KVCG: A Heterogeneous Key-Value Store for Skewed Workloads

dePaul Miller, Jacob Nelson, Ahmed Hassan, Roberto Palmieri
Lehigh University, USA
{dsm220,jjn217,ahh319,palmieri}@lehigh.edu

ABSTRACT

We present KVCG, a novel heterogeneous key-value store whose primary objective is to serve client requests targeting frequently accessed (hot) keys at sub-millisecond latency and requests targeting less frequently accessed (cold) keys with high throughput. To accomplish this goal, KVCG deploys an architecture where requests on hot keys are routed to a software cache operated by CPU threads, while the remainder are offloaded to a data repository optimized for execution on modern GPU devices. Cold/hot partitioning is done at runtime through a model trained with the incoming workload. Against a state-of-the-art competitor, we obtain up to 34x improvement in latency.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; • **Computing methodologies** → *Graphics processors.*

KEYWORDS

Key-value Stores, Heterogeneous Computing, Concurrency

ACM Reference Format:
dePaul Miller, Jacob Nelson, Ahmed Hassan, Roberto Palmieri. 2021. KVCG: A Heterogeneous Key-Value Store for Skewed Workloads. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456727.3463779

1 INTRODUCTION

Key-value stores play a fundamental role in many widely used applications that offer (possibly millions of) users the ability to access a shared state in a consistent way [16, 22, 23, 25, 26, 28, 32, 45]. Because of its flexible data model, and a simpler software architecture than traditional database management systems [2, 39], the adoption of key-value stores has rapidly grown over the last decade. Today, this class of data repositories must handle an extremely high volume of operations, which has motivated a large effort to improve performance [25, 31, 35, 45].

Looking deeper into the traditional key-value store operations (e.g., atomic GET, PUT, and DELETE) we find that they require few CPU cycles to execute. Despite their simplicity, the number of operations that need to be processed is large because of the high application demand.

The vital observation is that key-value store workloads suggest a lower affinity to CPUs than to accelerators, such as the Graphic Processing Unit (GPU). CPUs are designed to process fewer, but more complex, instructions than its GPU counterpart, which is evident when comparing the x86_64 ISA to PTX [10, 21]. On the other hand, the GPU architecture has more, simpler, cores offering higher parallelism and can exploit a much higher memory bandwidth when compared to the CPUs as noted in [38].

GPUs are also more affordable hardware than large multi-core architectures [3, 4], and nowadays they are also available in major cloud providers [5–7]. However, the high throughput offered by GPUs come at the cost of a high latency for individual tasks. In fact, well-known downsides such as traversing the external bus (e.g., PCIe) to transfer instructions and data between CPU and GPU, implementing efficient synchronization barriers, and launching kernels to execute on the GPU can stretch latency when offloaded tasks have short execution time, like key-value store operations.

In this work, we present KVCG, a heterogeneous key-value store designed to leverage the relative merits of both CPUs and GPUs. KVCG aims to maximize the effectiveness of CPU resources while cooperating with the GPU. Our work builds on the observation that key-value store workloads are typically skewed, meaning a subset of keys is more popular than others [13, 20, 44]. This divides the key-space into frequently accessed (*hot*) and less frequently accessed (*cold*) keys. Hot keys require low latency and are accessed by several application clients; cold keys can have a lower priority because they are not accessed with the same intensity. Previous works detailing real-world application behavior (e.g., at Facebook [36] and Twitter [44]) demonstrate that workloads have innate

SYSTOR '21, June 14–16, 2021, Haifa, Israel
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8398-1/21/06…$15.00
https://doi.org/10.1145/3456727.3463779

tendencies to access a subset of keys more frequently, and also that such a set of frequently accessed keys changes over time. These observations motivate the following design.

KVCG deploys three main components: the *Hot Cache*, a coherent software cache, accessed by CPU threads, to serve operations on hot keys; the *Canonical Store*, a data repository whose core component is a lock-based GPU hashmap (L-Slab) optimized for high throughput and able to store values of arbitrary size; and the *Router*, which relies on a model trained at runtime to classify incoming requests as hot or cold, and directs them to either the Hot Cache or the Canonical Store.

Operations that are not served by the Hot Cache on the CPU are batched and scheduled for execution on the GPU, where the Canonical Store operates. Effectively, the entire data repository resides on GPU accessible memory and keys are transitioned to/from the CPU based on the output of the model in the Router. In order to circumvent the size limits of GPU memory, KVCG's memory management exploits NVIDIA Unified Memory technology [37] to host both key-value pairs and metadata.

KVCG is implemented using CUDA 11.2 and evaluated using NVIDIA GTX 1080 and the YCSB benchmark [20] at different skews and mix of operation types (read/write). Our competitors are MegaKV [45], a key-value store specifically designed to exploit the GPU for providing high throughput; MICA [31], a high-performance CPU key value store; as well as a CPU-only version and a GPU-only version of KVCG.

Overall, performance results show that KVCG is able to provide a latency for hot requests in the range 19us to 150us on typical read-heavy skewed workloads [44] (i.e., θ=0.5). The latency of cold requests also stays within the range 0.2ms to 2.2ms. Compared to MegaKV, KVCG achieves a 34x mean latency improvement with respect to the hot key's while retaining 1.1x throughput improvement with respect to all keys. Compared to MICA, KVCG achieves at least a 1.4x improvement in throughput.

When the workload changes during runtime, KVCG adapts its partitioning scheme between hot and cold keys by updating the model in the Router.

This paper makes the following contributions:

- We propose a novel design for effectively deploying a key-value store across both the CPU and GPU to accelerate skewed workloads.
- We demonstrate that isolating hot keys to the CPU and processing requests targeting cold keys on the GPU improves throughput while dramatically lowering latency for operations on hot keys.
- We present L-Slab, an extension to the high-performance GPU-based hashmap, Slab [12], that supports arbitrarily sized values on the GPU.

The source code and benchmarks for KVCG and competitors is released at github.com/sss-lehigh.

2 RELATED WORK

Both academia and industry have proposed many key-value stores in the last decade. Examples include Memcached [23], LevelDB [26], RocksDB [16], EvenDB [25], Oak [32], FARM [22], FaSST [28]. Since KVCG focuses on cooperation between the CPU and GPU to serve requests, in the rest of the related work we focus on previous effort in executing atomic operations on heterogeneous devices.

