gLSM: Using GPGPU to Accelerate Compactions in LSM-tree-based Key-value Stores

HUI SUN, JINFENG XU, XIANGXIANG JIANG, and GUANZHONG CHEN,
Anhui University, China
YINLIANG YUE, Zhongguancun Laboratory, China
XIAO QIN, Auburn University, USA

Log-structured-merge tree or LSM-tree is a technological underpinning in key-value (KV) stores to support a wide range of performance-critical applications. By conducting data re-organization in the background by virtue of compaction operations, the KV stores have the potential to swiftly service write requests with sequential batched disk writes and read requests for KV items constantly sorted by the compaction. Compaction demands high I/O bandwidth and CPU speed to facilitate quality service to user read/write requests. With the emergence of high-speed SSDs, CPUs are increasingly becoming a performance bottleneck. To mitigate the bottleneck limiting the KV-store's performance and that of the applications supported by the store, we propose a system - *gLSM* - to leverage GPGPU to remarkably accelerate the compaction operations. gLSM fully utilizes the parallelism and computational capability inside GPGPUs to improve the compaction performance. We design a driver framework to parallelize compaction operations handled between a pair of CPU and GPGPU. We employ data independence and GPGPU-orient radix-sorting algorithm to concurrently conduct compaction. A key-value separation method is devised to slash the transfer of data volume from CPU-side memory to the GPGPU counterpart. The results reveal that gLSM improves the throughput and compaction bandwidth by up to a factor of 2.9 and 26.0, respectively, compared with the four state-of-the-art KV stores. gLSM also reduces the write latency by 73.3%. gLSM exhibits a performance improvement by up to 45% compared against its variant where there are no KV separation and collaboration sort modules.

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

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

ACM Reference format:
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, Article 5 (January 2024), 41 pages.
https://doi.org/10.1145/3633782

This work was supported in part by the National Natural Science Foundation of China under Grants 61702004 and 62072001. The work of Xiao Qin was supported by the U.S. National Science Foundation under Grants IIS-1618669 and CCF-0845257.
Authors' addresses: H. Sun, J. Xu, X. Jiang, and G. Chen, Anhui University, 111 jiulong RD, Hefei 230601, Anhui, China; e-mails: {sunhui, jfxu, xxjiang, gzchen}@ahu.edu.cn; Y. Yue, Zhongguancun Laboratory, Cuihu North Road 2, Beijing, Bejing, China; e-mail: yylhust@qq.com; X. Qin, Auburn University, Auburn; e-mail: xqin@auburn.edu.
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.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1553-3077/2024/01-ART5 $15.00
https://doi.org/10.1145/3633782

1 INTRODUCTION

The **log structured-merge tree** (**LSM-tree**) secures a champion's title as the mainstream data structure for key-value stores in large-scale storage systems. LSM-tree-based key-value stores - KV stores - are increasingly applied in a raft of write-intensive applications to boost system performance. The LSM-tree structure converts random write I/Os into sequential ones to optimize the random-write throughput. Thus, KV stores achieve high write performance at the expense of storing redundant data. A compaction procedure is triggered to perform garbage collection that recycles invalid data. This mechanism averts redundant keys in **sorted string tables** (**SSTables**) at the same level; meanwhile, high read performance is maintained. Although LSM-tree is popularly deployed in KV stores, some issues and breakthrough points exist in KV stores as follows. In applications, storage devices such as NVMe SSD, and PCIe SSD achieve rapid improvements, making CPUs rather than I/Os a performance bottleneck - a major issue centered around KV stores. A distributed storage system manages large-scale unstructured data that cost computing and storage resources in the cloud, where servers running high-and-full workloads in 7/24 hours hardly meet the requirement of applications due to overly loaded system resources: resources on these high-load servers are, more often than not, insufficient. A large amount of computing resources are consumed in the compaction of KV stores. The overall system performance inevitably degrades when the server has no required resources. A small value size can introduce a high computing overhead. Thus, mitigating the CPU bottleneck problem in KV stores motivates researchers to investigate approaches to reducing CPU overhead in compaction.

Existing studies primarily focus on CPU-assisted methods to optimize compaction. Pan et al. designed a virtual SSTable accompanied by a virtual merge mechanism to cut back frequent I/Os during compaction [26], but this scheme increases the number of tasks when compaction is kicked in, leading to a high resource cost. The pipelined compaction [45] is a pipelined merge mechanism, where the merge process contains multiple stages - read SSTable, merge-sort, and write SSTable. This mechanism improves CPU and I/O parallelism by parallelizing the three stages with multiple resources, thereby enhancing compaction performance and CPU and I/O utilization. Bandwidth usage, however, can impair performance in applications. A candidate solution is a new compaction mode that reduces the number of compaction operations. Still, redundant data recovery is migrated to a lower level, and the KV store conducts numerous unnecessary compaction operations under write-intensive workloads, leading to CPU burden as well as resource contention.

The compaction performance in KV stores is restrained by the contention of CPU-side computing resources. Employing many CPU cores to accelerate compaction raises the following issues. In the realm of client-server computing, it is desirable to keep a server running with a CPU utilization to minimize operational cost: gathering resources for real-time compaction becomes a challenge. In addition, the **non-uniform memory access** (**NUMA**) effect of multiple CPUs is serious, entailing a non-linear increment in marginal gains [10, 16]. Thus, it makes sense to offload compaction tasks to heterogeneous computational devices to avert resource contention and performance degradation on the host. In MatrixKV [42], level L_0 is placed in NVM-PEME instead of DRAM. Compaction at levels $L_0 \sim L_1$ is separated to weaken I/O resources competition. Xue et al. [33] proposed a FPGA-based compaction acceleration framework that combines hardware and software to improve compaction performance. Zhang et al. [44] configure a FPGA-based acceleration engine to conduct compaction tasks offloaded from a host, thereby alleviating resource content and improving throughput.

In this study, the following observations motivate us to offload compaction tasks to GPGPUs. The large-scale multi-process and multi-threading processing can aggravate the CPU overload for thread switching, adversely affecting CPU computing resources and system performance. To optimize compaction performance, the existing approaches are classified as CPU-based methods that

are dedicated to reducing compaction overhead, slashing the number of compaction operations, and delaying compaction. It is challenging to address the problem of performance degradation when the CPU overload is high with resource constraints. Figure 2(d) shows the CPU execution time (CPU cycles) required for a compaction task with 96 SSTables: this task consumes 14 seconds on a single-CPU platform and 1.8 seconds on a GPGPU. We confirm that the GPGPU compaction task can only accelerate compaction performance compared with the CPU when the number of SSTables in a single task is greater than 4 – meaning that the task processing time of the GPGPU is shorter than that of the CPU: the speedup ratio is greater than 1. In addition, GPGPU offloading can relieve resource contention and boost application throughput. Thanks to GPGPUs offering parallel computing resources, high-degree parallelism is likely to accelerate compaction. GPGPUs exhibit higher usability than FPGAs. Last, a small-value SSTable that contains numerous KV pairs induces a high compaction load, which burdens CPUs. Thus, GPGPUs with high computational capability are a solution to alleviate the aforementioned CPU bottleneck.

We propose gLSM - a GPGPU-empowered collaborative compaction accelerator catering for KV stores. gLSM, a highly parallel-compaction system running in a GPGPU, encompasses data transfer management, parallel coding and decoding, GPGPU-orient sorting and deduplication, and task collaboration. We implement a run-time driver to facilitate collaborations between CPU and GPGPU. A controller for parallel coding and decoding resides in the GPGPU. CPU-side task offloading and GPGPU-side compaction are performed in parallel. gLSM aims at revamping the compaction performance and divert CPU resource contention away from the host.

Our contribution can be tabulated as follows:

- ◇ We offer evidence demonstrating that a high proportion of small values in KV pairs is the primary culprit to compaction performance: multi-threaded compaction mechanisms lead to resource contention as well as degraded KV store performance.
- ◇ We propose gLSM - a parallel compaction system on a heterogeneous GPGPU for KV stores. gLSM, accelerating compaction operation, embraces a collaborative CPU-GPGPU driver catering for task offloading and parallel compaction.
- ◇ We implement the gLSM system on a popular GPGPU P100. In empirical studies, we compare gLSM against the well-known KV stores in terms of throughput, compaction bandwidth, write average latency, and CPU-GPGPU usage. The results that our gLSM enhances the throughput of the alternatives by up to a factor of 2.46.

The rest of article is organized as follows. In Section 2, we introduce the background knowledge. In Section 3, we show the related work. In Section 4, we show the motivation. In Section 5, we describe the proposed method. In Section 6, we conduct a series of experiments to evaluate the performance of the proposed method. In Section 7, we conclude our work.

The rest of the article is organized as follows. In Section 2, we introduce the background and the related work. Section 3 presents the motivation. In Section 4, we describe the proposed method. Section 5 elaborates on the design of gLSM, and Section 6 is dedicated to extensive experiments to evaluate the performance of the proposed method. In Section 7, we conclude our work.

2 BACKGROUND AND RELATED WORK

We elaborate on the background and shed light on the existing KV stores that inspire us to propose gLSM.

2.1 Background

LSM-tree is a hierarchical and ordered data structure catering for disks. LSM tree-based KV stores are applied in workloads with large-scale write and retrieve operations. Taking LevelDB [7] as an

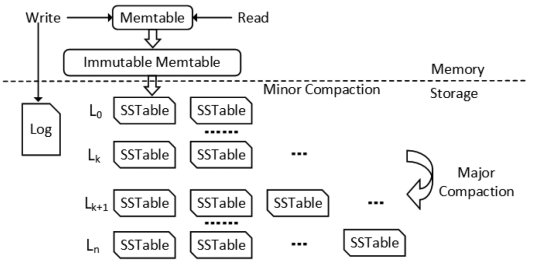


Fig. 1. Structure of LevelDB.

example, we present the structure of KV stores in Figure 1. Symbols L_0, L_k, and L_{k+1} represent level *0*, level *k*, and level *k+1* ($k \geq 1$) in LevelDB, respectively, in this article. In LevelDB, one component (Memtable) resides in the main memory, and the others are stored on a disk. Upon the arrival of a write request, a KV pair is initially written into MemTable. Once data volume in a MemTable reaches the threshold, MemTable is converted to immutable, which is flushed to a disk as a Sorted-String Table or SSTable. This process is named minor compaction. Afterward, a new MemTable is created for service requests. A new SSTable is placed at level L_0 where SSTables exhibit overlapping ranges. At the other levels, the key ranges of SSTables are non-overlapping. At level L_0, according to the key ranges of compacted SSTables, a thread searches the key range in the rest of the SSTables to elect an SSTable with overlapping key ranges. The overlapped-key-range SSTables are performed compaction at level L_0. At level L_k (k>1), as the number of SSTables at level L_k exceeds a threshold, an SSTable at level L_k and one or more SSTables at level L_{k+1} that have overlapped key ranges with the SSTable are selected to undertake merge-sorting for KV pairs in these SSTables. Then, new SSTables are produced and written at the level L_{k+1}; meanwhile, old SSTables are marked obsolete and deleted from the disk. This procedure is referred to as major compaction.

In compaction, a high write latency for foreground applications occurs and data more than what users require are written to the device (*a.k.a.* write amplification - WA), which impairs the overall performance.

2.2 Related Work

We introduce existing KV store optimization techniques, such as LSM-tree structure and compaction, collaborative task offloading, and hardware acceleration.

Optimization of compaction. Zhang et al. designed a pipelined compaction - PCP - for the LSM-tree. PebblesDB [27] introduces a fragmented log-structured merge tree and guards for new data organization, reducing write amplification. In LSM-trie [38], the hierarchical structure is divided into a prefix tree to optimize the linear-growth pattern between two levels, and write amplification is reduced. The trie structure conserves the footprint of index entries. Wisckey [19], which involves keys in compaction and changes the value offset, reduces compaction I/Os. dCompaction [26] delays some compacted SSTables to a later compaction, suppressing compaction frequency. Existing studies improve KV store performance under the I/O-constraint or CPU-constraint cases.

LSM-tree Structure. LSM-trie [38] - an extension of LSM-tree - deals with small-value KV pairs through metadata management. Then, the footprint of large-scale metadata is shrunk and the performance cannot be degraded under workloads with small values. PebblesDB [27] incorporates a fragmented log-structured merge tree and guards to organize SSTables, decreasing WA. WipDB [46] writes KV items immediately in an approximately sorted list instead of incrementally sorting

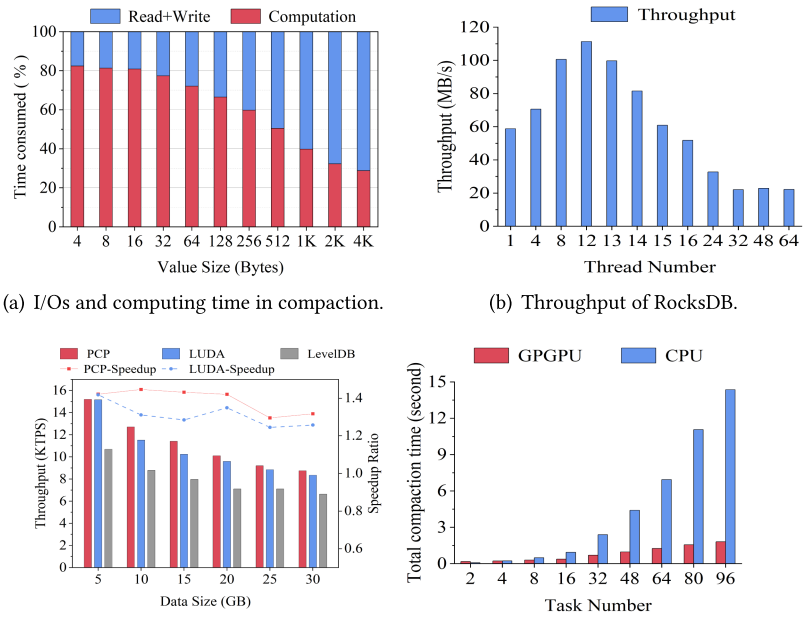
KV pairs. SkipStore [43] merges each KV at level L_0 to a low level - avoiding the merge process with intermediate levels. FloDB [3] manages in-memory components via a two-level structure. A small concurrent hash table at an upper level supports fast writes. The large skip-list at the low level conducts range queries. SifrDB [20] is built on a split MS-forest structure to decrease WA and minimize compaction footprint.

PCP [45] advocates for a pipelined compaction to promote parallel I/Os and sorts through device-side parallelism. dCompaction [26] applies virtual compaction based on the metadata of physically compacted SSTables. A physical compaction is conducted as the number of compacted SSTables exceeds a threshold. Wisckey [19], HashKV [5], FenceKV [34], ALDC [4], and VT-Tree [28] deploy a key-value separation mechanism to involve keys in compaction to reduce the compaction cost of values. LWC [41] employs a **lightweight compaction tree** (**LWC-tree**) that appends data to the SSTable and merges small-size metadata and data in the SSTable, which revamps WA measures. TRIAD [2] improves the KV store performance under skewed workloads where hot keys are likely to be short-lived. TRAID stores hot keys in memTable and the write-ahead log rather than SSTables at level 0.

These existing studies can incur high footprint usage and management cost. The delay- and tire-based compaction mechanism triggers a large number of compaction tasks in a centralized mode, leading to resource occupation and real-time competition.

Task offloading. KVSSD [37] combines LSM-tree with a **flash translation layer** (**FTL**) inside an SSD. Compaction has no copy operations, achieving efficient garbage collection. A host and near-data processing-enabled storage device collaboratively conduct compaction tasks being offloaded to devices [30–32], improving the KV store performance. A cognitive SSD [18] is configured with an acceleration processor to provide deep-learning inference and graph search. PinK [12] is a KV-SSD based on the LSM-tree. Jin et al. [13] introduce the key-addressable multi-log SSD with a KV interface, advocating for variable-length records to store data. Wang et al. [36] propose a customized SSD that exposes its internal flash channels to applications combined with a KV store. FPGA enhances performance by favoring configurable computational resources to conduct compaction [44]. Xue et al. [33] designed an FPGA-based pipeline to separate key and value, thereby accelerating the KV store performance. Computational devices undertake offloading tasks to enhance resource utilization. However, such devices cannot fulfill performance requirements under intensive-compaction workloads owing to limited resources. FPGAs and GPUs are two different-structure heterogeneous devices. FPGAs have the advantage of short internal latency, simple and fast circuits. The disadvantage, however, lies in the complex programming model and procedure. To facilitate any changing requirements, one needs downtime to burn in the program. Unlike GPUs, FPGAs are too proprietary to run general programming. GPUs address these issues – and GPUs' high parallel resources furnish a large number of parallel computing tasks (e.g., intensive compaction offloading from the hosts). Moreover, the programming model of GPUs is known for its simple and general-purpose design.

Hardware acceleration. MatrixKV [42] is deployed in DRAM, NVM-PMEM, and SSD. Level L_0 is placed in NVM-PEME, and compaction in L_0 - L_1 is at other levels to alleviate I/Os competition. NoveLSM [14] extends a component on NVMs to store writes as memory components are full - averting write stalls. A concurrency control algorithm was designed to organize components in LSM-tree into a concurrent linked list to reduce the stall and improve multi-core KV store performance [6]. FD-tree [17] is built on SSDs to decrease random-write I/Os, thereby raising read performance in a cascading manner. A hybrid index and a B+-tree index [39] are constructed in NVM and DRAM to boost the performance of update and range scan operations. A FPGA-based acceleration engine for KV stores [11, 33] were designed to speed up the compaction via hardware-software collaboration - improving throughput and averting resource contention.



(a) I/Os and computing time in compaction.

(b) Throughput of RocksDB.

(c) Throughput and speedup ratio.

(d) Execution time.

