H-Rocks: CPU-GPU accelerated Heterogeneous RocksDB on Persistent Memory

SHWETA PANDEY, Indian Institute of Science, India
ARKAPRAVA BASU, Indian Institute of Science, India

Persistent key-value stores (pKVS) such as RocksDB are critical to many internet-scale services. Recent works leveraged persistent memory (PM) to improve pKVS throughput. However, they are typically limited to CPUs.

We develop H-Rocks to judiciously leverage both the CPU and the Graphics Processing Unit (GPU) for accelerating a wide range of RocksDB operations. H-Rocks selectively accelerates performance-critical parts of RocksDB on the GPU. It uses operation sub-batching and key-value versioning to leverage GPU's parallelism while maintaining compatibility with RocksDB. It harnesses GPU's high-bandwidth memory while limiting data movement between the CPU and GPU. In YCSB workloads, H-Rocks outperforms CPU-based pKVSs like Viper, Plush, and RocksDB-pmem by 3-18×.

CCS Concepts: • **Information systems** → **Key-value stores**; **Parallel and distributed DBMSs**.

Additional Key Words and Phrases: Graphics Processing Unit, Persistent Memory

ACM Reference Format:
Shweta Pandey and Arkaprava Basu. 2025. H-Rocks: CPU-GPU accelerated Heterogeneous RocksDB on Persistent Memory. *Proc. ACM Manag. Data* 3, 1, N1 (SIGMOD), Article 44 (January 2025), 28 pages. https://doi.org/10.1145/3709694

1 Introduction

A persistent key-value store (pKVS) forms the backbone of many internet-scale web services [24]. A pKVS enables fast retrieval and modification of schema-less data (values) uniquely identified by keys. It ensures reliability by storing key-value (KV) pairs on non-volatile media. Meta's RocksDB [60] and Google's LevelDB [32] are examples of commercially deployed pKVSs. RocksDB is widely deployed for logging web user interactions and queuing services [18, 41, 71], for indexing services, for caching web content [38], as storage engines for databases [39], and for stream processing [31]. It is thus no surprise that the likes of RocksDB often need to support more than millions of requests per second [18]. The throughput of a pKVS is a key factor in determining the performance of many important web services [14].

Significant research has focused on improving the throughput of pKVSs (e.g., [2, 28, 30, 49, 66, 70, 74]). Several recent academic works (*e.g.*, [2, 49, 66]), as well as a fork of RocksDB, called RocksDB-pmem [19], leveraged fast and fine-grained persistence enabled by persistent memory (PM) technologies such as Intel's Optane DC memory to achieve significant speedups [27]. Although Intel recently discontinued Optane, alternatives, e.g., Samsung's memory-semantic SSD, support a persistent memory mode [65].

Authors' Contact Information: Shweta Pandey, shwetapandey@iisc.ac.in, Department of Computer Science and Automation, Indian Institute of Science, Bengaluru, India; Arkaprava Basu, arkapravab@iisc.ac.in, Department of Computer Science and Automation, Indian Institute of Science, Bengaluru, India.

ACM 2836-6573/2025/1-ART44
https://doi.org/10.1145/3709694

Unfortunately, prior works on pKVSs have focused on CPUs. The only publicly known work to leverage GPUs in a pKVS, called gpKVS [50], is a proof-of-concept demonstrating GPU's ability to access PM-resident data. However, it has limited practical use. It *only* supports PUTs and GETs, that too, for fixed 8-byte keys and values. In contrast, commercially deployed pKVSs support a diverse set of requests *e.g.*, UPDATE, DELETE, RANGE queries. Although gpKVS can execute a batch of PUTs or GETs on a GPU, it fails for a stream of requests with PUTs and GETs interspersed, typical in a real-world deployment. In gpKVS, a GET is not guaranteed to return the latest value for a key. Further, it fails to harness the GPU's onboard high-bandwidth memory (HBM) and massive parallelism. Instead, it becomes constrained by PM's limited bandwidth.

CPU-based research pKVSs also often lack the essential functionalities needed for real-world deployments. Many constrain keys and/or values to 8 bytes [7, 35, 47, 49, 79], while real-world KV sizes are typically larger [15]. Works like Viper [2] and Plush [66], which support variable KV sizes, lack features like transactions or range queries. Importantly, prior works may fail to provide the same consistency guarantees as RocksDB or LevelDB. When presented with multiple concurrent operations to the same key, they can permanently lose updates and/or return stale data. Consequently, an existing application using RocksDB or LevelDB cannot leverage these proposals without altering the application logic. This limits their deployments in the real-world.

In this context, we aim to leverage GPU's parallelism to accelerate pKVS but *without* compromising its practical usefulness. We take a philosophically different approach from prior works whereby we judiciously accelerate parts of a commercially deployable pKVS such as RocksDB-pmem [1] on a GPU while retaining its rich functionalities needed for practical deployment. We name our software Heterogeneous RocksDB or H-Rocks in short. H-Rocks leverages both GPU and CPU, i.e., heterogeneous processing, to accelerate a broad set of operations while preserving full compatibility with the unmodified RocksDB. Importantly, H-Rocks produces the same results as RocksDB for the same sequence of operations.

We begin by exploring how RocksDB operations can benefit from GPU's parallelism. We identify performance-critical parts of RocksDB that can benefit from GPU acceleration. We find that accesses to RocksDB's in-memory (DRAM) data structures are often critical to its performance. For example, PUTs can spend 67-84% of their execution time in accessing the memtable, an in-memory buffer that temporarily holds newly inserted KV pairs. Further, 16-31% of time is spent writing to the write-ahead log (WAL) on PM. The background process of persisting KV pairs into the PM-resident Log-structured Merge (LSM) tree has little impact on the performance. Similarly, more than 80% of time in GET-mostly workloads (e.g., YCSB-C) can be attributed to looking up in-memory data structures such as memtables and the block cache that caches frequently read KV pairs. GETs to frequently accessed KV pairs can often be found in the in-memory data structures, without needing accesses to the PM-resident LSM tree. Even for requests that ultimately access PM-resident LSM tree, a considerable fraction of time is spent looking up several memtables (e.g., one active and five immutable memtables) and the block cache searching for the missing KV pair in the memory.

Driven by this analysis, H-Rocks *selectively* accelerates parts of RocksDB that offer the most headroom and are better suited for GPU acceleration. For PUTs and DELETEs, we accelerate insertions to memtable and logging to the WAL while leaving operations on the LSM tree unmodified on the CPU. We leverage GPU-optimized lock-free parallel logging for WAL [50]. Similarly, for GETs and RANGE queries, the GPU accelerates accesses to in-memory memtables and the block cache. H-Rocks utilizes RocksDB's existing functionalities to access PM-resident LSM tree and performs compaction on the CPU since they offer little benefit from acceleration.

[1]For brevity, we refer to RocksDB-pmem as RocksDB in the rest of the paper.

The division of labor between the GPU and CPU has several key benefits: ① Reduces the complexity and overheads of accessing the CPU-attached PM from the GPU. ② Minimizes modifications to RocksDB while allowing most of the benefits from GPU acceleration. ③ Preserves rich functionalities of RocksDB such as its diverse compaction strategies of the LSM tree and data integrity checks that are important for deployments across various real-world scenarios.

RocksDB must service high request arrival rates to cater to the demands of internet-scale services [54]. This provides an opportunity to leverage GPU's parallelism to execute many requests concurrently (i.e., a batch). However, naïve batching of incoming requests is ill-suited to GPU's Single Instruction Multiple Thread (SIMT) execution model. While the GPU thrives when presented with hundreds of thousands of the same operations, a stream of requests can have a combination of disparate operations – PUTs, GETs, UPDATEs, etc. Further, a batch may contain PUTs and GETs for the *same* key. Executing them concurrently can lead to inconsistency where a GET may not return the latest value and/or lose updates.

Towards this, H-Rocks employs two techniques – sub-batching and versioning. It segregates requests within a batch into sub-batches based on their type (write, update, or read). Requests within a sub-batch execute concurrently, leveraging GPU's SIMT execution. However, different sub-batches execute sequentially. The write sub-batch executes first, followed by the update, and finally, the read sub-batch. This ordering ensures that updates and reads (*e.g.* GET) observe newly inserted or updated KV pairs. However, since requests within a given sub-batch execute concurrently, it is possible to overwrite updates to the same key in an arbitrary order. To avoid this, H-Rocks maintains all versions of a KV pair within a batch and tags each request with a unique request ID (reqID). Each request retrieves or modifies the *correct* version of a key vis-à-vis its reqID (§ 3.2). This ensures that reads return the latest write and no update is lost.

To support GPU's parallel execution, one must service many concurrent memory access. H-Rocks, thus, places performance-critical in-memory data structures such as memtables on GPU's high-bandwidth memory (HBM). However, the overheads of copying data between the CPU's DRAM and GPU's HBM over the PCIe interconnect can outweigh the benefits, especially for large values [37, 75]. To strike a balance, it allocates values on DRAM while placing *pointers* to values, their keys, and metadata on the HBM-resident memtables. The GPU accesses DRAM-resident values through pointers by leveraging Unified Virtual Addressing (UVA) [45]. For the block cache, H-Rocks allows both the GPU and the CPU to look it up while allowing *only* the CPU to add/remove contents from the block cache to avoid costly synchronization between the CPU and the GPU. Furthermore, H-Rocks uses NVIDIA's unified virtual memory (UVM) technology to transparently migrate parts of the block cache onto the HBM at runtime based on access patterns.

Finally, H-Rocks further leverages heterogeneous processing, i.e., both CPU and GPU, in three ways. First, it leverages GPU to enable high throughput under a high request arrival rate while seamlessly falling back to CPU at a lower request rate to avoid overhead batching and to maintain low latency. Second, for requests that must access the LSM tree, *e.g.* GET that misses in the memtable and block cache, H-Rocks leverages batching to concurrently (to GPU) service multiple requests on the multi-core CPU. Finally, while H-Rocks accelerates a broad set of operations on the GPU including PUT, GET, UPDATE, DELETE, RANGE and transactions, RocksDB also supports uncommon ones such as 'DeleteRange'[1, 53, 55, 56, 59]. H-Rocks falls back to unmodified CPU execution for such operations to maintain compatibility with RocksDB.

H-Rocks improves PUT throughput by 13.2×, 3.5×, 4.4× and 15× over RocksDB, Viper [2], Plush [66], and gpKVS [50], respectively. For GETs the improvements are 46.8×, 10.5×, 11.3× and 52×. For YCSB workloads[8], the improvements are 3-18× over Viper, Plush, and RocksDB.

In summary, we make the following contributions.

• H-Rocks judiciously uses both CPU and GPU to selectively accelerate parts of RocksDB-pmem while retaining its rich features.
• It employs sub-batching of operations and versioning of KV pairs for leveraging GPU's SIMT architecture while adhering to RocksDB's consistency guarantees.
• It leverages GPU's HBM but balances the cost of data movement, leveraging UVA and UVM for better overall performance.

The source code of H-Rocks is available at: https://github.com/csl-iisc/H-Rocks-SIGMOD25.git.

2 Background

Key-value stores (KVSs) are widely used across internet infrastructure as hashmaps (*e.g.*, Amazon Dynamo [11]), in-memory databases (*e.g.*, Redis [51]), and on persistent stores (*e.g.*, Google's LevelDB [32] and Meta's RocksDB [60]). This work focuses on Meta's RocksDB, which shares a similar fundamental structure with LevelDB.

2.1 RocksDB internals

RocksDB is deployed by major online services such as Yahoo, LinkedIn, Alibaba, and Meta [15, 23, 44, 67]. To support the needs of diverse applications, RocksDB provides a rich set of request types beyond the basic PUT, GET, and DELETE. For example, a MERGE (UPDATE) can update a value associated with a key using a user-provided function [59]. The iterator (scan) can be used for range queries, iterating over a range of KV pairs starting from a given start_key [53]. RocksDB also supports transactions [55].

