

gParaKV: A GPGPU-accelerated Key-Value Separation-based KV Store with Optimized Compaction and Garbage Collection

Hui Sun
Anhui University
Hefei, China
sunhui@ahu.edu.cn

Xiangxiang Jiang
Ahhui University
Hefei, China
jxxahu@gmail.com

Xiao Qin
Auburn University
Auburn, USA
xqin@auburn.edu

Song Jiang
University of Texas, Arlington
Arlington, USA
sjiang88@gmail.com

Enhui Wang
Anhui University
Hefei, China
o375a96354@163.com

Abstract

LSM-tree-based key-value stores or KV stores are widely deployed in modern cloud storage systems thanks to high data storage efficiency and retrieval capabilities. The compaction process in the LSM-tree, however, results in severe performance bottlenecks, especially in scenarios involving large volumes of data. While key-value separation methods mitigate the performance bottlenecks caused by compaction, the existing methods do not fully address merge-sorting during compaction and expensive garbage collection (GC). We propose **gParaKV**, a GPGPU-empowered KV store with a KV separation mechanism, leveraging the GPGPU parallel technology to accelerate merge-sorting in compaction and GC. gParaKV embraces unique features like a GPGPU bitmap structure, parallel data marking, and a parallel GC mechanism. These critical components effectively curtail the overhead of merge-sorting and GC operations by virtue of parallel computing. We compare it with state-of-the-art KV stores (e.g., RocksDB, BlobDB, Wisckey, DiffKV, UniKV, and HPDK) under various workloads. The experimental results show that gParaKV can improve the write performance and GC efficiency compared to the existing key-value separation-based KV stores.

CCS Concepts

• **Computer systems organization** → *Parallel architectures*; • **Information systems** → *Information storage technologies*; *Storage management*.

Keywords

Key-Value Store, Compaction, GPGPU, Garbage Collection

ACM Reference Format:

Hui Sun, Xiangxiang Jiang, Xiao Qin, Song Jiang, and Enhui Wang. 2025. gParaKV: A GPGPU-accelerated Key-Value Separation-based KV Store with Optimized Compaction and Garbage Collection. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25, St Louis, MO, USA)*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC '25, St Louis, MO, USA

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1466-5/25/11
<https://doi.org/10.1145/3712285.3759896>

'25), November 16–21, 2025, St Louis, MO, USA. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3712285.3759896>

1 Introduction

Log-structured merge-tree (LSM-tree)-based key-value stores (KV stores) have become prevalent in modern cloud storage systems for wide applications [4, 16, 22]. Due to high performance, KV stores are suitable for processing large-scale unstructured data [20, 21, 26, 41]. In addition, KV stores can meet the requirements of low latency and high throughput in a wide range of applications [8, 15, 39]. While read operations account for the majority of workloads in these applications [1], the distribution of read and write operations within workloads tends to become more balanced in applications such as cloud computing, online transaction processing systems (OLTP) [3], and mobile software platforms [7, 17, 20, 28] that demand high write performance. Furthermore, the production environments increasingly exhibit workloads with a greater proportion of writes compared to reads. For instance, Twitter's production workloads show that over 30% involve more writes than reads [33, 38].

Prior studies highlight the importance of optimizing write performance in KV stores [7, 17]. LevelDB [12] and RocksDB [4] improve write performance by converting random writes into sequential ones, while enhancing read performance through caching frequently accessed blocks, reducing I/Os. The LSM-tree maintains high performance by managing duplicate data through compaction, ensuring that KV pairs in a sorted string table (SSTable) at each level are ordered and unique, which improves read performance [30]. However, compaction introduces high CPU and I/O costs due to merge-sorting and write amplification (WA), which increases the number of writes required beyond applications [7, 21].

The heterogeneous devices that accelerate compaction [30, 32, 42] reduce CPU overhead but still suffer from significant WA. Key-value separation methods, such as Wisckey [21], HashKV [7], DiffKV [19], FenceKV [34], and UniKV [41], reduce WA by minimizing data volume during compaction. However, existing methods inadequately optimize the compute-intensive merge-sorting process that involves decoding, sorting, and encoding each KV pair, leading to sequential CPU execution and limited utilization of multi-core hardware resources, restricting compaction performance improvements. Key-value separation leverages log files to store values and reclaims invalid data during compaction. Garbage collection (GC) is periodically triggered to free disk space. Existing GC strategies are

either executed by a background thread or integrated into the compaction process. In the former, a background thread sequentially scans log files, checks KV pair validity, and rewrites valid KV pairs to the LSM-tree. This sequential process hinders the parallelization of scanning and validity checks, especially during large-scale data processing, slowing down GC and degrading performance. In the latter, GC is embedded in compaction, increasing computational overhead and reducing system throughput.

To address the above issues, we propose a GPGPU-accelerated solution to optimize compaction and GC in key-value separation-based KV stores. However, we face the following challenges:

- **Challenge 1:** How to design an efficient GPGPU parallel algorithm to execute the compute-intensive merge-sorting in compaction?

- **Challenge 2:** How to implement an efficient GPGPU parallel GC strategy and fast mark valid or invalid KV pairs during GC?

To address the above challenges, we propose gParaKV, a KV store leveraging heterogeneous GPGPU parallel processing to enhance traditional key-value separation. To address **Challenge 1**, gParaKV uses GPGPU to accelerate merge-sorting during compaction, including parallel decoding, sorting, marking, uniqueness filtering, and encoding. We leverage the following techniques to tackle **Challenge 2**. Notably, during GC, the marking method identifies invalid keys. To optimize the marking of invalid KV pairs, we design a parallel marking technique in gParaKV, and introduce a GPGPU bitmap structure that sequentially records the validity of all KV pairs in a log file, enabling quick parallel validity checks through index computations. gParaKV also tracks the number of invalid KV pairs per log file in GPGPU memory, facilitating rapid determination of whether log files meet the GC threshold. Additionally, a dynamic GC threshold based on the number of KV pairs per log file ensures efficient GC operations. Furthermore, gParaKV accelerates the GC process via GPGPU, employing a dynamic thread allocation strategy that allocates GPGPU threads to handle log files based on the number of contained KV pairs, thereby balancing thread allocation overhead and GC parallelism. For valid KV pairs, gParaKV forms batches with KV pairs from foreground threads and writes the KV pairs back to the LSM-tree in bulk, preventing foreground threads from being blocked.

Our contributions are listed as follows:

- We design a GPGPU bitmap to facilitate the parallel marking of invalid data: each bit represents the validity of a KV pair. All KV pairs across log files are stored sequentially, and their validity can be quickly determined in parallel using index computation. This mechanism accelerates the marking of invalid data and the querying of KV pair validity.
- We leverage GPGPU to speed up the merge-sorting operation accompanied by a parallel data marking strategy powered by the GPGPU bitmap. Additionally, we record the number of invalid KV pairs for each log file to facilitate delayed GC.
- We propose a GPGPU-empowered parallel GC strategy, which enables parallel GC triggering. Both the GC threshold and parallelism degree for handling GC can be dynamically adjusted based on the number of log files, achieving low resource consumption and high GC efficiency.
- We implement gParaKV on top of LevelDB, leveraging parallel computing to optimize performance. gParaKV adopts a key-value

separation to mitigate WA and harnesses the GPGPU to enable parallel GC and merge-sorting during compaction. Experimental results demonstrate that gParaKV significantly outperforms the state-of-the-art KV stores.

The remainder of this paper is organized as follows. Section 2 presents the background and motivation of this study. In Section 3, we articulate the design of gParaKV. After describing the experimental platform in Section 4, we evaluate in Section 5 the devised system. Section 6 elaborates on the related studies. Finally, we conclude the paper in Section 7.

2 Background and Motivation

2.1 LSM-tree and compaction

In KV stores, LSM-tree leverages sequential data written to memory and periodically transfers data to the disk in batches, enhancing write performance and minimizing the write overhead. In Fig. 1a, KV pairs are initially written to the Memtable, which is converted into an Immutable Memtable – awaiting to be flushed to the disk as an SSTable – if the Memtable is full. The *Get* operation first queries the Memtable and Immutable Memtable; if no match is found, the operator searches the LSM-tree on the solid-state drive (SSD).

In the LSM-tree, compaction is triggered when the data volume in level 0 exceeds a predefined threshold or when SSTables with duplicate KV pairs exist between adjacent levels. The compaction process involves SSTable read, merge-sorting, and write. During the read, data blocks from selected SSTables are loaded into memory, and iterators are constructed. In the merge-sorting phase, blocks are decoded to obtain KV pairs. Duplicate KV pairs are merged by comparing their sequence numbers (sequences of keys). The KV pair with the larger sequence number is written into a new block, while non-duplicate KV pairs are added. The KV pairs are then encoded into new blocks. Finally, in the write phase, the new blocks are written to a new SSTable. The three steps are repeated until all blocks are processed.

The merge-sorting process during compaction involves decoding, merging, and encoding each KV pair, which is a compute-intensive task requiring significant CPU resources. If CPU resources are limited, merge sorting can potentially impede other operations (*e.g.*, put), increasing latency. Fig. 1b shows the LSM-tree optimized with heterogeneous computing devices. For example, FPGAs and GPGPUs alleviate CPU resource-limited performance bottlenecks by parallel decoding, sorting, and encoding in the merge-sorting process. However, this solution struggles to lessen the performance overhead caused by moving large values during decoding and encoding, and high WA amid the read/write of SSTables.

