A GPU-accelerated Compaction Strategy for LSM-based Key-Value Store System

Hao Zhou, Yuanhui Chen, Lixiao Cui, Gang Wang* and Xiaoguang Liu*

College of Computer Science, TMCC, SysNet, DISSec, GTIISC, Nankai University, Tianjin, China

{*zhouh, chenyh, cuilx, wgzwp, liuxg*}*@nbjl.nankai.edu.cn*

***Abstract*—Key-value storage systems based on LSM-tree exhibit superior write performance, making them a popular choice as the underlying storage engine for various Internet applications. However, compaction operations, responsible for maintaining the pyramidal storage structure of the LSM-tree to ensure acceptable read performance, pose significant performance bottlenecks. As high-performance storage devices like NVMe SSD are integrated into LSM-tree, the KV storage system obtains a huge performance improvement. Meanwhile, these fast I/O devices magnify the computational resource consumption of compaction when KV is small and medium, shifting the bottleneck from I/O to computation.**

In this paper, we focus on leveraging GPU to accelerate the compaction of LSM storage engines built on high-performance SSDs when the size of KV is small and medium, eliminating the computational bottleneck of compaction. Specifically, we design efficient GPU compaction units for each process of compaction, introducing a hierarchical acceleration strategy for different compaction levels. Additionally, We design two SSD-GPU data transfer mechanisms, the Pipeline mechanism, and the P2P mechanism, to minimize the data transfer overhead during compaction. We implement the proposed GPU-accelerated compaction strategy based on LevelDB and compare its performance with naive CPU compaction strategy and the state-of-the-art GPU-accelerated Compaction method, *LUDA*. Compared to CPU-based method and LUDA, the compaction performance is improved by up to 3.61x/2.24x, the write throughput is improved by up to 2.34x/1.51x and the mixed read/write throughput is improved by up to 2.02x/1.30x, respectively.

***Index Terms*—Key-value store, LSM-tree, Compaction, GPU, GPUDirect Storage**

I. Introduction

The Log-Structured Merge tree (LSM-tree) [1] stands out as one of the predominant persistent Key-Value (KV) storage structures in modern storage systems. It serves as the underlying storage engine in various contemporary systems, owing to its exceptional write performance. LSM-tree-based KV storage systems are widely employed in various internet applications, such as online e-commerce services [2], social networks [3], and more. Common LSM-tree-based KV storage systems, such as LevelDB [4], RocksDB [5], and Cassandra [6], maintain a tiered, multi-level ordered storage structure on persistent storage devices like HDDs or SSDs. In this structure, new data is initially written to the upper-level. When the upper-level approaches its capacity threshold, a *compaction* operation is triggered. This operation involves merging KV from two adjacent levels into the next level. *Compaction* is to sustain the pyramidal storage structure of the LSM-tree, ensuring acceptable read performance, and managing storage costs by performing garbage collection for expired and deleted data.

Compaction operations represent a significant performance bottleneck in LSM-tree-based (KV) storage systems [7]–[9]. On one hand, untimely compaction can block the foreground thread's write operations, leading to a decrease in writing performance, called *write stalls*. On the other hand, *compaction* itself is resource-intensive, involving substantial I/O and computational overhead. It requires reading numerous Sorted Strings Table files (SST files, containing key/value string pairs in LSM-tree) from the storage device, parsing KV pairs from all files, reordering all KV pairs, and then generating new SST files before writing them back to the storage device. Consequently, *compaction* competes fiercely for resources with write and read requests, undermining the system's overall throughput.

In this paper, we mainly focus on accelerating compaction, reducing the blocking time of the writing operations of the foreground threads, and alleviating the write stalls. In recent years, the emergence of high-performance storage devices (such as SATA SSD, NVMe SSD) has brought a huge performance improvement to the KV storage systems [10], [11]. The application of these storage devices to the LSM-tree storage engine can greatly alleviate its I/O bandwidth bottleneck but also magnifies the computational resource consumption of compaction.

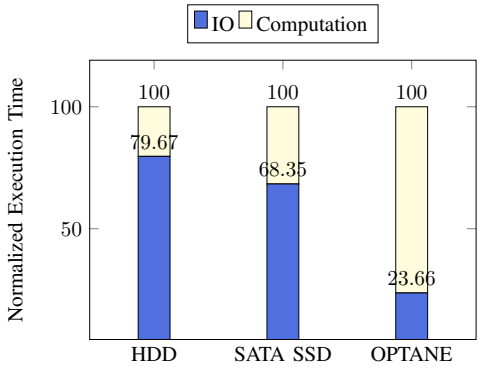


Fig. 1: The breakdown of LSM Compaction based on three storage devices (Value Size = 32)

Fig. 1 shows the profiling of compaction operation in LevelDB [4] from IO and computation for different types of storage devices. With HDD, SATA SSD, and NVMe SSD (INTEL OPTANE SSD) as back-end storage devices, the computation time percentage of compaction are 20.33%, 31.65%, and 76.34%, respectively. This observation underscores that, on high-performance storage devices, the bottleneck for *compaction* has shifted towards computation.

Further, we profile the *compaction* task for various value sizes on the high-performance SSD (i.e., INTEL OPTANE SSD), as depicted in Fig. 2. For smaller value sizes, such as 32 bytes, 64 bytes, computation consumes close to 70% of the time. In the case of medium-sized KV pairs (128 bytes and 256 bytes), computation accounts for approximately 60% of the total time. This analysis indicates a significant potential for accelerating the computation in *compaction*. However, as the value size increases, the percentage of IO operation (i.e., reading and writing SST files) time grows, and *compaction* becomes IO-bound. Consequently, accelerating the computation in *compaction* becomes less effective in these scenarios.

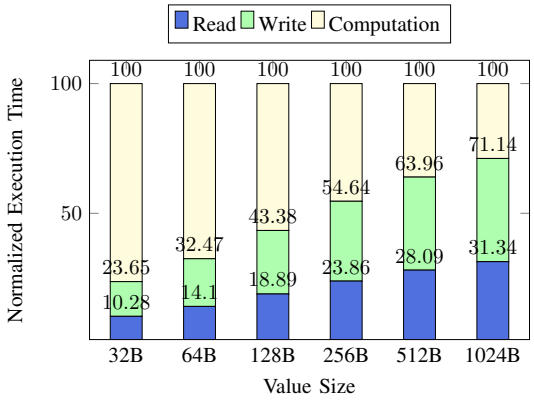


Fig. 2: The breakdown of LSM Compaction based on INTEL OPTANE (Key Size = 16 bytes)

Therefore, we focus on accelerating compaction in LSM storage engines built on high-performance commercial SSDs when dealing with small to medium-sized KV, eliminating the computational bottleneck of compaction. Prior work [9], [12] have emphasized that using multiple threads to speed up compaction can heavily consume CPU resources, leading to competition with foreground read/write requests and an overall performance degradation. To overcome these challenges, hardware acceleration methods such as FPGA [9] and DPU [12] have been proposed. However, programming FPGAs can be challenging, and DPUs might lack the computational power needed for compaction task. GPUs possess significant computational power, hardware concurrency, and a programming-friendly environment, making them well-suited for handling the compaction task efficiently. Notably, LUDA [13] introduced GPU acceleration for compaction, although it did not fully leverage the GPU for the entire *Compaction* process. Instead, it still relied on the CPU for sorting key-value pairs, resulting in high CPU-GPU data transmission overhead. Additionally, it did not effectively address the data transmission overhead between SSD and GPU, limiting its acceleration impact. Moreover, GPUs have been widely utilized to accelerate KV storage systems and databases [14], [15]. For instance, Mega KV [14] maximized the high memory bandwidth and latency-hiding capabilities of GPUs, achieving a high-performance and high-throughput in-memory KV system. OurRocks [15] directly offloaded scan operations to the GPU in a write-optimized database system, accelerating analytic queries. This highlights the potential and value of GPU-accelerated databases and storage systems.

