Dynamic Hash Tables on GPUs

Yuchen Li, Jing Zhang, Yue Liu, Zheng Lyu, Zhongdong Huang, Jianling Sun

Abstract—The hash table is a fundamental structure that has been implemented on graphics processing units (GPUs) to accelerate a wide range of analytics workloads. Most existing works have focused on static scenarios and occupying large GPU memory to maximize the insertion efficiency. In many cases, data stored in hash tables get updated dynamically and existing approaches use unnecessarily large memory resources. One naïve solution is to rebuild a hash table (known as rehashing) whenever it is either filled or mostly empty. However, this approach renders significant overheads for rehashing. In this paper, we propose a novel dynamic cuckoo hash table technique on GPUs, known as *DyCuckoo*. We devise a resizing strategy for dynamic scenarios without rehashing the entire table that ensures a guaranteed filled factor. The strategy trades search performance with resizing efficiency, and this tradeoff can be configured by users. To further improve efficiency, we propose a two-layer cuckoo hashing scheme that ensures a maximum of *two* lookups for find and delete operations, while retaining similar performance for insertions as a general cuckoo hash. Extensive experiments have validated the proposed design's effectiveness over several state-of-the-art hash table implementations on GPUs. *DyCuckoo* achieves superior performance while saving up to four times the memory over the state-of-the-art approaches against dynamic workloads.

1 INTRODUCTION

Exceptional advances in general-purpose graphics processing units (GPGPUs) in recent years have completely revolutionized computing paradigms across multiple fields such as cryptocurrency mining [27], [34], machine learning [10], [1], and database technologies [5], [20]. GPUs bring phenomenal computational power that had previously only been available from supercomputers in the past. Hence, there is a prevailing interest in developing efficient parallel algorithms for GPUs to enable real-time analytics.

In this paper, we investigate a fundamental data structure, known as the *hash table*, which has been implemented on GPUs to accelerate applications, ranging from relational hash joins [15], [14], [16], data mining [30], [39], [38], key value stores [36], [17], [9], and many others [8], [29], [13], [26], [35]. Existing works [2], [36], [18], [17], [9] have focused on static scenarios in which the size of the data is known in advance and a sufficiently large hash table is allocated to insert all data entries. However, data size varies in different application scenarios such as sensor data processing, Internet traffic analysis and analysis of transaction logic such as in web server logs and telephone calls. When data size varies, the static allocation strategy leads to poor memory utilization [4]. The static strategy is thus inefficient when an application requires multiple data structures to coexist on GPUs. One must resort to expensive PCIe data transfer between CPUs and GPUs, as the hash table takes up an unnecessarily large memory space. Addressing this shortcoming calls for a dynamic GPU hash table that adjusts to the size of active entries in the table. Such a hash table should support efficient memory management by sustaining a guaranteed *filled factor* of the table when the data size changes. In addition to efficient memory usage, the dynamic approach should retain the performance of common hash table operations such as find, delete, and insert. Although dynamically-sized hash tables have been studied across academia [25], [40] and industry [23], [11] for CPUs, GPU-based dynamic hash tables have largely been overlooked.

In this paper, we propose a dynamic cuckoo hash table on GPUs, known as `DyCuckoo`. Cuckoo hashing [28] uses several hash functions to give each key multiple locations instead of one. When a location is occupied, the existing key is relocated to make room for the new one. Existing works [2], [3], [36], [9] have demonstrated great success in speeding up applications using parallel cuckoo hashes on GPUs. However, a complete relocation of the entire hash table is required when the data cannot be inserted. In this work, we propose two novel designs for implementing dynamic cuckoo hash tables on GPUs.

First, we employ the cuckoo hashing scheme with d subtables specified by d hash functions, and introduce a resizing policy to maintain the filled factor within a bounded range while minimizing entries in all subtables being relocated at the same time. If the filled factor falls out of the specified range, insertions and deletions would cause the hash tables to grow and shrink. Our proposed policy only locks one subtable for resizing and ensures that no subtable can be more than twice as large as any other to efficiently handle subsequent resizing. Meanwhile, the hash table entries are distributed to give each subtable a nearly equivalent filled factor. In this manner, we drastically reduce the cost of resizing hash tables and provide better system availability than the static strategy, which must relocate all data for resizing. Our theoretical analysis demonstrates the scheduling policy's optimality in terms of processing updates.

Second, we propose a two-layer cuckoo hashing scheme to ensure efficient hash table operations. The proposed resizing strategy requires d hash tables, which indicates d

- *Y. Li is with the School of Information Systems, Singapore Management University. E-mail: yuchenli@smu.edu.sg*
- *J. Zhang, Y. Liu, Z. Huang and J. Sun are with the College of Computer Science and Technology, Zhejiang University. E-mail: {zhangjing000, liuyue1013, hzd, sunjl}@zju.edu.cn*
- *Z. Lyu is with the Alibaba Group.*

TABLE 1
Frequently Used Notations

(k, v)	a key value pair
d	the number of hash functions
h^i	the ith hash table
$\|h^i\|, n_i, m_i$	range, table size and data size of h^i
wid, l	a warp ID and the lth lane of the warp
θ	the filled factor of the entire hash table
θ_i	the filled factor of hash table i
loc	a bucket in the hash table

lookup positions for find and delete operations, and a larger d indicates less workload for resizing but more lookups for find and delete operations. To mitigate this tradeoff, we devise a two-layer approach that first hashes any key to a pair of hash tables where the key can be further hashed and stored in one of the two hash tables. This design ensures that there are a maximum of two lookups for any find and deletion operations. Furthermore, the two-layer approach retains the general cuckoo hash tables' performance guarantee. Empirically, the proposed hash table design can operate efficiently at filled factors exceeding 90%.

Thus, we summarize our contributions as follows:

- We propose an efficient strategy for resizing hash tables and demonstrate the near-optimality of the resizing strategy through theoretical analysis.
- We devise a two-layer cuckoo hash scheme that ensures a maximum of two lookups for find and deletion operations, while still retaining similar performance for insertions as general cuckoo hash tables.
- We conduct extensive experiments on both synthetic and real datasets and compare the proposed approach against several state-of-the-art GPU hash table baselines. For dynamic workloads, the proposed approach demonstrates superior performance and reduces memory usage by up to a factor of four over the compared baselines.

The remainder of this paper is organized as follows. Section 2 introduces the preliminary information and provides a background on GPUs. Section 3 documents related work. Section 4 introduces the hash table design and the resizing strategy against dynamic updates. Section 5 presents the two-layer cuckoo hash scheme along with parallel operations on GPUs. The experimental results are reported in Section 6. Finally, we conclude the paper in Section 7.

2 PRELIMINARIES

In this section, we first introduce some preliminary information on general hash tables and present background material on GPU architecture. Frequently used notations are summarized in Table 1.

2.1 Hash Table

The hash table is a fundamental data structure that stores KV pairs (k, v), and the value could refer to either actual data or a reference to the data. Hash tables offer the following functionalities: INSERT (k, v), which stores (k, v) in the hash table; FIND (k), in which the given k values returns the associated values if they exist and NULL otherwise; and

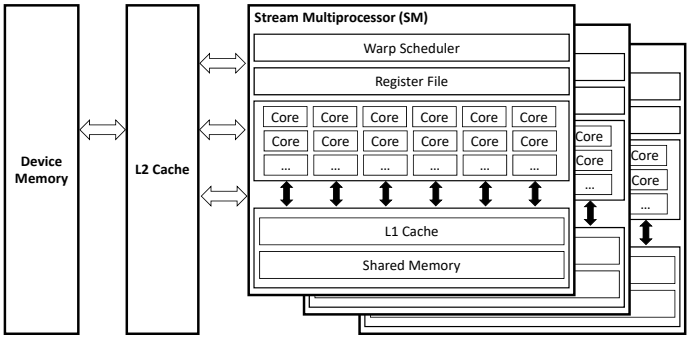

Fig. 1. Layout of an NVIDIA GPU architecture

DELETE (k), which removes existing KV pairs that match k if they are present in the table.

Given a hash function with range $0 \dots h-1$, collisions must happen when we insert $m > h$ keys into the table. There are many schemes to resolve collisions: linear probing, quadratic probing, chaining and etc. Unlike these schemes, cuckoo hashing [28] guarantees a worst case constant complexity for FIND and DELETE, and an amortized constant complexity for INSERT. A cuckoo hash uses multiple (i.e., d) hash tables with independent hash functions $h^1, h^2, \dots, h^d$ and stores a KV pair in *one* of the hash tables. When inserting (k, v), we store the pair in $loc = h^1(k)$ and terminate if there is no element at this location. Otherwise, if there exists k' such that $h^1(k') = loc$, k' is evicted and then reinserted into another hash table, e.g., $loc' = h^2(k')$. We repeat this process until encountering an empty location.

