Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores

Kai Zhang[1] Kaibo Wang[2] Yuan Yuan[2] Lei Guo[2] Rubao Lee[2] Xiaodong Zhang[2]

[1]University of Science and Technology of China [2]The Ohio State University

ABSTRACT

In-memory key-value stores play a critical role in data processing to provide high throughput and low latency data accesses. In-memory key-value stores have several unique properties that include (1) data intensive operations demanding high memory bandwidth for fast data accesses, (2) high data parallelism and simple computing operations demanding many slim parallel computing units, and (3) a large working set. As data volume continues to increase, our experiments show that conventional and general-purpose multicore systems are increasingly mismatched to the special properties of key-value stores because they do not provide massive data parallelism and high memory bandwidth; the powerful but the limited number of computing cores do not satisfy the demand of the unique data processing task; and the cache hierarchy may not well benefit to the large working set.

In this paper, we make a strong case for GPUs to serve as special-purpose devices to greatly accelerate the operations of in-memory key-value stores. Specifically, we present the design and implementation of Mega-KV, a GPU-based in-memory key-value store system that achieves high performance and high throughput. Effectively utilizing the high memory bandwidth and latency hiding capability of GPUs, Mega-KV provides fast data accesses and significantly boosts overall performance. Running on a commodity PC installed with two CPUs and two GPUs, Mega-KV can process up to 160+ million key-value operations per second, which is 1.4-2.8 times as fast as the state-of-the-art key-value store system on a conventional CPU-based platform.

1. INTRODUCTION

The decreasing prices and the increasing memory densities of DRAM have made it cost effective to build commodity servers with terabytes of DRAM [30]. This also makes it possible to economically build high performance SQL and NoSQL database systems to keep all (or nearly all) their data in main memory or at least to cache the applications' working sets [9, 25, 26]. In-memory key-value store (IMKV) is a typical NoSQL data store that keeps data in memory for fast accesses to achieve high performance and high throughput. Representative systems include widely deployed open source systems such as Memcached [2], Redis [3], RAMCloud [26] and recently developed high-performance prototypes, such as Masstree [22] and MICA [21]. As a critical component in many Internet service systems, such as Facebook [25], YouTube, and Twitter, the IMKV system is critical to provide quality services to end users with high throughput. With the ever-increasing user populations and online activities of Internet applications, the scale of data in these systems is experiencing explosive growth. Therefore, a high performance IMKV system is highly demanded.

An IMKV is a highly data-intensive system. Upon receiving a query from the network interface, it needs to locate and retrieve the object from memory through an index data structure, which generally involves several memory accesses. Since a CPU only supports a small number of outstanding memory accesses, an increasingly long delay is spent on waiting for data to be fetched from memory. Consequently, the high parallelism of query processing is hard to be explored to hide the memory access latency. Furthermore, an IMKV system has a large working set [6]. As a result, the CPU cache would not help much to reduce the memory access latency due to its small capacity. Each memory access generally takes 50-100 nanoseconds; however, the average time budget for processing one query is only 10 nanoseconds for a 100 MOPS (Million Operations Per Second) system. Consequently, the relatively slow memory access speed significantly limits the throughput of IMKV systems.

In summary, IMKVs in data processing systems have three unique properties: (1) data intensive operations demanding high memory bandwidth for fast data accesses, (2) high data parallelism and simple computing demanding many slim parallel computing units, and (3) a large working set. Unfortunately, we will later show that conventional general-purpose multicore systems are poorly matched to the unique properties of key-value stores because they do not provide massive data parallelism and high memory bandwidth; the powerful but the limited number of computing cores mismatch the demand of the special data processing [12]; and the CPU cache hierarchy does not benefit the large working set. Key-value stores demand simple but many computing units for massive data parallel operations supported by high memory bandwidth. These unique properties of IMKVs exactly match the unique capability of graphics processing units (GPUs).

Proceedings of the VLDB Endowment, Vol. 8, No. 11

In this paper, we make a strong case for GPUs to serve as special-purpose devices in IMKVs to offload and dramatically accelerate the index operations. Specifically, we present the design and implementation of Mega-KV, a GPU-based in-memory key-value store system, to achieve high throughput. Effectively utilizing the high memory bandwidth and latency hiding capability of GPUs, Mega-KV provides fast data accesses and significantly boosts overall performance. Our technical contributions are fourfold:

1. We have identified that the index operations are one of the major overheads in IMKV processing, but are poorly matched to conventional multicore architectures. The best choice to break this bottleneck is to shift the task to a special architecture serving high data parallelism on high memory bandwidth.

2. We have designed an efficient IMKV called Mega-KV which offloads the index data structure and the corresponding operations to GPUs. With a GPU-optimized hash table and a set of algorithms, Mega-KV best utilizes the unique GPU hardware capability to achieve unprecedented performance.

3. We have designed a periodical scheduling mechanism to achieve predictable latency with GPU processing. Different scheduling policies are applied on different index operations to minimize the response latency and maximize the throughput.

4. We have intensively evaluated a complete in-memory key-value store system that uses GPUs as its accelerators. Mega-KV achieves up to 160+ MOPS throughput with two off-the-shelf CPUs and GPUs, which is 1.4-2.8 times as fast as the state-of-the-art key-value store systems on a conventional CPU-based platform.

The roadmap of this paper is as follows. Section 2 introduces the background and motivation of this research. Section 3 outlines the overall structure of Mega-KV. Sections 4 and 5 describe the GPU-optimized cuckoo hash table and the scheduling policy, respectively. Section 6 describes the framework of Mega-KV and lists the major techniques used in the implementation. Section 7 shows performance evaluations, Section 8 introduces related work, and Section 9 concludes the paper.

2. BACKGROUND AND MOTIVATION

2.1 An Analysis of Key-Value Store Processing

2.1.1 *Workflow of a Key-value Store System*

A typical in-memory key-value store system generally provides three basic commands that serve as the interface to clients: 1) GET(key): retrieve the value associated with the key. 2) SET/ADD/REPLACE(key, value): store the key-value item. 3) DELETE(key): delete the key-value item. Figure 1 shows the workflow of GET, SET, and DELETE operations. Queries are first processed in the TCP/IP stack, and then parsed to extract the semantic information. If a GET query is received, the key is searched for in the index data structure to locate its value, which will be sent to the requesting client. For a SET query, a key-value item is allocated, or evicted if the system does not have enough memory space, to store the new one. For a DELETE query, the

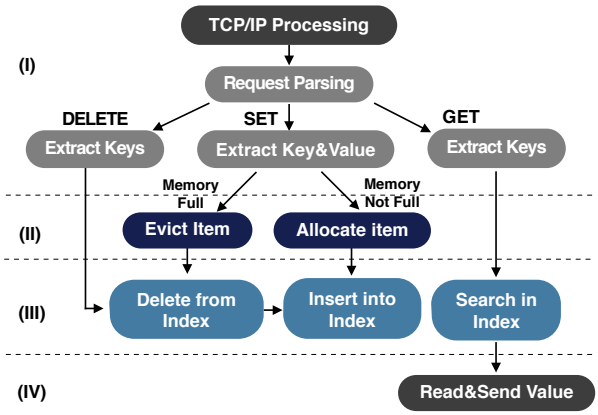

Figure 1: Workflow of a Typical Key-Value Store System

key-value item is removed from both the main memory and the index data structure. In summary, there are four major operations in the workflow of a key-value store system: (I) Network Processing: including network I/O and protocol parsing. (II) Memory Management: including memory allocation and item eviction. (III) Index Operations: including Search, Insert, and Delete. (IV) Read Key-Value Item in Memory: only for GET queries.