Utilizing GPUs to accelerate compaction poses two primary challenges: **1) Efficient Compaction Unit Design.** Designing efficient compaction units is essential for swift compaction operations and harnessing the full potential of hardware acceleration. **2) Dealing additional Data Transfer Overhead.** The introduction of GPUs adds additional complexity in data transfers. When GPU is integrated into the compaction process, data must be transferred from the storage device to GPUs for processing. Subsequently, the computed results need to be transferred back from the GPU to the storage device. Reducing the overhead associated with moving data between storage and GPUs, is essential for achieving an efficient compaction process.

Building upon these challenges and insights, our proposal focuses on leveraging GPUs to accelerate LSM compaction. Our contributions can be outlined as follows:

- We design efficient GPU compaction units for each stage of compaction, including units for parsing key-value pairs from SST files, parallel sorting of key-value pairs, and generating new SST files (encompassing data blocks, the index block, and the Bloom filter block).
- We design a hierarchical acceleration strategy for the compaction tasks at different levels. For the upper-level compaction tasks (i.e., L0→L1, L1→L2), we introduce a quick compaction strategy (Q-Compaction), where all compaction processes are executed on the GPU to ensure quick merging of the upper-level SST files. For the lower-level compaction tasks, we propose a CPU-GPU cooperative compaction strategy (C-Compaction). Here, the GPU accelerates the parsing and sorting of key-value pairs, while the CPU performs garbage collection for expired and deleted key-value pairs. Subsequently, the garbage-collected key-value pairs are again passed to the GPU for generating the new SST files.
- We design two data transmission mechanisms aimed at reducing data transfer overhead: the *Pipeline mechanism* and the *SSD-GPU P2P mechanism*. The *Pipeline mechanism* utilizes CUDA streams to overlap data transmission and computing operations. The *SSD-GPU P2P mechanism* utilizes Nvidia GPUDirect Storage [16] to establish a direct data transfer path between GPU memory and PCIe SSD, eliminating redundant data transfers.
- We implement the GPU-accelerated compaction based on LevelDB and compare the performance of our proposed strategy with naive CPU compaction strategy and

the state-of-the-art GPU-accelerated Compaction method LUDA. Notably, our acceleration strategy is adaptable to any LSM-Tree storage system. Compared to CPU-based method and LUDA, our compaction performance demonstrates improvements of up to 3.61x/2.24x, write throughput enhancements of up to 2.34x/1.51x, and mixed read/write throughput improvements of up to 2.02x and 1.30x, respectively.

II. Background

A. LevelDB, LSM-Tree and Compaction

LevelDB is a persistent key-value storage system with LSM-Tree as the underlying storage engine. As shown in Fig. 3, the new data is first written to a data structure in memory (the SkipList, called *Memtable*), which switches to a read-only mode (called *Immutable Memtable*) when the *Memtable* reaches its capacity threshold. The background thread serializes the *Immutable Memtable* into a *Sorted String Table* file (SST file) and writes it to the top-level structure of the LSM Tree, which is called a *Flush* operation.

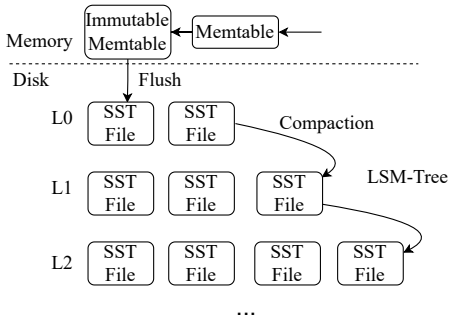


Fig. 3: The architecture of LevelDB

LSM is a pyramidal, multi-level ordered storage structure, with the size of each level growing exponentially. Once the upper-level structure reaches the capacity threshold, an operation called *compaction* is triggered which selects the overlapping SST files of two adjacent levels, parses all the key-value pairs in the files, and reorders them, then generates new SST files and writes them to the next level.

B. SST file

The SST file is persistent in the LSM-tree, which stores an ordered sequence of key-value pairs. The layout of the SST file is shown in Fig. 4. SST File consists of three types of blocks: the Data block, the Bloom Filter block, and the Index block. The key-value pairs are formed into *Datablock* in a "prefix-compressed" format. Data blocks are independent of each other. Further, the data block is organized into separate groups (KV Group). Prefix compression means that the adjacent keys in each group share the same prefix to save storage space. The first key in a key-value group is called the *restart point*, and the *restart point* does not share prefixes with other keys. A new restart point will be created for every fixed number of key-value pairs, and this fixed number is called the *restart point interval*. For example, *KV1* in Fig. 4 is the *restart point*, and the *restart point interval* is 4.

Bloom filter [17] is essentially a bit vector and several hash functions. When inserting a key, the key is first hashed by using the hash functions to obtain multiple hash values, and the bits in the bit vector with the corresponding indexes are set to 1. When searching, the Bloom filter will check whether the corresponding bits are 1. If any of these bits is 0, the key must not exist. If all of them are 1, the element may exist. The Bloom filter used in LevelDB is to speed up the query of key-value pairs. If the Bloom filter determines that the key-value pair to be queried does not exist, there is no need to search for the corresponding data block.

The Index block in the SST file is used to store the index information of all data blocks and is still stored in the form of key-value pairs. The number of key-value pairs in the Index block is equal to the number of data blocks. The key in the index block is the largest one in the corresponding data block, and the value is the offset and length of the corresponding data block in the SST file. Different from the data block, the *restart point interval* in the index block is fixed at 1, that is, each key-value pair is an independent restart point.

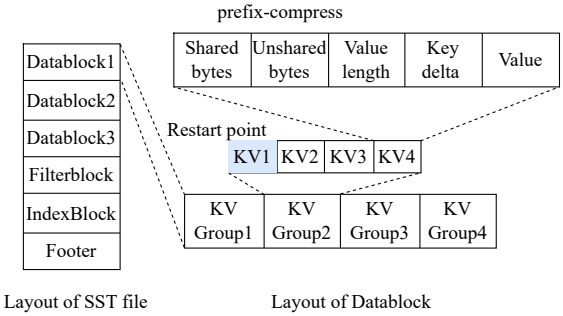


Fig. 4: The layout of SST file and Datablock

Compaction is divided into four steps: 1) Parse key-value pairs. This step deserializes the SST files and parses all the key-value pairs. Parsing key-value pairs is a highly parallel task, matching the GPU. 2) Sort key-value pairs. This step sorts key-value pairs with overlapping ranges to keep them completely ordered. 3) Merge and garbage collect key-value pairs. In this step, LSM will merge and clean up the expired data for all key-value pairs to facilitate the generation of new SST files. 4) Generate new SST files and write them to the storage device. This process needs to generate three types of blocks: Data block, Bloom Filter block, and Index block.

C. Nvidia GPUDirect Storage

Nvidia GPUDirect Storage [16] (GDS) is a technology enabling a direct data transfer path between GPU memory and PCIe storage device based on Nvidia GPUDirect RDMA [18]. GDS transfers the data by direct memory access (DMA) without burdening the CPU. In GPU applications involving disk IO, data has to be loaded from the disk to the main memory and then transferred from the main memory to the GPU

memory. Redundant data transfers increase latency, especially for applications like compaction that require transferring large amounts of data. GDS applied to accelerate compaction will provide higher data throughput and better utilization of GPU resources.