Fig. 2. Compaction performance affected by KV pairs size (a), multiply threads (b), data volume (c), and CPU time overhead (d). We employed i7-10700 CPU for (a) and (b), and E5-2630 v4 and NVIDIA TESLA P100 GPGPU are used for (c) and (d).

Modern hardware architectures eliminate I/O limitations thanks to high-performance interfaces. Thus, the performance bottleneck of KV stores lies in CPUs instead of I/Os.

GPGPU-related work. Evidence shows that GPGPUs strengthens the performance of KV stores. Tayler et al. [9] evaluated the performance of memory KV stores on a GPGPU. GDM [35], managing device-side memory with both block- and object-level information, employs three heuristics to offer overwrite elimination and double-transfer avoidance. GPUfs [29], providing a POSIX API for GPGPU programming, *exploits parallelism and extends a buffer cache to optimize file accesses.* LUDA [40] uses GPGPU kernels to accelerate compaction performance. LUDA [40] uses parallelism resources inside GPGPUs to improve compaction performance. Although GPGPUs are used to accelerate performance, there is a lack of GPGPU-empowered KV stores. Computing bottleneck impairs compaction performance: we argue that a high-performance GPGPU acceleration engine in KV stores is a potential solution.

3 MOTIVATION

In this section, we present the following challenges amid improving the performance of KV stores. Then, we elaborate on the motivations behind the development of the GPGPU-enabled KV store. In this section, the testbed is configured with Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, and the results are presented in Figures 2(a), (b), and (c). In Figure 2(d), we use a server with a GPGPU, the feature of which is outlined in Section Section 6.1.

Problem 1 (Under workloads with small-size values, a computation procedure costs a large amount of time.). *Notably, a compaction procedure comprises of multiple stages, i.e., read, write, decoding, merging, and encoding. We define the stages of decoding, merging, and encoding as*

computation. Figure 2(a) shows the percentages of execution time of read, write, and computation under workloads with 8-B keys. The percentage of computation time increases as the value size decreases. When the value size is 4 KB, the computation time becomes 30%. When the value size is smaller than 512 B, the computation time is surpassing 50%. The percentage of computation time rises to 80% at value sizes smaller than 16 B. In case of value sizes less than 512 B, computation supplants I/O as a performance bottleneck. A previous work [1] demonstrates that 90% of Memcached KV pools in Facebook store KV items, whose values are smaller than 500 bytes. Thus, it is crucial to cope with the small value issue - a computing bottleneck in compaction.

High CPU utilization benefits cloud service providers thanks to profitable revenues made by high utilization. In KV store applications, however, a high load on servers can cost computing resources in such a way that the rest cannot meet computing requirements: user requests are delayed with degraded quality of services. Thus, deploying a heterogeneous collaborative system is paramount to reshape the compaction performance of KV stores and the user experience. Solutions that facilitate heterogeneous computing include multiple CPUs, FPGAs, and GPGPUs. The usability of FPGAs is inferior to GPGPUs, and multiple CPUs have an NUMA effect [10, 16].

We investigate the NUMA effect on a server, see Section 6.1 for details. Our tested platform has two NUMA nodes (node0 and node1) and two CPUs (CPU0 and CPU1). Each node includes a CPU with 20 cores, i.e., node0 - CPU0 and node1 - CPU1. CPU0 and CPU1 are associated with 16-GB memory DRAM0 and DRAM1,[1] respectively. In this study, we use *numactl* and *dd* commands[2] to study the performance of systems with the effect of NUMA tunings in terms of the write/read throughput. When both CPU and memory are bound to the same node, the write throughput is the highest at 1945.6 MB/s on node0 and 1843.2 MB/s on node1, respectively. When the CPU and memory are bound to different nodes, the write throughput is degraded by up to 22% compared with binding the same node. Similarly, when we force execution on a node with memory allocation forced on another one, the read performance is reduced by about 11.2% compared with binding the same node. The NUMA effect can cause some issues. Multiple CPU cores are organized into a unified addressing access structure. The distance between each CPU core and the memory might be different. The latency of cores-node accessing memory from different CPUs becomes large as the access distance is long – expanding the execution time of applications on the CPU, and leading to performance degradation, especially for write operations. The approach to deploying a large number of CPUs can further degrade performance rather than deliver a linear increment in performance.

Hence, we propose a GPGPU-accelerated KV store (gLSM) – a collaborative system – to revamp compaction performance. In Section 6.7, we validate the scalability of gLSM.

Problem 2 (Performance optimization offered by multiple threads degrades under heavy workloads). *Multi-threaded compaction (e.g., RocksDB) is a viable method for optimizing the performance of KV stores. In this process, many threads are applied to compaction, enabling concurrent multiple compaction tasks. However, increasing the number of threads escalates the CPU usage, resulting in frequent context switches - a culprit for degraded performance. Notably, CPU usage is*

[1]The CPU interconnect is: CPU0 - DRAM0 on node 0 and CPU1 - DRAM1 on node1

[2]For example, we use command *numactl –cpubind=0 –membind=0 dd if=/dev/zero of=/dev/shm/A bs=1M count=1024* in the test. It means that we force to execute *dd* command to study the performance of writing a 1-GB tmpfs file with 1-MB block size on node0 with memory allocation forced on node0. Afterward, we record the write throughput. Notably, the values of parameter "cpubind" (or membind) are "0" and "1". When cpubind and membind have the same value, it means forcing execution on the node with memory allocation forced on the same one (e.g., cpubind=0 and membind=0, cpubind=1 and membind=0). Otherwise, configuration (cpubind=0 and membind=1) and (cpubind=1 and membind=0) means forcing execution on one node with memory allocation forced on another node.

gauged as a percentage of CPU cycles used to complete its compaction tasks. The CPU usage is an essential metric because high CPU usage indicates that the compaction procedure requires high CPU processing power while adversely affecting the performance of other applications. Figure 2(b) compares the performance of KV stores in terms of different numbers of threads. The parameters in this experiment are configured as 16-B key and 1-KB value. A thread handles 100,000 KV pairs. Although throughput increases up to a thread count of 12, performance significantly drops beyond 14. Under the competition for resources in this environment, the multi-threading method fail to enhance KV store performance, and it is prudent to balance the benefits and costs of multi-threading in KV stores. The throughput of multi-thread numbers in RocksDB under different value sizes: there is a similar performance trend for RocksDB under workloads with different sized values. Facing a small number of threads, the throughput is enhanced with an increasing number of multiple threads. Afterward, the number of threads reaches a threshold, and the system performance shows signs of downgrading – declined throughput – due to the CPU context switch overhead. Since the performance trend has little dependency on key and value sizes, we primarily focus on the system performance when the number of threads increases (see Figure 2(b)), the data size expands (see Figure 2(c)), and the number of tasks goes up (see Figure 2(d)). We neglect the impact of special key and value sizes on the performance in this test.

In GPGPU-based collaborative systems, high-priority applications (e.g., graph computation, real-time video analysis) overrun computational resources in a cloud server. When resources competition in KV stores becomes intense, it makes sense to accelerate the compaction in KV stores using collaborative GPGPUs. When CPUs are kept busy by application requests, the response time of storage operations in the background enlarges because on-demand resources cannot meet the compaction procedure's requirements: it is desirable to offload computational operations in the storage to a heterogeneous computing device like GPGPU and FPGA [33, 44]. We advocate for GPGPU to collaboratively accelerate KV stores. Our experimental results validate that GPGPU collaborative computing is beneficial to applications under high CPU loads. Notably, Figure 2(b) suggests that increasing the number of threads on a CPU sparks resource contention leading to degraded throughput. We may raise the number of threads with an increasing number of CPUs; however, the NUMA effect diminishes the gains offered by CPU cores. Besides, it is more expensive to use CPUs than the GPGPU to achieve the same number of threads.[3] Xilinx Virtex UltraScale+ VU9P board (4,500$) [44] and Xilinx Kintex UltraScale KCU1500 card (2,000$) [33] was employed to accelerate performance in a system.[4]

Problem 3 (The throughput speedup ratio decreases when using large-scale data). *Existing methods improve KV store performance under small-size-data workloads; however, performance decreases as the data volume increases. Figure 2(c) shows that the throughput (KTPS or thousands of transactions per second) and speedup ratios ofKV stores decrease as the data volume goes up. In this article, LUDA-Speedup and PCP-Speedup to represent the performance speedup ratio of LUDA and PCP compared with LevelDB. In the CPU, KV store performance deteriorates as the number of threads increases due to the overhead of creating and maintaining threads. Furthermore, when the number of threads is beyond twice that of cores in the CPU, context-switching further weakens performance. In addition, CPU-side intensive compaction reduces KV store performance.*

GPGPUs, by contrast, prefers computing-intensive and large-scale parallel tasks to mitigate this issue. We apply a CPU-GPGPU heterogeneous framework to boost compaction performance. A compaction procedure is divided into multiple stages, some of which are floated to the GPGPU

[3]The price of NVIDIA Tesla P100 16 GB: https://www.amazon.com
[4]Xilinx FPGA products: https://www.xilinx.com

to lessen multi-threads overhead in the CPU while accelerating KV performance via the GPGPU parallelism. In Figure 2(c), the speedup ratio of LUDA decreases when large-scale data are processed. The merge-sort scheme mostly constrains these CPU-side approaches to revamping the KV store performance. Although the parallelism mechanism is a popular method for boosting the merge-sort performance on the CPU, parallelism resources are limited to the number of threads in the CPU. This issue motivates us to design a parallel sort algorithm running in GPGPUs, thereby improving merge-sort performance.

4 OUR SOLUTION

How can GPGPUs be used to improve compaction performance in KV stores? Deploying a GPGPU, multiple concurrent KV store stages are aimed at improving compaction performance. The conventional KV store compaction procedure is divided into the following steps.

Step 1. Read SSTables involved in compaction to host-side memory from the disk.

Step 2. Parse the SSTables to obtain the required keys from a data block.

Step 3. Perform merge-sort on keys.

Step 4. Encode the new data block that is assembled into an SSTable.

Step 5. Write the new SSTable into an underlying disk.

When there are N SSTables involved in a compaction procedure, the total time under CPU processes (T_{cpu}) is

$$T_{cpu} = \sum_{i=1}^{5} \sum_{j=1}^{N} (T_{ij}), \tag{1}$$

where T_{ij} denotes the execution time of step i^{th} sub-procedure of SSTable j^{th}. Steps 2 and 4 are serially executed on the CPU; therefore, the two steps are not parallelizable. This serial computing is expected because, on the CPU side, the merge-sorting is used for the decoding algorithm: steps 2, 3, and 4 are serially executed – and they are not independent sub-processes. In the merge-sorting operation, we obtain the minimum key in each data block during step 2 (decoding). Furthermore, we elect the minimum key in these keys from all data blocks (step 3). We encode these keys and corresponding values into new data blocks (step 4). The above steps are performed cyclically to originate multiple data blocks, which are assembled into a new SSTable. We conclude that on the CPU side, steps 2 and 3 are executed serially instead of in parallel.

For a GPGPU, steps 2 and 4 can be parallelized; then, the total time for all steps is

$$T_{2max} = \max\{T_{21}, T_{22}, \ldots, T_{2N}\}, \tag{2}$$

and

$$T_{4max} = \max\{T_{41}, T_{42}, \ldots, T_{4N}\}. \tag{3}$$

where T_{2max} and T_{4max} are the maximum execution time in steps 2 and 4, respectively. Then, we have the execution time of GPGPU (T_{GPGPU})

$$T_{GPGPU} = \sum_{j=1}^{N} (T_{1j} + T_{5j)} + T_{2max} + T_{merge-sort} + T_{4max}. \tag{4}$$

T_{GPGPU} is smaller than T_{cpu}. Cloud servers are subject to heavy workloads and large-scale data volumes. The optimization methods achieving high performance under this production environment build up CPU burden, degrading performance under heavy workloads. GPGPUs easily facilitate collaborative compaction owing to abundant computing resources and scalability. A handful of studies use GPGPUs to accelerate KV store performance. Although GPUs outperform FPGAs

in terms of availability and cost, GPGPU-based heterogeneous systems are subject to two challenges: (1) the adverse impact of transmission latency between a device and its host; (2) whether the execution speed of a GPGPU is faster than that of its CPU host.

Figure 2(d) reveals the execution time on a GPGPU and the total execution time on a CPU. As the number of tasks surpasses eight, the execution time of the GPGPU falls below that of the CPU. Due to the limited number of threads on the CPU, there is a roadblock to allocating the above large number of serial operations to threads – and serial operations are periodically performed. GPUs have large-scale threads, and the number of threads on a CPU is twice the number of cores. Then, the GPGPU has the advantage of performing compaction. In this study, the data transmission costs associated with GPGPU task offloading is within 2% of the compaction procedure under workload with 16-B key, 128-B value, and 10-GB data volume. The computational capability of CPU threads is higher than that of GPGPU. We primarily focus on the number of compaction tasks – referred to as a threshold – that trigger task offloading from the CPU to GPGPU. In case offloading compaction tasks to GPGPU boosts system performance, the GPGPU is an edge over the CPU from the perspective of compaction task offloading. The massive parallelism offered by the GPGPU is fully utilized to advance compaction in KV stores. This performance improvement becomes apparent as the number of tasks increases. In gLSM, collaborative drivers execute tasks asynchronously and synchronize execution results. The asynchronous transfer module reduces the latency of data movement between the CPU and GPGPU. To carry out this approach, we design a parallel encoder-decoder coupled with a collaborative-sorting module for the GPGPU. We also implement a key-value separation module on the GPGPU to reduce data movement.

How does our proposed gLSM outperform LUDA in terms of compaction performance? LUDA - a GPGPU-enabled KV store - overlooks the implementation of a full-procedure compaction task on the GPGPU: LUDA performs merge-sort operations on a CPU. As a result, the GPGPU's computational capability cannot be entirely utilized, and LUDA's performance is not on par with our gLSM. Notably, the CPU-oriented merge-sort algorithm relies on CPU threads to perform k-way merge operations, leading to a high cost of parallel resources. However, this merge-sort scheme cannot thoroughly utilize parallel resources on the GPGPU because k-way merge-sorting decreases sort parallelism, which, unlike the CPU, is unacceptable at the GPGPU.

In addition, the CPU-based time complexity of merge-sort in LevelDB is ***O(n)***. We propose a radix sort for a GPGPU to decrease the time complexity of the merge-sort process. The radix sort algorithm is selected because CPU-oriented k-way merge-sorting cannot distribute all sort tasks evenly to all kernels on the GPGPU. In a k-way merge-sort operation, it is necessary to choose the smallest data piece from all ways of sorting the overall body of data, which is compared with the minimum-sized data. Similarly, all data are sorted in k ways. To evenly distribute sorting tasks across kernels in the GPGPU, the k used in the *k-way* process must be set as a large value. In this case, the amount of data in each way is largely reduced and the merge-sort operation is converted from a parallel algorithm to a near-serial one. This approach slashes the utilization of GPGPU resources, impairing parallel performance. Thus, we employ radix sorting as a GPGPU-oriented sorting algorithm in gLSM. In a KV store, keys are easily split into several groups according to the number of bits. The key range in each group is relatively small and the total number of keys is large, indicating that it is beneficial to kick in the radix-sort algorithm to sort data on the GPGPU to achieve high parallelism [8, 22, 23].

We compare, in Figure 3, our gLSM against LUDA, which writes buffer twice to obtain a key - the associated data copy and movement operations are time-consuming. Thanks to the high computational capability of the GPGPU-oriented radix sorting algorithm on a GPGPU, gLSM shortens sorting time compared to that in LUDA.

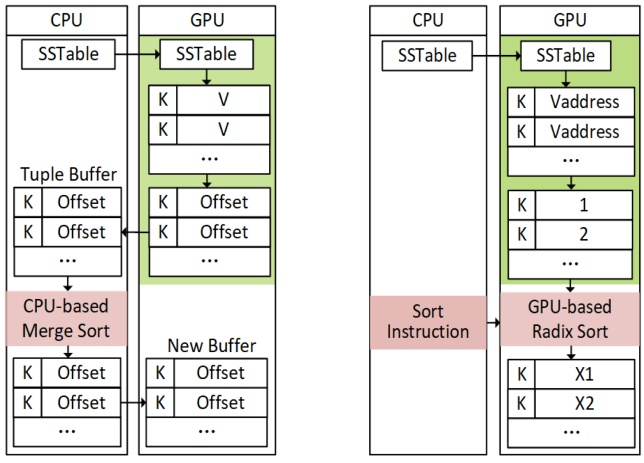


Fig. 3. Comparison of LUDA and gLSM sorting algorithms. Notably, the sort operation cannot be proactively performed on the GPGPU; therefore, the CPU side issues the *'sorting instruction'* to the collaborative-sort module on the GPGPU side and trigger the sort operation.

We articulated the discrepancy between gLSM, LUDA, and FPGA. FPGA implements the process of pipelined compaction tasks on the CPU to the hardware, and the compaction procedure fully complies with the CPU-side compaction mode rules. LUDA, implemented on the GPGPU, transfers sorting tasks to a CPU rather than running a completely independent and efficient GPGPU sorting algorithm. The percentage of computational resources consumed by sorting is remarkably high under workloads with small-sized values. gLSM, having the distinctions from the above solutions, renders a high-efficient GPGPU-based sorting algorithm accompanied by the GPGPU-side compaction implementation, thereby optimizing the sorting efficiency and compaction performance.

We face the following challenges amid the implementation of the compaction procedure on the GPGPU. (1) How to achieve efficient data copy and movement inside a GPGPU? There is a significant performance gap in the memory speed of GPUs: the global memory is large, but the memory speed is low. The on-chip memory, which is usually small, provides fast storage speed. Achieving high-efficiency data movement between different memory is a grand challenge. (2) How to change the compaction mode from serial to parallel? CPUs and GPUs belong to two different architectures, and a GPGPU embraces massive parallel resources. A second challenging goal is to design a high-efficiency parallel compaction mode on the GPGPU.