In recent years, synchronization on the GPU has gained substantial attention. Transactional memory [15, 17–19, 24, 43], lock-based synchronization [29, 42, 45] and lock-free approaches [33, 40] have all been explored as viable options for designing concurrent applications for the GPU. Furthermore, evidence that fine-grained synchronization can benefit GPU programs provides a foundation for the continued efforts along this line of research [29, 34]. KVCG follows suit by allowing write operations to perform concurrently with read operations on the CPU and GPU.

MegaKV [45] is a key-value store that handles operations on the GPU and uses a static dispatching policy to provide predictable performance. It is similar to KVCG in its aim to accelerate key-value stores. It utilizes a concurrent cuckoo hash table and a protocol that deterministically schedules GPU kernels to execute batches of requests. Furthermore, only an index is maintained on the GPU, with key-value pairs in host memory. KVCG is able to utilize the oversubscription of Unified Memory to provide indexing that can extend beyond the limits of GPU memory, while MegaKV must evict items when GPU memory fills up. KVCG is also able to utilize the CPU for performing low-latency operations on hot keys.

Because it is constrained to atomic operations for key-value pairs, MegaKV only stores 4B hashes to 4B pointers on the GPU, which forces MegaKV to resolve collisions on the CPU in post-processing. Instead, KVCG reduces overhead by utilizing a lock-based approach and using 8B hashes and 8B pointers on the GPU.

Closer related works to KVCG are Slab [12], a hashmap based on a GPU data structure called a slab list, and HCC [14], a hybrid cache coherent hashmap. Slab is extensively described in Section 5 since KVCG includes an improved version of Slab, named L-Slab, to process operations on the GPU. Instead, here we focus on HCC. HCC uses the CPU to perform write operations in a non-blocking fashion while `GET` operations are scheduled on the GPUs. Similar to KVCG, HCC also uses Unified Memory but differs in that it offloads `PUT` operations to the CPU. Therefore, for HCC, the GPU workload is read-only, whereas KVCG processes both read and write workloads concurrently on the GPU. HCC is specifically optimized to take advantage of IBM's Power9 architecture with NVLINK; KVCG targets off-the-shelf hardware.

3 SYSTEM OVERVIEW

KVCG implements the following widely used atomic linearizable APIs: GET, PUT, and DELETE. In KVCG the return value of a GET is the data associated with the provided key. The PUT and DELETE APIs return the value of the provided key before the update takes effect, if any existed.

KVCG is designed to take advantage of the high throughput of the GPU and the low latency of the CPU. At a high level KVCG includes three main components.

- The *Canonical Store*, which holds all the key-value pairs in the data repository. This store is has values placed in CPU memory when values are over 8B, with an index accessible by the GPU. The Canonical Store also includes a concurrency control to let GPU threads process atomic read and write operations on the key-value pairs. KVCG avoids limiting the total size of the index to the memory resident on the GPU by relying on NVIDIA Unified Memory (UM) [37]. UM enables a unified view of the address space and on demand paging across both the CPU and GPU since CUDA 6 [27]. Through a combination of UM and explicit memory management, KVCG can utilize more memory than is available on the GPU while avoiding excessive paging overheads [30].
- The *Hot Cache*, which serves operations on *hot* (i.e., frequently accessed) keys to ensure low latency for popular requests. The Hot Cache is coherent, meaning it caches the latest value of current hot keys. This cache allows KVCG to effectively exploit the available CPU threads (significantly outnumbered by GPU threads) to perform low-latency operations without the interference of requests targeting *cold* (i.e., less frequently accessed) keys, which would otherwise saturate CPU threads.
- The *Router*, which includes a model to determine whether or not a key should be destined for the Hot Cache or the Canonical Store. Importantly, a wrong prediction by this model does not impact operations' correctness, it only affect their performance. The model is continuously trained with incoming client requests so that KVCG can dynamically change the composition of cold and hot storage to respond to application workload changes over time.

To accommodate the needs of modern key-value stores, KVCG allows clients to issue a batch of requests as opposed to individual operations. This mechanism is a generalization of batched operations, like the multi-Get API supported in Memcached [23], a well-known key-value store. Our expectation is that KVCG is paired with a middleware that aggregates end-user requests into batches that are then submitted to the store. We call these batches *Client Request Batches* (CRB).

In the following sections, we first describe the lifetime of a CRB then follow up with a discussion of the design of each of the components that make up KVCG.

4 EXECUTION WORKFLOW

This section follows the journey of a CRB as its requests are processed by KVCG. Each entry in the CRB consists of the request type, the key-value pair, and a response field. The latter is used to notify the client that the request has completed successfully. A response either informs the client that the operation completed or that the operations should be retried. As we show later, a retry is required in two cases. First, when the GPU cannot accept new requests to avoid overrunning memory. Second, when updates are blocked during a model change.

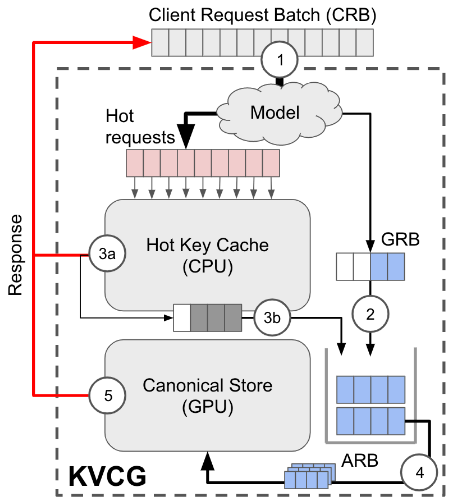


Figure 1: KVCG architecture and request workflow.

Figure 1 illustrates the software architecture of KVCG and embeds the sequence of steps performed by client requests to accomplish their execution.

(1) The first stage of processing is partitioning all requests in a CRB into hot requests or cold requests based on the current knowledge of the Router. Note that this partitioning scheme can change over time as a consequence of an update to the Router's model (more details in Section 7).

(2) After partitioning, the cold requests are batched in a GPU Request Batch (GRB) and enqueued for execution on the Canonical Store through a GRB queue.

(3a) Requests that were not sent to the Canonical Store are concurrently executed on the Hot Cache by a pool of dedicated threads. If a request can be serviced by the Hot Cache, meaning there was a hit, then the client is notified immediately. Otherwise, misses are batched for execution on the GPU where the Canonical Store resides, similar to the cold requests. The term GRB also applies to these batches.