RocksDB uses an LSM (log-structured merge) tree as its primary data structure for storing KV pairs on a persistent medium [61]. An LSM tree comprises multiple levels. KV pairs are sorted by keys within each level. Each deeper level is larger than its previous level (default, 10×). For better performance, RocksDB employs in-memory data structures such as memtables to temporarily buffer new KV pairs [58] and a block cache for keeping frequently read pairs [57].

On a PUT or an UPDATE, a new or an updated KV pair is first inserted into an in-memory active memtable. Entries of memtables are sorted by keys for faster reads and efficient merging into the LSM tree. For recovery, new pairs are logged to a write-ahead log (WAL) on a persistent media [62]. When an active memtable becomes full, it is marked immutable and is eventually flushed to the LSM tree as a Sorted String Table file (SSTFile) by a background thread at a later time. RocksDB maintains multiple immutable memtables (configurable, default 5) to hide the latency of flushing memtables from the critical path. SSTFiles keep sorted KV pairs divided into blocks. A level in an LSM tree contains multiple SSTFiles, each covering disjoint key ranges. When a level reaches its maximum size, an SSTFile of that level is merged with SSTFiles having overlapping key ranges in the next level. Background threads flush memtable's contents to the LSM tree and perform compaction of the tree [52]. Flushing of memtables can stall RocksDB if the L0 level (top level of the tree) is full [63]. Similarly, a compaction stall occurs when slow compaction stops the processing of new requests.

A GET request first looks up the active memtable keeping the latest data. On a miss, all immutable memtables are looked up sequentially. Next, the block cache is queried. The block cache keeps the frequently read pairs. If the key is still not found, RocksDB uses bloom filters to identify which levels of the LSM tree may contain the desired pair. The levels are then looked up in ascending order to retrieve the latest value.

Consistency guarantee: RocksDB is often used by applications that require strong data consistency. Thus, it provides strict guarantees that these applications can rely on. First, it guarantees that a read operation *e.g.* GET, RANGE, UPDATE, will read the latest committed write to that key [13, 72].

Second, all requests to a given key are performed as if they were executed sequentially upon arrival. Thus, no updates are lost. We call these properties the RocksDB's *consistency guarantee*.

We focus on RocksDB-pmem, a version of RocksDB that places the LSM tree and WAL on persistent memory. In the rest of the paper, we refer to it as RocksDB for brevity.

2.2 Persistent Memory (PM)

PMs such as Intel's Optane DC-PMM [27] enable byte-addressable loads and stores to persistent data structures. PMs enable fine-grain persistence at latencies *comparable* to volatile DRAM [73]. While Intel discontinued Optane, Samsung announced memory-semantic SSD technology (MS-SSD) having a persistent memory mode [65]. MS-SSD combines SSD storage with a battery-backed DRAM cache for high throughput random reads and writes to persistent media. Alternative PM technologies such as 3D flash memory [69] and Everspin's STT-RAM [68] are also emerging. In addition, Compute Express Link (CXL) [64], the emerging industry standard for disaggregated computing, has incorporated a global persistent flush operation for CXL-attached PM [10]. It suggests the industry's expectation that many vendors are likely to offer PM products in the future [9]. In short, while Optane was the first commercial PM, it is unlikely to be the last.

2.3 Graphics Processing Unit (GPU)

A GPU contains tens of Streaming Multiprocessors (SMs), each containing multiple CUDA cores, with several lanes for individual GPU threads. A kernel is launched with a grid of GPU threads. A grid consists of threadblocks, each containing up to 1024 threads, all executing on the same SM. A threadblock contains multiple warps, each containing 32 threads executing in a Single Instruction Multiple Threads (SIMT) fashion. GPUs typically have high-bandwidth memory (HBM) with up to 2TB/sec bandwidth [46]. However, its capacity is often limited to a few tens of GBs. NVIDIA's Unified Virtual Addressing (UVA) enables GPU to access data on the CPU's DRAM [45]. UVA can map (parts of) the DRAM onto the GPU's virtual address space, enabling kernels to perform loads and stores to DRAM-resident data. Additionally, NVIDIA's Unified Virtual Memory (UVM) enables transparent migration of data across DRAM and HBM on-demand [26].

A GPU is typically connected to a CPU through a PCIe interconnect [33]. Crossing the PCIe adds significant latency and offers limited bandwidth. Thus, minimizing data movement across the PCIe is essential for good performance [37, 75].

3 Opportunities and Challenges for GPU acceleration

We quantitatively and qualitatively analyze the opportunities and challenges in leveraging GPU for accelerating RocksDB.

3.1 Analyzing opportunities for GPU acceleration

GPUs excel in the presence of abundant parallelism in computation and data access but falter when waiting for data from slower mediums, such as persistent media. Furthermore, GPUs typically rely on CPUs for accessing files since system components (*e.g.*, the OS and drivers) cannot run on GPUs.

In this context, we quantitatively analyze time spent in various components of RocksDB while executing PUTs and GETs to identify opportunities for GPU acceleration. We configure RocksDB following its tuning guide [20] for optimal performance. For instance, we set the memtable size to 512MB, allow up to five immutable memtables, and use a 4GB block cache as recommended.

PUTs: We perform 100 million PUTs with randomly generated key-value (KV) pairs with varying sizes and analyze the time spent on its different sub-operations. Table 1 reports percentages of execution time spent on various sub-operations for PUTs. We find that 67-84% of the time is spent on inserting new KV pairs into memtables while 16-31% on logging (WAL) them. Logging time

Table 1. Breakdown of time to execute PUTs on RocksDB.

Key size	Value size	Memtable	WAL	Compaction	Flush
8	128	83.52%	16.48%	0%	0%
8	512	77.24%	22.67%	0%	0%
32	128	79.05%	20.7%	0%	0.25%
32	1KB	67.1%	31.16%	1.5%	0.24%

increases with KV sizes as more data is written to the PM-resident log. Only up to 1.7% of the time is attributed to stalls due to flushing of the memtables and compaction of SSTFiles. Fast accesses to the PM-resident LSM tree, coupled with efficient flushing and compaction of the tree by background threads, ensures negligible flush and compaction stalls.

In summary, we observe that insertions to memtables and logging are the critical sub-operations to accelerate for speeding up PUTs. The sub-operations such as flushing the memtables to the LSM tree and compacting the tree, are not.

GETs: The performance of GETs depends on the distribution of keys being retrieved. We, thus, evaluate the performance of GETs using popular workload generators like YCSB [8] and Meta's ZippyDB. Note that we here focus on read-only and read-mostly workloads and ignore write-mostly YCSB-A since the goal is to evaluate GETs. These workloads mimic real-world scenarios where a subset of keys is accessed more frequently.

Table 2 reports percentages of time spent in important components of RocksDB while serving several read-mostly workloads. Across the workloads, 78-84% of the time is spent in accessing in-memory data structures – memtables and the block cache. Only 17-22% of the time is spent accessing the LSM tree. We notice that many GET requests find the KV pairs in the memtables or in the block cache since a subset of keys are accessed more frequently than the rest. Even the requests that ultimately retrieve keys from the LSM tree, RocksDB must first look up the active and immutable memtables since the latest copy of the KV pair may reside there. On missing in the memtables, the block cache is searched before querying the LSM tree. Thus, all GETs that access the LSM tree also spend time looking up the in-memory data structures.

The analysis shows accelerating accesses to memtables and the block cache can significantly speed up GETs. However, optimizing accesses to the LSM tree can offer only limited benefits.

Fortunately, accesses to the in-memory data structures are amenable to the GPU's parallel execution and can leverage HBM. While our analysis here focuses on PUTs and GETs, other requests follow similar patterns. For example, a DELETE is a PUT with a special 'tombstone marker' as its value. An UPDATE combines a GET and PUT. A RANGE query behaves similarly to a batch of GETs.

3.2 Challenges in leveraging GPU acceleration

While our analysis finds significant opportunities to accelerate accesses to memtables, block caches, and logging, we encounter four key challenges in harnessing GPU's capabilities for RocksDB.

Table 2. Breakdown of time spent on GETs on different RocksDB components.

Workload	Memtables	Block $	LSM tree
ZippyDB [4]	73.56%	9.65%	16.79%
YCSB-B (Zipfian R:W=90:10)	60.85%	16.57%	22.58%
YCSB-C (Zipfian R:W=100:0)	62.71%	18.91%	18.38%
YCSB-D (Read latest R:W=90:10)	66.95%	13.38%	19.67 %

Table 3. Memtable lookup times in μsecs with increasing memtable sizes.

Memtable size	64MB	128MB	256MB	512MB	1GB	2GB	4GB
Lookup time	6.3	6.6	7.04	7.25	7.56	7.80	8.29

① **Concurrent execution of requests on a GPU:** H-Rocks must execute multiple requests concurrently on a GPU but should also abide by RocksDB's consistency guarantees i.e., requests should be serviced as if they were executed sequentially (§ 2.1). While an obvious approach to leverage GPU's parallelism would be to batch many requests to execute them concurrently, it presents two challenges. (A) To harness GPU's SIMT execution model (§ 2.3), many requests of the same type (e.g., GET, PUT) must be executed concurrently. This can allow the same operations to execute concurrently (on different data items) to leverage GPU's data-parallel execution. However, a stream of incoming requests can have a mix of disparate request types. Naïvely batching requests to execute on the GPU would lead to poor SIMT efficiency, severely limiting the usefulness of a GPU. (B) Importantly, a batch of requests can contain requests on the *same* key. Executing these requests concurrently could violate the RocksDB's consistency guarantee. Consider the following request stream: PUT(k_1, v_1), GET(k_1), PUT(k_1, v_2), GET(k_1). Adherence to the consistency guarantee requires that the first GET returns v_1, while the second must return v_2. Ensuring this is non-trivial while executing requests in parallel on the GPU.

② **Scaling memtable structure for GPU:** While the analysis showed that memtables are critical to RocksDB's performance, we find that its structure is fundamentally not scalable to GPUs in two ways. First, the skip-list-based memtable design in RocksDB necessitates a data-dependent walk of the list upon a lookup or insertion to the memtable. Such data-dependent accesses are at odds with the data-parallel execution of GPUs. Further, the larger the memtables, the longer the walks, leading to slower lookups. Table 3 reports the time to look up an entry in the memtable with increasing size. While larger memtables typically ensure a higher hit rate, the benefits of fewer accesses to the LSM tree can be outweighed by longer memtable lookup times. We also ran YCSB (A-D) workloads with varying memtable sizes and found that performance peaks at memtable size of 512MB before it starts dropping with larger memtables. Slower lookups outweigh the benefits of higher hit rates.

Second, to look up and/or insert multiple KV pairs concurrently in skip lists, one must acquire locks. However, locks are fundamentally detrimental to GPU acceleration as they constrain SIMT execution. It is thus necessary to re-imagine the memtable structure for GPUs.

③ **Balancing benefits of HBM with overheads of PCIe:** GPU's onboard HBM supports up to multi-TBs/second bandwidth – manifold that of CPU's DRAM. It is thus natural to place performance-critical in-memory data structures such as memtables on the HBM to support concurrent accesses. However, the data movement between CPU's DRAM and GPU's HBM over the slow PCIe interconnect is costly. Since requests originate and conclude at the CPU, the overheads of data movement can outweigh the benefits of high memory bandwidth if one naïvely places all in-memory data structures on the HBM. Further, HBM is capacity constrained – while a server can easily have TBs of DRAM, HBM sizes are typically limited to tens of GBs. Thus, one must carefully balance the conflicting goals of effectively harnessing HBM and limiting the overheads of data movement over the PCIe.