5 GPGPU-BASED GLSM

In this section, we present a system overview of gLSM.

5.1 Overview System

The overarching goal of this study is to devise a GPGPU-based storage engine named *gLSM* for KV stores, see Figure 4. Compaction tasks in gLSM can be offloaded to a GPGPU to ramp-up the performance of KV stores. Under heavy workloads, compaction procedures adversely affect the performance of multi-threaded PCP and RocksDB-based storage systems, and a massive amount of data slow down LevelDB performance. Deploying parallel decoding, merge sorting, and encoding modules, gLSM fully implements parallelism within the GPGPU to improve compaction performance under heavy workloads by offloading compaction tasks to the GPGPU.

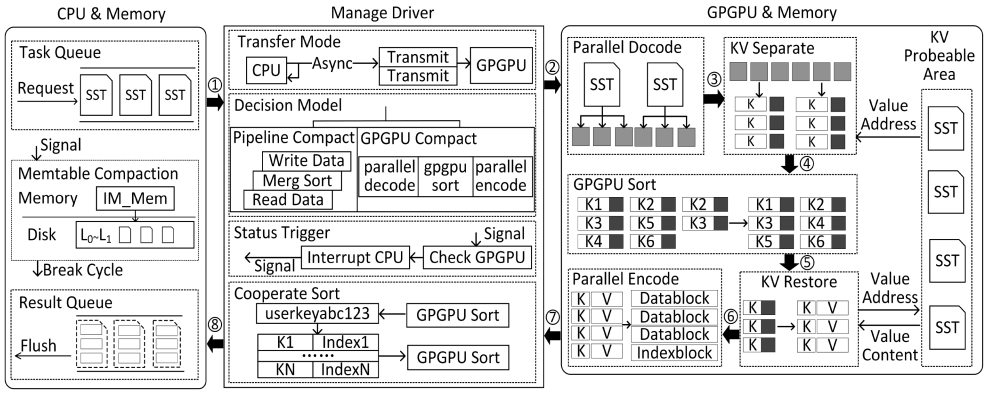


Fig. 4. Overall system of gLSM.

When the compaction process is triggered, gLSM reads SSTables into memory from a host-side storage device. These SSTables are transferred to the GPGPU memory (see ① in Figure 4 and Section 5.3). If the decision module in the management driver determines that the number of compacted SSTables is smaller than the threshold of compaction-task offloading on the GPGPU, the compaction is conducted on the CPU. Otherwise, if the compaction is performed at level 0, gLSM will invoke the GPGPU compaction interface to offload the compaction task to the GPGPU (②). The manager driver allocates tasks to the GPGPU kernels. The CPU and GPGPU parallel compaction process is outlined in Section 5.4 and ③. In case of the GPGPU-enabled compaction, the key-value separation module merely performs compaction on keys, avoiding values copy and movement in the low-performance global memory (see ④ and Section 5.5.3. The sort operations cannot be proactively performed on the GPGPU, and the CPU should first trigger the sort operation. Then, a collaborative-sort module executes the radix sort instead of merge-sort in the GPGPU during compaction (see ⑤ and Section 5.5). When the GPGPU completes the sort operation, KV pairs are assembled through the key-value separation module (see ⑥ and Section 5.5.3). Blocks are generated through parallel coding and decoding (⑦). Upon the finalization of all the GPGPU tasks, a completion message is delivered to the CPU-side management driver. ⑧ in Figure 4 presents the write-back procedure of data block and metadata management handled by the CPU (Section 5.4).

5.2 Data Preprocessing

To process compacted SSTables, the GPGPU is obliged to handle the configuration files of the SSTables (e.g., SSTable and block sizes), which are stored in the CPU-side memory. In gLSM, data ought to be shared in independent memory located in the GPGPU and CPU, which do not share the same memory space. This data isolation makes the GPGPU unable to allocate resources for individual tasks according to entries in the configuration file. To overcome the issue, gLSM preprocesses data and initializes GPGPU memory prior to compaction. In compaction, configuration data are sent to the GPGPU from the CPU. When a collaborative compaction procedure begins, the CPU reads configuration entries from the parameters into the memory. The configuration information is then sent to the constant memory on the GPGPU, which calculates the number of subtasks for each SSTable according to the block and SSTable sizes. gLSM allocates kernels and organizes thread grids to handle these tasks.

In the decoding stage, these footer, index, and datablock decoding are conducted in parallel. The computing resource allocation for the three sub-stages depends on the parallel granularity. Each

thread block in the *indexblock* and *datablock* decoding stages comprises 32 threads and multiple thread blocks form a thread grid, which is used to parse the two blocks of an SSTable. The data accessing memory is automatically aligned in memory, reducing access I/Os arising from data misalignment. Such a strategy is in line with the single instruction stream & multiple data stream (SIMD) in the GPGPU.

In each thread block, the maximum number of threads is denoted as $MaxThreadNumber$ and the number of stream processors is represented as $SMNumber$. Then, we have

$$P_{task} = \frac{MaxThreadNumber}{SMNumber}, \tag{5}$$

where P_{task} represents the maximum number of threads on a stream processor. Because one thread processes an encoding task, the maximum number of encoding tasks allocated on a stream processor cannot exceed P_{task}. Otherwise, the stream processor is sparked more than once, enlarging encoding time and slowing down encoding performance. It is necessary to designate computing resources in a kernel and configure a suitable thread grid size on GPGPU. In the encoding procedure in our test, each thread block comprises 64 threads, which can handle 64 encoding tasks in parallel – and multiple thread blocks also concurrently encode the datablock and indexblock.

A GPGPU expects, in data pre-processing, some constant memory to store configuration entries. Thus, the memory allocation module in the driver allocates a portion of constant memory for a configuration file to keep parameters. This memory has the same lifetime as that of the entire system. Receiving parameters, the memory allocation module allots sufficient global memory to preserve the SSTable. The global memory size is gauged according to the configuration parameters. When the GPGPU takes in pre-processed data, the SSTables involved in compaction are read from the disk to the buffer in the CPU memory and then wait to be sent to the GPGPU. As the SSTables are available in page-separable memory, SSTables - being able to be swapped out of the memory - cannot be asynchronously sent to the GPGPU. Thus, gLSM assigns page-locked memory for the buffer. The data are then asynchronously transferred at the page size. Following memory allocation, the data preprocessing and space allocation for compaction are completed, and the SSTables move to the GPGPU global memory.

Upon the completion of data transmission, data preprocessing is invoked in the GPGPU to dynamically allot appropriate computing resources for different configurations, thereby processing tasks with different sizes. This preprocessing allows the GPGPU to flexibly accelerate compaction.

5.3 Asynchronous Transmission Module

The GPGPU processes a massive amount of data in compaction, resulting in simultaneous transfers of multiple SSTables - a time-consuming operation. These compacted SSTables are placed in the host-side memory, which the GPGPU cannot access. To perform compaction on the GPGPU, the SSTables are transferred to the GPGPU; then, compaction results are copied back to the host-side memory. It is desirable to design an efficient data transfer mechanism to reduce the transmission time spent in this process. Synchronous transfer mode is, more often than not, adopted to ship data between a host and a GPGPU. In this mode, one SSTable is transferred at a time. The host monitors the transfer status of each SSTable and; a copy operation finishes when the results return. The overall transfer procedure increases the latency of the host-side compaction. To implement bidirectional copying, it is vital to configure the transmission direction parameters, which can block the CPU data transfer process and handicap parallel operations between the host and GPGPU. To overcome this problem, gLSM employs an asynchronous transfer mode.

The CUDA stream supports an asynchronous transfer mode in which the next command is emitted by the CPU immediately after executing a transfer instruction. The host-side SSTables are

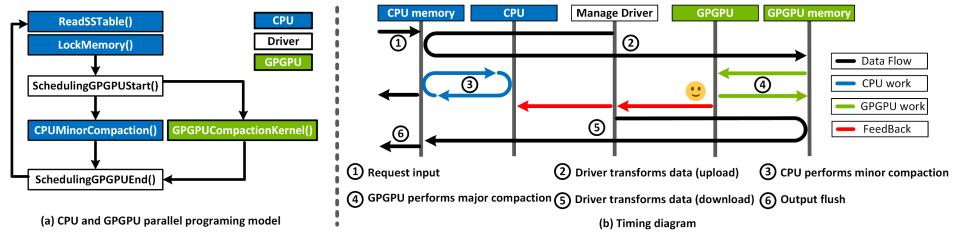


Fig. 5. Timeline for CPU-GPGPU parallel compaction. In Figure 5(a), the boxes with blue background represent execution procedures on the CPU side, whereas the boxes with the red background denote the procedures on the GPGPU component.

transferred to the GPGPU in this asynchronous stream. This mode primarily focuses on whether all SSTables are obtained by GPGPU rather than on whether each SSTable copy operation is completed. Notably, under the asynchronous transfer mode, the SSTables must be placed in the locked-page memory because data in swap memory might not be found when the data are copied. Multiple data transmission processes may be overlapped in the asynchronous mode, and SSTables can be transferred in parallel within the PCIe bandwidth. The maximum transmission time in this mode is equal to the time required to move one SSTable to the GPGPU.

5.4 Collaborative Compaction Mechanism in gLSM

The CPU-side compaction procedure encompasses minor compaction of the MemTable and major compaction of the SSTables. These two compactions are conducted in serial rather than in parallel. Offloading compacted SSTables to the GPGPU prevents minor compaction from impeding the entire process. We define a complete compaction procedure in LevelDB as a cycle. On the CPU, gLSM continually conducts minor compaction in a cycle without stalling to acquire data for major compaction, slashing write pauses. The GPGPU utilizes many threads to handle SSTables from the host. Each SSTable is disassembled into small blocks. By using many kernels, it is practical to process blocks at a fine granularity. The compaction efficiency can be exponentially increased because the compaction is undertaken in parallel between the host and GPGPU. In gLSM, the two-side compaction are concurrently evoked; however, one of the two-side compaction is enabled and the other is blocked in the conventional KV stores. Then, a collaborative mechanism is required to synchronize compaction tasks between the CPU and GPGPU, see Figure 5.

Figure 5(a) presents a basic framework of CPU-and-GPGPU parallel compaction. gLSM uses the GPGPU to independently perform the major compaction and uses the CPU to concurrently conduct minor compaction and the major compaction. When the compaction sparks, gLSM loads data and locks the memory, following which the driver performs data transfer and collaborative sorting, collects the task completion status on the GPGPU, and returns a result. Meanwhile, the CPU executes minor compaction as the driver kicks off. After the driver returns the feedback of GPGPU completion, the CPU deals with the subsequent write operations.

Figure 5(b) illustrates the workflow of the parallel process between the CPU and GPGPU. When a request arrives, the CPU first reads the SSTable involved in compaction into memory (①). Next, the driver delivers the SSTable to the GPGPU memory through an asynchronous transfer mode (②). After the SSTables are shipped to the GPGPU-side memory, the GPGPU launches major compaction in parallel (④). The CPU conducts the minor compaction immediately after initiating the driver (③), followed by awaiting the GPGPU-completion feedback from the driver. During this period, both the CPU and GPGPU independently handle compaction tasks (③ and ④). When the GPGPU-side tasks are accomplished, the driver collects the feedback from the GPGPU and returns it to the CPU (⑤). The CPU processes metadata and flushes data into the disk (⑥).

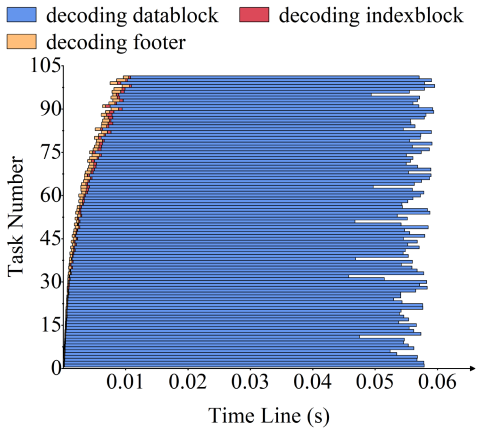


Fig. 6. Parallel decoding of real-time GPGPU series data.

According to Figure 5(b), we present the method to calculate the GPGPU utilization [21]. We first obtain the total time from creating a KV store to its end, see ①~⑥, represented as T_{sum}. We record the instantaneous GPGPU utilization in each 500 ms interval. We added all instantaneous utilization recorded in the total time to obtain a total utilization, denoted as U_{gpu}. However, the total time T_{sum} includes many GPGPU idle time when the CPU does not offload tasks but only read and write, see ①②⑤⑥. In this case, GPGPU does not receive the compaction tasks. Its instantaneous occupancy will be reduced to around zero. We remove the GPGPU idle time, and time T_{sum} is the sum of the time consumed by each compaction procedure of the GPGPU, see ④. If the GPGPU performed N compaction operations, we have $T_{sum} = \sum_{i=1}^{N} T_{i4}$. The average GPGPU utilization is given as $U_{avg} = U_{gpu}/T_{sum}$, which provides a method to study GPGPU resource utilization.

gLSM is adept at carrying out fine-granularity parallelism for compaction. The two procedures used in the process - parsing SSTables and repeatedly reading the data indexed according to the block and KV pair offset - can be parallelized. gLSM has three parallelism modes. First, by converting entries in an SSTable into records of a fixed size, gLSM eliminates dependency on adjacent records within the data. Entries are independently fetched - entailing data-level parallelism. Second, a task is triggered to parse a compacted SSTable, and these three small tasks (i.e., decoding footer, decoding indexblock, and decording datablock, see Figure 6 - being independent of one another for each SSTable - can be combined into a complete task. This task-level parallelism accelerates compaction.Third, the thread-level parallelism in the GPGPU provides simultaneous data reads according to a fixed offset: all active threads concurrently parse the data at the same time. The file content in GPGPU memory is treated as a long one-dimensional character array; each thread fetches content from this array with a start address and an offset. To perform these three tasks on a microsecond scale, a parallel pipeline of sequential tasks is taking in place. Each thread's task selects KV pairs from a one-dimensional buffer using an offset address and a record size. However, each task requires an initial period. The GPGPU immediately performs the tasks after each start command is emitted. The CPU subsequently sends the next task command, implementing a pipeline in which the start and execution tasks are separated. It is feasible to quickly convert the CPU-side sequential compaction operations into parallel compaction on the GPGPU by the virtue of an index with a uniform entry length to enhance write performance in the KV store.

The decision module is incorporated to achieve collaborative CPU-GPGPU sorting, as GPUs might be unavailable owing to driver updates. There is a necessity to check GPGPU availability by the decision module and to offload tasks according to the offload threshold. The use of a

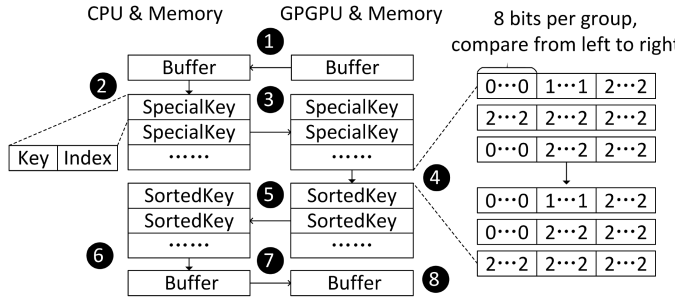


Fig. 7. Flow of collaborative-sort algorithm in gLSM. ❶ copies keys to the CPU. ❷ converts keys to special keys with pointer Vptr. ❸ copies keys to the GPGPU. ❹ GPGPU radix sort. ❺ copies keys to the CPU. ❻ restores keys. ❼ copies keys to the GPGPU. ❽ covers original buffer.

state trigger enables the CPU and the GPGPU to independently fulfill offloading tasks. The trigger module initiates communication between the two sides. Together, the decision and trigger modules render collaborative compaction in gLSM.

5.5 GPGPU-aware Radix Sort Algorithm in gLSM

5.5.1 Collaborative Sort Agent. GPGPUs have no byte stream syntax libraries to warrant compaction in KV stores. Per the KV data structure and the GPGPU computing capability, we define a data structure including key information, integer converted from the byte-size keys, and index information of the value address buffer, CPU-side byte-size keys, and long-integer keys in the GPGPU.

As shown in Figure 7, the collaborative-sort module conducts transfer in the compaction and conversion procedures. The byte stream conversion process depends on the GPGPU. As the GPGPU has no run-time system to process data, the key buffer is sent after each key-value separation to the sort agent module in the CPU. To do this, a multi-threaded to partition byte stream to restore each key is put forth. The subscript of each key, referred to as the index of the value address buffer, is appended to the data structure to be sorted, facilitating a search for the value address and KV pairs. Then, a tuple (Key, Vsubscript) is obtained from the key stream buffer. If the value is small, there will be a number of keys included per compaction. As the conversion process is time-consuming, we define values of less than 128 B as small values for which multi-threading for parallel conversion are initiated. When all keys are independently restored, the sorting agent automatically hands over the keys in the CPU-side address space to the GPGPU-side memory. Using a hardware channel, a rule governs the conversion of a key stream structure to a SpecialKey, in which a bit-size operation converts the key to an unsigned 64-bit long integer every eight bytes. This rule is accompanied by the radix sort rule to form a comparison rule. In gLSM, the performance of the radix sort algorithm anchored on GPGPU is efficient [8, 23]. Given the GPGPU-side radix sort algorithm, we divide a key into multiple buckets, each of which encompasses eight bytes. We convert the 8-B byte stream into a 64-bit unsigned integer. Then, we compare the size of integers in this bucket from left to right (see ❹ in Figure 7). When the comparison completes in the position order, all keys are sorted in the dictionary order. We implement the GPGPU radix sort algorithm for byte streams on the GPGPU side, and the radix sort algorithm accelerates the compaction performance more substantially than the CPU counterpart.