Overheads from network processing and concurrent data access had been considered to be the major bottlenecks in a key-value store system [36]. 1) Network processing in a traditional OS kernel, which involves frequent context switches between the OS kernel and user space, causes high instruction cache misses and virtual memory translation overhead. For instance, 70% of the processing time is spent on network processing in CPHash [23], and MemC3 [11] suffers 7× performance degradation with network processing. A set of techniques have been proposed to alleviate this overhead, such as UDP [6], Multiget [2], and bypassing the OS kernel with new drivers [14, 1]. 2) The locks for synchronization among cores in concurrent data accesses can significantly reduce the potential performance enhancement offered by multicore architectures. In recent works, techniques such as data sharding [21, 23] and optimized data structures [11, 22] are proposed to tackle this issue. After the overhead is significantly mitigated with the proposed techniques, the performance bottleneck of an IMKV system shifts. In the following section, we will show that the performance gap between CPU and memory becomes the major factor that limits key-value store performance on the multicore architecture.

2.1.2 *Bottlenecks of Memory Accesses in a CPU-Based Key-Value Store*

Memory accesses in a key value store system consist of two major parts: 1) accessing the index data structure, and 2) accessing the stored key-value item. To gain an understanding of the impact of the two parts of memory accesses in an in-memory key-value store system, we have conducted experiments by measuring the execution time of a GET query of MICA [21]. MICA is a CPU-based in-memory key-value store with the highest known throughput. In the evaluation, MICA adopts lossy index data structure and runs in *EREW*

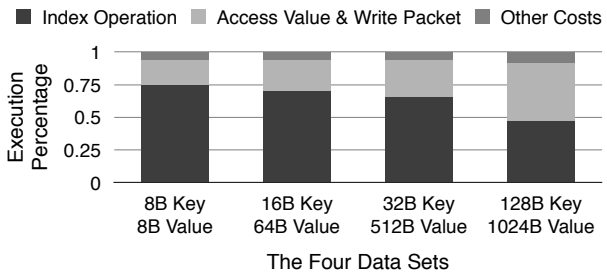

Figure 2: Execution Time Breakdown of a GET Query in a CPU-Based Key-Value Store System

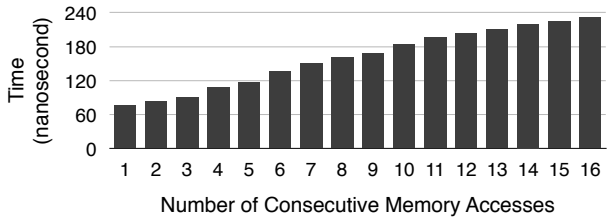

Figure 3: Sequential Memory Access Time

mode with a uniform key distribution. The following four data sets are used as the workloads: 1) 8 bytes key and 8 bytes value; 2) 16 bytes key and 64 bytes value; 3) 32 bytes key and 512 bytes value; 4) 128 bytes key and 1,024 bytes value. This evaluation is conducted on a PC equipped with two Intel Xeon E5-2650v2 CPU and 8×8 GB memory. As shown in Figure 2, index operations take about 75% of the processing time with the 8 bytes key-value workload (data set 1), and take around 70% and 65% of the time for data sets 2 and 3, respectively. For data set 4, the value size increases to 1,024 bytes, and the index operation time portion decreases, but still takes about 50% of the processing time.

With DPDK and UDP protocol, receiving and parsing a packet takes only about 70 nanoseconds, and the cost for each key is significantly amortized with Multiget. For instance, eleven 128-byte keys can be packed in one packet (1500 bytes MTU) so that the amortized packet I/O and parsing cost for each key can be as low as only 7 nanoseconds. MICA needs one or more memory accesses for its lossless index, and one memory access for its lossy index data structure. The key comparison in the index operation may also load the value stored next to the key. That is why the proportion of the accessing value is smaller although they both take one memory access for the data set 1. The CPI (cycles per instruction) of a CPU-intensive task is generally considered to be less than 0.75. For example, the CPI of Linpack on an Intel processor is about 0.42-0.59 [20]. We have measured that the CPI of MICA with the data set 1 is 5.3, denoting that a key-value store is memory-intensive, and the CPU-memory gap has become the major factor that limits its performance.

We have analyzed other popular key-value store systems, and all of them show the same pattern. The huge overhead of accessing index data structure and key-value items is incurred by the memory accesses. The index data structure commonly uses a hash table or a tree, which needs one or more memory accesses to locate an item. With a huge working set, the index data structure may take hundreds of megabytes or even several gigabytes of memory space. Consequently, it cannot be kept in a CPU cache that has only a capacity of tens of megabytes, and each access to the index data structure may result in a cache miss. To locate a value, it generally takes one or more **random** memory accesses. Each memory access fetches a fixed size data block into a cache line in the CPU cache. For instance, with n items, each lookup in Masstree [22] needs $\log_4 n - 1$ random memory accesses, and a cuckoo hash table with k hash functions would require $(k+1)/2$ random memory accesses per index lookup in expectation. The ideal case for a linked-list hash table with load factor 1 is that items are evenly distributed in the hash table and each lookup requires only one memory access. However, the expected worst case cost is $O(\lg n / \lg \lg n)$ [8]. On the other hand, accessing the key-value item is the next step, which consists of **sequential** memory accesses. For instance, a 1 KB key-value item needs 16 sequential memory accesses with a cache line size of 64 bytes. However, sequential memory accesses are much faster than random memory accesses, because the processor can recognize the sequential pattern and prefetch the following cache lines.

We have conducted another experiment to show the performance of sequential memory accesses. We start from accessing one cache line (64 bytes), and continue to increase the number of cache lines to read, up to 16 cache lines (1,024 bytes). Figure 3 shows the time of 1-16 sequential memory accesses. A random memory access takes about 76 nanoseconds in our machine, while 16 sequential memory accesses take 231 nanoseconds, only about 3 times higher than one access. In conclusion, the random memory accesses involved in accessing the index data structure are the main overhead in an in-memory key-value store system.

Memory access overhead cannot be easily alleviated for the following three technical reasons. 1) Memory access latency hiding capability for a multicore system is limited by its CPU instruction window size and the number of Miss Status Holding Registers (MSHRs). For example, an Intel X5550 CPU is capable of handling only 4-6 cache misses [14]. Therefore, it is hard to utilize the inter-thread parallelism. 2) As a thread cannot proceed without the information being fetched from the memory, the intra-thread parallelism can not be explored either. 3) The working set of an IMKV system is very large [6]. Therefore, the CPU cache with a limited capacity is helpless in reducing the memory access latency. With a huge amount of CPU time being spent on waiting for memory to return the requested data, both CPU and memory bandwidth are underutilized. Since accessing the key-value item is inevitable, the only way to significantly improve the performance of an in-memory key-value store system is to find a way to accelerate the random memory accesses in the index operations.

2.2 Opportunities and Challenges by GPUs

2.2.1 *Advantages of GPUs for Key-value Stores*

CPUs are general-purpose processors which feature large cache size and high single-core processing capability. In contrast to CPUs, GPUs devote most of their die areas to large array of Arithmetic Logic Units (ALUs), and execute code in an SIMD (Single Instruction, Multiple Data) fashion. With

the massive array of ALUs, GPUs offer an order of magnitude higher computational throughput than CPUs for applications with ample parallelism. The key-value store system has the inherent massive parallelism where a large volume of queries can be batched and processed simultaneously.

GPUs are capable of offering much higher data accessing throughputs than CPUs due to the following two features. First, GPUs have very high memory bandwidth. Nvidia GTX 780, for example, provides 288.4 GB/s memory bandwidth, while the most recent Intel Core E5-2680v3 processor only has 68 GB/s memory bandwidth. Second, GPUs effectively hide memory access latency by warp switching. Warp (or wavefront called in OpenCL), the basic scheduling unit in Nvidia GPUs, can benefit zero-overhead scheduling by the GPU hardware. When one warp is blocked by memory accesses, other warps whose next instruction has its operands ready are eligible to be scheduled for execution. With enough threads, memory stalls can be minimized or even eliminated [29].

2.2.2 *Challenges of Using GPUs in Key-value Stores*

GPUs have great capabilities to accelerate data-intensive applications. However, they have limitations and may incur extra overhead if are utilized in an improper way.

