DyCuckoo: Dynamic Hash Tables on GPUs

Yuchen Li*†, Qiwei Zhu†, Zheng Lyu#, Zhongdong Huang†, Jianling Sun†
*Singapore Management University
yuchenli@smu.edu.sg
†Zhejiang University
†{zhuqiweiyx, hzd, sunjl}@zju.edu.cn
#Alibaba Group
#lvzheng.lz@alibaba-inc.com

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00070

***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 occupy 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 2-in-d 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 efficiency while enables fine-grained memory control, which is not available in existing GPU hash table approaches.**

I. 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 [1], [2], machine learning [3], [4], network analysis [5], [6] and database technologies [7], [8]. 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 [9], [10], [11], data mining [12], [13], [14], key-value stores [15], [16], [17], and many others [18], [19], [20], [21], [22]. Existing works [23], [15], [24], [16], [17] 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 in web server logs and telephone calls. When data size varies, the static allocation strategy leads to poor memory utilization [25]. 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 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 [26], [27] and industry [28], [29] 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 [30] 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 [23], [31], [15], [17] provide solutions in speeding up applications using parallel cuckoo hashes on GPUs. However, complete relocation of the entire hash table is required when the data cannot be inserted. To avoid complete relocation, we propose two novel designs for implementing dynamic cuckoo hash tables on GPUs.

First, we employ the cuckoo hashing scheme with d subtables where each subtable is configured by a hash function, 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 handle subsequent resizing efficiently. 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 2-in-d cuckoo hashing scheme to ensure efficient hash table operations. The proposed resizing strategy requires d hash tables, which indicates d 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, the 2-in-d 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 up to two lookups for any find and deletion operations. Furthermore, the 2-in-d approach retains the general cuckoo hash tables' performance guarantee. Empirically, the proposed hash table design can operate efficiently and consistently at a filled factor of 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 2-in-d 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 enables fine-grained memory control, which is not available in existing approaches.

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

II. PRELIMINARIES

In this section, we first introduce some preliminary information on general hash tables and present background material on GPU architecture.

A. Hash Table

The hash table is a fundamental data structure that stores key-value (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 **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 [30] 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) independent hash functions $h^1, h^2, \dots, h^d$ and stores a KV pair in a position corresponding to *one* of the hash functions. 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.

Existing implementation practice for cuckoo hash is to allocate a single hash table for d hash functions [32]. However, such approach would trigger a complete rebuild when the hash table is subject to either upsize or downsize operations for dynamic workloads. In this work, we allocate d hash tables where each hash table is configured by a unique hash function. 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 $\theta = (\sum_i m_i)/(\sum_i n_i)$.

B. GPU Architecture

We introduce the background of the NVIDIA GPU architecture in this paper because of 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. 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 a large number of CUDA cores. 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.

III. Related Works

Alcantara *et al.* [23] 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.* [20] investigated the use of GPU-based cuckoo hashing for on-the-fly model checking. A proposal for accelerating the nearest neighbor search is presented in [18]. To improve on [23], stadium hash was proposed in [34] to support out-of-core GPU parallel hashing by building signatures for hash buckets to enable early search termination. However, this technique uses double hashing which has to rebuild the entire table for any deletions. Zhang *et al.* [15] proposed another efficient design of GPU-based cuckoo hashing, named MegaKV, to boost the performance for KV store. Subsequently, Horton table [17] improves the efficiency of **FIND** over MegaKV by trading with the cost of **INSERT** by introducing a KV remapping mechanism. WarpDrive [35] 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], [14].

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.

To the best of our knowledge, there is only one existing work on building dynamic hash tables on GPUs [25]. 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 [25], 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. Such approach not only results in degraded performance for **FIND**, but 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.

IV. 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 IV-A. Subsequently, the resizing strategy is introduced in Section IV-B. In Section IV-C, 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 IV-D.

A. Hash Table Structure

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 III, 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, 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 [36].