④ **Logging from GPU and supporting diverse deployment:** A significant time is spent in write-ahead logging for PUTs. It is imperative to enable efficient and parallel logging from the GPU to the PM-resident log without overheads of mediation by the CPU. Further, H-Rocks aims to be widely deployable. While using GPU can help achieve high throughput under a high request arrival rate, one must also ensure low latency under low request arrival rates. This requires H-Rocks to dynamically

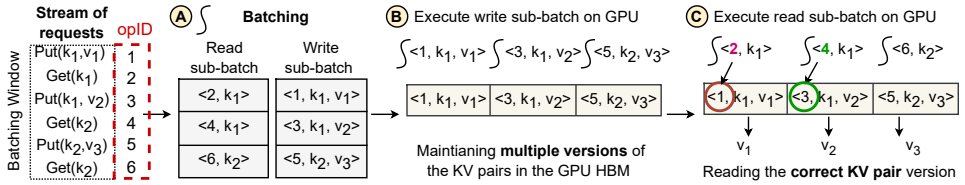

Fig. 1. Batching and versioning of KV pairs in H-Rocks.

adapt to a varying request arrival rates. Finally, RocksDB supports a diverse set of operations, and H-Rocks must maintain backward compatibility.

4 H-Rocks: Key Ideas and Design

The primary goals of H-Rocks are as follows. (A) Leverage GPU acceleration *when and where* it is beneficial to achieve high throughput of servicing requests. It must seamlessly fall back to using the CPU only where necessary. (B) Accelerate a wide range of requests beyond PUTs and GETs, that are typical in real-world deployments of pKVSs. (C) Must return the same results (*e.g.*, values returned by GETs) as RocksDB when presented with the same request stream – it must adhere to RocksDB's strict consistency guarantees. (D) Must preserve RocksDB's rich set of features, *e.g.* different compaction strategies, and data integrity checks, for backward compatibility.

To achieve these goals, H-Rocks must address the challenges listed earlier. We provide an overview of H-Rocks's strategy towards them.

(1) **Concurrent execution of requests:** Driven by the empirical analysis, H-Rocks uses GPU to accelerate accesses to RocksDB's in-memory data structures – memtables, and block cache. It also accelerates logging for PUTs and UPDATEs. However, access, updates, and compaction of the PM-resident LSM tree are performed on the CPU as in unmodified RocksDB. This division of labor between the CPU and the GPU has two advantages. (A) It allows RocksDB to effectively harness GPU acceleration with minimal modifications. (B) It preserves the rich set of features offered by RocksDB, *e.g.* data integrity checks, and diverse compaction strategies, that are important for wide real-world deployments.

To accelerate requests on GPUs, H-Rocks must leverage its SIMT architecture while maintaining RocksDB's consistency guarantees. Toward this, H-Rocks creates *sub-batches*, grouping requests of the same type (*e.g.* GET, PUT). While requests in a sub-batch are executed concurrently, different sub-batches within a batch are ordered by the request type. A sub-batch is classified into one of the three categories: (A) Writes (blind-writes): includes PUT and DELETE, which alter the pKVS without reading from it. (B) Updates (read-modify writes): includes UPDATE, which reads the value, modifies it, and writes it back. (C) Reads (read-only): includes requests like GET and RANGE, which only retrieve the KV pairs. Requests in the write sub-batch execute first, followed by updates, and finally, the read sub-batch. This order ensures that requests that produce new values (*e.g.* PUT, UPDATE) are executed before those that may consume those values (*e.g.* GET, RANGE). Note that each batch may not have all three sub-batches as it depends on the request mix in the incoming stream.

While sub-batching leverages the SIMT architecture, it alone cannot ensure adherence to consistency guarantees when multiple requests operate on the same key. H-Rocks keeps *all versions* (values) of a KV pair produced by a sub-batch until all requests in its *encompassing batch* complete execution. Each request is assigned a *unique* request ID (reqID) that captures its arrival order. A new or updated KV pair is tagged with the reqID of PUT or UPDATE that creates it. A GET returns the version whose reqID is smaller than its own reqID but is the largest among the matching versions. Thus, a

GET returns the latest value based on its arrival order as in RocksDB. All updates to a key are applied in the increasing order of their reqID, ensuring no updates are lost. Together, they ensure adherence to RocksDB's consistency guarantee.

Figure 1 depicts H-Rocks's sub-batching and versioning. A batch of requests is created from a request stream on the CPU (Ⓐ). Each request within the batch is assigned a unique reqID based on the arrival order and a batch is assigned a batch ID. Requests are segregated into write and read sub-batches (no update batch in this example). The kernel to process the write sub-batch processing is then launched on a GPU (Ⓑ). Each GPU thread inserts a KV pair. In the example, H-Rocks maintains two versions of the key k_1, with different reqIDs and values corresponding to two PUTs. Next, the read sub-batch is executed, where each GPU thread executes one GET (Ⓒ). The first GET with reqID '2' returns the value v_1 while the GET with reqID '4' returns v_2 for the same key k_1. It chooses the version with the largest reqID that is smaller than that of a given GET request.

② **Scalable memtable design for GPU:** While H-Rocks enables concurrent servicing of multiple requests on the GPU through batching and versioning, the memtable must also support multiple concurrent lookups and insertions. Traditional skip-list-based memtables, including their GPU implementations [42], fail to scale due to locking. H-Rocks thus employs a GPU-friendly lock-free memtable design. We make two observations. First, thanks to (sub-) batching, the set of KV pairs to be inserted or looked up are known before the memtable(s) are accessed. Second, existing efficient GPU-accelerated sorting and binary search algorithms can be leveraged to efficiently sort and look up KV pairs. Driven by these, H-Rocks designs its GPU memtable as a table sorted by the keys. KV pairs are sorted before insertion into the memtable. Then they are inserted concurrently without requiring locks (detailed in § 5). GPU-accelerated binary search helps to quickly look up KV pairs.

③ **Judiciously leveraging HBM:** It is natural to place performance-critical and frequently accessed data structures such as memtable on HBM to harness its ability to support many concurrent memory accesses. However, naïvely placing the *entire* memtable on HBM can suffer from high overheads of data movement between the DRAM and the HBM. To strike a balance, H-Rocks places the memtable, *excluding* the values, on HBM. The values, which are often much larger than keys, are placed on the DRAM. Each entry in the HBM-resident memtable keeps a pointer to the corresponding value in DRAM. This way, values are not moved across the PCIe but the GPU can still perform fast concurrent lookups to the memtables. H-Rocks leverages NVIDIA's UVA technology [45] for mapping DRAM-resident values onto GPU's virtual address space and uses pointers to access them.

Further, the use of pointers allows memtables to scale independently of the value size. The pointers are always 8 bytes, while value sizes can be hundreds to thousands of bytes. Since the values are not a part of the HBM-resident memtable, larger value sizes do not affect the number of entries a memtable can host. This is also beneficial since HBM capacity is often limited.

Finally, H-Rocks harness NVIDIA's UVM technology [26] to dynamically migrate parts of the block cache to HBM based on access patterns. Specifically, the block cache is accessible to both CPU and GPU (detailed in § 5) and is placed on the CPU DRAM. At runtime, the UVM driver automatically migrates the block cache's frequently accessed pages from the DRAM to the HBM.

④ **GPU friendly logging and leveraging CPU-GPU heterogeneity:** H-Rocks leverages GPU-optimized hierarchical coalesced logging (HCL) to efficiently write to the PM-resident log directly from the GPU without needing slow mediation from the CPU [50]. This is possible due to the byte-grained persistence of PM. HCL mimics a GPU's thread hierarchy within the log, allowing each thread to write to a pre-defined location in the log without a lock. Further, it effectively leverages GPU's hardware coalescing [25] to limit the number of accesses to PM. We couple this with the bulk transfer of values to PM for better utilization of PM's limited write bandwidth.

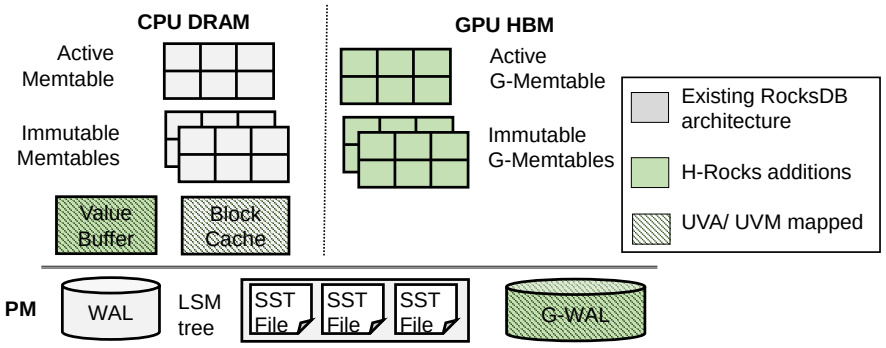

Fig. 2. The primary data structure of H-Rocks.

H-Rocks deploys the GPU only when necessary to service requests under high arrival rates. At a low request rate, the entire execution happens on the CPU as in RocksDB. However, as the rate increases, requests queue up due to RocksDB's low service rate. When the queue size exceeds a threshold (configurable; default: 10K requests), H-Rocks batches and processes them on the GPU. Thus, at low request rates, it delivers latencies consistent with RocksDB, while at higher rates, it leverages the GPU to deliver higher throughput. Further, H-Rocks executes uncommon request types not worth accelerating, e.g., DeleteRange, on the CPU to maintain compatibility with RocksDB.

Summary: ① H-Rocks accelerates a wide range of requests by selectively executing parts of requests on the GPU while leveraging heterogeneous processing. ② It uses sub-batching and versioning to utilize GPU's SIMT execution while adhering to RocksDB's consistency guarantees. ③ H-Rocks judiciously places critical data structures on HBM while employing the UVA and UVM technologies to limit data movement over the PCIe. ④ It adapts to a varying request arrival rate by dynamically choosing between the CPU and the GPU to strike a balance between the latency and throughput.

5 Data structures of H-Rocks

We here detail the primary data structures of H-Rocks before discussing how they participate in servicing requests. Figure 2 shows the data structures employed by H-Rocks for GPU acceleration and their placement on the HBM, DRAM, or PM (see legends). These include newly introduced data structures and those leveraged and/or extended from RocksDB. It also retains all of RocksDB's data structures and are used to support H-Rocks's heterogeneous operations when it leverages CPU. However, we focus on new or modified data structures that H-Rocks employs for GPU acceleration.

G-memtables and value buffer: G-memtables are HBM-resident data structures that temporarily buffer newly inserted or updated KV pairs, similar to RocksDB's memtables. They are accessible *only* from the GPU and are maintained as lock-free tables sorted by keys. As discussed in § 4, sorted tables allow H-Rocks to leverage GPU's data-parallel processing without needing locks, unlike skip-list based CPU memtables. After sorting the keys on the GPU, KV pairs are inserted into designated locations in the list without locks. An efficient binary search on the GPU is used to look up the memtables. G-memtables can store multiple versions for the same key, uniquely identified by reqID

reqID (8B)	keySz (4B)	key (variable length)	valueSz (4B)	valuePtr (8B)

Fig. 3. A G-memtable entry.

tag (1B)	invalid (1B)	accessCtr (4B)	keySz (4B)	key (variable length)	valueSz (4B)	valuePtr (8B)

Fig. 4. A block cache entry.

of the request creating the version. Note, H-Rocks does not modify RocksDB's memtables on the DRAM that are solely accessed from the CPU.

Like RocksDB, H-Rocks maintains two types of G-memtables in HBM – active and immutable. H-Rocks has one active and multiple immutable G-memtables (configurable, default 5 as in RocksDB). The latest KV pairs are inserted into the active G-memtable. Every write and update sub-batch creates an active memtable. An active G-memtable is marked immutable before processing subsequent writes or updates sub-batch. The immutable G-memtables are arranged to host KV pairs from newest to oldest. For instance, i^{th} immutable G-memtable contains newer pairs than the $i + 1^{th}$ immutable G-memtable. Once the number of G-memtables reach the threshold, the oldest immutable G-memtable is copied onto the DRAM and then flushed to the LSM tree by background CPU threads.

H-Rocks maintains values in a DRAM-resident *value buffer* to limit data movement over PCIe (§ 3.2). It is mapped to the GPU's address space using UVA and allocated for each G-memtable, then de-allocated once its contents are flushed to the LSM tree.

