Columbus State University, Columbus, United States
YINLIANG YUE, Zhongguancun Laboratory, Beijing, China
XIAO QIN, Computer Science, Auburn University, Auburn, United States

The rapid growth of unstructured data has driven the widespread adoption of LSM-tree-based key-value stores (KV stores). The write amplification resulting from compaction in LSM-trees causes a performance bottleneck. Existing solutions attempt to address this issue through key-value separation strategies. However, these studies fail to optimize the memory components of LSM-trees or provide efficient garbage collection (GC) strategies that achieve high performance while minimizing CPU overhead. These limitations motivate us to propose a GPGPU-empowered gradient data hierarchy and key-value separation for optimizing KV stores, named ***GDH+***. We utilize GPGPU acceleration for sorting and flushing operations, optimizing the memory components of the LSM-tree. Additionally, we enhance read performance with an LRU-based memory component that distinguishes between hot and cold data, in combination with an adaptive migration strategy. Furthermore, we propose an in-place GC strategy to reduce CPU overhead while maintaining high performance. GDH+ achieves a 2×, 1.5×, and 40% improvement in write performance, read performance, and CPU utilization, respectively, compared to state-of-the-art KV stores.

CCS Concepts: • **Information systems → Storage management**; **Information storage technologies**; **Hierarchical storage management;**

Additional Key Words and Phrases: LSM-tree, key-value store, GPGPU, compaction, GC

This work is supported in part by National Natural Science Foundation of China under Grants 62472002 and 62072001. Xiao Qin's work is supported by the U.S. National Science Foundation (Grants IIS-1618669 and OAC-1642133), the National Aeronautics and Space Administration (Grant 80NSSC20M0044), the National Highway Traffic Safety Administration (Grant 451861-19158), and Wright Media, LLC (Grants 240250 and 240311).
Authors' Contact Information: Hui Sun (corresponding author), School of Computer Science and Technology, Anhui University, Hefei, China; e-mail: sunhuiworking@gmail.com; Xiangxiang Jiang, School of Computer Science and Technology, Anhui University, Hefei, Anhui, China; e-mail: xxjiang@ahu.edu.cn; Jinfeng Xu, School of Computer Science and Technology, Anhui University, Hefei, Anhui, China; e-mail: jfxu@ahu.edu.cn; Enhui Wang, School of Computer Science and Technology, Anhui University, Hefei, Anhui, China; e-mail: ehwang@ahu.edu.cn; Zheng Fang, School of Computer Science and Technology, Anhui University, Hefei, Anhui, China; e-mail: zhfang@ahu.edu.cn; Yi Zou, Columbus State University, Columbus, GA, USA; e-mail: zhou_yi@columbusstate.edu; Yinliang Yue, Zhongguancun Laboratory, Beijing, Beijing, China; e-mail: yylhust@qq.com; Xiao Qin, Computer Science, Auburn University, Auburn, AL, USA; e-mail: xqin@auburn.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).
ACM 1544-3973/2026/06-ART58
https://doi.org/10.1145/3799980

ACM Reference Format:
Hui Sun, Xiangxiang Jiang, Jinfeng Xu, Enhui Wang, Zheng Fang, Yi Zou, Yinliang Yue, and Xiao Qin. 2026. GDH+: A GPGPU-Empowered Gradient Data Hierarchy and Key-Value Separation for Optimizing LSM-Tree-Based KV Stores. *ACM Trans. Arch. Code Optim.* 23, 2, Article 58 (June 2026), 27 pages. https://-doi.org/10.1145/3799980

1 Introduction

According to IDC DataSphere, unstructured data is expected to constitute 86.8% of the global data volume by 2027, amounting to an estimated 246.9 ZB [24]. **Log-structured tree** (**LSM-tree**)-based **key-value stores** (**KV stores**), a critical storage solution for unstructured data in big-data storage systems [13, 14, 21]. While existing KV stores have been extensively deployed across cloud computing platforms and **online transaction processing** (**OLTP**) systems, these designs cater to legacy workloads. However, distributed **machine learning** (**ML**) and publish-subscribe systems challenge the prevailing assumption that read-intensive operations dominate workloads. For instance, over 30% of workloads in X (formerly known as Twitter) involve a larger amount of writes than reads [32, 40].

Thus, to achieve a high-performance KV store, existing studies encompass data structure [5, 21], indexing techniques [19, 42], memory management, caching strategies [15, 35], and hardware acceleration [29, 31, 39, 43, 45]. The LSM-tree is used in KV stores to optimize write performance by transforming random writes into sequential ones. The memory component in the LSM-tree utilizes an ordered skiplist to facilitate fast reads and writes. To enhance the read performance, KV stores implement caching to store frequently accessed data in memory [1–3]. Additionally, the LSM-tree requires periodic compaction to merge SSTables with overlapping key ranges in adjacent levels to optimize storage space and improve access efficiency [29, 31, 39, 43, 45].

Compaction processes SSTables in neighboring levels, consuming substantial computational and I/O resources. As data volumes grow and system resources become constrained, the KV store becomes overloaded with compaction tasks, which block users' writes, increase write amplification, and degrade the performance of KV stores [6, 19, 21]. To address this issue, researchers have proposed various solutions, including key-value separation [6, 19–21, 33, 36, 42], pipelined compaction [44], task offloading [27], and hardware acceleration [29, 31, 39, 43, 45].

The key-value separation separately stores keys and values' addresses while the real values are placed in log files. This approach reduces write amplification and memory utilization. However, existing KV separation approaches fail to enhance read performance under modern workloads (*e.g.*, WPI workloads with 75% lookups and 25% writes) due to suboptimal caching strategies. First, these approaches rely on traditional caching methods instead of data popularity (hotness). Additionally, they encounter challenges in managing the tiered SSTables, where hot and cold KV pairs are stored similarly. Consequently, the access performance of hot KV pairs cannot be optimized according to the popularity, degrading read performance on high-latency devices. Second, most prior work retains the skiplist as the memory component, leading to $O(\log n)$ time complexity for writes/reads. Each write also triggers a subsequent sorting, which, in high-load and large-scale data cases, imposes significant synchronization overhead, impairing response time. Third, the reliance on **garbage collection** (**GC**) for managing invalid log data exacerbates this issue. The computational overhead of GC increases the flush and compaction time, hindering system performance.

These observations motivate us to propose a GPGPU-empowered gradient data hierarchy and key-value separation for optimizing KV store, named ***GDH+***, which leverages GPGPU to accelerate

Table 1. Comparison of CPU-Empowered and GPGPU-Empowered Sorting

Scheme	Sorting time (sec)	Throughput (MB/s)
CPU	29.3	124.9
GPGPU	1.3	176.6

a key-value separated KV store. We focus on reducing write amplification and CPU overhead under high-write and mixed-read-write workloads, while maintaining stable performance and low tail latency. Therefore, we designed the five key techniques in GDH+ as follows.

Specifically, to address the high insertion latency of the memtable skiplist and the I/O imbalance inherent in B/B+ trees, we **first** employ a hash table as an alternative to the skiplist. The hash table used as the memory component offers two key advantages for KV stores: (1) hash tables support $O(1)$ time complexity for both read and write operations and (2) hash tables can be efficiently sorted using GPGPU acceleration. Because sorting hash tables on the CPU is resource-limited and suffers from poor performance (see Table 1), we exploit the computational capability of GPGPUs to accelerate this process, improving sorting efficiency. **Second**, we implement an asynchronous hash table insertion and sorting, offloading the flush to GPGPUs to enhance its efficiency and overall system performance. **Third**, given that the cache hit rate in KV stores is influenced by the key popularity, with hot keys comprising the majority of user requests, we utilize **Least Recently Used** (**LRU**) caches to categorize keys based on access frequency. Hot data is stored in **persistent memory** (**PM**) to ensure low latency and high throughput, while cold data is stored on SSDs to balance storage cost and performance while leveraging SSDs' capacity and random access. **Fourth**, we design an adaptive migration strategy for SSTables based on their popularity, achieving the tiered storage. **Finally**, we introduce an in-situ GC strategy to optimize resource consumption in common GC methods, improving performance. These designs aim to address the performance bottlenecks in LSM-trees in terms of the limitations in the parallelism of the memtable components, the low cache hit rate for hot data, and the inefficiency and high overhead of traditional GC strategies. Unlike prior work that only optimizes compaction or caching, GDH+ integrates key–value separation, GPGPU-accelerated flush, PM-based SST-level tiering, and in-situ GC to address these challenges in a holistic manner. In summary, our contributions are listed as follows:

— We implement an asynchronous hash table insertion and sorting mechanism in GDH+, which accelerates sorting by leveraging the parallel computing resources of GPGPU. Combined with parallel flushing on GPGPU, this approach enhances the performance of KV store and flush efficiency.
— We utilize an LRU-cache-based storage component within GDH+ to distinguish between hot and cold data, ensuring that hot data is stored in PM and cold data is placed on SSDs. Furthermore, we design an adaptive migration strategy to support tiered storage and implement a heat-based cross-tier data migration strategy to improve read performance.
— We propose an in-situ GC strategy in GDH+ to optimize the GC process, minimizing the CPU overhead during GC.
— We conduct extensive experiments to evaluate performance of GDH+. The experimental results show that our memory component achieves a read hit rate of 36.87%. Under various hot and cold workloads, we achieve up to a 94.9% reduction in average latency and a 96.7% reduction in P99.9 latency. Additionally, with various read-to-write ratios, we achieve up to a 67.3% reduction in average latency and a 91.84% reduction in P99.9 latency.

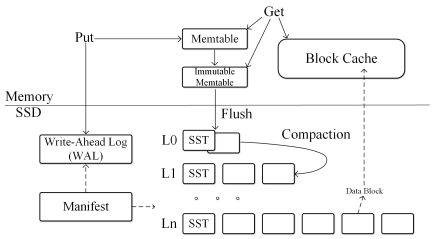


Fig. 1. LSM-tree structure.

2 Background and Related Work

2.1 Background

2.1.1 LSM-tree. The LSM-tree is an ordered hierarchical data structure for disk, allowing KV stores to process workloads with large-scale reads and writes. We employed LevelDB as an example to illustrate the LSM-tree, see Figure 1. The Memtable in memory employs a skiplist to maintain the order of KV pairs. Upon receiving a write, the KV pairs are first recorded in the log files and then inserted into the Memtable. When the Memtable size reaches a preset threshold, it is converted into an Immutable Memtable, and a new Memtable is created to process subsequent writes. The Immutable Memtable is then flushed to disk and becomes an SSTable. Once the number of SSTables in level L_k exceeds a threshold, an SSTable in level L_k is compacted with SSTables in L_{k+1} that have overlapping key ranges. This compaction process generates a new sorted order at level L_{k+1} and removes invalid SSTables. For reads, the Memtable is checked first. If no matching entry is found, a level-by-level lookup is performed on the LSM-tree.

Skiplist. The Memtable's structure is based on a skiplist, which is an ordered data structure. Each insertion performs a logarithmic-time search ***O(logn)*** to locate the position, followed by linking the new node to maintain order—averting the need for a separate sorting phase. The skiplist operations in the Memtable consist of a synchronized insertion (search + link), and the new entry becomes visible only after this insertion is completed. As a result, under high-load and large-scale workloads, the multi-level skiplist incurs significant performance overhead, degrading the system's response time. An alternative to the in-memory skiplist is the B/B+-tree, a balanced data structure that enhances both read and write performance. However, B/B+-trees increase memory consumption. Additionally, under workloads with frequent access, B/B+-trees may lead to unbalanced I/Os.