Figure 1 depicts the design of a single hash table h^i on GPUs. The keys are assumed to be 4-byte integers; a bucket of 32 keys, which are all hashed to the same value h^i_j, 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 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

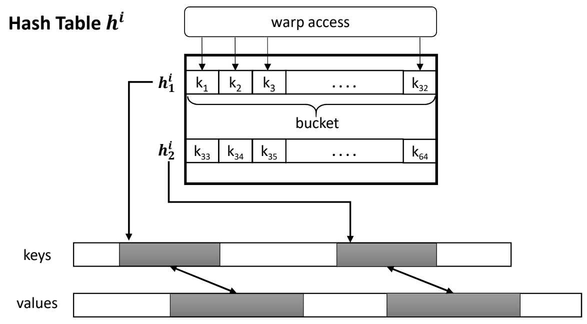
Hash Table h^i
warp access
h_1^i
k_1 k_2 k_3 .... k_{32}
bucket
h_2^i
k_{33} k_{34} k_{35} .... k_{64}
keys
values

Fig. 1. The hash table structure.

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.

B. 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. 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 $[\alpha, \beta]$. As a result, 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.

Example 1. *In Figure 2, we show a simplify example of the cuckoo hash with 4 hash tables. We only present the keys and omit the values for ease of presentation. Each bucket can hold at most two KV pairs. The current filled factor is* $\frac{22}{32} = 0.69$*. An upsize doubles one of the hash table, which increases the filled factor to* $\frac{22}{40} = 0.55$*, whereas a downsize halves one hash table only, which decreases the filled factor to* $\frac{22}{28} = 0.79$.

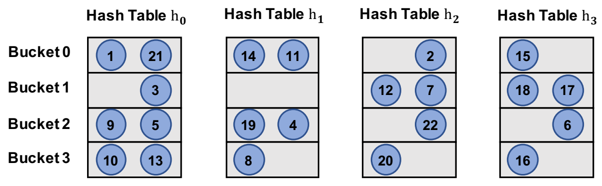

Fig. 2. An example of four hash tables.

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 lowers the filled factor to $\frac{\theta \cdot (d+d')}{d+d'+1} \geq \frac{\beta \cdot d}{d+1}$ as $\theta > \beta$ triggers the upsize. 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.

C. 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:

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

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

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

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

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

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.,

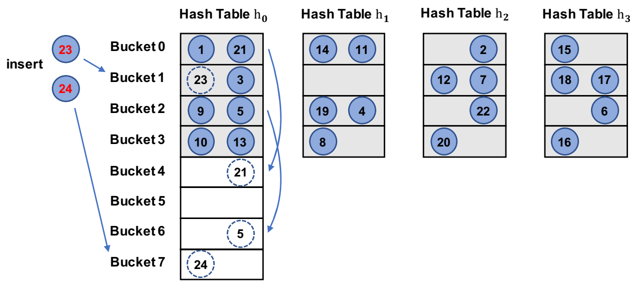

Fig. 3. Illustration for upsize.

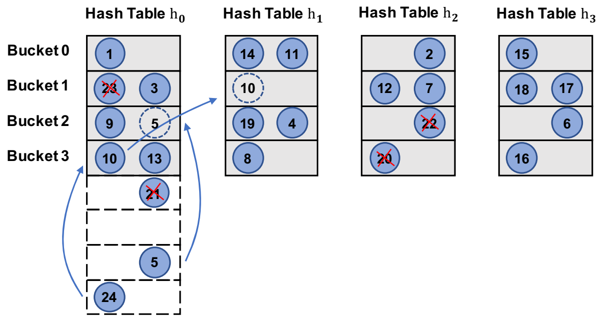

Fig. 4. Illustration for downsize.

$\theta_i = \theta_j$ 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. To ensure the KVs are evenly distributed, we use randomized strategy for table assignment.

D. *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. 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.

Example 2. *In Figure 3, we continue from Example 1 to show a case of upsize. Suppose inserting KV pairs with keys* 23 *and* 24 *triggers an upsize, we first double* h_0 *and rehash all KV pairs in* h_0*, followed by inserting* 23 *and* 24 *into* h_0*. The rehash does not generate conflicts as there is always room to reallocate the KV pairs in* h_0*.*

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, it does not 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, we employ two GPU kernels to handle them separately.