Figure 3 shows the contents of a G-memtable entry. An entry holds the reqID, key, a pointer to value (valuePtr), and KV size. It keeps the key and value sizes to support variable-sized KV pairs. However, if all keys and/or values are uniform in size, a global size entry for the entire G-memtable is maintained. H-Rocks also maintains a batch identifier (batchID) for each G-memtable (not shown).

Block cache: As in RocksDB, the block cache stores frequently accessed KV pairs. However, in H-Rocks, the block cache is shared between the CPU and GPU. It is allocated using NVIDIA's UVM API. This allows it to be accessible by both CPU and GPU. Further, the UVM driver also migrates contents of the block cache onto HBM on-demand. While the block cache is accessible to both the CPU and GPU, only the CPU is allowed to insert new entries. This limits synchronization overheads and avoids the complexities of accessing the files from the GPU (§ 6.2).

H-Rocks's block cache is a set associative structure with a least frequently used (LFU) replacement policy. Insertions and replacements within a given set occur sequentially on the CPU, while those across different sets occur in parallel. The CPU acquires a write lock on a set before inserting entries. GPU looks up multiple keys in parallel without acquiring locks since it only reads but does *not* insert entries into the block cache. However, if it finds the set is locked by the CPU, it skips it and assumes that the GPU has failed to find the entry. This relieves the GPU from the need to acquire locks and the resultant stalls. We discuss this more in detail in § 6.

The block cache is optimized for real-world access patterns where new KV pairs are typically inserted infrequently while a subset of pairs are read frequently accessed (hot entries). The pages containing hot KV pairs are dynamically migrated onto HBM by the UVM driver, while those containing cold ones stay back in DRAM. This avoids unnecessary data movement while also leveraging available HBM capacity for frequently accessed KV pairs on the GPU.

Figure 4 shows a block cache entry. An entry contains cache tag bits and information for a KV pair. AccessCtr is used by the replacement policy. Upon an access by the GPU, the entry's access is incremented. The GPU uses an atomic add operation for lock-free counter update.

memtableIdx (8B)	oid (8B)	keySz (4B)	key (variable length)	valueSz (4B)	valuePtr (8B)/ value

Fig. 5. A G-WAL entry.

G-write ahead log (G-WAL): The G-WAL facilitates recovery of G-memtables after a crash. It resides in PM and is accessed from the GPU using UVA. G-WAL is created only for write and update sub-batches as they produce new values. A G-WAL entry depends on the request type. Before executing a write sub-batch on the GPU, H-Rocks copies user-provided values for the PUTs in the sub-batch onto a contiguous space in the G-WAL. This optimizes PM bandwidth utilization by persisting the entire batch of values on a contiguous region. This is important since random writes to PM are inefficient [73]. During execution, the GPU logs the key and value pointers. Unlike PUT, there are no user-provided values for DELETEs and UPDATEs. For DELETEs, GPU logs the deleted key with a tombstone marker. For UPDATEs, it logs the key with its modified value.

Figure 5 shows a G-WAL entry. MemtableIdx stores location in G-memtable where a KV pair resides, helping restore G-memtable's contents during recovery. G-WAL keeps reqID, key, and value pointer or value itself, based on request. Note that a G-WAL entry does not contain a lock even though GPU performs parallel logging. This is because the entries of the G-WAL are organized to reflect the GPU kernel's thread hierarchy [50]. Each thread has pre-designated entry(s) in the G-WAL based on its thread ID and, thus, needs no locking.

6 Processing in H-Rocks

While H-Rocks retains RocksDB's ability to service requests on the CPU when needed, we focus on its primary contributions – GPU accelerations and judicious heterogeneous processing. Requests are serviced in three phases. ① Pre-processing on the CPU, ② Execution on the GPU, and ③ Post-processing on the CPU. Pre-processing must be completed before GPU execution begins, whereas post-processing typically overlaps with GPU execution (detailed later). First, we discuss pre-processing, which is similar across all request types. We then discuss how different requests are executed on the GPU and are post-processed.

Pre-processing: During pre-processing H-Rocks ① decides whether and when to accelerate requests on a GPU and ② creates a request batch and allocates associated data structures for GPU execution.

At a low request arrival rate, H-Rocks executes them on the CPU as in RocksDB. If the rate increases, requests queue up. When the queue length crosses a parameterized threshold, H-Rocks executes them on the GPU. The minimum (initial) threshold, here 10K, is chosen to amortize the overheads of initiating GPU execution. H-Rocks dynamically changes the threshold to balance the frequency of initiating GPU execution with the queuing latency. If the current threshold on the queue length is reached within a short time window (e.g., 1 ms) under a high request rate, H-Rocks increases the threshold (configurable; default 10×). The threshold is capped at a maximum value (default 250 million), based on the maximum number of KV pairs that the HBM can accommodate. Similarly, the threshold is halved until the minimum value (parameter) if the request rate continues to fall.

For GPU execution, H-Rocks batches requests into three sub-batches: write, update, and read. Each request in a batch is assigned a unique, monotonically increasing reqID that captures its arrival order. Each batch also has a unique batchID. For every write and update sub-batch, H-Rocks allocates G-memtables on HBM, value buffer on DRAM, and G-WAL on PM. It tags them with a batchID.

For writes, user-provided values are placed in the value buffer. G-memtables are allocated for each write and update sub-batch. The previous active G-memtable is marked as immutable, and the sequence of immutable ones is adjusted accordingly. Once the number of immutable G-memtables reaches the user-defined maximum, the contents of the oldest one are copied to the DRAM and then flushed to the LSM tree by background CPU threads, as in RocksDB. In the special case where the maximum allowed G-memtable size is insufficient to accommodate all new entries to be produced by the requests in a write/update sub-batch, a sub-batch is chunked into smaller groups of requests such that a G-memtable can accommodate entries produced by one chunk. These chunks are executed on

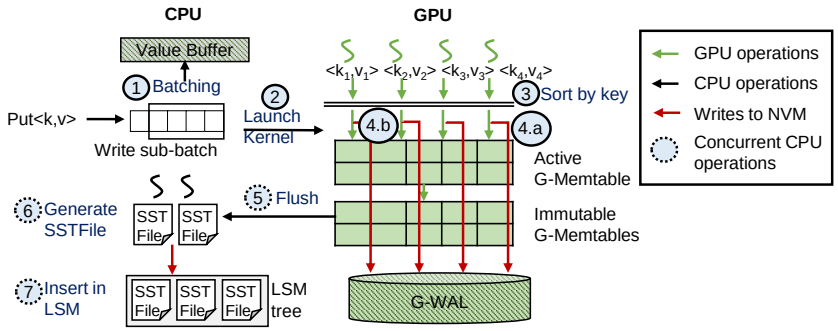

Fig. 6. Processing of write sub-batches on the GPU.

the GPU one after another, with each producing an active G-memtable and correspondingly flushing out the oldest immutable G-memtable.

Read sub-batches do not generate new data, so G-memtable allocation is unnecessary. However, H-Rocks allocates a buffer on HBM to track keys that the GPU fails to find in the G-memtables and block cache. The CPU processes these requests during post-processing. Once a batch is ready, H-Rocks launches the write sub-batch on the GPU, followed by the update and read sub-batches.

6.1 Servicing PUT and DELETE requests

Figure 6 outlines key steps in executing a write sub-batch with PUTs. GPU inserts KV pairs in active G-memtable and G-WAL. Steps (1) and (2) show the pre-processing detailed earlier. Each GPU thread in the kernel executing a write sub-batch is assigned a subset of keys to process (the number of keys in a sub-batch / GPU threads). Each thread inserts its KV pairs in their *correct* position in the active G-memtable on the HBM. Recall that a memtable is organized as a table sorted by keys. H-Rocks uses parallel most-significant-digit radix sort [12] for efficient sorting on the GPU (step (3)). It takes all keys in the sub-batch as input and determines their correct indices in the table.

If a sub-batch contains multiple PUTs for the same key, all versions (values) are maintained in G-memtables, sorted by their reqIDs. This allows later sub-batches to efficiently find appropriate versions to read. Once the position of each key is determined, threads insert the KV pairs into the G-memtable and log them into G-WAL (steps **(4.a)** and **(4.b)**). G-WAL entries are immediately flushed out of HBM for recoverability.

Post-processing: The oldest immutable G-memtable is evicted from HBM when the number of immutables reaches the maximum allowed. It is first copied to DRAM (step (5)). Multiple background **CPU** threads partition its contents and generate SSTFiles using unmodified RocksDB functionalities (step (6)). If an entry for a key exists in the block cache, it is invalidated (not shown). The CPU threads insert SSTFile into the LSM tree using RocksDB's functionalities (step (7)). Further, the G-WAL entries are persisted on the PM by the CPU threads at the end of a write sub-batch (not shown).

DELETEs: As in RocksDB, DELETEs are implemented as PUTs with a special value of 'tombstone marker'. However, unlike PUTs, there are no user-provided values to copy onto the G-WAL during pre-processing. GPU itself logs the 'tombstone marker', along with the key. During compaction, KV pairs with the marker are removed from the LSM tree, as in RocksDB.

6.2 Servicing GET requests

Figure 7 depicts the processing of a read sub-batch with GETs. Steps (1) and (2) show pre-processing. Like the write sub-batch, each GPU thread is assigned a subset of keys to process. A GPU thread

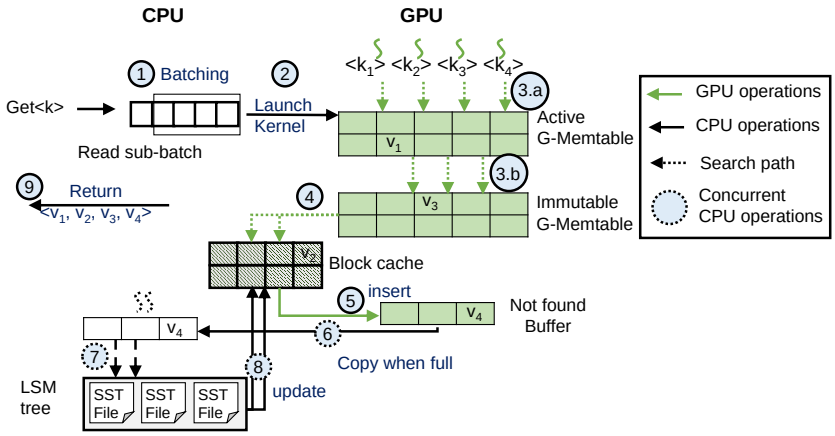

Fig. 7. Processing of read sub-batches on the GPU.

performs a binary search to find matching entries in G-memtables (steps **3.a** and **3.b**). The search can yield multiple candidate KV pairs (versions). The batchID and reqID of all matching pairs (if any) are compared to the reqID of the GET request. A GET returns the version whose batchID is the largest, and the reqID is smaller than that of the GET itself but is the largest amongst the matching entries. This ensures that GETs returns the latest value.

The block cache is searched (step **4**) on a miss to G-memtables. If the key misses or the CPU locks the corresponding set in the block cache, the key is placed into a 'not-found' buffer (step **5**). The buffer keeps keys that the GPU failed to find but *may* exist in the LSM tree.

Post-processing: During post-processing, the CPU finds KV pairs that the GPU might fail to locate. Once the size of the 'not-found' buffer reaches a threshold, GPU raises an interrupt (doorbell) to the CPU to look up KV pairs for the keys in the buffer. This buffer's contents are transferred to the DRAM (step **6**). Multiple CPU threads concurrently search for the keys in the buffer in the LSM tree using RocksDB functionalities. If the keys are found, the KV pairs are inserted into the block cache (step **8**). Finally, the values or 'not-found' messages are returned to the client (step **9**).

True to the spirit of heterogeneous processing, H-Rocks services requests on the GPU while the CPU concurrently searches the LSM tree for keys that the GPU couldn't find. Second, the CPU performs multi-threaded lookup for missing KV pairs. This is facilitated by sub-batching, which allows parallel execution of GETs within a sub-batch to improve throughput.