After the conversion is achieved, the collaborative-sort module invokes compaction, and the GPGPU executes sort and de-duplication operations. SpecialKey is converted to the original byte-stream key when the sort agent completes compaction. The CPU stores the sorted key into

the SortKey buffer and records the sorted Vsubscript. This information, along with the SortedKey buffer, is sent to the GPGPU-side buffer and value address index buffer, which may overwrite the original buffer. Meanwhile, the number of key de-duplications is registered for the subsequent parallel encoder, and the collaborative-sort process completes. Except for the subscript statistics and byte stream partitioning into keys, the GPGPU multi-threads discharge merge-sort operations for keys in compaction. gLSM adopts the radix sort to realize large-scale parallelism even though the ordering in SSTables is discarded, thereby boosting system performance.

Discussion. In the radix sort used in gLSM for GPGPU, we pad all variable-sized keys to the largest key, which must be divisible by 8 B. This procedure that we convert every 8-B key to an unsigned 64-bit integer reduces the number of comparisons in the sort and maximize the utilization of GPGPU parallel resources.

Why is the key size multiple of 8 B? In the radix sorting algorithm used in gLSM for GPGPU, we pad all variable-sized keys to the largest key, which must be divisible by 8 B. An essential technique of the algorithm is to convert every 8-Bytes (8-B) kye into an unsigned 64-bit integer (i.e., $uint64\_t$). This method can enhance sorting efficiency by reducing the number of sorting operations. Notably, $uint64\_t$, a data type in C++ that can represent numbers from 0 to $2^{64} - 1$, can represent a maximum positive integer $2^{64} - 1$. The number of bits in this data type corresponds to the number of bits in an 8-B key. Thus, we can convert 8-B keys to $uint64\_t$. In this case, for each 8-B key, we only compare it once on the GPGPU as opposed of 8 times on the CPU. This approach reduces the number of comparisons. However, this method has its limitation: if the key size does not satisfy an integer multiple of 8 B, it may fail to convert the key to an unsigned 64-bit integer when converting the last part of the key. For this reason, we optimize the efficiency of the sorting algorithm by padding keys to be a multiple of 8 B – ensuring that each key is converted to an unsigned 64-bit integer.

Why should we convert the key to the maximum-size key? To maximize the parallel resources of GPGPU, we need to exactly calculate the position of each block (index block and data block) in each SSTable. Then, we can achieve the multi-thread processing in parallel through the global threads in CUDA. We need to determine the position of each key. If the key size is variable, it will be nontrivial to compute the position of each block in an SSTable. Meanwhile, in case of processing the data block based on the global thread, there may be a locating error that leads to a coding error. Thus, it is challenging to maximize parallel resources utilization on the GPGPU – and we convert the keys to the maximum-sized key. This technique streamlines the computation of block positions by fully utilizing the parallel computational capability of the GPGPU to improve the overall system performance.

How can we pad keys in gLSM? First, when the fixed key size is a multiple of 8 B, padding keys are unnecessary. Second, if the fixed key size is not a multiple of 8 B, the key ought to be padded bytes corresponding to the smallest number greater than the current key size divisible by 8 B. Third, given the variable-sized key, we should pad all the keys in the key range to the key, the size of which is not smaller than the maximum key in the range – and can be divided by 8 B. For example, suppose the range of key sizes is between 6 B and 100 B, we pad all the keys in this range to 104 B which is divided by 8 B.

We face two challenges in implementing an efficient GPGPU-based parallel sort algorithm. First, two index accesses are essential to obtain a KV pair in an SSTable. On the CPU, the KV pairs are serially retrieved; however, the KV is decoded in parallel to fully utilize massive parallel resources on the GPGPU. Second, for each obtained KV pair, it is impossible to load all KVs into the on-chip memory because its limited-size memory is expensive. Thus, it is necessary to store values in the low-speed global memory and place the important indexes of keys and values in the on-chip memory. We present our methods to address these two challenges.

5.5.2 Parallel Decoder/Encoder Controller. The two-level index of the LSM-tree increases the GPGPU's parallel granularity exponentially to process the SSTable encoder-decoder. Figure 8(a) delineates that the footer decoder parses the index blocks, which are also used to parse index information in data blocks. Each SSTable contains a footer, indexblock, and datablocks. The footer includes an offset and an indexblock, which has an offset and the size of a datablock. We allocate a KV pin area in GPGPU global memory to store N SSTables denoted as $S_1, S_2, \ldots, S_i, \ldots, S_N$ $(1 \leq i \leq N)$, one of which contains an indexblock (IB_i). Because the number of index entries in a indexblocks is variable, we employ M_i $(M_i \geq 1)$ instead of a constant value to represent the number of data entries in indexblock IB_i, in which the serial numbers of index entries are in the range between 1 and M_i. One datablock corresponds to an index entry (e.g., IE_1^1 and IB); thus, symbol M_i also represents the maximum datablock serial number corresponding to the i^{th} index block in SSTable S_i, where the serial-number range of datablocks is between one and M_i. Similar, let X_j^i denote the maximum key number in j^{th} data block corresponding to i^{th} index block, where the number range of KV pairs is between one and X_j^i. Then, we have

$$M_{sum} = \sum_{i=1}^{N} M_i, \tag{6}$$

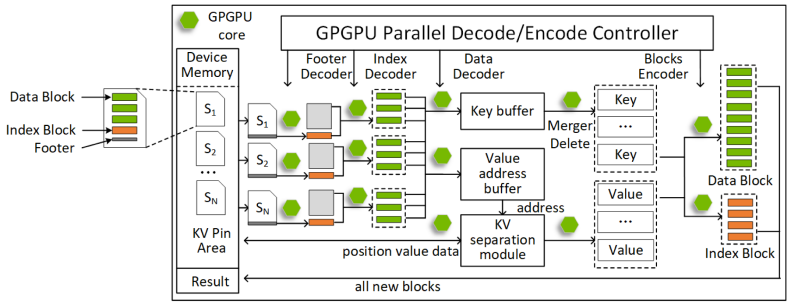
which means the number of all data blocks corresponding to N index blocks in N SSTables. Let $\sum_{j=1}^{M_i} (X_j^i)$ represent the number of KV pairs in all data blocks corresponding to i^{th} index block. Thus, the total number of KV pairs in N SSTables is given as

$$X_{sum} = \sum_{i=1}^{i=N} \left(\sum_{j=1}^{M_i} \left(X_j^i \right) \right). \tag{7}$$

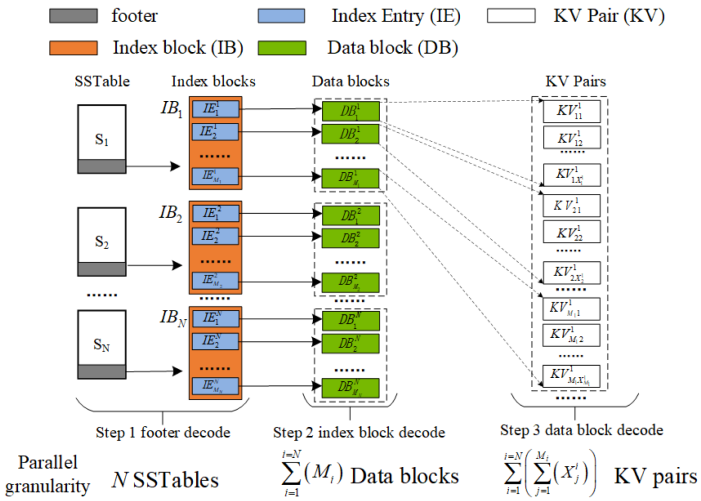
Notably, the degree of parallel granularity is equal to the number of threads currently processing the task with different data granularity (SSTable, datablock, and KV pairs, see Figure 8(b)). For these KV pairs, the parallel-granularity degree of ρ is given as $\rho = X_{sum}$ where we have $(N \leq \rho \leq X_{sum})$. For these N SSTables, the decoder controller kicks off N parallel decoders to parse the footer, see Figure 8(b). Then, the degree of parallel granularity is N. We obtain the size and offset of the initial address of the index when the footer decoder module parses the string - SSTables stream. Then, we initialize N index parallel decoders to parse the indexblock and acquire the size of all datablocks and offsets in the SSTable, see index decoder procedure; meanwhile, we calculate the value of M_i. Finally, M_{sum} datablock parallel decoder threads retrieves key addresses and values in the data decoder module. Then, the degree of parallel granularity rises from N to X_{sum}.

Parsing the footer and index block twice entails three parallel granularity, namely, SSTable level (largest), block level, and key level (smallest) with processing time reducing in order. Then, the GPGPU-side parsing latency is lower than that of the CPU.

The parallel granularity decreases in an encoder procedure. The most efficient coding parallelism is expected to be secured in the GPGPU. The encoding controller assembles values of separated keys and values. gLSM employs a radix sort via a sorting agent that sort-deletes keys. The sort structure contains the subscript index of each value in the buffer. By finding a value address in the address buffer, it is viable to locate the value's address in GPGPU global memory. The key and value are then assembled to restore an original KV entry. The number of keys to be assembled in conjunction with a value determines the number of threads used to search and assemble values. The encoder has the highest parallel granularity. When all the threads are collaboratively finished, the parallel datablock and indexblock encoders are provoked; meanwhile, all the values have been restored. The datablock encoder module encodes all the KV pairs in a group, whereas



(a) Procedure of compaction decode and encode



(b) Different parallel granularity

Fig. 8. Compaction decode and encode in gLSM. Three data granularity: SSTable, datablock, and KV pairs.

the indexblock records the metadata for the grouped keys. At the same time, the metadata information for each datablock is recorded. Applying this approach, the parallel granularity is reduced by an order of magnitude.

For example, for a block size of 4 MB, a KV pair value of 32 B, and a number of keys of around 80, the degree of key-level parallel granularity in the parallel encoder is 80, whereas the degree of block-level parallel granularity is one. When the datablock and indexblock are fully generated, the parallel granularity diminishes to the same value as that of the generated SSTables. There are around 100 parallel kernels being used in this procedure. Thus, the parallel granularity drops by many magnitudes relative to that of the key parallel encoder. There are a number of negative effects associated with this parallel granularity. If the GPGPU originates an SSTable, the parallel granularity will be that of the SSTable. If the GPGPU is unsuitable for generating the SSTable, gLSM counts on the CPU to generate tasks.

5.5.3 Key-Value Separation. There is a challenge, not surprisingly, raised in gLSM: a large number of KV pairs move between the global memory and on-chip memory in the GPGPU. Owing to jumbo data size, data is generally stored in a global memory. Data that must be read and modified frequently resides in the on-chip memory; however, this memory size has a cap and becomes

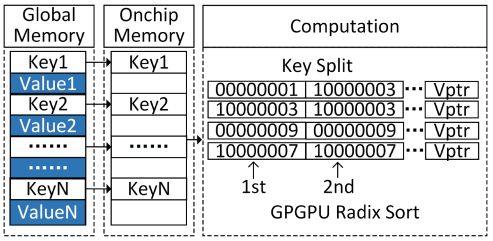


Fig. 9. Details of GPGPU-side key-value separation in which keys and values address are stored separately and in the same order.

Table 1. Symbols used in the Key-value Separation Module

Symbols	Description
s	The size of an SSTable
k	The size of a key
v	The size of a value
α	The size of a part other than the key value (e.g., shared prefix encoding, tail encoding, and tail magic numbers in an SSTable)
p	The size of system address pointer of the value in a KV pair
n	Numbers of KV pairs in an SSTable
Q	The decreased amount of data in a compaction task when gLSM is configured with the key-value separation method

insufficient for storing all compaction data for fast access. The latency of the global memory is higher than that of the on-chip memory by several orders of magnitude. As it is prudent to address the effect of data movement on latency, we propose a GPGPU-side key-value separation mechanism (see Figure 9) for gLSM. When the CPU transfers all data to the global memory in the GPGPU at once, the values need to be retained in the memory to the greatest extent after parallel decoder parsing. The keys are stored in the buffer, while the address of the values is kept in the memory. Figure 9 delineates that the value address and key are placed in the value address buffer and key buffer, respectively. Table 1 lists symbols used in the key-value separation module. In gLSM, the memory address size is 8 B. When we set key size to 16 B, the value sizes include 32 B, 64 B, 128 B, 256 B, 512 B, 1 KB, and 4 KB. We define the size of an SSTable, a key, and a value as s, k, and v, respectively. Each SSTable embraces n KV pairs. We have $n = (s - \alpha)/(k + v)$, where α denotes data volume except KV pairs. All the values are constants except v. The number of KV pairs increases as the value length becomes small. The amount of data movement is cut back by Q using the key-value separation method. Then, we obtain $Q = (v - p) \times n$, where large-size KV pairs decrease n. For example, we surmise that the SSTable size is fixed (e.g., 4 MB) with a KV pair size being 1 B. We have n = 4 MB/1 B = 4*1024*1024. If a KV pair becomes large (e.g., 1 KB), n is (4 MB/1 KB) = 4*1024. We infer that n drops as the size of a KV pair becomes large according to $n = (s - \alpha)/(k + v)$.

When the value length is greater than p, Q, the reduction of the data amount, is positive. As the value is larger than p, the key-value separation is used to process values. A large value reduces data movement and optimizes the overall performance. However, a large value also lowers N and decreases the parallel granularity in gLSM. The key-value separation may expand the size of the KV pair by p: (Key Size + p). In a 16-bit operating system, each pointer occupies two bytes; similarly, four and eight bytes are consumed in 32- and 64-bit systems, respectively. The amount of moved data is reduced to the volume of (16 B + p) down from the original of (16 B + value size). In a 64-bit operating system, the key-value separation reduces the amount of moved data by a factor of up to 170: such a reduction becomes more pronounced at large-size values.

Table 2. The Impact of Key-value Separation Method on the Performance of gLSM

Data Volume (GB)	Key-value separation	GPGPU Total Input (MB)	GPGPU compaction Time (sec)	Throughput (MB/s)	Speedup ratio
5	Without	40.89	6.54	6.3	170.1
	With	7007.4		1071.5	
50	Without	417.41	67	6.2	172.2
	With	71515.53		1067.4	

We present the procedure for calculating a speedup ratio measure in this study. We chose the large-sized value (4 KB) to validate the method of obtaining the speedup ratio. Notably, this is not the limit of the maximum value because the size of the KV pair can be set to a large one, as the large of the size of an SSTable. When we have a 16-B key and 4096-B value, the size of a KV pair size is measured as 16 B + 4096 B = 4112 B without the key-value separation technique. Otherwise, when the key-value separation method is deployed, we merely record the key and a pointer to the value in a tuple (Key, Vsubscript), where a pointer size is 8 B. Then, a KV-pair size – (16 B + 8 B) = 24 B – is transferred to GPGPU-side memory. The speedup ratio in terms of throughput offered by gLSM's key-value separation method is approximately 170 (4112 B/24 B). We conducted experiments to study the throughput of gLSM with/without the key-value separation method under DB_Bench with 16-B key, and 4096-B value, 5- and 50-GB data. As listed in Table 2, the results reveal the minimum throughput before key-value separation and the maximum throughput after deploying key-value separation. The ratio between the two throughputs is around 170 times.

Our finding reveals that this system achieves a premium performance near a 128-B value. The value of 128 B is important because, in the key-value separation method, we replace the value with a pointer rather than dealing with the key. The value size determines the amount of reduced data volume that the GPGPU can process. The size of the key-value pairs (144=16+128) is equally important because, ideally, an SSTable with a fixed size only stores KV pairs: n is determined by the KV-pair size. Reducing the data movement nurtures efficient computation by shortening the execution time of the GPGPU with intensive computational resources and high-latency global memory.

5.6 Workflow of gLSM

Algorithm 1 - the workflow of gLSM - shows that when compaction offloading is invoked, the CPU first performs preprocessing for compaction and determines whether to trigger compaction-task offloading, The transmission module in the management driver establishes an asynchronous data transfer channel between the CPU and GPGPU for task offloading. The asynchronous transmission mode hides the latency of serialization data processing, and parallel data transmission increases bandwidth utilization. When the task offloading kicks off, While acquiring offloading tasks from the CPU end, gLSM prompts the parallel encode and GPGPU sorting modules being executed in parallel to hide compaction latency. The CPU and GPGPU conduct compaction tasks in parallel and the GPGPU performs parallel decoding, sorting, and parallel encoding tasks until the task on the GPGPU is complete. The CPU processes the following SSTables.

6 PERFORMANCE EVALUATION

6.1 Experimental Setup

We compare the performance of gLSM with the alternative KV stores under various YCSB-C workloads and CPU pressures tabulated in Table 3. We run tested KV stores on a server with two Intel(R) Xeon(R) CPUs with E5-2630 v4 @ 2.20GHz CPUs, 32-GB DDR4, and 500-GB Samsung 870 EVO SATA SSD. Each CPU had 10 cores and 20 threads. The operating system for the tests is an Ubuntu 20.04. We deploy the NVIDIA TESLA P100 GPGPU [25] with 16-GB memory on a PCI-E 3 × 16

ALGORITHM 1: The Workflow of gLSM

```
Input: The N number of SSTables involved in compaction (SSTables S_1, S_2, . . . , S_N)
K: The offloading thresholds of decision model
Output: All block of new SSTables
DoCompactionWork(){
    // The CPU performs the compaction
    preprocess(S_1, S_2, . . . , S_N) // Preprocessing dataflow, e.g., pinned swap memory
    if N > K then
        // Triggering to offload tasks to GPGPU. K is the threshold that denotes the number of SSTables
        do in parallel
            // Performing collaborative compaction between the CPU and GPGPU
            GPGPU performs major compaction
            cudaHostAlloc(DeviceBuffer)
            cudaMemcpyAsync(DeviceBufferPtr,H2D)
            kernelDecode(Grids,Blocks,Streams)
            kernelGpuSort(Grids,Blocks,Streams)
            kernelEncode(Grids,Blocks,Streams)
            cudaMemcpyAsync(cpuBuffer,allNewTable,D2H)
            cudaFreeHost(DeviceBuffer)
        end
        do in parallel
            CPU performs minor compaction
            while Status trigger not triggered do
                performing minor compaction circularly;
            end
        end
        processAllNewTableData() //The new data are processed and the CPU flushes the GPGPU-side data to the
        disk
        FLUSH()
    end
    else
        CPU performs compaction
        processAllNewTableData()
        FLUSH()
    end
}
```

lanes interface. The GPGPU P100 boasts 3,584 CUDA cores with 16-GB memory; the NVIDIA K80 [24] encompasses 4,992 CUDA cores with 24-GB memory. To implement the gLSM, we employ CUDA v11.4 and LevelDB v1.20, into which 3.0 K CUDA codes and 1.3 K C++ codes are added.