III. Design

A. Overview

The design of GPU-accelerated compaction comprises three main components:

1) Efficient GPU Compaction Units: In the steps of compaction, there is a notable concurrency in the tasks of parsing key-value pairs, sorting key-value pairs, and generating new SST files, making them all suitable for GPU acceleration. As detailed in Section III-B, we have designed three GPU Compaction Units: *Unpack Unit*, *Sort Unit*, and *Pack Unit*. The *Unpack Unit* is to parse key-value pairs from SST files, the *Sort Unit* sorts all key-value pairs, and the *Pack Unit* generates new SST files, including data blocks, the index block, and the Bloom filter block.

2) Hierarchical acceleration strategy: In the compaction process, the merging and garbage collection of key-value pairs primarily involve serial logic, making them unsuitable for GPU acceleration. To address this, we propose two GPU-accelerated compaction strategies: Quick-Compaction (referred to as Q-Compaction) and Cooperative-Compaction (referred to as C-Compaction), as illustrated in Fig. 5.

Q-Compaction is designed for upper-level compaction tasks, specifically L0 to L1 and L1 to L2. The latency of these upper-level compaction tasks plays a crucial role in alleviating write stalls in LSM. To minimize latency, we leverage the GPU to accelerate all processes involved in these compaction tasks, including parsing key-value pairs, sorting, and generating SST files. To avoid impacting the performance of Q-Compaction, we deliberately exclude garbage collection for expired and deleted key-value pairs within Q-Compaction. This choice comes at the expense of a slight increase in storage space waste.

C-Compaction is assigned to handle the lower-level compaction tasks. In contrast to Q-Compaction, C-Compaction retains the garbage collection function inherent in the native compaction process. It first initiates GPU acceleration for two primary processes: parsing key-value pairs and sorting them. Subsequently, the sorted key-value pairs are transferred back to the main memory, where the CPU performs garbage collection for expired and deleted key-value pairs. The garbage-collected key-value pairs are then fed back to the GPU for the generation of new SST files. Finally, the freshly generated SST files are written to the storage device. Both Q-Compaction and C-Compaction share the same GPU Compaction Unit.

3) Data transmission mechanism: While the GPU can accelerate the computational processing of compaction, transferring SST files from SSD to GPU introduces new overhead. Efficient data transmission mechanism is crucial not only for reducing the time overhead of compaction tasks but also for making optimal use of GPU resources. We design two data transmission mechanisms based on hardware characteristics: the Pipeline mechanism and the SSD-GPU P2P mechanism, as detailed in Section III-C.

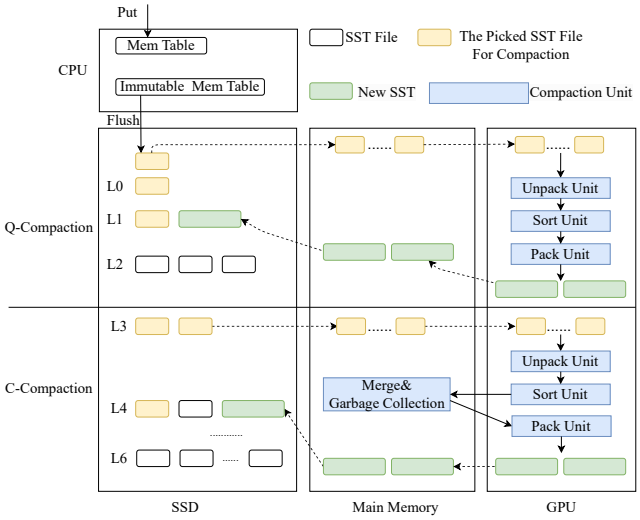


Fig. 5: The Hierarchical architecture of GPU-accelerated Compaction Strategy

B. GPU Compaction Units

1) Unpack Unit: As depicted in Fig.4, the SST file comprises multiple independent data blocks, each containing several distinct groups (KV groups). Within each KV group, adjacent keys share the same prefix. Consequently, the KV group serves as the smallest independent task unit. We employ a GPU thread to parse a KV group, with a one-dimensional thread block assigned to parsing a data block. A thread grid, comprising multiple one-dimensional thread blocks, concurrently parses key-value pairs in an SST file. To enhance the concurrency of unpack operations, CUDA Streams are utilized to launch multiple GPU thread grids, enabling the parsing of key-value pairs in multiple SST files simultaneously and achieving grid-level parallelism. An overview of the Unpack Unit is presented in Fig.6.

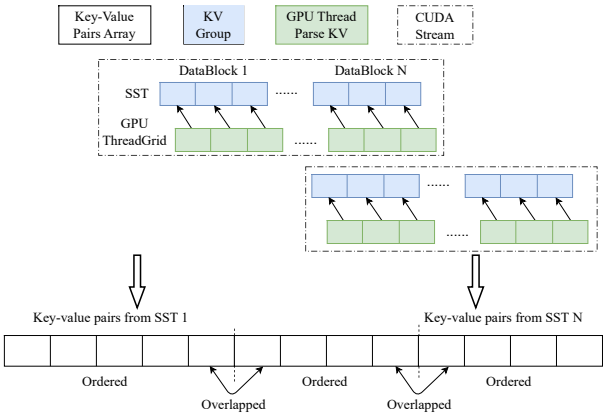


Fig. 6: The overview of Unpack Unit

All the parsed key-value pairs are placed into an array, where the key-value pairs from the same SST file are ordered, and key-value pairs from different SST files may overlap. In this array, the elements of the key-value pairs include the full key, not the complete value. Instead, the offset and size of the value in the corresponding SST file are included to economize on the cost of data copying in subsequent operations.

2) Sort Unit: In LSM, the keys in each SST file are ordered. The SST files belonging to the L_0 level obtained by the *Flush* operation are partially ordered, and there is an overlap in the range of keys between each SST file. The remaining levels (i.e., L_1, L_2, L_3, etc.) have ordered keys for the entire level. Therefore, after parsing the key-value pairs, the arrays need to be sorted to ensure that the newly generated SST files remain fully ordered, maintaining an acceptable read performance.

The sorting of key-value pairs in the compaction operation can be conceptualized as merging multiple ordered arrays, ultimately converging into a fully ordered array. We implement the merging of multiple ordered key-value pair arrays on the GPU using a binary search, as illustrated in Algorithm 1.