The key-value separation in the LSM-tree can address the issues of movement overhead and WA caused by large values. Fig. 1c unveils that the key-value separation stores keys and value pointers in the LSM-tree, whereas values are stored in log or blob files. Log or blob files contain keys, values, and sequence numbers, facilitating rapid recovery in the case of a system crash. The value pointer records the value size, file number, and offset of a log or blob file. The key-value separation strategy reduces the size of KV pairs processed during the read/write phases, eliminating the overhead of

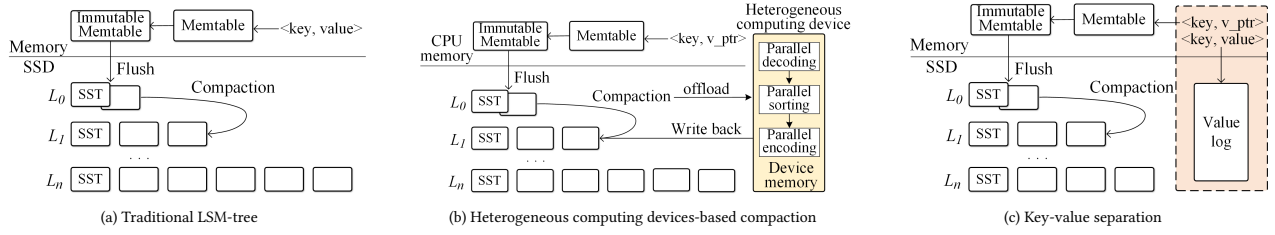


Figure 1: Traditional LSM-tree structure (a), compaction optimized with heterogeneous computing devices (b), and key-value separation (c).

processing large values during the merge-sorting process. This reduces I/O overhead and WA during compaction, thereby improving compaction performance.

The key-value separation KV stores keys and value pointers to the LSM-tree – and the values are stored in log or blob files. During compaction, invalid keys and value pointers are removed, but clearing invalid KV pairs in log requires a garbage collection (GC). For example, Wisckey’s GC maintains a hash table during compaction to record the number of invalid KV pairs in each log file. When the number of invalid KV pairs in a log file reaches the GC threshold, the GC process is triggered in the background. The GC process scans and checks the validity of each KV pair using the Get operation in the log file, writing valid KV pairs back to the LSM-tree. DiffKV’s GC also maintains a hash table for compaction to track invalid KV pairs in each log file. A log file that exceeds the GC threshold is marked as a GC candidate, and its collection is implicitly completed during the next compaction. BlobDB’s [10] GC iterates through each key during compaction, filtering out expired KV pairs (*i.e.*, keys with smaller sequence numbers) for duplicate keys and writing the new ones into a new log file, reclaiming the old file.

2.2 Inefficient merge-sorting in key-value separation KV Stores

Existing key-value separation strategies overlook the optimization of decoding, merging, and encoding of KV pairs during the merge-sorting process. These strategies traverse each block in SSTables using an iterator and perform decoding, merging, and encoding on each KV pair in the block, imposing a high burden on the CPU. Given larger datasets and a higher number of KV pairs, this CPU overhead inevitably slows down the compaction process, blocks foreground *Put* operations, and downgrades system performance.

To illustrate the CPU pressure imposed by the merge-sorting process, we conduct experiments on various key-value separation-based KV stores under different value sizes and data volumes, measuring the ratio of computation time to read and write time, see Fig. 2. Computation time refers to the time spent in the merge-sorting process, while Read + Write time corresponds to the time consumed in the read and write phases. In Fig. 2a, GPGPU represents the ratio of computation time to read and write time and the throughput, during merge-sorting with a 16-B value. Compared to Wisckey at 16 B, the parallel processing significantly reduces computation time and improves throughput. When it comes to Wisckey, DiffKV, UnIKV, and BlobDB, the computation time ratio is greater than the read and write time across all the value sizes. As the value size decreases and the number of KV pairs increases, this

ratio intensifies, leading to a sustained decline in throughput. Therefore, gParaKV reduces computation time during compaction while improving throughput in the case of 16 B value, demonstrating a load balancing between CPU and GPGPU.

Fig. 3 shows that the throughput of BlobDB with CPU overheads initially increases with the number of compaction threads, and declines. This is because the merge-sorting operation is a serial computing task, which is difficult to be parallelized. Thus, BlobDB underutilizes multi-core CPU resources, limiting performance gains. When the number of compaction threads exceeds the maximum number of CPU threads (40 threads, see Section 4), thread contention arises, thereby declining the performance of BlobDB. In contrast, gParaKV is designed with an efficient GPGPU parallel algorithm to execute the merge-sorting process by utilizing the GPGPU parallel computing resources, thereby overcoming the limitations of CPU resources and improving the performance of CPU-based KV stores.

2.3 Inefficient and compute-intensive GC

For large-scale data volumes, the existing background GC methods – a compute-intensive task – require extensive data scanning to identify valid and invalid data. This process involves significant traversal, filtering, and conditional checks. As shown in Fig. 4a, the computation time during GC includes data scanning and identifying valid or invalid data, and the read and write time includes reading KV pairs into memory and writing valid data back to the LSM tree. The former far exceeds the latter because GC is a sequential process, and the scanning and identification of valid data are difficult to parallelize; thus, multi-core CPUs are under-utilized. Sequential traversal of the large number of KV pairs prolongs GC execution time, degrading system performance. Furthermore, as implicit GC is embedded into the compaction process, GC is performed after a compute-intensive merge-sorting task. Fig. 4b reveals that this GC process extends the computational time of the compaction process, leading to high compaction time and low throughput in KV stores.

To alleviate heavy CPU load and performance bottlenecks in the merge-sorting and GC processes of the existing key-value separation KV stores, we leverage a GPGPU to accelerate the merge-sorting process. By employing parallel decoding, marking, merging, and encoding, we significantly expedite the merge-sorting process and relieve the stress on the CPU. We also propose a GPGPU-empowered GC strategy to accelerate the GC process and reduce the CPU overhead incurred by GC, enhancing overall system performance.

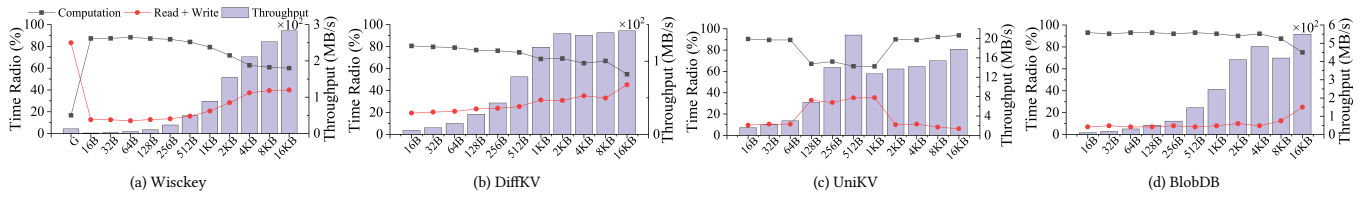


Figure 2: For key-value separation KV stores, the proportion of CPU-based computation time and I/O time during compaction under DB_Bench with a 16 B key, value sizes (from 32 B to 16 KB), and a 10 GB dataset. "G" means GPGPU-accelerated gParaKV with a 16 B value.

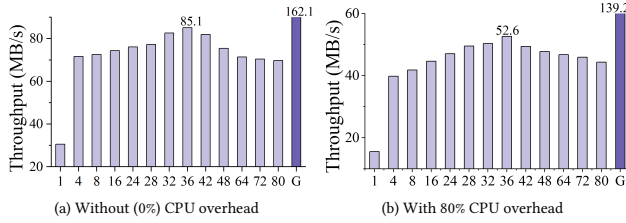


Figure 3: The throughput of BlobDB with various compaction threads. We configure a 1 KB value and a 100 GB dataset in the workload. We employ stress-ng tool [13] to inject stress to each CPU core, where CPU resources consumed by the stress are configured at a preset level (0% 80%). "G" represents GPGPU-accelerated gParaKV.

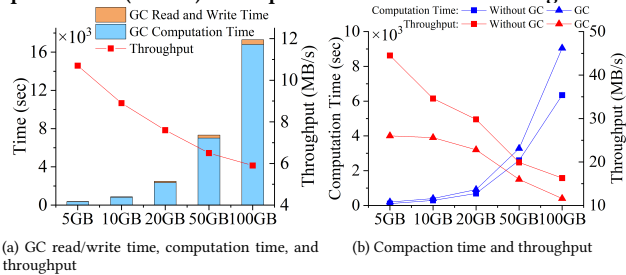


Figure 4: Comparison of read and write time versus computation time in GC and throughput (a), compaction computation time and throughput of BlobDB with GC enabled and disabled (b), under data volumes with a 128 B value.

3 Design

3.1 System Architecture of gParaKV

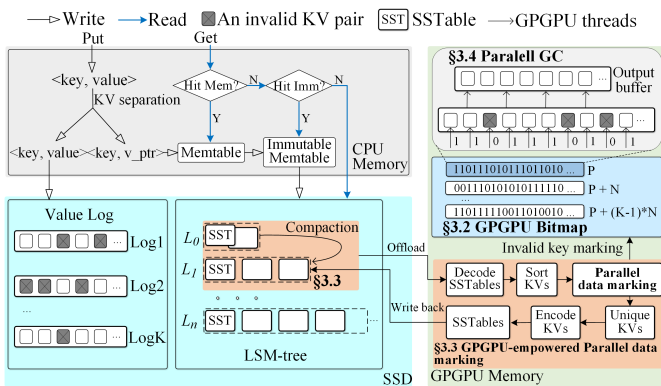


Figure 5: Architecture of gParaKV.

Fig. 5 unravels that gParaKV, a GPGPU-empowered KV store with a key-value separation, can optimize compaction and GC in parallel. gParaKV consists of CPU, GPGPU, and SSD modules. In

gParaKV, key and value pointers are stored in the LSM-tree, and both keys and values are appended to a log file. gParaKV leverages a GPGPU to accelerate merge-sorting and GC processes. The merge-sorting process embraces decoding SSTables, sorting, parallel data marking, removing redundant KV pairs, and encoding (see §3.3). This technique addresses **Challenge 1** in this paper. In addition, during parallel data marking, duplicate keys are identified, and those with smaller sequence numbers are marked invalid. To streamline the efficiency of invalid data marking, a GPGPU bitmap structure is incorporated in the GPGPU memory, where each bit represents the validity of a KV pair. GC is invoked contingent on the amount of invalid KV pairs in the log file. The reclamation process dynamically adjusts the number of GPGPU threads to parallelize GC. These modules can deal with **Challenge 2**. GPGPU threads rely on the GPGPU bitmap to discover the KV pair validity to be written into the output buffer. This design is slated to effectively integrate GPGPU capabilities to optimize merge-sorting and GC processes, enhancing overall system performance. The GPGPU-accelerated bitmap, parallel merge-sorting, and parallel GC are considered as an integrated unit in gParaKV. The absence of any one component can render the system non-running because both merge-sorting and GC depend on the bitmap to identify and mark invalid data.