We configure tested KV stores with 4-MB SSTables and 4-MB Memtable. The remainder of the parameters are set to the default configurations. In alternative KV stores, we employ four and eight threads in **RocksDB** (**RDB**) and PCP [45], respectively. LUDA [40] is a GPGPU-enabled KV store with CPU-oriented merge-sorting algorithm. LevelDB [7] is a de facto baseline version for most KV stores. In the experiments, we compared the performance of the alternative KV stores with LevelDB. RocksDB, extended from LevelDB, is a popular and multi-threaded KV store furnished with new features for applications. We obtained the performance improvement of gLSM compared with LevelDB – the baseline KV store. Comparing gLSM against RocksDB, we validate the advantages of gLSM over representative multi-thread KV stores. In addition, the stress-ng tool [15] launches stress tests in a wide range of selectable ways - exercising various physical subsystems in a computer as well as various operating system kernel interfaces.

Table 3. Characteristics of Workloads

Workload (YCSB-C)			
Type	**Features**	**Key size**	**Value size (B)**
Load 5 GB	100% writes	16 B	32, 64, 128, 256, 512, 1 K, 4 K
Load 50 GB	100% writes	16 B	32, 64, 128, 256, 512, 1 K, 4 K
CPU stress workload (stress-ng)			
Type	**Features**	**Numbers**	**Stress (%)**
CPU Stress	All Method, 0 = Sleep, 100 = Full	40 Cores	0, 20, 40, 60, 80

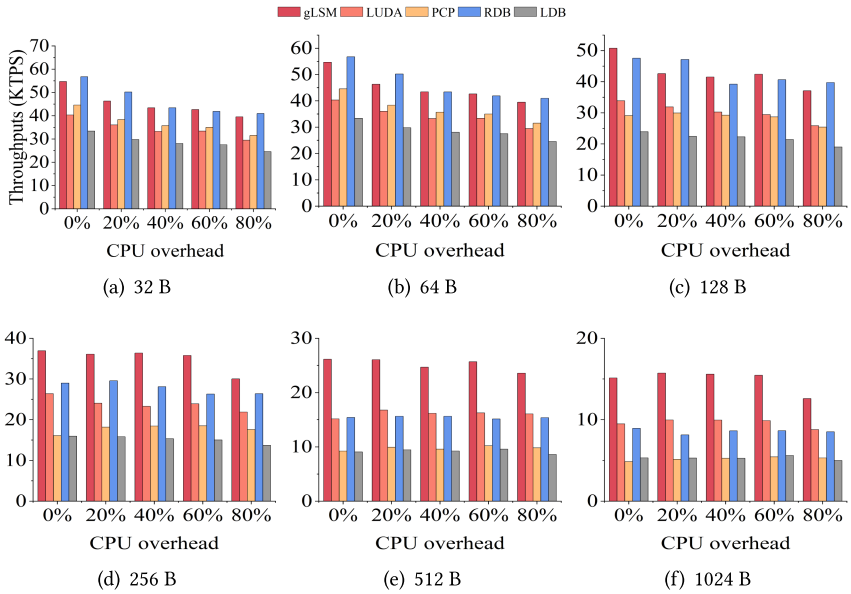


Fig. 10. The throughput under workloads with 5 GB data volumes.

CPU overhead. *Stress* is a stress test tool for CPU, Memory, IO, and disks in Linux. Stress-ng, extended from stress, has hundreds of optional parameters to generate various loads for complex stresses. The stress-ng tool is designed to exercise various physical subsystems of a computer as well as operating system kernel interfaces. Stree-ng embraces 80+ CPU-specific stress tests that exercise floating point, integer, bit manipulation, and control flow. We rely on standard computational loads to simulate workloads in a realistic server environment where higher-priority applications occupy computational resources. We inject stress to each CPU core during the test, where CPU resources consumed by the stress are configured at a preset level (0% 80%). The stress load includes calculation PI, CRC16, FFT, and the like. Notably, we present the approach to calculating the GPGPU utilization method in Section 5.3.

6.2 Throughput

Figures 10 and 11 unveil the throughput under workloads with value sizes (32 B ~ 1024 B). For a fixed value size, we study the throughput under multiple CPU overloads. We study the throughput of the KV stores under different CPU overheads and value sizes to gauge the impact of CPU overhead on the throughput in Figures 10 and 11.

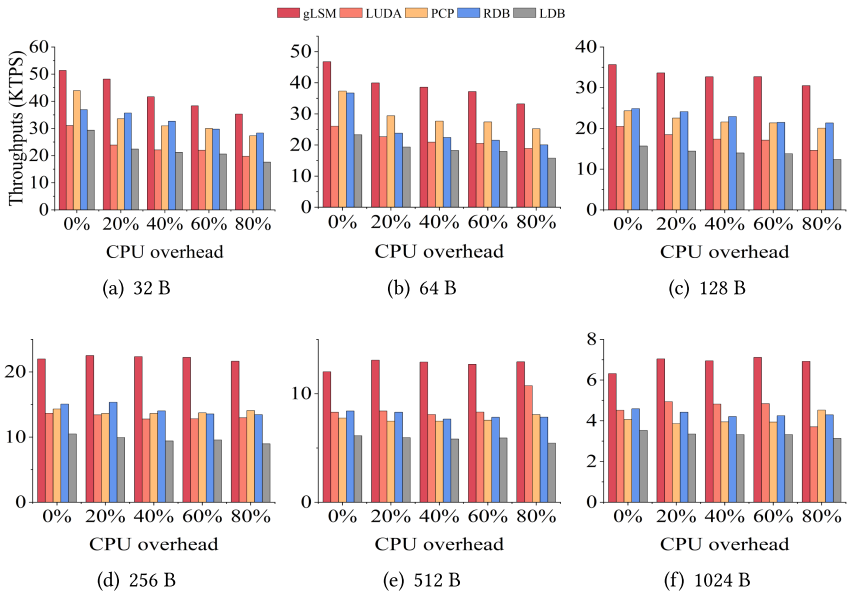


Fig. 11. The throughput under workloads with 50 GB data volumes.

Small Data Volumes (5 GB). Under workloads with a small data volume, the throughput of gLSM and RocksDB are similar, see Figure 10. This is because with low CPU overhead (0 and 20%) and small values (32 B and 64 B), the data in RocksDB cannot be written into higher levels, and its multi-thread mechanism rarely incurs frequent write stalls. As the CPU overhead surges, gLSM achieves an edge over RocksDB. The first three levels in RocksDB, in which nearly all SSTables are stored, are much wider than those in gLSM. Governed by the decision module, gLSM's compaction procedure is rarely spurred at a small data volume. Thus, a majority of compaction tasks are performed on the CPU. gLSM is a front runner in the performance competition against the other KV stores: when the key size is larger than 128 B, gLSM outperforms the alternative KV stores.

Large Data Volumes (50 GB). Given large data volumes, gLSM delivers preeminent performance - clearly surpassing the alternatives under various CPU overheads and sizes of keys in Figure 11. The performance of gLSM is also consistent at value sizes larger than 128 B and is not affected by the extra CPU overhead. When the number of tasks offloaded to the GPGPU surges, the KV-store performance gain yielded by the parallel and sort modules becomes more pronounced. Large data volume causes data to be written at higher levels, and frequent compaction deteriorates the RocksDB performance. gLSM exhibits similar advantages (1.49 - 1.67×) under the four CPU overheads. PCP has a performance advantage at low CPU overhead, but increasing the CPU overhead impairs the advantage because PCP is a CPU-side multi-threaded KV store. A similar trend exists in RocksDB. LUDA offloads compaction tasks to the GPGPU to accelerate computation: the LUDA performance remains stable as the CPU overhead hikes. The merge-sort algorithm of LUDA is CPU-oriented and depends on CPU resources. gLSM optimizes the overall performance of LUDA by up to a factor of 2 thanks to the GPGPU-oriented radix-sort operation.

Discussion. Figure 10 shows the experimental results where the data volume is set to 5 GB. In this case, RocksDB stores the data in the first three levels: the system overhead becomes small. gLSM slightly optimizes performance in terms of throughput. Nevertheless, we reckon that when the tested data volume is 50 GB, gLSM is a clear winner over RocksDB with respect to throughput.

As shown in Figures 10 and 11, when CPU overhead is heating up, the performance of RocksDB is flat: the value size, in these cases, is larger than 128 B. When the value size is expanded, the number of KV pairs in a compaction task becomes small – costing little computing resources. Therefore, the consumption of CPU resources becomes less; therefore, the CPU computing resources costed by RocksDB never reach a bottleneck: the throughput of RocksDB is not obviously affected.

In addition, the 128 B configuration in the study is a deduced theoretical value. According to $n = (s - \alpha)/(k + v)$, the number of KV pairs in SSTable increases in the face of a small key-value pair. The GPGPU compaction task demands a large number of KV pairs, which fully utilize the parallel resources inside the GPGPU - and one GPGPU thread processes one KV pair. For a task including many SSTables, a significant quantity of SSTables simultaneously handled by the GPGPU gives rise to a high parallelism degree, thereby curtailing the latency of data movement, copying data, and computation operations. For example, the latency of copying N SSTables and $2{\times}N$ SSTables is similar in the absence of the maximum transmission bandwidth. Transferring a substantial amount of data at a time reduces the number of transfers with shortened transfer latency. However, a large quantity of KV pairs introduce the cost of replication and movement within the GPGPU. Even though a key-value separation method is applied to avoid a large amount of data movement, this method will not cut back the number of movement operations – the system performance is degraded. Meanwhile, small-sized key-value pairs for the SSTable are CPU unfriendly, and the CPU computing overhead is higher than that of SSTables with large-sized KV pairs. Therefore, we estimated that the system peak performance should occur when the value is around 128 B (key size is fixed in the test).

In the experiments, however, the performance trend is that the system throughput decreases with the increase of value size. The results illustrate that GPGPU compaction performs well under the workloads with large-scale small-sized values. The system overhead is well offset by the performance gain offered by GPGPU. Large-sized values lead to high CPU computational capability, but the disk I/O becomes a system bottleneck: the system performance tends to slump with an increasing value size.

6.3 CPU and GPGPU Resource Usage

Small Data Volumes (5 GB). Under small data volumes, the GPGPU usage of gLSM decreases as the value size expands, but that of LUDA increases (see Figure 12). The reasons are presented as follows. Large-sized data volumes cost high computational resources in GPGPU. Under workloads with small-size values, the key-value separation module of gLSM rarely reduces data volume in compaction. Thus, more GPGPU resources are consumed by compaction – increasing the GPGPU usage. gLSM and LUDA share similar GPGPU usage under this workload. When the value size goes up, the key-value separation module decreases the data volume involved in compaction. The GPGPU resources used in compaction are cut back; then, gLSM has lower GPGPU usage than LUDA.

Large Data Volumes (50 GB). In Figure 13, facing high data volume, the CPU usage of gLSM surges because exceeding CPU resources are used to process compaction at level L_i (i>1) in which compaction is processed in the CPU rather than the GPGPU. LUDA has no key-value separation and uses the CPU-oriented sort algorithm, reducing its efficiency in processing tasks and causing high CPU usage. PCP frequently invokes pipeline compaction - consuming CPU resources and achieving high CPU usage. RocksDB primarily allocates CPU resources to procedures instead of compaction; then, the CPU usage of RocksDB is lower than that of LevelDB, whereas the GPGPU usage of gLSM is at the same level as that of LUDA under small data volumes. The GPGPU processes offloaded data regardless of data size; thus, data volume has little impact on the GPGPU usage: the value size in compaction is a dominating factor in the number of KV pairs.

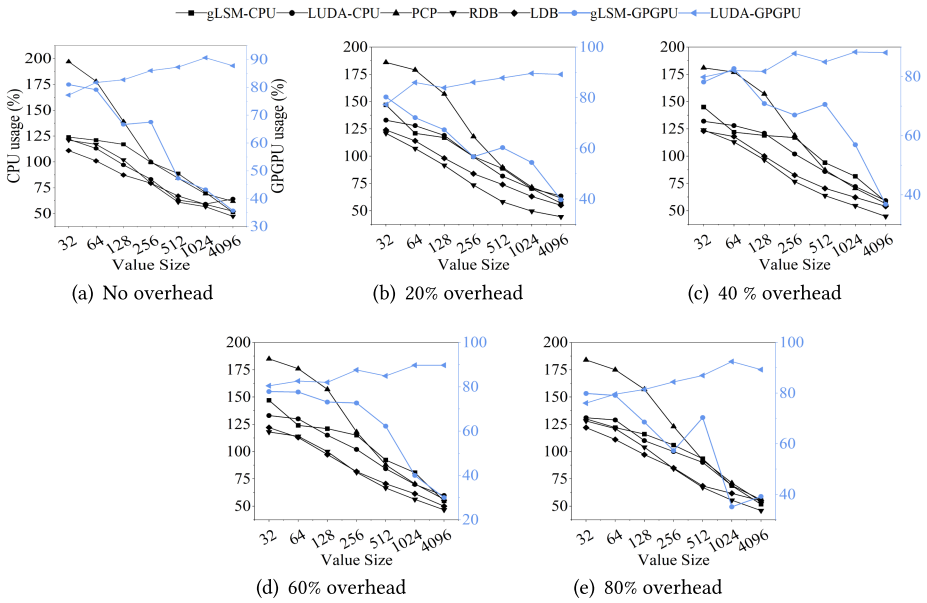


Fig. 12. CPU and GPGPU usage under 5 GB workloads. gLSM-CPU and gLSM-GPGPU denote CPU- and GPGPU-side resource utilization in gLSM, respectively.

In summary, large values cut down the CPU usage of KV stores because the CPU resource cost declines while I/O access time expands as the value goes up. In gLSM, In the management driver, the preprocessing, transmission, and co-sort modules drain computing resources. The CPU usage of gLSM is slightly higher than that of LevelDB (see Figures 12 and 13). On the contrary, gLSM can use considerable computing resources to accelerate compaction performance at the expense of reduced CPU cost. One advantage of our proposed gLSM is that gLSM has the potential to decrease the occupancy of GPGPU cores as the value size becomes large. The cores occupancy of gLSM is lower than that of LUDA with an increasing value size because the key-value separation method in gLSM reduces the cores' occupancy. Thus, the power consumption of GPGPU is conserved with the improved GPGPU utilization, whereas LUDA maintains a high GPGPU occupancy that inevitably leads to high power consumption.

6.4 Average Write Latency

We study the latency of gLSM under the workloads with various data volumes and value sizes.

Small Data Volumes (5 GB). In Figure 14, with 0 and 20% CPU overheads, gLSM and RocksDB exhibit a low latency as the value size is smaller than 128 B. This is because the co-sort module of gLSM has a high load under workloads with a small data volume and small-sized values. Then, it is arduous to exploit the advantage of GPGPU to process a large data volume. Once the value size exceeds 128 B, gLSM achieves the lowest latency than the other KV stores.

In case of large values, gLSM reduces the latency by up to 3.1×, 2.88×, 1.69×, and 1.72× compared with PCP, LevelDB, RocksDB, and LUDA, respectively. When the CPU overhead goes beyond 40%, gLSM cuts down the latency by up to 1.56×, 3.14×, 1.81×, and 2.96× compared with LUDA, PCP, RocksDB, and LevelDB, respectively. As the CPU overhead surges, the latency of PCP rises up but gLSM reduces the latency by 3.52× compared with PCP. With a 512-B value and 80% CPU overhead, the latency of the other KV stores climbs owing to the lack of CPU computing resources. In contrast,

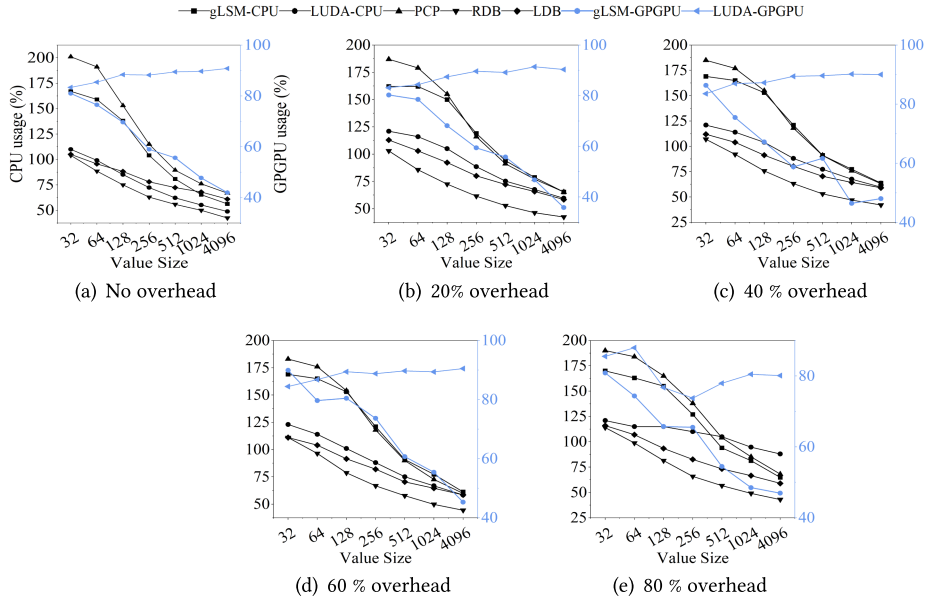


Fig. 13. CPU and GPGPU usage under workloads with 50 GB data volume.