6.3 Servicing UPDATE requests

An UPDATE is an amalgamation of a GET and a PUT. A sub-batch of updates is formed during pre-processing, and G-memtable, G-WAL, and value buffer are allocated. As in a write sub-batch, the update sub-batch starts by finding indices to insert keys in the G-memtable. Multiple UPDATEs to the same key are sorted by their reqIDs and are processed *sequentially* by a single GPU thread in the increasing order of their reqIDs. This ensures an UPDATE reads the latest value, possibly generated by a prior UPDATE. Updates to distinct keys are processed concurrently by different GPU threads.

A GPU thread reads the appropriate version of the KV pair as in a GET. Specifically, it looks up active, immutable G-memtables and/or the block cache and uses batchID and reqID to find the correct version vis-à-vis its reqID. Thereafter, the value is modified and is written to the value buffer. The pointer to the new value is inserted into the G-memtable. Like PUTs, all versions (values) of a

KV pair from multiple UPDATEs are maintained in the G-memtable and logged into the G-WAL. If a key is not found in G-memtables or block cache, it is inserted into a 'not-found' buffer.
Post-processing: Updates to keys in the 'not-found' buffer are processed on the CPU similar to GETs. Finally, similar to PUTs, G-memtable is converted into SSTFile and flushed to LSM tree.

6.4 Servicing RANGE queries and supporting Transactions

A sub-batch with RANGE queries is executed similarly to a sub-batch of GETs. We expose RANGE (begin_key, end_keys) API for searching for KV pairs that fall within a range of keys. Like a GET, begin_key and end_key for each range query are searched in the G-memtables. The number of matching keys is noted. Memory is allocated on the CPU for all matching KV pairs, and the value pointers are passed from G-memtables to the CPU. During post-processing, the CPU further finds if any KV pairs fall in the given range from the LSM tree and includes them in the results.
Transactions: RocksDB supports atomic transactions, ensuring that all operations within a transaction are either committed or none are [21]. H-Rocks supports atomic transactions by exposing Transaction->Open, Transaction->opType(params..) and Transaction->Close APIs. Operations within a transaction are batched and processed together. Atomicity is ensured by flushing writes and updates to the LSM tree only after all operations are completed. During recovery, if the transaction is fully executed, the keys are recovered; otherwise, the transaction is re-executed.

6.5 Recovery in H-Rocks upon a crash

After a crash, H-Rocks starts in the recovery mode, restoring G-memtable from G-WAL. GPU threads use memtableIdx to insert KV pairs into their correct locations in the recovered G-memtable. If a crash occurs during the execution of a write or an update sub-batch, the sub-batch is relaunched. However, it skips PUTs or UPDATEs whose results have been recovered. H-Rocks thus resumes subsequent sub-batches without re-executing the entire batch. For example, if a crash happens during a read sub-batch, G-memtable from prior write/update sub-batches is recovered, resuming the read sub-batch without re-executing prior sub-batches. A G-WAL is deleted once a G-memtable is flushed to SSTFiles. If a crash occurs while flushing G-memtable to the LSM tree, partially written SSTFiles are discarded and are re-generated from recovered G-memtable. H-Rocks's recovery time is low, needing only 3.5 seconds to recover a G-memtable with 100 million 8-byte KV pairs.

6.6 Consistency guarantees provided by H-Rocks

A goal of H-Rocks is to remain *functionally* indistinguishable from RocksDB to the applications for a wide practical usefulness. Toward this, a key challenge is to adhere to RocksDB consistency guarantees (§ 2), i.e., GETs must return the latest committed write, and no writes/updates must be lost. An UPDATE must modify the latest value. GETs and UPDATEs read the latest value since H-Rocks maintains all versions of a KV pair and returns the correct version using batchID and reqID. Updates are applied sequentially by reqID, ensuring none ca be lost.

H-Rocks can execute requests on the CPU. It is possible that the latest value was written from the CPU (GPU) while the GET executes on the GPU (CPU). Since the CPU and the GPU maintain separate memtables, H-Rocks avoids concurrently inserting requests to the CPU and GPU memtables. Further, if a previous batch of requests was executed on the GPU (CPU), while the next ones would be executed on the CPU (GPU), the memtables in the GPU (CPU) memory are flushed to the LSM tree. This ensures that the LSM tree remains consistent; thus, H-Rocks maintains consistency guarantees.

We empirically stress-tested H-Rocks with four micro-benchmarks to further confirm that it strictly follows the consistency guarantees of RocksDB. The first three ensured that GETs always returned the latest committed value under various scenarios, while the fourth confirmed that no updates

Table 4. Configuration of the evaluation platform.

CPU	4× Intel Xeon Gold 6242 (4 × 16 cores) @ 2.80GHz
GPU	NVIDIA Titan RTX (72 SMs, 24 GB GDDR6)
Memory	768 GB DDR4, 8 x 128 GB Intel Optane PM
Interconnect	PCIe 3.0, 16 GBps
Software	Ubuntu 20.04, CUDA 11.7, PMDK 1.8

are lost. The first micro-benchmark performs thousands of random PUTs and GETs on the *same* key with different values and compares the value returned by each GET in H-Rocks and RocksDB. The second one included operations on a single key and multiple randomly generated keys and values. The third micro-benchmark introduced DELETEs alongside PUTs and GETs, involving operations on single and multiple keys. Finally, the fourth added UPDATEs to random keys, including multiple updates to the same key. We also varied the relative ratios of different types of operations in the micro-benchmarks. Across all experiments, the results under H-Rocks exactly matched that of RocksDB, demonstrating its strict adherence to consistency guarantees.

6.7 Implications of heterogeneous processing

In rare cases where the request rate oscillates between low and high values, H-Rocks may frequently switch between devices and flush memtables. To avoid this, H-Rocks keeps a count of the number of switches per second. If it goes beyond a threshold (default 3), H-Rocks executes all requests on the CPU or the GPU based on the range of request arrival rate. For example, if the request rate mostly varies between a relatively low number, (parameterized, here 50K ops/sec), all the requests are executed on the CPU. If the request rate often reaches high values (e.g., millions of ops/sec), H-Rocks relies on GPU for servicing all requests even when the request rate drops for brief periods.

7 Evaluation

We first qualitatively compare H-Rocks and closely related pKVSs in terms of the features and consistency guarantees each provides. We then quantitatively compare their performances using a set of microbenchmarks and YCSB workloads. To mimic real-world deployment scenarios, we expose each pKVSs to varying request arrival rates (load). We then measure the sustained throughput delivered by each pKVS under these arrival rates. Sustained throughput is the number of requests serviced (completed) per second by a pKVS in the steady state under a given load. The steady-state excludes the warm-up (initial) and cool-down periods (after the workload generator stops generating new requests). It captures the request rate that a system can continue servicing over time and is reported in millions of ops (requests) per second. The sustained throughput matches the request arrival rate when a system is unloaded. However, as the arrival rate increases, the sustained throughput falls behind it as requests queue up after a point. Besides throughput, we demonstrate how H-Rocks leverages both CPU and GPU for low latency under varying request arrival rates in Section 7.3.

Table 5. Properties of various persistent key-value stores.

Name	Variable length KV	Range	Transaction	Consistency
Viper [2]	Yes	No	No	Lost updates
Plush [66]	Yes	Yes	No	Stale KV, lost updates
μTree [7]	No	No	No	Stale KV, lost updates
gpKVS [50]	No	No	No	Stale KV, lost updates
RocksDB [19]	Yes	Yes	Yes	Yes
H-Rocks	**Yes**	**Yes**	**Yes**	**Yes**

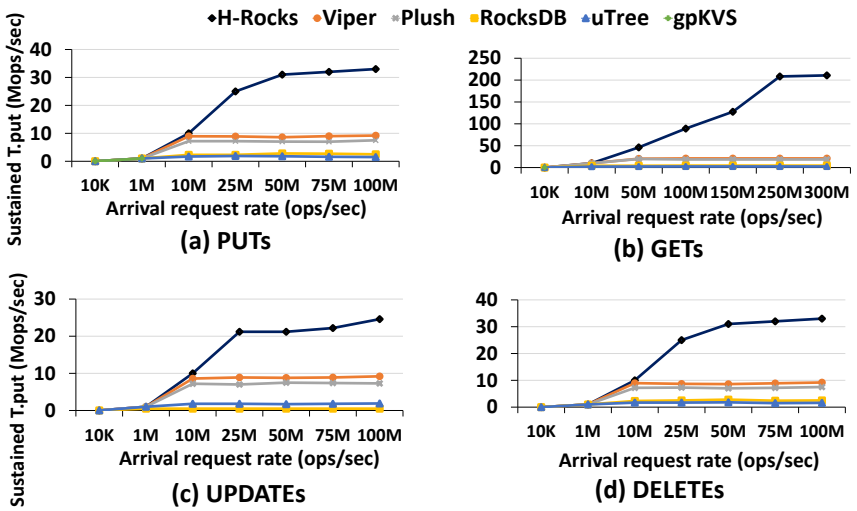

Fig. 8. Throughput evaluation using microbenchmarks.

We configure RocksDB following its tuning guide [20], e.g., with 512MB memtables and up to 5 immutable memtables. While G-memtables can scale to larger sizes, we limit their size and number of immutable G-memtables to match RocksDB for a fair comparison. Later, we demonstrate G-memtable's scalability. Other pKVSs, such as Viper and Plush, are also optimized for their best performance, typically using 16-32 threads.

We evaluate all pKVSs on an evaluation setup mentioned in Table 4. H-Rocks extends RocksDB with about 1000 C++ and 7000 CUDA lines of code. A link to H-Rocks's source code is available at https://github.com/csl-iisc/H-Rocks-SIGMOD25.git.

7.1 Qualitative comparison with other pKVSs

Table 5 lists key features supported (or missing) by various pKVSs, including H-Rocks. Many research pKVSs lack features essential for real-world deployment. For example, gpKVS only supports KV sizes of 8-bytes, while μTree supports only 8-byte keys. Viper, μTree and gpKVS do not support range queries. None support transactions. Besides RocksDB, only H-Rocks supports features such as range queries, variable-sized KV pairs, and transactions.

We then analyzed if these pKVSs support consistency guarantees of RocksDB. Adherence to these guarantees is necessary for a new design to preserve the application behavior. We notice that Viper loses updates when two threads concurrently update the *same* key since it avoids locks while reading. Plush can return a stale value when multiple PUTs and DELETEs are performed to the same key. Plush keeps multiple versions of the same key across multiple levels. During migration across levels, GETs can read a stale pair while it is being deleted. Plush also loses updates when two threads try to update the same KV pair. Similarly, μTree may not return the latest value and can lose updates too. gpKVS does not provide any guarantees as it does not control the order in which concurrent GPU threads read or write a KV pair. In contrast, H-Rocks does not suffer from these deficiencies.

7.2 Evaluation with microbenchmarks

We pre-load pKVSs with 100 million 8-byte KV pairs before executing each microbenchmark. The sizes of keys and values sizes are set to 8 bytes for microbenchmarks unless mentioned otherwise. We later demonstrate H-Rocks's performance in the presence of varying KV sizes. The keys are generated randomly, and the request arrival rates follow uniform distributions for microbenchmarks.

PUTs: Figure 8(a) shows the sustained throughput (y-axis) of various pKVSs against varying PUT request rates (x-axis). H-Rocks achieves a high sustained throughput of 33 Mops/sec under a request rate of 100 Mops/sec. Improvements stem from parallel, lock-free insertions of KV pairs to HBM-resident G-memtables, with minimal data movement due to storing only pointers in HBM.