3.2 The GPGPU Bitmap

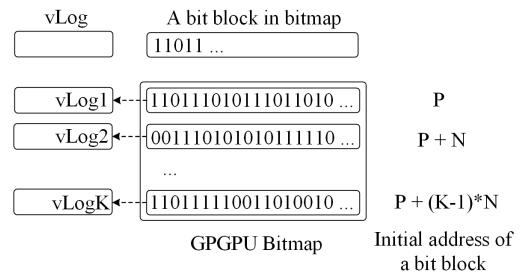


Figure 6: GPGPU bitmap structure in gParaKV.

To maximize GPGPU parallelism in parallel marking invalid data (§3.3) and enabling parallel GC (§3.4), we propose a GPGPU-based bitmap structure (Fig. 6). This structure tracks the validity of KV pairs across all log files, with each pair mapped to a single bit. The bitmap is linearly organized in GPGPU memory, with each bit occupying one byte. '0' indicates invalid and 1 for valid data. If a workload has a 100 GB dataset, a 16 B key, and a 1 KB value, the GPGPU bitmap requires around 100 MB of memory. This design optimizes memory usage while supporting high-speed parallel processing for data validation and GC operations.

To facilitate multi-thread parallel marking of log files in the GPGPU bitmap, we fix the number of KV pairs in each log file

to N . The starting position of a log file (P') in the bitmap can be represented as

$$P' = P + (K - 1) * N, \quad (1)$$

where P is the initial address of the bitmap, K is the log file number (starting from 1), and N is the number of KV pairs in a log file. Each log file occupies N bits in the bitmap, referred to as a block. Next, in the GPGPU bitmap, to set the bits within each block in parallel, we should obtain the position (pos) of each KV pair in the log file. Let kv_{size} and idx denote the fixed size of each KV pair and the KV pair's index in the log file, respectively. Thus, we have

$$pos = idx * kv_{size}, \quad (2)$$

where pos grows linearly in the log file. Accordingly, the bit index of the KV pair in the block is given $idx = \frac{pos}{kv_{size}}$ in the bitmap; therefore, the corresponding position of the KV pair (P_{bit}) in the bitmap is defined as

$$P_{bit} = P + (K - 1) * N + \frac{pos}{kv_{size}}. \quad (3)$$

We map each GPGPU thread to a bit of a block in the GPGPU bitmap, achieving parallel marking of invalid KV pairs. As shown in Fig. 7, each GPGPU thread corresponds simultaneously to a KV pair in the log file and a bit in the bitmap. Let $kv_{size} = 1$, and we have $pos = idx$, which means the position of the KV pair in the log file is equal to the index in the bitmap. Notably, the corresponding bit value of an invalid KV pair in the bitmap is marked as 0. During GC, gParaKV selects the invalid KV pair according to its corresponding bit value in the bitmap.

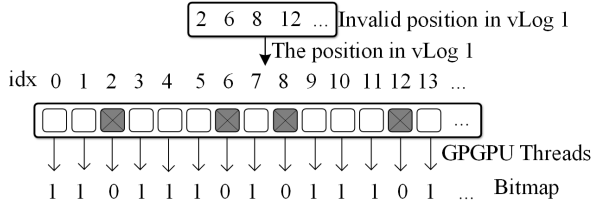


Figure 7: The process of marking invalid bits. The gray rectangle represents an invalid KV pair.

The GPGPU bitmap structure fully utilizes memory bandwidth to enhance access efficiency and optimize GPGPU parallel computation. Through simple index calculations, GPGPU threads can quickly identify the data to be processed, optimizing GPGPU parallel computation, reducing memory fragmentation, and improving memory bandwidth utilization. Furthermore, this structure offers high scalability and adaptability to various data volumes.

To implement delayed GC, we design an array of `uint32_t` to store the number of invalid KV pairs for each log file. Each element represents a log file, and its value is the number of invalid KV pairs. A parallel approach quickly counts invalid KV pairs, and the 32-bit data can accommodate a large number of invalid pairs. This array serves as a criterion for triggering GC. Without transferring the array to CPU memory for sequential scanning, we use GPGPU threads to search the array in parallel and determine if the GC condition is satisfied. When a GPGPU thread meets the condition, other threads exit, and the index of the thread that met the condition is recorded and transferred to CPU memory.

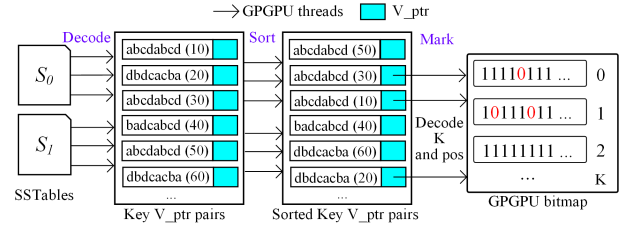


Figure 8: Parallel marking invalid keys. Taking 'abcdabcd(10)' for example, 'abcdabcd' is a key and '(10)' indicates the sequence number of the key.

3.3 GPGPU-empowered Parallel Data Marking

In a key-value separation-based KV store, the marking process is a critical step for merge-sorting. By counting invalid KV pairs, delayed GC can reduce system overhead from frequent GC operations. Recording invalid KV pairs' information facilitates faster cleanup during GC, minimizing query overhead. The CPU-based merge-sorting is compute-intensive and consumes significant resources, which struggles to parallelize the merge-sorting process, leading to performance bottlenecks. To address this issue, we leverage GPGPU to accelerate merge-sorting with parallel decoding, sorting, marking, uniqueness checks, and encoding. We also implement a parallel marking process that records the positions and total number of invalid KV pairs in each log file. This process enhances GC efficiency, supports delayed GC, and reduces query overhead. These GPGPU tasks may contend for device resources. We schedule them in CUDA streams so that data transfer completes before subsequent kernels start, preventing data inconsistency.

As shown in Fig. 8, we implement parallel marking on GPGPU to store the positions of invalid KV pairs in the `gpgpu_bitmap`. The key-v_ptr pairs decoded from STables are first sorted using the `sort` function from the Thrust library, arranging them in lexicographical order based on the key. In cases of duplicate keys, they are further sorted in the descending order of sequence numbers (the number in parentheses represents the sequence number). This approach ensures that duplicate keys are positioned together and sorted by sequence number, facilitating the subsequent marking process. Next, parallel marking is performed. Using the v_ptr of invalid keys, the GPGPU concurrently decodes the log file number K and position pos . The bit corresponding to K and pos in the GPGPU bitmap is computed and set to 0, indicating that this position is invalid, as highlighted in red in Fig. 8. Simultaneously, the number of invalid KV pairs in the log file is updated and stored in `invalid_count`.

By leveraging GPGPU for parallel merge-sorting and marking, we achieve significant improvements in compaction performance, reduce CPU load, and enhance overall system efficiency. Furthermore, by recording the number and positions of invalid KV pairs, we effectively minimize the overhead caused by frequent GC operations. In addition, the positions of invalid KV pairs allow us to bypass unnecessary query overhead, further optimizing the GC process.

After parallel marking, we use the `[thrust :: unique]` function to eliminate duplicate entries in `key_values`, facilitating the parallel merge-sorting of duplicate STables on the GPGPU. This function retains the key with the largest sequence number among the duplicate keys, discarding the others. Finally, the rest of KV pairs in

Algorithm 1 Parallel GC

```

1: Input:
2:   value_log: starting address of the log file data in GPGPU memory.
3:   valid_count: the number of valid KV pairs in the log file.
4:   max_value_logs: the maximum number of log files.
5: Output:
6:   output: starting address of the valid KV pairs in GPGPU memory.
7:
8: thread_workload  $\leftarrow$  GetThreadWorkload()
9: total_thread_count  $\leftarrow$  max_value_logs / thread_workload
10: Kernel Function ParallelGC():
11:   tid  $\leftarrow$  threadIdx.x + blockIdx.x * blockDim.x
12:   if tid  $\geq$  total_thread_count then
13:     return
14:   end if
15:   for idx in (tid * thread_workload, (tid + 1) * thread_workload) do
16:     if gpgpu_bitmap[idx] == 1 then
17:       memcpy(output + valid_count * kv_size, value_log + idx * kv_size, kv_size)
18:       /* Copy the KV pair to the output buffer */
19:       atomicAdd(valid_count, 1) /* Atomic increment operation */
20:     end if
21:   end for
22: End of Kernel ParallelGC

```

the container are encoded in parallel to produce SSTables that are subsequently written to the LSM-tree.

3.4 Parallel GC

We propose a GPGPU-empowered parallel GC mechanism in gParaKV to efficiently reclaim invalid data in log files. In the CPU-based key-value separation, the CPU sequentially traverses each KV pair in a log file to determine whether the KV pair is valid. It then discards the invalid data. While key-value separation reduces CPU usage, the GC process increases CPU overhead, degrading system performance. To enable parallel GC, we standardize the size and format of KV pairs in log files, allowing rapid computation of a KV pair's positions using indexes for fast access. For variable-sized KV pairs, we adopt a block-based partitioning approach, where each KV pair occupies one or more fixed-sized blocks. For instance, if the block size is 4 KB, a 2 KB KV pair occupies one block, while a 14 KB pair spans three blocks. This method facilitates parallel GC while minimizing storage overhead.

Before initiating GC, it is necessary to determine whether GC should be triggered. In gParaKV, when the number of invalid KV pairs in a log file exceeds a threshold (25%, set via flag `clean_threshold`), GC is triggered. This threshold balances GC overhead and storage space efficiency. We implemented a parallel GC triggering algorithm. When the number of invalid KV pairs in a log file accessed by a thread reaches the threshold, the parallel GC process is executed on the log file, and other threads exit.

When GC is triggered, the log file number K retrieves the log file, which is transferred to GPGPU memory with the starting address `value_log`. We initialize `valid_count` to zero to track valid KV pairs. To balance GPGPU thread overhead and GC parallelism, we dynamically calculate the number of KV pairs per thread (`thread_workload`, Line 7 in Algorithm 1) based on the total number of KV pairs in the log file. The total number of GPGPU threads (`total_thread_count`, Line 8) is obtained to avoid excessive thread allocation while maintaining efficient GC parallelism, thereby optimizing resource utilization and system performance. Next, parallel GC is performed, with each thread processing `thread_workload` KV pairs, verifying their validity. Valid KV pairs are written to the `output` buffer in

GPGPU memory (Lines 14–16). This buffer size is given as $(N_{\text{curr_invalid_count}} \times kv_{\text{size}})$, where `valid_count` tracks the number of valid KV pairs to ensure they are written sequentially. The `idx` – the index of each KV pair in `value_log` – allows quick location of each KV pair in `value_log`. When writing a valid KV pair, `valid_count` is atomically incremented (Line 17). Afterward, all valid KV pairs are written in parallel to the output buffer.