Challenge 1: Limited Memory Capacity and Data Transfer Overhead. The capacity of GPU memory is much smaller than that of main memory [31]. For example, the memory size of a server-class Nvidia Tesla K40 GPU is only 12 GB, while that of a data center server can be hundreds of gigabytes. Since a key-value store system generally needs to store tens of or hundreds of gigabytes key-value items, it is impossible to store all the data in the GPU memory. With the low PCIe bandwidth, it is nontrivial to use GPUs in building a high performance IMKV.

Challenge 2: Tradeoffs between Throughput and Latency. To achieve high throughput, GPUs need data batching to improve resource utilization. A small batch size for a GPU will result in low throughput, while a large batch size will lead to a high latency. However, a key-value store system is expected to offer a response latency of less than 1 millisecond. Therefore, tradeoffs have to be made between throughput and latency, and optimizations are needed to match IMKV workloads.

Challenge 3: Specific Data Structure and Algorithm Optimization on GPUs. Applications on GPUs require a well-organized data structure and efficient GPU-specific parallel algorithms. However, IMKV needs to process various-sized key-value pairs, which makes it hard to well utilize the SIMD vector units and the device memory bandwidth. Furthermore, as there are no global synchronization mechanisms for all threads in a GPU kernel, a big challenge is posed for algorithm design and implementation, such as the Insert operation.

3. MEGA-KV: AN OVERVIEW

To address the memory access overhead in the IMKV system, Mega-KV offloads the index data structure and its corresponding operations to GPUs. With GPUs' architecture advancement on high memory bandwidth and massive parallelism, the performance of index operations is significantly improved, and the load of CPU is dramatically lightened.

3.1 Major Design Choices

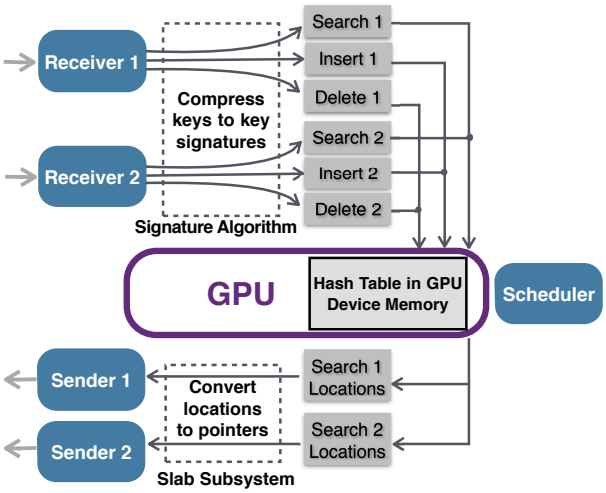

Figure 4: The Workflow of Mega-KV System

Decoupling Index Data Structure from Key-Value Items. Due to the limited GPU device memory size, the number of key-value items that can be stored in the GPU memory is very small. Furthermore, transferring data between GPU memory and host memory is considered to be the major bottleneck for GPU execution. Mega-KV decouples index data structure from key-value items, and stores it in the GPU memory. In this way, the expensive index operations such as Search, Insert, and Delete can be offloaded to GPUs, significantly mitigating the load of CPUs.

GPU-optimized Cuckoo Hash Table as the Index Data Structure. Mega-KV uses a GPU-optimized cuckoo hash table [27] as its index data structure. According to GPUs' hardware characteristics, the cuckoo hash table data structure is designed with aligned and fixed-size cells and buckets for higher parallelism and less memory accesses. Since keys and values have variable lengths, keys are compressed into 32-bit key signatures, and the location of the key-value item in main memory is indicated by a 32-bit location ID. The key signature and location ID serve as the input and output of the GPU-based index operations, respectively.

Periodic GPU Scheduling for Bounded Latency. A key-value store system has a stringent latency requirement for queries. For a guaranteed query processing time, Mega-KV launches GPU kernels in pre-defined time intervals. At each scheduled time point, jobs accumulated in the previous batch are launched for GPU processing. GET queries need fast responses for quality of services, while SET and DELETE queries have a less strict requirement. Therefore, Mega-KV applies different scheduling policies on different types of queries for higher throughput and lower latency.

3.2 The Workflow of Mega-KV

Figure 4 shows the workflow of Mega-KV in a CPU-GPU hybrid system. Mega-KV divides query processing into three stages: pre-processing, GPU processing, and post-processing, which are handled by three kinds of threads, respectively. *Receiver* threads are in charge of the pre-processing stage, which consists of packet parsing, memory allocation and eviction, and some extra work for batching, i.e., components (I) and (II) in Figure 1. *Receivers* batch Search,

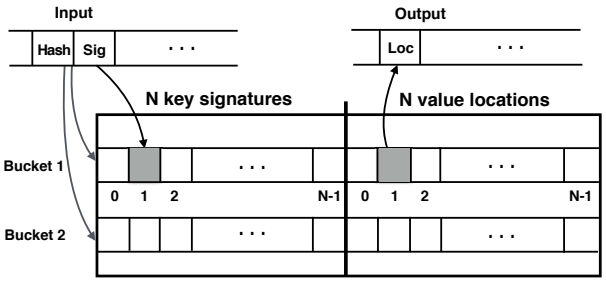

Figure 5: GPU Cuckoo Hash Table Data Structure

Insert and Delete jobs separately into three buffers. The batched input is 32-bit key signatures which are calculated by performing a signature algorithm on the keys. *Scheduler*, as the central commander, is a thread that is in charge of periodic scheduling. It launches the GPU kernel after a fixed time interval to process the query operations batched in the previous time window. *Sender* threads handle the post-processing stage, including locating the key-value items with the indexes received from the GPU, and sending responses to clients. Mega-KV uses slab memory management where each slab object is assigned with a 32-bit location ID. After the key-value item locations for all the Search jobs are returned, *Sender* threads convert the locations to item pointers through the slab subsystem (see Section 6.2 for details). Because the overhead from packet I/O, query parsing, and the memory management are still heavy, several *Receiver* and *Sender* threads are launched in pairs to form the pipelines, while there is only one *Scheduler* per GPU.

4. GPU-OPTIMIZED CUCKOO HASH TABLE

4.1 Data Structure

Since the GPU memory capacity is small, the hash table data structure should be efficiently designed for indexing a large number of key-value items. Mega-KV adopts cuckoo hashing, which features a high load factor and a constant lookup time. The basic idea of cuckoo hashing is to use multiple hash functions to provide each key with multiple locations instead of one. When a bucket is full, existing keys are relocated to make room for the new key.

There are two parameters affecting the load factor of a cuckoo hash table and the eviction times for insertion: the number of hash functions, and the number of cells in a bucket. Increasing either of them can lead to a high load factor [10]. Since the GPU memory size is limited and the random memory access overhead of a cuckoo eviction is high, we use a small number of hash functions (two) and a large number of cells per bucket in our hash table design.

The various-sized keys and values not only impose a huge overhead on data transfer, but also make it hard for GPU threads to locate their data. In our hash table design, a 32-bit key signature is used to identify a key, and a 32-bit location ID is used to reference the location of an item in the main memory. As shown in Figure 5, a hash bucket contains N cells, each of which stores a key signature and the location of the corresponding key-value item. The key signatures and locations are packed separately for coalesced memory access. Each key is hashed onto two buckets, where a 32-bit hash value is used as the index of one bucket, and the index of the other bucket is calculated by performing an XOR operation on the signature and the hash value. The compactness of the key signatures and locations also lead to a small hash table size. For instance, for an average key-value item size of 512 bytes, a 3 GB hash table is capable of indexing 192 GB data.

The overhead of generating key signatures and hash values is small with the built-in SSE instructions of Intel x86 CPU, such as AES, CRC, or XOR. In a 2 GB hash table with 8 cells (one cell has one key signature and its location) per bucket, there will be 2^{25} buckets with 25 bits for hashing. With 32 bits for signature, the hash table can hold up to $2^{31}/8 = 2^{28}$ elements, with a collision rate of as low as $1/2^{25+32} = 1/2^{57}$.