(3b) Requests that missed the Hot Cache are enqueued on the GRB queue for processing on the Canonical Store.

④ As GRBs are enqueued, dedicated threads are responsible for dequeuing them from the head of the queue and creating *Aggregated Request Batches* (ARBs). ARBs are necessary to fully leverage the massive parallelism of the GPU.

⑤ Finally, ARBs are fed to the GPU. Requests are executed in parallel with the appropriate support for mutual exclusion to ensure atomicity of operations. At this point and if needed (see Section 6), the Hot Cache is updated to avoid subsequent misses. The return values of operations is transferred through the value field of the corresponding request in the ARB to clients.

According to the just described execution flow, the CPU and GPU concurrently execute requests. Consistency is guaranteed because effectively the key-value store is partitioned at any given moment between the Hot Cache and the Canonical Store. The only potentially dangerous scenario for correctness is when the model in the Router is updated. We will detail how we handle that in Section 7.

5 THE CANONICAL STORE

The Canonical Store supports `GET`, `PUT`, and `DELETE` operations on a given key. The core component that guarantees the correct and effective execution of these operations is a GPU lock-based hashmap, we name *L-Slab*.

L-Slab relies on readers-writer locks [11, 34] to enforce ordering between conflicting operations, and warp-cooperation [12] for increasing performance of the lookup functionality (Section 5.1). L-Slab uses 8-byte keys and values, meanwhile exploiting Unified Memory to support a total capacity exceeding that of GPU memory alone. When variable sized values are required, pointers to non-adjacent memory can be used (see Section 5.1.1).

As described in Section 4, a GRB queue is used to accumulate requests that are either directly passed to the GPU processing pipeline via the Router, or requests that missed in the Hot Cache. As GRBs are received, they are formed into the larger ARBs for dispatch to the GPU. As long as there are GRBs in the queue, a thread will continue to fill its ARB until there are twice as many requests as the number of threads available on the GPU or no more GRBs are available. To avoid blocking in the latter case, we deploy a dequeue budget that limits the number of times a queue is checked.

ARBs are aggregated batches of requests that contain the target key, the key hash, an associated value, and the request type (i.e., `GET`, `PUT`, `DELETE`). Since the ARB is accessed by the GPU, it is first provisioned in pinned host memory, then copied to the GPU with a call to `cudaMemcpy`. Upon completing execution on the GPU, the results of each operation are then copied back to the ARB in host memory before returning to the client. Explicit memory management allows for more efficient ARB processing because all request data and metadata resides on the GPU when the kernel is launched. We decided against allocating ARBs using Unified Memory because it incurs the additional cost associated with paging memory to the the GPU, which negatively impacts performance when interleaved with request processing.

ARB dispatch is handled by *orchestrator threads*, which handle ARB creation, GPU kernel launch and client response. In our implementation, we associate each orchestrator thread with a CUDA stream [8] to pipeline execution. The above choice is critical to minimize the impact of the overhead of offloading work to the GPU, therefore retaining low latency while still enjoying the high throughput of the GPU.

5.1 Lock-Based Warp-Cooperative Slab

L-Slab is a lock-based warp-cooperative hashmap data structure and builds upon the GPU-based Slab hashmap [12]. As opposed to conventional hashmap designs, Slab is appropriate for GPU because each bucket stores a linked list of *slabs*, and each slab holds 31 key-value pairs and a pointer to the next slab in the bucket. This design matches the GPU execution structure where an entire warp, meaning a set of 32 GPU-threads, can be assigned to work on a slab. In order to atomically modify key-value pairs in the hashmap, Slab uses Compare-And-Swap (CAS) operations. Because of that, it can only accept key-value pairs of size 8 bytes total (e.g., 4-byte for the key and 4-byte for the value), which is the hardware limitation to guarantee correctness of the CUDA CAS operations. Accommodating larger size of key-value pairs in Slab would require a secondary index to be queried using the original 8B key-value pair.

L-Slab inherits Slab's use of warp-cooperation because *i)* it provides high performance due to favouring coalesced accesses to the GPU memory [1]; and *ii)* GPU threads within a warp can use fast hardware instructions (e.g., `__shfl_sync`, `__ballot_sync`) to coordinate their activities and guarantee correct concurrent accesses over shared data. Unlike the lock-free design of Slab, L-Slab deploys a reader-writer spin-lock to protect each bucket. This modification makes KVCG more practical because it can efficiently work with any size of key-value pairs, as we will show later in Section 5.1.1.

L-Slab serves requests of all types with the following general pattern. At each kernel launch, the ARB is first partitioned among warps. We oversubscribe the streaming multiprocessors (SMs) in the GPU by launching as many blocks as twice the number of SMs, which offsets the latency involved with launching a kernel and transferring the ARB to the GPU. Each block consists of 16 warps and we assign each thread a single operation to perform.

During execution, the warp works in unison to complete all requests that its threads are assigned, one request at a time. For each request, a single thread within the warp orchestrates

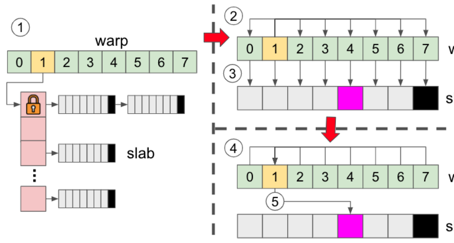


Figure 2: Execution of a GET operation on L-Slab.

the operation. Per-bucket locks allow threads in a warp to safely cooperate despite conflicting concurrent operations.

Figure 2 demonstrates this execution for a GET operation, with a simplified warp consisting of 8 threads. (1) Initially, the *leader thread* (in this case Thread 1) locks the bucket corresponding to its request. (2) Then, it broadcasts the target key of its request to the entire warp. (3) Together, all threads in the warp check their corresponding entry in the first slab for the broadcast key. (4) Next, all threads in the warp report back to the leader whether their entry matched the key. (5) If the key is found, then the leader performs the GET and releases the lock. If the key is not found, and there is a pointer to another slab, then the leader broadcasts the address of the next slab and the search continues. Finally, this entire process is repeated for every request designated to the warp, with each warp thread acting as the leader for its assigned requests. We will discuss PUT and DELETE operations in more detail later, but we first explain the layout of slabs.