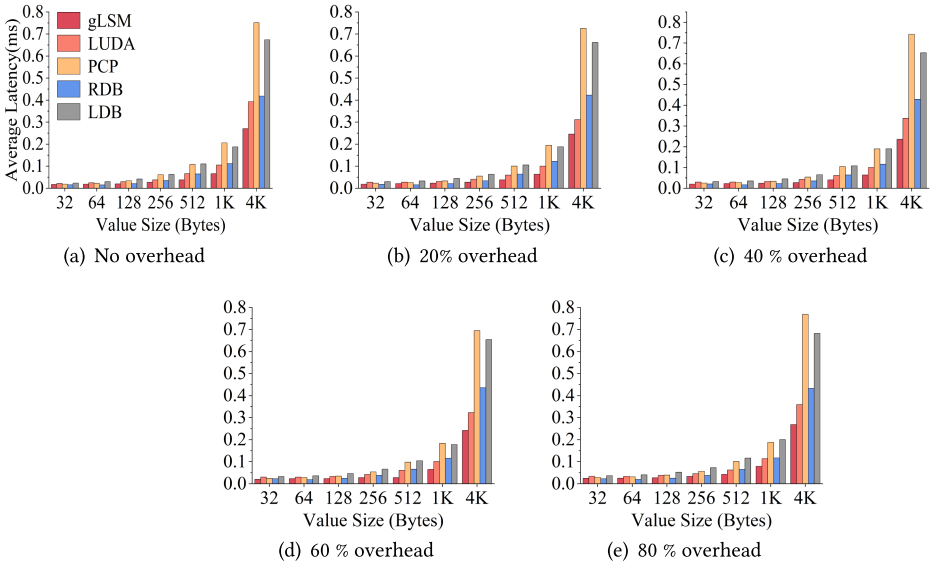


Fig. 14. The average write latency under 5 GB workload.

gLSM shortens latency by up to a factor of 1.4×, 2.86×, 1.61×, and 2.74× compared against LUDA, PCP, RocksDB, and LevelDB, respectively.

Large Data Volumes (50 GB). At a write data volume of 50 GB in Figure 15, lower-level compaction tends to block user requests, increasing the latency relative to smaller data volumes. gLSM has a performance edge in terms of low latency over the other KV stores, because large data volumes leverage GPGPU to undertake compaction tasks. When the CPU overhead hikes from 20%

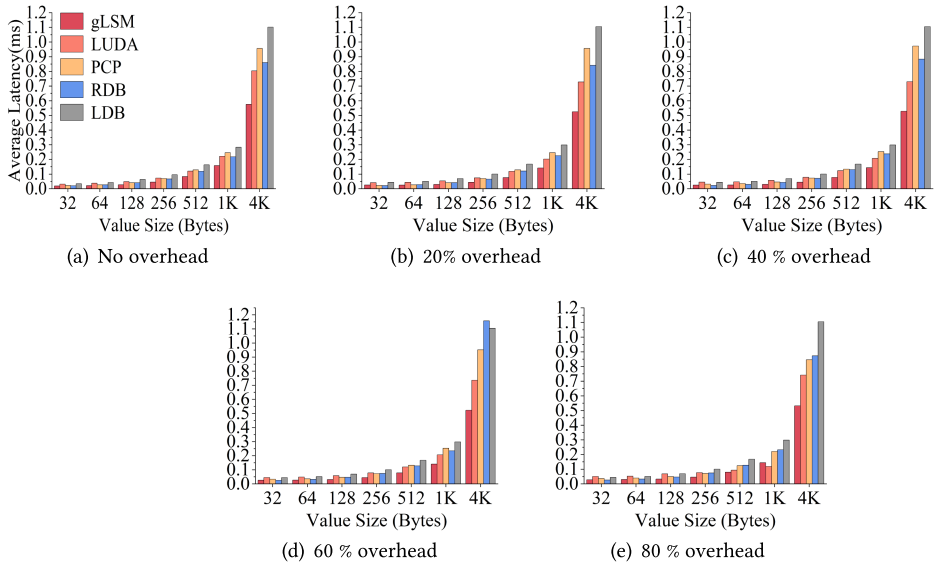


Fig. 15. The average write latency under 50 GB workload.

to 60%, the latency of gLSM remains unchanged: the use of the GPGPU improves the compaction to the greatest extent, achieving the most stable and lowest latency. Compared with LUDA, gLSM boasts a latency reduction of 2.09× when CPU overhead and value are 80% and 128 B. gLSM slashes the latency of PCP by a factor of 1.84 in case of CPU overhead and value being 40% and 4 KB. At 60% CPU overhead and a 4-KB value, gLSM reduces the latency of RocksDB by a factor of more than 2.21. When it comes to LevelDB, gLSM reduces the latency by a factor anywhere between 1.7 and 2.47.

6.5 Compaction Bandwidth

Compaction bandwidth, defined as a ratio of total compacted data to total compaction time, serves as an intermediate metric to gauge compaction performance. Table 4 lines up the compaction bandwidths under a diversity of workloads with five CPU overhead. Notably, gLSM-CPU and gLSM-GPGPU means the compaction bandwidth of gLSM in the CPU and GPGPU sides. LUDA-CPU represents the CPU-side compaction bandwidth of LUDA. We observe that the CPU-side compaction bandwidth of gLSM is close to those of PCP and RocksDB. Even though there is no GPGPU acceleration, the CPU-side compaction speed is very impressive. gLSM speeds up the CPU-side compaction of the baseline solution by a factor of two. When the CPU and GPGPU collaboratively perform compaction, the compaction bandwidth climbs as the value size increases. The reason is that (1) the key-value separation module trims the data accesses to GPGPU memory and (2) the compaction procedures at large values contain fewer KV pairs, shortening time spent in sorting and de-duplicating keys. Under a workload with 4-KB value, gLSM boosts compaction bandwidth by 26× relative to the host.

6.6 Discussion and Summary

The GPGPU-oriented radix sort algorithm in gLSM is a dominant factor of compaction performance. The key-value separation module and parallel decoder/encoder modules are designed to boost the performance of the radix sort. When compaction is handled in a GPGPU, a global cache in the GPGPU registers all SSTables involved in compaction for the parallel code-decode procedure.

Table 4. Compaction Bandwidth (MB/s) on the CPU and GPGPU Sides under 5 and 50 GB Data Volumes Workloads with Five CPU Overheads (0%, 20%, 40%, 60%, and 80%)

CPU overheads	Data volume (GB)	5							50						
	Value size (Bytes)	32	64	128	256	512	1024	4096	32	64	128	256	512	1024	4096
0%	gLSM-CPU	62.6	65.7	73.6	77.9	93.3	92.5	89.4	59.3	71.6	81.3	86.6	88.2	89.1	96.1
	LUDA-CPU	24.3	29.2	38.0	49.6	53.0	64.3	64.2	28.3	34.1	44.3	53.4	61.6	64.0	69.1
	PCP	58.0	70.6	78.0	78.5	85.4	84.3	91.0	60.1	79.6	90.5	95.6	98.0	101.0	100.8
	RDB	64.0	73.9	88.1	94.8	94.8	106.3	110.0	54.1	61.2	70.2	76.5	80.6	85.1	84.3
	LDB	29.6	35.6	43.0	52.1	55.8	62.2	66.2	30.3	37.9	44.6	54.3	35.3	68.1	40.5
	LUDA-GPGPU	9.4	15.5	27.4	47.0	71.7	99.9	132.2	9.3	15.0	27.8	47.3	72.2	97.0	127.7
	gLSM-GPGPU	48.6	156.2	284.3	547.2	901.2	1531.0	2516.5	100.8	164.1	321.5	587.2	913.9	1525.3	2504.1
20%	gLSM-CPU	46.0	65.7	73.6	77.9	94.3	92.5	89.4	50.4	66.5	83.3	94.0	101.0	104.7	110.7
	LUDA-CPU	19.4	26.2	36.8	47.9	58.5	67.5	81.5	21.7	29.9	40.5	53.2	63.6	72.1	79.0
	PCP	45.9	62.9	82.1	89.5	93.0	89.6	94.4	46.5	64.5	85.0	93.1	96.8	97.7	98.3
	RDB	58.2	70.7	87.5	97.0	96.6	96.7	109.5	50.2	59.4	68.2	77.9	79.4	82.2	86.0
	LDB	23.2	31.7	40.4	51.8	58.5	62.4	66.6	23.1	31.5	41.2	52.1	60.1	64.7	69.5
	LUDA-GPGPU	6.9	11.3	20.5	36.4	59.6	84.5	116.6	6.7	11.1	20.2	35.5	56.9	81.4	118.4
	gLSM-GPGPU	46.9	156.2	284.3	547.2	600.2	1531.0	2516.5	92.5	136.7	307.5	533.8	776.4	1494.4	2492.3
40%	gLSM-CPU	40.6	57.0	69.4	86.3	88.8	99.0	103.5	46.7	65.1	82.7	93.9	102.2	104.0	110.7
	LUDA-CPU	17.8	24.3	35.1	46.4	57.1	65.8	74.8	20.1	27.4	38.1	50.4	61.2	70.6	79.1
	PCP	42.5	60.0	78.8	90.8	89.4	91.8	92.1	42.8	60.7	81.7	93.0	96.8	99.0	100.5
	RDB	50.2	67.8	81.9	92.4	96.1	102.8	108.1	48.4	56.6	64.8	71.1	73.3	78.0	82.0
	LDB	21.6	29.9	40.0	50.0	56.9	61.7	67.6	21.8	29.6	39.9	49.0	58.8	64.2	69.3
	LUDA-GPGPU	6.3	10.5	19.0	33.6	55.7	82.2	109.0	6.2	10.2	18.5	32.8	54.1	78.2	120.0
	gLSM-GPGPU	26.7	114.0	219.0	352.5	511.3	846.6	2220.2	65.0	164.3	310.2	586.2	847.2	1375.7	2198.8
60%	gLSM-CPU	39.9	56.8	71.3	82.7	88.1	99.2	100.6	44.1	63.0	80.9	93.1	103.7	106.8	111.7
	LUDA-CPU	17.4	24.4	33.3	45.6	56.9	66.4	78.4	19.9	26.9	37.1	50.6	62.6	70.7	78.1
	PCP	41.8	58.4	79.1	90.4	95.2	96.0	98.4	41.4	59.6	80.8	93.2	97.2	100.0	102.3
	RDB	53.1	58.6	75.2	86.7	93.4	102.6	105.8	43.0	51.7	60.7	68.8	75.1	78.9	83.9
	LDB	21.1	29.3	38.6	48.8	58.9	66.0	67.6	21.1	29.1	39.3	50.0	59.8	64.1	70.7
	LUDA-GPGPU	6.1	10.2	18.2	32.7	54.8	80.2	120.1	6.0	9.9	18.0	31.9	52.6	76.5	111.6
	gLSM-GPGPU	26.9	118.1	195.9	290.0	900.2	1096.7	2143.4	47.4	125.1	154.8	278.5	509.2	841.4	2043.5
80%	gLSM-CPU	38.4	49.8	59.5	69.4	87.6	77.5	90.9	41.8	56.9	78.1	92.9	103.5	106.4	112.6
	LUDA-CPU	15.4	21.4	29.1	41.7	56.6	59.3	70.3	17.9	24.8	14.3	21.0	30.9	45.1	56.8
	PCP	36.3	51.8	69.6	84.7	89.5	90.4	88.1	37.5	54.2	75.5	98.5	103.8	107.1	108.7
	RDB	45.1	57.8	73.7	86.7	94.6	101.1	106.3	41.6	50.6	60.2	68.2	75.1	79.6	83.0
	LDB	18.8	25.9	34.2	44.7	52.4	57.8	64.8	18.0	25.6	35.2	46.9	54.9	60.7	66.1
	LUDA-GPGPU	5.4	9.1	16.4	30.1	51.7	79.1	123.3	5.4	8.9	5.5	9.1	15.5	28.9	52.6
	gLSM-GPGPU	95.5	152.0	272.8	535.0	897.3	1526.0	2540.4	93.5	158.4	305.4	474.8	829.1	1477.9	2397.5

The procedure decomposes each KV item into parallel read tasks, and each thread wrap deals with a read task of KV items in a data block. The keys, instead of values, are merely read from the global memory. These string-mode keys are converted into integer-mode ones processed by the GPGPU. These integer keys, being quickly sorted by radix sort, are converted into a byte stream and stored in the key buffer. In addition, the parallel encoding module reads the key buffer and value in the global cache, and each thread assembles a data block. Then, the two modules and the sort algorithm accomplish the GPGPU-side compaction.

We employed DB_Bench to generate a workload with 1,000,000 KV pairs. These KV pairs are first written following by read the same-volume KV pairs. The results are listed in Table 5. The high-speed GPGPU has memory overhead: the memory cost of gLSM is higher than those of the other methods. In this article, memory ratio is the proportion of the memory used by the running application in the total memory space. We used the memory ratio to reflect the CPU-side memory usage of the tested KV stores. We present the memory ratios of the alternative KV stores under the workload conditions. In this test, the lookup order of the keys in workloads is fillrandom. The lookup cache is cold cache space - a small memory space - that is used to buffer write data for subsequent writes to the disk rather than cached data.

As shown in Table 5, the memory footprint of gLSM is 4%, higher than that of RocksDB. The GPGPU driver threads and runtime libraries in the background take up memory; however, these drivers and libraries facilitate GPGPU-side compaction procedures. The proposed gLSM incorporates the pipelining technology and GPGPU compaction module. We optimized the performance of KV stores using the pipelining, and the query performance can be improved further – an objective

Table 5. Look-up Performance and Memory Ratio in Tested KV Stores

Test project	gLSM	LUDA	RDB	PCP	LDB
Lookup (micros/op)	42.857	45.603	6.7	54.774	10.288
Memory ratio(%)	4	1.8	0.4	2	0.5
Numbers of found keys	1000000	1000000	1000000	1000000	1000000

Table 6. Throughput with Different Sizes of Tasks in Parallel

Table Size (MB)	4	6	8	10	12	14
Throughput (KTPS)	45.91	46.44	40.24	38.64	36.49	38.04
Block Size (MB)	1	2	4	8	12	16
Throughput (KTPS)	44.60	44.16	43.85	44.50	44.15	45.79

that is subject to a future study. For example, the lookup performance of gLSM is in the middle of the five KV stores. This is because gLSM is featured with lazy compaction, and we need to send as many tasks as possible to the GPGPU at once to tradeoff the time cost of the data transfer. Under the workload with 1,000,000 KV pairs, gLSM looks up 1,000,000 KV pairs that have been written. Thus, gLSM has good stability. We also plan to exploit GPGPU direct storage technology to minimize the memory footprint, and the GPGPU-oriented cache structure should be designed for KV stores. We will optimize the read performance of gLSM to eliminate the gap with RocksDB in our future work. In this study, we implemented the GPGPU-accelerated software stack for KV stores, achieving a promising compaction performance improvement.

Notably, the data volume of 1 million KV pairs is no more than 144 MiB, which is easily loaded into the cache. Nevertheless, we employed this configuration in the test to study whether we have lost key-value pairs during processing KV pairs by the GPGPU rather than measuring the look-up performance of KV pairs. The results validate the good robustness of our proposed gLSM: we did not lose any KV pairs in this procedure.

6.7 Scalability Experiments

Task size in parallel. We analyze the impacts of the compaction task size in parallel on the throughput of gLSM. Table 6 lists the six SSTable sizes configured in the experiments. The size of an SSTable affects the throughput; at a given parallel task granularity, a large SSTable results in very large amounts of data per task, in turn creating more data transfer and write-back operations, which cost much time. Furthermore, as the SSTable size inflates, the number of SSTables per level shrinks, leading to frequent compaction on the CPU coupled with an decreased overall throughput. The results in Table 6 show the impact of parallel task size on throughput in the case of the six block sizes: the effect of block size on throughput is limited to 4% - a range subject to the systematic error.

Impact of thread-grid size on performance. The thread grid is a three-dimensional structure that divides tasks in gLSM. We set the lengths of the x- and y-dimensions to one each and variously configured the z-dimension to lengths of 4, 8, 16, 32, 64, and 128 to study the effect of task size on throughput. Figure 16(a) shows that the thread grid size imposes a limited effect on throughput, with each task divided according to different grid sizes, but the overall parallel granularity remains unchanged. The logical partitioning simply allocates threads to perform each task rather than affect the occupation of the physical SMs. As a result, the grid size has a limited impact on the throughput.