4.2 Hash Table Operations

A straightforward implementation of the hash table operations on GPUs may result in low resource utilization, and consequently low performance. As the L2 cache size of a GPU is very small (1.5 MB for Nvidia GTX 780), only a few buckets can be operated at the same time. When tens of thousands of threads are operating on different buckets simultaneously, the fetched memory block may be immediately evicted out of the cache after one access. Consequently, if a thread is trying to match all the signatures in a bucket, the fetched memory block has a high possibility of being evicted after each access; thus it has to be fetched from memory repeatedly. To address this issue, we propose to form multiple threads as a *processing unit* for cooperative processing. With a bucket of N cells, N threads can form a *processing unit* to check all the cells in the bucket to find an empty one simultaneously. Since the threads forming a *processing unit* are selected within one warp, they perform the operations simultaneously. Therefore, after the operations are performed, the data is no longer needed and can be evicted from cache. Based on this approach, this section describes the specific hash table operations optimized for GPU execution.

Search: A Search operation checks all 2×N candidate key signatures in the two buckets, and writes the corresponding location into the output buffer. When searching for a key, the threads in the *processing unit* compare the signatures in the bucket in parallel. After that, the threads use the built-in vote function *__ballot()* to inform each other with the information of the corresponding cells. With the *__ballot()* result, all the threads in the *processing unit* know if there is a match in the current bucket and its position. If none of the threads find a match, they will do the same process on the alternative bucket. If there is a match, the corresponding thread will write the location ID to the output buffer.

Insert: For Insert operation, the *processing unit* firstly tries to find if there are the same signatures in the two buckets, i.e., conflicts. If there are conflicts, the conflicting location is replaced with the new one, or the *processing unit* will try to find an empty cell in the two buckets. With *__ballot()*, all threads will know the positions of all the empty cells. If either of the two buckets has an empty cell, the thread that is in charge of the corresponding cell tries to insert the key-value pair. After the insertion, *__synchronize()* is performed to make sure all memory transactions have been done within the thread block for checking whether the insertion is successful. If not, the *processing unit* will try again. There will be at least one successful insertion for the conflict in a cell.

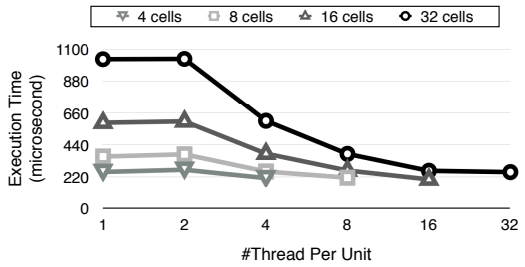

Figure 6: Processing time of Search operation with 60K jobs and 7 CUDA streams

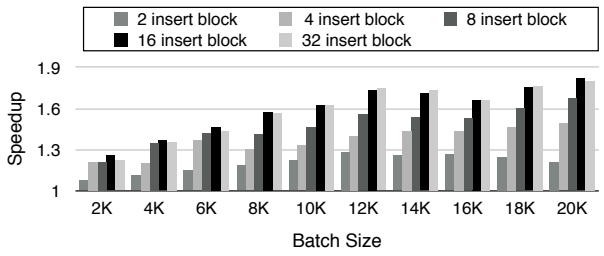

Figure 7: Speedups for Insert with Multiple Blocks

If neither bucket has empty cells, a randomly selected cell from one candidate bucket is relocated to its alternative location. Displacing the key may also require kicking out another existing key, which will repeat until a vacant cell is found, or until a maximum number of displacements is reached.

There may be severe write-write conflicts if multiple *processing units* are trying to insert an item in the same position. To alleviate the conflict, instead of always trying to insert into the first available cell in a bucket, a *preferred cell* is assigned for each key. We use the highest bits of a signature to index its *preferred cell*. For instance, 3 bits can be used to indicate the *preferred cell* in a bucket with 8 cells. If multiple available cells in a bucket are found, the cell left nearest to the *preferred cell* is chosen for insertion. There will be no extra communication between the threads in a *processing unit*, since each of them knows whether its cell is chosen with the *__ballot()* result.

Delete: Delete operation is almost the same with Search. When both the signature and location are matched, the corresponding thread clears the signature to zero to mark the cell as available.

4.3 Hash Table Optimization and Performance

In this section, we use a Nvidia GTX 780 in evaluating the hash table performance.

4.3.1 The Choice of Processing Unit and Bucket Size

To evaluate the effectiveness of *processing unit*, Figure 6 shows the GPU execution time for Search operation with a different number of cells in one bucket and a different number of threads in a *processing unit*. Since a memory transaction is 32 bytes in the non-caching mode of Nvidia GPUs, the bucket size is set as multiples of 32 bytes to efficiently utilize memory bandwidth. As shown in the figure, decreasing the number of cells and increasing the number of threads in *processing unit* lead to a reduced execution time. We choose 8 cells for each bucket, which allows a higher load factor. And correspondingly, 8 threads form a *processing unit*.

4.3.2 Optimization for Insert

Insert operation needs to make sure whether the key-value index has been successfully inserted, and a *__synchronize()* operation is performed after the insertion. However, this operation can only synchronize threads within a thread block, but cannot synchronize threads in different thread blocks. A naive approach to implementing Insert operation is to launch only one thread block. However, a thread block can

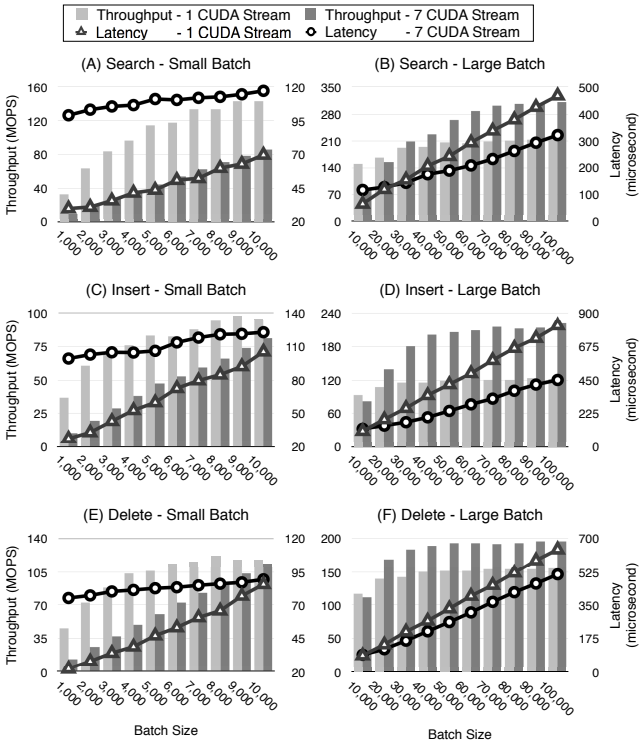

Figure 8: GPU Hash Table Performance

only execute on one streaming multiprocessor, leading to resource underutilization and low performance.

We divide the hash table into several logical partitions. According to the hash value, key signatures are batched into different buffers with each buffer belonging to a logical partition exclusively. For the alternative bucket in cuckoo hash, a mask is used to make it still locate in the same partition. With each thread block processing one input buffer and one logical partition exclusively, throughput of Insert operation is boosted by utilizing more streaming processors. Figure 7 shows the performance improvement with multiple insert blocks. As can be seen in the figure, an average of 1.2 speedup is achieved with 2 blocks, and the throughput becomes 1.3-1.8 times higher with 16 blocks.

