The VLDB Journal (2017) 26:729–750
DOI 10.1007/s00778-017-0479-0

REGULAR PAPER

A distributed in-memory key-value store system on heterogeneous CPU–GPU cluster

Kai Zhang[1] · **Kaibo Wang**[2] · **Yuan Yuan**[3] · **Lei Guo**[2] · **Rubao Li**[3] · **Xiaodong Zhang**[3] · **Bingsheng He**[4] · **Jiayu Hu**[5] · **Bei Hua**[5]

Received: 13 February 2017 / Revised: 23 June 2017 / Accepted: 11 August 2017 / Published online: 21 August 2017

Abstract In-memory key-value stores play a critical role in many data-intensive applications 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. However, our experiments show that homogeneous multicore CPU 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 does 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 present the design and implementation of Mega-KV, a distributed in-memory key-value store system on a heterogeneous CPU–GPU cluster. Effectively utilizing the high memory bandwidth and latency hiding capability of GPUs, Mega-KV provides fast data accesses and significantly boosts overall performance and energy efficiency over the homogeneous CPU architectures. Mega-KV shows excellent scalability and processes up to 623-million key-value operations per second on a cluster installed with eight CPUs and eight GPUs, while delivering an efficiency of up to 299-thousand operations per Watt (KOPS/W).

Keywords Key-value store · GPU · Heterogeneous systems · Distributed systems · Energy efficiency

✉ Kai Zhang
kay21s@gmail.com

Kaibo Wang
kaibo@google.com

Yuan Yuan
yuanyu@cse.ohio-state.edu

Lei Guo
leguo@google.com

Rubao Li
li.961@osu.edu

Xiaodong Zhang
zhang@cse.ohio-state.edu

Bingsheng He
hebs@comp.nus.edu.sg

Jiayu Hu
humasama@mail.ustc.edu.cn

Bei Hua
bhua@ustc.edu.cn

1 Fudan University, Shanghai, China

2 Google Inc., Mountain View, CA, USA

3 The Ohio State University, Columbus, OH, USA

4 National University of Singapore, Singapore, Singapore

5 University of Science and Technology of China, Hefei, China

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 [50]. 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 [3], Redis [5], RAMCloud [43], and recently developed high-performance prototypes, such as Masstree [39] and MICA [36].

As a critical component in many Internet service systems, such as Facebook [42], YouTube, and Twitter, an

IMKV system is a distributed system which is deployed on a server cluster to provide key-value caching service to web servers. 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 and high scalability IMKV system is highly demanded. Moreover, because a large number of servers would be involved in deploying a distributed IMKV system, energy efficiency is also a major consideration in its design and implementation.

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. Each memory access generally takes 50–100 ns, while the average time budget for processing one query is only 10 ns for a 100-million operations per second (MOPS) system. However, memory access overhead is not able to be easily alleviated which significantly limits the throughput of IMKV systems. The main reasons are threefold. First, an IMKV system has a large working set [7]. As a result, the CPU cache would not help much to reduce the memory access latency due to its small capacity. Second, the supported number of concurrent memory accesses is small and depends on the CPU instruction window size and the number of Miss Status Holding Registers (MSHRs). For instance, the Intel X5550 CPU only supports 4–6 cache misses [19]. As a result, it is hard to utilize the inter-thread parallelism to overlap memory accesses on the CPU. Third, due to the strict data dependency, the IMKV thread is unable to proceed without the data from the memory, making the intra-thread parallelism unable to be utilized either.

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 homogeneous multicore CPU 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 mismatches the demand of the special data processing [15]; 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 capability of graphics processing units (GPUs).

In this paper, we propose Mega-KV, a distributed IMKV system on a heterogeneous CPU–GPU cluster. On each machine of the cluster, Mega-KV utilizes GPUs as the accelerator to offload and accelerate index operations. By effectively utilizing the high memory bandwidth and latency hiding capability of GPUs, the cluster shows high scalability and achieves significantly high throughput and high energy efficiency. Our technical contributions are fivefold:

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 proposed a real-time power management scheme for enhancing the energy efficiency of Mega-KV. By dynamically adjusting the frequency of the CPU and the GPU, Mega-KV is capable of achieving an efficiency of up to 299 KOPS/W.
5. We design and evaluate Mega-KV on a heterogeneous CPU–GPU cluster. Mega-KV shows near-linear scalability and achieves up to 623 MOPS throughput with eight commodity CPUs and GPUs.

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 energy management scheme of Mega-KV. Section 7 shows performance evaluations, Sect. 9 introduces related work, and Sect. 10 concludes the paper.

2 Background and motivation

2.1 An analysis of key-value store processing

In-memory key-value store system is generally implemented as a distributed system, where the key-value objects are partitioned among the IMKV nodes with consistent hashing [29]. As each IMKV node performs same operations and works independently with each other, we analyze and evaluate the query processing on a single IMKV node in this section.

2.1.1 *Workflow of a key-value store system*

A typical in-memory key-value store system generally provides three basic queries 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.

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 looked up 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 key-value item is removed from both the main memory and the index data structure. There are four major operations in the workflow of a key-value store system: (1) **Network processing**, including network I/O and protocol parsing. (2) **Memory management**, including memory allocation and item eviction. (3) **Index operations**, including Search, Insert, and Delete. (4) **Read key-value item in memory**: only for GET queries.

In recent works, a set of techniques have been proposed to improve the performance of CPU-based IMKVs, such as fast network processing [2,7,19], and accelerate concurrent data accesses [14,39]. In the following section, we will show that the performance gap between CPU and memory has become 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*

We have made the following observations on running IMKV on a single CPU node. More experimental setup can be found in Sect. 8.

Index operations are the major bottleneck of IMKV 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 [36]. 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* mode with a uniform key distribution. The following four data sets are used as the workloads: (1) 8-byte key and 8-byte value; (2) 16-byte key and 64-byte value; (3) 32-byte key and 512-byte value; (4) 128-byte key and 1024-byte value. This evaluation is conducted on a single IMKV node equipped with two Intel Xeon E5-2650v2 CPU and 8×8 GB memory. As shown in Fig. 1, index operations take about 75% of the processing time with the 8-byte 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 1024 bytes, and the index operation time portion decreases, but still takes about 50% of the processing time.

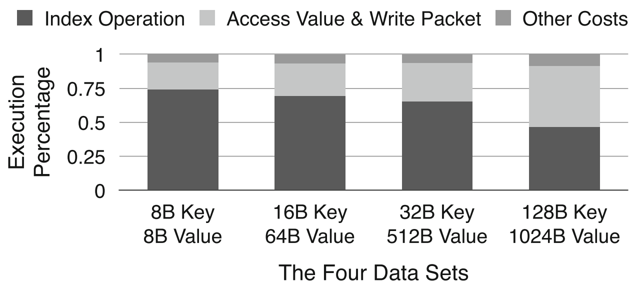

Fig. 1 Execution time breakdown of a GET query in a CPU-based key-value store system

With techniques including DPDK [2] and Multiget [3], the amortized packet I/O and parsing cost for a key can be as low as only 7 ns. MICA needs one or more memory accesses for its lossless 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 [33]. 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.

Random memory accesses dominate the performance of index operations 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 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 whose capacity is only 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 [39] needs $\log_4 n - 1$ random memory accesses, and a cuckoo hash table [44] 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 $O(\lg n / \lg \lg n)$ [11]. On the other hand, accessing the key-

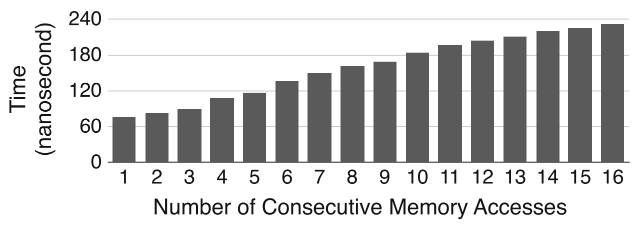

Fig. 2 Sequential memory access time

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.

The CPU node has prohibitively high cost of random memory accesses 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 (1024 bytes). Figure 2 shows the time of 1–16 sequential memory accesses. A random memory access takes about 76 ns in our machine, while 16 sequential memory accesses take 231 ns, only about three times higher than one access. In conclusion, the random memory accesses involved in accessing the index data structure may result in a huge 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 [19]. 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 cannot be explored either. (3) The working set of an IMKV system is very large [7]. 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 that 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 Single Instruction, Multiple Data (SIMD) fashion. With the massive array of ALUs, GPUs offer an order of magnitude higher computational throughput than CPUs for applications with ample parallelism. A 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 throughput than CPUs due to the following two features. First, GPUs have very high memory bandwidth. NVIDIA Tesla K40c, for example, provides 288 GB/s memory bandwidth, while the most recent Intel Xeon E7-8890 v4 processor has 85 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 [49].

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 utilized in an improper way.

Challenge 1: limited memory capacity and data transfer overhead The capacity of GPU memory is much smaller than that for main memory [51]. 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: trade-offs 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, trade-offs 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. Further-

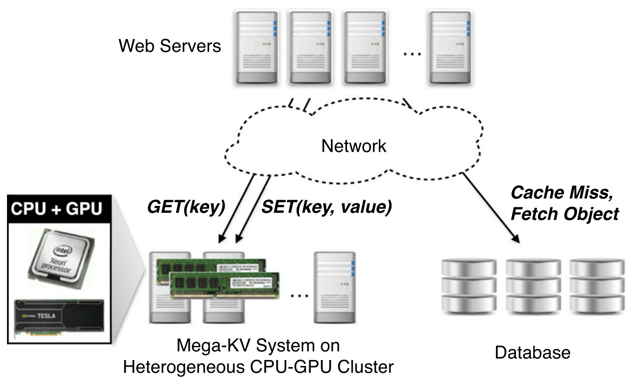

Fig. 3 Key-value store in Web service systems

more, 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.