Each slab in our design is composed of two distinct arrays that are adjacent in memory. Considering the case of keys and values of 8B each, a slab is constructed as an array of 32 8-byte entries, 31 are reserved for keys and the last for the pointer to the next slab, followed by an array of 31 values. This layout is motivated by the following intuition. For any request, a value is accessed (i.e., read or written) only after the slab's keys have been analyzed and matched with the key that was broadcast by the leader of the current operation. Packing keys together allows accesses by the warp to be coalesced, which is a well-known optimization to improve performance on the GPU [1]. Once found, the location of the value corresponding to the key can also be easily determined, since each entry in the slab is a fixed size. We describe how we leverage these entries to store arbitrary sized keys and values later in this subsection.

We now detail how each operation on L-Slab is handled. As described above, the warp cooperatively indexes to the bucket corresponding to the hash. The first thread in the warp acquires the lock on the bucket. Then, the warp proceeds to search for the key or an empty key if the key cannot be found for a PUT. One of the following then occurs:

- GET. The leader reads the value associated with the key and writes it to the ARB.
- PUT. The leader writes the key and value to the slab, and writes back the previous value, if any, to the ARB.
- DELETE. The leader sets the matching key, if found, to the empty key and writes back the previous value to the ARB.

After the operation finishes, the lock is released.

5.1.1 Handling large size key-value pairs. As mentioned earlier, L-Slab can support keys and values of arbitrary sizes. In the above description we assumed that keys and values have both a fixed size of 8 bytes (if less, they are padded to 8 bytes). If larger sizes are desired, the keys and values in the map are pointers to immutable memory blocks that can be unrestrictedly allocated either in UM, GPU, or main memory.

The only additional overhead of using values larger than 8 bytes, which therefore cannot be embedded into the internal memory layout of L-Slab, is the memory copy performed before delivering the return value of an operation back to the application on the CPU.

5.2 Memory Management

As said earlier, the Canonical Store leverages Unified Memory (UM) for over-subscription, which allows more memory than the one physically available on the GPU to be allocated. Utilizing UM introduces advantages and flexibility through a hardware-implemented on-demand page fault mechanism that allows the CPU and GPU to cooperate over a coherent shared memory. The downside of this increased flexibility is the possible performance penalty due to repeated page faults that let memory pages continuously migrate between devices [30]. Another issue of UM is that memory cannot be dynamically allocated by GPU threads. This capability is critical for KVCG since slabs in the L-Slab hashmap should be allocated/deallocated at runtime to accommodate update operations on the GPU without involvement of the CPU.

By taking into account the above limitations, KVCG leverages UM as follows. In order to allow for dynamic memory allocation on the GPU, regions of memory are pre-allocated in UM. At runtime, GPU threads can request chunks of the pre-allocated memory. If the requested amount exceeds the total memory previously accounted for, then the CPU reclaims the kernel and expands the pre-allocated region.

In addition to that, UM is used by KVCG only for allocating the actual L-Slab hashmap structure (i.e., the index) and the key-value pairs. ARBs are not allocated in UM because their size is often smaller than the granularity at which UM transfers memory between CPU and GPU. Since the performance of operations involving the ARB is critical for achieving low-latency GPU computation, we prefer to explicitly move ARBs from CPU to GPU and vice-versa.

6 THE HOT CACHE

The Hot Cache wraps a high-performance concurrent data structure to hold some key-value pairs of KVCG. The purpose is to provide low-latency operations for a subset of keys that are classified as *hot* (i.e., frequently accessed) by the Router. It should be noted that the integration of this component into KVCG is made in a way the Hot Cache design and implementation can be independently replaced and optimized with respect to KVCG.

The Hot Cache is implemented as a readers-writer lock-based hashmap, where each bucket consists of a linked list of nodes containing 8 elements each. The hashmap is pre-allocated based on an initial memory budget, but can expand beyond that in the case of conflicts. However, given that each node consists of 8 elements, we expect this to be rare.

After locking the bucket corresponding to the target key, each operation performs its necessary work then releases the lock. Operations on keys that are present in the cache simply update, delete or return the existing value. When there is no entry corresponding to the key, then some additional steps are required. In more detail:

- `PUT`. If the entry already exists, then it simply updates the value. A `PUT` operation on a non-resident key reserves an unused entry – allocating a new node as necessary – and populates it with the value contained in the request.
- `DELETE`. For existing keys, the entry is simply marked as deleted. For non-cached keys, a `DELETE` will reserve a new entry – again allocating nodes as needed – and mark it as deleted to serve as a tombstone for future operations. A subsequent `GET` can therefore be serviced immediately without querying the Canonical Store. Similarly, subsequent `PUT` operations need not to allocate memory for the new key, but can operate directly on the existing entry.
- `GET`. If the bucket contains an entry corresponding to a key, and the value is not logically deleted, its value is returned. When a miss occurs (i.e., there is no entry for the key) the request is added to a GRB to be executed on the GPU. A special flag indicates that this request originated from a miss and instructs the GPU thread responding to populate the cache after the operation completes.

Recall that the key space is partitioned between the Hot Cache and the Canonical Store, hence requests remain consistent through the underlying concurrency control of each storage component. However, when the model changes, all `DELETE` and `PUT` operations made on the Hot Cache must be propagated to the Canonical Store at the moment the model changes. Hence, in addition to generating and updating entries in the Hot Cache, update operations (i.e., `PUT` and `DELETE`) should be logged. This log is implemented as a secondary hashmap with a one-to-one mapping between it and the Hot Cache. It is stored in contiguous memory to ease scan operations and capture the most recent updates made to keys on the Hot Cache that must be propagated to the Canonical Store. The next subsection describes this process.

6.1 Replaying Logged Operations

The goal of the Hot Cache is to make frequently accessed keys available to application requests so that they can be executed with low latency. Our system is designed to do so while adapting to changing workloads by dynamically updating the Router's model in response to new patterns. During this process we must ensure that operations maintain correctness. With keys whose hot-cold assignment does not change, there is no risk for inconsistency since each component guarantees linearizability. For keys that were originally cold but become hot, the cache miss policy will provide the most up-to-date version from the Canonical Store. However, if a key was previously hot, but now cold, any future request routed to the GPU must observe the most recent value.