Example 3. *We continue from Example 2 to show a case of downsize. Suppose deleting KV pairs in Figure 4 triggers a downsize, we half* h_0 *and rehash all KV pairs in* h_0*. Note that when* 24 *is rehashed to bucket* 3*, it will evict* 10*, which is then reallocated to* h_1*.*

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 $O(1)$.

V. 2-in-d Cuckoo Hash

In this section, we present a 2-in-d approach that ensures a maximum of two lookups for **FIND** and **DELETE** (Section V-A). Subsequently, in Section V-B, we give details on optimizing GPUs for paralleling hash table operations such as **FIND**, **INSERT**, and **DELETE**.

A. *The 2-in-d 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 IV-B. Nevertheless, the benefit of employing more tables does not come for free. For each **FIND** and **DELETE** operations, one must perform d lookups, which are translated 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.

Hence, we propose a 2-in-d approach for efficient lookup processing where at most two random accesses are needed. Given d hash tables where each is configured by one hash function, we insert a KV pair (k, v) by first selecting only two out of d hash tables. The selection is achieved by hashing the key k to the range $\{0, 1, \ldots, \binom{d}{2}\}$ where each hash value indicates a unique pair of hash tables. Subsequently, (k, v) will be inserted into one table among the selected hash table pair with the standard cuckoo hash scheme. In this way, we perform at most two lookups when searching for (k, v). Note that there is little overhead for the selection process as we only compute a hash value of k and map to the corresponding pair of hash tables with a small lookup table of size $O(\binom{d}{2})$. The following example demonstrates how to insert a KV pair with the 2-in-d approach.

Example 4. *A KV pair (k, v) first selects a hash table pair (h_i, h_j). We then hash k and try to insert (k, v) into h_i. Assuming the corresponding bucket in h_i is full, we evict another KV pair (k', v'). We then discover that (k', v') is mapped to the hash table pair (h_i, h_t). Then, 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. Furthermore, the 2-in-d cuckoo hash has the same asymptotic insertion performance as a plain cuckoo hash table with two hash functions.

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

Proof. Assuming d hash tables for the 2-in-d 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 2-in-d 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 [37], to prove Theorem 2. $\square$

B. Parallel Hash Table Operations

In the remainder of this section, we discuss how to utilize GPUs for the 2-in-d cuckoo hash. Following existing works [23], [15], [17], we assume that the **FIND**, **INSERT** and **DELETE** operations are batched and that each batch contains only one type of operations. For **INSERT**, we focus on presenting how to insert a KV pair into one hash table. When an eviction occurs, we follow the 2-in-d approach to find another hash table for inserting the evicted pair as discussed in Section V-A.

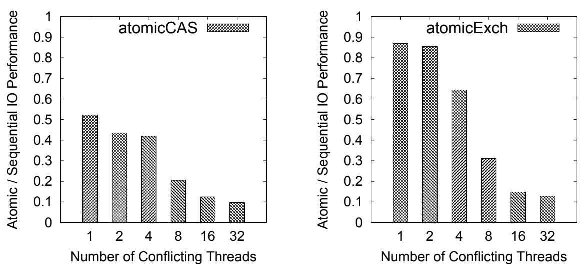

Fig. 5. The performance of atomic operations for increasing conflicts.

Find. It is relatively 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 (k, v), we first select the hash table pair (h_i, h_j) that corresponds to k 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 [15]. 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, and thus a large number of conflicts would happen.

Algorithm 1 Insert(lane l, warp wid)

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

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.

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 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 $\_\_shfl$ [1]. 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

[1]https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

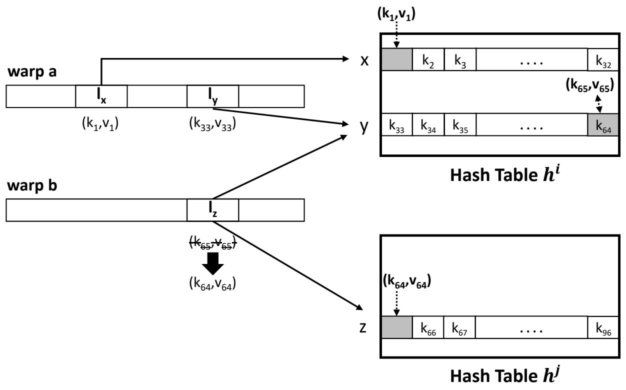

Fig. 6. Example for parallel insertions.

pair (k^*, v^*) in the bucket and inserts the evicted KV pair to hash table h_{j^*} of k^* 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 parallel insertion:

Example 5. *In Figure 6, we visualize a scenario for three threads: 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

TABLE I
THE DATASETS USED IN THE EXPERIMENTS.