For a fixed number of KV pairs, the memory usage of B/B+-tree and skiplist depends on the KV size and node fill factor. For small-sized KV pairs (*e.g.,* 8B key and 16B value), a B+-tree may save memory when leaf nodes are fully filled. However, in practical scenarios, node fill rates are typically 70% [23], so the average metadata per key increases. If we have an 8B key and 8B pointer, the total metadata is 8m + 8(m+1) = 16m + 8. With branching factor $m = 100$, the internal node metadata is 1608B for 100 keys (i.e., 16.08B per key). For the skiplist, each node has an average of 1/(1-p) layers, where p is the probability growth of the layer. Each layer requires an 8B pointer. At p = 0.5, the average layers per node are 1.33, and the metadata is 8*1.33 $\approx$ 10.64B. In this case, skiplists exhibits lower memory usage than B+-trees. For large-sized KV pairs (*e.g.,* values > 1 KB), the metadata ratio can be ignored. B+-trees may waste space due to incomplete filling, whereas the dynamic allocation advantage of skip lists becomes more pronounced. B+-trees require more metadata (node pointers, markers), and these costs increase with smaller nodes (*e.g.,* a node containing only 10 keys). Therefore, B/B+-trees increase memory consumption in the case of dynamic write workloads with fill rates of 70%–80% and small-sized KV pairs.

Caching strategy. To mitigate disk I/Os caused by read operations, LevelDB employs a block cache. When a key is not found in the cache, the data block containing the key is fetched and cached. Due to spatial and temporal locality, subsequent read requests are likely to be served directly from the cache, reducing disk I/Os and enhancing read performance. However, this block cache strategy introduces a memory overhead, as it caches the entire data block, leading to a large-sized memory footprint. Additionally, the cache hit rate decreases under workloads with frequent data updates or unpredictable access patterns. An efficient method to improve memory utilization is to cache only keys instead of entire data blocks, increasing the cache hit rate.

2.1.2 Key-Value Separation. As data volume increases and access patterns become more complex, traditional LSM-tree structure faces challenges: write amplification and inefficient memory utilization. To address these issues, the key-value separation strategies are introduced to store keys and the value offset in the LSM-tree, while the actual values are stored in an append-only log file (*e.g.,* vlog files in WiscKey). By reducing the amount of data during compaction, this approach mitigates write amplification and reduces the size of the LSM-tree. Furthermore, by separating keys from values, memory utilization is enhanced, enabling the system to cache more keys within the same memory space, which improves the performance of the KV store. However, the key-value separation introduces a prominent challenge—GC. Two primary GC strategies are employed. The first one involves scan all KV pairs in the log files to determine the keys' validity and deleting invalid keys from the log file (e.g., WiscKey). The second strategy determines key validity based on sequences, which reduces the query overhead (*e.g.,* HashKV and DiffKV).

2.2 Related Work

In this section, we present existing studies on the performance improvement of KV stores through key-value separation, compaction optimization, and SSD/NVM-based LSM-tree optimization.

2.2.1 Key-Value Separation. KV stores without key-value separation offer high write performance but suffer from I/O amplification. The key-value separation approach can mitigate I/O amplification and enhance overall system performance by storing keys and values separately [16]. For instance, WiscKey [21], a popular KV-separated KV store, leverages both sequential and random access patterns of SSDs to improve system performance. However, WiscKey underperforms in update-intensive workloads due to its high GC overhead. To address this issue, HashKV [6] adopts a hash-based group to enhance the update and GC efficiency. Bourbon [11], built on top of WiscKey, introduces a learned index mechanism for key-value separation to improve query performance. UniKV [42] enhances read, write, and scan performance by integrating hash indexing and LSM-tree into a unified system. DiffKV [19] employs a fine-grained sorting of keys and values to maintain high scan efficiency, using full sorting for keys and partial sorting for values. Furthermore, DiffKV implements a fine-grained key-value separation strategy that balances performance under mixed workloads. FenceKV [33] focuses on optimizing range query performance while maintaining update efficiency. Through the partitioning strategy and key-range-based GC, FenceKV delivers high range-query and update performance under workloads.

Parallax [38], a system that embraces a hybrid KV placement strategy, is proposed to dynamically switch between in-place and log-based storage mechanisms based on the KV size. It implicitly performs GC for medium-sized key-values through in-place backfilling during compaction at the last level of the LSM-tree, thereby mitigating the GC overhead caused by key-value separation. GDH+ employs a generic in-situ GC to track and reuse invalid positions across levels during writes. It provides key-value separation to avoid large-value migration. The hotness-aware and hierarchical storage are leveraged to reduce write amplification and enhance overall system performance. GDH+ accelerates the sorting phase of memtable flush using GPGPU, and while integrating a key-value

separation strategy and a PM + SSD storage architecture to mitigate the synchronization overhead and resource contention during high-frequency inserts.

2.2.2 Compaction Optimization. LSM-tree improves write performance by batching merges of ordered data. However, traditional compaction methods mostly cause high tail latency and write amplification. To address these challenges, PebblesDB [22] uses a fragmented LSM-tree to store SSTables with overlapping key ranges in a guard, reducing write amplification, but at the cost of high query latency and space amplification. LDC [4] and ALDC [5] are designed to mitigate tail latency and enhance throughput through a bottom-up triggering mechanism and adaptive strategies. PStore [27] employs an asynchronous parallel scheme to reduce write amplification [34], while maximizing the utilization of both host and device-side computational resources. Additionally, hardware acceleration techniques, such as FPGA [17, 31, 43] and GPGPU (*e.g.,* an LUDA [39], gLSM [29], and GPU Comp-GDS [45]), enhance compaction performance. FPGA accelerators improve compaction performance and throughput through pipelining, combined with a key-value separation strategy, while gLSM leverages the parallel computational capabilities of GPGPU to accelerate compaction. These optimizations have demonstrated performance improvements across various applications.

The **near-data processing** (**NDP**) architecture [26][27][28][30] migrates computation closer to data in storage, performing computations at the data storage end to suppress data transfer overhead. The NDP aims to enhance system performance through a collaborative computing model; however, NDP can struggle to adequately address the issue of insufficient computing resources compared with GPGPU. Both NDP and GPGPU represent heterogeneous computing architectures.

2.2.3 SSD/NVM-based LSM-tree Optimization. Software frameworks such as hierarchical merging and key-value separation are designed to fully leverage the capabilities of high-performance storage devices [9, 10, 25]. These techniques improve both read and write performance while extending the device's lifetime. **Non-volatile memory** (**NVM**) is mostly employed in hybrid storage systems to achieve high throughput and low latency. MatrixKV [41] based on a hybrid NVM storage architecture utilizes NVM's byte-addressing capability to store the level L0 in NVM. MatrixKV also applies fine-grained columnar compaction on data at both L0 and L1 levels to minimize write stalls. SpanDB [7] optimizes KV stores for NVMe and commercial SSDs. SpanDB selectively allocates critical and frequently accessed data in a hybrid storage system. For instance, large data volume is stored on cost-effective and high-performance SSDs, while **write-ahead logs** (**WALs**) and data at the top level of the LSM tree are relocated to small-capacity and high-speed NVMe SSDs to enhance KV store's throughput. Hybrid KV stores, such as KVell [18] and MioDB [12], have become a popular solution in storage systems. WaLSM [8] uses NVMs to reduce write amplification in hybrid NVM-SSD storage systems and employs a reinforcement learning approach to balance update and search efficiency under workloads. GDH+ uses a PM-SSD hierarchical structure to store hot data in higher-performance PM, thereby improving access speed for hot data.

3 Motivation

In this section, we focus on the write latency caused by the in-memory skiplist, read performance bottlenecks for hot data, and CPU utilization and space amplification incurred by traditional GC strategies. These three challenges impose severe performance bottlenecks on KV stores under modern workloads with high-frequency access, large-scale writes, and uneven hot-cold data distribution. These challenges also motivate us to design GDH+ in this article.

3.1 Limitations of Memtable Components

Compaction consumes significant CPU time and I/Os, which leads to CPU bottlenecks. To improve the write performance of KV stores, most existing studies, such as LUDA and gLSM, offload the

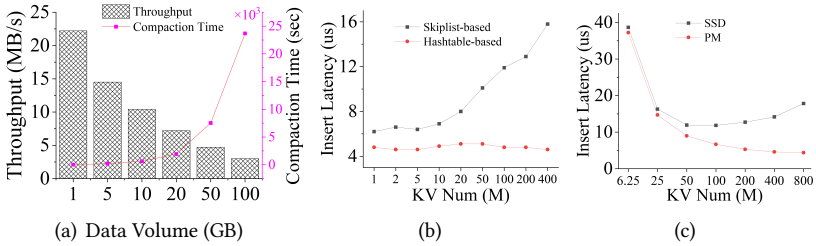


Fig. 2. Acceleration of compaction time and throughput on CPU-GPGPU under different data volumes (a), insert latency of CPU and GPGPU with the number of KV pairs ranging from 1M to 400M when value size is 128B (b), and insert latency of SSD and PM with value size from 64 KB to 128B and 100-GB data volume (c).

computation to heterogeneous devices alleviates the CPU burden, thereby enhancing the compaction efficiency. Although flush operations are similar to compaction in terms of CPU and I/O resource costs, the smaller data volume involved in flush makes CPU bottlenecks less likely. In contrast, as data volume increases, the amount of compacted data grows significantly. Since the computational capacity of heterogeneous devices is constrained by the data volume processed, this limitation reduces resource utilization and degrades the performance of KV stores. For example, Figure 2(a) shows that the compaction time increases significantly as data volume grows. This is because both keys and large-sized values must be transferred to device memory; however, only the keys are used in computation, while the large-sized values become invalid data, leading to excessive write amplification. Moreover, the limited memory space of heterogeneous devices restricts their ability to efficiently process large amount of data, reducing the performance of KV stores.

To address the challenge of suboptimal compaction in heterogeneous devices, we propose a key-value separation strategy to minimize write amplification during compaction, thereby improving compaction efficiency and overall system performance. One issue we identified is that, when handling large-scale data, the synchronization of insertion and sorting operations in the Memtable introduces additional overhead. Figure 2(b) illustrates the insertion latency as the number of KV pairs increases, using a 128-B value and varying the number of KV pairs from 1 million to 400 million. Insert latency refers to the average delay for inserting a KV pair into the memtable, which is the time from the write initiation of a KV pair to when it is written into the memtable. The results demonstrate that the insertion latency of the skiplist-based synchronous approach is substantially higher than that of the hashtable-based asynchronous approach. This discrepancy is primarily attributed to the ***O(1)*** insertion complexity of the hashtable. Moreover, the asynchronous nature of the hashtable-based approach allows insertions and sorting to occur concurrently, eliminating the need to complete one operation before starting the next. On the contrary, the skiplist-based synchronous approach requires sorting to wait until the insertion is complete. As a result, the write performance of the hashtable-based asynchronous approach is superior to that of the skiplist-based synchronous approach. Furthermore, the time complexity of lookup operations in the hashtable-based asynchronous approach is ***O(1)***, providing an additional performance advantage over the skiplist-based synchronous approach.