The moment the model is updated marks the start of a new epoch for requests. Requests that had previously been hot may now be cold and therefore destined for the GPU. As such, it is crucial to maintain the consistency as this change is made. Before new requests can be served, any update to the Hot Cache must be reflected in the Canonical Store. As mentioned, we maintain a log of updates performed during the previous epoch. These logged requests are batched and enqueued on the GRB queue. Once the log is replayed on the Canonical Store, new update requests can be served since any operations going to the Canonical Store will be ordered after the replayed log. Note that it is sufficient to only replay the last epoch as requests of prior epochs will already be present on the Canonical Store. In other words there is only a single epoch of requests outstanding on the Canonical Store.

6.2 Cleaning the Hot Cache

Another important part of ensuring consistency is the removal of cache entries for keys that are no longer hot. Under our assumptions, the size of the entire data repository can fit in main memory. A consequence of this is that evictions from the Hot Cache are not performed to make room on the cache, as traditionally done [31, 45], but rather to remove keys that are no longer hot. This work is performed by a helper thread every time the model in the Router changes. When that happens, the thread iterates over the Hot Cache removing all entries that are cold according to the new updated model.

To guarantee correctness (i.e., linearizability) of operations, the helper thread must complete its eviction before any subsequent model changes. The above requirement combats the case in which some model, $m1$, specifies a given key as hot; its successor, $m2$, classifies the key as cold; then a third model, $m3$, again labels the key as hot. In this scenario, if the key is not evicted from the cache when $m2$ is active then an operation directed to the Hot Cache by $m3$ may observe a value written when $m1$ routed requests. All keys that are no longer hot are removed from the Hot Cache upon each model transition to ensure no stale values are read.

It is important to note that KVCG exploits the massive parallelism of the Canonical Store when replaying the logged operations. Accordingly, we envision a quick transitioning process (e.g., < 500ms in our experiments).

7 THE ROUTER

KVCG relies on the Router to decide which keys are *hot* and which are *cold* at runtime, and route them to the appropriate store for execution. A machine learning based classifier, denoted in this paper as the model, is integrated into the Router and its purpose is to accept a key and the hash of that key as input and simply return a classification of that key as hot or cold. KVCG then adapts to reflect the classification made by the Router. Thanks to its design, other models can also be used in the Router to tailor it to the application's needs.

While running KVCG, a shadow model is trained by sampling incoming requests. Periodically, the shadow model replaces the current model with the goal of continuously adapting the partitioning of the store between Hot Cache and Canonical Store in order to meet workload changes.

Replacing the current instance of the model is done without impacting the correctness of ongoing operations. In order to do that, a lock is held for the entire duration of the process to prevent triggering another model change. Once the lock is acquired, all incoming update operations must retry on the Hot Cache while ongoing operations and new `GET` operations complete on the Hot Cache. After that, `PUT` and `DELETE` operations on the Hot Cache are replayed on the Canonical Store utilizing the process described in Section 6.1.

After that, the model can be updated and the Hot Cache resume serving all operations. Evictions of no-longer-hot keys should now occur on the Hot Cache; this is done using the procedure described in Section 6.2. After the eviction is complete, the lock is released.

During the change of the model, the Canonical Store and Hot Cache can execute previously enqueued operations. This is because any request previously enqueued for execution on the Canonical Store will be able to access the most up-to-date values by relying on the step (described previously in Section 6.2) that replays the latest Hot Cache operations on the Canonical Store. After the Hot Cache resumes execution, all operations previously batched and still to be executed on the Canonical Store are effectively concurrent with operations to be executed on the Hot Cache. Correctness is still guaranteed in this case, as shown below.

Let us consider a key k previously classified as cold and currently hot according to the new model. Let us also consider an operation o_i on k that was batched for execution on the Canonical Store before the changing of the model started. If a new conflicting operation o_j is issued after the model is changed, then it will be routed to the Hot Cache. At this moment, o_i and o_j are conflicting and operating concurrently on the two stores of KVCG. In this case, Linearizability is maintained since these operations are concurrent thus no order must be enforced between them. Anyway, due to the installation of the new model, all new requests will then be routed to the Hot Cache and therefore any possible update of the value of k in the Hot Cache will be the one finally read by future operations. Eventually, future evictions will reflect the most recent value of k in the Canonical Store.

In our implementation we use a histogram trained on the incoming client requests at runtime. We set a threshold where any bucket of keys accessed at a proportion above the threshold is classified as hot (e.g. a threshold of 0.1 means that any bucket where the proportion of total accesses is greater than 0.1 is classified as hot). We find experimentally that a well-trained model alleviates contention on the Canonical Store and on the Hot Cache, especially in the presence of skewed workloads (see Section 8.4 in the evaluation study).

8 EVALUATION

We evaluate the design of KVCG by comparing performance with our primary GPU-based competitor MegaKV [45], as well as a high-performance CPU-only concurrent hashmap, MICA [31]. Additionally, we include two special configurations of KVCG that correspond to a GPU-only version (i.e., just the Canonical Store), we denote as *KVG*, and a CPU-only version (i.e., just the Hot Cache), we name *KVC*.

The testbed consists of two Intel Xeon Platinum 8160 CPUs, totaling 48 available cores, with hyperthreading disabled, and a GeForce GTX 1080 GPU with 2560 CUDA cores across 20 SMs. For each competitor we tune the number of CPU threads and GPU streams to maximize their respective performance. All systems, but MICA, use a batch size of 512. MICA leverages all available CPU threads and extends batch size to 4096 requests to guarantee high utilization; KVCG utilizes only a single NUMA zone (24 cores) for the Hot Cache to reduce hardware contention. Because of its pipelined architecture, we empirically verified that MegaKV performs best with 10 CUDA streams. In addition to the 24 cores dedicated to the Hot Cache, KVCG also allocates 10 threads to handle GPU streams on the Canonical Store.

Unless otherwise stated, the key-range of each competitor is fixed to 1 billion keys to reflect large stores, the hash table is prefilled with 10 million key-value pairs. We report both latency and throughput results averaged across 512 million requests executed in a closed loop. Update operations are performed such that the final load factor is close to the starting load factor. All experiments measure performance under a workload following a Zipfian distribution provided by the YCSB benchmark [20]. To emulate application workloads with varying skew, we report results at different values of theta (θ). Finally, all competitors use the same hash function: $H(x) = x\%s$, where s is the size of the data repository.