Datasets	KV pairs	Unique keys
TW	50,876,784	25,297,548
RE	96,209,750	82,933,364
LINE	100,000,000	90,319,761
COM	10,000,000	4,583,941
RAND	100,000,000	100,000,000

TABLE II
THE PARAMETERS IN THE EXPERIMENTS.

Parameter	Settings	Default
Filled Factor θ	70%, 75%, 80%, 85%, 90%	85%
Lower Bound α	30%, 35%, 40%, 45%, 50%	50%
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

are independently executed by threads, the analysis of a single thread's complexity is the same as in the sequential version of cuckoo hashing [30]: $O(1)$ 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 [23]. 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 VI.

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 [38], the complexity of which is $O(\binom{m}{2}/H)$. Given that GPUs have many threads, there could be a significant amount of conflicts. Therefore, we propose the voter coordination scheme to reduce the cost of spinning in locks.

VI. 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. Section VI-A introduces the experimental setup. Section VI-B presents a discussion on the hash table setting of DyCuckoo. In Sections VI-C and VI-D, we compare all approaches under the static and the dynamic experiments, respectively.

A. Experimental Setup

Baselines. We compare DyCuckoo with both static and dynamic hash table approaches on GPUs as follows:

- Libcuckoo is a well-established CPU-based concurrent hash table that parallelizes a cuckoo hash [39].
- CUDPP is a popular CUDA primitive library containing the cuckoo hash table implementation published in [23]. 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 warp-centric approach for GPU-based hash tables [35]. Warp employs a linear probe approach to handle hash collisions. We set the group size of Warp to 4 as it is the best overall setting [35].
- MegaKV is a warp-centric approach for GPU-based key value store published in [15]. 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 [25], 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 [2]. 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 in MegaKV [15]. Since this technique is orthogonal to our proposed approaches, 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 I. We use the unique KV pairs from the datasets.

- 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[3] for the following trending topics: US presidential 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* [4] and query the Reddit API for *post* actions during the same period. The dataset contains 96,209,750 KV pairs.
- LINE: Lineitem is a synthetic table generated by the TPC-H benchmark[5]. We generate 100,000,000 rows of the lineitem table and combine the *orderkey*, *linenumber* and *partkey* column as keys.
- COM: Databank is a PB-scale data warehouse that stores Alibaba customer behavior data for 2017. Because of confidentiality concerns, we sample 10,000,000 transactions and the dataset contains 4,583,941 encrypted customer IDs as KV pairs.
- RAND: Random is a synthetic dataset generated from a normal distribution. We have deduplicated the data and generated 100,000,000 KV pairs.

Static Hashing Comparison (Section VI-C). 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 queries to search for all KV pairs.

[2]https://github.com/zhuqiweigit/DyCuckoo
[3]https://dev.twitter.com/streaming/overview
[4]https://www.kaggle.com.reddit/reddit-comments-may-2015
[5]https://github.com/electrum/tpch-dbgen

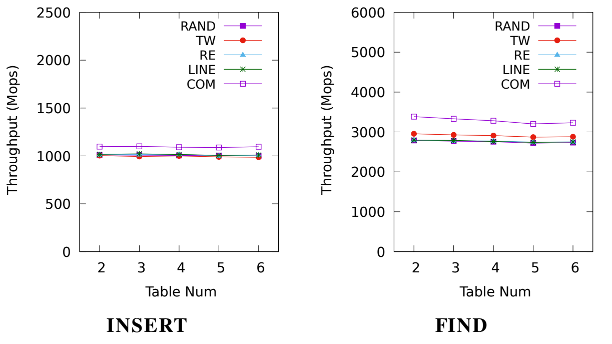

Fig. 7. Throughput of DyCuckoo for varying the number of hash tables.