Algorithm 1 Merge multiple ordered key-value pair arrays

Input: Key-value pairs arrays $SST_0, SST_1, \ldots, SST_n$
Output: Fully ordered Key-value pairs array $Array$

1: **for** each GPU thread **do**
2: $idx = blockIdx.x \times blockDim.x + threadIdx.x$
3: $j =$ Current array Id
4: $KV^j_{idx} = SST_j[idx]$
5: $I_{Array} = 0$ ▷ Initialize the index of KV^j_{idx} in $Array$
6: **for** $i = 0$ to n **do**
7: **if** $(i == j)$ **then**
8: $I = idx$
9: **else**
10: $I = BinarySearch(KV^j_{idx}, SST_i)$
11: **end if**
12: $I_{Array} : += I$
13: **end for**
14: **end for**
15: $Array[I_{Array}] = KV^j_{idx}$
16: **return** $Array$

We assign a GPU thread to calculate the index of the corresponding key-value pair in the final key-value pair array, establishing a one-to-one correspondence between the key-value pair and the GPU thread. For the key-value pair KV^j_{idx} belonging to SST_j (Line 4), its index in SST_j is the same as the GPU thread's index (i.e., idx) (Line 8). We use binary search to obtain the indexes $I^0_{idx}, I^1_{idx}, \ldots, I^{j-1}_{idx}, I^{j+1}_{idx}, I^n_{idx}$ when KV^j_{idx} is inserted into other key-value pair arrays (i.e., $SST_0, SST_1, \ldots, SST_{j-1}, SST_{j+1}, SST_n$), respectively (Line 10). In this case, the index of KV^j_{idx} in the final array is $I_{Array} = \sum_{j=0}^{n} I^j_{idx}$ (Line 12). Binary search inherently exhibits parallelism. And the key stored in the SST file is called *Internal Key*, which consists of *User Key*, *Sequence Number* and *ValueType*. *Internal Key* is unique. Even if *User Key* is the same, the corresponding *Internal Key* is different because *Sequence Number* and *ValueType* are different. Therefore, the calculation of the final index for each key-value pair is independent. Consequently, we concurrently calculate the final index of all key-value pairs and write the results into the final array $Array$.

3) Pack Unit: As depicted in Fig. 4, the SST file comprises data blocks, an index block, a Bloom filter block, and a footer. Among these, the generation of data blocks, the index block, and the Bloom filter block is the most time-consuming process. Therefore, *Pack Unit* primarily focuses on the parallel generation of these components—data blocks, the index block, and the Bloom filter block.

- Generate data blocks.
  Data blocks are utilized to store the actual KV pairs and consist of independent KV pairs groups (KV group), where adjacent keys in the group share the same prefix. The number of KV pairs in a KV group is referred to as the *restart point interval*. Similar to parsing data blocks (as shown in Fig.6), generating a KV group remains the smallest independent task in the process of generating data blocks. Consequently, we still employ a GPU thread to generate a KV group, as illustrated in Fig.7. The process of generating a KV group involves each GPU thread obtaining the corresponding KV pairs as input according to the *restart point interval* and performing prefix compression on adjacent keys. A one-dimensional thread block can generate a data block, and a thread grid comprising multiple one-dimensional thread blocks can concurrently generate all data blocks of an SST file. Similarly, we use multiple CUDA Streams to launch multiple thread grids, enabling the concurrent generation of data blocks in multiple SST files.

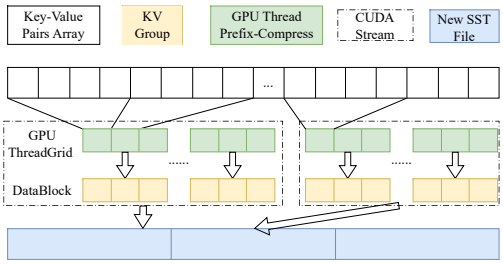


Fig. 7: The overview of the Pack Unit

- Generate the Index block
  The index block takes a special form when the *restart point interval* is set to 1, resembling a data block. The number of KV pairs in an index block equals the number of data blocks. In this scenario, the generation of an index block can be accomplished using a one-dimensional thread block. Each GPU thread calculates the largest key in the corresponding data block and encodes its offset and length into the value using variable-length encoding, thereby generating the index block.
- Generate the Bloom filter block

Algorithm 2 Generate Bloom Filter block

Input: Key-Value pairs Array $Array$, Number of hash functions K
Output: Bit Vector $BitVector$

1: **for** each GPU thread **do**
2: $idx = threadIdx.x$
3: **for** $i = 1$ to K **do**
4: $h = Bloomhash(Array[idx])$
5: $bytepos = h\%ByteVector\_Len$
6: $ByteVector[bytepos] = 1$
7: **end for**
8: _syncthreads() ▷ Synchronize GPU threads
9: $BitVector\_Len = (ByteVector\_Len + 7)/8$
10: **if** $(idx < BitVector\_Len)$ **then**
11: $base = idx \times 8$
12: **for** $j = 0$ to 7 **do**
13: $bytepos = base + j$
14: **if** $(ByteVector[bytepos] == 1)$ **then**
15: $BitVector[idx] : | = (1 << j)$
16: **end if**
17: **end for**
18: **end if**
19: **end for**
20: **return** $BitVector$

Generating the Bloom filter block in an SST file is also a highly concurrent task because the calculation of the hash functions for each key is independent. A one-dimensional thread block is responsible for generating a Bloom Filter block for all key-value pairs in a data block. The algorithm for generating a Bloom filter block is shown in Algorithm 2.

We utilize a GPU thread to calculate the K hash values of the corresponding key, obtain the hash positions, and store the results in a byte vector $ByteVector$ (Line 3 to 7), drawing inspiration from [19]. Here, a byte, rather than a bit, represents a Boolean value (i.e., 0 and 1). In this case, we don't need to adopt a GPU thread synchronization strategy. Even if multiple GPU threads modify the content of the same byte, there will be no conflict since they write the same value (i.e., 1). Subsequently, to conserve storage space, we use one-eighth of the GPU threads to convert the byte vector $ByteVector$ to a bit vector $BitVector$ (Line 9 to 18).

C. Reduce data transfer overhead between storage device and GPU

1) Pipeline mechanism: In the pipeline mechanism, we leverage CUDA streams to exploit the parallelism of three operations: disk I/O, main memory-GPU memory copy, and GPU computation. The overview of the *Pipeline mechanism* is illustrated in Fig. 8.

After the first SST file is read from the SSD to the Main memory, we use CUDA Stream to launch a Main memory-GPU copy operation and execute the GPU kernel. CUDA Stream operates asynchronously to the host, allowing us to read the next SST file and launch the second CUDA Stream after the first CUDA Stream is initiated. This overlapping of data transmission (i.e., Disk I/O and Main Memory-GPU copy) and GPU computation operations enhances throughput. Synchronization is necessary between different compaction processes (i.e., unpack, sort, and pack). In the *Pipeline mechanism*, newly generated SST files are written to the storage device after the SST files are copied from the GPU memory to the main memory. The pipeline mechanism is universal and imposes no specific requirements on storage devices and GPUs.

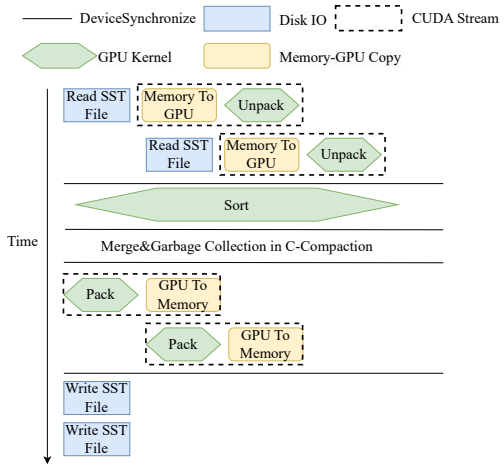


Fig. 8: The overview of Pipeline mechanism

2) SSD-GPU P2P mechanism: While the *pipeline mechanism* mitigates some of the data transfer overhead, the data transmission remains somewhat redundant. The SST file must traverse the main memory to reach the GPU memory. As outlined in Section 2.2, Nvidia GPUDirect Storage (GDS) establishes a direct data transfer path between GPU memory and the PCIe SSD when the GPU and the PCIe SSD share the same PCIe Root Complex. GDS executes data transfer via direct memory access (DMA), relieving the CPU from the burden. The data transmission path of GPU-accelerated compaction optimized with GDS is illustrated in Fig. 9.