Finally, the KV pairs stored in the `output` buffer are batched with those generated by foreground Put threads and written to the LSM-tree in bulk. This approach reduces the blocking time for Put operations and leverages GPGPU resources to accelerate GC. The proposed mechanism mitigates the performance degradation caused by GC overhead, thereby enhancing overall system efficiency.

4 Experimental Setup

In the experiment, we use a server with two Intel Xeon(R) E5-2630 v4 @ 2.20GHz CPUs, 32 GB of DDR4 DRAM, and an Intel Optane PCIe SSD 900P Series with 480 GB. Using the FIO tool [11], the sequential write performance of the SSD is 1265 MB/s under a 100 GB data volume. A CPU has 10 cores and supports 20 threads. The operating system is Ubuntu 22.04, and its kernel is Linux Kernel 6.5.0-14-generic. We deploy an NVIDIA TESLA P100 GPGPU [24] with 16 GB HBM2 on a PCIe 3×16 lane interface. The P100 has 3,584 CUDA cores. We employ CUDA Toolkit v12.3 and LevelDB v1.23 to implement gParaKV. We study the performance of gParaKV using DB_Bench and YCSB-C [37]. There are seven workloads: Load, A, B, C, D, E, and F in YCSB-C.

These tested KV stores have a 4 MB Memtable and a 2 MB SSTable. Other configurations are set to the default values. We use BlobDB as the baseline and compare gParaKV with RocksDB [4], Wisckey [21], DiffKV [19], UniKV [41], GDH [31], and gLSM [30]. Notably, BlobDB, Wisckey, DiffKV, UniKV, and GDH are implemented based on the key-value separation. gLSM employs a GPGPU to accelerate compaction in KV stores without key-value separation, whereas gParaKV is designed to accelerate compaction and GC for key-value separation-based KV stores using the GPGPU.

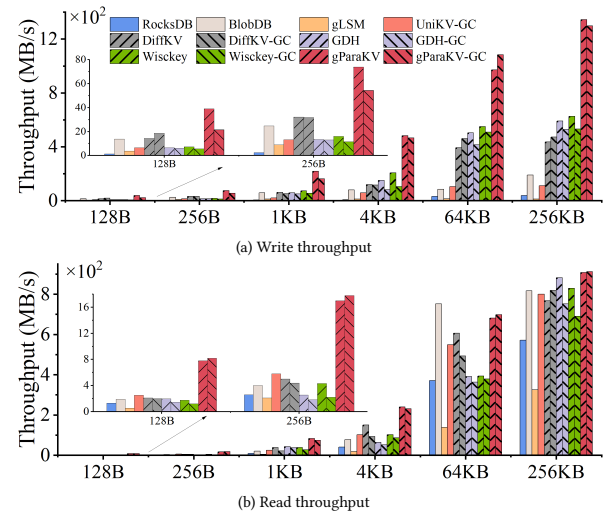


Figure 9: Throughput of tested KV stores under DB_Bench with different value sizes.

5 Performance Evaluation

5.1 Performance with DB_Bench

We study the performance of gParaKV in terms of write and read throughput, write latency, power consumption, space amplification (SA), and other metrics compared with alternative KV stores. In the experiment, we configure a 16 B key, six value sizes (128 B, 256 B, 1 KB, 4 KB, 64 KB, and 256 KB), and a 100 GB data volume in the workload.

5.1.1 Throughput. We evaluate the write performance of gParaKV under DB_Bench with a 100% write workload, see Fig. 9a. Symbol 'Wiskey-GC' means GC is triggered when invalid KV pairs exceed 25%, while 'Wiskey' indicates no conditions that trigger GC. gParaKV achieves higher write throughput than other KV stores as value sizes increase to 256 KB from 128 B. Specifically, with a 1 KB value size, gParaKV-GC outperforms BlobDB, UniKV-GC, DiffKV-GC, and Wiskey-GC by 2.47 $\times$, 6.86 $\times$, 2.71 $\times$, and 2.62 $\times$, respectively. This is attributed to GPGPU-accelerated merge-sorting during compaction and parallel GC, overcoming CPU bottlenecks and enhancing write throughput.

We also study the read performance of gParaKV under a 100% read workload in Fig. 9b. gParaKV outperforms other KV stores in most cases. With a 1 KB value size, gParaKV-GC achieves 3.64 $\times$, 3.05 $\times$, 3.47 $\times$, and 2.71 $\times$ higher read throughput than BlobDB, UniKV-GC, DiffKV-GC, and Wiskey-GC, respectively. This is because gParaKV-GC leverages GPGPU to accelerate merge-sorting and data-marking during compaction, achieving optimized SSTable layouts, low query overhead, and high read performance.

5.1.2 Write latency. Fig. 10 shows the write latency of gParaKV, including average, P90¹, and P99 latency. gParaKV achieves lower write latency than other KV stores. When the value size is 1 KB, gParaKV-GC reduces write latency by 64.42%, 87.20%, 67.62%, 69.40%, and 66.56% compared to BlobDB, UniKV-GC, DiffKV-GC, GDH-GC, and Wiskey-GC, respectively. This superior performance stems from efficient GPGPU-accelerated merge-sorting during compaction and GC, significantly reducing computational overhead and maintaining low latency even when GC is triggered.

5.1.3 Power consumption. We study the power consumption of tested KV stores by using the sysstat [14] to record CPU power consumption every two seconds and the nvidia-smi tool [23] to monitor GPGPU power consumption every second. In Fig. 11, gParaKV and gParaKV-GC achieve lower power consumption across all value sizes. With a 1-KB value, gParaKV-GC reduces power consumption by 60.28%, 82.99%, 47.19%, 23.80%, and 5.55% compared to BlobDB, UniKV-GC, DiffKV-GC, GDH-GC, and Wiskey-GC, respectively. This is because gParaKV-GC leverages GPGPU to accelerate merge-sorting during compaction and GC, and reduces computation overhead, lowering the power consumption.

5.1.4 Space amplification. The space amplification (SA) is defined as the ratio of actual storage space (including SSTables and log files) to the amount of data stored. In Fig. 12, gParaKV achieves lower SA across various workloads due to its efficient key-value separation,

parallel compaction, and GPGPU-accelerated GC, significantly reducing data redundancy and fragmentation. When the value size is 128 B, there are maximum numbers of KV pairs in a fixed-size data volume, causing the highest GC overhead. With a 128 B value size, gParaKV-GC reduces SA by 23.48%, 58.44%, 6.48%, 25.74%, and 16.53% compared to BlobDB, UniKV-GC, DiffKV-GC, GDH-GC, and Wiskey-GC, respectively, demonstrating superior GC efficiency and optimized space usage.

5.1.5 Compaction throughput. We evaluate the compaction throughput of various KV stores to assess their merge-sorting performance, as shown in Fig. 13. Compaction throughput is calculated by dividing the total data read and written during compaction by the compaction time. Since compaction is not triggered in the case of a 256 KB value, the valid compaction throughput is capped at 64 KB. Compared to other schemes, gParaKV and gParaKV-GC achieve higher compaction throughput. For instance, with a 1 KB value, their throughput is 12.09 $\times$ and 11.25 $\times$ higher than BlobDB, respectively. This improvement is attributed to the GPGPU acceleration of the compute-intensive merge-sorting task during compaction. This highlights merge-sorting as a major computational bottleneck in both traditional and key-value separation KV stores. Although gLSM also uses GPGPU acceleration, its partial offloading strategy and large-value handling overhead limit performance, especially for small values. For larger values, gParaKV and gParaKV-GC slow down the throughput because larger values result in fewer KV pairs involved in merge-sorting and GC, which impairs the GPGPU acceleration effect. Additionally, more frequent data transfers decrease compaction performance.

5.1.6 GPGPU utilization and memory usage. In Fig. 14, we measure the average GPGPU utilization, as well as the average and peak CPU memory usage for gLSM, GDH, GDH-GC, gParaKV, and gParaKV-GC. gParaKV and gParaKV-GC show the highest GPGPU utilization, indicating that our approach effectively leverages GPGPU parallel computing resources to achieve higher performance. Although gParaKV-GC consumes more GPGPU memory due to GC operations, its usage remains within 2 GB. Since most modern servers are equipped with significantly larger GPGPU memory, this does not cause a memory shortage.

5.1.7 Data transfer time. To evaluate the impact of data transfer on performance, we measure the transfer time and its ratio for gParaKV-GC, see Table 1. The data transfer time includes the time required to transfer the selected SSTables from the SSD to GPGPU memory during compaction, the time to transfer new SSTables from GPGPU memory back to the SSD, the data transfer time between CPU memory and GPGPU memory, and the time to transfer log files from the SSD to GPGPU memory during GC. At a 200 GB data volume, although the data transfer time ratio reaches its maximum of 26.8%, the system still maintains a high throughput and is not significantly affected by transfer overhead. Thus, data transfer does not cause a performance bottleneck in gParaKV.

5.1.8 Variable-sized values. To verify gParaKV-GC's ability to support variable-sized values in a GPGPU-accelerated setting, we conducted extensive tests with value sizes uniformly distributed from 32 B to 16 KB. In Table 2, gParaKV-GC maintains high throughput

¹P90 represent latency percentiles, indicating that the latency of 90% of requests is less than or equal to the value of latency, while the remaining 10% are slow.