Impact of Offloading Thresholds. The offloading threshold determines the number of tasks to be executed in parallel on the GPGPU. We set six thresholds to investigate the effect of the threshold on throughput. Figure 16(b) unfolds that a large offloading threshold boosts the

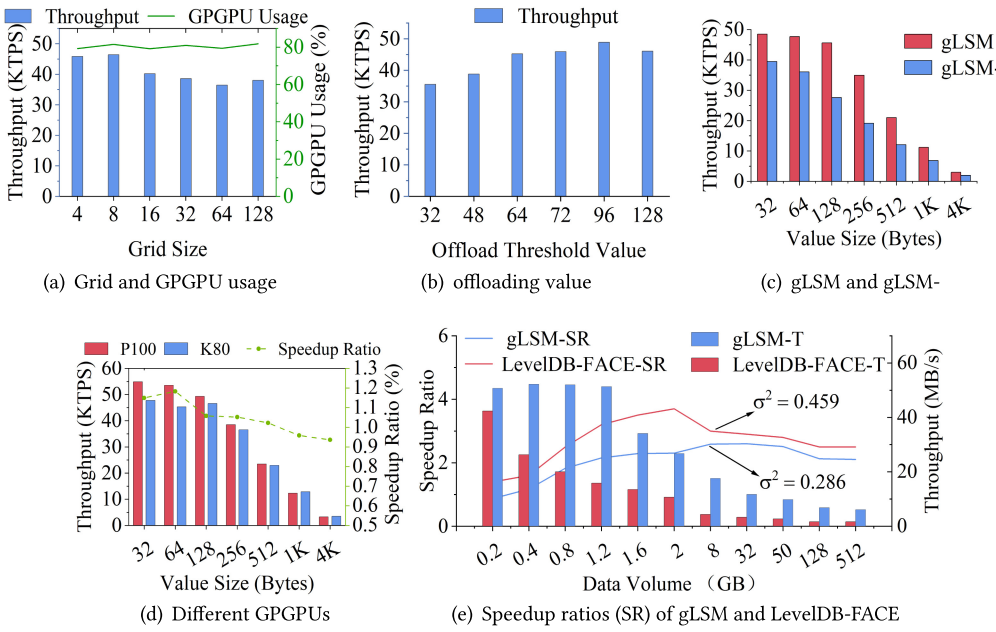


Fig. 16. GPGPU parameters based on extended studies in gLSM.

throughput because the number of parallel tasks grows - improving the compaction efficiency. A large threshold also reduces the number of data transfer operations at a given number of tasks, thereby lowering the total transfer time.

Impact of KV Separation and GPGPU-side Sort Modules on Performance. We investigate the baseline **gLSM** in which the GPGPU did not have key-value separation or GPGPU sort modules, denoted by **gLSM-**. Figure 16(c) shows that the gLSM-'s throughput declines - an expected outcome as its value size is around 128-256 B. The GPGPU sort module fully utilizes GPGPU computing resources. To shorten access latency amid sorting operations, the key-value separation module reduces the overhead of data movement and copy operations. The findings prescribed in Section 6.3 suggest that without deploying these two modules, LUDA computing resource consumption is inflated by a factor of 2.1× under the 4-KB-value workloads in GPGPU (see Figures 12 and 13). Figure 16(c) unveils that gLSM revamps the throughput by up to 82.6% over gLSM-: the two modules play a crucial role in gLSM to optimize the compaction performance in the GPGPU-enabled KV stores.

Impact of GPGPU Resources on Performance. We incorporate a GPGPU K80 with fewer resources into gLSM to delve into the impact of GPGPU resources on performance. The K80 differs from our GPGPU P100 in terms of CUDA cores and memory capacity. gLSM optimizes the utilization of GPGPU resources and boosts the compaction performance, even though the K80 has weak resources. The K80-based gLSM performs similarly to the P100-based gLSM. The results plotted in Figure 16(d) unveils that the K80-based gLSM offers stable performance: the remarkable stability of gLSM is made possible by GPUs with diverse computing resources. The key-value separation module in gLSM not only conserves resource costs, but also fully utilizes GPGPU resource to speed up performance at the expense some GPGPU resources. Overall, gLSM deliver better performance on P100 as opposed to the K80 counterpart. gLSM utilizes a key-value separation module to cuts down the number of resource-consuming access operations. In addition, the parallel encoder-and-decoder controller enables the parallel encoder at the block level, averting degradation in GPGPU

performance under heavy access and control tasks. Then, gLSM vastly reduces the resource costs needed to expedite parallel compaction. It is flexible to run gLSM on various GPGPUs to achieve considerable acceleration in KV-store performance under low resource usage.

We conducted an experiment based on K80 and P100 GPGPUs to validate the robustness of the gLSM. We find that the performance of K80 is similar to that of P100. This is because the GPGPU-enabled collaborative compaction procedure does not fully occupy GPGPU resources, see Figures 12 and 13. In addition, the computational capability (TFLOPS) of P100 is higher than that of K80 [24, 25]. The GPGPU-side compaction subsystem of gLSM costs few computational resources whereas the throughput is determined by the computational capability and all modules of gLSM. With the same data volume, both P100 and K80 are not fully loaded; then, P100-based gLSM has a similar throughput to K80-based gLSM.

Comparison of Speedup Ratios (SR) of gLSM and LevelDB-FACE. LevelDB-FACE [33] is a heterogeneous system to stimulate KV-store performance using an FPGA. We configure the value size of 512 B to mimic that in LevelDB-FACE, and Figure 16(e) plots the speedup ratios of gLSM and LevelDB-FACE. We present the basic system throughput when we test the speedup ratio of gLSM and LevelDB-FACE, and gLSM has higher basic system throughput than the counterpart. gLSM reveals stable speedup ratios as the data volume increases, but this metric of LevelDB-FACE is unstable under the workloads with small-sized data owing to the settings of the memory-table size. In case of large data volume, LevelDB-FACE's speedup ratio significantly declines. In addition, we show the variance (δ^2) of the speedup ratios of the two KV stores: the speedup ratio of gLSM is smaller than that of LevelDB-FACE. This result unveils that our gLSM is more reliable than LevelDB-FACE under the production environments where the system stability is sensitive.

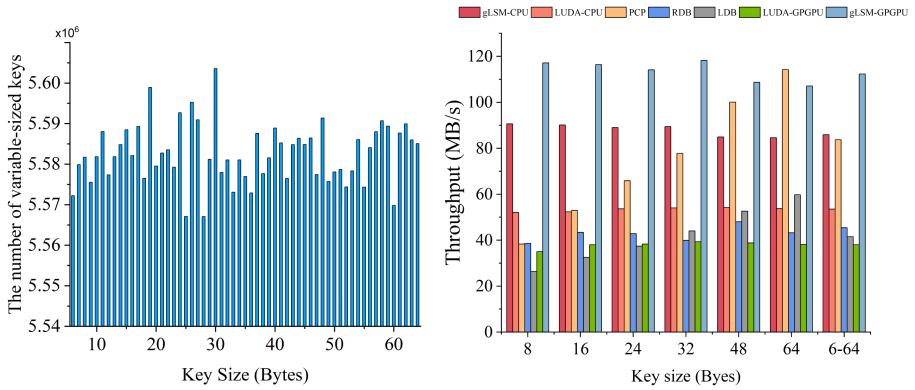
Impact of variable-sized keys on the performance of gLSM. We conducted two types of experiments based on YCSB-C to validate gLSM under workloads with variable-sized keys. First, we set the key size to 8 B, 16 B, 24 B, 32 B, 48 B, and 64 B, and the value size is 128 B. The data volume is 5 GB. Second, amid the lack of tools forging variable-sized keys, we created variable-sized keys with a range of key sizes between 6 B and 64 B. We also employ 5-GB data in which there are about 32,936,866 KV pairs. We measure the throughput of CPU and GPGPU, the sorting time, and the ratio of sorting time to the compaction time under YCSB-C.

As shown in Table 7, the comparison count is two when the key size is 8 B. We find that the number of comparison operations is more than one. Let us justify why we must convert to the maximum key length – and why we ought to convert the key size to an integer multiple of 8 B. This procedure aims at enhancing sorting performance: we convert every 8-Bytes key to an unsigned 64-bit integer; the 8-B keys need to be sorted at most once; the 32-B keys are sorted at most four times. In gLSM, we have one more comparison: we compare the sequence in the internal key to ensure that if the keys are identical, the keys can be sorted correctly according to the sequence. As shown in Table 7, when the key size increases from 16 B to 32 B, the throughput of GPGPU compaction has a small increment because the large-sized keys enlarge the amount of data. Then, the number of sorting operations rises from 3 to 5. There is a slight difference in sorting time. The GPGPU-side compaction throughput declines when the key size grows to 48 B or even larger, because the number of comparisons and the sorting time significantly increase, which in turn worsens compaction time and throughput.

As shown in Figure 17(a), the x-axis represents the key sizes (key range between 6 B and 64 B). The y-axis denotes the number of different-size keys. The minimum number of keys, the size of which is 28 B is 556,704. We counted the number of different key sizes. The maximum number of keys - the size of which is 30 B – is 560,354. Since the number of these variable-sized keys is basically uniform, we take the average size of these variable-sized keys as 35 B; then, we launch six comparison operations to achieve the value. As listed in Table 7, we observe that the GPGPU

Table 7. The Performance of gLSM under YCSB-C with Variable-sized Keys

Key size (Bytes)	Comparison count	Throughput (MB/s)		Compaction time (sec)	Sort time (sec)	Avg. Sort time (sec)	percentage
		CPU	GPGPU				
8	2	90.6	117.1	644.3	65.4	1.49	10.1%
16	3	90.1	116.4	647.3	65.5	1.49	10.1%
24	4	89.0	114.1	657.3	66.5	1.51	10.1%
32	5	89.4	118.2	653.9	65.0	1.48	9.9%
48	7	84.9	108.7	685.3	69.2	1.57	10.1%
64	9	84.6	107.1	693.8	70.1	1.59	10.1%
Variable-sized keys	6 (Avg.)	85.9	112.3	682.4	67.3	1.53	9.90%



(a) Numbers of various-sized keys (b) Throughput

Fig. 17. Performance of gLSM under workloads with different-sized keys. (a) presents the number of various-sized keys under the workloads. (b) shows the throughput of KV stores under fixed- and variable-sized keys.

throughput of gLSM is 112.3 MB/s, which lies between that of gLSM under YCSB-C with 32-B and 48-B keys. For the two fixed-sized keys, the number of comparison operations is five and seven, respectively. While the average number of comparisons for these variable-sized keys is 6, which still lies between 5 and 7. When the number of comparisons expands, the sorting time is enlarged. Such an impact is marginal in case of small-sized keys because as the key size increases, the sorting time and throughput show a minor change. Given a large key, such as 48 B and above, the system performance significantly drops due to growing total sorting time increases coupled with downgraded throughput.

In addition, we conducted experiments on fixed- and variable-sized keys to validate the advantages of gLSM under workloads compared with RocksDB, LevelDB, PCP, and LUDA. Other parameters are consistent with that in the test of gLSM. We achieved the following results in Figure 17(b). Under YCSB-C with variable-sized keys, the throughput of gLSM is the same as the case of a key size that lies between 32-B and 48-B as illustrated in the test with fixed-sized keys. In the fixed-sized keys, the throughput of LevelDB and PCP grows as the key size increases because large-sized keys can expand data volume, increasing throughput. In the PCP case, when the key size reaches 64 B, the throughput of PCP outperforms that of gLSM – but both the CPU and GPGPU throughput of gLSM are larger than that of PCP under variable-sized keys. When the key size is 48 B or even larger, the throughput of RocksDB deteriorates, which may be related to the increment in the

Table 8. Impact of Disk I/O on the Performance of gLSM

Tested Types	CPU+SATA SSD		CPU+GPGPU+SATA SSD		CPU+GPGPU+PCIe SSD	
Data volume (GB)	5	50	5	50	5	50
CPU compaction throughput (MB/s)	86.8	74.6	138.2	141.2	259.5	298.6
GPGPU compaction throughput (MB/s)	–	–	1063.8	1071.8	1062.2	1078.4

number of comparisons. In summary, we validate that our gLSM outperforms the other KV stores under workloads with variable-sized keys.

Impact of Disk I/O on the system performance. To validate whether disk I/O is the system bottleneck, we opt for three configurations for the tested platform and conduct experiments, namely, (1) CPU+ SATA SSD, (2) CPU+GPGPU+SATA SSD, and (3) CPU+GPGPU+PCIe SSD. We use an Intel Optane PCIe SSD 900P Series with 480 GB and Samsung SSD 870 EVO with 500 GB in this test. The workload is configured with a 1-KB value and 5 GB/50 GB data.

Given CPU + SATA SSD, we quantified the CPU compaction throughput of LevelDB under a diversity of workloads. For the other two tested platform configurations, we measured the CPU and GPGPU compaction throughput of gLSM. Compared gLSM with LevelDB - a baseline solution, we concluded that there is a computational bottleneck of CPU + SATA SSD in LevelDB, which is significantly alleviated by our gLSM through the heterogeneous GPGPU. In this experiment, the performance of CPU + SATA SSD using gLSM is inaccurately measured because gLSM performs compaction on the CPU in the context of data volume involved in compaction that does not reach the offloading threshold. The advantage of conducting compaction on GPGPU is opaque. For large-scale data volumes, we offloaded the compaction to GPGPU, achieving collaborative processing. In the experiments based on CPU+GPGPU+SATA SSD, we demonstrated performance improvements in terms of CPU throughput increases. This trend is expected because we offload the compaction tasks with large-sized data volumes to GPGPU, improving the overall performance. Therefore, we employed gLSM in the second experiment, the tested results of which are listed in Table 8.

Comparing the first and second experiments, we observe that the computation becomes a performance bottleneck – and the disk I/O in the second experiment is the performance bottleneck illustrated through the comparison between the second and third experiments. In the second experiment, some compaction tasks are offloaded to the GPGPU, and the disk I/O becomes the primary bottleneck of compaction procedures. The experimental results unveil that GPGPU reshapes the compaction performance because of its high parallel resources, but the disk I/O becomes the main compaction bottleneck. In particular, we discovered that the PCIe SSD-based system – achieving high overall system performance – has a performance edge over the SATA+SSD system.

We believe that there will be some opportunities to optimize the computation process on the CPU and GPGPU sides. For example, we may bypass the CPU and directly read data from the SSD by the virtue of the advanced GPGPU direct storage technology. Although the computational resources of the CPU cannot be used to manipulate data anymore, the data transmission path is shortened to lower the latency. This GPGPU direct storage technology has not been popularized.

Impact of memory usage on the performance of gLSM. To investigate the maximum memory usage of gLSM, we configured different-sized values to test the maximum memory usage of GPGPU and CPU for gLSM. We employed 16-B key and 5-GB data in this test.

As listed in Table 9, the results show that the maximum GPGPU memory of gLSM declines as the value size increases because the large-sized values reduce the number of KV pairs in the SSTable – and the amount of data involved in computation in the GPGPU is shrinking. In addition, the maximum CPU memory usage increases as the value size becomes large. The maximum CPU

Table 9. Impact of Memory Size on the Performance of gLSM

CPU and GPGPU memory usage of gLSM							
Value size (Bytes)	32	64	128	256	512	1024	4096
GPGPU Maximum Memory usage (MB)	3766	3432	3172	3008	2892	2870	2826
CPU Maximum Memory usage (MB)	1841	1763	1699	1669	1695	1756	1735
Impact of memory size on the throughput of gLSM							
CPU memory Limit (GB)	2	4	8	16	unrestricted		
CPU Throughput (MB/s)	97.52	103.68	107.2	113.6	114.35		
GPGPU Throughput (MB/s)	508.66	513.67	508.05	519.42	519.44		

memory usage by gLSM decreases as the value size goes up in a range between 32 B and 512 B. The reason for this performance trend is similar to that of GPGPU. When the value size is 1024 B, the maximum memory usage of the CPU rises because the size of data blocks involved in computing and transmission is enlarged. Notably, the data block size is generally configured as 4096 B. The minimum size of the data block in gLSM is 3687 B. For the 512-B value, the data block size shrinks when the value size becomes large. The data block size is (16+8+512) * 7 = 3752 B, where (16+8) B is the internal key size – and this block size is larger than 3687 B. For the 1024-B value, if we add 3 KV pairs, the data block size is (16+8+1024)*3=3144 B. The size of the data block is smaller than the minimum value of the data block 3687 B. Therefore, we ought to insert the fourth KV pair – and the size of the data block is 4192 B, which is larger than the block size with 512-B values. The maximum CPU memory usage surges when the value size is 1024 B.

To delve into the impact of memory limitation on system performance, we limited the system's memory usage to 2 GB, 4 GB, 8 GB, 16 GB, and no limitation on the CPU. Then, we measured the changes in system performance. In this test, we configured 16-B key, 1-KB value, and 35-GB data. We obtain the following observations as shown in Table 9:

First, when memory size increases from 2 GB to 16 GB, the throughput of both CPU and GPGPU rises. This result reveals that memory size affects the performance of the system. To achieve better performance, it is necessary to ensure that the system has large-sized memory. Second, when memory is not limited, the CPU's throughput enhances, but GPGPU's throughput has no obvious change and is similar to that of a 16-GB-enabled GPGPU. This result unveils that GPGPU performance may not be driven by available memory because GPGPU memory is not limited, and restricting CPU memory has zero impact on GPGPU-side throughput. Third, regardless of the memory limitation, the GPGPU-side throughput is always higher than that of the CPU: GPGPU may be more effective when processing large-scale data in KV stores.

Studying GPGPU utilization of gLSM. When the number of compaction tasks exceeds the predefined threshold, the compaction tasks are offloaded to the GPGPU with an expectation to fully utilize the GPGPU parallel computing resources. The large-sized values consume few GPGPU resources, which cannot fully exploit the advantage of parallel computing. Meanwhile, the performance bottleneck lies in disk I/Os in this case. In addition, GPGPU has an advantage in processing large-scale data; whereas for small-scale data, the time spent in triggering the GPGPU and allocating threads may be greater than the computation time. In this case, we perform compaction on the CPU, and the GPGPU utilization is still low. We study the GPGPU utilization of gLSM and recorded the GPGPU utilization every 0.5 second. Figure 18 illustrates the GPGPU utilization over time under DB_Bench configured with a 16-B key, 128-B value, and 5 GB data volume.

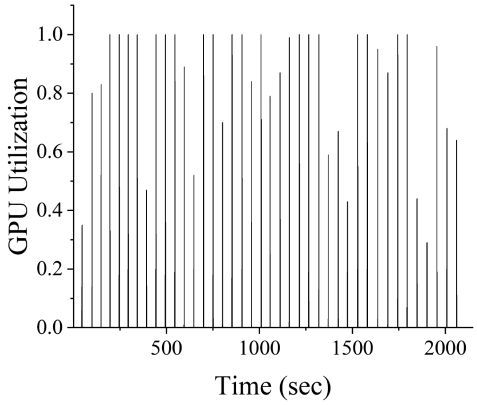


Fig. 18. The GPGPU utilization of gLSM under DB_Bench. *X*-axis represents the execution time, and *y*-axis denotes GPGPU utilization.