The SST files will bypass the main memory and undergo direct transfer to the GPU memory. Similarly, the SST files generated in the GPU will be directly transmitted from the GPU memory to the PCIe SSD, eliminating redundant data transfers. Although the CPU retains control over GDS's file reading and writing operations, the data flow bypasses the CPU. GDS uses *direct IO* mode instead of *cache IO* mode when writing to the storage device. GDS has special requirements for GPU, and only Nvidia data center-level GPUs have this function. Moreover, GDS requires data center-level SSDs with PCIe interfaces, such as Intel OPTANE SSD.

D. Implementation

We implement GPU-accelerated compaction based on LevelDB 1.23 [4], focusing primarily on enhancing write perfor-

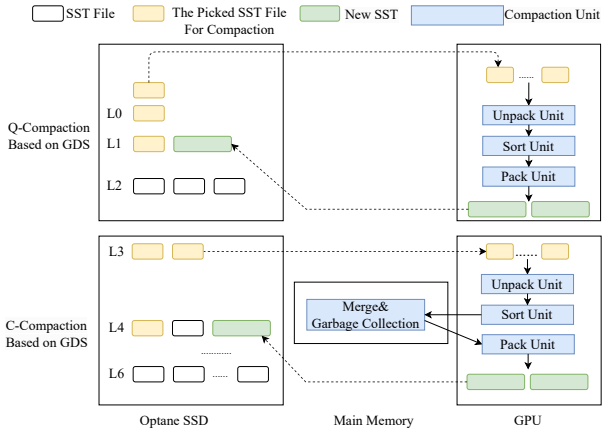


Fig. 9: The overview of GPU-accelerated Compaction Strategy based GDS

mance with minimal impact on read performance. To execute compaction processes on the GPU, we allocate essential resources, namely GPU memory and CUDA Streams. GPU memory is utilized for storing input SST files, parsed key-value pairs, newly generated SST files, and other metadata. However, the overhead associated with frequent allocation and release of GPU resources cannot be ignored. To address this, we adopt a *lazy allocation strategy*—allocating all GPU resources during key-value system initialization and deallocating them upon system closure. For the SSD-GPU P2P mechanism, we leverage the *cuFileRead* and *cuFileWrite* functions in GDS [16] 1.7.2 (built-in CUDA 12.2) to facilitate the transfer of SST files from the OPTANE SSD to the GPU and vice versa, enabling efficient data exchange.

IV. Experiments

A. Experimental Setup

We conduct our experiments on a server running Ubuntu 20.04.1 with the Linux kernel version 5.15.0. This server is configured with two NUMA nodes, each containing a CPU with 10 physical cores (Intel Xeon Silver 4210 @ 2.20GHz) and 128GB of DRAM. Additionally, it is equipped with various storage devices, including a 6TB SATA HDD (Seagate ST6000NM0115), a 480GB SATA SSD (INTEL SSDSC2BB48), and a 280GB NVMe SSD with a PCIe interface (INTEL OPTANE SSD 900P). Furthermore, the server features an Nvidia Quadro A6000 GPU with a 535.104.05 GPU driver, sharing the same PCIe Root Complex with the OPTANE SSD.

KV System Configurations. Both the *Memtable* and *SST file* sizes are configured to 8MB. The size of the *L1* Level is 64MB. The parameters related to the data block are as follows: *block_restart_interval* is 4, *block_size* is 32KB, and the number of bits of the Bloom filter bloom_bits is 10. Default settings are used for parameters controlling the write rate: *kL0_CompactionTrigger* is set to 4, *kL0_SlowdownWritesTrigger* to 8, and *kL0_StopWritesTrigger* to 12.

Workloads and Baselines. We employ *db_bench* with LevelDB to evaluate various compaction strategies. Specifically, we assess the performance across three workloads: **1) FillRandom:** Involves the random insertion of key-value pairs. **2) ReadRandom:** Focuses on randomly retrieving key-value pairs. **3) ReadWriteMix:** Involves both random retrieval and insertion of key-value pairs, based on a specified get/insert ratio. We primarily compare four compaction strategies: **1) CPU-based Compaction:** This serves as the default compaction strategy in LevelDB. **2) LUDA [13]:** Represents the state-of-the-art GPU-accelerated LSM strategy. **3) GPU Comp-Pipe:** Utilizes the *Pipeline mechanism* to minimize the cost of data transmission. **4) GPU Comp-GDS:** Implements GDS to directly read SST files from OPTANE SSD and write the generated SST files back to OPTANE SSD, aiming to enhance the throughput of data transmission. To mitigate the impact of Page Cache, we employ the *Direct IO* mode across all strategies.

B. Evaluate Fillrandom workload

In this workload, we insert 200M key-value pairs with keys uniformly distributed. The key size is fixed at 16 bytes. We manipulate the size of the value in the key-value pairs to create various experimental configurations. Specifically, the size of the value ranges from 32 bytes to 256 bytes.

For each configuration, we maintain the number of levels in the LSM-tree at 5. This ensures a comprehensive evaluation of the proposed hierarchical acceleration strategy. We conduct an analysis across several metrics, including compaction throughput, average throughput, real-time throughput, workload latency, and GPU utilization.

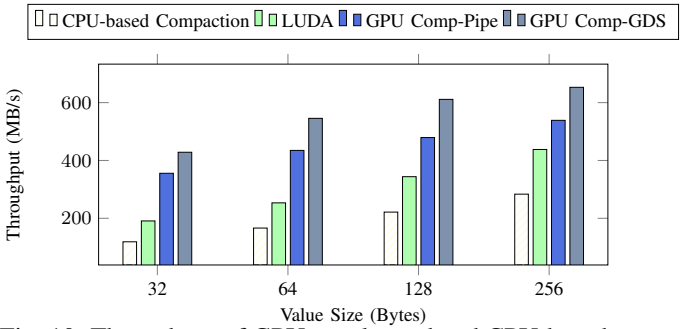


Fig. 10: Throughput of GPU-accelerated and CPU-based compactions with different KV settings

1) The Throughput of Compaction: We measure the compaction throughput by considering the quantity of SST files compacted per unit of time [20]. Fig. 10 illustrates the compaction throughput of the four strategies across various experimental configurations.

LUDA [13] exclusively leverages the GPU for accelerating the parsing of key-value pairs and the generation of SST files. This implies that the parsed key-value pairs need to be transferred back to the main memory, where the CPU performs sorting and performs garbage collection for expired and deleted data. Subsequently, the ordered and garbage-collected key-value pairs are transferred back to the GPU

for SST file generation. This process introduces two redundant copies of data between the main memory and GPU. Additionally, *LUDA* relies on the main memory as an intermediate buffer for transferring SST files between the SSD and GPU. Specifically, SST files are initially read from the SSD to the main memory and then transferred from the main memory to the GPU, incurring redundancy. As a consequence, *LUDA* achieves a performance higher than *CPU-based Compaction* by 1.61x/1.52x/1.55x/1.55x when the value size is 32/64/128/256 bytes, respectively. Nevertheless, there remains room for improvement in acceleration.

Both *GPU Comp-Pipe* and *GPU Comp-GDS* employing a hierarchical acceleration strategy, including *Q-Compaction* and *C-Compaction*. *Q-Compaction* executes all compaction units (i.e., parsing key-value pairs, sorting, and generating new SST files) on the GPU, ensuring the minimum compaction latency. *C-Compaction*, similar to *LUDA*, involves CPU participation but only in the garbage collection process, with the sorting process being handled on the GPU, resulting in improved efficiency compared with *LUDA*. This design eliminates some unnecessary data transfer overhead, contributing to enhanced GPU-accelerated compaction performance.