3.2 Processing Efficiency of High-Frequency Access Data

KV stores enhance read performance and optimize memory utilization by caching recently accessed data through the in-memory Memtable and block cache, which stores data blocks from disk in memory. However, the in-memory Memtable uses a skiplist structure, resulting in an O(log n) query time complexity. Similarly, for the block cache, the query time complexity is $O(logn)$, due to

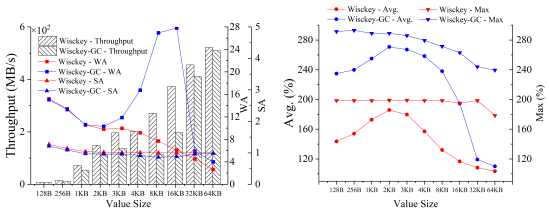


Fig. 3. Throughput, **write amplification** (**WA**), and **space amplification** (**SA**) of WiscKey and the GC-enabled version WiscKey-GC as the value size increases with a 100-GB data volumeB (a). Average and maximum CPU utilization of WiscKey and WiscKey-GC (b).

the need to query the index structure to retrieve data blocks. Therefore, in traditional KV stores, even when all data is cached in memory, the characteristics of the Memtable and cache lead to suboptimal query efficiency, negatively impacting the read performance of KV stores. Additionally, as data volume grows and access patterns become more diverse, the existing system continues to face challenges in processing both high-frequency access data and large-scale data. While the traditional key-value separation technique improves system performance to some extent, it still encounters a performance bottleneck when dealing with high-frequency accessed data (hot data). In scenarios requiring frequent access and updates, the access latency of hot data in SSDs is high, which impacts the overall response time of the system.

To address the aforementioned issues, we propose an improved scheme that combines the key-value separation technique and PM to optimize the performance of KV stores. We store all SSTables in PM because it offers speeds comparable to DRAM and the persistence of SSD, which significantly improves read efficiency and reduces access latency. Figure 2(c) shows the insertion latency (the average latency of inserting into Memtable after a write request arrives) as the number of KV pairs increases, with a constant data volume of 100 GB. As the value size decreases from 64 KB to 128B, the number of KV pairs increases from 6.25M to 800M, with the SSD latency becoming significantly higher than that of PM as the number of KV pairs increases. We categorize values as hot and cold based on access frequency, storing high-frequency accessed data (hot logs) in PM and low-frequency accessed data (cold logs) in SSD. Consequently, fast access to hot data enhances system responsiveness, while storing cold data in SSD optimizes storage costs.

3.3 Limitations of Existing GC

In existing GC strategies, WiscKey uses full-table scanning, while HashKV employs partition scanning. WiscKey scans a fixed-size data region during GC, writes valid data to a new log file, and discards invalid data. This strategy is similar to the merge sorting process in compaction. However, if the data region involved in GC is too large, it consumes substantial system resources, increases CPU utilization, and impacts the execution of other tasks. In HashKV's partition scanning, double pointers are used to move the invalid region into a valid region with more available space, a process that also consumes system resources. Both GC strategies have limitations: during GC, other tasks, such as flushing and compaction, are slowed down, lowering the throughput.

To demonstrate the limitations of existing GC strategies, we conduct experiments using WiscKey as a case study. We evaluate its performance in terms of throughput, write amplification, space amplification, and average (maximum) CPU utilization, considering both active and inactive GC scenarios. The workloads are configured with varying value sizes and a 100-GB data volume. Figure 3(a) shows that GC operations increase write amplification and reduce throughput,

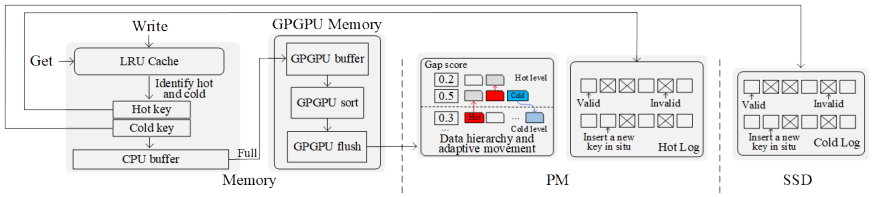


Fig. 4. Overview System of GDH+.

especially for value sizes ranging from 2 KB to 32 KB. Additionally, GC leads to high resource utilization. Figure 3(b) illustrates both average and maximum CPU utilization with and without GC. Although GC results in higher write amplification, lower throughput, and increased CPU utilization, these factors do not significantly reduce space amplification, making the use of GC less beneficial.

Therefore, we propose a location table in the KV store to record the starting position and size of invalid spaces. When writing a value, if an invalid space exists, the system overwrites it with the new value, thereby releasing the invalid space. If the invalid space is empty, the log file is simply appended. This strategy prevents user operations from being blocked by large-scale GC and reduces CPU utilization, space amplification, and I/O overhead.

4 Design

To address the above challenges, we design GDH+ with GPGPU-assisted memory processing, hierarchical log storage for hot and cold data, CPU-GPGPU heterogeneous co-processing, data hierarchy and adaptive movement, and in-situ GC.

4.1 Overview System

In this section, we provide a detailed description of our proposed ***GDH+***, comprising DRAM, PM, SSDs, and GPGPU. As shown in Figure 4, when receiving a user's write, GDH+ checks whether the key exists in the LRU cache. If the key exists, GDH+ updates its access count; otherwise, GDH+ moves the key to the head of the LRU cache and initializes its access count to one. The access count is always logged alongside the key. We categorize the key as hot or cold based on its access count, with cold keys being written to the slower-performance SSD, while hot keys are written to the faster-performance PM. Then, a key-value separation is performed, with keys written to the KV Buffer. Once the KV buffer is full, the KV pairs are transferred from the buffer to the GPGPU for sorting and flushing. The new SSTable files are stored in PM as an LSM-tree, with a hotness value recorded in the metadata of each SSTable. With the key-value separation, a compaction procedure handles keys and value pointers, thereby reducing compaction overhead. In addition, to minimize GPGPU data transfer and resource overhead, we use GPGPU to accelerate the sorting and flushing of the KV buffer. Using this metadata, the system can efficiently move hot and cold data, enabling adaptive data movement. In addition, an in-place GC strategy is realized by recording the addresses of invalid records during log writing. Instead of appending the newly written key to the log, GDH+ checks for invalid spaces within the log. If found, the key is written to the available space, overwriting the invalid data and realizing in-place GC.

4.2 Hierarchical Log Storage

In this section, we leverage spatio-temporal locality through hot-cold identification and hierarchy storage, placing hot values in PM and cold data on SSDs. This approach improves hit rates, reduces read latency, alleviates SSD read/write contention, and balances cost with durability.

4.2.1 Recognition of Hot and Cold Data. Data exhibit spatiotemporal locality, with frequently accessed (hot) data likely to be accessed again in the future. However, simply classifying data as hot or cold does not fully exploit the spatiotemporal locality. To address this issue, we employ an LRU-based module to discriminate hot and cold data. Then, we partition the storage space for hot and cold data across storage devices with different speeds. This strategy maximizes the cache hit rate, as data with varying hotness will be accessed differently in the future. Given a key, GDH+ first checks if it is present in the LRU cache. If found, its access count is updated; otherwise, the access count is initialized to zero. And the classification of a key as hot or cold is based on its number of accesses. A key is initially classified as cold with a weight value of 0. After an access, it is classified as warm with a weight value of 5. With multiple accesses, it is classified as hot with a weight value of 10. The access count and weights of each key are persistently stored in the SSTable, with the weights aggregated to calculate the hotness of the SSTable.

4.2.2 Hierarchical Log Storage Strategy. In GDH+, we employ a key-value separation strategy, with values stored in log files. During lookup, a value's offset in a log file is determined by its corresponding key. Using the log file number and offset, the actual value can then be located and retrieved. We store log files on SSDs because they do not undergo frequent write operations. Instead, log files require a substantial number of read operations. During compaction, data must be written to SSDs, which creates a performance bottleneck. SSDs struggle to handle both read and write operations for large-scale data, impairing read performance. PM offers a speed between that of DRAM and SSDs, enabling high-performance data access.

To overcome the limitations of existing SSDs, we propose a hierarchical log storage strategy that classifies data into hot and cold logs based on key access frequency. Hot logs, accessed more than once, are stored in PM to ensure high-speed retrieval of frequently accessed data, thereby reducing latency and boosting read performance. Whereas, cold logs, accessed only once, are stored on SSDs to improve overall storage utilization. By distributing hot and cold logs across storage devices with varying speeds, we address the read performance limitations of SSDs when handling large volumes of data. While SSDs are more cost-effective than PM, they are slower. Storing cold logs on SSDs optimizes storage costs while meeting the lower performance requirements for infrequently accessed data. This tiered storage strategy enhances space utilization by preventing high-cost PM from being occupied by low-frequency data. Additionally, PM ensures the persistence of hot data during a system crash, thereby improving data reliability.

4.3 Heterogeneous Co-Processing

In GDH+, the sorting/deduping/encoding batch processing in a flush operation are offloaded to the GPGPU, which is pipelined with CPU-side insertion to mitigate CPU overhead and write stalls, thereby enhancing throughput and stabilizing tail latency. The co-processing module consists of the GPGPU Cache, GPGPU Sort, and GPGPU flush modules. To enhance computational efficiency and minimize data movement, GDH+ performs hash table sorting and flush on the GPGPU.

GPGPU Cache. The CPU and GPGPU caches are structured as contiguous arrays, with KV pairs stored sequentially. This arrangement enhances data movement during KV pair insertion and sorting, and allows multi-threaded parallel access to the KV pairs in the GPGPU cache.

The GPGPU cache primarily holds KV pairs that are ready to be sorted. When a KV pair is written, it is initially stored in the CPU cache, which has limited capacity. Once the CPU cache reaches its maximum capacity, the data is transferred to the GPGPU cache, and the CPU cache is reclaimed for new KV pairs. These KV pairs are then sorted by the GPGPU sort module to produce ordered KV pairs. Additionally, when a CPU cache miss occurs, the KV pairs in the GPGPU cache can be

ALGORITHM 1: GPGPU Sort and Flush

```
1: Input:
2:     KVs: Key-value pairs in GPGPU cache
3: Output:
4:     SSTables: A set of SSTables
5:
6: Sort KVs in the GPGPU in dictionary order
7: Put the duplicate KVs to the end of all the keys by
   the sequence field
8: Delete the duplicated KVs in the GPGPU
9: KVs are divided into several SSTables
10: OpenSSTables()
11: Function Flush():
12:     Allocate device memory and move constant to
   constant memory
13:     Function kernel EncodeBlocks():
14:         sst_idx ← threadIdx.x, block_idx ← blockIdx.x
15:         Select the SSTable based on sst_idx and select
   the block based on block_idx
16:         EncodeDataBlocks()
17: EncodeIndexBlocks()
18:     End of kernel EncodeBlocks
19:     Transfer the encoded results to CPU memory
20:     EncodeMetaBlocks() and EncodeFooters()
21:     Record the metadata of all SSTables
22:     Write all SSTables to the PM
23: End of Flush
24: FinishSSTables()
25: ReleaseSources()
```

accessed for fast data retrieval, taking advantage of the GPGPU's parallel computing capabilities. If a thread in the GPGPU hits the key being searched, it broadcasts the result to other threads, causing them to terminate their lookup. If no threads hit the key, the lookup process is ended.

GDH+ performs sorting operations using a skiplist on the CPU. As user keys are inserted into a memtable, the skiplist sorts the incoming keys. However, using the skiplist on a GPGPU is challenging because the data arrives with delays rather than simultaneously, resulting in inferior performance compared to the CPU. To overcome this, we adopt a parallel sorting approach in the GPGPU cache, leveraging the GPGPU's parallelism to enhance sorting efficiency.

The GPGPU Sort module sorts unordered KV pairs in the GPGPU cache into ordered KV pairs for subsequent operations. To this end, we design a key-specific sorting algorithm to sort all keys in the GPGPU cache, which involves three steps. First, all keys in the GPGPU cache are sorted in dictionary order (see Line 6 in Algorithm 1). Our comparison rule sorts keys character by character. For duplicate keys, GDH+ uses the sequence field to determine the order, placing the newer key (i.e., key with the larger sequence number) before the older key. Second, the keys are de-emphasized after sorting. In this step, for duplicates, the older key is moved to the end of the sequence (Line 7 in Algorithm 1). Third, the duplicate keys at the end of the list are removed (Line 8). As a result, the GPGPU cache contains only ordered, non-duplicated keys, and the total number of keys can be determined. This count is then used to decide the number of SSTables involved in the flush.