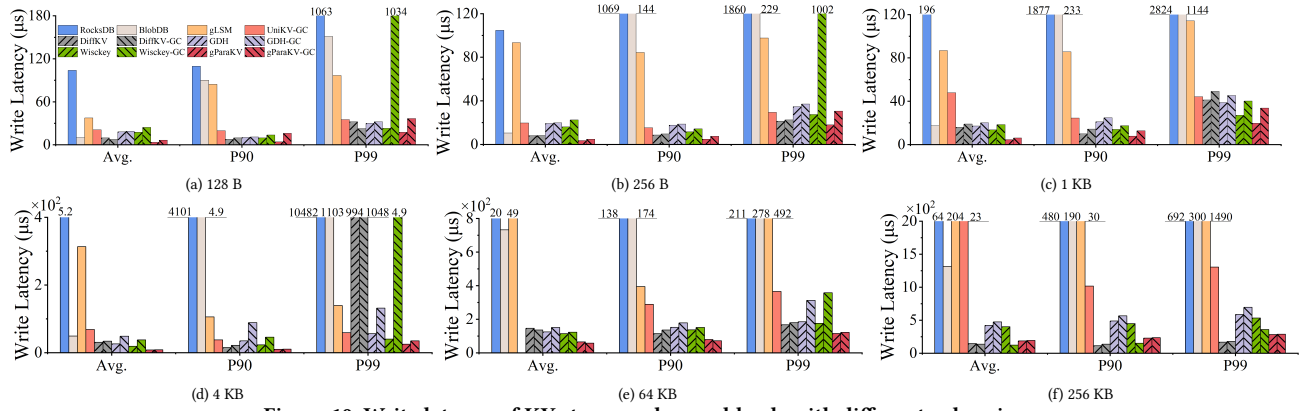


Figure 10: Write latency of KV stores under workloads with different value sizes.

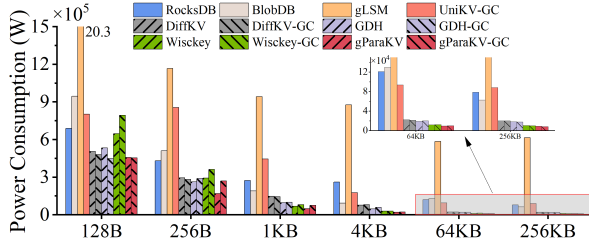


Figure 11: Power consumption of tested KV stores under different value sizes.

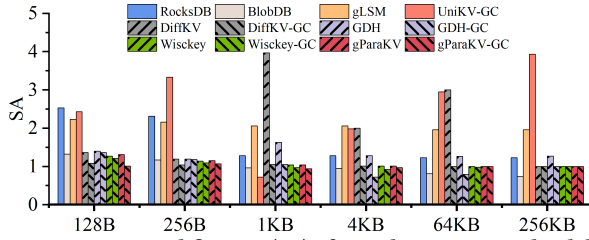


Figure 12: Space amplification (SA) of tested KV stores under different value sizes.

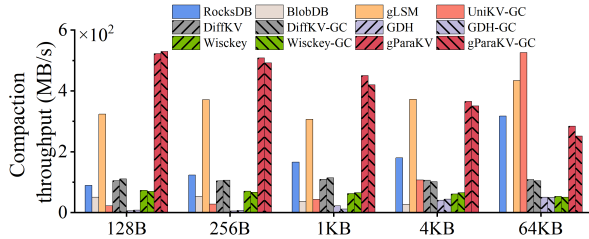


Figure 13: Compaction throughput of tested KV stores under different value sizes.

Table 1: Data transfer time and its ratio under a 1KB value size with different data volumes.

Data volume (GB)	Transfer time (sec)	Total time (sec)	Ratio (%)	Throughput (MB/s)
5	2.6	24.1	10.8	206.1
10	7.7	54.8	14.1	181.0
20	18.4	111.0	16.6	178.6
50	58.2	270.9	21.8	183.1
100	159.6	611.7	26.1	162.1
200	310.9	1159.1	26.8	171.1

Table 2: Throughput and space amplification of gParaKV-GC under variable value sizes and different numbers of KV pairs (in millions).

Number (M)	Throughput (MB/s)	Compaction throughput (MB/s)	SA
1	473.9	100.8	0.78
2	479.2	161.4	0.80
5	389.5	238.9	0.80
10	417.7	230.5	0.78
20	413.6	246.3	0.78

across all scenarios. It also performs GC efficiently, reducing space amplification under workloads with variable-sized values.

5.2 Performance under YCSB-C

Fig. 15 shows the performance of gParaKV under YCSB-C with a 16 B key, three value sizes (1 KB, 4 KB, and 16 KB), and a 100 GB data volume.

With various value sizes, gParaKV shows better performance than other KV stores under workloads. With 1 KB value, the performance of gParaKV-GC achieves 328.5%, 5.0%, 40.6%, 178.7%, 71.0%, 405%, and 24.8% higher than that of BlobDB under workloads Load, A, B, C, D, E, and F, respectively. This is because gParaKV leverages GPGPU parallelism to accelerate merge-sorting during compaction and GC, improving gParaKV-GC's performance. In contrast, BlobDB's merge-sorting suffers from computational bottlenecks, resulting in slower compaction and poorer performance. gParaKV and gParaKV-GC have an edge over other schemes. gLSM also uses GPGPU to improve performance; however, gParaKV solves the high WA issue of gLSM through the key-value separation. gParaKV applies GPGPU to accelerate compaction and GC in parallel, which alleviates the high CPU overhead of GC in the key-value separation-based KV stores.

gParaKV shows better performance under write-intensive workloads with all value sizes. We use GPGPU to accelerate compaction and improve merge-sorting performance. This advantage clearly indicates that gParaKV has addressed the issues in key-value-separated KV stores. First, existing key-value-separated KV stores have not fully resolved the computational bottlenecks in merge-sorting during compaction. Second, in GC, scanning log files and querying the validity of data increase GC computation time. Implicit GC further

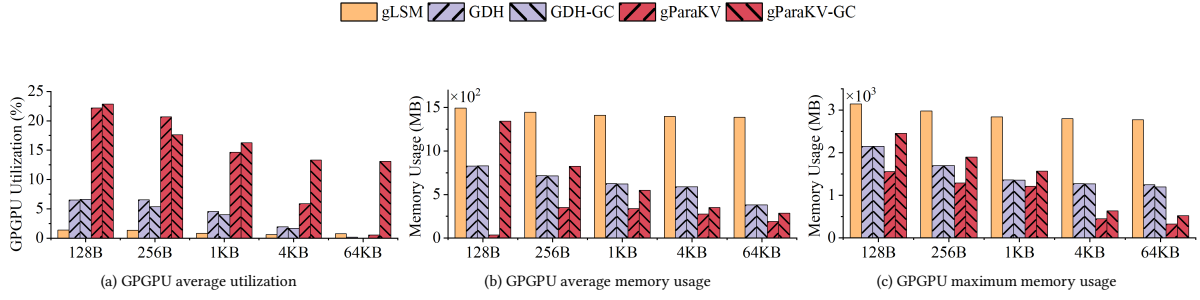


Figure 14: GPGPU average utilization (a) and average (b) and maximum memory usage (c) with different value sizes.

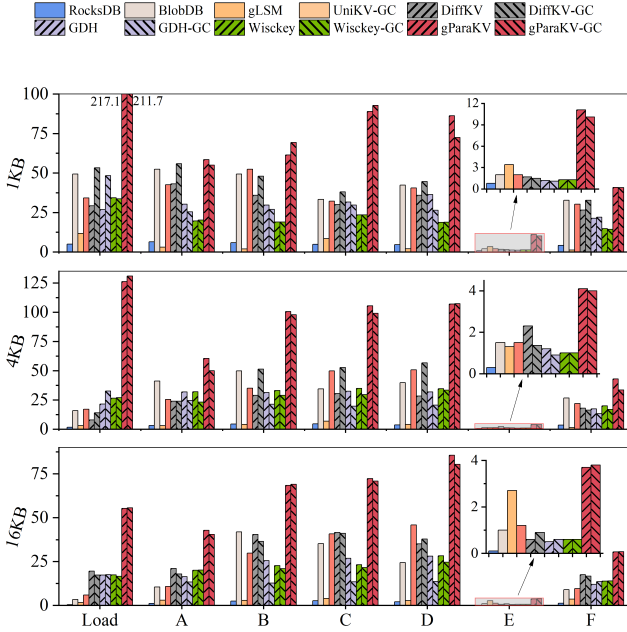


Figure 15: Throughput (KPTS) of KV stores under YCSB-C with three value sizes.

increases compaction time, which means that existing GC strategies degrade overall system performance.

Under read-intensive workloads B, C, and D, gParaKV and gParaKV-GC outperform other KV stores across all value sizes due to GPGPU-accelerated merge-sorting during compaction, reducing access to duplicate SSTable and enhancing write/read performance. Under mixed read-write-update workloads A and F, gParaKV and gParaKV-GC maintain better performance by accelerating merge-sorting and data-marking, improving write/read performance. Under update workloads, gParaKV-GC identifies invalid data in parallel using a GPGPU bitmap, accelerating the GC process. Additionally, under range-query workload E, gParaKV-accelerated compaction reduces CPU usage, leaving more available CPU resources to process range queries efficiently.

6 Related Work

Key-Value Separation. LSM-based KV stores without key-value separation offer high write performance but suffer from significant

WA. Key-value separation approaches effectively reduce I/O amplification and improve the overall performance by storing keys and values separately [2, 25]. WiscKey [21] optimizes performance via key-value separation but incurs high GC overhead under update-intensive workloads. Bourbon [9] builds upon WiscKey by incorporating learned indexes to enhance query efficiency. HashKV [7] employs hash-based data grouping for efficient GC. UniKV [41] integrates hash indexing with LSM-trees using data locality discretizations. DiffKV [19] optimizes key-value separation through fine-grained sorting. FenceKV [34] improves range-query performance using a partitioning strategy and key-range-based GC. Scavenger [40] balances the trade-offs between performance and space amplification. However, existing methods have not optimized computational overhead during compaction and GC, causing performance bottlenecks. gParaKV employ GPGPU to accelerate merge-sorting and GC, alleviating these performance bottlenecks.

Compaction optimization. The LSM-tree structure in KV stores improves write performance by compaction, but compaction introduces high tail latency and WA. Several methods have been proposed to mitigate these issues. PebblesDB [27] employs fragmented LSM-trees with overlapping SSTable key ranges, reducing WA at the cost of higher query overhead and increased space amplification. Lower-level driven compaction (LDC) [5] and adaptive LDC [6] utilize bottom-up triggering mechanisms to reduce tail latency. PStore [29] leverages asynchronous parallel compaction via multi-task scheduling, exploiting near-data processing [35]. Hardware acceleration approaches (e.g., FPGA [18, 32, 42] and GPGPU [30, 36, 43]) have also been proposed. LUDA uses the pipeline method to boost compaction, whereas gLSM [30] accelerates compaction using GPGPU. Despite these improvements, optimizations have not sufficiently addressed the substantial data movement overhead. Additionally, WA constrains performance from heterogeneous computing.