Notably, *GPU Comp-GDS* utilizes GDS for P2P transfer of SST files from the SSD to GPU memory, mitigating redundant data transfer costs and achieving higher throughput, especially with larger value sizes (e.g., 128 Bytes, 256 Bytes). Consequently, *GPU Comp-GDS* outperforms *CPU-based Compaction* by 3.61x/3.28x/2.76x/2.30x when the value size is 32/64/128/256 bytes, respectively. Moreover, *GPU Comp-GDS* achieves 2.24x/2.15x/1.78x/1.49x higher throughput compared to *LUDA*, respectively. The gains from GDS are evident, with *GPU Comp-GDS* improving by 61.28%, 66.70%, 59.65%, and 40.29% over *GPU Comp-Pipe*, respectively.

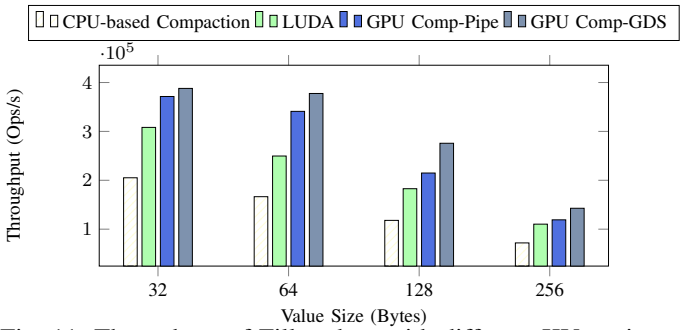
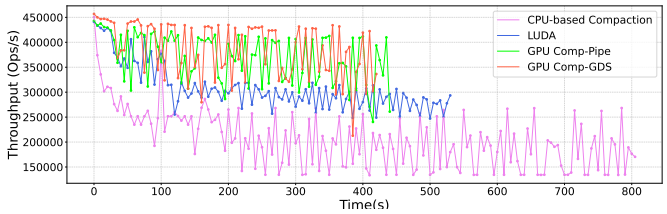
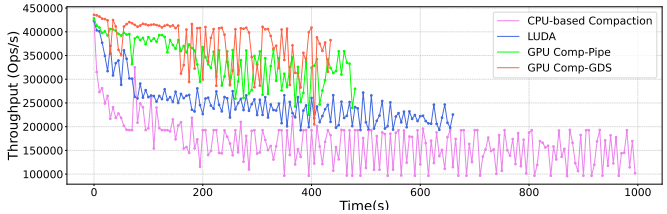


Fig. 11: Throughput of Fillrandom with different KV settings

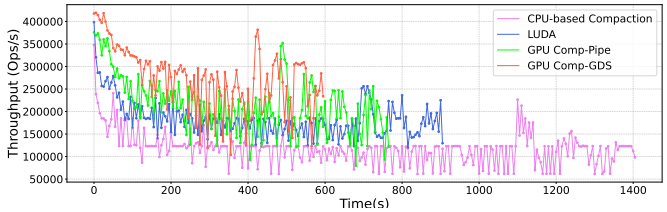
2) Average Fillrandom Throughput Analysis: In Fig. 11, the average throughput (i.e., Operations written per second) of random writes is depicted for the four compaction strategies across different experimental configurations. The enhancement in random write throughput is attributed to the improvement in compaction throughput. Notably, when the value size is relatively small (e.g., 32 Bytes, 64 Bytes), the SST file can accommodate more key-value pairs, allowing for a larger proportion of computation during compaction and providing more room for acceleration. In such scenarios, GPU-accelerated compaction strategies achieve significant acceleration. As the value size increases (e.g., 128 Bytes, 256 Bytes), the costs associated with disk IO and data transfer between main memory and GPU rise, reducing the effectiveness of acceleration. However, *GPU Comp-GDS* demonstrates further improvements in system performance through the P2P data transfer mechanism, particularly in cases with larger value sizes. Consequently, *GPU Comp-GDS* achieves 1.89x/2.27x/2.34x/1.99x higher throughput than *CPU-based Compaction* and 1.25x/1.51x/1.51x/1.29x higher throughput than *LUDA*, respectively. The gains from GDS are substantial, with *GPU Comp-GDS* improving by 8.06%, 21.99%, 51.61%, and 32.94% over *GPU Comp-Pipe*, respectively.



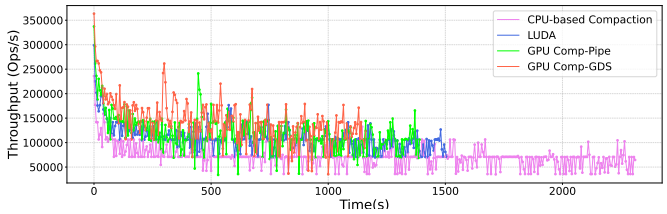
(a) value-size: 32 Bytes



(b) value-size: 64 Bytes



(c) value-size: 128 Bytes



(d) value-size: 256 Bytes

Fig. 12: The Real-time Throughput of Fillrandom workload

3) Real-time Throughput Analysis: Fig. 12 shows the real-time throughput of random writes for the four strategies. Across all experimental configurations, *CPU-based Compaction* takes the longest time to finish writing 200M key-value pairs, while *GPU Comp-GDS* takes the shortest time to

run. And *CPU-based Compaction* exhibits periodic instances of write pauses, where the real-time throughput drops significantly, attributed to untimely compaction blocking foreground threads. The purpose of GPU-accelerated compaction is to reduce the delay of compaction by accelerating compaction and reducing the blocking time of compaction on the foreground write thread, alleviating write pauses.

In GPU-accelerated compaction strategies (*LUDA*, *GPU Comp-Pipe*, and *GPU Comp-GDS*), the maximum allowable number of SST files in compaction is set to 40 due to GPU resource constraints. If the number of SST files exceeds 40, the compaction switches to a CPU-based approach, introducing some slight fluctuations in the throughput of GPU-accelerated compaction strategies. Additionally, throughput fluctuations in *GPU Comp-Pipe* and *GPU Comp-GDS* arise from the conversion of *Q-Compaction* and *C-Compaction*. Overall, *GPU Comp-GDS* exhibits consistently high overall throughput, and even its lowest throughput surpasses the baselines by a considerable margin. The rapid compaction in *GPU Comp-GDS* ensures minimal write pauses, making it the most stable and high-performing strategy.

4) Latency Analysis: Table I presents the write latency, including Average Latency (Avg), P99 Tail Latency (P99), and P99.9 Tail Latency (P999), with a key size of 16 bytes. Tail latency is a critical factor that influences user experience in applications. The results indicate that the key-value system, integrated with the GPU-based compaction strategy, can significantly optimize tail latency.

5) The GPU Usage Analysis: Fig. 13 illustrates the GPU utilization of the GPU-accelerated compaction strategies when running a Fillrandom workload. *LUDA* executes only two compaction processes, parsing key-value pairs and SST files generation, on the GPU, relying on the CPU to perform sorting key-value pairs. Consequently, the GPU utilization is lower compared to *GPU Comp-Pipe* and *GPU Comp-GDS*. *GPU Comp-Pipe* and *GPU Comp-GDS* complete the entire compaction process on the GPU, resulting in higher GPU consumption. As the value size increases, the cost of data transfer between SSD and GPU rises, leading to a decrease in GPU utilization. However, GDS mitigates this cost, causing only a negligible decrease in GPU utilization for GPU Comp-GDS. The peak GPU utilization is 22.51%, indicating that the GPU-accelerated Compaction strategy does not heavily consume GPUs while achieving a significant performance improvement.