Table 10. Average GPGPU Utilization of gLSM under Workloads with Different Value Sizes

Value size (Bytes)	32	64	128	256	512	1024	4096
Average GPGPU utilization (%)	3.14	2.65	1.94	1.28	0.85	0.53	0.35

We drew the following observations from the results. First, gLSM has not yet hit its performance plateau with the available resources of GPGPU. This observation is expected because our design goal of gLSM to reserve a portion of GPGPU resources through optimization methods to cope with urgent applications depending on GPUs. By analyzing the instantaneous GPGPU utilization, we discover that the highest value of GPGPU utilization is 100%, which implies that the compaction tasks utilize all the GPGPU cores. GPGPU utilization during some periods, such as at 98 seconds, does not reach the peak: we record the GPGPU utilization at 0.5 seconds. In this case, the GPGPU may perform the computation, or the computation is to be completed. Because the GPGPU computation speed is high, the GPGPU utilization at this instant may appear in the moment before or after the record point. Second, our gLSM thoroughly utilizes the GPGPU resources once every a fixed period and it is unable to trigger GPGPU consistently under the workload. The GPGPU utilization is 0% when the CPU does not spark an offloading task: the GPGPU is sitting idle for a time period. gLSM conducts disk I/Os, CPU computation, and other processes. Thanks to fast GPGPU computation, most of the computation completes in GPGPU within one second; therefore, the peak GPGPU utilization does not last long. The compaction, disk I/Os, and CPU computation account for the most time. Our calculation method is to remove this portion of the time without triggering the GPGPU-side compassion tasks – and we merely calculate the time of invoking GPGPU-side compaction tasks. If we count the duration in which GPGPU does not perform compaction, the GPGPU utilization is reduced such as only 10% or less.

We also measured the average GPGPU utilization of gLSM under the workloads with different value sizes, see Table 10.

As shown in Figure 16(d), when the value size is smaller than 512B, there are a large number of KV pairs in an SSTable. GPGPU computing resources are used to perform coding and sorting operations for these KV pairs. A powerful GPGPU's throughput becomes higher than a low-power GPGPU. Table 10 shows that the average GPGPU utilization of gLSM is low. In Figure 18, the GPGPU is idle at some periods of time, but the maximum GPGPU utilization of gLSM reaches 100%. The peak GPGPU utilization is much higher than the average one because gLSM experiences

Table 11. Impact of 100-GB Data Volume on Performance of gLSM

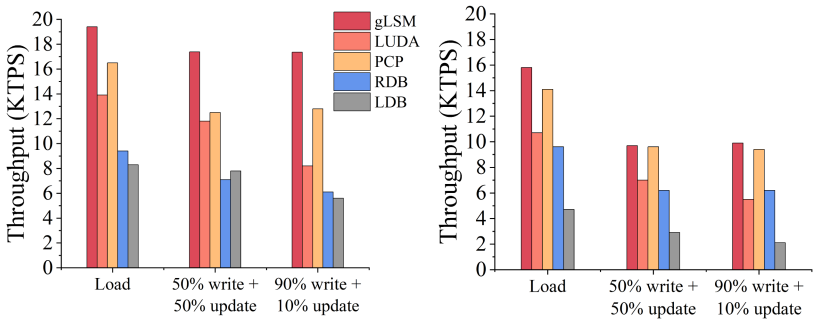
	gLSM-GPGPU	gLSM-CPU	LUDA-CPU	LUDA-GPGPU	PCP	RDB	LDB
Compaction Throughput (MB/s)	1004.2	126.8	83.9	173.6	74.4	102.9	32.1

an array of computation-intensive phases. A low-powerful GPGPU, however, may not be able to provide a high performance for data-intensive computational tasks in our system. Therefore, gLSM benefit from a powerful GPGPU that delivers high performance during computation-intensive phases.

However, when the value size is large (e.g., larger than 512 B), the throughput of the low-powerful GPGPU exceeds that of the more powerful one. This is because the amount of data offloaded to the GPGPU becomes small under workloads with large-sized values. In this case, the parallel computing resources in the GPGPU cannot be fully utilized. The time consumed by GPGPU in resource allocation exceeds the performance improvement brought by the GPGPU computation. Therefore, it is impossible to fully utilize parallel resources in the more powerful GPGPU device; however, we benefit from the less powerful GPGPU. In Figure 16(d), we observe that gLSM powered by a variety of optimization methods slashes GPGPU utilization – gLSM still carries out high-efficient compaction tasks without impairing the system performance even facing a low-performance GPGPU.

Impact of Large-sized Data Volume on Performance of gLSM. In this experiment, we study the impact of large-sized data volume on the performance of tested KV stores in terms of compaction throughput. We employed workload Load (100% write) in YCSB-C with 100-GB data volume, 1-KB value, and the default size key (key range is between 22B- and 24B). As listed in Table 11, gLSM significantly outperforms LUDA (83.9 MB/s), PCP (74.4 MB/s), RocksDB (102.9 MB/s), and LevelDB (32.1 MB/s) when the workload runs on CPU (126.8 MB/s). The experimental results show that gLSM achieves efficient implementation on the CPU. When compaction tasks are offloaded to the GPGPU, the performance of gLSM (1004.2 MB/s) outperforms all other KV stores under the workload on the GPGPU. The results reveal that gLSM fully exploits the parallel computing resources of GPGPU, and gLSM achieves significant performance improvement. We find that gLSM can offer superior performance on both CPU and GPGPU for data processing, which enables it to be apt for diverse computation-intensive applications.

Performance of gLSM under updated workloads. As shown in Figure 19, the experimental results show that gLSM achieves the best performance in terms of throughput under all workloads. The performance of all tested KV stores decreases when the data size increases from 5 GB to 50 GB; however, gLSM has a smaller decrement compared with alternative KV stores under workload with 90% write and 10% update. This is because our offloading compaction mechanism to the GPGPU improves the write performance and enables gLSM to achieve stable performance under the workload. Under 50-GB workload, the performance gLSM achieves an upward trend under 90% write and 10% update workload compared with that under 50%-write and 50%-update workload, see Figure 19(b). The results demonstrate our superior performance of gLSM. LUDA also offloads compaction tasks to the GPGPU, but its performance is lower than gLSM. This is because when triggering compaction, LUDA also performs merge operations on the CPU, which is the most time-consuming operation in compaction. Therefore, the compaction time of LUDA increases in the case of a large number of writes in workloads. The merge operations ultimately affect the performance of LUDA. For example, the performance of LUDA with a 90%-write and 10%-update workload is lower than that with a 50%-write and 50%-update workload. The results indicate that the slow compaction degrades the performance of LUDA. The performance of GPGPU-empowered gLSM



(a) 5 GB data volume (b) 50 GB data volume

Fig. 19. Performance of gLSM under updated workloads.

Table 12. Characteristics of YCSB-C used in this Experiment

YCSB-C	Load	A	B	C	D	E	F
Parameters	100% writes	50% Reads, 50% Updates	95% Reads, 5% Updates	100% Reads	95% Reads, 5% Inserts	95% Range Queries, 5% Inserts	50% Reads, 50% Read-Modify-Writes
KV pairs	Key size: 24 B; Value size 1 KB; Count: 50,000,000						

outperforms PCP with CPU multi-thread under 5-/50-GB workloads. This is because GPGPUs can provide sufficient computing resources, and our techniques in gLSM can meet the performance requirements of KV stores when dealing with large-scale data.

Performance of gLSM under YCSB-C. In this section, we study the performance of gLSM, LUDA, PCP, RocksDB, and LevelDB under YCSB-C with 24-B key, 1-KB value, and 50-GB data volume. The default configuration parameters of RocksDB and LevelDB are configured in the experiment. There are six types of workloads in YCSB-C, the characteristics of which are listed in Table 12. We use the Intel Optane 900P Series PCIe SSD with 480 GB as the storage device.

As shown in Figure 20, gLSM presents a higher performance than LUDA, also designed based on GPGPU. Compared with LUDA, gLSM improves the throughput by 47.7%, 22.8%, 91.1%, 79.6%, 56.6%, 200%, and 86% under workloads Load, A, B, C, D, E, and F, respectively. This is because the latency of CPU-based merge operations in LUDA is higher than that of the GPGPU-empowered radix-sort in gLSM, which increases compaction time and disk I/Os. The read I/Os must wait for the compaction to complete; therefore, the waiting time for the read I/Os increases. However, gLSM shortens the waiting time for reads from the disk and lessens the impact of compaction on read performance. The impact of different sorting algorithms on performance also exists in write- and read-intensive workloads.

Under read-intensive workloads (e.g., B and C), gLSM and RocksDB present similar performance, and the difference between the two KV stores is only 0.3 KTPS. This is because gLSM primarily improves compaction and write performance, whereas RocksDB offers techniques (e.g., block cache) to optimize the read performance. Then, RocksDB can read data from memory instead of the disk, thereby improving read performance. However, gLSM still achieves better performance under read-intensive workloads. For example, gLSM outperforms RocksDB by 3.6%, 3.2%, and 12.2% under workloads B, C, and D, respectively. The reason is that gLSM accelerates the compaction by offloading compaction to GPGPU, and high-performance compaction can reduce the disk I/O latency. This allows us to reduce storage costs and improve read performance. Meanwhile, the high-speed data transmission of PCIe SSD alleviates the I/O bottleneck, which lowers the compaction time of the gLSM and alleviates the I/O competition with reads.

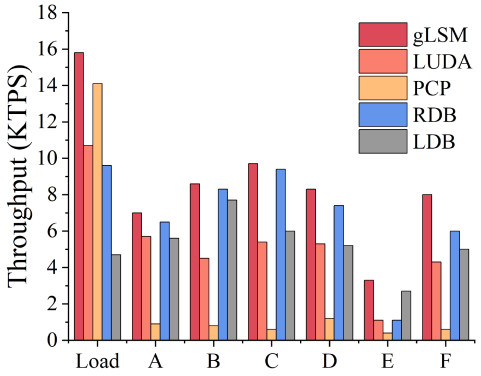


Fig. 20. The throughput of gLSM under YCSB-C.

For mixed read-write workloads (e.g., A and F), the compaction mechanism of gLSM improves compaction performance, reducing high-latency disk I/Os. These mixed read and write operations complete quickly, lowering data processing latency. High-concurrent applications issue read and write requests simultaneously. In this case, LSM can efficiently process data and keeps a stable performance, avoiding performance degradation caused by compaction. For example, under workload A, gLSM improves throughput by 22.8%, 7.7%, and 25% over LUDA, RocksDB, and LevelDB, respectively. Under F, gLSM achieves higher throughput than LUDA, RocksDB, and LevelDB by 86%, 33.3%, and 60%, respectively. It outperforms PCP by 12.3× in terms of throughput.

Under workload E, gLSM maintains excellent write, read, and range-query performance. For the range-query performance, gLSM outperforms LUDA, RocksDB, and LevelDB by 200%, 200%, and 22.2%, respectively. Especially, gLSM has 7.2× higher than PCP in terms of throughput. This is because gLSM enhances the compaction performance through GPGPU parallel computing resources. The fast-compaction procedure can quickly merge duplicated KV pairs on the disk, making the data orderly, which is favorable for the range query. The read performance becomes high when the data is ordered.

7 CONCLUSION

In this article, we addressed the problem of resource contention in CPU-sided and multi-threaded schemes, in which compaction performance is unacceptably low. We discovered that multi-threaded techniques have limitations when dealing with large-scale data volumes. After analyzing the changes in time spent on CPU and I/Os while varying the value size, we pointed out a computational bottleneck - motivating us to design gLSM as an approach to offloading compaction tasks to the GPGPU to accelerate compaction. We proposed a KV store gLSM anchored on GPGPU-accelerated compaction that is capable of operating in parallel with the CPU. We designed a parallel interaction subsystem for the heterogeneous CPU and GPGPU pairing and; then, we implemented in gLSM a collaborative driver to fully utilize ultra-parallel compaction units in the GPGPU. An asynchronous transfer method and key-value separation mechanism to achieve high-efficiency compaction task offloading was also devised. We compared gLSM against the state-of-the-art KV stores under the workloads with various data volumes. gLSM improves the throughput by up to a factor of 2.9 and cuts down write latency by 73.3%. In addition, the compaction bandwidth is enhanced by a factor of 26 over the other KV stores. The KV separation and collaboration-sort modules assist gLSM to ramp up the performance by 45% over the case in which these modules are disabled in gLSM.

REFERENCES

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* 40, 1 (2012), 53–64.

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the USENIX Annual Technical Conference*. 363–375.

[3] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking memory in persistent key-value stores. In *Proceedings of the 12th European Conference on Computer Systems*. ACM, 80–94.

[4] 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 Transactions on Knowledge and Data Engineering* 34, 6 (2022), 2595–2609.

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

[6] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the 10th European Conference on Computer Systems* 1–14.

[7] Google. 2017. LevelDB, LevelDB is a fast key-value storage. Retrieved March 30, 2023 from https://github.com/google/leveldb

[8] Linh Ha, Jens Krüger, and Cláudio T. Silva. 2009. Fast four-way parallel radix sorting on GPUs. *Computer Graphics Forum* 28, 8 (2009), 2368–2378. DOI: https://doi.org/10.1111/j.1467-8659.2009.01542.x

[9] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, and Tor M. Aamodt. 2012. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software*. 88–98.

[10] Christopher Hollowell, Costin Caramarcu, William Strecker-Kellogg, Antonio Wong, and Alexandr Zaytsev. 2015. The effect of numa tunings on cpu performance. In *Proceedings of the Journal of Physics: Conference Series*. IOP Publishing, 092010.

[11] 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.

[12] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the USENIX Annual Technical Conference*. 173–187.

[13] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture*. 373–384.

[14] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of the USENIX Annual Technical Conference*. 993–1005.

[15] Colin Ian King. 2020. Stress-ng. Retrieved March 15, 2023 from https://github.com/ColinIanKing/stress-ng/.

[16] Christoph Lameter. 2013. NUMA (Non-uniform memory access): An overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue* 11, 7 (2013), 40–51.

[17] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 1195–1206.

[18] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. 2019. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the USENIX Annual Technical Conference*. 395–410.

[19] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 133–148.

[20] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*. 477–489.

[21] Nvidia. 2012. Nvidia-smi - NVIDIA System Management Interface program. Retrieved February 13, 2023 from https://developer.nvidia.com/nvidia-system-management-interface.

[22] Nvidia. 2016. Single-pass Parallel Prefix Scan with Decoupled Look-back. Retrieved February 15, 2023 from https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf.

[23] Nvidia. 2020. A Faster Radix Sort Implementation. Retrieved February 20, 2023 from https://developer.download.nvidia.cn/video/gputechconf/gtc/2020/presentations/s21572-a-faster-radix-sort-implementation.pdf.

[24] Nvidia. 2022. NVIDIA TESLA K80. Retrieved February 25, 2023 from https://www.nvidia.cn/data-center/tesla-k80/

[25] Nvidia. 2022. NVIDIA TESLA P100, The world's first AI supercomputing data centre GPU. Retrieved March 11, 2023 from https://www.nvidia.com/en-us/data-center/resources/pascal-architecture-whitepaper/.

[26] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325.
[27] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.
[28] Pradeep Shetty, Richard P. Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 17–30.
[29] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a file system with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. 485–498.
[30] Hui Sun, Wei Liu, Jianzhong Huang, Song Fu, Zhi Qiao, and Weisong Shi. 2019. Near-data processing-enabled and time-aware compaction optimization for LSM-tree-based key-value stores. In *Proceedings of the 48th International Conference on Parallel Processing*. 5:1–5:11.
[31] Hui Sun, Wei Liu, Jianzhong Huang, and Weisong Shi. 2018. Co-KV: A collaborative key-value store using near-data processing to improve compaction for the LSM-tree. arXiv preprint arXiv:1807.04151 (2018).
[32] Hui Sun, Wei Liu, Zhi Qiao, Song Fu, and Weisong Shi. 2018. DStore: A holistic key-value store exploring near-data processing and on-demand scheduling for compaction optimization. *IEEE Access* 6 (2018), 61233–61253. DOI: 10.1109/ACCESS.2018.2873579
[33] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. FPGA-based compaction engine for accelerating LSM-tree key-value stores. In *Proceedings of the 2020 IEEE 36th International Conference on Data Engineering*. 1261–1272.
[34] Chenlei Tang, Jiguang Wan, and Changsheng Xie. 2022. FenceKV: Enabling efficient range query for key-value separation. *IEEE Transactions on Parallel Distributed System* 33, 12 (2022), 3375–3386.
[35] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. 2014. GDM: Device memory management for gpgpu computing. In *Proceedings of the ACM Int. Conf. Measurement and Modeling of Computer Systems*. 533–545.
[36] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the European Conference on Computer Systems*.
[37] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. 2018. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition* . 563–568.
[38] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the Usenix Annual Technical Conference*. 71–82.
[39] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of the USENIX Annual Technical Conference*. 349–362.
[40] 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).
[41] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building efficient key-value stores via a lightweight compaction tree. *ACM Transactions on Storage* 13, 4 (2017).
[42] 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 *Proceedings of the USENIX Annual Technical Conference*. 17–31.
[43] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. 2017. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel Distributed System*. 28, 4 (2017), 961–973.
[44] 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 *Proceedings of the USENIX Conference on File and Storage Technologies*. 225–237.
[45] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. 2014. Pipelined compaction for the LSM-tree. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* 777–786.
[46] Xingsheng Zhao, Song Jiang, and Xingbo Wu. 2021. WipDB: A write-in-place key-value store that mimics bucket sort. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering*. 1404–1415.

Received 14 March 2023; revised 1 August 2023; accepted 15 November 2023