H-Rocks outperforms Viper, Plush, μTree, gpKVS and RocksDB by up to 3.5×, 4.4×, 13.2×, 15×, and 17.5×, respectively. Viper and Plush are the closest competitors, managing to sustain throughput of ~ 10 MOps/sec. The performance of Viper, Plush, μTree are hamstrung by the CPU's limited parallelism and DRAM bandwidth. While gpKVS uses GPU, it is constrained by the bandwidth of the PCIe and PM. Unlike H-Rocks, it maintains the hashmap on PM and fails to leverage the HBM. Consequently, it performs all writes to the PM over the PCIe. RocksDB's throughput remains under 2 MOps/sec due to limited concurrency.

GETs: Figure 8(b) shows the same for GETs. H-Rocks achieves a significantly higher sustained throughput of 210 MOps/sec for GETs. This is because, unlike PUTs, there is no need to write to PM-resident G-WAL. Importantly, heterogeneous processing helps speed up H-Rocks significantly. The requests that hit in G-memtables or in the block cache are accelerated manifold by the GPU. The GPU puts the keys that are missed in these in-memory structures in the 'not-found' buffer. In the true spirit of heterogeneous processing, tens of CPU threads look up the LSM tree for these missing keys concurrently with the GPU execution. The performance of GET is primarily capped by the maximum allowed batch size, which, in turn, is limited by HBM capacity (§ 6, preprocessing).

H-Rocks outperforms Viper, Plush, μTree, gpKVS and RocksDB by up to 10.54×, 11.3×, 84.3× 40×, and 46.8×, respectively. Among competitors, Viper achieves the highest sustained throughput of 20 Mops/sec by performing concurrent lookups in its volatile index but must still fetch KV pairs from the PM. Plush maintains *only* the lowest level hashtable (L0) of the LSM tree in the DRAM. Hence, it ends up servicing many requests from the PM-resident LSM tree. μTree concurrently look up the volatile inner nodes of the B+tree, but is limited by CPU's parallelism as are Viper and Plush. gpKVS fails to harness GPU's parallelism since each GET must read from the PM over the PCIe. Finally, RocksDB's throughput is limited to 4 Mops/sec, constrained by the CPU's parallelism.

UPDATEs: Figure 8(c) reports UPDATE's throughput. We perform increment operations on randomly selected keys to trigger UPDATEs. H-Rocks outperforms Viper, Plush, μTree and RocksDB by up to 3×, 2.6×, 2.5× and 14.6×, respectively and achieves up to 24.6 Mops/sec of sustained throughput. gpKVS does not support UPDATEs. Unlike prior works, H-Rocks harness GPU's parallelism to update KV pairs in the HBM-resident G-memtable. GPU-optimized lock-free logging (HCL) further aids performance. Notice that the throughput for UPDATEs is lower than that of PUTs. This is because UPDATEs involve additional steps, such as looking up the key, fetching value from the value buffer, and writing updated values back.

DELETEs: Figure 8(d) shows the performance of DELETEs. DELETE is similar to PUT in H-Rocks, Plush, and RocksDB as it simply inserts a tombstone marker as the value in the deleted KV pair. In Viper, the performance is slightly worse than PUTs, as it first performs a lookup in the volatile index

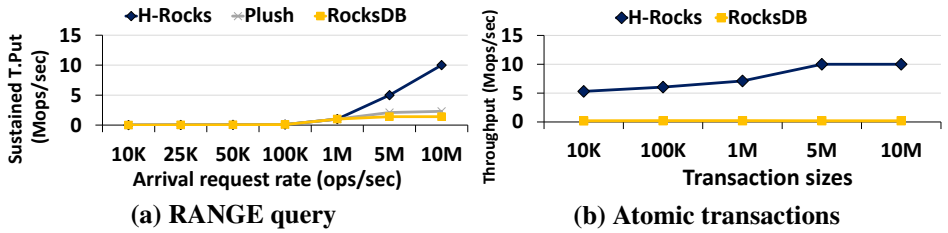

Fig. 9. Throughput of RANGE and atomic transactions.

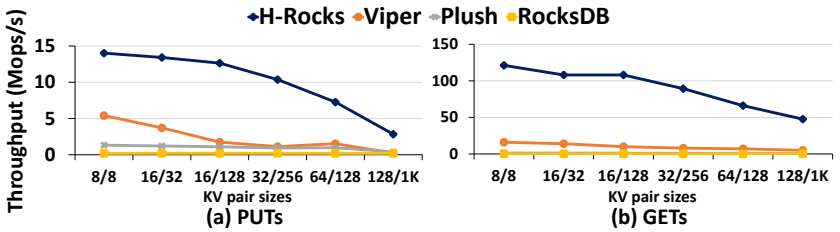

Fig. 10. PUTs and GETs with varying key-value sizes.

and then marks the KV pair for deletion in the pKVS. μTree searches for the key to be deleted in the B+tree and then marks it for deletion. gpKVS does not support the delete operation. In the case of DELETE too, H-Rocks sustains significantly better throughput than its competitors.

RANGE: Figure 9(a) shows the sustained throughput of H-Rocks, Plush, and RocksDB for RANGE queries with varying arrival rates. Each query retrieves approximately 100 pairs. H-Rocks sustains a significantly higher throughput of up to 10 Mops/sec, compared to 2.2 and 1.4 Mops/sec for Plush and RocksDB, respectively. Other pKVSs do not support range queries.

Atomic transactions: Only RocksDB and H-Rocks support transactions. To evaluate transaction performance, we perform ten million requests (50% GETs and 50% PUTs) while varying the number of requests per transaction (i.e., transaction size). Figure 9(b) shows throughput (y-axis) with varying transaction sizes (x-axis). At a transaction size of five million, H-Rocks can process two such transactions in a second, delivering a throughput of 10 MOps/sec. Leveraging GPU's parallelism, H-Rocks significantly outperforms RocksDB.

7.3 Analyzing H-Rocks's behavior

Varying KV sizes: We first study how pKVS behaves with varying sizes of KV pairs to assess its generality. Sub-figures in Figure 10 show the impact of varying KV sizes on the throughput when performing 10 million PUTs and GETs. Note that μTree and gpKVS do not support variable-size KV pairs. On the x-axis, KV sizes are denoted as key size/value size, e.g., "16/128" represents a 16-byte key and 128-byte value. The y-axis shows throughput. As expected, throughput of all pKVS reduce with larger KV sizes. However, H-Rocks continues to outperform competitors across KV sizes.

Figure 10(a) shows throughput for PUTs. In H-Rocks, larger keys increase the time to copy them to the HBM and that to sort the G-memtable. The time to insert into G-memtable is agnostic to the value size due to the use of pointers. However, the time to write to the G-WAL increases. Throughput of Viper and Plush decrease linearly with increasing KV size due to limited PM bandwidth. RocksDB's slows down with large KV sizes due to frequent flushing of memtables to SSTFiles.

Figure 10(b) shows the throughput for GETs. H-Rocks's throughput decreases with increasing key sizes but remains mostly unaffected by larger value sizes. Its use of value pointers minimizes the overhead of copying values across PCIe. Nonetheless, larger keys increase read sub-batch setup and G-memtable search times. Meanwhile, throughputs of Plush and Viper decreases with both large key and value sizes as they become PM bandwidth-bound. The throughput of RocksDB decreases with larger KV pairs as fewer requests are serviced from memtables and block caches.

Benefits of value pointers: Figure 11 quantifies the benefits of keeping only the pointers in G-memtables while the values reside on the DRAM. We report the throughput achieved by H-Rocks and that when it is forced to keep values in G-memtable ('H-Rocks w/ values'). We perform ten million requests for write-intensive (R50:W50) and read-intensive (R90:W10) workloads, reported in Figure 11(a) and Figure 11(b), respectively. On the x-axis, we vary the value size. The y-axis reports throughput. The key size is kept unchanged at 8 bytes.

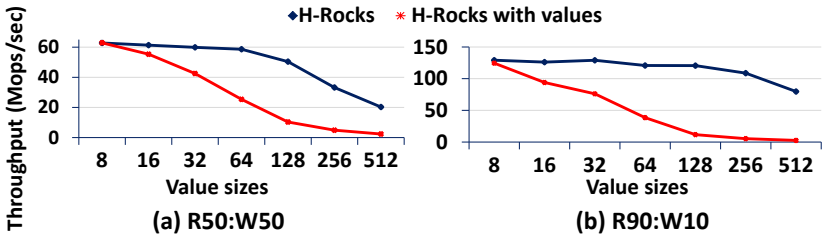

Fig. 11. Impact of value pointers with varying value sizes.

Table 6. Lookup time (μsecs) with increasing G-memtable sizes.

G-memtable Size	64MB	128MB	256MB	512MB	1GB	2GB	4GB
Lookup time	3.10	3.12	3.14	3.18	3.22	3.2	3.28

H-Rocks performs significantly better when using pointers for both write-intensive (up to 10×) and read-intensive workloads (up to 40×), especially with large value sizes. However, there is no advantage when values are 8 bytes, as pointers are also 8 bytes. Larger values increase PCIe data movement overhead, requiring more bytes to transfer to HBM-resident G-memtables and later back to DRAM or the LSM tree during flushing. For GETs, it must return entire values over the PCIe instead of pointers to entries in value buffers or block cache.

Even with pointers, H-Rocks's throughput decreases for write-intensive workloads with large values, primarily due to the longer latency of persisting these values to the G-WAL. In contrast, read-intensive workloads remain largely unaffected as GETs do not log values.

Scalability of larger G-memtable sizes: Unlike RocksDB's memtables, G-memtable is designed to scale effectively. Previously, its size was conservatively matched to RocksDB's best memtable size for fair comparison. Here, we demonstrate the scalability of G-memtable's design.

Table 6 reports the average lookup latency of 100 million 8-byte KV pairs to G-memtable with increasing G-memtable sizes. The latency remains unchanged across various G-memtable sizes (3.1-3.28 seconds), unlike RocksDB's memtable, which increases with size (Table 3). The traditional skip-list-based design of memtable involves pointer chasing to lookup a key. In contrast, G-memtable efficiently performs binary searches on its sorted tables. While the access latency remains unaffected, larger G-memtables typically produce more hits and thus limit accesses to the LSM tree.

Latency across varying request rates: Hitherto, we focused on throughput. We now discuss how H-Rocks optimizes latency under varying request arrival rates using heterogeneous processing. Figure 12 shows the median latency (y-axis) of PUTs and GETs with varying requests arrival rate (x-axis) for RocksDB and H-Rocks. At lower rates, H-Rocks matches RocksDB's latency as the CPU services requests. At higher rates, RocksDB experiences significant queuing and longer latencies. H-Rocks batches the requests and executes them on the GPU to limit queuing, achieving up to 3× lower latency. The performance gap between H-Rocks and RocksDB widens as request rates increase.

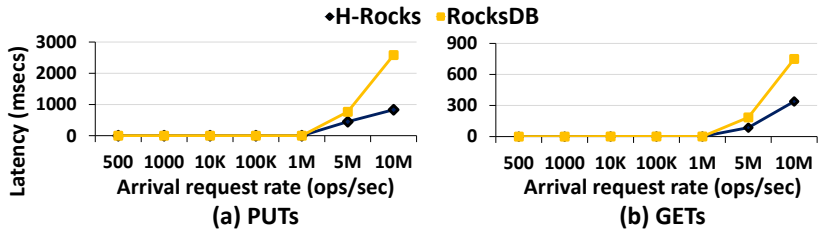

Fig. 12. PUT and GET latencies with varying request rate.