C. Evaluate ReadRandom workload

In this workload, we initially insert 50 million key-value pairs into the KV system and subsequently perform 30 million queries (GET). Table II displays the read latency. GPU-accelerated compactions, including LUDA, do not optimize the query operation (GET) based on the GPU. Therefore, the random read performance of a GPU compaction-based system is barely affected compared to a CPU compaction-based system. *GPU-Comp Pipe* and *GPU-Comp GDS* do not conduct garbage collection during *Q-Compaction*, leading to higher space amplification. Additionally, the GET operation might need to traverse more SSTables to find the latest/valid key-value pair in L1 and L2. This can degrade read performance in certain cases. For instance, when the value size is 32 bytes, *GPU-Compaction Pipe* exhibits a slight increase in average latency and tail latency compared to *CPU-based Compaction*.

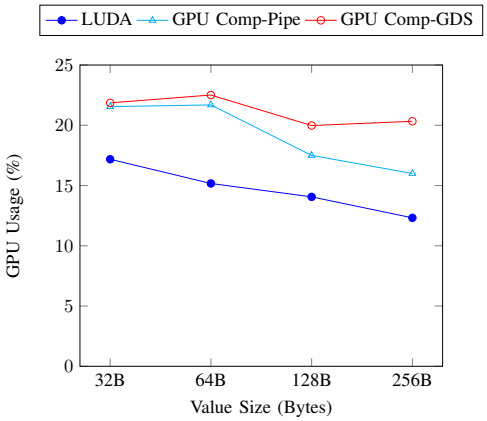


Fig. 13: The GPU Usage of GPU-accelerated compaction strategies

D. Evaluate ReadWriteMix workload

In this workload, we compare the performance of a mix of write and read requests, keeping the key size fixed at 16 bytes and varying different write ratios and the size of the value to create various experimental configurations. In all tests, we first load 50 million key-value pairs into the key-value storage system and then complete 100 million mixed read/write requests. Fig. 14 illustrates the performance of all configurations.

In LSM-Tree, the latency of read requests is typically higher than that of write requests. Therefore, when the write ratio is small, e.g., 50%, 70%, the ReadWriteMix workload leans towards being read-heavy, while as the write ratio increases, the ReadWriteMix workload becomes write-heavy. GPU-accelerated compaction doesn't optimize the query (GET). Therefore, the performance enhancement of the ReadWriteMix workload stems from write optimizations, particularly the acceleration of compaction. Consequently, the performance enhancement of the ReadWriteMix workload is significant only when the write ratio is large.

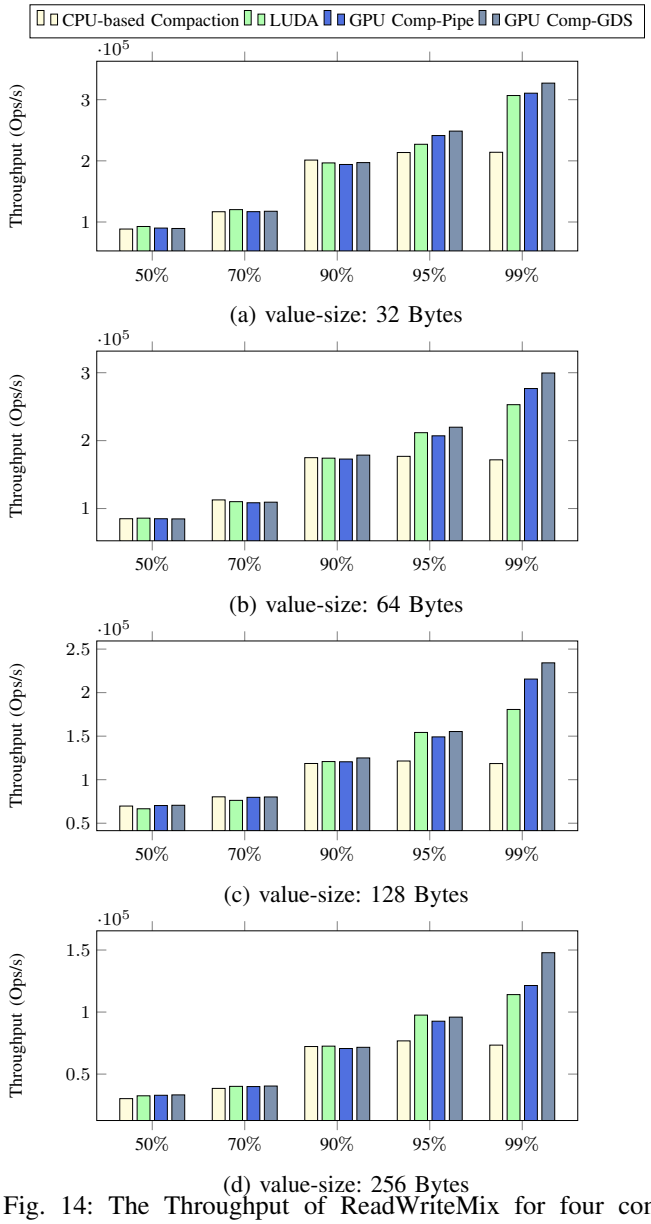
As a result, when the write ratio is 99%, *LUDA* improves by 43.35%/47.29%/52.42%/55.50% over *CPU-based Compaction* for value sizes of 32/64/128/256 bytes, respectively. Similarly, *GPU Comp-Pipe* demonstrates improvements of 45.14%/47.29%/52.42%/65.49% over *CPU-based Compaction* for the same value sizes. *GPU Comp-GDS* tends to perform optimally when the value size is large and the write ratio is high, allowing GDS to achieve maximum gains. When the write ratio is 99%, *GPU Comp-GDS* achieves 1.53x/1.74x/1.98x/2.02x higher throughput than *CPU-based Compaction*, and 1.07x/1.18x/1.30x/1.30x higher throughput than *LUDA*, respectively.

TABLE I: The write latency (μs) of four compaction strategies

Value Size	CPU-based Compaction			LUDA			GPU-Comp Pipe			GPU-Comp GDS		
	Avg	P99	P999	Avg	P99	P999	Avg	P99	P999	Avg	P99	P999
32	4.875	5.833	1114.980	3.245	5.688	19.217	2.692	5.853	12.794	2.578	5.701	13.503
64	6.008	6.448	1139.174	4.006	6.378	1050.779	2.933	6.178	14.928	2.648	5.988	14.639
128	8.475	7.623	1164.857	5.471	7.362	1124.480	4.654	7.166	1097.097	3.627	6.892	1006.968
256	13.910	1005.193	1182.285	9.069	8.719	1167.709	8.397	8.414	1163.193	7.004	7.998	1151.458

TABLE II: The read latency (μs) of four compaction strategies

Value Size	CPU-based Compaction			LUDA			GPU-Comp Pipe			GPU-Comp GDS		
	Avg	P99	P999	Avg	P99	P999	Avg	P99	P999	Avg	P99	P999
32	21.827	51.564	67.994	23.106	58.096	69.782	24.137	63.767	69.967	21.888	51.368	67.795
64	22.98	57.452	69.375	21.821	52.243	67.01	22.187	52.116	67.643	23.99	63.083	69.83
128	22.484	53.082	68.769	25.327	66.333	79.932	23.147	55.161	69.293	23.994	62.109	69.826
256	38.258	137.773	174.638	33.964	119.264	164.621	31.878	118.244	150.211	33.472	118.717	156.287