4.3.3 Hash Table Performance

Figure 8 shows the throughput and processing latency for hash table operations with different input batch size. Comparing with 1 CUDA stream, a 24%-60% performance improvement is achieved with 7 CUDA streams, which ef-

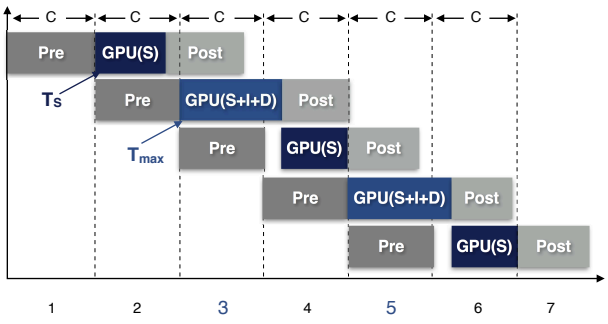

Figure 9: Periodical Delayed Scheduling

fectively overlaps kernel execution with data transfer. Our hash table implementation shows the peak performance for Search, Insert, and Delete operations with large batch sizes are 303.7 MOPS, 210.3 MOPS, and 196.5 MOPS, respectively. For a small batch, the performance of 7 streams is lower than that of 1 stream. This is because GPU resources are underutilized for a small batch, which is partitioned into smaller ones with the multiple CUDA streams.

4.4 The Order of Operations

The possibility of conflicting operations on the same key within such a microsecond scale time interval is very small, and there is no need to guarantee their orders. This is because, in multicore-based key-value store systems, the operations may be processed by different threads, which may have context switching and compete for a lock to perform the operations. Moreover, the workload of each thread is different and the delays of packets transferred in the network also vary over a large range. For example, TCP processing may take up to thousands of milliseconds [18], and queries may wait for tens of milliseconds to be scheduled in a data processing system [25]. Therefore, the order of operations within only hundreds of microseconds can be ignored, and the order of operations within one GPU batch is not guaranteed in Mega-KV. For the read-write or other conflicts on accessing the same item among the CPU threads, a set of optimistic concurrency control mechanisms are proposed in the implementation of Mega-KV (Section 6.4).

5. SCHEDULING POLICY

In this section, we study the GPU scheduling policy to balance between throughput and latency.

5.1 Periodical Scheduling

To achieve a bounded latency for GPU processing, we propose a periodical scheduling mechanism based on the following three observations. First, the majority of the hash table operations are Search in a typical workload, while Insert and Delete operations account for a small fraction. For example, a 30:1 GET/SET ratio is reported in Facebook Memcached workload [6]. Second, as shown in Section 4.3, the processing time for Insert and Delete operations increase very slowly when the batch size is small. Third, SET queries have less strict latency requirement than that of GET queries.

Different scheduling cycles are applied on Search, Insert, and Delete operations, respectively. Search operations are launched for GPU processing after a query batch time C.

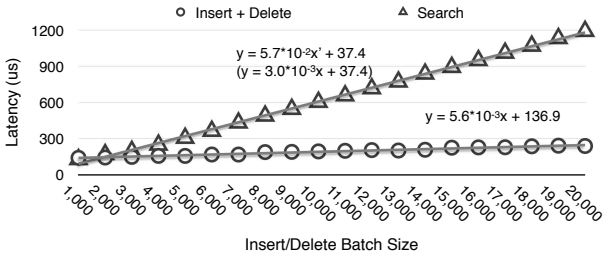

Figure 10: Fitting Line for Search and Insert/Delete Operation (95% GET, 5% SET)

Insert and Delete operations, however, are processed for every $n \times C$. We define the GPU execution time for Search operations batched in C as T_S. In Mega-KV, we assume that GPUs are capable of handling the input queries, and we can get $T_S < C$. Figure 9 shows an example with $n = 2$. In the example, Search operations accumulated in the last time interval C are processed in the next time interval, while Insert and Delete operations are launched for execution every $2 \times C$. We define the sum of T_S and the time for Insert and Delete operations batched in $n \times C$ as T_{max}. To guarantee that Search operations that have been batched in a time interval C can be processed within the next time interval, the following rule should be satisfied:

$$T_S + T_{max} \leq 2 \times C \tag{1}$$

With the time for reading the value and sending the response, a maximum response time of $3 \times C$ is expected for the GET queries.

5.2 Lower Bound of Scheduling Cycle

Figure 10 shows the fitting lines for the relation of processing time and batch size for Search and Insert & Delete operations on Nvidia GTX 780. The workload has 95% GET and 5% SET queries with a uniform distribution. Both lines are drawn with the same horizontal axis, which is the batch size for Insert. With an Insert batch size x, the corresponding batch size for Search is $19x$. As is shown in the figure, the processing time of Search operations increases more quickly than Insert/Delete operations with the growth of the batch size, which also proves that Insert and Delete operations are fit to be batched for a longer time in read heavy IMKV systems.

Since the processing time T_S of Search operation almost increases linearly with the increase of the batch size x_S, we define it as $T_S = k_S \cdot x_S + b_S$, where $k_S = 3.0 \times 10^{-3}$ and $b_S = 37.4$. Similarly, for the Insert/Delete operations, we define its relation between batch size x_I and processing time T_I as $T_I = k_I \cdot x_I + b_I$, where $k_I = 5.6 \times 10^{-3}$ and $b_I = 136.9$. With a fixed input speed V, the relations between the batching time t and the processing time are $T_S = k_S \cdot p_S \cdot V \cdot t + b_S$ and $T_I = k_I \cdot p_I \cdot V \cdot t + b_I$, where p_S is the proportion of Search operations, and $p_I = 1 - p_S$ is the proportion of Insert operations, which are 95% and 5% in the figure, respectively.

The maximum processing time $T_{max} = (k_S \cdot p_S \cdot V \cdot C + b_S) + (k_I \cdot (1 - p_S) \cdot V \cdot C \cdot n + b_I)$. To satisfy the inequation $T_S + T_{max} \leq 2 \times C$, we get

$$C \geq \frac{2 \cdot b_S + b_I}{2 - 2 \cdot k_S \cdot p_S \cdot V - n \cdot k_I \cdot (1 - p_S) \cdot V} \tag{2}$$

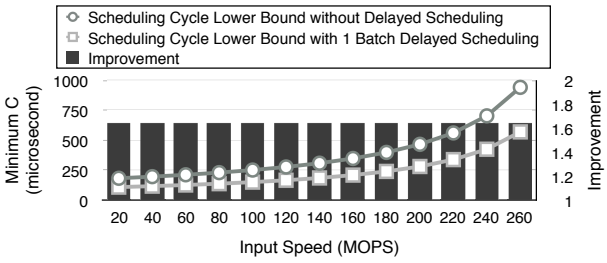

Figure 11: Latency Improvement with Delayed Scheduling

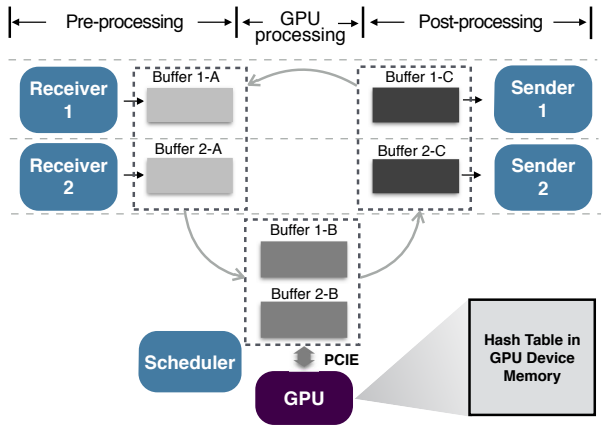

Figure 12: The Framework of Mega-KV System

where $V < 2/(2 \cdot k_S \cdot p_S + n \cdot k_I \cdot (1 - p_S))$, and $n \geq 2$. From the formula, we learn that an increasing n leads to a larger lower bound of C. Therefore, with the same input speed, we get the minimum C with $n = 2$.

Without the delayed scheduling, C should follow $C \geq T_{max}$, and we get $C \geq (b_S + b_I)/(1 - k_S \cdot p_S \cdot V - k_I \cdot (1 - p_S) \cdot V)$, where $V < 1/(k_S \cdot p_S + k_I \cdot (1 - p_S))$. Figure 11 shows the reduction on the lower bound of scheduling cycle C with the delayed scheduling, where an average of 65% reduction is achieved. The delayed scheduling policy not only offers a reduced overall latency, but also makes the system capable of achieving a higher throughput with the same latency.