Challenge 4: reducing energy consumption Energy efficiency is a major consideration in the design and implementation of IMKV systems on heterogeneous architectures. The workload of IMKVs changes radically during a day. In Facebook, for instance, the request rate of IMKVs at 16:00 can be two times higher than that at 8:00 [7]. If both CPUs and GPUs keep running at the highest frequency to meet the highest throughput, there will be a significant waste of energy in most time of a day.

3 Mega-KV: an overview

3.1 Mega-KV as a distributed system

Mega-KV is designed as a distributed in-memory key-value store system, which works in the same way as the general IMKV systems such as Memcached. As an example, Fig. 3 illustrates the role Mega-KV plays in Web service systems. A Web service system consists of Web servers, in-memory key-value stores, and databases. In this scenario, Mega-KV speeds up Web applications and alleviates database load by caching objects and data in the DRAM of each Mega-KV node, working in exactly the same way as Memcached.

In the system, the key-value space is partitioned among the Mega-KV nodes, where consistent hashing [29] is utilized by Web servers to decide the Mega-KV node that queries should be sent to. Each Mega-KV node works independently on processing queries to its assigned key-value objects. With consistent hashing, when a new node is added into the system or a node goes down due to power failure, only K/n keys need to be remapped, where K is the number of key-value objects and n is the number of IMKV nodes.

The workflow of the entire system is as follows. If a Web server wants to get the value of key *k1*, it first sends GET(*k1*) to the corresponding Mega-KV node. If the node has cached the key-value object, it sends the value back to the client, or it responses with a message, denoting that the requested value is not found. If the object is not cached in the node, the Web server fetches the value from the database (denote as *v1*); then, it sends a SET(*k1, v1*) to the corresponding Mega-KV node for future reuse.

3.2 Major design choices

On each node, Mega-KV enhances the throughput by adopting GPUs for acceleration. To address the memory access overhead, Mega-KV offloads the index data structure and its corresponding operations to GPUs.

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 [44] 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 predefined 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.

Improving energy efficiency with dynamic frequency scaling As there can be a performance disparity between the CPU and the GPU, Mega-KV improves its energy efficiency by reducing the energy consumption of the underutilized processor without degrading the overall throughput. To achieve this goal, Mega-KV adjusts the core frequency and the mem-

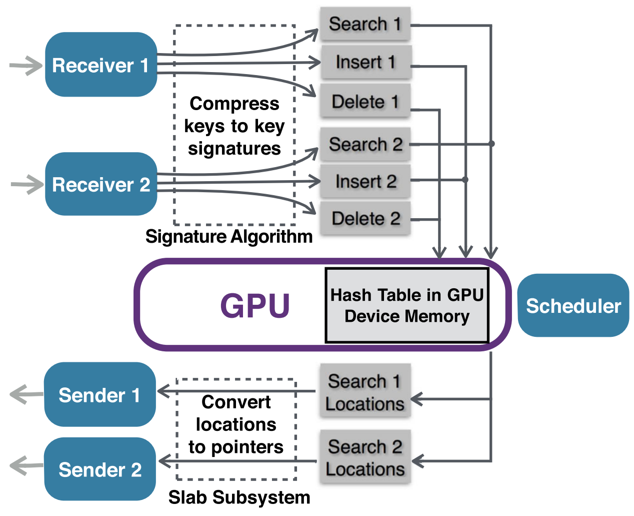

Fig. 4 Workflow of Mega-KV system

ory frequency of the underutilized processor, so that the processor has the lowest energy consumption while matching the throughput of the other processor.

3.3 The workflow of Mega-KV

Figure 4 shows the workflow of Mega-KV in a single node with CPU–GPU architectures. Mega-KV divides query processing into three stages: preprocessing, GPU processing, and post-processing, which are handled by three kinds of threads, respectively. *Receiver* threads are in charge of the preprocessing stage, which consists of packet parsing, memory allocation and eviction, and some extra work for batching. *Receivers* batch Search, 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*, and please refer to our previous paper [58] for details. After the key-value item *location IDs* for all the Search jobs are returned, *Sender* threads convert the *location IDs* to object pointers through the slab subsystem. Because the overhead from packet I/O, query parsing, and the memory management is still high, several *Receiver* and *Sender* threads are launched in pairs to form pipelines, while there is only one *Scheduler* per GPU.

In the following sections, we will describe the data structure and the algorithm of the GPU-optimized hash table and introduce the scheduling policy and the energy management scheme adopted by Mega-KV.

4 GPU-optimized cuckoo hash table

4.1 Data structure

For the IMKV workload characteristics and GPU architecture features, the GPU-based hash table for Mega-KV should take the following major design considerations for high efficiency. First, to utilize the massive number of cores, the hash table should avoid heavy synchronization operations between GPU threads. Second, since the GPU memory capacity is relatively small, the hash table data structure should be designed as compact as possible for indexing more key-value items. Third, as GET operations are latency sensitive and the number of key-value objects in an IMKV system can be quite large, the lookup performance of the hash table should be fixed and minimized.

CPU-based IMKV systems generally employ two general classes of index data structures, i.e., trees [39] and hash tables that resolve conflicts with chaining [3,36]. The time complexity of a lookup in trees [39] is $O(\log n)$. Moreover, after a number of Insert and Delete operations, the tree needs to be balanced to maintain its lookup performance. The overhead of balancing is extremely huge, which is unacceptable for IMKVs that demand predictable low latency. For hash tables with chaining to resolve conflicts, their lookup performance varies and depends on many factors, such as the load factor of the hash table and the stored position in the conflict data structure. Most importantly, the worst-case lookup performance is $O(n)$. Therefore, they are not competent to be implemented as the index data structure of Mega-KV.

Mega-KV adopts cuckoo hashing [44], which features a high load factor and a constant lookup time. Among different hashing methods, cuckoo hashing is reported to achieve the highest lookup performance with high load factors (90%) for general key distributions [48]. This matches the workload characteristic of IMKV systems. 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 [12]. 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

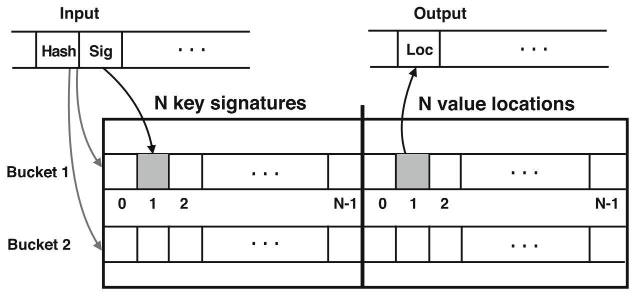

Fig. 5 GPU cuckoo hash table data structure

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 there are several items conflict in one bucket in the cuckoo hash table, a Search operation has to access every key-value object in the hash bucket to compare their keys, which overhead is extremely huge. The signature, which can be considered as another hash, represents the key. It avoids accessing keys stored in the host memory when multiple keys conflict in the same bucket. As shown in Fig. 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.

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 eight 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

To achieve high performance for hash table operations, 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. The threads forming a *processing unit* are selected within one warp; thus, they perform the operations simultaneously. After the *processing unit* performs the corresponding operations, the data are no longer needed and can be evicted from the cache. Based on this approach, this section describes the specific hash table operations optimized for GPU execution.

Search A Search operation checks all $2 \times N$ 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 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. There may be write–write conflicts if multiple *processing units* are trying to insert an item in the same position. 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 the signature is not inserted, the *processing unit* will try again. There will be at least one successful insertion for the conflict in a cell.

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.

To alleviate the write–write 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 eight 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.

For simplicity in system design, we limit that there cannot have two same signatures in one hash bucket. The Insert operation will evict the conflicted item by replacing the location ID. As a caching system, Mega-KV works in the same way of Memcached. The conflict key-value object in the host memory will be evicted by the eviction process after it is not accessed for a period of time. Moreover, the possibility of two keys having the same signature is only $1/2^{57}$ (32 bits signature and 25 bits hash bucket index), which is

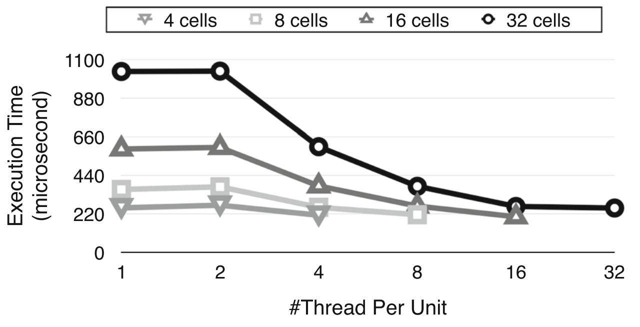

Fig. 6 Processing time of Search operation with 60-K jobs and 7 CUDA streams

extremely low. Therefore, this mechanism works in practical systems.

Delete Delete operation is almost the same with Search. When both the signature and *location ID* 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 Tesla K40c GPU in evaluating the hash table performance.

4.3.1 The choice of processing unit and bucket size

To evaluate the effectiveness of *processing unit*, Fig. 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 noncaching 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 eight cells for each bucket, which allows a higher load factor. Correspondingly, eight threads form a *processing unit*.