(a) value-size: 32 Bytes

(b) value-size: 64 Bytes

(c) value-size: 128 Bytes

(d) value-size: 256 Bytes

Fig. 14: The Throughput of ReadWriteMix for four compaction strategies

V. Related Work

Reducing the amount of IO of compaction or delaying the execution of compaction: Wisckey [21] and HashKV [22] proposed a strategy of separating Key and Value (*KV separation strategy*). In this case, LSM only stores the address of the value, which makes the entire LSM occupy a small storage space and reduces IO during compaction. The *KV separation strategy* is especially suitable for scenarios with large values and heavy write workloads, but it does not perform well in queries. PebblesDB [23] draws on the idea of SkipLists, inserts Guard into LSM, and proposes a *Fragmented Log-Structured Merge Tree* structure, which avoids a large amount of data rewriting at the same LSM level and further improves write throughput. Similarly, PebblesDB increases the query cost to a certain extent. TRIAD [24] only flushes the cold data in the Memtable to the disk and keeps the hot data in the Memtable to avoid triggering a large number of compaction operations on the disk. At the same time, it delays the compaction operation until the compacted files have enough key overlaps. However, delayed execution of compaction can only guarantee high throughput in the early stage and will cause a large number of high-level compactions in the later stage, which will still undermine system throughput. Therefore, accelerating compaction to ensure its timely completion is the key to achieving long-term stable throughput.

Offloading or accelerating compaction and GPU-accelerated KV storage systems and databases: The offloading compaction strategy is usually used in distributed key-value storage systems. To prevent the compaction operation from occupying the IO and computing resources of the *datastore* node, [25] offloads the compaction operation to a dedicated compaction server (*Remote Compaction*) so that the *datastore* node can better serve read/write requests. For the same purpose, [8] offloads compaction tasks to the FaaS (Functions as a Service) cluster for execution (*Faas Compaction*). Compared with *Remote Compaction*, *Faas Compaction* can provide more stable throughput for bursty workloads. Accelerated compaction mainly occurs in stand-alone scenarios. High-performance storage devices (such as SATA SSD, and NVMe SSD) make the bottleneck of LSM's compaction change from IO to computing, providing space for accelerating the computing in com-

paction. For example, PCP [20] proposes to use the pipeline mechanism to overlap the calculation and IO in the compaction operation. Specialized acceleration hardware FPGA [9] and DPU [12] are used to accelerate compaction, which reduces the delay of compaction and prevents compaction from competing with foreground read/write requests for CPU and IO resources. In addition, LUDA [13] first proposed using GPU to accelerate compaction, but it didn't use the GPU to implement the full Compaction process. It still relied on the CPU to sort key-value pairs, resulting in high CPU-GPU data transmission overhead, and did not effectively handle the data transmission overhead between SSD and GPU, discounting the acceleration effect. GPUs are also widely used to accelerate KV storage systems and databases. Mega KV [14] took full advantage of the high memory bandwidth and latency hiding capability of GPUs, achieving a high-performance and high-throughput in-memory KV system. OurRocks [15] offloaded the scan operation directly to GPU in Write-optimized database system, accelerating the analytic queries. In addition, OurRocks resolved the data transfer bottleneck with DMA.

VI. Conclusion

In this paper, we focus on using GPU to accelerate the compaction of LSM storage engines built on high-performance SSDs, eliminating the computational bottleneck of compaction. We design efficient GPU compaction units for each process of compaction. Based on that, we design a hierarchical acceleration strategy for the compaction tasks at different levels. We design two SSD-GPU data transfer mechanisms, *Pipeline mechanism* and *P2P mechanism*, to reduce the data transfer overhead during compaction. We compare the performance of our proposed strategy with naive CPU compaction strategy and the state-of-the-art GPU-accelerated Compaction method *LUDA*. The evaluation results show that our proposed compaction strategy can effectively improve compaction performance and overall system performance.

VII. Acknowledgements

This research is supported by National Science Foundation of China under Grant 62272252 and 62272253, the Key Research and Development Program of Guangdong under Grant 2021B0101310002, and the Fundamental Research Funds for the Central Universities.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[2] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li, "X-engine: An optimized storage engine for large-scale e-commerce transaction processing," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 651–665.

[3] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: a database benchmark based on the facebook social graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1185–1196.

[4] S. Ghemawat and J. Dean. (2014) Leveldb: A fast and lightweight key/value database library by google. [Online]. Available: https://github.com/google/leveldb

[5] FaceBook. (2014) Rocksdb: A persistent key-value store for flash and ram storage. [Online]. Available: https://rocksdb.org/

[6] Apache. (2021) Apache: Cassandra. [Online]. Available: https://www.https://cassandra.apache.org/_/index.html

[7] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "Matrixkv: Reducing write stalls and write amplification in lsm-tree based kv stores with matrix container in nvm," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 17–31.

[8] J. Li, P. Jin, Y. Lin, M. Zhao, Y. Wang, and K. Guo, "Elastic and stable compaction for lsm-tree: A faas-based approach on terarkdb," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 3906–3915.

[9] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao *et al.*, "Fpga-accelerated compactions for lsm-based key-value store." in *FAST*, 2020, pp. 225–237.

[10] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "Spandb: A fast, cost-effective lsm-tree based kv store on hybrid storage." in *FAST*, vol. 21, 2021, pp. 17–32.

[11] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 447–461.

[12] C. Ding, J. Zhou, J. Wan, Y. Xiong, S. Li, S. Chen, H. Liu, L. Tang, L. Zhan, K. Lu *et al.*, "Dcomp: Efficient offload of lsm-tree compaction with data processing units," in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 233–243.

[13] P. Xu, J. Wan, P. Huang, X. Yang, C. Tang, F. Wu, and C. Xie, "Luda: Boost lsm key value store compactions with gpus," *arXiv preprint arXiv:2004.03054*, 2020.

[14] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[15] W. G. Choi, D. Kim, H. Roh, and S. Park, "Ourrocks: offloading disk scan directly to gpu in write-optimized database system," *IEEE Transactions on Computers*, vol. 70, no. 11, pp. 1831–1844, 2020.

[16] NVIDIA. (2020) Nvidia gpudirect storage overview guide. [Online]. Available: https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[18] NVIDIA. (2012) Nvidia gpudirect rdma. [Online]. Available: https://docs.nvidia.com/cuda/gpudirect-rdma/

[19] L. B. Costa, S. Al-Kiswany, and M. Ripeanu, "Gpu support for batch oriented workloads," in *2009 IEEE 28th international performance computing and communications conference*. IEEE, 2009, pp. 231–238.

[20] Z. Zhang, Y. Yue, B. He, J. Xiong, M. Chen, L. Zhang, and N. Sun, "Pipelined compaction for the lsm-tree," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 777–786.

[21] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.

[22] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling efficient updates in KV storage via hashing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018, pp. 1007–1019.

[23] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 497–514.

[24] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "Triad: Creating synergies between memory, disk and log in log structured key-value stores," in *2017 USENIX Annual Technical Conference*, 2017, pp. 363–375.

[25] M. Y. Ahmad and B. Kemme, "Compaction management in distributed key-value datastores," *Proceedings of the VLDB Endowment*, vol. 8, no. 8, pp. 850–861, 2015.