6. SYSTEM IMPLEMENTATION AND OPTIMIZATION

To build a high performance key value store that is capable of processing hundreds of millions of queries per second, the overheads from data copy operations, memory management, and locks should be addressed. In this section, we illustrate the framework of Mega-KV and the major techniques used in alleviating the overheads.

6.1 Zero-copy Pipelining

Data copy is known to be a big overhead in a high-speed networking system, which may limit the overall system performance. To avoid the expensive memory copy operations between pipeline stages, each pipeline is assigned with three buffers, and data between the stages is transferred by passing the buffers. Figure 12 shows the zero-copy pipelining framework of Mega-KV . At any time, each *Receiver* works on one buffer to batch incoming queries. When GPU kernel launching time arrives, *Scheduler* uses an available buffer to swap the one that *Receiver* is working on. In a system configured with N *Receivers*, N CUDA streams are launched to process the buffer of each *Receiver*. After the GPU kernel completes execution, *Scheduler* handles the buffer to *Sender* for post-processing. *Sender* marks its buffer as available after it completes the post-processing. With this technique, the overhead of data transferring between pipeline stages is significantly mitigated.

6.2 Memory Management

Slab Memory Management. Mega-KV uses slab allocation. The *location* of an item, which is 32 bits, is used in the hash table to reference where the item is located in the main memory. Through the slab data structure, a mapping is made between the *location* and the corresponding item, where the highest bits in the *location* are used to indicate which slab it belongs to, and the rest of the bits stand for the offset.

CLOCK Eviction. Each slab adopts a bitmap-based CLOCK eviction mechanism where the item offset in a bitmap is the same with its offset in the slab. A walker pointer traverses the bitmap and performs the CLOCK algorithm for eviction. By tightly associating *location* with the slab and bitmap data structures, both locating an item and updating the CLOCK bitmap can be performed with extremely low overhead.

If there is a conflict when inserting an item in the hash table, the conflicting *location* is replaced with the new one (Section 4.2), and the conflicting item stored in main memory should be evicted for memory reuse. With the CLOCK eviction mechanism, the conflicting items in the main memory will be evicted after it is not accessed for a period. Therefore, no further actions need to be performed on the conflicting items. This also works for the items that are randomly evicted when a maximum number of displacements is reached in cuckoo hash insertion.

Batched Lock. With a shared memory design among all threads, synchronization is needed for memory allocation and eviction. Since acquiring a lock in the critical path of query processing has a huge impact on the overall performance, batched allocation and eviction are adopted to mitigate its overhead. Each allocation or eviction will return a memory chunk, containing a list of fixed-size items. Correspondingly, each *Receiver* thread maintains a local slab list for storing the allocated and evicted items. By amortizing the lock overhead across hundreds of items, the performance of memory management subsystem is dramatically improved.

6.3 APIs: get and getk

The same as Memcached, Mega-KV has two APIs for GET: *get* and *getk*. When a *get* query is received, Mega-KV is responsible for making sure that the value sent to the client matches the key. Therefore, before sending a found value to clients, its key stored in the main memory is compared to confirm the match. If the keys are the same, the value is sent to the client, or *NOT_FOUND* is sent to notify that the key-value item is not stored in Mega-KV.

Since a *getk* query asks a key-value store to send the key with the value to the client, the client is capable of matching the key with its value. Therefore, Mega-KV does not com-

pare the keys to confirm the match, and requires its client to do the job. Our design choice is mainly based on two factors: 1) the false positive rate/conflict rate is very low; and 2) the key comparison cost is comparatively high. Therefore, avoiding the key comparison operation for each query will lead to a higher performance.

6.4 Optimistic Concurrent Accesses

To avoid adopting locks in the critical path of query processing, the following optimistic concurrent accesses mechanisms are applied in Mega-KV.

Eviction An item cannot be evicted under the following two situations. First, an free slab item that has not been allocated should not be evicted. Second, if an item is deleted and recycled to a free list, it should not be evicted. To handle the two situations, we assign a status tag to each item. Items in the free lists are marked as *free*, and the allocated slab items are marked as *using*. The eviction process checks the item's status, and will only evict items with a status tag of *using*.

Reading vs. Writing. An item may be evicted when other threads are reading the value. Under such a scenario, the thread checks the status tag of the item after finishing reading its value. If the status tag is not *using* any more, the item has already been evicted. Since the value read may be wrong, a *NOT_FOUND* response will be sent to the client.

Buffer Swapping. *Receiver* does not know when *Scheduler* swaps its buffer. Therefore, a query may not be successfully batched in the buffer if the buffer is swapped before the last moment. To address this issue without using locks, we record the buffer ID before a query is added into the buffer, and check if the buffer has been swapped after the insertion. If the buffer has been swapped during the process, *Receiver* is not sure whether the query has been successfully inserted, and the query is added into the new buffer again.

For Search operation, if the buffer has been swapped, the total number of queries in the buffer is not increased so that the new query, whether or not it has been added into the buffer, will not be processed by the *Sender*. For Insert operation, the object can be inserted twice, as the latter one will overwrite the former one. Therefore, the correctness will not be affected, and so does the Delete operation.

7. EXPERIMENTS

In this section, we evaluate the performance and latency of Mega-KV under a variety of workloads and configurations. We show that Mega-KV achieves a super high throughput with a reasonably low latency.

7.1 Experimental Methodology

Hardware. We conduct the experiments on a PC equipped with two Intel Xeon E5-2650v2 octa-core processors running at 2.6 GHz. Each processor has an integrated memory controller installed with 8×8 GB 1600MHz DDR3 memory, and supports a maximum of 59.7 GB/s memory bandwidth. Each socket is installed with a Nvidia GTX 780 GPU as our accelerator. GTX 780 has 12 streaming multiprocessors and a total of 2,304 cores. The device memory on each GPU is 3 GB GDDR5, and the maximum data transfer rates between main memory and device memory are 11.5 GB/s (Host-to-Device) and 12.9 GB/s (Device-to-Host). The operating system is 64-bit Ubuntu Server 14.04 with Linux kernel version 3.13.0-35. Each socket is installed with an Intel 82599 dual port 10 GbE card, and the open source DPDK [1] is used as the driver for high-speed I/O. We also use another two PCs in the experiments as clients.

Mega-KV Configuration. For the octa-core CPU on each socket, one physical core is assigned with a *Scheduler* thread. Each *Scheduler* controls all the other threads on the socket and launches kernels to its local GPU. Since *Receiver* is compute-intensive while *Sender* is memory-intensive, we enable hyper-threading in the machine, and assign each of the left 7 physical cores with one *Receiver* and one *Sender* thread, forming a pipeline on the same physical core. Therefore, there are 7 pipelines in our system on each socket. The AES instruction from the SSE instruction set is used in calculating key signature and hash value.

Data sharding is adopted for the NUMA system. By partitioning data across the sockets, each GPU will only index the key-value items located in its local socket. This avoids remote memory accesses, which are considered to be a big overhead. In this way, Mega-KV achieves good scalability with multiple CPUs and GPUs.

Comparison with CPU-based IMKV system. We take the open source in-memory key-value store MICA as our baseline for comparison. According to MICA's experimental environment, the hyper-threading is disabled. We modify the *microbench* in MICA for the evaluation, which includes loading different size key-value items from local memory and writing values to packet buffers. All the experiments are performed in its EREW mode with MICA's lossy index data structure. On the same hardware platform with Mega-KV, the performance of MICA is measured and shown in Section 7.2.3.

Workloads. We use four datasets in the evaluation: *a*) 8 bytes key and 8 bytes value; *b*) 16 bytes key and 64 bytes value; *c*) 32 bytes key and 512 bytes value; and *d*) 128 bytes key and 1024 bytes value. In the following experiments, workload *a* is evaluated by feeding queries via network. To allow a high query speed via network transmission, clients batch requests and Megak-KV batches responses in an Ethernet frame as much as possible. For the large key-value pair workloads *b*, *c* and *d*, the network becomes the bottleneck. Thus, keys and values are generated in local memory to evaluate the performance.