GDH+, which adopts a key-value separation, merely transfers keys, value pointers, and small metadata to GPGPU memory. With a 16-B key, 8-B value, and additional metadata, a KV pair occupies 35 Bytes. Therefore, when a dataset size reaches millions or billions, the GPGPU memory cost remains within the range of tens of MB to a few GB, which is fully compatible with the maximum memory requirements of GDH+. Consequently, the GPGPU parallel execution algorithm for flush described in this article does not exceed the GPGPU's memory capacity.

GPGPU Flush. The flush operation on the CPU serially encodes a set of ordered KV pairs, which forms an SSTable to be written to disk. This operation involves significant computation and I/Os. Although it does not require encoding more KV pairs than the compaction process, the flush remains a computing-intensive operation, demanding considerable CPU resources. As a result, it competes with the compaction for limited CPU capacity. To address this issue, we offload flush operations to the GPGPU, thereby reducing CPU resource consumption for compaction. This

approach offers two key benefits. First, by harnessing the GPGPU's substantial parallel processing power, we accelerate the flush process, thereby improving throughput. Second, by concurrently encoding multiple SSTables, we further enhance the overall performance of the KV store.

These ordered KV pairs are encoded into multiple SSTables, which are written to the PM subsequently through flush. Since the number of KV pairs in the GPGPU cache exceeds that of an SSTable, these KV pairs are partitioned into multiple SSTables. These SSTables are encoded in parallel, leveraging the GPGPU's parallel computing (Line 9). To optimize the encoding process, we standardize the size of data blocks in the SSTables, ensuring that each SSTable and its data block are mapped to a GPGPU thread. Each thread processes either an SSTable or a data block, enabling the parallel encoding of multiple SSTables. The results are stored in the GPGPU encoding result buffer (see Lines 12–18). Once encoding is complete, the encoded result buffer is transferred to CPU memory (Line 19) for processing, which includes encoding the footer for each SSTable (Line 20) and saving the metadata (Line 21). Then, each SSTable is written to PM (Line 22). GDH+ primarily relies on the WAL to recover from an interrupted flush. In the event of a crash, the system recovers all data through log files so that no writes are lost, and GPU-accelerated flush operations do not compromise data consistency.

We opt to store SSTables in PM because of its high access performance, which is close to DRAM and faster than SSDs. Storing SSTables in PM reduces latency, particularly for frequently accessed data. Moreover, writing multiple SSTables to disk benefits greatly from PM's higher operational concurrency and lower write amplification, making it advantageous for compaction and mitigating the performance impact on KV stores. It is not the PM itself that directly reduces write amplification, but rather the operational strategies enabled by PM in KV stores that have the potential to reduce write amplification. These strategies help mitigate the performance impact of compaction on KV stores. Moreover, with key-value separation, each SSTable can store a greater number of KV pairs, even with large datasets. As a result, a small-sized PM is sufficient to hold these SSTables. Therefore, we choose to store LSM-tree structures in PM to further enhance the performance of GDH+

Upon receiving newly written KV pairs at the GPGPU, they are processed before being written to PM. All three steps are executed on the GPGPU to minimize data movement and mitigate performance degradation caused by data transfers. Furthermore, GDH+ fully utilizes the parallel computing resources of the GPGPU to significantly accelerate sorting and flushing operations, compared to traditional CPU-based implementations. As a result, the heterogeneous co-processing module enhances the overall performance of KV stores.

Notably, in GDH+, we offload the memtable's flush operation to the GPGPU, leveraging the GPGPU's parallelism to accelerate sorting and encoding tasks in the flush operation, thereby achieving high throughput. To mitigate the data transmission overhead, GDH+ adopts a key-value separation to transfer only key and value pointers, whereas large values are retained on the SSD. Sorting and encoding are executed on the GPGPU to construct a new SSTable. Because this procedure is pipelined with the CPU side's KV insertion to hide the data transfer latency, the memtable-to-GPGPU technique improves performance without introducing any new bottlenecks. In addition, GPGPU is not used to accelerate the compaction procedure. First, when the key-value separation is incorporated in GDH+, the compaction procedure primarily processes key and value pointers. During the CPU-based compaction, the CPU reduces the movement of large-sized values, significantly cutting back data-movement overhead. Second, GPGPU-accelerated flush is asynchronously performed with the procedure of inserting KV pairs into the memtable on the CPU, bolstering write and flush performance. Third, if the GPGPU simultaneously speeds up both flush and compaction operations, the burdened GPGPU will hinder performance improvement.

GDH+ avoids offloading compaction to the GPGPU because key-value separation and a PM-resident LSM-tree have already reduced compaction to a lightweight CPU-bound task; instead, the

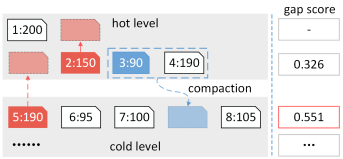


Fig. 5. Movement of hot and cold data. In value 1:200, "1" is the file number and "200" is the hotness of the file.

GPGPU accelerates memtable flushing—enabling efficient CPU-GPU pipelining that boosts overall throughput without introducing NVMe-GPGPU transfer overhead or resource contention.

4.4 Data Hierarchy and Adaptive Movement.

In this section, we elaborate on a data layering and adaptive movement mechanism anchored on access hotness and hierarchical movement, moving hot data upward and cooling the data before moving it downward to optimize read performance. We model a database as a vast data repository, akin to an ocean, where data is classified as "hot" and "cold" based on access frequency. Hot data, which is frequently accessed, is moved upward from lower levels to higher levels in the hierarchy to facilitate faster access. This process is referred to as an "upward" operation. Conversely, cold data, which is infrequently accessed, is shifted downward to lower levels of the hierarchy, known as the "downward" operation. This dynamic data movement allows for efficient cache management, ensuring that the cache adapts to changing access patterns to enhance overall performance.

To manage SSTables and their key-value metadata, the disk component employs a hot and cold hierarchy. Each SSTable is assigned a hotness value based on the access frequency of its keys, which determines whether it is classified as a hot or cold SSTable. The SSTable granularity for hot-cold classification simplifies the migration process and reduces migration cost. If a finer-grained hot-cold tiering approach (e.g., key) is applied, it would track and move a large number of small data fragments: significant metadata overhead is introduced with an increased fragmentation in PM. With coarser-granularity SSTables, multi-megabyte units are well-suited to modern SSDs and keep the migration algorithm simple.

The hotness value facilitates hierarchical data movement: cold SSTables are moved downward to lower levels, while hot SSTables are moved upward to higher levels. This hierarchical storage strategy effectively utilizes storage devices with varying performance characteristics. For example, SSTables with lower hotness values are stored on slower devices such as HDDs, while those with higher hotness values are placed on high-performance devices like SSDs and NVMs. This design achieves an efficient hot and cold SSTable storage hierarchy. We define the first three levels of the LSM-tree as hot levels and the subsequent levels as cold levels. Additionally, we introduce a gap score to measure the difference between the hottest SSTable in a level ($H_{max} = \max\{H_i\}, i \in \{1, 2, \ldots, N\}$) and the average hotness of all SSTables in that level ($\bar{H} = (\sum_{i=1}^{N} H_i)/N$). The gap score for a level is calculated as $S_{gap} = \frac{H_{max}}{\bar{H}} - 1$, where N is the number of SSTables in the level. However, the gap score cannot be calculated for level 0, as it is the highest level in the LSM-tree, and no SSTables can be moved further upward. Similarly, cold SSTables at the lowest level cannot be moved downward.

To determine when to move the hot and cold SSTables, we employ gap scores. For each level, we calculate the gap score and select the level with the highest score. Hot and cold SSTables are then transferred to this level, triggering the adaptive data movement. Figure 5 shows that computing the gap score at level 0 is not feasible. For the second and third levels, we compute the gap scores using

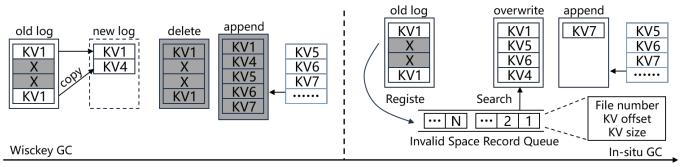


Fig. 6. Comparison of two GC strategies.

$S_{gap} = \frac{H_{max}}{H} - 1$. We omit the gap score of other levels. The gap score for the third level exceeds that of the second level, indicating that the SSTable at the third level has a greater disparity in hotness compared to the SSTable at the second level. Therefore, it is beneficial to transfer the SSTable from the third level to the second level while moving the hottest SSTable to the third level.

Hot and cold SSTable migration can occur either before or after compaction. However, post-compaction migration may negate the benefits of compaction, as migrating SSTables afterward could result in overlapping key ranges, leading to frequent compaction cycles. Pre-compaction migration, on the other hand, is more efficient as it helps avoid such overlaps, ensuring that the benefits of compaction are preserved. Hot SSTables without key range overlap are directly moved to the upper level. However, if there is overlap, a bottom-up compaction is triggered to generate new SSTables and update their hotness values. Cold data migration downward is simpler, involving only gap score calculations and metadata updates. During compaction, SSTables with the lowest hotness values are prioritized, and undersized files are filtered out to ensure efficiency.

4.5 In-situ Garbage Collection

GDH+ achieves in-place GC to lower write amplification by maintaining invalid locations to be reused during writes, without incurring additional background GC overhead. The GC in WiscKey selects a fixed-size data region and queries its LSM-tree to determine the validity of the KV pairs in that region. The valid KV pairs will be copy-rewritten to a new log file in appending-write mode, while the invalid KV pairs are discarded. After the scan of the log file, the log file processed in GC is deleted. Subsequently, any new KV pairs are written only in a new log file in append-write mode.

In GDH+, we introduce an in-situ GC strategy (see Figure 6) to address the issue of invalid space that cannot be reclaimed during the lightweight online GC process. The invalid space in GDH+ is tracked in the log file using metadata associated with invalid KV pairs during compaction. This metadata includes key information, including the log number, the offset of the invalid space, and its size, but does not require the KV pair data itself to be recorded. The metadata is stored in a queue, with an atomic variable managing the queue size. New metadata for invalid spaces is added to the front of the queue, and the queue size is incremented by one. When metadata is processed, it is deleted from the back of the queue, and the queue size is decremented accordingly. Data is then written into the invalid space based on the metadata, ensuring efficient space reclamation.

In our proposed queue-based model, the recording and recycling of invalid space follow a producer-consumer paradigm, ensuring that the recording and recycling operations do not interfere with each other. The thread-safety mechanism ensures that this approach is viable in multi-threaded environments. We also verify whether the invalid space is sufficient to store newly inserted KV pairs. If the space is insufficient, the KV pairs are written to the most recent log file in append-write mode. Data outside the log file is written randomly in the available space. In in-situ GC, the SSTable containing the invalid space cannot be the same as one of the most recent log files, as the append-write mode cannot support random writes. A large data region in GC requires a significant log-file

table. However, our proposed queue-based method eliminates the need for a large-sized log table in the data region undergoing GC. The queue size is dynamically adjusted based on the performance gap between the producer and consumer threads. The in-situ GC mitigates the accumulation of invalid space and minimizes space amplification, reducing the space overhead of KV stores.

The in-situ GC method tracks freed space segments and reuses them for storing new values rather than the bulk-copy approach. This design mitigates the write amplification and performance degradation caused by background GC. A queue records the locations of freed values in the value log. Since new writes preferentially reuse these holes, the value log does not expand indefinitely due to invalid data; instead, new values fill the vacancies.