KVCG is implemented from the ground up in C++ and the GPU kernels are written using CUDA 11.2. We also reimplement our primary competitor (i.e., MegaKV [45], available at [9]) in the same environment. We replace the underlying GPU hash table (i.e., `libgpuhash`) with the same Slab hash table that is used for KVCG's Canonical Store. Table 1 demonstrates that even for a single-warp block, this replacement improves `GET` and `PUT` performance by about 2x, albeit `DELETE` operations are slower because MegaKV's `libgpuhash` leverages a lock-free logical deletion. Under the read-intensive workloads evaluated, we believe a Slab-based MegaKV to be a more suitable competitor.

Operation Throughput	Slab	`libgpuhash`
PUT (MOPS)	61.0	27.3
DELETE (MOPS)	120.4	233.5
GET (MOPS)	269.7	160.0

Table 1: Slab versus MegaKV's `libgpuhash`

In all of our experiments, we implement KVCG's model to be an approximation of the workload to capture the incoming requests without matching them perfectly. To do so, we use a histogram model that splits the key space into 10,000 bins. During training, a bin accessed at a proportion above a predefined threshold is classified as hot. At inference, if a request's target key falls within a hot bin, the request is routed to the Hot Cache. Note that when calculating performance metrics, a miss in the Hot Cache counts towards the Canonical Store since it is the first time accessing this missed key after the model changed and the request is loading it from the GPU. Misses are expected to occur infrequently if the model accurately reflects the application workload.

The rest of the evaluation section is organized as follows. First, we explore KVCG in isolation with 8-byte keys and values in Section 8.1. Then, in Sections 8.2 and 8.3, we explore how KVCG compares to competitors. Later, in Section 8.4, we explore how the model impacts the performance of KVCG. In the legends of the included plots, we use `HC` to refer to the Hot Cache and `CS` to refer to the Canonical Store.

8.1 Analysis of KVCG performance

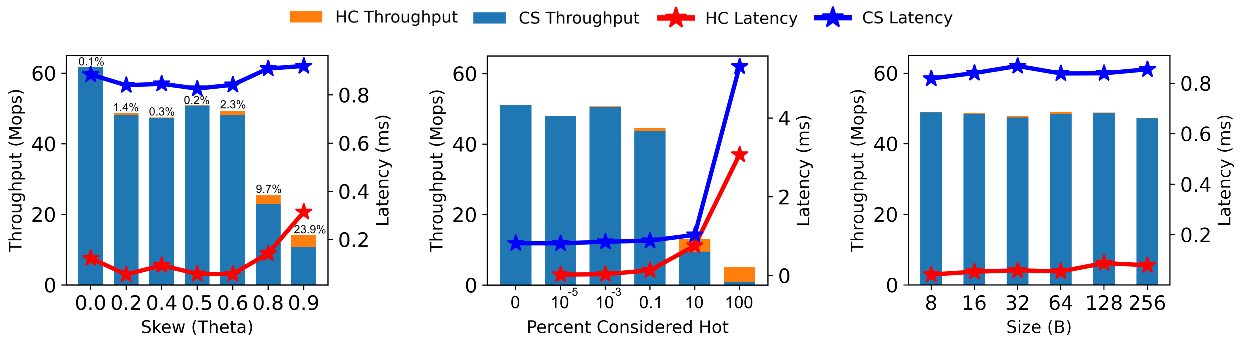
We begin our discussion with an experiment to demonstrate the performance of KVCG for 8-byte values across various workloads. Figure 3a shows how our system responds to an increase in skew. For this experiment, we use the aforementioned histogram model from Section 7 with a threshold of $1.3 * 10^{-5}$ to approximate the incoming workload. For each skew, we retrain the model on 5 million requests.

In any key-value storage system, we expect that the overall throughput decreases as the skew becomes greater because of higher contention on the internal data store. KVCG is able to adapt to this change by offloading hot requests to the Hot Cache, which can be observed in the steady increase in the Hot Cache's overall proportion of the achieved throughput. Although latency of requests to hot keys increases with the skew, it remains below 0.15ms until a skew of 0.9; after that, it only increases up to 0.32ms. Because of the massive parallelism offered by the Canonical Store, and the general lack of contention, we observe the highest overall throughput at the lowest skew. This, however, is at the cost of higher latency due to more processing on the GPU. While routing requests, each classification takes 315 ns on average.

Next, in Figure 3b we measure throughput and mean latency for the system when fixing the skew to θ=0.5 and varying the percentage of keys that are classified as hot by the Router. We choose 0.5 ($\alpha = 2$) since this is the highest theta reported by Twitter in their read heavy workloads [44]. The model for this experiment uses prior knowledge, corresponding to the Zipfian distribution used to generate the workload, to classify a fixed proportion of the most popular keys as hot. Because of the skew, a small fraction of hot keys may result in a larger proportion of overall requests. In other words, when the Router only considers 0.1% of the key space as hot, the Hot Cache serves 1.83% of the overall requests. An interesting trend emerges when we consider a scenario where all operations are routed to the GPU only. In this case, throughput remains relatively high, but at the cost of an average latency of 0.82ms.

It is of note in Figure 3b that KVCG performs best when few operations are fulfilled by the Hot Cache. When many requests are routed to the Hot Cache, the utilization of the GPU is low and the Canonical Store is unable to provide high throughput. With few requests routed to the Hot Cache however, few requests get low latency. When running KVCG, we look to find the trade off between throughput and latency. By routing the hottest 0.1% of requests to the Hot Cache, we provide much lower latency (i.e., 131.7us) for those requests while retaining similar latency and throughput on the GPU.

Up to 0.1% of the keyspace being considered hot, the throughput is greater than 40 Mops. Between 0.1% and 10%, there is a drop in performance because the Canonical Store is not fed enough requests. To the extreme, when all requests are served by the Hot Cache, the throughput is low and the latency is high because contention prevents the Hot Cache from performing well. We do notice some of the requests



(a) Varying θ. **(b) Varying Partitioning. θ=0.5.** **(c) Varying Value Size. θ=0.5.**

Figure 3: Throughput and mean latency of KVCG. Reads are 95%; Writes are 5%. Skew is noted as θ.

where 100% of the requests are routed to the Hot Cache are executed by the GPU. This is because compulsory-misses, misses that initially occur because the key is not present in the Hot Cache, occur and are counted as GPU execution.