Both uniform and skewed workloads are used in the evaluation. The uniform workload uses the same key popularity for all queries. The key popularity in the skewed workload follows a Zipf distribution of skewness 0.99, which is the same with YCSB workload [7]. Our clients use approximation Zipf distribution generation described in [13] for fast workload generation. For all the workloads, the working set size is 32 GB, and a 2 GB GPU hash table is used to index the key-value items. The key-value items are preloaded in the evaluation, and the hit ratio of GET query is higher than 99.9%.

7.2 System Throughput

7.2.1 Theoretical Maximum System Throughput

As discussed in Section 5, the system throughput V is closely related to the scheduling cycle C. Since our system needs to guarantee that the GPU processing time for each batch is less than the scheduling cycle C, the theoretical maximum throughput should be known in advance before the evaluation.

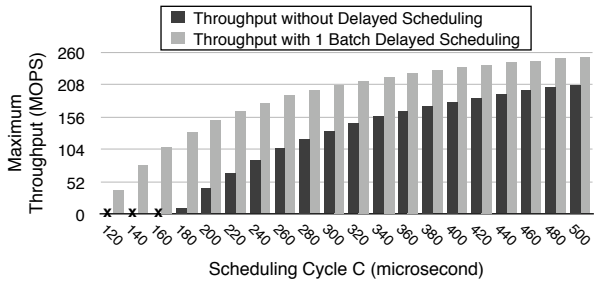

Figure 13: Theoretical Maximum Speed for One GPU

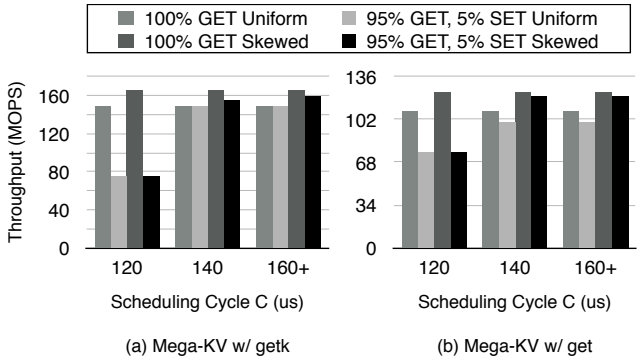

Figure 14: System Throughput with Two CPUs and GPUs (8 Bytes Key & Value)

The major mission of the periodical scheduling policy is to satisfy the inequation (1) $T_S + T_{max} \leq 2 \times C$. With the delayed scheduling, we get

$$V \leq \frac{2 \cdot C - 2 \cdot b_S - b_I}{2 \cdot k_S \cdot p_S \cdot C + 2 \cdot k_I \cdot (1 - p_S) \cdot C} \quad (3)$$

where $C > (2 \cdot b_S + b_I)/2$. Without the delayed scheduling, V and C form the relation $0 \geq V \geq (C - b_S - b_I)/(k_S \cdot p_S \cdot C + k_I \cdot (1 - p_S) \cdot C)$, where $C > b_S + b_I$. For the 95% GET workload, we get $C > 105.8$ with the delayed scheduling, and $C > 174.3$ without the delayed scheduling.

With the same parameters k_S, b_S, k_I, and b_I listed in Section 5.2, Figure 13 demonstrates the relation between the allowed maximum throughput and the scheduling cycle C for the workload with 95% GET and 5% SET. The theoretical maximum throughput of Mega-KV increases with the increasing of the scheduling cycle, and will reach an upper bound with a large enough scheduling cycle, which is the maximum GPU throughput. Compared with scheduling all operations with the same scheduling cycle, system performance is improved by 14%-400% with delayed Insert and Delete scheduling. It is worth noting that for $C \leq 174.3$ microseconds, the system cannot work without the delayed scheduling. This is because the total GPU execution and data transfer overhead for Search, Insert, and Delete kernels is higher than the scheduling cycle when the batch size is small. With the delayed scheduling, the scheduling cycle C is required to be greater than 105.8 microseconds.

7.2.2 System Throughput with Network Processing

To measure the maximum throughput of Mega-KV, we use workload *a* where both keys and values are 8 bytes so that the network is capable of transferring a huge amount of queries. This experiment is performed with network processing, where Mega-KV receives requests from clients and sends back responses through the NIC ports. In the experiments, we find that the throughput that GPUs offer is much higher than that of CPUs in the pipeline of Mega-KV. After the expensive index operations are offloaded to GPUs, the memory accesses to the key-value items become the major factor that limits the system performance.

Figure 14 plots the throughput of Mega-KV with workload *a*. Both the performance of *getk* query and *get* query are measured. With *getk* query, Mega-KV achieves the maximum throughput of 160 MOPS for 95% GET and 5% SET, and 166 MOPS for 100% GET skewed workload, respectively. With *get* query, the throughput of Mega-KV is 120 MOPS (95% GET and 5% SET) and 124 MOPS (100% GET) for the skewed workload. For the uniform workload, the throughput is 6%-10% lower. In a real-world scenario where a mix of *get* and *getk* queries are sent by clients, the performance of Mega-KV should lie between *get* and *getk*'s maximum throughput, i.e., 120-166 MOPS for the skewed workload and 100-150 MOPS for the uniform workload. With the 95% GET workload, Mega-KV is 1.6-2.2 times as fast as MICA (76.3 MOPS), the CPU-based IMKV system with the highest known throughput. To address the limitations from the memory accesses, these experiments are performed with a GET hit ratio of more than 99.9%.

The skewed workload has a better performance than the uniform workload. This is because our system applies a shared memory design within each NUMA node; thus the skewed workload will not result in an imbalanced load on the CPU cores. As accessing key-value items in memory is still the major bottleneck, the most frequently visited key-value items may benefit from the CPU cache, and consequently the skewed workload leads to a higher throughput.

7.2.3 System Throughput with Large Key-Value Sizes

Because the network bandwidth may become a bottleneck for large key-value items, the performance for various sized keys and values is measured by loading generated key-value items from the local memory, and values are written into packet buffers as responses.

Figure 15 shows the performance of Mega-KV with workloads *b*, *c*, and *d* by loading data from memory. For the 95% GET 5% SET skewed workload, Mega-KV achieves 110 MOPS, 55 MOPS, and 34 MOPS with *get* query, and achieves 139 MOPS, 60 MOPS, and 30 MOPS with *getk* query. For the 100% GET skewed workload, Mega-KV achieves 107 MOPS, 56 MOPS, and 36 MOPS with *get* query, and achieves 144 MOPS, 62 MOPS, and 33 MOPS with *getk* query, respectively. As is shown in the figure, the key comparison operation involved in the *get* query processing degrades system throughput for 20%-30% with the small size key-value workload. With a larger key-value size, the overhead of accessing the large values in the memory becomes dramatically higher, and thus the ratio of key comparison cost is much smaller and even neglectable.