4.3.2 Optimization for Insert

An 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 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

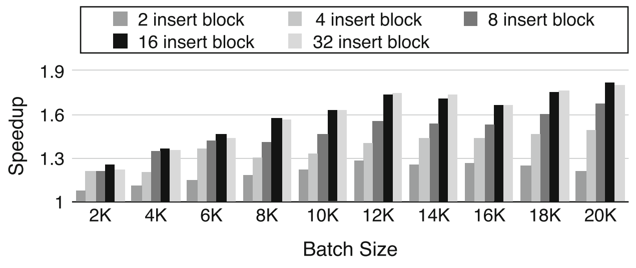

Fig. 7 Speedups for Insert with multiple blocks

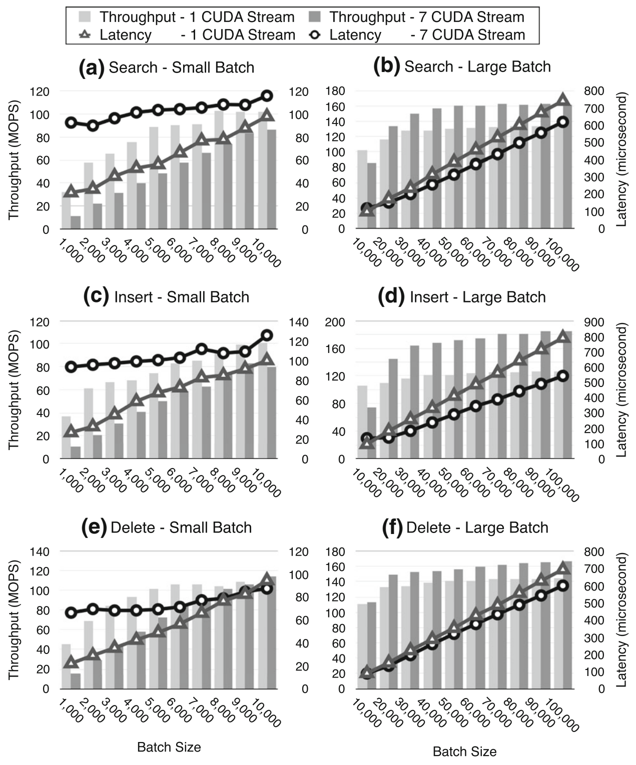

Fig. 8 GPU hash table performance

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 two 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 sizes. Comparing with 1 CUDA stream, a 24–60% performance improvement is achieved with 7 CUDA streams, which effectively 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, 210.3, 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 multiple CUDA streams.

With a large batch size, there is supposed to be more conflict for Insert and Delete operations. As shown in Fig. 8, the performance will not degrade with a large number of Insert/Delete operations in a batch. There are two main techniques in alleviating the conflicts in Mega-KV. First, when multiple keys are hashed to the same bucket, they have preferred slots to perform the operation. This significantly reduces the possibility of conflicting. Second, even if multiple operations conflict on the same slot, there will be at least one operation successfully performs the Insert operation in each round. Therefore, with the highly optimized hash table operations, the overhead of conflict is low.

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 [26], and queries may wait for tens of milliseconds to be scheduled in a data processing system [42]. 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 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 (Sect. 7.4).

4.5 The index size and GPU capacity

In Mega-KV, the index data structure should fit in the GPU memory. As the GPU memory capacity is relatively small comparing with the host memory, we have addressed this limitation in two aspects. (1) With the consideration on the limited GPU memory capacity, our highly optimized GPU index data structure is designed to index a high volume of memory space. Each entry in the hash table is compressed to be extremely compact, which only contains 64 bits, i.e., a 32-bit signature and a 32-bit location ID. In the workloads in Facebook, values close to 500 B take up nearly 80% of the entire cache's allocation for values [7]. With the keys and other information stored, a key-value object generally takes more than 600 B. Therefore, for the 12 GB memory of the K40c GPU in our evaluation, the index data structure is able to index nearly 1-TB objects stored in the host memory. We believe such a huge data volume is large enough for a single in-memory key-value store server. (2) In-memory key-value store system is a distributed system, which is able to achieve near-linear scalability with consistent hashing. Thus, we adopt a scale-out approach to support large index. When the key-value objects stored in a single node exceed the capacity of the host memory or their index exceeds the GPU memory capacity, parts of the key-value objects can be moved to other nodes. Therefore, Mega-KV is also able to address the capacity issue with scaling out. To conclude, the above two techniques effectively address the limitation of GPU memory capacity, making it not a concern in our system design and implementation.

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 [7]. Second, the processing time for Insert and Delete operations increases very slowly when the batch size is small. Third, SET queries have less strict latency requirement than GET queries.

In our scheduling policy, 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. Insert and Delete operations, however, are processed for every $n \cdot C$. 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 $2 \cdot C$. We define the sum of T_S and the time for Insert and Delete operations batched in $n \cdot C$ 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:

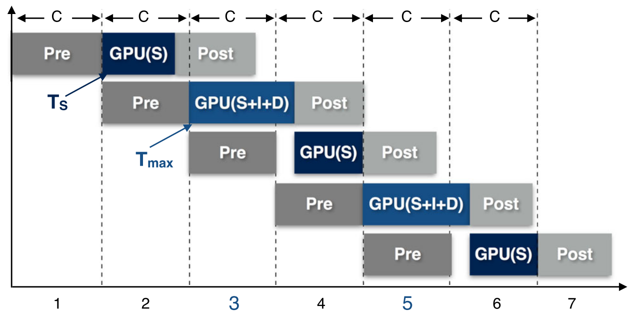

Fig. 9 Periodical delayed scheduling

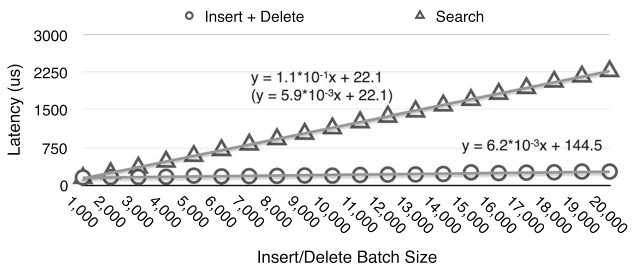

Fig. 10 Fitting lines of the performance of Search and Insert/Delete operation (95% GET, 5% SET)

$$T_S + T_{\max} \le 2 \cdot C \tag{1}$$

With the time for reading values and sending responses, a maximum response time of $3 \cdot C$ 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 Tesla K40c. 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 of x, the corresponding batch size for Search is $19x$. As shown in the figure, the processing time of Search operations increases more quickly than that of 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 in the batch size x_S, we define it as $T_S = k_S \cdot x_S + b_S$, where $k_S = 5.9 \times 10^{-3}$ and $b_S = 22.1$. 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 $k_I = 6.2 \times 10^{-3}$ and $b_I = 144.5$. 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 $T_S + T_{\max} \le 2 \cdot C$, we get

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

where $V < 2/(2 \cdot k_S \cdot p_S + n \cdot k_I \cdot (1 - p_S))$ and $n \ge 2$. 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 $C \ge T_{\max}$, and we get $C \ge (b_S + b_I)/(1 - k_S \cdot p_S \cdot V - k_I \cdot (1 - p_S) \cdot V)$, where $V < 1/(k_S \cdot p_S + k_I \cdot (1 - p_S))$. With the delayed scheduling, the lower bound of scheduling cycle C is reduced by an average of 43.4%. 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 Energy management scheme

Energy efficiency is critical for IMKV systems, as the energy consumption cost plays an important role in the total ownership cost in production systems. As Mega-KV adopts a pipelined execution model, the CPU and the GPU in the system are in charge of different pipeline stages and perform different operations in the query processing. The performance of the CPU and the GPU that can be achieved depends on the hardware architecture and the workload that is being processed by the system. Therefore, the throughput of the CPU and the GPU may not match with each other. However, as the overall system throughput depends on the pipeline stage with the lowest throughput, the processor with higher throughput would be underutilized. This will inevitably result in a low energy efficiency.

We propose an energy management scheme to improve the energy efficiency of Mega-KV. The first priority of the scheme is to meet the current throughput requirement. After that, our scheme sets the core and memory frequency to achieve the lowest energy consumption. Our energy management scheme is consisted of two steps. Firstly, we identify which processor is underutilized in the system. Secondly, we adopt Dynamic Voltage and Frequency Scaling (DVFS) to reduce its energy consumption without degrading the overall throughput. DVFS is a technique for reducing the energy consumption of processors by adjusting the voltage and frequency at runtime.

Step 1 We measure and compare the throughput of the CPU and the GPU. Because the throughput of the GPU is

Table 1 Available frequencies of NVIDIA K40c GPU

Memory (MHz)	3004				324
Core (MHz)	875	810	745	666	324

related to the GET:SET ratio of the workload and the scheduling cycle, its maximum throughput can be estimated with the same method as Eq. 2. The CPU performance, however, only depends on the workload, because system configurations such as scheduling cycle have little impact on it. We measure the CPU performance in real time. Given a system latency requirement, the maximum GPU throughput and the CPU throughput can be known to identify which processor is the bottleneck and which is underutilized.

Step 2 We reduce the energy consumption of the processors without compromising the overall throughput. In Particular, we have the following two cases:

1. If the CPU throughput is higher than the GPU throughput, we adjust the CPU frequency gradually with *CPUFreq* [1]. Our mechanism tries to decrease the CPU frequency by 100 MHz each time until the following condition is met.

$$T_C - T_G > \mu \cdot T_G, \tag{3}$$

where T_C is the CPU throughput, T_G is the GPU throughput, and μ is a parameter as the threshold of the throughput difference. In the implementation of Mega-KV, we set μ to 0.1 as a balance between improving the energy efficiency and guaranteeing the overall throughput.