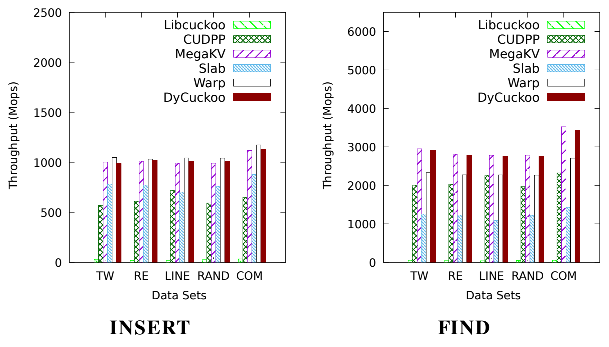

Fig. 8. Throughput of all compared approaches under the static setting.

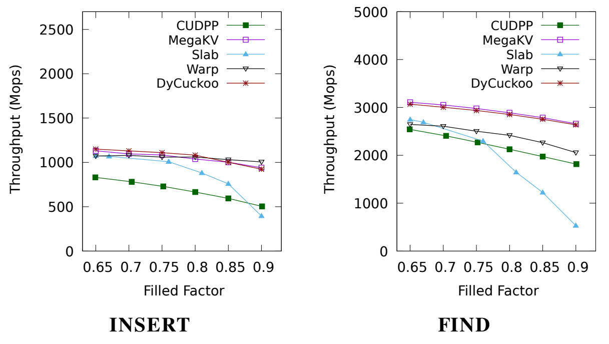

Fig. 9. Throughput of all compared approaches for varying the filled factor against the RAND dataset.

Dynamic Hashing Comparison (Section VI-D). We generate workloads under the dynamic setting by batching hash table operations. We partition the datasets into batches of 1 million insertions. For each batch, we augment 1 million **FIND** operations and r million **DELETE** operations, where r is a parameter for controlling the ratio between 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 r million **INSERT** operations. The above process is repeated for five runs. We only evaluate the performance of Slab and DyCuckoo because they are the methods that can support dynamic insertions, deletions, and resizing. For DyCuckoo, if an insertion failed, the upsize strategy will be triggered.

Parameters. α represents the lower bound for the filled factor θ, β is the upper bound, and r is the ratio of deletions over insertions in a processing batch as introduced in the dynamic setting. We also vary the batch size from $2e5$ to $10e5$. The parameter settings and their defaults are listed in Table II. All parameters are set to their defaults in Table II unless stated otherwise. We use *million operations/seconds* (Mops) as a metric to measure the performance.

Experiment Environment. We conduct all experiments on an Intel Xeon(R) Gold 6140 CPU @ 2.30GHz server equipped with an NVIDIA Titan V GPU. Evaluations are performed using CUDA 11 on Ubuntu 18.04 LTS. The optimization level (-O3) is applied for compiling all programs.

B. Varying the Number of Tables

A key parameter in 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 ensures the default filled factor of θ at 85%. We note that a larger number of hash tables imply a more precise control on the filled factor. Given d hash tables, the percentage change of memory usage for upsize/downsize is at most $1/d$. According to our results reported in Figure 7, the performance numbers of both **INSERT** and **FIND** are not affected significantly by the number of hash tables. This is because, despite the increasing number of hash tables, the number of possible locations for any KV pair remains to be *two*. The results validate the superiority of our 2-in-d cuckoo hashing approach as it guarantees a stable performance while enables a fine-grained memory control for hash tables on GPUs. In the remaining part of this section, we fix the number of hash tables at four. Another interesting observation is that we achieve the best performance with the COM dataset. This is because COM has the smallest data size, which can partly fit into the L2 cache of GPUs (4.5MB for Titan V) and thus incurs fewer cache misses. Hence, ensuring economic memory usage on GPUs is also critical for efficiency optimization.