For a hash table with the hash function h^i, $|h^i|$ is defined to be the number of unique hash values for h^i and n_i to be the total memory size allocated for the hash table. A location or a hash value for h^i is represented as $loc = h^i_j$ where $j \in [0, |h^i| - 1]$. If the occupied space of the hash table is m_i, the filled factor of h^i is denoted as $\theta_i = m_i/n_i$. The overall filled factor of the cuckoo hash table is thus denoted as $\theta = (\sum_i m_i)/(\sum_i n_i)$.

2.2 GPU Architecture

We introduce the background of the NVIDIA GPU architecture in this paper because its popularity and the wide adoption of the CUDA programming language. However, our proposed approaches are not unique to NVIDIA GPUs and can also be implemented on other GPU architectures. Figure 1 presents a high-level layout of a NVIDIA GPU. An application written in CUDA executes on GPUs by invoking the *kernel* function. The kernel is organized as several *thread blocks*, and one block executes all its threads on a *streaming multiprocessor* (SM), which contains several CUDA cores as depicted in Figure 1. Within each block, threads are divided into *warps* of 32 threads each. A CUDA core executes the same instruction of a warp in lockstep. Each warp runs independently, but warps can collaborate through different memory types as discussed in the following.

Memory Hierarchy. Compared with CPUs, GPUs are built with large register files that enable massive parallelism. Furthermore, the shared memory, which has similar performance to L1 cache, can be programmed within a block

to facilitate efficient memory access inside an SM. The L2 cache is shared among all SMs to accelerate memory access to the device memory, which has the largest capacity and the lowest bandwidth in the memory hierarchy.

Optimizing GPU Programs. There are several important guidelines to harness the massive parallelism of GPUs.

- *Minimize Warp Divergence.* Threads in a warp will be serialized if executing different instructions. To enable maximum parallelism, one must minimize branching statements executed within a warp.
- *Coalesced Memory Access.* Warps have a wide cache line size. The threads are better off reading consecutive memory locations to fully utilize the device memory bandwidth, otherwise a single read instruction by a warp will trigger multiple random accesses.
- *Control Resource Usage.* Registers and shared memory are valuable resources for enabling fast memory accesses. Nevertheless, each SM has limited resources and overdosing register files or shared memory leads to reduced parallelism on an SM.
- *Atomic Operations.* When facing thread conflicts, an improper locking implementation causes serious performance degradation. One can leverage the native support of atomic operations [33] on GPUs to carefully resolve the conflicts and minimize thread spinning.

3 RELATED WORKS

Alcantara *et al.* [2] presented a seminar work on GPU-based cuckoo hashing to accelerate computer graphics workloads. This work has inspired several applications from diverse fields. Wu *et al.* [35] investigated the use of GPU-based cuckoo hashing for on-the-fly model checking. A proposal of accelerating the nearest neighbor search is presented in [29]. Because of the success of cuckoo hashing on GPUs, the implementation of [2] has been adopted in the CUDPP library[1]. To improve on [2], stadium hash was proposed in [21] to support out-of-core GPU parallel hashing. However, this technique uses double hashing which must rebuild the entire table for any deletions. Zhang *et al.* [36] proposed another efficient design of GPU-based cuckoo hashing, named MegaKV, to boost the performance for KV store. Subsequently, Horton table [9] improves the efficiency of FIND over MegaKV by trading with the cost of introducing a KV remapping mechanism. WarpDrive [19] employs cooperative groups and multi-GPUs to further improve efficiency. Meanwhile, in the database domain, several SIMD hash table implementations have been proposed to facilitate relation join and graph processing [32], [38].

It is noted that these works have focused on the static case: the data size for insertions is known in advance. The static design would prepare a large enough memory size to store the hash table. In this manner, hash table operations are fast as collisions rarely happen. However, the static approach wastes memory resources and, to some extent, prohibits coexistence with other data structures for the same application in the device memory. This motivates us to develop a general dynamic hash table for GPUs that actively adjusts based on the data size to preserve space efficiency.

1. https://github.com/cudpp/cudpp

To the best of our knowledge, there is only one existing work on building dynamic hash tables on GPUs [4]. This proposed approach presents a concurrent linked list structure, known as *slab lists*, to construct the dynamic hash table with *chaining*. However, there are three major issues for slab lists. First, they can frequently invoke concurrent memory allocation requests, especially when the data keeps inserting. Efficient concurrent memory allocation is difficult to implement in a GPU due to its massive parallelism. Although a dedicated memory management strategy to alleviate this allocation cost is proposed in [4], the strategy is not transparent to other data structures. More specifically, the dedicated allocator still has to reserve a large amount of memory in advance to prepare for efficient dynamic allocation, and that occupied memory space cannot be readily accessed by other GPU-resident data structures. Second, a slab list does not guarantee a fixed filled ratio against deletions. It symbolically marks a deleted entry without physically freeing the memory space. Hence, memory spaces are wasted when occupied by deleted entries. Third, the chaining approach has a lookup time of $\Omega(log(log(m)))$ for some KVs with high probability. This not only results in degraded performance for FIND, it also triggers more overhead for resolving conflicts when multiple INSERT and DELETE operations occur at the same key. In contrast, the cuckoo hashing table adopted in this work guarantees $O(1)$ worst case complexity for FIND and DELETE, and $O(1)$ amortized INSERT performance. Moreover, we do not introduce extra complication in implementing a customized memory manager, but rather rely on the default memory allocator provided by CUDA, while at the same time, ensuring fixed filled ratios for the hash table.

4 DYNAMIC HASH TABLE

In this section, we propose a resizing strategy against dynamic hash table updates on GPUs. We first present the hash table design in Section 4.1. Subsequently, the resizing strategy is introduced in Section 4.2. In Section 4.3, we discuss how to distribute KV pairs for better load balancing with theoretical guarantees. Lastly, we present how to efficiently rehash and relocate data after the tables have been resized in Section 4.4.

4.1 Hash Table Structure

Following cuckoo hashing [28], we build d hash tables with d unique hash functions: $h^1, h^2, \ldots, h^d$. In this work, we use a set of simple universal hash functions such as $h^i(k) = (a_i \cdot k + b_i \mod p) \mod |h^i|$. Here a_i, b_i are random integers and p is a large prime. The proposed approaches in this paper also apply to other hash functions as well. There are three major advantages of adopting cuckoo hashing on GPUs. First, it avoids chaining by inserting the elements into alternative locations if collisions occur. As discussed in Section 3, chaining presents several issues that are not friendly to GPU architecture. Second, to look up a KV pair, one searches only d locations as specified by d unique hash functions. Thus, the data could be stored contiguously in the same location to enable preferred coalesced memory access. Third, cuckoo hashing can maintain a high filled factor,

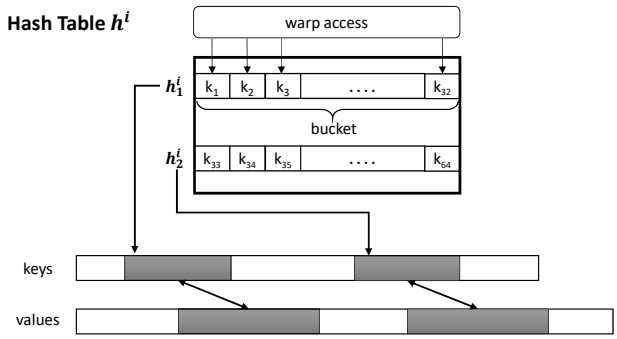

Fig. 2. The hash table structure

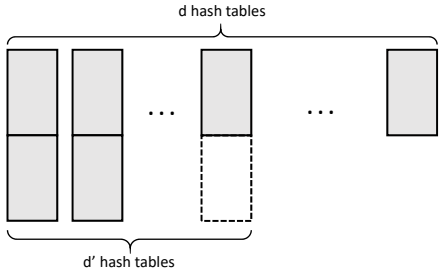

Fig. 3. The resizing strategy

which is ideal for saving memory in dynamic scenarios. For $d = 3$, cuckoo hashing achieves a filled factor of more than 90% and still efficiently processes **INSERT** operations [12].