5 Performance Evaluation

5.1 Experimental Setup

We conducted extensive experiments on a server with dual Intel®Xeon®E5-2630 v4 processors, each featuring 10 cores and 20 threads, complemented by 64 GB DDR4 memory and a 480 GB Intel Optane 900P Series PCIe SSD. The operating system was Ubuntu 22.04, running on the Linux Kernel 6.5.0-14-generic. For acceleration, we utilized an NVIDIA Tesla P100 GPGPU with 3584 CUDA cores and 16GB of memory, connected via a PCIe 3.0 ×16 interface. In GDH+, 32 GB memory was allocated for PM. GDH+ was developed with CUDA Toolkit v12.3 and LevelDB v1.20.

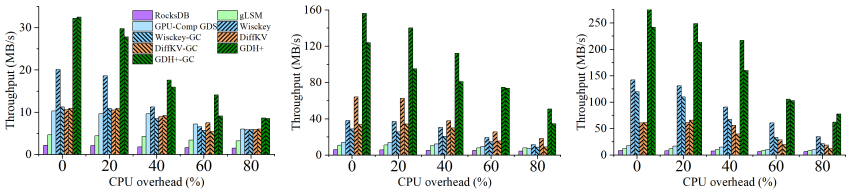
We use WiscKey as the baseline, alongside RocksDB, ALDC, DiffKV, GDH, gLSM, and GPU-Comp GDS. WiscKey, DiffKV, and GDH also employ the key-value separation strategy. gLSM and GPU-Comp GDS utilize GPGPU acceleration for compaction, whereas our proposed GDH+ employs the GPGPU to accelerate insertion sort and flush operations. Additionally, ALDC introduces an adaptive lower-level-driven compaction strategy, which compares with the adaptive hot-cold data migration strategy in GDH+. Furthermore, all tested KV stores in this study are configured to use the leveled compaction strategy as a default setting. The Bloom filter embraces a default size of 10 bits per key to maintain point query performance for all KV stores supporting Bloom filters. In addition, the block cache size is set to 64 MB for all the KV stores. The SSTable size is explicitly defined: LevelDB-based KV stores adopt 2 MB, whereas RocksDB-based ones use a default size of 64 MB. The memtable size of LevelDB and RocksDB is 4 MB and 64 MB, respectively. Each KV store is configured with its default number of background compaction and flush threads. The default WAL sync mode is employed in the tested KV stores.

We employ the default queue depth for the tested SSD in GDH+. We measured the storage I/O throughput of GDH+ during flush/compaction under YCSB Load. GDH+ maintains average write throughput 125.1 MB/s (range: 117.6–133.9 MB/s) across queue depths with queue depths of 16, 32, 64, 128, 256, and 1,023. Under YCSB-A (mixed read-write), it achieves 30.0 MB/s read and 75.52 MB/s write throughput. This queue-depth-independent performance confirms storage I/O is not the bottleneck; thus, GDH+ addresses the computational and memory management constraints which limit performance.

5.2 Basic Experiments

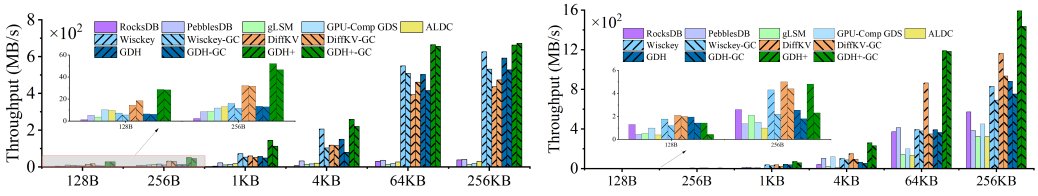
GDH+ focuses on write throughput, read throughput, write latency, write amplification, space amplification, and CPU utilization. In all tests, the key size is 16 B, six value sizes, and a 100-GB data volume. Additionally, we assess the mixed read-write performance using the YCSB-C workload with three value sizes and 100 GB data volume.

5.2.1 CPU Overhead. We use stress-ng to evaluate the performance of KV stores under CPU loads and value sizes. This approach captures the performance variations of the KV stores in the case of different CPU loads. As shown in Figure 7, GDH+ outperforms other KV stores under



(a) 128 B (b) 1 KB (c) 4 KB

Fig. 7. Throughput of GDH+ in cases of various CPU overheads.



(a) Write throughput (b) Read throughput

Fig. 8. Performance of GDH+ in terms of write throughput (a) and read throughput (b).

workloads. Compared with GPU-Comp GDS, GDH+ delivers up to 11.1× higher throughput. For example, with 1 KB values and no CPU load, GDH+ achieves 11.1× the throughput of GPU-Comp GDS. At 80% CPU load, the advantage remains 7.2×. GDH+ with active GC—GDH+-GC—shows no significant performance degradation despite increased CPU load. In this case, GDH+ outperforms other GC-enabled KV stores. These results suggest that GDH+-GC incurs lower CPU overhead while maintaining high performance, demonstrating the advantages of in-situ GC in GDH+-GC.

5.2.2 Write Throughput. We use db_bench to evaluate the write performance of GDH+, see Figure 8(a). GDH+ outperforms other KV stores under workloads with value sizes from 128 B to 256 KB. This advantage is particularly evident with small-sized values. Compared to RocksDB, DiffKV-GC, GPU-Comp GDS, and gLSM, GDH+-GC achieves 21.92×, 1.53×, 2.77×, and 7.70× higher throughput for small value sizes such as 128 B, respectively. The superior performance of GDH+ is primarily attributed to its hash-based asynchronous insertion and sorting strategy, which offloads sorting to the GPGPU for parallel execution with insertion. GPGPU-accelerated flush operations further boost performance. Compared to RocksDB, DiffKV-GC, GPU-Comp GDS, and gLSM, GDH+-GC achieves 29.47×, 1.90×, 12.63×, and 17.68× higher throughput for large values such as 4 KB, respectively.

Additionally, GDH+ stores SSTables and hot logs in PM, which offers higher write performance compared to traditional storage devices, thereby enhancing write performance. However, with large-sized values, GDH+-GC does not significantly improve performance. This is because, as the value size increases and the total data volume remains fixed, the number of KV pairs in sorting and flush operations decreases. Since GPGPU advantages are most apparent with large datasets, it cannot fully exploit its parallel processing capability in this case. Moreover, GDH+-GC suffers from GC overhead, which cannot significantly enhance write performance in these scenarios.

5.2.3 Read Throughput. In Figure 8(b), GDH+ exhibits superior read performance. With large-sized values, the read performance of GDH+ significantly surpasses that of other KV stores by a

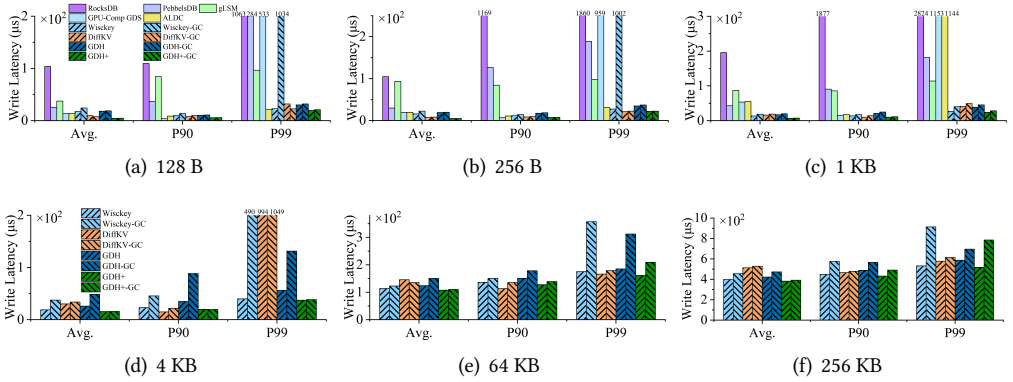


Fig. 9. Write latency of GDH+ under db_bench with various value sizes.

considerable margin. Compared to RocksDB, gLSM, GPU-Comp GDS, and ALDC, the throughput for large values (4 KB and 64 KB) improves by 3.2× to 6.4×, 8.6× to 14.2×, 5.9× to 10.5×, and 9.1× to 36.5×, respectively. This is primarily attributed to the adoption of key-value separation and an efficient caching mechanism. The key-value separation strategy reduces the size of the keys and values cached, storing a larger number of KV pairs in the cache. Additionally, the caching mechanism selectively stores KV pairs rather than entire blocks, thereby enhancing cache hit rates and improving read performance. Furthermore, we store both SSTables and hot logs in PM, whose high read/write throughput further boosts read performance. In contrast, WiscKey retrieves a reference to the actual value from the SSD during a read request and subsequently fetches the corresponding value from the log. Since SSDs are featured with lower read/write performance than PM, GDH+ leverages the superior read/write performance of PM to fulfill read requests more quickly; therefore, GDH+ outperforms the baseline in terms of read performance. Compared to RocksDB, ALDC, GPU-Comp GDS, and DiffKV, GDH+ achieves 7.6×, 48.1×, 13.2×, and 1.9× higher throughput, respectively, when reading large values (1 KB to 256 KB), respectively.

5.2.4 Write Latency. We evaluated the write latency of GDH+ in Figure 9. We studied average write latency, 90th percentile latency (P90), and 99th percentile latency (P99) under workloads with various value sizes. GDH+ outperforms or matches the performance of other KV stores in most metrics. For example, the P99 tail latency of GDH+-GC is reduced by 98.99%, 75.07%, 97.53%, 42.06%, and 37.18% compared to RocksDB, gLSM, GPU-Comp GDS, DiffKV-GC, and GDH-GC, respectively. Specifically, compared to the baseline WiscKey, GDH+ delivers lower average, P90, and P99 latencies under workloads. This is because GDH+ uses a hash table instead of a skiplist, combined with asynchronous insertion and sorting. Additionally, the approach to leveraging GPGPU for sorting and flushing further enhances the write efficiency of GDH+ and reduces latency. In contrast, although gLSM also benefits from GPGPU acceleration, GDH+ achieves lower write latency. This discrepancy highlights a limitation of gLSM's GPGPU acceleration for compaction: the real values consume a substantial portion of the GPGPU memory, restricting the utilization of computational resources for merge-sorting in compaction. As a result, the efficiency of compaction decreases, which blocks write requests and increases latency.

5.2.5 Write Amplification. As illustrated in Figure 10(a), we studied the write amplification of GDH+, which outperforms other KV stores under workloads with various value sizes. For instance,

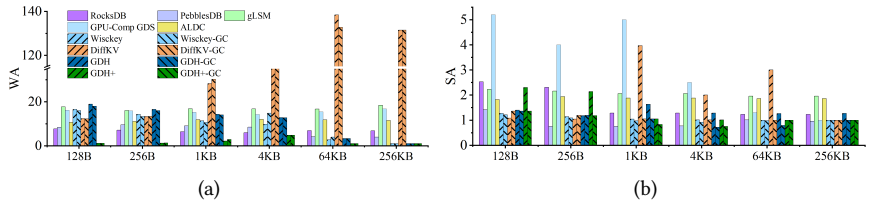


Fig. 10. WA (a) and SA (b).