2. If the GPU throughput is higher than the CPU throughput, we decrease the GPU frequency to reduce energy consumption. Current GPUs are able to set both the core frequency and its device memory frequency. However, different with CPUs, only specific core and memory frequency combinations are allowed to be set in GPUs. For instance, we list the five available combinations for NVIDIA K40c GPU in Table 1.
We estimate the GPU performance for all frequency combinations with the same method in Sect. 5 and get the parameters of Eq. 2 for each frequency configuration (shown in Table 2). With a scheduling cycle, we are able to calculate the maximum throughput for n frequency configurations as $\mathbb{T} = \{T_1, T_2, \ldots, T_n\}$. To reduce energy consumption without compromising the overall throughput, we choose the frequency configuration with $\min\{T \mid T > T_C, T \in \mathbb{T}\}$, where T_C is the CPU throughput.

When the workload changes in the runtime, the current frequency setting policy may result in a suboptimal overall throughput. To dynamically adapt to the changing workload, the energy management scheme periodically measures the CPU performance at runtime. Because the CPU performance is sensitive to the workload, a variation in the CPU performance (5% in Mega-KV) means that the workload is experiencing a change. If this happens, we set both the CPU and the GPU to their highest frequencies and then perform the above two steps again to adjust the energy consumption while delivering the highest throughput. In Mega-KV, we set the cycle of measuring the CPU performance as 10 seconds.

In the implementation, we use NVIDIA Management Library (NVML) [4] to set the core frequency and the memory frequency of the GPU. Functions *nvmlDeviceGetSupportedMemoryClocks* and *nvmlDeviceGetSupportedGraphicsClocks* are used to get the available frequency combinations in the GPU, and function *nvmlDeviceSetApplicationsClocks* is used to set the frequencies.

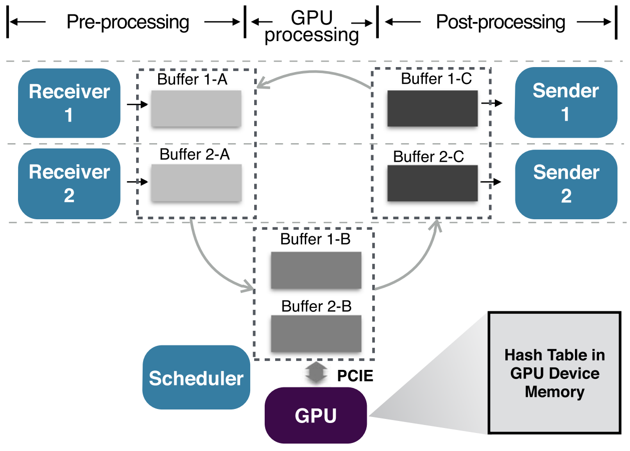

Fig. 11 Framework of Mega-KV system

7 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.

7.1 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 are transferred by passing the buffers. Figure 11 shows the zero-copy pipelining

Table 2 Performance parameters for each frequency combination

M.	3004				324
C.	875	810	745	666	324
k_S	5.9×10^{-3}	6.4×10^{-3}	6.9×10^{-3}	7.8×10^{-3}	1.5×10^{-2}
b_S	22.1	18.0	23.2	25.3	176.9
k_I	6.2×10^{-3}	6.9×10^{-3}	7.6×10^{-3}	9.2×10^{-3}	3.6×10^{-2}
b_I	144.5	147.5	152.3	154.0	282.5

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. The framework is similar with our previously built real-time SRTP reverse proxy [57]. With this technique, the overhead of data transferring between pipeline stages is significantly mitigated.

7.2 Memory management

Slab memory management Mega-KV uses slab allocation. The *location ID* 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 ID* and the corresponding item, where the highest bits in the *location ID* 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 ID* 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 ID* is replaced with the new one (Sect. 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 are 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.

7.3 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.

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

7.4 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 versus 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 during the batching process. 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.

8 Experiments

In this section, we evaluate the performance and latency of Mega-KV under a variety of workloads and configurations.

8.1 Experimental methodology

Hardware We conduct the experiments on a cluster with four CPU–GPU servers as Mega-KV nodes and four CPU servers as clients. Each Mega-KV server is 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 1600 MHz DDR3 memory and supports a maximum of 59.7 GB/s memory bandwidth. Each socket is installed with a NVIDIA Tesla K40c GPU. Tesla K40c has 15 streaming multiprocessors and a total of 2880 cores. The device memory on each GPU is 12 GB GDDR5, and the maximum data transfer rates between main memory and device memory are 10.3 GB/s (host to device) and 10.3 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 XL710 dual port 40 GbE card, and the open-source DPDK [2] is used as the driver for high-speed I/O.

In the time we buy the server, the price of the Intel Xeon E5-2650 v2 CPU is $1162. We have evaluated Mega-KV with the NVIDIA GTX 780 GPUs ($500) in our previous work [58]. As the consumer-class GTX GPUs do not support NVML library to adjust the GPU frequency, we use the server-class NVIDIA Tesla K40c to enable our energy management scheme. The price of NVIDIA Tesla K40c GPU is $2150 when we bought it. In the following experiments, we will report the performance evaluated with the NVIDIA Tesla K40c GPU.

Workloads We use four data sets in the evaluation: (*a*) 8-byte key and 8-byte value; (*b*) 16-byte key and 64-byte value; (*c*) 32-byte key and 512-byte value; and (*d*) 128-byte key and 1024-byte 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 Mega-KV batches responses in an Ethernet frame as much as possible.

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 [10]. Our clients use approximation Zipf distribution generation described in [17] for fast workload generation.

Cluster configuration We form the eight servers as a cluster to evaluate the throughput of Mega-KV as a distributed system. Since Mega-KV can support very high throughput, supporting the network traffic between clients and IMKV nodes needs a switch supporting hundreds of gigabits traffic, which is prohibitively high cost. We cannot afford such high cost switches as in major data centers. Instead, we use half of the servers to simulate the generation of traffic from the switch and the other half of the servers serving the requests. We simulate the environment with the following method. In the evaluation of each workload, we generate 10-billion key-value queries and partition them into four parts with consistent hashing. In this way, the queries stored in each partition are to be sent to the corresponding Mega-KV node. We store the partitions on the four clients and connect the clients to their corresponding Mega-KV nodes to feed the queries.

For all the workloads, the working set size of each Mega-KV node is 64 GB, and a 2-GB GPU hash table is used to index the key-value objects on each GPU. To perform GET operations, the key-value objects in the Mega-KV nodes are preloaded in the evaluation. The hit ratio of GET queries is higher than 99.9%. The experiments for small key-value objects are performed with network processing, where Mega-KV receives requests from clients and sends back responses through the NIC ports. For the large key-value objects workloads where the network becomes the bottleneck in the evaluation, key-value objects are stored locally to evaluate the performance of Mega-KV.

Single node configuration in Mega-KV 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 seven physical cores with one *Receiver* and one *Sender* thread, forming a pipeline on the same physical core. Therefore, there are seven 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.

APIs: get and getk The same as Memcached, Mega-KV supports 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. A *getk* query asks a key-value store to send the key with the value to the client, and the client is implemented to be capable of matching the key with its value. Therefore, Mega-KV does not compare the keys to confirm the match and requires its client to do the job.

Comparison with a CPU-based IMKV system We take the open-source in-memory key-value store MICA [36] as the state-of-the-art system for comparison. MICA is one of the fastest CPU-based implementations. MICA partitions key-value objects among multiple cores and uses client-assisted information to directly directing queries to their corresponding cores with a NIC feature called Receive-Side Scaling (RSS). MICA adopts a circular log to store key-value objects. Its index data structure is a hash table where items are also compressed into fixed length and stored sequentially, where each hash item includes a tag and the key-value object offset in the log. The tag is like the signature in Mega-KV, which is used for reducing extra memory accesses in the index operations. Moreover, it aggressively adopts prefetch instructions to prefetch data such as the key-value objects. According to MICA's experiment configuration, the hyper-threading is disabled. We have tried to enable hyper-threading in MICA, but the performance drops, as also reported in [35]. 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 Sect. 2.1.2.

8.2 Theoretical maximum system throughput of a single node

As discussed in Sect. 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.

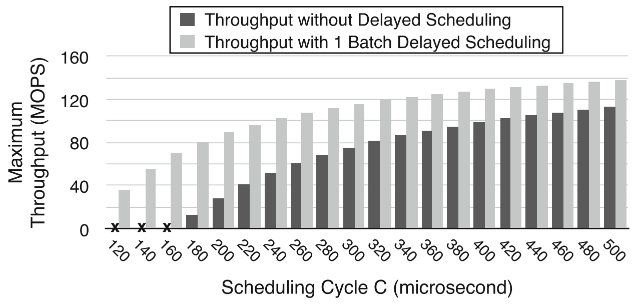

Fig. 12 Theoretical maximum speed for one NVIDIA Tesla K40c GPU

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

$$V \leq \frac{2 \cdot C - 2 \cdot b_S - b_I}{2 \cdot k_S \cdot p_S \cdot C + 2 \cdot k_I \cdot (1 - p_S) \cdot C} \tag{4}$$

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 $C > 94.4$ with the delayed scheduling and $C > 166.6$ without the delayed scheduling.

With the same parameters k_S, b_S, k_I, and b_I listed in Sect. 5.2, Fig. 12 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 a single Mega-KV node 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 17.8–68.4% with delayed Insert and Delete scheduling. It is worth noting that for $C \leq 166.6\,\mu$s, 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 $94.4\,\mu$s.