Figure 2 depicts the design of a single hash table h^i on GPUs. The keys are assumed to be 4-byte integers and a bucket of 32 keys, which are all hashed to the same value h_j^i, are stored consecutively in the memory. The design of buckets maximizes memory bandwidth utilization in GPUs. Consider that the L1 cache line size is 128 bytes. Only a single access is required when one warp is assigned to access a bucket. The values associated with the keys in the same bucket are also stored consecutively, but in a separate array. In other words, we use two arrays, one to store the keys and one to store the values respectively. However, the values can take up a much larger memory space than the keys; therefore storing keys and values separately avoids memory access overhead when it is not necessary to access the values, such as when finding a nonexistent KV pair or deleting a KV pair.

For keys larger than 4 bytes, a simple strategy is to store fewer KV pairs in a bucket. If keys are 8 bytes, a bucket can then accommodate 16 KV pairs. Furthermore, we lock the entire bucket exclusively for a warp to perform insertions and deletions using intra warp synchronization primitives. Thus, we do not limit ourselves to supporting KV pairs with only 64 bits. In the worst case, a key taking 128 bytes would occupy one bucket, which is unnecessarily large in practice.

4.2 Structure Resizing

To efficiently utilize GPU memory, we resize the hash tables when the filled factor falls out of the desired range $[\alpha, \beta]$. One possible strategy to address this is to double or half all hash tables and rehash all KV pairs. However, this simple strategy renders poor memory utilization and excessive rehashing overhead. First, doubling hash table size results in the filled factor being immediately cut in half, whereas downsizing hash tables to half the original size followed by rehashing is only efficient when the filled factor is significantly low (e.g., 40%). Both scenarios are not resource friendly. Second, rehashing all KV pairs is expensive and it harms the performance stability for most streaming applications as the entire table is subject to locking.

Thus, we propose an alternative strategy, illustrated in Figure 3. Given d hash tables, we always double the smallest subtable or chop the largest subtable in half for upsizing or downsizing, respectively, when the filled factor falls out of the desired range. In other words, no subtable will be more than twice the size of others. This strategy implies that we do not need to lock all hash tables to resize only one, thus achieving better performance stability than the aforementioned simple strategy.

Filled factor analysis: Assuming there are d' hash tables with size $2n$, $d - d'$ tables with size n and a current filled factor of θ, one upsizing process when $\theta > \beta$ lowers the filled factor to $\frac{\theta \cdot (d+d')}{d+d'+1} \geq \frac{\beta \cdot d}{d+1}$. Because the filled factor is always lower bounded by α, we can deduce that $\alpha < \frac{d}{d+1}$. Apparently, a higher lower bound can be achieved by adding more hash tables, although it leads to less efficient **FIND** and **DELETE** operations. We allow the user to configure the number of hash tables to trade off memory and query processing efficiency.

4.3 KV distribution

Given a set of KV pairs to insert in parallel, it is critical to distribute those KV pairs among the hash tables in a way that minimizes hash collisions to reduce the corresponding thread conflicts. We have the following theorem to guide us in distributing KV pairs.

Theorem 1. *The amortized conflicts for inserting m unique KV pairs to d hash tables are minimized when $\binom{m_1}{2}/n_1 = \ldots = \binom{m_d}{2}/n_d$. m_i and n_i denote the elements inserted to table i and the size of table i, respectively.*

Proof. The amortized insertion complexity of a cuckoo hash is $O(1)$. Thus, like a balls and bins analysis, the expected number of conflicts occurring when inserting m_i elements in table i can be estimated as $\binom{m_i}{2}/n_i$. Minimizing the amortized conflicts among all hash tables can be modeled as the following optimization problem:

$$\begin{array}{ll} \min_{m_1,\ldots,m_d \geq 0} & \sum_{i=1,\ldots,d} \binom{m_i}{2}/n_i \\ \text{s.t.} & \sum_{i=1,\ldots,d} m_i = m \end{array} \quad (1)$$

To solve the optimization problem, we establish an equivalent objective function:

$$\min \sum_{i=1,\ldots,d} \frac{\binom{m_i}{2}}{n_i} \Leftrightarrow \min \log(\frac{1}{d} \sum_{i=1,\ldots,d} \frac{\binom{m_i}{2}}{n_i})$$

According to the Jensen's inequality, the following inequality holds:

$$\log(\frac{1}{d} \sum_{i=1,\ldots,d} \frac{\binom{m_i}{2}}{n_i}) \geq \frac{1}{d} \sum_{i=1,\ldots,d} \log(\frac{\binom{m_i}{2}}{n_i})$$

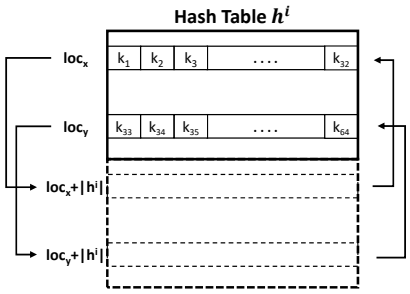

Fig. 4. Illustration for upsizing and downsizing.

where equality holds when $\binom{m_i}{2}/n_i = \binom{m_j}{2}/n_j \ \forall i,j = 1,\ldots,d$ and we obtain the minimum. □

Based on our resizing strategy, one hash table can only be twice as large as the other tables. This implies that the filled factors of two tables are equal if they have the same size, i.e., $\theta_i = \theta_j$ if $n_i = n_j$, while $\theta_i \simeq \sqrt{2} \cdot \theta_j$ if $n_i = 2n_j$. Thus, larger tables should have a higher filled factor. Following Theorem 1, we employ a randomized approach: a KV pair (k, v) is first assigned to table i with a proportional probability to $n_i/\binom{m_i}{2}$ to ensure the distribution of KVs.

4.4 Rehashing

Whenever the filled factor falls out of the desired range, rehashing relocates KV pairs after one of the hash tables is resized. An efficient relocation process maximizes GPU device memory bandwidth and minimizes thread conflicts. We discuss two scenarios for rehashing: *upsizing* and *downsizing*, both of which are processed in a single kernel.

Upsizing. Here, we introduce a conflict-free rehashing strategy for the upsizing scenario. Figure 4 illustrates the upsizing of a hash table h^i. As we always double the size for h^i, a KV pair that originally resides in bucket loc could be rehashed to bucket $loc + |h^i|$ or stay in the original bucket. With this observation, we assign a warp for rehashing all KV pairs in the bucket to fully utilize the cache line size. Each thread in the warp takes a KV pair in the bucket and, if necessary, relocates that KV pair. Moreover, rehashing does not trigger any conflicts as KV pairs from two distinct buckets before upsizing cannot be rehashed to the same bucket. Thus, locking of the bucket is not required, meaning we can make use of the device's full memory bandwidth for the upsizing process.

After upsizing hash table h^i, its filled factor θ_i is cut in half, which could break the balancing condition emphasized in Theorem 1. Nevertheless, we use a sampling strategy for subsequent KV insertions, in which each insertion is allocated to table i with a probability proportional to $n_i/\binom{m_i}{2}$, to recover the balancing condition. In particular, m_i remains the same but n_i doubles after upsizing, and the scenario leads to doubling the probability of inserting subsequent KV pairs to h^i.

Downsizing. Downsizing h^i is the reverse process of upsizing h^i. There is always room to relocate KV pairs in the same table for upsizing. However, downsizing may rehash some KV pairs to other hash tables, especially when $\theta_i > 50\%$. Because the KV pairs located in loc and $loc + |h^i|$ are hashed to loc in the new table, there could be cases in which the KV pairs exceed the size of a single bucket. Hence, we first assign a warp to accommodate KV pairs that can fit the size of a single bucket. Like upsizing, downsizing does require locking as there will be no thread conflict on any bucket. For the remaining KV pairs that cannot fit in the downsized table, known as *residuals*, we insert them into other subtables. To ensure no conflict occurs between inserting residuals and processing the downsizing subtable, both of which are executed in a single kernel, we exclude the downsizing subtable when inserting the residuals. As an example when we have three subtables and one of them is being downsized, we only insert the residuals to the remaining two subtables.

Complexity Analysis. Given a total of m elements in the hash tables, upsizing or downsizing rehashes at most m/d KV pairs. To insert or delete these m elements, the number of rehashes is bounded by $2m$. Thus, the amortized complexity for inserting m elements remains $O(1)$.

5 Two-layer Cuckoo Hash

In this section, we present a two-layer approach that ensures a maximum of two lookups for FIND and DELETE (Section 5.1). Subsequently, in Section 5.2, we give details on optimizing GPUs for paralleling hash table operations such as FIND, INSERT, and DELETE.