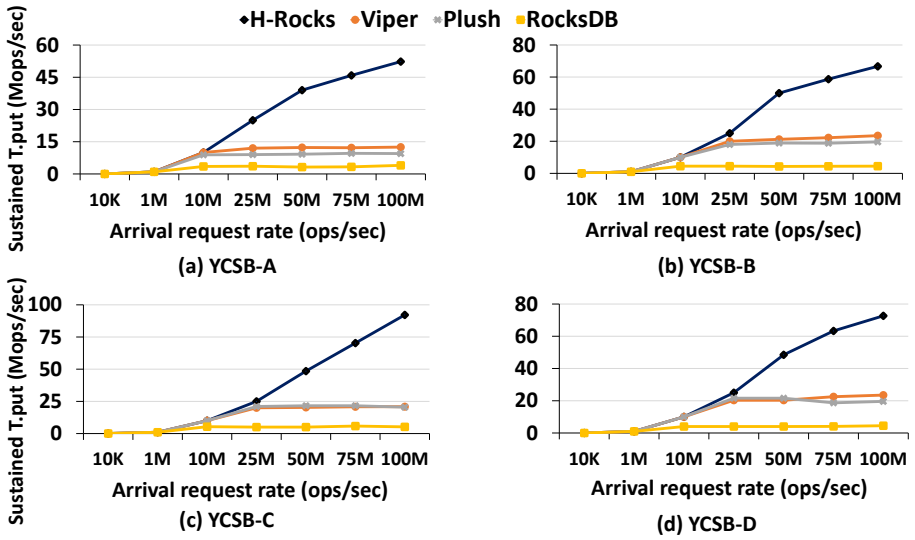

Fig. 13. Throughput for YCSB workloads.

7.4 Evaluation with YCSB workloads

We also evaluate using the popular Yahoo Cloud Serving Benchmark (YCSB) workload generator [8] with four configurations – YCSB-A (write-intensive), YCSB-B (read-heavy), YCSB-C (read-only), and YCSB-D (read-heavy with reads to latest data). We use YCSB's default configuration where keys for PUTs are generated with a uniform distribution while that for GETs follow a Zipfian distribution with a coefficient of 0.99. The key size is 16 bytes, and the value size is 100 bytes. The pKVSs are prefilled with an initial set of 100 million KV pairs before the measurements begin.

Subgraphs in Figure 13 report sustained throughputs (y-axis) of different pKVSs for YCSB workloads (A-D) with varying request arrival rates (x-axis). H-Rocks achieves a sustained throughput upto 52 MOps/sec, 64 MOps/sec, 92 MOps/sec and 72 MOps/sec, respectively for YCSB-A to D. Amongst other pKVSs, Viper performs the best. However, even Viper could achieve only 12 Mops/sec, 19 Mops/sec, 21 Mops/sec, and 18.7 Mops/sec, respectively. We could not evaluate μTree and gpKVS as they only support 8-byte keys.

The reasons for significant speedups for H-Rocks over its competitors vary based on workloads. The PUTs in write-intensive workloads benefit from GPU-accelerated insertions to G-memtables and block cache, besides the fast logging to PM directly from the GPU. The GETs not only benefit from GPU-accelerated lookups to G-memtables and to the block cache but also from concurrent (to GPU execution) and multi-threaded lookup of missing keys in the LSM tree at the CPU. The GETs also typically enjoy better hit rates in similarly sized G-memtable for larger value sizes due to the use of value pointers. Consequently, H-Rocks provides higher throughput for read-mostly workloads.

YCSB-A is a write-intensive workload with 50:50 GETs and PUTs interspersed with each other. While PUTs benefits primarily from GPU acceleration, GETs benefits from both GPU acceleration and concurrent heterogeneous processing across the CPU and GPU. On the other extreme is the YCSB-C workload that only has GETs and, consequently, sustains a higher thought as GETs enables higher sustained throughput as explained with microbenchmarks. Interestingly YCSB-C's throughput can be slightly higher than the microbenchmark with GETs (Figure 8(b)). This is because keys are randomly generated for microbenchmarks while YCSB-C's keys follow Zipfian distribution (skewed), i.e., a subset of keys are looked up more frequently. Consequently, it generates more hits in the G-memtable and block caches and, thus, benefits from GPU acceleration for a larger fraction of

requests. The rest of the requests that miss in these in-memory data structures still benefit from concurrent lookups by multi-threaded CPU programs but cannot match GPU's parallelism.

YCSB-B and YCSB-D are both read-heavy workloads with slightly different behaviors. In YCSB-D, keys that are inserted recently (i.e., latest data) are read back frequently. This leads to higher hit rates in the G-memtables. For YCSB-B, although a set of keys are read more frequently, they are not necessarily the latest data. Consequently, YSCB-B witnesses relatively fewer hits in G-memtables and more in the block cache. A larger fraction of requests in YCSB-B is also serviced from the LSM tree by the CPU. Thus, H-Rocks delivers slightly less throughput for YCSB-B than YCSB-D.

Summary: H-Rocks achieves 3-18× more throughput across diverse request types and configurations over its closest competitors by harnessing GPU and heterogeneous processing. It achieves this while retaining RocksDB's rich features and consistency guarantees.

8 H-Rocks and disaggregated KVS

H-Rocks's primary goal is to improve throughput. It is natural to wonder how H-Rocks may fare against the traditional alternative of disaggregated persistent KVSs, such as FoundationDB (FDB)[78], and disaggregated RocksDB [16]. These distribute requests across multiple shards to scale up. We first analyze the cost-performance analysis of H-Rocks and a disaggregated pKVS. We then discuss how H-Rocks can be extended to work under disaggregation seamlessly.

8.1 Cost versus performance analysis

We quantitatively compare H-Rocks with FoundationDB (FDB) [78], a state-of-art persistent KVS [2]. We analyze FDB's scalability under ideal conditions by deploying multiple server instances on a four-socket server (please refer to Table 4) with zero network latency amongst instances. Each instance manages a disjoint subset of keys, and a client issues multiple requests in a transaction, following FDB's execution model. Each instance keeps its part of the KVS on PM.

Figure 14 demonstrates how FDB's throughput scales under various YCSB workloads with an increasing number of instances. We initially prefill the KVS with one million KV pairs and then perform one million operations. The throughput increases by up to 4.7-5.2× as the number of server instances is scaled up from one to seven before plateauing. This establishes how FDB can improve throughput through disaggregation on our setup.

We now explore if a single node of H-Rocks, deploying a GPU, can be more cost-effective than disaggregated FDB deploying several CPU-only instances. We measure the performance (throughput) per dollar (Ops/sec/$) for the two aforementioned alternatives (higher is better). For FDB, we calculate the maximum throughput achieved with seven instances, divided by the cost of the four-socket server (Table 4), *excluding* the GPU. For H-Rocks, we consider the maximum sustained throughput while adding the cost of the GPU. Note that this is conservative for H-Rocks since it only uses one of the four sockets in the server, unlike FDB. The total cost of our setup, detailed in Table 4, totals 7657

[2]We chose FDB since disaggregated RocksDB could not run on our system. It is built atop of Tectonic FS [48] that does not support PM. Recall that H-Rocks leverages PM for logging from the GPU.

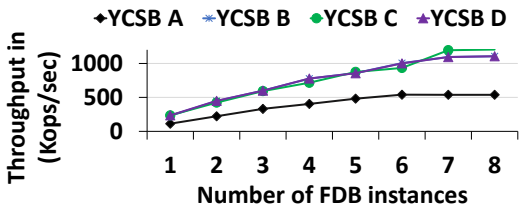

Fig. 14. FDB's performance for YCSB workloads.

Table 7. Ops/sec per $ for FDB and H-Rocks for YCSB workloads.

	YCSB-A	YCSB-B	YCSB-C	YCSB-D
FDB	85.23	174.79	190.58	172.64
H-Rocks	7240.43	8469.94	11202.18	8572.36

USD, with the GPU contributing only 1337 USD or 17.5% of the total cost. This is because we use only a relatively cheap *consumer-grade* GPU for H-Rocks.

We report the throughput-per-dollar measurements in Table 7 for the YCSB workloads. The throughput enabled by H-Rocks for each dollar spent is $48 - 85\times$ higher than a state-of-the-art distributed KVS like FDB. This firmly establishes that to achieve higher throughput in a given dollar budget, H-Rocks is the better choice.

8.2 H-Rocks in a distributed environment

We now explore how H-Rocks itself can be deployed in a disaggregated environment. H-Rocks is designed to achieve high throughput of data operations, e.g., GET, PUT, on a single node. Importantly, it retains the same guarantees of a RocksDB node and does not interfere with the control plane functionalities of a disaggregated KVS, such as distributed transaction management, replica management, load balancing, and key-space partitioning. From the perspective of the control plane, a H-Rocks node is *functionally indistinguishable* from a RocksDB node. This allows H-Rocks to be seamlessly incorporated as (data) node(s) in a disaggregated deployment.

We detail how H-Rocks can work with the primary disaggregation functionalities – distributed transaction management and replica management of both FDB and disaggregated RocksDB (dRDB). FDB divides a transaction into sub-transactions based on the sharded key space and distributes them to respective shards. Shards using H-Rocks would benefit from GPU acceleration, enabling faster turnaround. Each shard notifies the manager upon completing its sub-transaction. After receiving completion messages from all shards, the manager instructs them to commit transactions locally. Since H-Rocks supports atomic transactions, it can execute and commit sub-transactions seamlessly. If any shard fails to commit, the manager rolls back the entire transaction. Thus, incorporating a shard running H-Rocks needs no alteration to the distributed transaction manager.

Next, we discuss how FDB manages replicas, particularly in the presence of write-heavy workloads [22]. Write requests by clients are first logged by the transaction managers. The writes are then propagated to all the replicas for writing them to the shards that they map to. One or all of these replicas can be hosted by a H-Rocks node without needing any alteration in the replica management.

Unlike FDB, dRDB does not support atomic transactions across multiple shards [16]. For write requests, dRDB's replica management designates one of the shards (replicas) among all shards mapping the given key(s) as the *writer*. A writer shard, including those running H-Rocks performs the write as usual in RocksDB (e.g., PUTs). While the shards running H-Rocks can significantly accelerate these PUTs using a GPU, they do not alter functionality. After the writer completes the write in its shard, the replica manager asynchronously updates the other replicas in the background. For example, the updated SST files created by the writer are copied to the replicas to update their respective LSM trees. Note that H-Rocks does not alter the process of generation of SST files as it happens unmodified on the CPU. In short, dRDB's replica management remains unaltered even if one or more replicas are maintained by H-Rocks.

While it is evident that H-Rocks's design seamlessly aligns with disaggregation control logic, extending it to operate under disaggregation is a project by itself, which we leave as future work.

9 Related works

Many popular KVSs such as Redis [51], memcached [40], and MICA [34] provide only in-memory caching, sacrificing recoverability to avoid the cost of persistence. MegaKV [77] is an in-memory KVS leveraging GPU acceleration.

Persistent KVSs like RocksDB [60], LevelDB [32], and Cassandra [5] store data on SSD or HDD. FASTER [6] combines an in-memory hash index with a hybrid log to store records but risks losing updates by keeping frequently modified KV pairs in a volatile tail. A recent work adapts RocksDB to a dis-aggregated setup [17]. While H-Rocks focuses on GPU and PM acceleration.

Many recent studies focus on efficient pKVS implementations using PM. They often leverage DRAM for performance and PM for persistence. For example, Viper [2] stores KV pairs in PM but uses a DRAM hashmap for fast lookups. It adopts a PM-aware storage layout to minimize random writes to PM. Plush [66] replaces the LSM tree layers with a hashmap but retains its merge and compact properties. μTree [7] employs a B+tree, with the internal B+tree nodes in DRAM and leaf nodes in PM. RocksDB-pmem [19] extends RocksDB and places the LSM tree and WAL on PM. Unlike H-Rocks, these works are limited by CPU parallelism and DRAM bandwidth.

HiKV [70] combines a PM hash index with a DRAM-resident B-tree. Similarly, FPTree [47] and DPTree [79] are persistent B-trees that store the inner nodes or keys in DRAM. Both only support 8-byte KV sizes and a limited set of operations. They often lack consistency guarantees, unlike H-Rocks. None leverage a GPU. FluidKV [36] combines an in-memory B+ tree and a PM based LSM tree. BonsaiKV [3] consists of an in-memory Masstree index and stores data in PM. The only publicly known work that leverages both PM and GPU, gpKVS [50], has several significant shortcomings as detailed in the § 1.