For comparison, we also measure the performance of CPU-based MICA as the baseline. The throughput of Mega-KV is 1.3-2.9 times as high as MICA with 95% GET 5% SET,

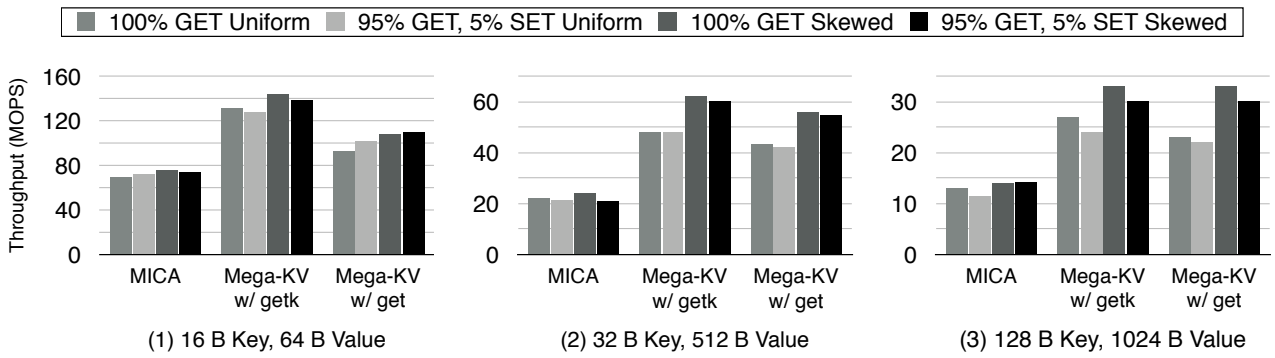

Figure 15: Mega-KV Performance with Large Key-Value Size

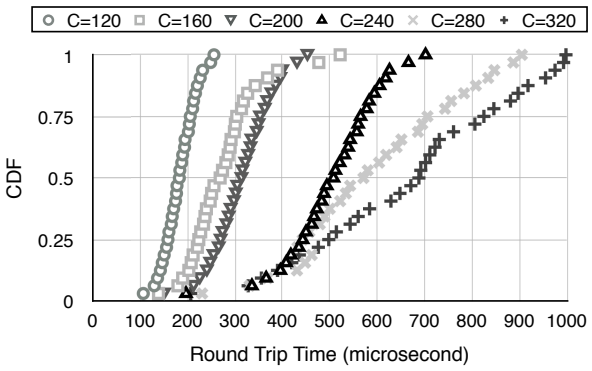

Figure 16: System Latency

and is 1.3-2.6 times as high as MICA with 100% GET.

7.3 Response Time Distribution

We evaluate system processing latency by measuring the time elapsed from sending a *getk* query to receiving its response. The client keeps sending queries with a 95% GET and 5% SET workload, and the client IP address is increased by 1 for each packet. By sample logging the IP addresses and the sending/receiving time, the round trip time can be calculated as the time elapsed between the queries and responses with matched IP addresses.

The scheduling cycle has an impact on the both the system performance and the processing latency. Figure 16 shows the CDF (Cumulative Distribution Function) of query round trip latency with different scheduling cycles. We fix the scheduling cycle C to 120, 160, 200, 240, 280, and 320 microseconds, respectively, and use the allowed maximum input speed for each scheduling cycle. As shown in the figure, the round trip time of queries is effectively controlled within $3 \times C$, which demonstrates the effectiveness of our GPU scheduling policy in achieving predictable latency.

The latency of Mega-KV is comparatively higher than a fast CPU-based implementation. Specifically, if an application requires a strictly low latency of less than 317.4 (105.8 $\times$ 3) microseconds, Mega-KV cannot guarantee that the deadline could be met by all the queries. This is because the minimum scheduling cycle C for Mega-KV is 105.8 microseconds (Section 7.2.1). MICA, which adopts DPDK as its NIC driver, achieves a latency lying between 24-52 microseconds. As is shown in Figure 13, Mega-KV achieves a maximum throughput of 204 MOPS for the 95% GET and 5% SET workload with $C = 160$ microseconds. With this configuration, the average and 95th percentile latencies of Mega-KV are about 280 microseconds and 410 microseconds (Figure 16). However, in Facebook, the average and 95th percentile latencies of web servers in production getting keys are about 300 microseconds and 1,200 microseconds, respectively [25]. Therefore, although the latency of Mega-KV is higher than that of a fast CPU-based implementation, it is still capable of meeting the requirements of the current data processing systems.

8. RELATED WORK

CPU-based in-memory key-value stores [11, 22, 24, 23] have been focusing on designing efficient index data structure and optimizing network processing to achieve higher performance. MICA has compared itself with RAMCloud, MemC3, Memcached, and Masstree in its paper, and shown an at least 4 times higher throughput than the next best system. That is why we choose MICA for performance comparison. Systems such as Chronos [19], Pilaf [24], and RAMCloud [26] focus on low latency processing, which achieve latencies of less than 100 microseconds. Specifically, by taking advantage of Infiniband, RAMCloud and Pilaf achieve average latencies of as low as 5 microseconds and 11.3 microseconds, respectively.

Rhythm [4] proposes to use GPUs in accelerating web server workloads, which also batches requests to trade response time for higher throughput. The latency of the requests in Rhythm is above 10 milliseconds, which is hundreds of times higher than that of a CPU-based web server, but its throughput is about 10 times higher.

Although previous work has adopted GPUs in a key-value store system [17], it only ports the existing Memcached implementation to an OpenCL version, and does not explore GPUs' hardware characteristics for higher performance.

Graphic applications do not need a persistent hash table; instead, they need hash tables to be quickly constructed, such as [5]. The methods adopted in building the hash tables may result in construction failures, and do not fit for key-value store systems.

There are already a set of research papers on adopting GPUs in database systems [35, 33, 32, 28, 15, 16, 34]. The techniques we developed in Mega-KV can also be utilized to

accelerate the relational database query processing. For example, cuckoo hash table is adopted in implementing GPU-based hash join [35]. Therefore, Mega-KV's GPU-optimized cuckoo hash table with its corresponding operations is a good candidate for accelerating the hash join operation.

9. CONCLUSION

Having conducted thorough experiments and analyses, we have identified the bottleneck of IMKV running on multi-core processors, which is a mismatch between the unique properties IMKV for increasingly large data processing and the CPU-based architecture. We have made a strong case by designing and implementing Mega-KV for GPUs to serve as special-purpose devices to address the bottleneck that multicore architectures cannot break. Our evaluation results show that Mega-KV advances the state of the art of IMKV by significantly boosting its performance up to 160+ MOPS.

Mega-KV is an open source software. The source code can be downloaded from `http://kay21s.github.io/megakv`.

10. ACKNOWLEDGMENT

We thank the reviewers for their feedback. This work was supported in part by the National Science Foundation under grants CCF-0913050, OCI-1147522, and CNS-1162165.

11. REFERENCES

[1] Intel dpdk. "http://dpdk.org/".
[2] Memcached. "http://memcached.org/".
[3] Redis. "http://redis.io/".
[4] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *ASPLOS*, pages 19–34, 2014.
[5] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the gpu. In *SIGGRAPH Asia*, pages 154:1–154:9, 2009.
[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, pages 53–64, 2012.
[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.
[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithm, Thrid Edition*. The MIT Press, 2009.
[9] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. In *PVLDB*, 2013.
[10] Ú. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *WDAS*, pages 1–6, 2006.
[11] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, pages 371–384, 2013.
[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi. A case for specialized processors for scale-out workloads. In *Micro*, pages 31–42, 2014.
[13] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.
[14] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *SIGCOMM*, pages 195–206, 2010.
[15] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. In *PVLDB*, 2011.
[16] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. In *PVLDB*, pages 709–720, 2013.
[17] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *ISPASS*, pages 88–98, 2012.
[18] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *NSDI*, 2014.
[19] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *SoCC*, pages 9:1–9:14, 2012.
[20] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 2002.
[21] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *NSDI*, pages 429–444, 2014.
[22] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, pages 183–196, 2012.
[23] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: A cache-partitioned hash table. In *PPoPP*, pages 319–320, 2012.
[24] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, pages 103–114, 2013.
[25] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.
[26] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, pages 92–105, 2010.
[27] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, pages 122–144, 2003.
[28] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *ICDE*, pages 508–519, 2014.
[29] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP*, pages 73–82, 2008.
[30] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.
[31] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang. Gdm: Device memory management for gpgpu computing. In *SIGMETRICS*, pages 533–545, 2014.
[32] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. In *PVLDB*, pages 1543–1554, 2012.
[33] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. In *PVLDB*, pages 1011–1022, 2014.
[34] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *MICRO*, pages 107–118, 2012.
[35] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. In *PVLDB*, pages 817–828, 2013.
[36] H. Zhang, B. M. Tudor, G. Chen, and B. C. Ooi. Efficient in-memory data management: An analysis. In *PVLDB*, 2014.