5.1 The Two-layer Approach

Given the proposed dynamic hash table design, a larger d implies a smaller workload for each resizing operation, as each single table will be smaller with fixed filled factor. In addition to efficient resizing, a higher filled factor can be maintained for a larger d as discussed in Section 4.2. Nevertheless, the benefit of employing more tables does not come for free. For each FIND and DELETE operations, one must perform d lookups, which translates to d random accesses to the device memory. Random accesses are particularly expensive as GPUs contain limited cache size and simplified control units compared to CPUs.

One possible approach to reduce the number of lookups is to first hash all KV pairs into d' partitions. For each partition, we employ a cuckoo hash with two hash functions. In this manner, one only performs at most two lookups for any FIND and DELETE operations. However, this approach cannot prevent skewness across the d' partitions, especially with frequent delete operations. It is possible that the deleted KVs all fall into one partition c_i, which results in low filled factor for the table or tables allocated to c_i. Furthermore, when inserting KVs into other partitions, e.g., $c_j \neq c_i$, efficiency could be severely degraded due to the high filled factor of the table or tables allocated to c_j.

Hence, we propose a two-layer approach to resolve the skewness problem. The two-layer approach is inspired by data partitioning techniques [6], [7], [37], [40]. Given d hash tables, we first hash all KV pairs into $\binom{d}{2}$ partitions, each of which refers to a unique pair of hash tables. Then each KV pair is hashed and stored in only one of the corresponding pair. This only requires a maximum of two lookups for FIND and DELETE. The advantage of this approach is that each KV

pair could appear in any of the d hash tables, which provides opportunities to balance a skewed distribution. The following example illustrates a scenario where skewness is mitigated during the insertion process.

Example 1. *A KV pair (k, v) is hashed to the hash table pair (h_i, h_j) for the first layer. We then hash k and try to insert (k, v) into h_i for the second layer. Assuming the corresponding bucket in h_i is full for (k, v), we evict another KV pair (k', v'). We then discover that (k', v') is hashed to the pair (h_i, h_t). Henceforth, we insert (k', v') to h_t and the process repeats until no further evictions occur.*

The above example shows that the eviction could reinsert a KV pair into any hash table h_t. As each filled bucket contains 32 KV pairs (assuming 4 byte keys), one can pick a KV pair for reinsertion into a desired hash table based on the balancing strategy discussed in Theorem 1. In addition to the ability to mitigate data skewness, the two-layer cuckoo hash has the same asymptotic insertion performance as a plain cuckoo hash table with two hash functions.

Theorem 2. *The two-layer cuckoo hash approach has the same expected, amortized complexity of insertions as a plain cuckoo hash with two hash tables.*

Proof. Assuming d hash tables for the two-layer approach, without loss of generality we set the range for each hash function to be $[0, n)$. Given a KV pair (k, v), we denote hash function hp as the one that hashes (k, v) to a pair of hash tables. Now, we transform the two-layer approach to the plain cuckoo hash by constructing two new hash functions $H_1(k) = i \cdot n \cdot h_i(k)$ and $H_2(k) = j \cdot n \cdot h_j(k)$ where $hp(k) = (h_i, h_j)$. The apparent range of H_1 and H_2 is $[0, nd)$. Thus, we can build a random bipartite graph $G(U, V, E)$, where U represents the buckets for H_1, V represents the buckets for H_2, and E represents the KV pairs connecting the two buckets from H_1 and H_2. Each KV pair is independently hashed to a random edge $e \in E$ with the same probability, i.e., $1/(n^2d^2)$. Hence, we can follow a similar proof procedure that utilizes random bipartite graph analysis to show the amortized complexity of a cuckoo hash with two tables [22], to prove Theorem 2. □

5.2 Parallel Hash Table Operations

In the reminder of this section, we discuss how to utilize GPUs for the two-layer cuckoo hash. Following existing works [2], [36], [9], we assume that the FIND, INSERT and DELETE operations are batched and that each batch contains only one type of operations.

Find. It is relative straightforward to parallelize FIND operations as only read access is required. Given a batch of size m, we launch w warps (meaning we launch $32w$ threads), with each warp being responsible for $\lfloor \frac{m}{w} \rfloor$ FIND operations. To locate a KV pair, we hash the key to a hash table pair (h_i, h_j) and perform a maximum of two lookups in the corresponding buckets of h_i and h_j respectively.

Insert. Contention occurs when multiple INSERT operations target at the same bucket. There are two contrasting objectives for resolving contention. On one hand, we want to utilize a warp-centric approach to access a bucket. On the other hand, when updating a bucket, a warp requires a mutex to avoid corruption, and on GPUs locking is expensive. In the literature, it is a common practice to use atomic operations for implementing a mutex under a warp-centric approach [36]. We can still invoke a warp to insert a KV pair; however, the warp must acquire a lock before updating the corresponding bucket. The warp will keep trying to acquire the lock before successfully obtain control. There are two drawbacks to this direct warp-centric approach. First, the conflicting warps spin while locking, thus wasting computing resources. Second, although atomic operations are natively supported by recent GPU architectures, they become costly when the number of atomic operations issued at the same location increases. In Figure 5, we show the profiling statistics for two atomic operations that are often used to lock and unlock a mutex: atomicCAS and atomicExch, respectively. We compare the throughputs of the atomic operations against an equivalent amount of sequential device memory IOs (coalesced) and present the trend for varying the number of conflicting atomic operations. It is apparent that the atomic performance seriously degrades when a larger number of conflicts occur. Thus, it will be expensive for the direct warp-centric approach in contention critical cases. Suppose that one wants to track the number of retweets posted to active Twitter accounts in the current month by storing the Twitter ID and the obtained retweet counts as KV pairs. In this scenario, certain Twitter celebrities could receive thousands of retweets in a very short period. This causes the same Twitter ID to get updated frequently, thus a large number of conflicts would happen.

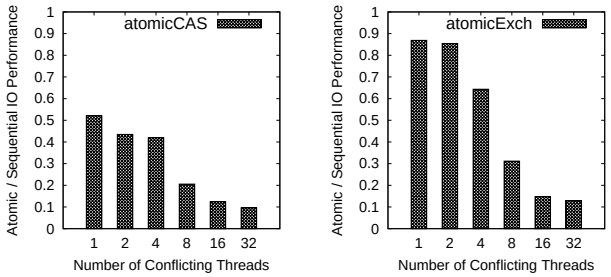

Fig. 5. The performance of atomic operations for increasing conflicts

To alleviate the cost of spinning, we devise a voter coordination scheme. We assign an INSERT to a thread rather than directly assigning the operation to a warp. Before submitting a locking request and updating the corresponding bucket, the thread will participate in a vote among threads within the same warp. The winning thread l becomes the leader of the warp and takes control. Subsequently, the warp inspects the bucket and inserts the KV pair in l if there are spaces left once l has successfully obtained the lock. If l fails to get the lock, the warp votes for another leader to avoid locking on the same bucket. Compared with locking in atomic operations, the cost of warp voting is almost negligible as it is heavily optimized in GPU architecture.

Parallel insertions with the voter coordination scheme is presented in Algorithm 1. The pseudocode in Algorithm 1 demonstrates how a thread (with lane l) from warp wid inserts a KV pair. The warp first conducts a vote among active threads using the ballot function and the process terminates if all threads finish their tasks (lines 1-5). This

Algorithm 1 Insert(lane l, warp wid)

```
1:  active ← 1
2:  while true do
3:      l' ← ballot(active == 1)
4:      if l' is invalid then
5:          break
6:      [(k', v'), i'] ← broadcast(l')
7:      loc = h^i'(k')
8:      if l' == l then
9:          success ← lock(loc)
10:     if broadcast(success, l') == failure then
11:         continue
12:     l* ← ballot(loc[l].key == k' || loc[l].key == ∅)
13:     if l* is valid and l' == l then
14:         loc[l*].(key, val) ← (k', v')
15:         unlock(loc)
16:         active ← 0;
17:         continue
18:     l* ← ballot(loc[l].key ≠ ∅)
19:     if l* is valid and l' == l then
20:         swap(loc[l*].(key, val), (k', v'))
21:         unlock(loc)
```

achieves better resource utilization as no thread will be idle when another thread in the same warp is active. The leader l' then broadcasts its KV pair (k', v') and the hash table $h_{i'}$ to the warp and attempts to lock the inserting bucket (lines 6-9). The ballot and broadcast functions are implemented using the CUDA warp-level primitives $\_\_ballot$ and $\_\_shfl$ [2]. The broadcast function ensures that all threads in the warp receive the locking result, and if l' fails to obtain a lock, the warp revotes. Otherwise, the warp follows l' and proceeds to update the bucket for (k', v') with a warp-centric approach like FIND. Once a thread finds k' or an empty space in the bucket, l' adds or updates it with (k', v') (lines 12-17). If no empty slot is found, l' swaps (k', v') with another KV pair (k^*, v^*) in the bucket and inserts the evicted KV pair to hash table j in the next round. The warp finishes the process when all KV pairs have been inserted.