7 Conclusion

In this study, we proposed a GPGPU-empowered KV store named gParaKV, which employs parallel computing to lessen the compaction performance bottleneck and high CPU resource usage for key-value separation-based KV stores. We designed a GPGPU bitmap structure to accommodate invalid data marking in parallel. We also devised a parallel merge-sorting and data marking that fully utilizes GPGPU to accelerate compaction. Additionally, we developed a parallel GC mechanism to accelerate computations during GC, alleviating CPU bottlenecks and improving GC efficiency.

The experimental results demonstrate that gParaKV, compared to the existing systems, achieves higher performance.

Acknowledgments

We would like to thank the paper and artifact reviewers for their helpful feedbacks. 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).

References

- [1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.
- [2] Bytedance. 2024. TerarkDB. <https://github.com/bytedance/terarkdb>.
- [3] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. POLARDB meets computational storage: Efficiently support analytical workloads in Cloud-Native relational database. In *18th USENIX conference on file and storage technologies (FAST 20)*. 29–41.
- [4] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.
- [5] Yungpeng 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 *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 722–733.
- [6] Yungpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, and Yangyang Wang. 2020. Adaptive Lower-Level Driven Compaction to Optimize LSM-Tree Key-Value Stores. *IEEE Transactions on Knowledge and Data Engineering* 34, 6 (2020), 2595–2609.
- [7] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.
- [8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Walach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
- [9] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnathan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.
- [10] Facebook. 2024. BlobDB Documentation. <https://github.com/facebook/rocksdb/wiki/BlobDB>.
- [11] Github. 2024. Fio. <https://github.com/axboe/fio>.
- [12] Github. 2024. LevelDB. <https://github.com/google/leveldb>.
- [13] Github. 2024. stress-ng. <https://github.com/ColinIanKing/stress-ng>.
- [14] Github. 2024. Sysstat. <https://github.com/sysstat/sysstat>.
- [15] Tyler Harter, Dhruva Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2014. Analysis of {HDFS} Under {HBase}: A Facebook Messages Case Study. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. 199–212.
- [16] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.
- [17] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 191–205.
- [18] 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 *2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 1–8.
- [19] Yongkun Li, Zhen Liu, Patrick PC 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 *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 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.* (2024).
- [21] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.
- [22] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.
- [23] NVIDIA. 2024. NVIDIA nvidia-smi. <https://developer.nvidia.com/nvidia-system-management-interface>.
- [24] NVIDIA. 2024. NVIDIA Tesla P100. <https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture>.
- [25] Pingcap. 2024. TiDB Documentation. <https://download.pingcap.org/tidb-stable-en-manual.pdf>.
- [26] Yunhui Qiu, Jinyu Xie, Hankun Lv, Wenbo Yin, Wai-Shing Luk, Lingli Wang, Bowei Yu, Hua Chen, Xianjun Ge, Zhijian Liao, et al. 2020. Full-kv: Flexible and ultra-low-latency in-memory key-value store system design on cpu-fpga. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1828–1844.
- [27] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proc. Symp. Operating Systems Principles*. 497–514.
- [28] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.
- [29] Hui Sun, Bendong Lou, Chao Zhao, Deyan Kong, Chaowei Zhang, Jianzhong Huang, Yinliang Yue, and Xiao Qin. 2023. An Asynchronous Compaction Acceleration Scheme for Near-Data Processing-enabled LSM-Tree-based KV Stores. *ACM Transactions on Embedded Computing Systems* (2023). In press.
- [30] 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 Transactions on Storage* 20, 1 (2024), 1–41.
- [31] Hui Sun, Jinfeng Xu, and Xiao Qin. 2023. Elevating Performance of LSM-Tree-Based Key-Value Stores with Gradient Data Hierarchy. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 360–369.
- [32] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. Fpga-based compaction engine for accelerating lsm-tree key-value stores. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1261–1272.
- [33] Mark Sutherland, Babak Falsafi, and Alexandros Daglis. 2022. Cooperative Concurrency Control for Write-Intensive Key-Value Workloads. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 1. 30–46.
- [34] Chenlei Tang, Jiguang Wan, and Changsheng Xie. 2022. FenceKV: Enabling Efficient Range Query for Key-Value Separation. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3375–3386.
- [35] Tobias Vincon, 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 htp workloads. *Proceedings of the VLDB Endowment* 15, 10 (2022), 1991–2004.
- [36] 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. *arXiv preprint arXiv:2004.03054* (2020).
- [37] Yahoo. 2024. YCSB-C. <https://github.com/basichinker/YCSB-C>.
- [38] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.
- [39] Heejin Yoon, Jin Yang, Juyoung Bang, Sam H Noh, and Young-ri Choi. 2024. Advocating for Key-Value Stores with Workload Pattern Aware Dynamic Compaction. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*. 124–130.
- [40] Jianshun Zhang, Fang Wang, Sheng Qiu, Yi Wang, Jiaxin Ou, Junxun Huang, Baoquan Li, Peng Fang, and Dan Feng. 2024. Scavenger: Better Space-Time Trade-Offs for Key-Value Separated LSM-trees. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 4072–4085.
- [41] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward high-performance and scalable KV storage in mixed workloads via unified indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 313–324.
- [42] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, et al. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 225–237.
- [43] 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 *38th International Conference on Massive Storage Systems and Technology (MSST 2024)*.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

- C₁ We developed a GPGPU-accelerated merge-sorting during compaction that decodes, sorts, marks duplicates, filters, and encodes KV pairs in parallel, thereby optimizing throughput, GPGPU resource utilization, and latency.
- C₂ We proposed a GPGPU bitmap structure to record the validity of each KV pair. An array is designed to record the number of invalid KV pairs for each log file to delay GC, boosting the parallel marking of invalid KV pairs for GC and lessening the GC overhead.
- C₃ We proposed a GPGPU-empowered parallel GC strategy that can dynamically adjust the number of GC threads according to the number of log files, achieving low space amplification and high GC efficiency.
- C₄ We implemented gParaKV and gParaKV-GC on top of LevelDB, achieving higher write throughput by 2.62× and lower latency by 67.62%, compared with state-of-the-art KV stores.

A.2 Computational Artifacts

This paper presents computational artifacts associated with the development and evaluation of gParaKV and gParaKV-GC. '**gParaKV-GC**' means GC is triggered when invalid KV pairs exceed 25%, while '**gParaKV**' indicates no conditions that trigger GC. The artifacts include the open-source codes for gParaKV with and without GC. The scripts are used to run benchmarks and all figures presented in the paper. These artifacts are available via two DOIs for full reproducibility.

A₁ gParaKV: <https://doi.org/10.5281/zenodo.15254195>

A₂ gParaKV-GC: <https://doi.org/10.5281/zenodo.15258795>

Artifact ID	Contributions Supported	Related Paper Elements
A ₁	C ₁ and C ₄	Figures 9 - 15
A ₂	C ₁ , C ₂ , C ₃ , and C ₄	Tables 1 - 2 Figures 9 - 15

B Artifact Identification

B.1 Computational Artifact A₁

Relation To Contributions

We proposed a GPGPU-based merge-sorting mechanism on top of LevelDB. We implemented five critical modifications to the source codes of LevelDB. First, we proposed the **DoGPUCompactionWork** function to accelerate the merge-sorting operation during compaction using the GPGPU resources. Second, in the **DecodeSSTables** function, we implemented the parallel decoding of SSTables and parallel sorting of KV pairs. Third, we devised the **Mark** member function in the **GPUGC** class to perform the parallel marking of invalid KV pairs. Fourth, the **GPUUnique** function is designed to

filter out invalid KV pairs. Fifth, we proposed the **BuildSSTables** function to perform parallel encoding of SSTables.

Expected Results

gParaKV achieves higher write throughput and lower latency than alternative KV stores under write-intensive workloads. The experimental results demonstrate that the GPGPU-accelerated merge-sorting mechanism enables gParaKV (without GC) to achieve the highest performance.

Expected Reproduction Time (in Minutes)

- **Artifact Setup:** 30 minutes – Installing dependencies (CUDA, LevelDB).
- **Artifact Execution:** 30 minutes – Running the benchmarks on gParaKV using 100 GB of data.
- **Artifact Analysis:** 10 minutes – Analyzing compaction throughput, GPGPU resource utilization, and latency.

B.2 Computational Artifact A₂

Relation To Contributions

We implemented a parallel GC strategy in gParaKV, named gParaKV-GC. To implement the parallel GC, we designed a GPGPU bitmap structure in the **GPUGC** class, using a `uint8_t` array called **gpu_flags**. We also designed the **MarkInvalidKeysKernel** kernel function to perform the parallel marking of invalid KV pairs. The **TriggerGCKernel** kernel function is proposed to trigger GC operations in parallel. The **GPUGCOptimizedKernel** kernel function is devised to perform parallel GC. In the **BeginGPUGCOptimized** function, we implemented a dynamic allocation strategy for GC threads according to the number of log files. In addition, we developed the **MaybeScheduleGarbageCollection** function in the **MakeRoomForWrite** function and designed the **TriggerGC** function to determine whether GC is triggered. A background thread is created to execute the parallel GC when the GC process is triggered. Through these modifications, we achieved low resource consumption and high GC efficiency in the key-value-separated gParaKV-GC.

Expected Results

gParaKV-GC with the parallel GC strategy significantly reduces GC latency and space amplification. This artifact shows that gParaKV achieves 2.62× higher throughput and 67.62% lower latency, compared to state-of-the-art KV stores.

Expected Reproduction Time (in Minutes)

- **Artifact Setup:** 30 minutes – Installing dependencies (CUDA, LevelDB).
- **Artifact Execution:** 45 minutes – Running the benchmarks on gParaKV-GC using 100 GB of data.
- **Artifact Analysis:** 15 minutes – Analyzing GC performance, throughput, and space amplification.