8.3 Throughput of a single Mega-KV node

Figure 13 compares the throughput of Mega-KV and MICA in a single node for all the workloads. For comparison, we measure the performance of CPU-based MICA as the baseline. Mega-KV outperforms MICA for all the workloads. The throughput of Mega-KV is 1.3–2.9 times as high as MICA for the 95% GET 5% SET workloads and is 1.3–2.6 times as high as MICA for the 100% GET workloads. The throughput improvement in Mega-KV over MICA is achieved for two main reasons. First, the random memory accesses in

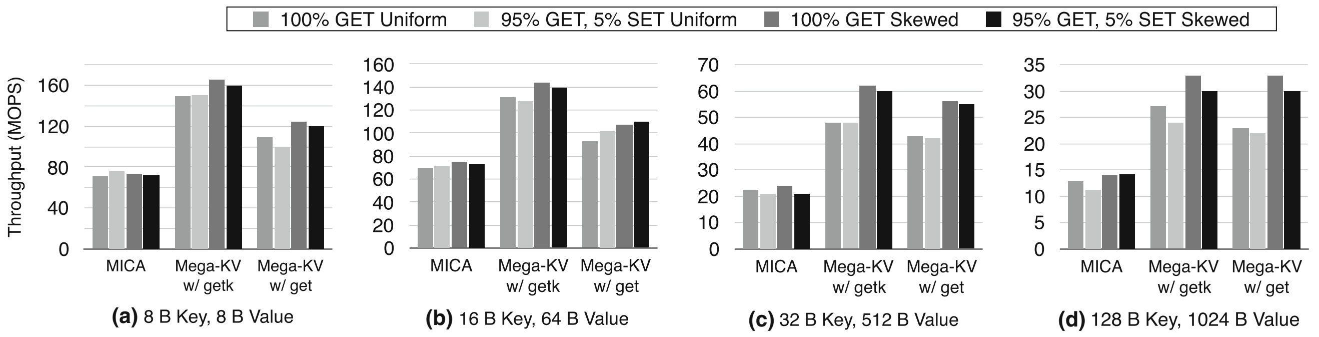

Fig. 13 Single node throughput comparison between Mega-KV and MICA

Table 3 Single node loading throughput of Mega-KV (100% SET)

Workload	8B k, 8B v	16B k, 64B v	32B k, 512B v	128B k, 1024B v
Thr.(MOPS)	171	149	55	32

index operations are offloaded to GPUs. Second, network processing and memory accessing are effectively overlapped with hyper-threading on each CPU core, which significantly improves the CPU utilization.

As depicted in the figure, the throughput improvement achieved by Mega-KV for skewed workloads is higher than that of the uniform workloads. This is because Mega-KV adopts a shared memory design where each CPU core is able to process queries for the entire key-value space cached in it. In contrast, MICA partitions the key-value space among CPU cores, where each core can only access its own partition exclusively. As a result, MICA's system design suffers load imbalance with the skewed workloads, and Mega-KV achieves higher throughput improvement by overcoming this drawback.

Table 3 shows the loading performance of Mega-KV on a single node, i.e., 100% SET queries. We can see from the table that the performance is high and varies with different key-value sizes. The hash table operations in the GPUs are not the bottleneck of the system. Instead, for write-dominant workloads, the main overhead is from the LRU eviction and writing key-value objects in the preprocessing stage. Therefore, the performance is highly relevant with the key-value size.

Without the GPUs, the price of the hardware in the evaluation is $4837 when the server is purchased in 2014. With the consumer-class NVIDIA GTX 780 GPUs ($500 each) used in our VLDB'15 paper [58], the price–performance ratio of Mega-KV is 7.7–198.3% higher than that of MICA. The NVIDIA Tesla K40c GPUs are used here as it supports the NVIDIA Management Library (NVML) for energy management. The GPUs have compatible performance, but the Tesla K40c GPU is designed to be small and without active cooling system (fan). Moreover, Tesla GPUs have many advanced features including ECC (error checking and correction) memory and setting core/memory clocks. Correspondingly, the price of the GPU is higher ($2150 when we bought it). With Tesla K40c GPUs, the price of hardware for Mega-KV is 88.9% higher than that of MICA. The price–performance ratio of Mega-KV with Tesla K40c is 0.3–52.2% higher than that of MICA with *getk* queries for all workloads besides the 16B key workload with uniform 95% GET queries. With *get* queries, the price–performance ratio of Mega-KV is higher than MICA for 32B key and 128B key workloads (up to 39.5%) and is 10.0–30.7% lower for 8B key and 16B key workloads.

8.4 Throughput of the Mega-KV cluster

Figure 14 shows the throughput of Mega-KV cluster for both *getk* query and *get* query. As shown in the figure, the throughput for small key-value sizes workloads is higher than that for large key-value sizes, and Mega-KV achieves the highest throughput for the data set with 8-byte key and 8-byte value. With the *getk* query, Mega-KV achieves the maximum throughput of 605 MOPS for the 95% GET 5% SET workload and 623 MOPS for the 100% GET workload. With the *get* query, the throughput of Mega-KV is 480 MOPS (95% GET and 5% SET) and 496 MOPS (100% GET). In a real-world scenario where a mix of *get* and *getk* queries are sent by clients, the throughput of Mega-KV should lie between *get* and *getk*'s throughput.

In the pipeline of Mega-KV, we find that the throughput of GPUs is much higher than that of CPUs. After the expensive index operations are offloaded to GPUs, the memory accesses to key-value objects become the major factor that limits the system performance. As a result, the key comparison operation involved in the *get* query processing degrades system throughput by 20–30% with the small size key-value workload. This is because adding extra overhead (key comparison) to the bottleneck pipeline stage would correspondingly degrade the overall throughput. With a larger

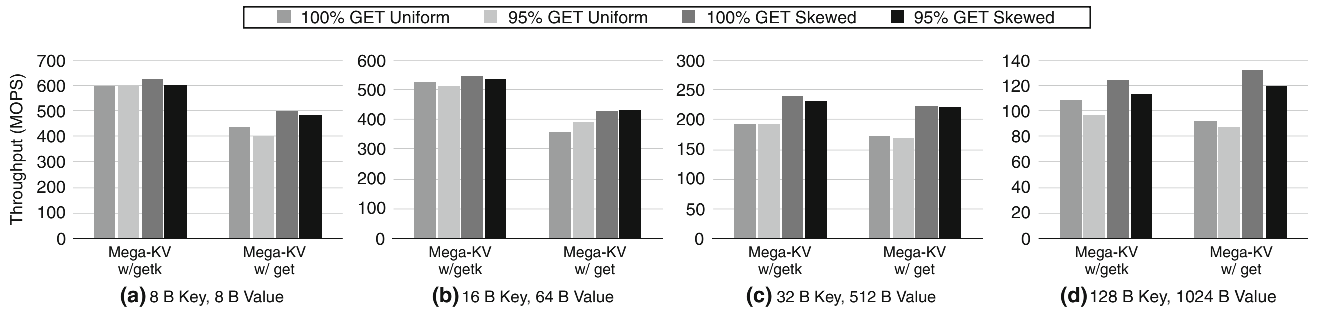

Fig. 14 Throughput of the Mega-KV cluster

key-value size, however, 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 the uniform workload, the throughput is 0.8–24.4% lower than the skewed workloads. The main reasons are twofold. First, our system applies a shared memory design within each NUMA node, and queries for a hot key can be processed by multiple cores simultaneously. Thus, the skewed workloads will not result in an imbalanced load among the CPU cores. Second, as reading the key-value objects in memory in the CPU is the major bottleneck of Mega-KV, the most frequently visited key-value objects in skewed workloads may be cached in the CPU cache. Consequently, a higher throughput is achieved by alleviating the CPU overhead in accessing the memory.

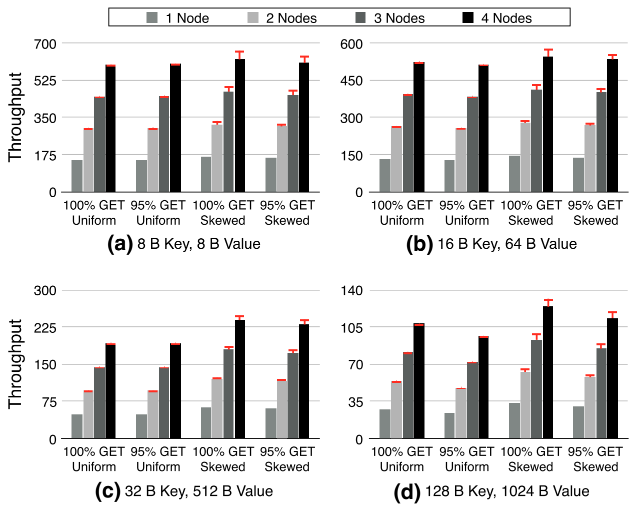

Fig. 15 Scalability of the Mega-KV cluster

8.5 Scalability

In production systems, there are generally a set of IMKV nodes to increase the caching capacity and the overall throughput to meet the demand of production systems. Therefore, scalability is critical for IMKV systems. We evaluate the scalability of the Mega-KV cluster in Fig. 15 with *getk* queries. We use the upper end of error bars to show the throughput when achieving linear scalability. Mega-KV cluster achieves near-linear scalability for workloads with uniform key distributions. This is because key-value stores have the nature to scale out, where the main reasons are twofold. (i) Each IMKV node stores a partition of the key-value store objects in the system separately. (ii) There is little communication and collaboration between different nodes; thus, they work independently. With consistent hashing [29], clients directly send queries to the corresponding IMKV node. Therefore, for workloads with uniform key distribution, key-value store systems are easy to achieve near-linear scalability.