Implementation Details. We use atomicCAS and atomicExch functions to lock and unlock buckets, respectively. The function $atomicCAS(address, compare, val)$ reads the value old located at the address $address$ in global or shared memory and computes $old == compare\ ?\ val\ :\ old$, and stores the result back to memory at the same address. The function returns the value old. The function $atomicExch(address, val)$ reads the value old located at the address $address$ in the global or shared memory and stores val back to memory at the same address. To implement the lock, we initialize a lock variable known as $lock$ for each bucket with a value of 0. We lock the bucket using the function $atomicCAS(\&lock, 0, 1)$, which is successful if the function returns 0. Similarly, we unlock the bucket using the function $atomicExch(\&lock, 0)$.

The following example demonstrates the parallel insertion process.

Example 2. *In Figure 6, we visualize a scenario for three threads:*

2. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

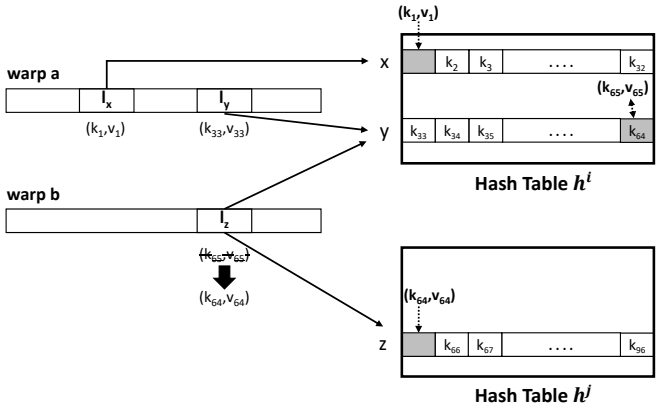

Fig. 6. Example for parallel insertions

l_x, l_y, l_z from warp a and warp b, which insert KV pairs (k_1, v_1), (k_{33}, v_{33}), and (k_{65}, v_{65}) independently. Suppose that l_y and l_z become the leaders of warp a and b, respectively. Both threads will compete for bucket y and l_z wins the battle. l_z then leads warp b to inspect the bucket and evict KV pair (k_{64}, v_{64}) by replacing it with (k_{65}, v_{65}). In the meantime, l_y does not lock bucket y and the new leader l_x is voted in warp a. Thread l_x locks bucket x and inserts KV pair (k_1, v_1) in place. Subsequently, l_y may regain the control of warp a and update k_{33} with (k_{33}, v_{33}) at bucket y. In parallel, l_z locks bucket z and inserts the evicted KV (k_{64}, v_{64}) into the empty space.

Delete. In contrast with INSERT, the DELETE operation does not require locking with a warp-centric approach. As with FIND, we assign a warp to process a key k on deletion. The warp iterates through the buckets of all d hash tables that could possibly contain k. Each thread lane in the warp is responsible for inspecting one position in a bucket independently, and erasing the key only if k is found, thus causing no conflict.

Complexity. Because FIND, INSERT and DELETE operations are independently executed by threads, the analysis of a single thread's complexity is the same as in the sequential version of cuckoo hashing [28]: $O(1)$ worst case complexity for FIND and DELETE, $O(1)$ expected time for INSERT for the case of two hash tables. It has been pointed out that analyzing the theoretical upper bound complexity of insertion in $d \geq 3$ hash tables is difficult [2]. Nevertheless, empirical results have shown that increasing the number of tables leads to better insertion performance. Please refer to the experimental results presented in Section 6.

We then analyze the number of possible thread conflicts. Assuming we launch m threads in parallel, each thread is assigned to a unique key, and the total number of unique buckets is $H = \sum_{i=1}^{d} |h^i|$. For FIND and DELETE, there is no conflict at all. For INSERT, computing the expected number of conflicting buckets resembles the *balls and bins* problem [31], which is $O(\binom{m}{2}/H)$. Given that GPUs have many threads, there could be a significant amount of conflicts. Thus, we propose the voter coordination scheme to reduce the cost of spinning in locks.

TABLE 2
The datasets used in the experiments.

Datasets	KV pairs	Unique keys
TW	50,876,784	44,523,684
RE	48,104,875	41,466,682
LINE	50,000,000	45,159,880
COM	10,000,000	4,583,941
RAND	100,000,000	100,000,000

TABLE 3
The parameters in the experiments

Parameter	Settings	Default
Filled Factor θ	70%, 75%, 80%, 85%, 90%	85%
Lower Bound α	20%, 25%, 30%, 35%, 40%	30%
Upper Bound β	70%, 75%, 80%, 85%, 90%	85%
Ratio r	0.1, 0.2, 0.3, 0.4, 0.5	0.2
Batch Size	2e5, 4e5, 6e5, 8e5, 10e5	10e5

6 EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments by comparing the proposed hash table design DyCuckoo, with several state-of-the-art GPU-based hash table approaches and one CPU-based concurrent hash table approach. Section 6.1 introduces the experimental setup. Section 6.2 presents a discussion on the sensitivity analysis of DyCuckoo. In Sections 6.3 and 6.4, we compare all approaches under the static and the dynamic experiments respectively.

6.1 Experimental Setup

Baselines. We compare DyCuckoo with several state-of-the-art hash table implementations on both CPUs and GPUs. These implementations include the following:

- Libcuckoo is a well-established CPU-based concurrent hash table that parallelizes a cuckoo hash [24].
- CUDPP is a popular CUDA primitive library containing the cuckoo hash table implementation published in [2]. In our experiments we use the default setup of CUDPP, which automatically chooses the number of hash functions based on the data to be inserted.
- Warp is a state-of-the-art warp-centric approach for GPU-based hash tables [19]. Warp employs a linear probe approach to handle hash collisions.
- MegaKV is a warp-centric approach for GPU-based key value store published in [36]. MegaKV employs a cuckoo hash with two hash functions and it allocates a bucket for each hash value.
- Slab is the state-of-the-art GPU-based dynamic hash table [4], which employs chaining and a dedicated memory allocator for resizing.
- DyCuckoo is the approach proposed in this paper.

We adopt the implementations of the compared baselines from their corresponding inventors. The code for DyCuckoo is released [3]. Performance numbers for GPU-based solutions are calculated based purely on GPU run-time. The overhead of data transfer between CPUs and GPUs can be hidden by overlapping data transfer and GPU computation, as proposed by MegaKV [36]. Since this technique is orthogonal to the approaches proposed in our paper, we focus solely on GPU computation.

Datasets. We evaluate all compared approaches using several real world datasets. The summary of the datasets can be found in Table 2.

- TW: Twitter is an online social network where users perform the actions *tweet*, *retweet*, *quote*, and *reply*. We crawl these actions for one week through the Twitter stream API[4] for the following trending topics: US president election, 2016 NBA finals and Euro 2016. The dataset contains 50,876,784 KV pairs.
- RE: Reddit is an online forum where users perform the actions *post* and *comment*. We collect all Reddit *comment* actions in May 2015 from *kaggle* [5] and query the Reddit API for *post* actions during the same period. The dataset contains 48,104,875 KV pairs.
- LINE: Lineitem is a synthetic table generated by the TPC-H benchmark[6]. We generate 100,000,000 rows of the lineitem table and combine the *orderkey*, *linenumber* and *partkey* column as keys.
- RAND: Random is a synthetic dataset generated from a normal distribution. We have deduplicated the data and generated 100,000,000 KV pairs.
- COM: Databank is a PB-scale data warehouse that stores Alibaba customer behavior data for 2017. Because of confidentiality concern, we sample 10,000,000 transactions and the dataset contains 4,583,941 encrypted customer IDs as KV pairs.

Static Hashing Comparison (Section 6.3). We evaluate **INSERT** and **FIND** performance among all compared approaches under a static setting. We insert all KV pairs from the datasets and then issue 1 million random search queries.

Dynamic Hashing Comparison (Section 6.4). We generate workloads under the dynamic setting by batching hash table operations. We then partition the datasets into batches of 1 million insertions. For each batch, we augment 1 million **FIND** operations and $1 \cdot r$ million **DELETE** operations, where r is a parameter for controlling insertions and deletions. After exhausting all the batches, we rerun the batches by swapping the **INSERT** and **DELETE** operations in each batch. In other words, we issue 1 million **DELETE** operations of the keys inserted, 1 million **FIND** operations and $1 \cdot r$ million **INSERT** operations. We then evaluate the performance of all compared GPU approaches except for CUDPP and Warp as they do not support deletions. We also exclude the CPU approach as it is significantly slower than the GPU-based approaches according to Figure 9. Since MegaKV does not provide dynamic resizing, we double/half the memory usage followed by rehashing all KV pairs as a resizing strategy if the corresponding filled factor falls out of the specified range. Moreover, if an insertion failure is found for a compared approach, we trigger its resizing strategy.