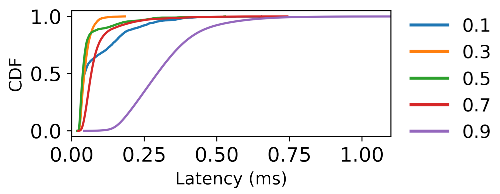


Figure 4: CDF of Hot Cache. 95% read ratio. 8B values.

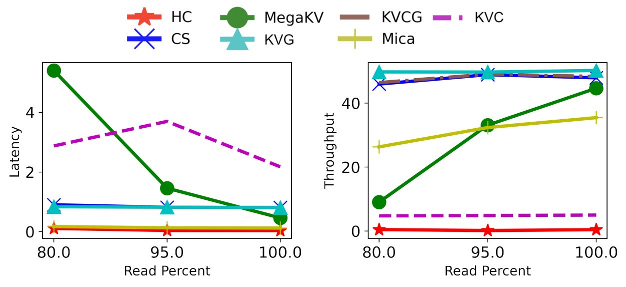
These trends help to highlight both the importance of the model and the impact of contention on the Canonical Store. If the model classifies too many operations as hot, then it can have a detrimental impact on performance. While the Hot Cache provides superior latency over the Canonical Store, the massive parallelism provided by the GPU is important to support higher throughput, especially when operations are non-conflicting. In doing so, KVCG can achieve good performance relative to the respective needs of operations.

When considering the impact of value size on performance in Figure 3c, we find that the size of the value up to 256B minimally impacts the performance of KVCG. The same trend emerged for all other competitors, which also rely on indirection to handle variable sized values.

We also explore the cumulative distribution function, CDF, of the latency for the Hot Cache in Figure 4 and find that while serving a lower proportion of the requests (a lower theta), the latency is lower at most points. As the theta increases, a higher proportion of requests are served on the Hot Cache and there is contention on the keys. This creates greater tail latency.

8.2 Varying Read/Write Ratio

We evaluate KVCG against MegaKV, KVC, KVG, and MICA by varying the read/write ratio at θ=0.5 with 8B keys and values. In addition to the competitors, KVCG latency is reported for each component (i.e., the Hot Cache and Canonical Store). When considering throughput, we include both their individual contributions along with the total.



(a) Mean Latency **(b) Throughput**

Figure 5: KVCG and competitors varying % of read /write operations. Skew θ=0.5. 8B value sizes.

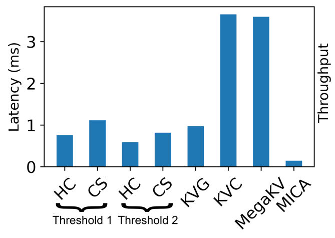
In Figure 5a we note that KVC has high mean latency when compared to other components. This is because the Hot Cache was designed and optimized to serve few requests. When KVC has to serve the entire data store, there is high contention which causes high latency. MegaKV has high latency because of its secondary index. In fact, with more update operations, MegaKV needs to modify the secondary index more, which causes a significant latency penalty. On a read-only workload, MegaKV has lower latency than the Canonical Store in KVCG and KVG because it has the lock-free structure of Slab while L-Slab requires acquiring locks.

At θ=0.5, KVG and KVCG's Canonical store have similar performance. KVCG on the other hand is able to keep up with the throughput of KVG while providing low latency operations on hot keys.

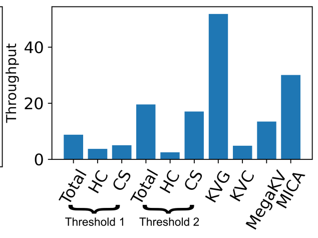
KVCG's Hot Cache shows slightly lower latency than MICA (0.11ms versus 0.17ms at 80%). MICA's throughput is however lower than the GPU's. KVCG is the best choice when a throughput greater than 35.4 MOPS is needed, but an average latency below 1ms is desirable.

8.3 YCSB Standard Skew

In Figure 6 we evaluate KVCG against competitors with a traditional YCSB workload B, which consists of 95% reads and 5% updates using θ=0.99. We evaluate two thresholds: T1 is the $1.3 * 10^{-5}$ threshold we chose for theta 0.5; T2 is 0.005. Similar to the plots in Figure 5, we find that KVC performs poorly. The latency is above 1ms while the throughput is low. MegaKV has similar issues to Figure 5, where the utilization of the secondary index impacts the performance. MICA similarly has a low throughput and low latency.



(a) Mean Latency



(b) Throughput

Figure 6: Performance of KVCG and competitors. Skew θ=0.99. 8B values. 95% reads/5% writes.

The most interesting change from theta 0.5 to 0.99 is that KVG performs better than KVCG at both T1 and T2. Its latency is only slightly greater than the latency of CS at T2, while the throughput is more than 2x. Although 0.99 is heavily skewed, L-Slab is able to perform well enough to achieve a large throughput when it receives enough operations to handle. At both T1 and T2, the Canonical store is unable to receive enough requests per second workload to provide a significant throughput. This is due to few keys being heavily accessed at a θ=0.99. Changing from T1 to T2 yields an increase in throughput, but compared to going from a T2 (a threshold of 0.005) to KVG (a threshold of >1), it is unable to achieve as much of a gain in performance. KVCG however is able to run with a lower latency on Hot Cache when compared to KVG. When using KVCG at this high a skew, a high threshold should be chosen if using a histogram model.

8.4 Changing the Model

One of the fundamental components of our design is the Router, which decides whether requests are to be served by the Hot Cache or the Canonical Store. In this experiment we aim to understand the impact that our dynamic model switching protocol has on performance. Additionally, we wish to demonstrate the importance of a well-trained model.

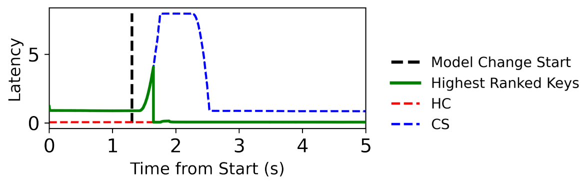


Figure 7: Moving average of system latency over time when changing the model.

To accomplish the above, we execute requests while training the model in the Router and then update it during execution. Initially, KVCG handles requests using a poorly trained model. At time 0s, a shadow model is trained for 5 million requests using a histogram with 10,000 bins and a threshold of $1.3 * 10^{-5}$ as described before. Once trained, the model is installed at time 1.3s, which completes 340ms later.