Artifact Setup (incl. Inputs)

Hardware. The experiment is conducted on a server equipped with two Intel Xeon(R) E5-2630 v4 @ 2.20 GHz CPUs, providing 10 cores and 20 threads per CPU. The system is configured with 32 GB of DRAM and an Intel Optane PCIe SSD 900P Series (480 GB). The SSD achieves a sequential write performance of 1265 MB/s under workloads with a 100 GB data volume. To accelerate the performance of KV stores, we employed an NVIDIA TESLA P100 GPGPU with 16 GB DRAM and 3,584 CUDA cores, and this GPGPU connects to the server through a PCIe 3×16 lane interface.

Software. We conducted experiments on Ubuntu 22.04 with Linux Kernel 6.5.0-14-generic. We utilized CUDA Toolkit v12.3 for GPGPU and LevelDB v1.23 to implement gParaKV and gParaKV-GC. The latest version of Snappy and gflags are also installed.

Datasets / Inputs. We employed DB_Bench and YCSB-C as benchmarks in the experiment. DB_Bench workloads are configured with a 16 B key, six value sizes (128 B, 256 B, 1 KB, 4 KB, 64 KB, and 256 KB), and a 100 GB data volume. YCSB-C workloads include a 16 B key, three value sizes (1 KB, 4 KB, and 16 KB), and a 100 GB data volume. YCSB-C includes seven workloads: Load, A, B, C, D, E, and F. BlobDB is the baseline KV store. We study the performance of gParaKV against alternative KV stores, such as RocksDB, Wisckey, DiffKV, UniKV, GDH, and gLSM.

Artifact Execution

The experiment workflow consists of the following tasks:

Task T_1 : Scripts. This task provides scripts for various experiments in the `.scripts` directory, with the executable script file `run.sh`. You need to modify the directories of the `db_bench` and `ycsb` executable files.

For DB_Bench, you need to modify the parameter `db` to resolve to a validated directory where the database is located. You can run different workloads by modifying the parameters `benchmarks`, `value_size`, and `num`. If you merely measure the write throughput, you can reconfigure `-benchmarks=fillrandom`. To study the read throughput, you can configure `-benchmarks=fillrandom,readrandom`. The `value_size` affects the value of `num`, and you need to ensure that the size of the KV pairs multiplied by `num` equals 100 GB.

For YCSB-C, you need to modify the parameter `filename` to resolve to a validated directory where the database is located. The YCSB-C workload files are stored in the `.workloads` directory, and the workload parameters are written in the `*.spec` files. For example, `recordcount` represents the number of KV pairs. `fieldlength` means the value size, while `requestdistribution` defines the workload distribution (such as uniform or Zipfian). You can specify the parameter `P` as a spec file to run different YCSB-C workloads.

Task T_2 : Scripts Execution. The second task is to run the scripts. You need to enter the command `./run.sh` to execute the script. If you run DB_Bench workloads, the script will be executed to obtain the write/read throughput, latency, and compaction throughput of tested KV stores in cases of six value sizes. If you run YCSB-C workloads, the script will execute different workloads in YCSB-C (e.g., Load, A, B, C, D, E, and F). These scripts are used to measure the performance of gParaKV (with and without GC) in terms of throughput and latency and compare it with alternative KV stores.

Task T_3 : Result Analysis. This task analyzes the output of tested KV stores using scripts, which are stored in `*.txt` file. The results present the performance metrics (such as throughput, latency, and space amplification), see Tables 1 - 2 and Figures 9 - 15. These results are then used to compare gParaKV with alternative KV stores in terms of write/read throughput, compaction throughput, and latency.

The tasks depend on each other as follows: $T_1 \rightarrow T_2 \rightarrow T_3$.

Experimental Parameters:

- **Dataset Size:** 100 GB of KV pairs, where a key size is 16 B and the value size varies across the experiments.
- **Numbers of KV Pairs:** The number of KV pairs is set to 10 million for each benchmark to simulate realistic workloads.
- **Value Sizes:** DB_Bench is configured with six value sizes—128 B, 256 B, 1 KB, 4 KB, 64 KB, and 256 KB. YCSB-C is featured with three value sizes: 1 KB, 4 KB, and 16 KB.

We opt to use the parameters above to validate the performance of gParaKV in most scenarios. The dataset size (100 GB) represents large-scale workloads, while these value sizes can be used to evaluate the performance across various KV-pair sizes.

Artifact Analysis (incl. Outputs)

Under DB_Bench and YCSB-C, the outputs of experiments are stored in `*.txt` files, which contain performance metrics, such as throughput and latency in cases of various value sizes and data volumes, see Tables 1 - 2 and Figures 9 - 15. These results show the performance of gParaKV and gParaKV-GC compared to other KV stores.

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

Installation and Deployment. To compile, deploy, and execute the experiments, the following tools and dependencies are required:

- (1) It ensures that the system has all necessary packages installed to build and run the project:

```
sudo apt update

sudo apt install build-essential cmake git libsnappy-
    ↪ dev libgflags-dev

sudo apt install nvidia-driver-550

sudo apt install nvidia-cuda-toolkit

sudo reboot
```

- (2) Clone the project repository and initialize the required sub-modules:

```
git clone https://github.com/AHUKV/gParaKV.git

cd gParaKV

git submodule update --init --recursive
```

- (3) Create the build directory and build the project:

```
mkdir build && cd build

cmake ..

cd ..
```

- (4) Build the targets required for workloads DB_Bench and YCSB-C:

```
cmake --build build --target db_bench -j8

cmake --build build --target ycsbc -j8
```

- (5) Configuration files: Before running the tests, it should ensure that the configuration files are set correctly. The database paths for db_bench and YCSB-C need to be configured:

- DB_Bench: In the './db_bench' directory, the '-db' parameter in the configuration file should point to the database directory you wish to use (e.g., '/tmp/test').
- YCSB-C: Ensure that the correct workload configuration file is used in the './workloads/100/' directory. The default file is 'workloadb.spec', but if you want to use another configuration (e.g., 'workloada.spec'), it can be specified using the '-P' parameter.

After the compilation is complete, the benchmark tests can be run. In the experiment, we used DB_Bench and YCSB-C as benchmarks.

Workload DB_Bench is configured with 16B keys, six value sizes (128B, 256B, 1KB, 4KB, 64KB, and 256KB), and a 100GB dataset. The following commands are used to measure write throughput, read throughput, and compaction performance:

```
./build/db_bench --benchmarks=fillrandom --value_size=<
  ↳ VALUE_SIZE> --num=100000000 --db=/tmp/test
```

In this command:

- 'benchmarks' specifies the type of workload.
- 'value_size' represents the size of each value.
- 'num' represents the number of KV pairs.
- 'db' is the directory where the database is located.

In the experiment, you can use DB_Bench to test various metrics such as write/read throughput and write latency. You can also modify the 'num' and 'value_size' parameters to conduct tests under different conditions. Then, the 'benchmarks' parameter (e.g., 'readrandom') is changed to test with other workloads.

Workload YCSB-C allows for configurable key sizes, value sizes, and read/write query ratios. In the experiment, we tested with 16B keys and three value sizes (1KB, 4KB, and 16KB) in a 100GB dataset. However, you can configure their own key and value sizes as well as adjust the read/write query ratio according to requirements. YCSB-C includes seven workloads: Load, A, B, C, D, E, and F. BlobDB is used as the baseline KV store.

To run workload YCSB-C, the following commands are used:

```
./build/ycsbc -filename /tmp/test -db leveledb -configpath 0
  ↳ -P ./workloads/100/workloadb.spec
```

In this command:

- 'filename' is the directory where the database is located.

- 'db' and 'configpath' specify the default configuration path.
- '-P' specifies the workload file path.

You can choose other workload files from the './workloads' directory or manually configure the workload files.

Artifact Execution

We present the experimental workflow as following:

Evaluation 1: Throughput: In this experiment, we evaluate the throughput of gParaKV under DB_Bench with 100% write (fillrandom) and 100% read (readrandom). We configure six value sizes (128 B, 256 B, 1 KB, 4 KB, 64 KB, and 256 KB), and insert 100 million key-value pairs with a 100 GB dataset. For each configuration, the following command is executed:

```
./build/db_bench --benchmarks=fillrandom,readrandom --
  ↳ value_size=<VALUE_SIZE> --num=100000000 --db=/tmp/
  ↳ test
```

After a benchmark, we record the throughput from the DB_Bench output, focusing on the lines ending with fillrandom: ... ops/sec and readrandom: ... ops/sec. The write throughput results are used to be presented in Figure 9a, while the read throughput results are used in Figure 9b.

Evaluation 2: Space Amplification: In a test, we evaluate the space amplification (SA) of gParaKV under the same fillrandom workloads. SA is defined as the ratio of actual disk space used (including SSTables and log files) to the total user-inserted data volume. In DB_Bench, the output includes an SA line such as:

```
Original write to disk: 992 MB, new write to
disk: 960 MB, space amplification: 0.97
```

This value is recorded after each test and represents the SA. The SA values across various configurations are summarized and presented in Figure 12. No additional scripts or parameter modifications are required beyond selecting the appropriate value_size.

Evaluation 3: Write Latency: In this experiment, we evaluate the write latency of gParaKV, including the average latency, P90 latency, and P99 latency. The latency is obtained using the histogram class in DB_Bench. The percentile function is used to calculate the percentiles (e.g., P90, P99). The latency calculation estimates the value of latency at the desired percentile based on the cumulative values and thresholds of each bucket.

The latency is output using the ToString function, which includes the following metrics:

- Average latency (Average)
- Standard deviation (Standard Deviation)
- Minimum value (Min)
- Maximum value (Max)
- Median value (Median)
- Latency at different percentiles: e.g., P50, P75, P90, P99, etc.

For example, we present an output as following:

```
Count: 100000000 Average: 0.0054 StdDev: 0.0023
Min: 0.0012 Median: 0.0043 Max: 0.0321
Percentiles: P50: 0.0043 P75: 0.0061 P90: 0.0087 P99: 0.0132
P99.9: 0.0153 P99.99: 0.0171
-----
```

In this experiment, we focus on P90 and P99 latencies, which represent the 90% and 99% of writes' latencies. We can obtain the data from runtime logs and used in subsequent analysis to evaluate gParaKV's performance in cases of different value_size configurations. The results are shown in Figure 10.