Parameters. We vary the parameters when comparing DyCuckoo with the baselines. Here, α represents the lower bound for filled factor θ for all compared approaches, β is the respective upper bound, and r is the ratio of insertions over deletions in a processing batch. The parameter settings are listed in Table 3. For all experiments, we use

3. https://github.com/pauline-ly/Dycuckoo
4. https://dev.twitter.com/streaming/overview
5. https://www.kaggle.com.reddit/reddit-comments-may-2015
6. https://github.com/electrum/tpch-dbgen

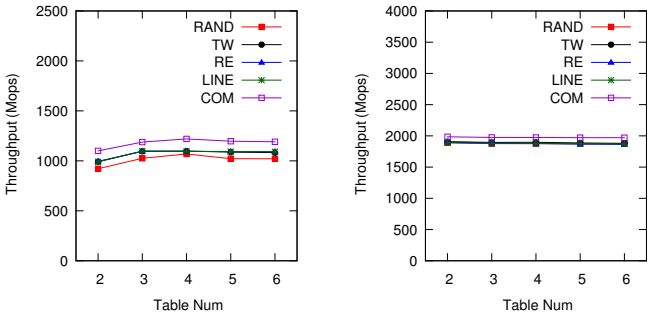

Fig. 7. Throughput of DyCuckoo for varying the number of hash tables.

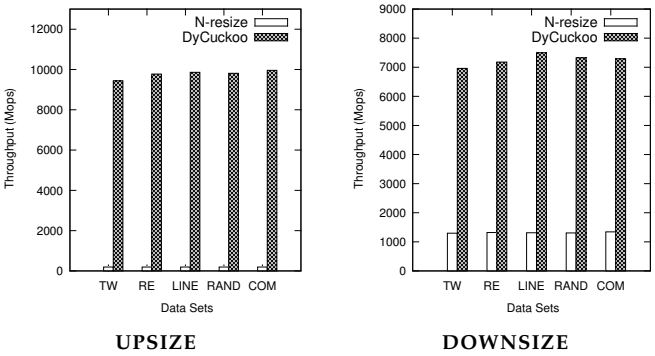

Fig. 8. Throughput of subtable resize.

million operations/seconds (Mops) as a metric to measure the performance of all compared approaches.

Experiment Environment. We conduct all experiments on an Intel Xeon E5-2682 Server equipped with an NVIDIA Tesla P100-PCIe GPU. Evaluations are performed using CUDA 9.1 on Centos 7.2. The optimization level (-O3) is applied for compiling all programs.

6.2 Sensitivity Analysis

Varying the number of tables. A key parameter affecting the performance of DyCuckoo is the number of hash tables chosen. For the static scenario, we present the throughput performances of INSERT and FIND for a varying number of hash tables, as shown in Figure 7, while fixing the entire structure's memory space ensure a default filled factor of θ. INSERT increases its throughput with more hash tables, since there are more alternative locations to relocate KV pairs. However, performance slightly degrades with more than four hash tables. This is because the total size of the allocated memory is fixed, and thus each table becomes smaller for more tables. This leads to more evictions for some overly occupied tables, thus slightly degrading the performance. Another interesting observation is that we achieve the best performance with the COM dataset. This is because COM has the smallest ratio of unique keys (Table 2). Inserting an existing key is equivalent to an update operation, which has better performance than inserting a new key. The throughput of FIND remains constant for additional hash tables as the two-layer cuckoo hashing guarantees a maximum of two look ups for FIND. In the remainder part of this section, we fix the number of hash tables at four.

TABLE 4
The number bucket accessed for subtable resizing (x1000).

	UPSIZE		DOWNSIZE	
	DyCuckoo	Rehash	DyCuckoo	Rehash
TW	4740	5642	628	629
RE	4739	5653	628	629
LINE	4746	5659	628	629
COM	4735	5649	628	629
RAND	4731	5647	628	629

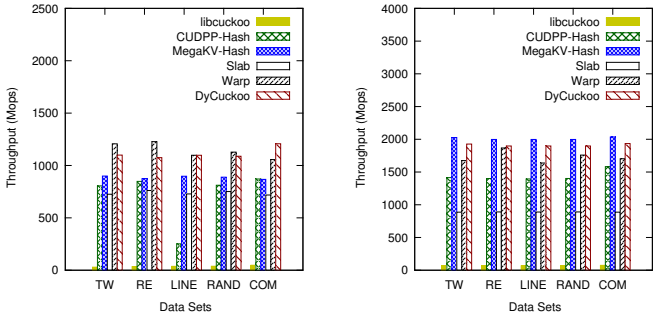

Fig. 9. Throughput of all compared approaches under the static setting.

Resizing analysis. To validate the proposed resizing strategy's effectiveness, we compare it with rehashing. For upsizing, we initialize DyCuckoo with all data and set the filled factor as the default upper bound of 85%. We perform a one time upsizing, i.e., we upsize one subtable, and then compare our resizing strategy against rehashing all subtable entries by reinserting the entries using Algorithm 1. The setup for downsizing is a mirror image of upsizing, except with an initial filled factor set as the default lower bound of 30%. We record the number of buckets accessed to process all resizing strategies in Table 4. First, downsizing accesses less buckets than that of upsizing as there are smaller number of KV pairs to be moved for downsizing (total size of the allocated table is fixed). Furthermore, DyCuckoo and rehashing access similar number of buckets for downsizing as the table is mostly empty at the filled factor of 30%. Second, DyCuckoo accesses smaller number of buckets than that of rehashing in the upsize scenario. This is because the tables are almost filled for upsize at 85%, which leads to more cuckoo evictions for rehashing whereas no eviction required for DyCuckoo. In Figure 8, we also measure the throughput of both methods as the number of KV pairs to be moved over the time to complete resizing. The throughput advantage of DyCuckoo cannot be explained solely by the smaller number of buckets accessed for resizing. In fact, DyCuckoo does not need to lock any buckets for upsizing and only needs to lock a small number of buckets for downsizing when eviction, whereas rehash needs to lock every accessed buckets. The experimental results confirm the superiority of our proposed resizing strategy.

6.3 Static Hashing Comparison

Throughput Analysis. In Figure 9 shows the throughput of all compared approaches over all datasets under default settings. The GPU-based approaches outperform the CPU-based baseline with a wide margin. For INSERT,

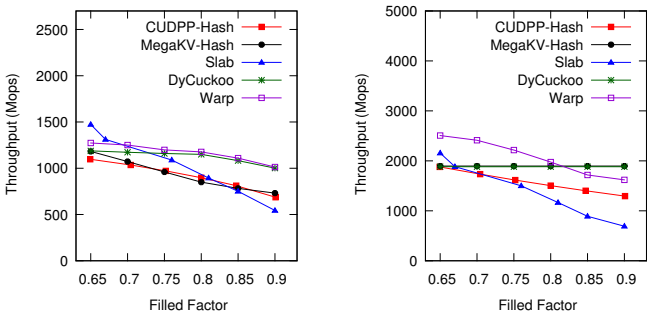

Fig. 10. Throughput of all compared approaches for varying the filled factor against the RAND dataset.

Warp demonstrates the best performance overall, because Warp employs a linear probing strategy that achieves better cache locality than the cuckoo strategy [19]. Nevertheless, DyCuckoo shows competitive throughput. For **FIND**, MegaKV shows the best performance at the default filled factor since it simply checks two buckets for locating a KV pair. Although DyCuckoo also checks two buckets, it has slightly inferior performance to MegaKV because DyCuckoo employs another layer of hashing that adds cost to the overall performance. Slab employs a chaining approach; therefore it requires more random accesses to locate a KV pair along the chain when a high filled factor is required. Hence, Slab has inferior performance to other GPU-based solutions, except for low filled factors.