Figure 7 shows latency as the model changes. At first, the highest ranked keys in the distribution are served in the Canonical Store because the model misclassifies requests. When the model switches, the latency of requests in the Canonical Store increases as the Hot Cache's log is replayed and compulsory-misses are enqueued for processing on the GPU. At 2.54s, these outstanding requests are completed, and the Canonical Store returns to less than 0.9ms latency. Note that once the highest-ranked keys are populated in the Hot Cache those requests are served with a latency of 0.1ms. We envision these transitions in workload and therefore the model to be infrequent similar to how Facebook's workloads exhibit high temporal locality in a short period (i.e. one hour) that decays exponentially over time [41]. We can expect the majority of the time to be spent with less than 1ms Hot Cache and Canonical Store latency.

9 CONCLUSION

We have presented KVCG, a cooperative heterogeneous key-value store designed to accelerate skewed workloads by offloading requests to less frequently accessed keys to the GPU. Our design, which includes classifying requests at runtime, effectively juggles requirements of modern applications by providing low-latency operations on hot keys, meanwhile supporting high overall throughput by leveraging the GPU for requests on cold keys.

ACKNOWLEDGMENTS

Authors thank our shepherd and all anonymous reviewers for their important comments. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

REFERENCES

[1] 2018. How to Access Global Memory Efficiently in CUDA C/C Kernels. https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

[2] 2021. https://www.mysql.com/

[3] 2021. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/

[4] 2021. https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x

[5] 2021. https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html

[6] 2021. https://cloud.google.com/gpu

[7] 2021. Azure VM sizes - GPU - Azure Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu

[8] 2021. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[9] 2021. MegaKV Github. https://github.com/pzrq/megakv

[10] 2021. Parallel Thread Execution ISA Version 7.1. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[11] Jade Alglave, Mark Batty, Alastair Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. *ACM SIGPLAN Notices* 50 (05 2015), 577–591. https://doi.org/10.1145/2775054.2694391

[12] S. Ashkiani, M. Farach-Colton, and J. D. Owens. 2018. A Dynamic Hash Table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 419–429. https://doi.org/10.1109/IPDPS.2018.00052

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. *SIGMETRICS Perform. Eval. Rev.* 40, 1 (June 2012), 53–64. https://doi.org/10.1145/2318857.2254766

[14] Rajesh Bordawekar and Pidad Gasfar D'souza. 2018. Evaluation of Hybrid Cache-coherent Concurrent Hash Table on Power9 System With Nvlink 2. http://on-demand.gputechconf.com/gtc/2018/video/S8172/

[15] A. Brownsword, W. W. Fung, I. Singh, and T. M. Aamodt. 2012. Kilo TM: Hardware Transactional Memory for GPU Architectures. *IEEE Micro* 32 (03 2012), 7–16. https://doi.org/10.1109/MM.2012.16

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies*. 209.

[17] Daniel Castro, Paolo Romano, Aleksandar Illic, and Amin M. Khan. 2019. HeTM: Transactional Memory for Heterogeneous Systems. arXiv:cs.DC/1905.00661

[18] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. 2010. Towards a Software Transactional Memory for Graphics Processors.. In *EGPGV*. 121–129.

[19] S. Chen, L. Peng, and S. Irving. 2017. Accelerating GPU hardware transactional memory with snapshot isolation. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 282–294. https://doi.org/10.1145/3079856.3080204

[20] 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, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[21] Intel corp. 2021. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference. (2021).

[22] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 401–414.

[23] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[24] Wilson W.L. Fung and Tor M. Aamodt. 2013. Energy efficient GPU transactional memory via space-time optimizations. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 408–420.

[25] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. 2020. EvenDB: optimizing key-value storage for spatial locality. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 27:1–27:16. https://doi.org/10.1145/3342195.3387523

[26] Google. 2019. LevelDB. https://github.com/google/leveldb.

[27] Mark Harris. 2013. Unified Memory in CUDA 6.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided ({RDMA}) Datagram RPCs. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 185–201.

[29] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. 2015. Fine-grained synchronizations and dataflow programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 109–118.

[30] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. 2015. An Evaluation of Unified Memory Technology on NVIDIA GPUs. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2015), 1092–1098.

[31] 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)*. 429–444.

[32] Hagar Meir, Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Idit Keidar, Eran Meir, Gali Sheffi, and Yoav Zuriel. 2020. Oak: a scalable off-heap allocated key-value map. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 17–31. https://doi.org/10.1145/3332466.3374526

[33] P. Misra and M. Chaudhuri. 2012. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 53–60. https://doi.org/10.1109/ICPADS.2012.18

[34] Jacob Nelson, dePaul Miller, and Roberto Palmieri. 2020. Don't forget about synchronization! Guidelines for using locks on graphics processing units. *Concurrency and Computation: Practice and Experience* n/a, n/a (2020). https://doi.org/10.1002/cpe.5757 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5757

[35] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 385–398.

[36] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398. https://www.usenix.org/

conference/nsdi13/technical-sessions/presentation/nishtala

[37] Nikolay Sakharnykh. 2018. Everything You Need to Know About Unified Memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[38] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. 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, 1617–1632. https://doi.org/10.1145/3318464.3380595

[39] Michael Stonebraker and Lawrence A Rowe. 1986. The design of POSTGRES. *ACM Sigmod Record* 15, 2 (1986), 340–355.

[40] S. Xiao and W. Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470477

[41] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2013. Characterizing facebook's memcached workload. *IEEE Internet Computing* 18, 2 (2013), 41–49.

[42] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. 2016. Lock-based Synchronization for GPU Architectures. In *Proceedings of the ACM International Conference on Computing Frontiers* (Como, Italy) *(CF '16)*. ACM, New York, NY, USA, 205–213. https://doi.org/10.1145/2903150.2903155

[43] Y. Xu, R. Wang, N. Goswami, T. Li, and D. Qian. 2014. Software Transactional Memory for GPU Architectures. *IEEE Computer Architecture Letters* 13, 1 (Jan 2014), 49–52. https://doi.org/10.1109/L-CA.2013.4

[44] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[45] 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. *PVLDB* 8, 11 (2015), 1226–1237. https://doi.org/10.14778/2809974.2809984