For workloads with skewed key distribution, the overall throughput can be severely influenced by the imbalanced load among nodes. This is a common issue for CPU-based IMKV systems [34], such as Memcached [42]. Unlike CPU-based IMKVs, the skewed workloads lead to only minor load imbalance in Mega-KV. As shown in the figure, the throughput of Mega-KV is only around 7% lower than the ideal throughput. The reasons are threefold. First, instead of partitioning key-value objects among multiple CPU cores, Mega-KV adopts a shared memory design in each node. With the RSS hardware NIC feature, queries are distributed to CPU cores based by hashing the packet header 5 tuples. As hot keys are generally from different clients, hot keys exert little impact on the load of each CPU core. Second, in the GPU kernel, GPU cores are evenly assigned with the same number of index operations. Therefore, hot keys would also not influence the throughput of the GPU. Third, the throughput of a Mega-KV node is hundreds of times higher than that of a CPU-based Memcached node. The number of key-value objects cached by a Mega-KV node can be orders of magnitudes more than a Memcached node. Therefore, there is a larger possibility that each Mega-KV node contains similar number of hot and not-hot keys.

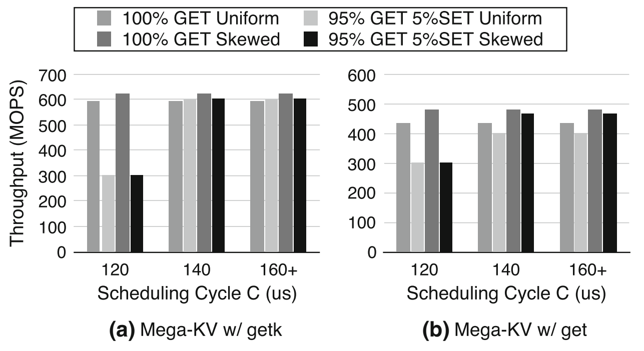

Fig. 16 Throughput of the Mega-KV cluster with different scheduling cycles (8B key and 8B value)

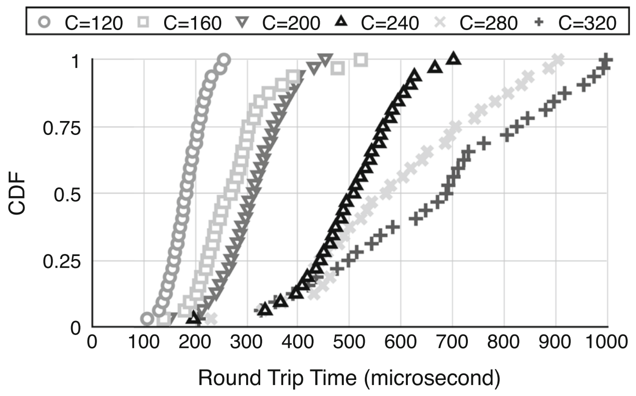

Fig. 17 System latency (95% GET 5% SET, 8B key and 8B value, skewed key distribution)

8.6 System throughput with different scheduling cycles

With our periodical scheduling policy in Sect. 5, we are able to effectively control the scheduling cycle in Mega-KV to achieve a controllable latency. Because of the nature of batch processing in GPUs, a high throughput will compromise the system latency. The reason is that although a larger batch will lead to a higher throughput for more efficiently utilizing the GPU and the PCIe bus, batching more queries generally demands more time. In this subsection, we evaluate the maximum throughput that can be achieved in Mega-KV with different scheduling cycles.

Figure 16 demonstrates the throughput of Mega-KV with scheduling cycles of 120, 140, 160 µs, and higher ones. The workloads adopted in the evaluation are with 8-byte key and 8-byte value. For 95% GET 5% SET workloads, the throughput of Mega-KV cluster increases by increasing the scheduling cycle. Under 160 µs, the throughput of the GPU is lower than that of the CPU. In the figure, the overall system throughput with $C = 120$ and $C = 140$ also denotes the throughput of the GPU. Therefore, increasing the scheduling cycle is capable of achieving a higher overall system throughput by improving the GPU throughput. The throughput of the CPU, however, is constant with different scheduling cycles, and the throughput of the CPU and the GPU becomes the same with the 160 µs scheduling cycle. As a result, the system throughput stops to increase with higher scheduling cycles as the CPU has become the bottleneck in the system. Therefore, the system throughput with $C = 160+$ also shows the throughput of the CPU.

For the 100% GET workloads, however, the throughput of the GPU is much higher than that for 95% GET workloads, as it does not have to perform the small number of Insert and Delete operations. And the throughput of the GPU is higher than of the CPU even with a scheduling cycle of 120 µs. Therefore, the system throughput is limited by the CPU and does not vary with the scheduling cycle.

8.7 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 latency 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 17 shows the CDF (Cumulative Distribution Function) of query round trip latency with different scheduling cycles. We fix the scheduling cycle C as 120, 160, 200, 240, 280, and 320 µs, respectively and use the allowed maximum input speed for each scheduling cycle. As can be seen from the figure that the latency of query processing in Mega-KV can be adjusted with the scheduling cycle. With a fixed scheduling cycle, the round trip latency of query varies in a range. For instance, the latency ranges from 300 microseconds to 700 µs with the scheduling cycle set as 200 µs. This is because the queries arrive in different periods in the batching. In the evaluation, the round trip latency of queries is effectively controlled within $3 \cdot C$, which demonstrates the effectiveness of our GPU scheduling policy in achieving predictable latency.

Figure 17 shows the latency of Mega-KV with 95% GET 5% SET workloads. We have also measured the latency of 100% GET workloads, which is almost the same with that of the 95% GET workloads. Batching is the main factor that affects the overall round trip latency. In all the three stages, each query needs to wait for the previous queries in the same batch to complete processing. In our evaluation, different workloads only lead to at most 20 µs difference in the average latency.

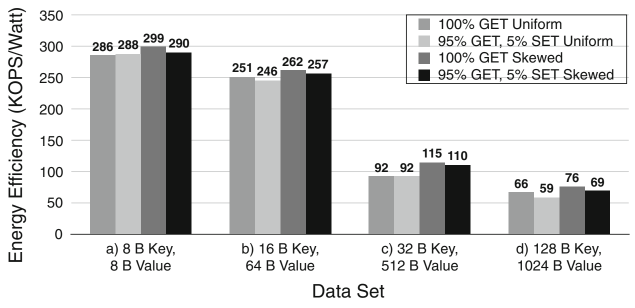

Fig. 18 Energy efficiency of the Mega-KV cluster with the energy-saving technique

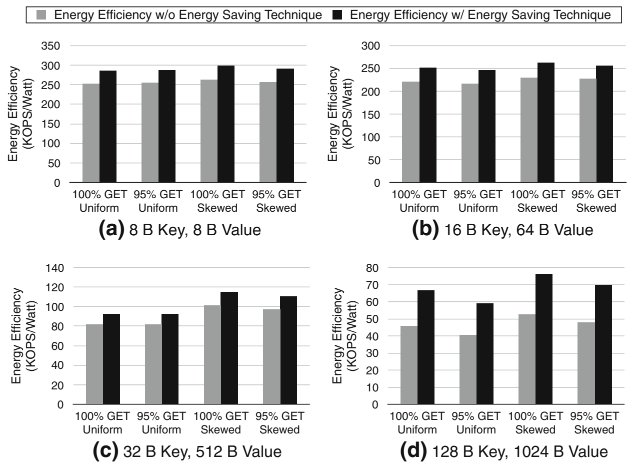

Fig. 19 Energy efficiency improvement with the energy-saving technique

The latency of Mega-KV is comparatively higher than that of a fast CPU-based implementation. According to the input query speed, the latency of CPU-based implementation MICA ranges from 24 to 52 µs. The Mega-KV cluster achieves a maximum throughput of 639 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 and 410 µs (Fig. 17). However, in Facebook, the average and 95th percentile latencies of Web servers in production getting keys are about 300 and 1200 µs, respectively [42]. 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.

8.8 Energy efficiency

In this subsection, we first measure the energy efficiency of our Mega-KV cluster. Then, we compare it with two recent IMKV systems that are designed for high energy efficiency. Following the previous studies [24], we use the number of queries processed per Watt as the metric for the energy efficiency in the comparison.

8.8.1 Energy efficiency of Mega-KV cluster

We measure the power consumption of our cluster to evaluate its energy efficiency. As shown in Fig. 18, Mega-KV achieves an energy efficiency of up to 299 KOPS/W for the small key-value data set. For the large key-value data sets, the CPU spends more time in reading and processing key-value objects. Therefore, the energy efficiency of Mega-KV ranges from 59 KOPS/W to 76 KOPS/W for the data sets with 128-byte key and 1024-byte value.

We show the improvement in energy efficiency brought by our energy-saving technique in Fig. 19. Overall, our energy-saving technique (introduced in Sect. 6) helps to save 10.2–31.1% of the total power consumption. The improvement decreases when the key-value size gets larger, but increases for the workloads with 128-byte key. For all workloads with key-value data sets *a*, *b*, and *c*, the memory frequency is set to be 3003 MHz and the core frequency is set as 666 MHz, and the GPU is able to meet all the throughput requirement for these workloads. Without the energy-saving scheme, the power consumption of the Tesla K40c GPU when executing the kernels of index operations can be up to 150 Watts. With the proposed scheme, power consumption drops to 135 Watts while preserving the throughput. The improvement in energy efficiency for large key-value data sets is less than that for small key-value data sets. The reasons are twofold. First, the throughput of Mega-KV in processing large key-value data sets is around three times lower. Second, the reduced power consumption for the workloads is almost the same (around 120 Watts for the cluster).