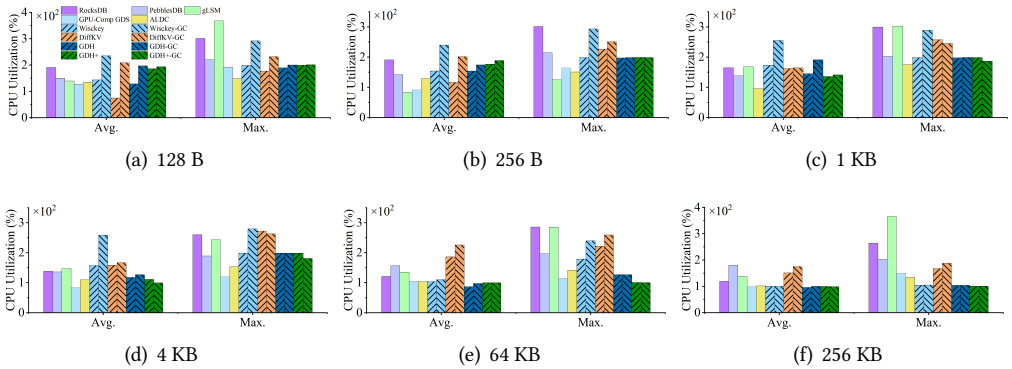
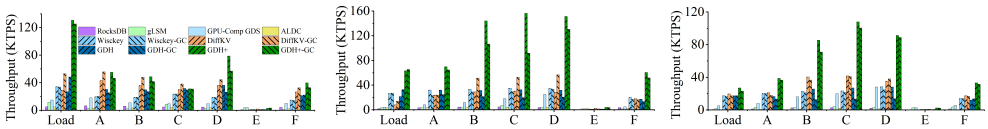


Fig. 11. CPU utilization of GDH+ under db_bench with various value sizes.

when the value size is 128 B, our GDH+ reduces write amplification by 84.68%, 92.86%, 88.92%, 90.47%, 93.76%, 93.33%, and 93.2% compared to RocksDB, WiscKey, ALDC, DiffKV, GDH, gLSM, and GPU-Comp GDS, respectively. GDH+ demonstrates an edge write amplification than alternative KV stores. This is because GDH+ uses a hot-cold cache, which temporarily stores both hot and cold keys. Once the cache is full, the data is transferred to GPGPU memory, where the KV pairs are grouped based on SSTable sizes. These groups are then written to disk as separated SSTables. The hot-cold cache with 256 MB can hold up to 1 million KV pairs, which are divided into 10 SSTable groups. The key ranges in these SSTables are non-overlapping, enabling them to be flushed to disk without requiring compaction. Compaction is only triggered when overlapping key ranges are encountered during subsequent flushes. This approach minimizes compaction operations, thereby reducing write amplification.

5.2.6 Space Amplification. We evaluated the space amplification of KV stores and compared the GC schemes of WiscKey (i.e., WiscKey-GC) and our proposed GDH+-GC. Figure 10(b) shows that GDH+-GC exhibits significantly lower space amplification than WiscKey-GC when the value size ranges from 256 B to 4 KB. This highlights the improved space utilization of our in-place GC. However, for 64 KB and 256 KB values, both GC schemes show similar space amplification.

5.2.7 CPU Utilization. We study the average and maximum CPU utilization of GDH+ in Figure 11. GDH+ achieves lower average and maximum CPU utilization than other KV stores in some cases. For example, with 1 KB and 4 KB values, the average CPU utilization of GDH+ and GDH+-GC is lower than that of other KV stores. With a 1-KB value, the CPU utilization of GDH+ and GDH+-GC is 46.7% and 12.9% lower than that of WiscKey and WiscKey-GC. GDH+ exhibits lower CPU utilization under workloads with large-sized values.



(a) 1 KB (b) 4 KB (c) 16 KB

Fig. 12. Throughput of GDH+ under YCSB-C with various value sizes.

5.2.8 YCSB-C. We used YCSB-C to test the performance of RocksDB, gLSM, ALDC, WiscKey, GPU-Comp GDS, WiscKey-GC, DiffKV, DiffKV-GC, GDH, GDH-GC, GDH+, and GDH+-GC under different value sizes (1 KB, 4 KB, and 16 KB), see Figure 12. Throughput of YCSB-C is measured in operations per second, reported in KOPS (thousands of operations per second). Nonetheless, comparing our solution with AC-Key in Section 5.3.4, we employed QPS (queries per second) as the metric. This choice was driven by the fact that AC-Key customized QPS as its performance benchmark. We adopted the same metric for the comparability purpose. We take the 1-KB value as an example. Under write-intensive workload, GDH+ delivers significant performance improvements of 2563.27%, 1017.95%, 781.76%, 2625.00%, 279.31%, 347.26%, and 385.79% compared to RocksDB, gLSM, GPU-Comp GDS, ALDC, WiscKey, DiffKV, and GDH, respectively. The performance of GDH+-GC improves by 2,502.04%, 972.6%, 746.62%, 2,498.96%, 271.57%, 135.99%, and 159.64% compared to RocksDB, gLSM, GPU-Comp GDS, ALDC, WiscKey-GC, DiffKV-GC, and GDH-GC, respectively. This is because GDH+ uses GPGPU to accelerate the sorting process, which improves the flush performance and optimizes the computational performance of the KV store.

Under read-intensive workloads B, C, and D, GDH+ outperforms RocksDB, gLSM, ALDC, WiscKey, DiffKV, and GDH under workloads B and D. The performance of GDH+GC under workload B improves by 608.47%, 2,100%, 287.04%, 1,166.67%, 120%, and 54.67% compared to RocksDB, gLSM, GPU-Comp GDS, ALDC, WiscKey-GC, and GDH-GC, respectively. This is because GDH+-GC employs an an in-place GC strategy and the GPGPU-accelerated sorting of Memtables, which reduces CPU overhead and enhances the performance of the KV store. Under workload C, GDH+ and GDH+-GC outperform RocksDB, gLSM, GPU-Comp GDS, ALDC, WiscKey, WiscKey-GC, DiffKV, GDH, and GDH-GC because an LRU cache is used to assess the temperature of keys and store hot keys in PM, boosting the read performance.

GDH+ and GDH+-GC demonstrate superiority over RocksDB, gLSM, GPU-Comp GDS, ALDC, WiscKey-GC, DiffKV-GC, and GDH-GC under mixed read-write workloads A and F. This is because GDH+ combines GPGPU and PM to enhance performance. Under workload E, GDH+ and GDH+-GC provide higher range query performance compared to most alternative KV stores.

5.3 Extended Experiments

5.3.1 GPGPU Utilization and Memory Usage. We evaluated the average and maximum GPGPU utilization and memory usage of gLSM, GDH+ and GPU-Comp GDS, see Figure 13. These metrics were measured every 0.5 seconds. The average and maximum GPGPU memory usage were recorded across the program's entire runtime, while GPGPU utilization was assessed during periods of active processing (*i.e.,* utilization exceeded 0). GDH+ achieves higher average and maximum GPGPU utilization during active processing than gLSM and GPU-Comp GDS. This is because GDH+ makes more efficient use of GPGPU's parallel computing resources, enhancing the efficiency of sorting and flushing. Furthermore, GDH+ exhibits lower average and maximum GPGPU memory usage. Thus, transferring data to GPGPU memory in GDH+ is primarily effective to improve computation performance, reducing redundant data.

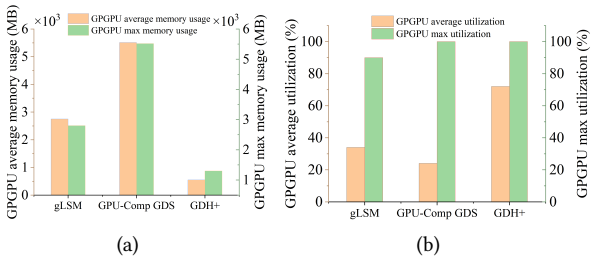


Fig. 13. GPGPU memory usage (a) and GPGPU utilization (b).

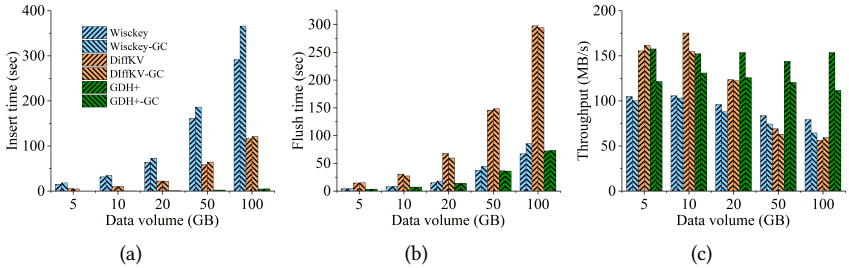


Fig. 14. Insert time (a), flush time (b), and throughput (c).

Table 2. Transfer Time Ratio

Data Volume (GB)	Transfer Time Ratio (%)		Throughput (MB/s)	
	GDH+	GDH+-GC	GDH+	GDH+-GC
5	5.4	5.5	142.5	123.2
10	6.4	5.7	131.1	115.2
20	7.1	6	134.5	133.3
50	7.4	5.4	154.8	106.8
100	8.1	6.1	167.2	128.3

5.3.2 Transfer Time Ratio. We analyzed the transfer time ratio for GDH+ and GDH+-GC. The transfer time encompasses the time spent transferring SSTables between GPGPU memory and SSD, allocating device memory, and transferring data between CPU memory and GPGPU memory. The transfer time ratio is defined as the ratio of transfer time to total test time for different data sizes, as listed in Table 2. For data sizes ranging from 5 GB to 100 GB, the transfer time ratio remains below 9%, and it does not significantly increase as the data volume grows. It means that data transfer is not a performance bottleneck for GDH+ and GDH+-GC.

5.3.3 Insert and Flush Time. We evaluated the memtable insertion and flush times of WiscKey-GC, DiffKV-GC, and GDH+-GC, see Figure 14. Insert time is the average delay for inserting a KV pair into the memtable from the write initiation of the KV pair to being written into the memtable. GDH+-GC demonstrates lower insertion time, flush time, and higher throughput. This is because GDH+ uses an asynchronous insertion and sorting approach based on a hash table, which is accelerated by GPGPU-based sorting, decreasing the insertion time. Additionally, GDH+ leverages GPGPU's parallel computing capabilities to speed up the flush operation, thereby enhancing flush

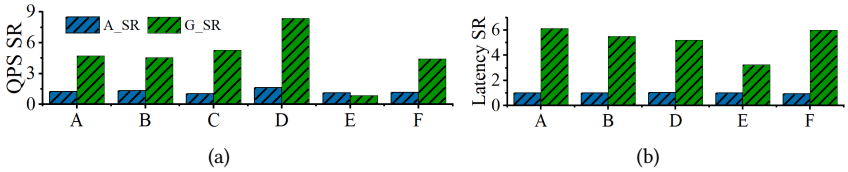


Fig. 15. Performance Comparison Between GDH+ and AC-key in terms of **query per second** (**QPS**) (a) and write latency (b) under YCSB. SR means speedup ratio.

Table 3. Processing Time, Idle Time, and Processing Time Ratio Under Different Data Volumes

Data Volume (GB)	Processing Time (sec)	Idle Time (sec)	Processing Time Ratio (%)
5	3.35	33.60	9.1
10	6.60	75.40	8.0
20	15.56	138.44	10.1
50	35.86	317.14	10.2
100	61.95	580.05	9.6

performance and throughput. In contrast, WiscKey and DiffKV rely on the skiplist structure, where insertion and sorting are performed synchronously, leading to higher insertion time. The flush operations, handled by the CPU, become slower, impairing flush performance and throughput.

5.3.4 Performance Comparison Between GDH+ and AC-Key. AC-Key [37] optimizes read performance via multi-level caching (KV, KP, and Block) with dynamic space adjustment. AC-Key lacks targeted optimizations for large-value writes. In contrast, GDH+ adopts a write-oriented design, GPGPU-assisted sorting, hotness-aware cache migration, and PM acceleration. We applied RocksDB as a baseline and conducted YCSB benchmark tests. We denote AC-Key's and GDH+'s performance ratios over RocksDB as *A_SR* and *G_SR*, respectively. Figure 15(a) presents the QPS improvement ratios under six workloads in YCSB. GDH+ shows advantages over AC-Key, especially under read-heavy workloads. For instance, *G_SR* reaches 426.7% and 738.4% under workloads C and D, validating GDH+'s scalability under high write pressure and large values. Figure 15(b) shows the average write latency reduction over RocksDB. GDH+ optimizes the write latency relative to RocksDB of 611%, 550%, and 600% under workloads A, B, and F, outperforming AC-Key, which offers marginal or even negative gains. These results validate GDH+'s write path design, which incorporates hotness-based SSTable group and cold-hot data separation to reduce write stalls.