Works such as Dash [35], CCEH [43], IcebergHT [49] and Intel pmemkv [28] are *PM-only* hashtables and are hence constained by PM's bandwidth. They group buckets into a segment to reduce PM writes. IcebergHT employs a new scalable and low associativity hashing mechanism. While pmemkv supports all KV sizes, Dash, CCEH, and IcebergHT only support 8-byte key sizes. Apart from pmemkv, none support range queries or atomic transactions. Spash [76] another PM-only hashtable, limits the writes to PM by leveraging persistent CPU caches enabled by Intel's eADR technology [29] and improves concurrency by exploiting hardware transactional memory. However, Spash's need for persistent CPU caches and transactional memories limits its applicability.

Some works adopt a hybrid design spanning between DRAM, PM, and SSDs. NoveLSM [30] places its memtables in PM and LSM tree in SSD. While MatrixKV [74] places its memtables in DRAM, lower levels of LSM tree in PM and higher levels in SSD. However, none use GPUs.

10 Conclusion

H-Rocks demonstrates the potential of judiciously leveraging GPUs to accelerate persistent key-value stores. We show that it is possible to retain compatibility to commercially deployed key-value stores such as RocksDB and yet achieve manifold speedups across popular workloads through a careful division of labor between the GPU and the CPU. Techniques such as sub-batching, versioning, efficient GPU-accelerated logging, and the use of UVA and UVM technologies of modern GPUs help match the need of the software with the underlying hardware.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We thank Jayant R. Haritsa and Srinivas Karthik V. for their feedback on the draft. This research is partially supported by the Google India Research Awards. Shweta is supported by the Google India Ph.D. fellowship.

References

[1] Andrew Kryczka Abhishek Madan. 2018. DeleteRange: A New Native RocksDB Operation. https://rocksdb.org/blog/2018/11/21/delete-range.html.

[2] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proc. VLDB Endow.* 14, 9 (oct 2021), 1544–1556. https://doi.org/10.14778/3461535.3461543

[3] Miao Cai, Junru Shen, Yifan Yuan, Zhihao Qu, and Baoliu Ye. 2024. BonsaiKV: Towards Fast, Scalable, and Persistent Key-Value Stores with Tiered, Heterogeneous Memory System. *Proc. VLDB Endow.* 17, 4 (March 2024), 726–739. https://doi.org/10.14778/3636218.3636228

[4] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 209–224.

[5] Cassandra. 2023. Open Source NoSQL Database. https://cassandra.apache.org/_/index.html. Accessed: 2023-06-22.

[6] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3183713.3196898

[7] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. UTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2634–2648. https://doi.org/10.14778/3407790.3407850

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[9] CXL. 2022. CXL Consortium and JEDEC Sign MOU Agreement to Advance DRAM and Persistent Memory Technology. https://www.computeexpresslink.org/_files/ugd/0c1418_7f109df8ee1c4d958cb21c24a669caa5.pdf.

[10] CXL. November 2020. CXL Consortium. Compute Express Link Specification Revision 2.0. https://www.computeexpresslink.org/download-the-specification.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[12] Aditya Deshpande and P J Narayanan. 2013. Can GPUs sort strings efficiently?. In *20th Annual International Conference on High Performance Computing*. 305–313. https://doi.org/10.1109/HiPC.2013.6799129

[13] Siying Dong. 2015. WritePrepared Transactions. WriteBatchWithIndex:UtilityforImplementingRead-Your-Own-Writes. Accessed: 2023-06-22.

[14] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (oct 2021), 32 pages. https://doi.org/10.1145/3483840

[15] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (oct 2021), 32 pages. https://doi.org/10.1145/3483840

[16] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proc. ACM Manag. Data* 1, 2, Article 192 (June 2023), 24 pages. https://doi.org/10.1145/3589772

[17] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proc. ACM Manag. Data* 1, 2, Article 192 (jun 2023), 24 pages. https://doi.org/10.1145/3589772

[18] Pintrest Engineering. 2024. Building Pinterest's new wide column database using RocksDB. https://medium.com/pinterest-engineering/building-pinterests-new-wide-column-database-using-rocksdb-f5277ee4e3d2.

[19] Facebook. 2023. Pmem-RocksDB. https://github.com/pmem/pmem-rocksdb.

[20] Facebook. 2023. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[21] Facebook. 2023. Transactions. https://github.com/facebook/rocksdb/wiki/Transactions. [Accessed 17-07-2024].

[22] FoundationDB. 2019. Technical Overview of the Database. https://github.com/apple/foundationdb/wiki/Technical-Overview-of-the-Database.

[23] Ankit Gupta. 2016. FollowFeed: LinkedIn's Feed Made Faster and Smarter. https://engineering.linkedin.com/blog/2016/03/followfeed--linkedin-s-feed-made-faster-and-smarter. Accessed: 2023-06-22.
[24] Ethan Hamilton. 2020. RocksDB Is Eating the Database World. https://rockset.com/blog/rocksdb-is-eating-the-database-world/. Accessed: 2023-06-22.
[25] Mark Harris. 2013. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/. Accessed: 2025-12-01.
[26] Mark Harris. 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.
[27] Intel. 2021. Intel Optane Persistent Memory. https://www.intel.in/content/www/in/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed: 2021-11-15.
[28] Intel. 2021. Intel PmemKV. https://github.com/pmem/pmemkv.
[29] Intel. 2021. Section "Power-Fail Protected Domains" of "Persistent Memory Learn More Series Part 2". https://www.intel.com/content/www/us/en/developer/articles/training/pmem-learn-more-series-part-2.html. Accessed: 2021-11-15.
[30] 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 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 993–1005.
[31] Jay Kreps. 2016. Introducing Kafka Streams: Stream Processing Made Simple. https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/. Accessed: 2023-06-22.
[32] leveldb. 2023. LevelDB. https://dbdb.io/db/leveldb. Accessed: 2023-06-22.
[33] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (jan 2020), 94–110. https://doi.org/10.1109/tpds.2019.2928289
[34] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim
[35] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134
[36] Ziyi Lu, Qiang Cao, Hong Jiang, Yuxing Chen, Jie Yao, and Anqun Pan. 2024. FluidKV: Seamlessly Bridging the Gap between Indexing Performance and Memory-Footprint on Ultra-Fast Storage. *Proc. VLDB Endow.* 17, 6 (May 2024), 1377–1390. https://doi.org/10.14778/3648160.3648177
[37] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705
[38] Scott Mansfield. 2016. Application data caching using SSDs. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef. Accessed: 2023-06-22.
[39] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3217–3230. https://doi.org/10.14778/3415478.3415546
[40] memcached. 2023. Memcached. https://memcached.org/. Accessed: 2023-06-22.
[41] Meta. 2017. LogDevice: a distributed data store for logs. https://engineering.fb.com/2017/08/31/core-data/logdevice-a-distributed-data-store-for-logs/. Accessed: 2023-06-22.
[42] Nurit Moscovici, Nachshon Cohen, and Erez Petrank. 2017. A GPU-Friendly Skiplist Algorithm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 246–259. https://doi.org/10.1109/PACT.2017.13
[43] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST'19)*. USENIX Association, USA, 31–44.
[44] Satheesh Nanniyur. [n. d.]. Sherpa: scales new heights. https://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights. Accessed: 2023-06-22.
[45] Nvidia. 2011. Peer-to-Peer and Unified Virtual Addressing. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.
[46] NVIDIA. 2023. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/.
[47] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing

Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251
[48] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar, Michael Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *USENIX Conference on File and Storage Technologies*. https://api.semanticscholar.org/CorpusID:232225564
[49] Prashant Pandey, Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2023. IcebergHT: High Performance Hash Tables Through Stability and Low Associativity. *Proc. ACM Manag. Data* 1, 1, Article 47 (may 2023), 26 pages. https://doi.org/10.1145/3588727
[50] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: Leveraging Persistent Memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 142–156. https://doi.org/10.1145/3503222.3507758
[51] redis. 2023. Redis. https://redis.io/. Accessed: 2023-06-22.
[52] RocksDB. 2022. Compaction. https://github.com/facebook/rocksdb/wiki/Compaction.
[53] RocksDB. 2022. Iterator. https://github.com/facebook/rocksdb/wiki/RocksDB-Overview.
[54] RocksDB. 2022. RocksDB Users and Use Cases. https://github.com/facebook/rocksdb/wiki/RocksDB-Users-and-Use-Cases.
[55] RocksDB. 2022. Transactions. https://github.com/facebook/rocksdb/wiki/Transactions.
[56] RocksDB. 2023. Basic Operations. https://github.com/facebook/rocksdb/wiki/Basic-Operations.
[57] RocksDB. 2023. Block Cache. https://github.com/facebook/rocksdb/wiki/Block-Cache.
[58] RocksDB. 2023. MemTable. https://github.com/facebook/rocksdb/wiki/MemTable.
[59] RocksDB. 2023. Merge Operator. https://github.com/facebook/rocksdb/wiki/Merge-Operator.
[60] rocksdb. 2023. RocksDB. https://rocksdb.org/. Accessed: 2023-06-22.
[61] RocksDB. 2023. RocksDB overview. https://github.com/facebook/rocksdb/wiki/RocksDB-Overview.
[62] RocksDB. 2023. Write Ahead Log (WAL). https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log/.
[63] RocksDB. 2023. Write stalls. https://github.com/facebook/rocksdb/wiki/Write-Stalls.
[64] Andy Rudoff. 2021. Persistent Memory on CXL. https://www.snia.org/educational-library/persistent-memory-cxl-2021. Accessed: 2021-11-15.
[65] Samsung. August 2022. Samsung Memory-Semantic CXL SSD. https://tinyurl.com/3daezum2.
[66] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. *Proc. VLDB Endow.* 15, 11 (sep 2022), 2895–2907. https://doi.org/10.14778/3551793.3551839
[67] J. Wang. 2017. Myrocks: best practice at alibaba. https://www.percona.com/live/17/sessions/myrocks-best-practice-alibaba. Accessed: 2023-06-22.
[68] Business Wire. 2022. Everspin Announces New STT-MRAM EM128LX xSPI Memory. https://www.businesswire.com/news/home/20220801005856/en/Everspin-Announces-New-STT-MRAM-EM128LX-xSPI-Memory. Accessed: 2025-01-06.
[69] Business Wire. 2022. Kioxia Launches Second Generation of High-Performance, Cost-Effective XL-FLASH™ Storage Class Memory Solution. https://www.businesswire.com/news/home/20220801005862/en/Kioxia-Launches-Second-Generation-of-High-Performance-Cost-Effective-XL-FLASH-Storage-Class-Memory-Solution/. Accessed: 2025-01-06.
[70] 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 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 349–362.
[71] Maxim Fateev Xu Ning. 2016. Cherami: Uber Engineering's Durable and Scalable Task Queue in Go. https://www.uber.com/en-IN/blog/cherami-message-queue-system/. Accessed: 2023-06-22.
[72] Maysam Yabandeh. 2017. WritePrepared Transactions. https://rocksdb.org/blog/2017/12/19/write-prepared-txn.html. Accessed: 2023-06-22.
[73] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 169–182.
[74] 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 a Matrix Container in NVM. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 2, 15 pages.
[75] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2491–2503. https://doi.org/10.14778/

3551793.3551809

[76] Bowen Zhang, Shengan Zheng, Liangxu Nie, Zhenlin Qi, Linpeng Huang, and Hong Mei. 2024. Exploiting Persistent CPU Cache for Scalable Persistent Hash Index. https://venero.github.io/files/publications/2024-ICDE.pdf. [Accessed 17-07-2024].

[77] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of in-Memory Key-Value Stores. *Proc. VLDB Endow.* 8, 11 (July 2015), 1226–1237. https://doi.org/10.14778/2809974.2809984

[78] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2022. FoundationDB: A Distributed Key Value Store. *SIGMOD Rec.* 51, 1 (June 2022), 24–31. https://doi.org/10.1145/3542700.3542707

[79] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment* 13 (12 2019), 421–434. https://doi.org/10.14778/3372716.3372717

Received July 2024; revised September 2024; accepted November 2024.