For the 128-byte key and 1024-byte value data set, GPU's memory frequency and core frequency are both set to be 324 MHz by the energy-saving scheme. With this frequency setting, the throughput of the index operations on one GPU is 22 MOPS, which is enough for matching the CPU throughput for the workloads. The improved energy efficiency thus becomes higher, as the energy consumption of a GPU is only 58 Watts. In summary, the energy-saving technique chooses two core frequency settings for the GPUs, i.e., 666 and 324 MHz, which saves around 240 W and 736 W power in total for the Mega-KV cluster, respectively. Most importantly, the overall system throughput is not compromised under the energy management scheme.

8.8.2 Comparison with CPU-based IMKV systems

Figure 20 compares the energy efficiency of MICA and Mega-KV. According to the workloads, the energy efficiency of Mega-KV is 0.17–57.87% higher than that of MICA. This has shown the effectiveness and efficiency of our system design on heterogeneous hardware. The main reasons

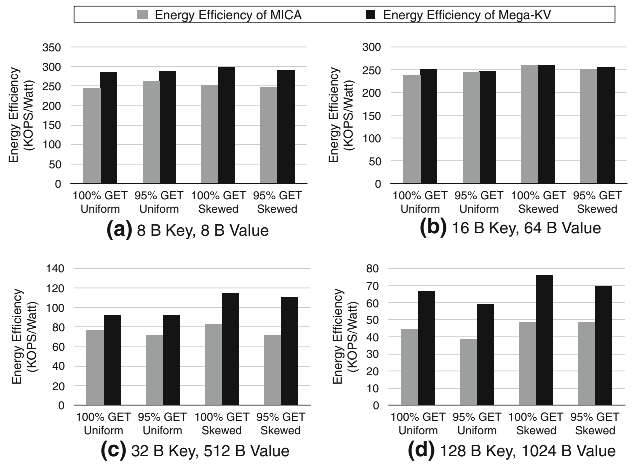

Fig. 20 Energy efficiency comparison with MICA

that Mega-KV achieves higher energy efficiency are twofold. First, although GPUs have higher power consumption than CPUs, a GPU contains thousands of cores and the power consumption of each core is much lower. For instance, the NVIDIA Tesla K40c GPU used by Mega-KV contains 2880 cores with a maximum power consumption of 215 W; therefore, each core only consumes at most 0.075 W. With our proposed energy-saving technique, a GPU core consumes 0.054 W for workloads with data sets *a*, *b*, and *c*. For workloads with data set *d*, the power consumption of a GPU core is further reduced to 0.011 W. Second, we carefully partition the query processing among CPUs and GPUs, where both CPUs and GPUs are working on tasks they are good at. This significantly enhances the overall system efficiency as the index operations handled by GPUs have sufficient parallelism, while the rest operations that have fewer random memory accesses and more branches in the execution path are fit for CPU processing.

9 Related work

GPU-based key-value store Concurrent with our previous work [58], MemcachedGPU [24] also uses GPUs to accelerate in-memory key-value stores. In contrast to Mega-KV, MemcachedGPU adopts the GPUDirect feature to directly DMA transfer packets to the GPU memory. Correspondingly, MemcachedGPU performs the network processing in the GPU. Due to the limited GPU memory space, both Mega-KV and MemcachedGPU store the key-value objects in host memory; thus, reading the objects is performed by CPUs.

The coupled CPU–GPU architecture integrates a CPU and a GPU in the same chip, which are able to share the host memory with recent hardware advancement. Different with discrete CPU–GPU architectures, the coupled architectures not only eliminate the overhead of transferring data between the CPU and the GPU, but also reduce the synchronization overhead significantly. By utilizing the architecture, DIDO [56] dynamically changes its pipeline partition between the CPU and the GPU according to the workload. DIDO significantly improves the overall IMKV system efficiency on coupled CPU–GPU architectures. Hetherington et al. [23] port the existing Memcached implementation to an OpenCL version, but the implementation does not explore APUs' hardware characteristics for higher performance. As the compute capability of coupled CPU–GPU architectures is still much weaker than discrete architectures, the throughput of IMKVs on coupled architectures is generally lower than that on discrete architectures.

CPU-based key-value store CPU-based in-memory key-value stores [14,39–41] have been focusing on designing efficient index data structure and optimizing network processing to achieve higher performance. MICA [36] has compared itself with RAMCloud [43], MemC3 [14], Memcached [3], and Masstree [39] in its paper and shown an at least four times higher throughput than the next best system. That is why we choose MICA for performance comparison. Systems such as Chronos [28], Pilaf [41], and RAMCloud focus on low latency processing, which achieve latencies of less than 100 μs. Specifically, by taking advantage of Infiniband, RAMCloud and Pilaf achieve average latencies of as low as 5 and 11.3 μs, respectively.

There have been previous studies that improve the data locality in index operations by batching requests [59]. However, they are not applicable to CPU-based key-value stores for the following main reasons. First, as batching itself may add additional overhead to degrade the overall performance, the system [59] needs to wait for seconds to batch large number of operations in improving the overall performance. Consequently, the latency of Search operations can be higher than 3 seconds [59]. As in-memory key-value stores demand fast response, the processing latency of key-value queries should be controlled within 1000 microseconds. Therefore, the approach of batching request is unable to meet the latency requirement of IMKVs. Second, the technique is designed for trees, while in-memory key-value store systems generally adopt hash tables as their index data structures. This is because frequent Insert and Delete operations in a tree will lead to the imbalance of the tree, while re-balancing the tree has an extremely high cost. This is unacceptable for key-value store systems where latency-sensitive queries are arriving continuously. Moreover, the time complexity of a lookup in trees is $O(\log n)$, where n is the number of keys in the IMKV, instead of $O(1)$ in hash tables. This makes trees unsuitable for read-intensive workloads in most in-memory key-value store systems. Third, the technique is hard to be applied to hash tables. Trees are different from hash tables, where different queries can have common paths on the top levels of the tree. Instead, the access pattern to hash tables is random. Considering that the object indexes are distributed in different parts of a hash table that contains millions of items, requests are unlikely to be effectively batched to improve the data locality. Therefore, there is little chance to achieve performance gain from batching hash table operations.

Distributed key-value store system There are lots of research conducted on distributed key-value store systems, including providing new functionality and improving the distributed software architecture. Comet [16] allows each key-value object to take dynamic, application-specific actions to customize its behavior. Its application handlers are written in a sandboxed extension language, providing properties of safety and isolation. Besides retrieving key-value objects with the primary keys, HyperDex [13] is a distributed key-value store that provides a search primitive to enable queries on secondary attributes. RAMCloud [43] is a distributed in-memory key-value store that keeps all the data in DRAM, disk relegated to a backup/archival role. Besides being used as a cache, RAMCloud offers a high level of durability and available via data replication.

Energy efficient key-value store Key-value stores include FAWN [6], TSSP [37], Mercury [18], and Tilera and FPGA-based Memcached [8,9,30] are built to be energy efficient. These designs address the inefficiency of general-purpose CPU in key-value store implementation and propose to use specific and low-power hardware. Mega-KV achieves high energy efficiency with general-purpose CPUs and GPUs for two main reasons. First, the massive number of cores and high memory bandwidth of GPUs make it a better candidate for key-value query processing. Second, our frequency scaling technique helps Mega-KV to further reduce energy consumption.

Energy efficiency of GPUs Lots of research are conducted on improving the energy efficiency of GPUs, including power models and energy management frameworks. GPUWattch [32] proposes a configurable power model to enable energy optimizations in GPUs, while Hong et al. [25] proposes an integrated power and performance prediction model for a GPU architecture to predict the optimal number of active processors for an application. Lee et al. [31] demonstrate that, under a power constraint, reducing the frequency of the GPU cores allows the GPU to utilize more cores. The technique is able to improve the throughput of applications with the same energy consumption. GreenGPU [38] develops a two-tier energy management framework for CPU–GPU heterogeneous systems. It first splits and distributes workloads among the CPU and the GPU; then, a lightweight machine learning algorithm is developed to adjust the frequencies of the GPU. GreenGPU is a general framework for applications with run-to-completion model, but Mega-KV adopts a pipelined CPU–GPU model where the workload for the GPU is fixed to performing index operations. Another our work DIDO [56] builds an in-memory key-value store system on coupled CPU–GPU architecture, which shows improved energy efficiency over discrete architectures.

The method of adjusting both the voltage and frequency in the paper of Price et al. [47] is an interesting approach. As the method needs to modify the GPU firmware to set the voltage, it cannot be applied at runtime. With dynamically changing workloads, the goal of our energy management scheme is to adjust the energy consumption accordingly at runtime. Therefore, this existing approach cannot be applied in key-value store systems.

GPU-based database system There are already a set of research papers on adopting GPUs in database systems [20–22,27,45,46,52–55]. 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 [54]. Therefore, Mega-KV's GPU-optimized cuckoo hash table with its corresponding operations is a good candidate for accelerating the hash join operation.

This paper extends our previous conference paper [58], where Mega-KV is implemented and evaluated on a single node. This paper builds Mega-KV as a distributed system and evaluates its scalability, performance, and energy efficiency in a cluster setting. Our energy management scheme further enhances the energy efficiency of Mega-KV. By adjusting the frequencies of the CPU and the GPU at runtime, Mega-KV effectively reduces its energy consumption and achieves an efficiency of up to 299 KOPS/W.