5.3.5 GDH+ Performance Under Concurrent GPGPU Workloads. In this section, we validate the GDH+ acceleration mechanism that maintains efficiency when sharing GPGPU resources with other workloads such as ML and **large language models** (**LLMs**). We calculated the processing time, idle time, and processing time ratio of GDH+ under the workloads with various data volumes. These metrics reflect the resource utilization of GDH+. The experimental results are summarized in Table 3. The processing time ratio of GDH+ remains below 10.2% across different data volumes, which indicates that GDH+ has limited utilization of GPGPU computational resources. Specifically, GDH+ retains over 90% of GPGPU idle resources for other workloads, *e.g.,* ML and LLMs. Therefore, when sharing GPGPU resources with other workloads, GDH+ rarely causes minimal performance degradation due to its low computational resource requirements.

5.3.6 Comparison of Speedup Ratios of GDH+ and NDP-based KV Stores. ProckStore—an NDP-based KV store—employs an asynchronous and multi-threaded compaction scheme. We configure

Table 4. Performance Speedup Ratio of GDH+ to NDP-Based KV Stores

Speedup Ratio	GDH+ LevelDB		GDH+ RocksDB		ProckStore S-RocksDB		HStore-S S-BlobDB_S		HStore-M S-BlobDB_M	
	KTPS	AVG	KTPS	AVG	KTPS	AVG	KTPS	AVG	KTPS	AVG
Load	21.02	21.48	32.76	33.40	3.44	3.44	7.15	7.45	6.25	6.78
A	4.39	7.86	11.32	13.70	1.10	1.10	3.31	3.31	1.58	1.57
B	4.93	8.76	10.54	14.02	1.08	1.08	1.86	1.86	1.68	1.67
C	4.59	9.73	16.09	18.74	1.21	1.21	4.24	4.23	2.75	2.74
D	5.85	12.39	15.76	21.87	1.07	1.07	2.79	2.79	2.10	2.09
E	0.72	1.22	9.35	3.12	1.01	1.01	2.63	2.63	0.04	0.04
F	4.02	8.60	9.69	15.49	1.07	1.07	3.15	3.14	1.45	1.45
	P90	P99	P90	P99	P90	P99	P90	P99	P90	P99
Load	109.30	50.92	15.80	149.29	2.92	1.39	0.33	11.23	0.35	145.40
A	5.09	21.93	11.36	45.56	1.13	1.60	3.08	31.26	2.39	3.08
B	7.29	31.79	15.29	65.71	1.09	1.08	3.76	4.12	3.33	4.48
C	6.08	36.58	18.62	76.18	1.11	1.10	6.33	6.54	4.24	5.60
D	8.46	41.62	18.00	86.41	1.12	1.06	3.89	4.65	3.19	5.55
E	0.74	1.78	3.54	6.76	1.07	1.07	2.78	2.80	0.04	0.05
F	5.92	18.83	14.62	39.23	1.08	1.25	4.35	17.41	2.99	2.52

First, the high speedup ratios (33.4× average, 39.23× P99) means that GPGPU-accelerated KV stores significantly outperform NDP-based ones under mixed read-write (A) and write-intensive (F) workloads. Second, GDH+ achieves these benefits by fully leveraging GPGPU parallelism for sorting/flushing, whereas NDP architecture suffers from limited computational resources. The performance gap becomes large in terms of tail latency (P99), where GDH+ maintains low latency while NDP-based KV stores show higher variability. Third, the highlighted entries validate that heterogeneous GPGPU acceleration is more effective than NDP-based key-value separation when WA and CPU overhead are critical.

a key-value separation in the NDP-based KV store, named HStore in this section. HStore_S and HStore_M denote that HStore uses an NDP device and multiple NDP devices, respectively. In addition, we modified RocksDB and BlobDB on the NDP architecture, denoted as S-RocksDB and S-BlobDB. Notably, S-BlobDB_M means S-BlobDB adopts a RAID architecture for scalability. While the NDP architecture slashes data transfer overhead, NDP does not well address the problem of insufficient computational resources. In contrast, GDH+ fully leverages GPGPU resources, and its performance gains outweigh that of the NDP architecture.

As listed in Table 4, GDH+ utilizes GPGPU to achieve significantly higher speedup ratios compared to ProckStore and HStore. For example, under workload A, the average speedup ratio of $\frac{GDH+}{RocksDB}$ reaches 33.40, while ProckStore/S-RocksDB achieves only 3.44. HStore_S/S-BlobDB_S achieves 7.45, and $\frac{HStore_M}{S-BlobDB_M}$ achieves merely 6.78. Under workload F, the P99 speedup ratio of $\frac{GDH+}{RocksDB}$ reaches 39.23, whereas ProckStore/S-RocksDB is only 1.25. $\frac{HStore_S}{S-BlobDB_S}$ is 17.41, and $\frac{HStore_M}{S-BlobDB_M}$ is 2.52. These results indicate that GDH+ outperforms NDP-based KV stores.

Although the NDP architecture effectively reduces data transfer overhead, NDP's limited computational resources constrain overall performance improvement. For instance, when HStore scales from an NDP device (HStore_S) to multiple NDP devices (HStore_M), the average speedup ratio under workload A decreases slightly from 7.45 to 6.78. In contrast, GDH+ fully leverages the high parallelism and computational capability of GPGPU. GDH+ maintains superior throughput and low tail latency across all YCSB workloads. This finding further confirms the advantages of GPGPU-accelerated heterogeneous architectures for KV stores.

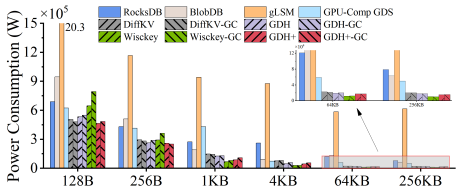


Fig. 16. Comparison of power consumption between GDH+ and other KV stores.

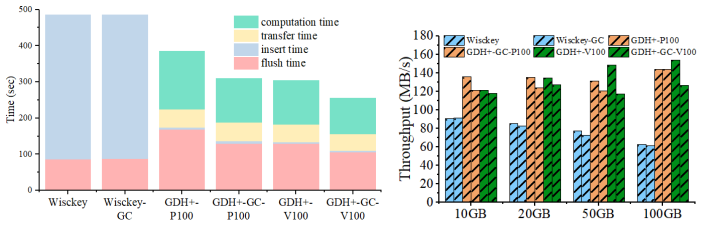


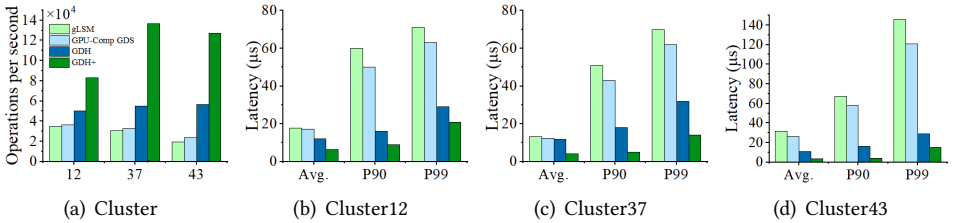
Fig. 17. Impact of GPGPU computational capability on (a) time breakdown and (b) throughput compared to a CPU-based baseline.

Table 5. Comparison of Power Consumption (10^4 W) Between GDH+ and NDP-Based KV Stores

value size	HStore_S		HStore_M		GDH+	GDH+-GC
	Host	Device	Host	Device		
4 KB	500.5268	1.7011	33.5231	0.5071	4.4087	5.4595
64 KB	73.6549	0.1929	19.0141	0.2209	1.7190	1.7199
256 KB	74.2749	0.1993	24.6071	0.1895	1.5364	1.5643

5.3.7 Power Consumption. GDH+ does not increase power consumption with GPGPU acceleration. We studied the CPU and GPGPU power consumption of KV stores. We used the sysstat tool to record CPU power consumption and the nvidia-smi tool to obtain GPGPU power consumption. As shown in Figure 16. GDH+ demonstrates low power consumption across all data sizes. For example, when the value size is 1 KB, the power consumption of GDH+-GC is reduced by 60.34%, 88.51%, 74.9%, 22.41%, and 16.25% compared to RocksDB, gLSM, GPU-Comp GDS, DiffKV-GC, and GDH-GC. We also provided a comparison with KV stores based on the NDP architecture. As listed in Table 5, the average power consumption of GDH+-GC is reduced by 98.16% and 89.57% compared with HStore_S and HStore_M, respectively. This is because the hash table-based asynchronous in GDH+ inserts KV pairs into a MemTable, that is, flushed into an SSTable on the SSD; meanwhile, GPGPU speeds up flushing, thereby lowering computational overhead and power consumption.

5.3.8 Effect of GPGPU Computational Capability. We studied the performance of GDH+ on two GPGPU architectures (P100 and higher-computational-capability NVIDIA Tesla V100) to assess GPGPU acceleration benefits. As shown in Figure 17(a), GPGPUs significantly reduce total runtime, particularly in the memtable insert stage where time is reduced from 397.9 s in CPU-based WiscKey to 4–6 s, improving by up to 98.6%. The approach in GDH+ to offloading sorting and encoding during flush to the GPGPU enhances parallelism, while pipelining CPU inserts with GPGPU flush overlaps computation and data transfer, avoiding performance bottlenecks. On the P100, transfer



(a) Cluster (b) Cluster12 (c) Cluster37 (d) Cluster43

Fig. 18. Throughput and latency comparison under X workloads. P90 and P99 denotes the 90th, and 99th percentile latency, respectively.

time accounts for only 12.9%–16.8% of total time and remains bounded relative to computation. This indicates that data movement overhead does not outweigh GPGPU benefits. Figure 17(b) shows throughput improvements across data sizes. At 100 GB, GDH+ achieves 144.3 MB/s (P100) and 154.2 MB/s (V100), which are 2.30× and 2.46× higher than WiscKey's 62.6 MB/s, respectively. On the V100, computation time decreases by 24.7% compared to P100, leading to a 21% shorter end-to-end runtime. These results demonstrate that P100-based GDH+ delivers performance gains over CPU-based WiscKey, with higher-computational-capability GPGPU (V100) yielding further gains.

5.3.9 Performance of GDH+ Under Realistic Workloads. We studied the performance of GDH+ under realistic X cache cluster traces: Cluster12, Cluster37, and Cluster43. Each trace has a 50 GB dataset. The write-intensive Cluster12 (80% set, 20% get) has an average 44 B key and 1,017 B value, totaling 50 million operations. Cluster37 is a read-intensive workload (37% set, 63% get) with an average key/value size of 72 B/15,001 B and 3,571,180 operations. Cluster43 is a mixed workload (50% set, 50% get) with an average 44-B key, 2,128-B value, and 25 million operations. As shown in Figure 18(a), GDH+ achieves the highest throughput across all workloads. Under Cluster12, GDH+ improves throughput by 2.38× over GDH and 4.62× over gLSM. For Cluster37 and Cluster43, the throughput gains are 2.48×/2.25× over GDH and 4.17×/6.46× over gLSM, respectively. As shown in Figure 18(b), (c) and (d), GDH+ still achieves the lowest average, P90, and P99 latencies. Compared with GDH, GDH+ reduces average latency by 64%, 69%, and 66% under Clusters 12, 37, and 43, respectively, with even larger reductions relative to gLSM. These results confirm that GDH+'s advantages in terms of throughput and tail latency under realistic traces. These performance benefits stem from its key-value separation, hot-cold data placement, and GPGPU-accelerated flush.