Evaluation 4: Compaction Throughput and Data Transfer Time: This experiment is performed to study the compaction throughput and data transfer time of gParaKV-GC under DB_Bench with 100% writes. The same command as in Evaluation 1 is used, and all metrics are collected from the DB_Bench output.

Compaction throughput is computed as the total amount of read/write data during compaction divided by the total compaction time. The output of DB_Bench includes the formula and calculated result. For example, we have

compaction throughput = (read + write) / time = (78.04 + 66.03) / 0.344832 = 417.80 MB/s

These results are plotted in Figure 13. Meanwhile, we also record the data transfer time during the same compaction process:

- Transferring SSTables from the SSD to GPGPU memory
- Transferring newly generated SSTables back to the SSD
- Data movement between CPU and GPGPU memory
- Transferring log files to GPGPU memory during GC

Our system reports the data transfer time directly, e.g.:

'data transfer time: 439601 us'

We calculate the ratio of data transfer time to total time. At a data volume of 200 GB, the ratio reaches 26.8% (maximum). However, gParaKV-GC maintains the high throughput despite the transfer overhead. These results are summarized in Table 1.

Evaluation 5: Throughput and space amplification: This experiment is designed to verify gParaKV-GC's ability to support variable-sized values in a GPGPU-accelerated setting. We configured different numbers of KV pairs (in millions) in this test. gParaKVGC maintains high throughput across all scenarios. It also performs GC efficiently, reducing space amplification under workloads with variable-sized values. These results are summarized in Table 2.

Evaluation 6: Power Consumption and GPGPU Utilization: This tested results are presented in Sections 5.1.3 and 5.1.6 in the paper, see Figures 11 and 14. This test is used to evaluate the power consumption and GPGPU usage of gParaKV, gParaKV-GC, GDH, and gLSM, under DB_Bench with a 1 KB value size and 100 GB dataset.

Measurement Setup

CPU power consumption is measured using Intel RAPL interfaces:

- /sys/class/powercap/intel-rapl:0/energy_uj
- /sys/class/powercap/intel-rapl:1/energy_uj

GPU power and resource usage is obtained by

- nvidia-smi -query-gpu=timestamp,power.draw,utilization.gpu, memory.used,memory.total

Sampling is conducted once per second throughout the workload.

How to Run

We provide two ready-to-use scripts located in the scripts/ directory:

1. cpu_record.sh Periodically polls CPU energy counters and logs memory and CPU utilization to cpu.log.
2. gpu_record.sh Executes the workload (db_bench) and concurrently monitors GPU power,

utilization, and memory usage using nvidia-smi, storing results in gpu.log.

To launch the experiment, simply run:

```
cd scripts/Power Consumption and GPGPU Utilization

./gpu_record.sh

sudo ./cpu_record.sh
```

It does not need to modify paths or environment variables. All parameters (e.g., workload size, value size, data path) are preconfigured in the script and can be modified as needed. It should be ensure that the dataset path has write permissions and that the necessary monitoring interfaces are accessible with sudo privileges.

Output Files

- cpu.log: Logs per-second CPU memory usage, CPU usage, and CPU power (W). - gpu.log: Logs per-second GPU power draw, utilization (%), and memory usage (MiB).

These files contain raw tested data in the readable format.

Result Reproduction (Postprocessing)

To produce the results shown in the paper:

- Figure 11 (Power Consumption):

Total CPU Energy (J): Sum all values in the Power Usage column of cpu.log.

Total GPU Energy (J): Sum all values in the power.draw column of gpu.log, after subtracting the baseline idle GPU power (e.g., measured before workload execution).

Total System Energy (J) = CPU Energy + GPU Energy

- Figure 14 (GPGPU Utilization and Memory Usage):

Average GPU Utilization (%) = average of utilization.gpu column in gpu.log.

Average/Peak GPU Memory Usage (MiB) = mean and max of memory.used column in gpu.log.

We parse the logs into spreadsheets (as shown in the figure data) and calculate the final values using standard averaging formulas.

Evaluation 7: Performance Under Realistic Workloads: In this experiment, we evaluate gParaKV's performance under realistic workloads – YCSB-C. These previous experiments focus on the performance of different modules of gParaKV; however, this test employs real-world workloads to evaluate the system's performance in different applications. YCSB-C is widely used to provide seven different workloads, including read-only and mixed read-write operations.

In the experiment, we configure three value sizes (1 KB, 4 KB, and 16 KB) in a 100 GB dataset. We provide a description of seven YCSB-C workloads:

- Load: Initialization workload, primarily used to insert data into the database.
- A: Read-only workload, where the client randomly reads data.
- B: Update workload, where the client randomly reads and updates data.
- C: Read and update workload, where the client performs both read and update operations.
- D: Range query workload, where data is read by range.
- E: Mixed read-write workload, where most operations are reads, with some writes.

- F: Fully mixed workload, with a 50:50 read/write ratio.

These workloads simulate database patterns in realistic scenarios and are used to validate gParaKV's performance.

The following command is used to run YCSB-C:

```
./build/ycsbc -filename /tmp/test -db leveldb -configpath 0
➔ -P ./workloads/100/workloadb.spec
```

This command runs YCSB-C (e.g., Load, A, B, C, D, E, and F) and records performance in terms of throughput, latency, and GC overheads for gParaKV. The results are shown in Figure 15, which demonstrates gParaKV's outstanding performance under realistic workloads.

Artifact Analysis (incl. Outputs)

Evaluation 1: Throughput: gParaKV-GC outperforms alternative KV stores in terms of write and read throughput, particularly in cases of smaller value sizes that incur more frequent compaction and GC operations. For example, with a 1 KB value size, gParaKV-GC achieves up to 6.86× higher write throughput than UniKV-GC. These improvements stem from GPGPU-accelerated merge-sorting and parallel GC, which effectively reduce CPU bottlenecks and improve I/O efficiency. The throughput is reported by DB_Bench and are shown in Figures 9a and 9b under write and read workloads, respectively.

Evaluation 2: Space Amplification: gParaKV-GC achieves lower space amplification (SA) than baseline KV stores by reducing redundant storage through parallel GC and efficient data placement. For instance, with a 128 B value size, gParaKV-GC reduces SA by up to 58.44% compared to systems such as BlobDB and DiffKV-GC. This benefit is particularly evident for smaller values, where frequent GC operations can lead to excessive disk usage. The SA values are reported by DB_Bench with the fillrandom workloads and are summarized in Figure 12.

Evaluation 3: Write Latency: In the write latency evaluation, gParaKV significantly outperforms other KV stores in terms of P90 and P99 latencies. The experimental results indicate that gParaKV-GC reduces write latency compared to other KV stores, thanks to the efficient merge sorting and GC accelerated by GPGPU. Even though GC is triggered, gParaKV maintains low latency, thereby reducing computational overhead.

Evaluation 4: Compaction Throughput and Data Transfer Time: In this test, we analyze both the compaction throughput and data transfer cost. gParaKV-GC shows significantly higher compaction throughput across all workloads due to the use of GPGPU-based merge-sorting. With a 1 KB value, gParaKV achieves up to 12.09× improvement over BlobDB. Simultaneously, we monitor the data transfer time between SSD, CPU, and GPGPU memory. Although the transfer time ratio reaches 26.8% at 200 GB data volume, the overall system throughput remains stable. These results are presented in Figure 13 and Table 1.

Evaluation 5: Throughput and space amplification: In this test, we conducted extensive tests with value sizes uniformly distributed from 32 B to 16 KB. gParaKVGC maintains high throughput across all scenarios. It also performs GC efficiently, reducing space amplification under workloads with variable-sized values. These results are presented in Table 2.

Evaluation 6: Power Consumption and GPGPU Utilization:

We evaluate both the system's power efficiency and GPU utilization. gParaKV and gParaKV-GC demonstrate significantly lower power consumption across all value sizes. With a 1-KB value, gParaKV-GC reduces power usage by up to 60.28%, 82.99%, 47.19%, 23.80%, and 5.55% compared to BlobDB, UniKV-GC, DiffKV-GC, GDH-GC, and Wiskey-GC, respectively. This benefit stems from offloading compute-intensive merge sorting and GC tasks to the GPGPU, thereby reducing CPU overhead.

Figure 14 further illustrates that gParaKV and gParaKV-GC achieve the highest GPGPU utilization, effectively exploiting GPU parallelism for performance gains. Although gParaKV-GC consumes more GPU memory due to GC operations, its peak memory usage remains within 2 GB below the capacity of modern GPGPUs, thereby avoiding memory bottlenecks.

Evaluation 7: Performance Under Realistic Workloads: Under real-world workloads, gParaKV demonstrates superior performance across various workloads in the YCSB-C. gParaKV outperforms other KV stores across different value_size configurations, especially in mixed read-write and update workloads. With GPGPU's parallel computation, gParaKV-GC enhances throughput and reduces latency during compaction and GC. Compared with BlobDB, gParaKV achieves excellent performance under realistic workloads, particularly when processing large-scale datasets.

C.2 Computational Artifact A₂

Artifact Setup (incl. Inputs)

The hardware, software, dependencies, compilation process, and dataset configurations for A2 (gParaKV-GC) are identical to those in A1. These differences are listed as following: (i) the source code repository is <https://github.com/AHUKV/gParaKV-GC>, and (ii) when running experiments, the compiled binaries from gParaKV-GC/build/ are used, with the GC flag enabled (-gc=1).

Artifact Execution

All experiments for A2 correspond to the same evaluations as in A1 (Evaluations 1–7), producing the same figures and tables, see Figures 9–15 and Tables 1–2. The only modification needed to reproduce A2 results is to execute the corresponding scripts from A1 with the GC flag enabled:

Set `ENABLE_GC=1` in the provided benchmark scripts (e.g., `compaction_throughput_data_transfer.sh`, `space_amplification.sh`, `throughput.sh`, `write_latency.s.ycsbc.sh`).

It is not required to change parameters, paths, or configurations.

Artifact Analysis (incl. Outputs)

All experiments for A2 follow the same procedure as in A1 (Evaluations 1–7). The analysis of gParaKV-GC results is included in the Artifact Analysis section of A1, where the performance of both gParaKV and gParaKV-GC is discussed together. The GC-enabled results correspond to the GC in Figures 9–15 and the GC columns in Tables 1–2.