10 Conclusion

Having conducted thorough experiments and analyses, we have identified the bottleneck of IMKV running on multicore processors, which is a mismatch between the unique properties IMKV for increasingly large data processing and the CPU-based architecture. We have designed and implemented a distributed IMKV system, Mega-KV, where GPUs serve as special-purpose devices to address the bottleneck that multicore architectures cannot break. Our evaluation results show that the Mega-KV cluster significantly boosts its throughput to 623 MOPS with excellent scalability and achieves an efficiency of up to 299 KOPS/W.

Acknowledgements This work has been partially supported by the US National Science Foundation under grants CCF-1513944, CCF-1629403, and IIS-1718450, as well as a MoE AcRF Tier 1 grant (T1 251RES1610) and a MoE AcRF Tier 2 grant (MOE2017-T2-1-122) in Singapore.

References

1. CPU Frequency Scaling. https://wiki.archlinux.org/index.php/CPU_frequency_scaling/
2. Intel dpdk. http://dpdk.org/
3. Memcached. http://memcached.org/
4. Nvidia management library. https://developer.nvidia.com/nvidia-management-library-nvml/
5. Redis. http://redis.io/
6. Andersen, D.G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V.: Fawn: a fast array of wimpy nodes. In: SOSP, pp. 1–14 (2009)
7. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: SIGMETRICS, pp. 53–64 (2012)
8. Berezecki, M., Frachtenberg, E., Paleczny, M., Steele, K.: Many-core key-value store. In: IGCC, pp. 1–8 (2011)
9. Chalamalasetti, S.R., Lim, K., Wright, M., AuYoung, A., Ranganathan, P., Margala, M.: An FPGA memcached appliance. In: FPGA, pp. 245–254 (2013)
10. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC, pp. 143–154 (2010)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithm, 3rd edn. The MIT Press, Cambridge (2009)
12. Erlingsson, Ú., Manasse, M., McSherry, F.: A cool and practical alternative to traditional hash tables. In: WDAS, pp. 1–6 (2006)
13. Escriva, R., Wong, B., Sirer, E.G.: Hyperdex: a distributed, searchable key-value store. ACM SIGCOMM Comput. Commun. Rev. **42**(4), 25–36 (2012)
14. Fan, B., Andersen, D.G., Kaminsky, M.: Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In: NSDI, pp. 371–384 (2013)
15. Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A., Ailamaki, A., Falsafi, B.: A case for specialized processors for scale-out workloads. In: Micro, pp. 31–42 (2014)
16. Geambasu, R., Levy, A.A., Kohno, T., Krishnamurthy, A., Levy, H.M.: Comet: an active distributed key-value store. In: OSDI (2010)
17. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: SIGMOD, pp. 243–252 (1994)
18. Gutierrez, A., Cieslak, M., Giridhar, B., Dreslinski, R.G., Ceze, L., Mudge, T.: Integrated 3d-stacked server designs for increasing physical density of key-value stores. In: ASPLOS, pp. 485–498 (2014)
19. Han, S., Jang, K., Park, K., Moon, S.: Packetshader: A GPU-accelerated software router. In: SIGCOMM, pp. 195–206 (2010)
20. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: SIGMOD, pp. 511–524 (2008)
21. He, B., Yu, J.X.: High-throughput transaction executions on graphics processors. In: PVLDB (2011)
22. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. In: PVLDB, pp. 709–720 (2013)
23. Hetherington, T., Rogers, T., Hsu, L., O'Connor, M., Aamodt, T.: Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In: ISPASS, pp. 88–98 (2012)
24. Hetherington, T.H., O'Connor, M., Aamodt, T.M.: Memcachedgpu: Scaling-up scale-out key-value stores. In: SoCC, pp. 43–57 (2015)
25. Hong, S., Kim, H.: An integrated GPU power and performance model. ACM SIGARCH Comput. Archit. News **38**(3), 280–289 (2010)
26. Jeong, E.Y., Woo, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., Park, K.: mTCP: A highly scalable user-level tcp stack for multicore systems. In: NSDI (2014)
27. Kaldewey, T., Lohman, G., Mueller, R., Volk, P.: GPU join processing revisited. In: DaMoN, pp. 55–62 (2012)
28. Kapoor, R., Porter, G., Tewari, M., Voelker, G.M., Vahdat, A.: Chronos: predictable low latency for data center applications. In: SoCC, pp. 9:1–9:14 (2012)
29. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: STOC, pp. 654–663 (1997)
30. Lavasani, M., Angepat, H., Chiou, D.: An fpga-based in-line accelerator for memcached. Comput. Archit. Lett. **13**(2), 57–60 (2013)
31. Lee, J., Sathisha, V., Schulte, M., Compton, K., Kim, N. S.: Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling. In: 2011 International Conference on Parallel Architectures and Compilation Techniques. Galveston, TX, pp. 111–120 (2011)
32. Leng, J., Hetherington, T., Eltantawy, A., Gilani, S., Kim, N.S., Aamodt, T.M., Reddi, V.J.: GPUWattch: enabling energy optimizations in GPGPUs. ACM SIGARCH Comput. Archit. News **41**(3), 487–498 (2013)
33. Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., Rooholamini, R.: An empirical study of hyper-threading in high performance computing clusters. Linux HPC Revolution (2002)
34. Li, C., Cox, A.L.: Gd-wheel: A cost-aware replacement policy for key-value stores. In: Proceedings of the Tenth European Conference on Computer Systems, EuroSys (2015)
35. Li, S., Lim, H., Lee, V.W., Ahn, J.H., Kalia, A., Kaminsky, M., Andersen, D.G., Seongil, O., Lee, S., Dubey, P.: Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In: ISCA, pp. 476–488 (2015)
36. Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: Mica: A holistic approach to fast in-memory key-value storage. In: NSDI, pp. 429–444 (2014)
37. Lim, K., Meisner, D., Saidi, A.G., Ranganathan, P., Wenisch, T.F.: Thin servers with smart pipes: designing soc accelerators for memcached. SIGARCH Comput. Archit. News **41**, 36–47 (2013)
38. Ma, K., Li, X., Chen W., Zhang, C., Wang, X.: Green GPU: a holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In: 2012 41st International Conference on Parallel Processing. Pittsburgh, PA, pp. 48–57 (2012)
39. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: EuroSys, pp. 183–196 (2012)
40. Metreveli, Z., Zeldovich, N., Kaashoek, M.F.: Cphash: A cache-partitioned hash table. In: PPoPP, pp. 319–320 (2012)
41. Mitchell, C., Geng, Y., Li, J.: Using one-sided rdma reads to build a fast, CPU-efficient key-value store. In: USENIX ATC, pp. 103–114 (2013)
42. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V.: Scaling memcache at facebook. In: NSDI, pp. 385–398 (2013)
43. Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S.M., Stratmann, E., Stutsman, R.: The case for ramclouds: scalable high-performance storage entirely in dram. SIGOPS Oper. Syst. Rev. **43**, 92–105 (2010)
44. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2003)
45. Paul, J., He, J., He, B.: GPL: A GPU-based pipelined query processing engine. In: SIGMOD, pp. 1935–1950 (2016)

46. Pirk, H., Manegold, S., Kersten, M.: Waste not... efficient co-processing of relational data. In: ICDE, pp. 508–519 (2014)
47. Price, D.C., Clark, M.A., Barsdell, B.R., Babich, R., Greenhill, L.J.: Optimizing performance-per-watt on GPUs in high performance computing. Comput. Sci. Res. Dev. **31**(4), 185–193 (2016)
48. Richter, S., Alvarez, V., Dittrich, J.: A seven-dimensional analysis of hashing methods and its implications on query processing. In: VLDB, pp. 96–107 (2015)
49. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.M.W.: Optimization principles and application performance evaluation of a multithreaded GPU using cuda. In: PPoPP, pp. 73–82 (2008)
50. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: SOSP (2013)
51. Wang, K., Ding, X., Lee, R., Kato, S., Zhang, X.: Gdm: Device memory management for GPGPU computing. In: SIGMETRICS, pp. 533–545 (2014)
52. Wang, K., Zhang, K., Yuan, Y., Ma, S., Lee, R., Ding, X., Zhang, X.: Concurrent analytical query processing with GPUs. In: PVLDB, pp. 1011–1022 (2014)
53. Wu, H., Diamos, G., Cadambi, S., Yalamanchili, S.: Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In: MICRO, pp. 107–118 (2012)
54. Yuan, Y., Lee, R., Zhang, X.: The yin and yang of processing data warehousing queries on GPU devices. In: PVLDB, pp. 817–828 (2013)
55. Zhang, K., Chen, F., Ding, X., Huai, Y., Lee, R., Luo, T., Wang, K., Yuan, Y., Zhang, X.: Hetero-db: next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. JCST **30**, 657–678 (2015)
56. Zhang, K., Hu, J., He, B., Hua, B.: Dido: Dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In: ICDE, pp. 671–682 (2017)
57. Zhang, K., Hu, J., Hua, B.: A holistic approach to build real-time stream processing system with GPU. JPDC **83**(C), 44–57 (2015)
58. Zhang, K., Wang, K., Yuan, Y., Guo, L., Lee, R., Zhang, X.: Mega-kv: a case for GPUs to maximize the throughput of in-memory key-value stores. Proc. VLDB Endow. **8**, 1226–1237 (2015)
59. Zhou, J., Ross, K.A.: Buffering accesses to memory-resident index structures. In: VLDB, pp. 405–416 (2003)