Varying filled factor θ. We vary the filled factor θ and show the performance of all GPU-based approaches against the RAND dataset in Figure 10. The other datasets show similar trends, thus we omit those results in this paper. For Slab, the filled factor dramatically affects the performance of both **INSERT** and **FIND**. This is because Slab employs a chaining approach, in which a high filled factor leads to long chains and poor performance. Overall, Warp demonstrates superior performance for the static setting. As previously mentioned, Warp has better cache locality than the other approaches. DyCuckoo shows competitive performance and could even outperform Warp for high filled factors, e.g., $\theta \geq 0.85$. Furthermore, the linear probing strategy Warp adopts may need to scan multiple memory positions for **FIND** and **DELETE**. Although CUDPP also employs the cuckoo strategy, it automatically determines the number of hash tables for a given data size. At high filled factors, CUDPP employs more hash tables and scans more buckets; hence, its performance decreases for high filled factors. In contrast, DyCuckoo and MegaKV use the cuckoo strategy and only inspect two buckets, which explains the stable throughput shown in Figure 10 for **FIND**. Furthermore, DyCuckoo shows better insertion performance than MegaKV because our proposed strategy efficiently resolves conflicts.

6.4 Dynamic Hashing Comparison

Varying insert vs. delete ratio r. In Figure 11, we report the results for varying the ratio r, which is number of deletions over the number of insertions in a batch. We found that for a larger value of r, the performance of DyCuckoo and MegaKV degrades whereas that of Slab improves. A larger value of r indicates more deletions, thus resizing operations are more frequently invoked for DyCuckoo and MegaKV. In contrast, a greater number of deletions leads to additional vacant spaces for Slab, as that technique simply symbolically marks a deleted KV pair. Insertions are processed more efficiently for Slab because the inserted KV pairs can overwrite symbolically deleted ones. Hence, Slab utilizes more GPU device memory than DyCuckoo and MegaKV. However, symbolic deletions cannot guarantee a bounded filled factor and may lead to arbitrary bad memory efficiency. This will be discussed with more experimental results later in this section. DyCuckoo shows the best overall performance. Furthermore, the throughput margin between DyCuckoo and MegaKV increases for larger r values. As mentioned previously, a larger r triggers more resizing operations, thus DyCuckoo is more efficient than MegaKV as MegaKV employs a total rehashing approach.

Performance stability. We evaluate the performance stability of the compared approaches in Figure 12. In particular, we track the filled factor after processing each batch. Slab shows good stability in terms of memory usage for the starting phases. Unfortunately, due to the symbolic deletion approach employed, the memory efficiency of Slab degrades significantly as more deletions are processed. In particular, its filled factor drops to less than 20% after processing fewer than 100 batches for the COM dataset, which deems a complete rebuild. MegaKV shows an unstable trend since it employs a simple double/half approach for resizing. We can see that its filled factor jumps up or down dramatically at resizing points. DyCuckoo demonstrates the best overall stability and saves up to 4x memory over the compared baselines (the COM dataset). These results have validated our design, with only one of the subtables being subject to resizing. Nevertheless, there is still rooms for improvement. We note that for some datasets, i.e., TW, RE, LINE, and RAND, the filled factor of DyCuckoo drops sharply. This is because, even after upsizing one time, the insertions fail due to too many evictions, which triggers another round of upsizing. We leave this as an area for future work.

Varying the batch size. We also varied the size of each processing batch. The results are reported in Figure 13. Slab continues to show inferior performance to MegaKV and DyCuckoo. This is because Slab accommodates new inserted KV pairs with the chaining approach and does not increase the range of the unique hash values. Hence, a stream of insertions will eventually lead to long chains, which hurts the performance of hash table operations. Furthermore, DyCuckoo presents better performance than MegaKV and the margin increases with a larger batch size. Note that one limitation of existing GPU-based approaches is that they apply updates at the granularity of batches. It is an interesting direction for exploring efficient GPU hashing when a required update order is enforced.

Varying the filled factor lower bound α. We vary the lower bound of the filled factor and report the results in Figure 14. We only compare MegaKV and DyCuckoo since Slab is unable to control the filled factor because of Slab's symbolic deletion approach. Apparently, the simple resizing strategy adopted by MegaKV incurs substantial overhead. Such overhead grows for a higher α since the number of

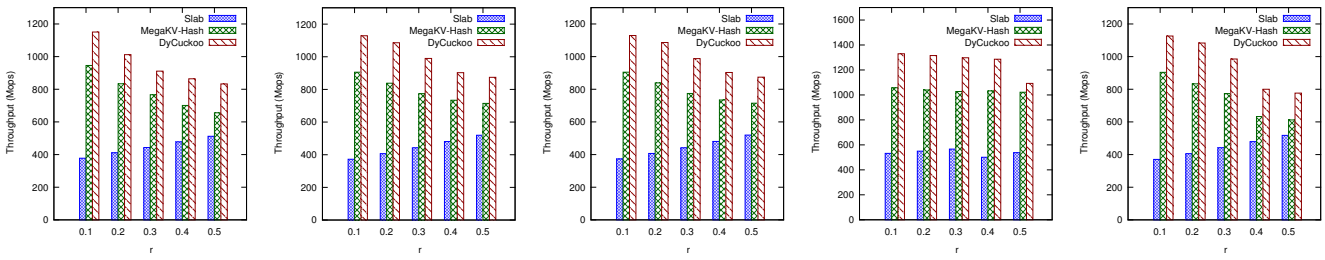

Fig. 11. Throughput for varying the ratio r.

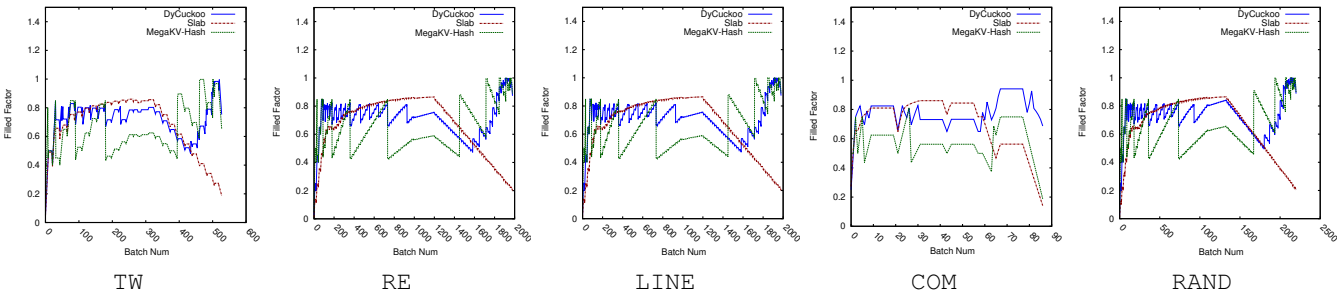

Fig. 12. Tracking the filled factor.

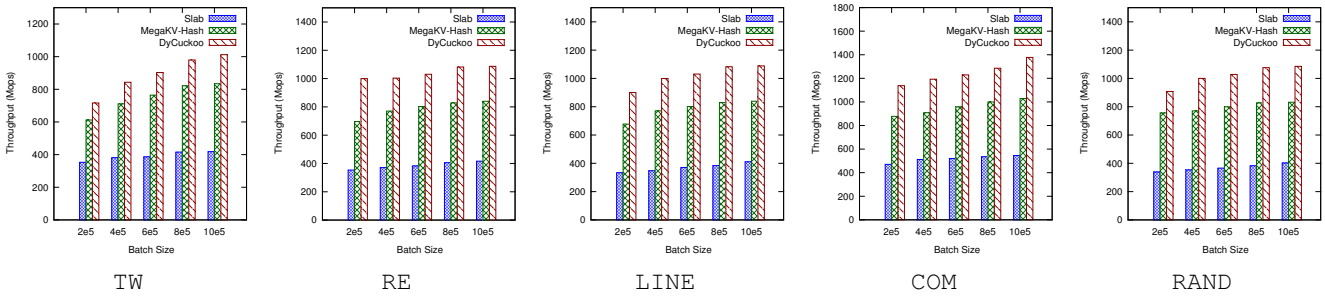

Fig. 13. Throughput for varying Batch Size .

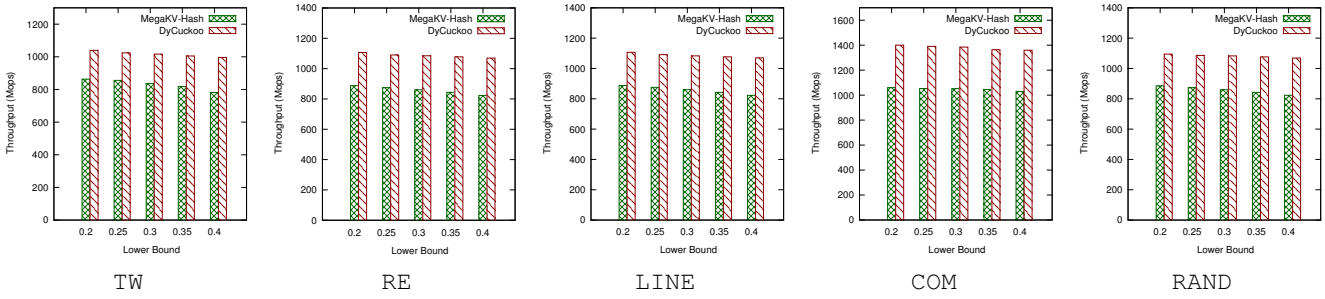

Fig. 14. Throughput for varying α .