5.3.10 End-to-end Time Breakdown. GDH+'s end-to-end time breaks down as follows according to our experimental results, see Table 6. CPU compute (148.59), GPU compute (109.53s), host↔device copies (25.72s), device I/O (114.48s), and stalls (1.32s). While GDH+ incurs higher CPU compute compared to gLSM (0.42) and GPU-Comp (0), this is a deliberate design trade-off: the CPU efficiently manages the hot-cold data tiering and PM operations, drastically reducing stalls (1.32s vs. 2301.52s in GPU-Comp) and host-device data movement (25.72s vs. 951.76s in GPU-Comp). The GPU is effectively utilized for parallel sorting and encoding tasks, keeping its computing time on par with gLSM. This division of labor allows GDH+ to achieve significantly lower total execution time by curbing I/O and synchronization overhead.

5.3.11 Co-location with LLM Workloads. To study GDH+'s impact on LLM performance, we co-ran GDH+ with the MiniCheck LLM under varying value sizes. As shown in Figure 19, GDH+ slightly affects LLM throughput with small values (256 B–1 KB) due to GPGPU resource contention during sorting and flushing. This overhead remains bounded and does not significantly degrade

Table 6. End-to-End Time Breakdown (Seconds) for the 1 KB Value-Size Sweep

System	Stalls	Device I/O	CPU	Host↔Device	GPU
gLSM	41.08	222.26	0.42	90.88	115.79
GPU-Comp	2,301.52	375.24	0	951.76	1,218.73
GDH+	1.32	114.48	148.59	25.72	109.53

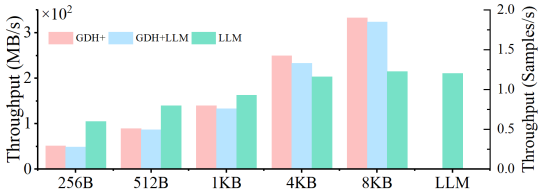


Fig. 19. Throughput of GDH+ co-running with MiniCheck LLM.

LLM performance. For large-sized values, GDH+ exhibits negligible interference. Since GDH+ only uses the GPGPU during memtable flushes—even during LLM inference—it introduces minimal disruption to co-located GPGPU workloads.

6 Conclusion

In this article, we proposed GDH+ to address the performance and CPU utilization challenges in key-value separation strategies. GDH+ introduces a hash-table-based asynchronous insertion and sorting approach that leverages the parallelism of GPGPU to accelerate both sorting and flush operations, thereby enhancing write performance. We designed a memory component using an LRU cache that places keys in either PM or SSD based on the access frequency. We implemented an adaptive migration strategy for SSTables that optimizes read performance by managing hot and cold data. Furthermore, we devised an in-place GC strategy to minimize CPU utilization while maintaining comparable performance, which addresses the limitations of existing key-value separation. GDH+ achieves a 2× improvement in write performance, a 1.5× improvement in read performance, and reduces CPU utilization by 40% compared to state-of-the-art KV stores.

References

[1] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: A database benchmark based on the facebook social graph. In *Proc. the 2013 ACM SIGMOD Int. Conf. on Management of Data*. 1185–1196.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proc. the 12th ACM SIGMETRICS/PERFORMANCE joint Int. Conf. on Measurement and Modeling of Computer Systems*. 53–64.

[3] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *Proc. the 18th USENIX Conf. on File and Storage Technologies (FAST 20)*. 209–223.

[4] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. 2019. LDC: A lower-level driven compaction method to optimize SSD-oriented key-value stores. In *Proc. the 2019 IEEE 35th Int. Conf. on Data Engineering (ICDE)*. 722–733.

[5] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, and Yangyang Wang. 2022. Adaptive lower-level driven compaction to optimize LSM-tree key-value stores. *IEEE Tran. Knowledge and Data Engineering* 34, 6 (2022), 2595–2609.

[6] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling efficient updates in KV storage via hashing. In *Proc. the USENIX Annual Technical Conf.* 1007–1019.

[7] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. {SpanDB}: A fast,{cost-effective}{LSM-tree} based {KV} store on hybrid storage. In *Proc. the 19th USENIX Conf. on File and Storage Technologies (FAST 21)*. 17–32.

[8] Lixiang Chen, Ruihao Chen, Chengcheng Yang, Yuxing Han, Rong Zhang, Xuan Zhou, Peiquan Jin, and Weining Qian. 2023. Workload-aware log-structured merge key-value store for NVM-SSD hybrid storage. In *Proc. the 2023 IEEE 39th Int. Conf. on Data Engineering (ICDE)*. 2207–2219.

[9] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proc. the 25th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.

[10] Lixiao Cui, Kewen He, Yusen Li, Peng Li, Jiachen Zhang, Gang Wang, and Xiaoguang Liu. 2023. SwapKV: A hotness aware in-memory key-value store for hybrid memory systems. *IEEE Trans. Knowl. Data Eng.* 35, 1 (2023), 917–930.

[11] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From wisckey to bourbon: A learned index for log-structured merge trees. In *Proc. the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

[12] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2023. Revisiting log-structured merging for kv stores in hybrid memory systems. In *Proc. the 28th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems* 2 (2023), 674–687.

[13] Facebook. 2024. Rocksdb. Retrieved from http://rocksdb.org/

[14] Google. 2017. LevelDB. Retrieved from https://github.com/google/leveldb

[15] Kewen He, Yujie An, Yijing Luo, Xiaoguang Liu, and Gang Wang. 2023. FlatLSM: Write-optimized LSM-tree for PM-based KV stores. *ACM Trans. Storage* 19, 2 (2023), 1–26.

[16] PingCAP Inc. 2024. TiDB Documentation. Retrieved from https://download.pingcap.org/tidb-stable-en-manual.pdf

[17] R. Madhava Krishnan, Naga Sanjana Bikonda, Shashwat Jain, Wook-Hee Kim, Hamid Hadian, Vishwanath Maram, and Changwoo Min. 2023. Retina: Cross-layered key-value store for computational storage. In *Proc. the 2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 1–8.

[18] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: The design and implementation of a fast persistent key-value store. In *Proc. the 27th ACM Symposium on Operating Systems Principles*. 447–461.

[19] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated key-value storage management for balanced I/O performance. In *Proc. the 2021 USENIX Annual Technical Conf.* 673–687.

[20] Bihui Liu, Zhenyu Ye, Qiao Hu, Yupeng Hu, Yuchong Hu, Yang Xu, and Keqin Li. 2024. HPDK: A hybrid PM-DRAM key-value store for high I/O throughput. *IEEE Trans. Comput.* 73, 6 (2024), 1575–1587.

[21] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage* 13, 1 (2017), 1–28.

[22] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. the Symp. Operating Systems Principles*. 497–514.

[23] Jun Rao and Kenneth A. Ross. 2000. Making B+- trees cache conscious in main memory. In *Proc. the 2000 ACM SIGMOD Int. Conf. on Management of Data*. New York, NY, USA, 475–486.

[24] Adam Wright. 2024. Worldwide global datasphere and global storagesphere structured and unstructured data forecast. 2024–2028. Retrieved from https://techcontentwave.com/files/1763389937_64b594fd4f7f5607256a.pdf

[25] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing key-value store for modern heterogeneous storage devices. In *Proc. the 28th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems* 2 (2023), 588–602.

[26] Hui Sun, Wei Liu, Jianzhong Huang, and Weisong Shi. 2019. Collaborative compaction optimization system using near-data processing for LSM-tree-based key-value stores. *J. Parallel and Distrib. Comput.* 131, C (2019), 29–43.

[27] Hui Sun, Bendong Lou, Chao Zhao, Deyan Kong, Chaowei Zhang, Jianzhong Huang, Yinliang Yue, and Xiao Qin. 2024. Asynchronous compaction acceleration scheme for near-data processing-enabled LSM-tree-based KV stores. *ACM Trans. Embed. Comput. Syst.* 23, 6 (2024), 1–33.

[28] Hui Sun, Qiang Wang, Yin Liang Yue, Yuhong Zhao, and Song Fu. 2022. A storage computing architecture with multiple NDP devices for accelerating compaction performance in LSM-tree based KV stores. *Journal of Systems Architecture* 130, C (2022), 102681.

[29] Hui Sun, Jinfeng Xu, Xiangxiang Jiang, Guanzhong Chen, Yinliang Yue, and Xiao Qin. 2024. gLSM: Using GPGPU to accelerate compactions in LSM-tree-based key-value stores. *ACM Trans. Storage* 20, 1 (2024), 1–41.

[30] Hui Sun, Chao Zhao, Yinliang Yue, and Xiao Qin. 2025. ProckStore: An NDP-empowered key-value store with asynchronous and multi-threaded compaction scheme for optimized performance. *Journal of Systems Architecture* 160, C (2025), 103342.

[31] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. FPGA-based compaction engine for accelerating LSM-tree key-value stores. In *Proc. the IEEE Int. Conf. Data Engineering*. 1261–1272.

[32] Mark Sutherland, Babak Falsafi, and Alexandros Daglis. 2022. Cooperative concurrency control for write-intensive key-value workloads. In *Proc. the 28th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems* 1 (2022), 30–46.

[33] Chenlei Tang, Jiguang Wan, and Changsheng Xie. 2022. Fencekv: Enabling efficient range query for key-value separation. *IEEE Trans. Parallel Distrib. Syst.* 33, 12 (2022), 3375–3386.

[34] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-data processing in database systems on native computational storage under htap workloads. *Proc. the VLDB Endowment* 15, 10 (2022), 1991–2004.

[35] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An efficient compaction approach for log-structured key-value store on persistent memory. In *Proc. the 2022 USENIX Annual Technical Conf. (USENIX ATC 22)*. 773–788.

[36] Zhiqi Wang and Zili Shao. 2023. MirrorKV: An efficient key-value store on hybrid cloud storage with balanced performance of compaction and querying. *Proc. the ACM on Management of Data* 1, 4 (2023), 1–27.

[37] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H. C. Du. 2020. AC-Key: Adaptive caching for LSM-based key-value stores. In *Proc. the 2020 USENIX Annual Technical Conf. (USENIX ATC 20)*. 603–615.

[38] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid key-value placement in LSM-based key-value stores. In *Proc. the ACM Symposium on Cloud Computing*. 305–318.

[39] Peng Xu, Jiguang Wan, Ping Huang, Xiaogang Yang, Chenlei Tang, Fei Wu and Changsheng Xie. 2020. LUDA: Boost LSM key value store compactions with GPUs. Retrieved from https://arxiv.org/pdf/2004.03054

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

[41] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *Proc. the USENIX Annual Technical Conf.* 17–31.

[42] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward high-performance and scalable KV storage in mixed workloads via unified indexing. In *Proc. the 2020 IEEE 36th Int. Conf. on Data Engineering (ICDE)*. 313–324.

[43] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-accelerated compactions for LSM-based key-value store. In *Proc. the USENIX Conf. File and Storage Technologies*. 225–237.

[44] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. 2014. Pipelined compaction for the LSM-tree. In *Proc. the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 777–786.

[45] Hao Zhou, Yuanhui Chen, Lixiao Cui, Gang Wang, and Xiaoguang Liu. 2024. A GPU-accelerated compaction strategy for LSM-based key-value store system. In *Proc. the 38th Int. Conf. on Massive Storage Systems and Technology*.

Received 25 January 2025; revised 30 November 2025; accepted 2 February 2026