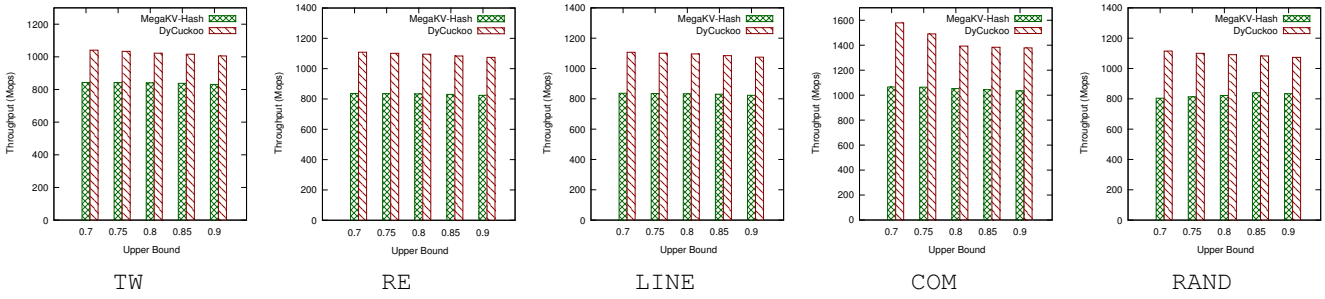

Fig. 15. Throughput for varying β .

downsizings increases. The performance of DyCuckoo is not affected significantly due to the incremental resizing approach by updating only one subtable at a time.

Varying the filled factor upper bound β. The results for varying β is reported in Figure 15. It is interesting to see that the upper bound does not significantly affect the overall performance for MegaKV and DyCuckoo. On one hand, a higher filled factor leads to slower INSERT performance. On the other hand, smaller number of rehashing is incurred for a higher filled factor. Thus, the overall performance remains stable as the opposing factors cancel each other. Nevertheless, DyCuckoo remains superior over MegaKV in terms of throughput while saves GPU memory substantially.

7 CONCLUSION

In this paper, we contribute a number of novel designs for dynamic hash table on GPUs. First, we introduced an efficient strategy to resize only one of the subtables at a time. Our theoretical analysis demonstrated the near-optimality of the resizing strategy. Second, we devised a two-layer cuckoo has scheme that ensures a maximum of two lookups for find and deletion operations, while still retaining similar performance for insertions as general cuckoo hash tables. Empirically, our proposed design achieves competitive performance against other state-of-the-art static GPU hash table techniques. Our hash table design achieves superior performance while saving up to four times the memory over state-of-the-art approaches against dynamic workloads.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the gpu. *TOG*, 28(5):154, 2009.

[3] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Building an efficient hash table on the gpu. In *GPU Computing Gems Jade Edition*, pages 39–53. Elsevier, 2011.

[4] S. Ashkiani, M. Farach-Colton, and J. D. Owens. A dynamic hash table for the gpu. In *IPDPS*, pages 419–429. IEEE, 2018.

[5] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *GPGPU*, pages 94–103. ACM, 2010.

[6] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[7] P. A. Boncz, S. Manegold, M. L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[8] J. Bowers, R. Wang, L.-Y. Wei, and D. Maletz. Parallel poisson disk sampling with spectrum analysis on surfaces. *TOG*, 29(6):166, 2010.

[9] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen. Horton tables: Fast hash tables for in-memory data-intensive computing. In *ATC*, pages 281–294, 2016.

[10] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *ICML*, pages 1337–1345, 2013.

[11] J. R. Douceur. Hash table expansion and contraction for use with internal searching, May 23 2000. US Patent 6,067,547.

[12] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

[13] I. García, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel hashing. In *TOG*, volume 30, page 161. ACM, 2011.

[14] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *TODS*, 34(4):21, 2009.

[15] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524. ACM, 2008.

[16] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[17] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. Memcachedgpu: Scaling-up scale-out key-value stores. In *SOCC*, pages 43–57. ACM, 2015.

[18] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and gpu. In *PACT*, pages 217–226. ACM, 2010.

[19] D. Jünger, C. Hundt, and B. Schmidt. Warpdrive: Massively parallel hashing on multi-gpu nodes. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 441–450. IEEE, 2018.

[20] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *DaMoN*, pages 55–62. ACM, 2012.

[21] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. In *PACT*, pages 63–74. IEEE, 2015.

[22] R. Kutzelnigg. Bipartite Random Graphs and Cuckoo Hashing. In *Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities*, pages 403–406. Discrete Mathematics and Theoretical Computer Science, 2006.

[23] P.-A. Larson, M. R. Krishnan, and G. V. Reilly. Scaleable hash table for shared-memory multiprocessor system, June 10 2003. US Patent 6,578,131.

[24] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[25] Y. Liu, K. Zhang, and M. Spear. Dynamic-sized nonblocking hash tables. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 242–251. ACM, 2014.

[26] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *TOG*, 32(6):169, 2013.

[27] K. J. O'Dwyer and D. Malone. Bitcoin mining and its energy footprint. 2014.

[28] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[29] J. Pan, C. Lauterbach, and D. Manocha. Efficient nearest-neighbor computation for gpu-based motion planning. In *IROS*, pages 2243–2248. IEEE, 2010.

[30] J. Pan and D. Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *SIGSPATIAL*, pages 211–220. ACM, 2011.

[31] M. Raab and A. Steger. "balls into bins"—a simple and tight analysis. In *RANDOM*, pages 159–170. Springer, 1998.

[32] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301. IEEE, 2007.

[33] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[34] M. B. Taylor. Bitcoin and the age of bespoke silicon. In *CASES*, page 16. IEEE Press, 2013.

[35] Z. Wu, Y. Liu, J. Sun, J. Shi, and S. Qin. Gpu accelerated on-the-fly reachability checking. In *ICECCS*, pages 100–109. IEEE, 2015.

[36] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores. *PVLDB*, 8(11):1226–1237, 2015.

[37] Z. Zhang, H. Deshmukh, and J. M. Patel. Data partitioning for in-memory systems: Myths, challenges, and opportunities. In *CIDR*, 2019.

[38] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *TPDS*, 25(6):1543–1552, 2014.

[39] J. Zhou, K.-M. Yu, and B.-C. Wu. Parallel frequent patterns mining algorithm on gpu. In *SMC*, pages 435–440. IEEE, 2010.

[40] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 461–476, 2018.

Yuchen Li is an assistant professor at the School of Information Systems, Singapore Management University (SMU). He received the double B.Sc. degrees in applied math and computer science (both degrees with first class honors) and the Ph.D. degree in computer science from NUS, in 2013 and 2016, respectively. He received the Lee Kong Chian Fellowship in 2019 for research excellence. His research interests include heterogeneous computing, graph analytics and computational journalism.

Jing Zhang is currently a graduate student at the College of Computer Science and Technology, Zhejiang University (ZJU). He received his B.Sc. from Huazhong University of Science and Technology (HUST) in 2017. His research interests include heterogeneous computing and data management.

Yue Liu is currently a graduate student at the College of Computer Science and Technology, Zhejiang University (ZJU). She received her B.Eng. in the Internet of Things from Hunan University (HNU) in 2017. Her research interests include heterogeneous computing and data management.

Zheng Lyu is now a staff engineer at Alibaba group, and responsible for development of GPU databases. He received his PhD in communication and information system from Shanghai institute of microsystem and information technology, Chinese Academy of Sciences in 2012. He mainly works in the area of high performance computing and his major research interest is heterogeneous computing in database system.

Zhongdong Huang is an associate professor in the College of Computer Science and Technology, Zhejiang University. He received his B.Sc in Telecommunication and PhD degree in Computer Science from Zhejiang University in 1989 and 2003 respectively. His research interests include big data analytics and database systems.

Jianling Sun is a professor at the School of Computer Science and Technology. He received his PhD degree in computer science from Zhejiang University, China in 1993. His research interests include database systems, distributed computing, and machine Learning. He currently serves as the director of the Lab of Next Generation Database Technologies of Alibaba-Zhejiang University Joint Institute of Frontier Technologies.