C. Static Hashing Comparison

Throughput Analysis. Figure 8 shows the throughput of all compared approaches over all datasets under default settings. On average, Libcuckoo has throughput of 25 Mops for **INSERT** and 46 Mops for **FIND**. The GPU-based approaches show at least one order of magnitude speedup over Libcuckoo (the bars for Libcuckoo are almost invisible). We will omit Libcuckoo for the remaining experiments as it is significantly inferior than other baselines. For **INSERT**, Warp, MegaKV and DyCuckoo show similar performance, while Warp has a slight advantage. This is because Warp employs a linear probing strategy that achieves slightly better cache locality than the cuckoo hash strategy. The performance of Slab and CUDPP is significantly slower than other compared methods. Slab employs a chaining approach and the performance can be severely affected when there is a long chain to insert. CUDPP is the slowest among all methods, despite it also employs a cuckoo hash approach as MegaKV and DyCuckoo. This is because CUDPP does not follow the warp-centric approach and under-utilizes the computing resources of GPUs. For **FIND**, MegaKV and DyCuckoo have clear advantages. This is because both methods only check at most two locations for **FIND**. In contrast, Warp and Slab may search for multiple locations to retrieve a KV pair. To support such an argument, we show the average number of probed locations for all compared methods in Table III. For MegaKV and DyCuckoo, the number of probed locations is close to 1, whereas the other methods have notably higher probes. Note that each search query can always find a KV in the hash table according to our setup. In the case of queries where the key is missing in the hash table, Warp and Slab will incur worse performance as there is a large number of probes required before they can detect that the key is missing. Lastly, the performance of CUDPP is again inferior to other cuckoo hash approaches due to its non-warp-centric approach.

TABLE III
THE AVERAGE NUMBER OF PROBED LOCATIONS FOR **FIND**.

	TW	RE	LINE	COM	RAND
CUDPP	2.07	1.99	1.79	2.08	2.05
Warp	1.30	1.30	1.30	1.30	1.30
MegaKV	1.14	1.14	1.14	1.14	1.14
Slab	3.04	3.04	3.45	3.05	3.05
DyCuckoo	1.14	1.14	1.14	1.14	1.14

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 9. 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, MegaKV and DyCuckoo demonstrate the best performance. Warp has a competitive performance of **INSERT** but falls short for **FIND**. As illustrated in Table III, Warp needs to probe more locations as it employs the linear probe approach to ensure cache efficiency. The experiments have confirmed that DyCuckoo is competitive against the state-of-the-art methods even for the static setting. MegaKV shares similar performance with DyCuckoo but it does not support dynamic resizing. In the remaining part of this section, we will demonstrate the superiority of DyCuckoo in the dynamic environment.

D. Dynamic Hashing Comparison

Varying insert vs. delete ratio r. In Figure 10, we report the results for varying the ratio r, which is the number of deletions over the number of insertions in a batch. We found that for a larger value of r, the performance numbers of both DyCuckoo and Slab improve. This is because, a larger deletion rate results in more empty locations in the hash tables, which are vacant for insertions. In terms of throughput, DyCuckoo can achieve between 2x-6x speedup compared with Slab. Furthermore, Slab employs a symbolic deletion approach and does not actually free the memory. This results in less memory efficiency as we discuss in the following.

Memory efficiency. We evaluate the memory efficiency of Slab and DyCuckoo in Figure 11. In particular, we track the filled factor after processing each batch. The trend shows a periodical pattern for both methods as we execute 5 runs of the same process described in the setup. We note that there are certain points where the filled factor becomes zero for all methods. This is because, at the end of each run, all KV pairs are deleted from the hash table. Slab shows efficient memory usage only at the starting phrases of each run. Unfortunately, due to the symbolic deletion approach employed, the memory efficiency of Slab degrades and fluctuates drastically as more deletions are processed, especially at the second half of each run. The reason behind such observation is that Slab does not physically free the memory occupied to favor insertions. Hence, it is unable to control the memory usage in a fine-grained manner. In contrast, DyCuckoo can keep the filled factor within the default range of $[0.5, 0.85]$ while achieving better throughput than Slab. The results have validated the superiority of DyCuckoo for precise memory utilization.

Varying the batch size. We also varied the size of each processing batch. The results are reported in Figure 12. A larger batch size can improve the GPU resource utilization and thus leads to a better throughput for both methods. Slab continues to show inferior performance to 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. There is an outlier point for LINE where the throughput of DyCuckoo drops at batch size $6e5$. We have examined the case and found that the outlier is caused by insertion failures. In the event of such failures, we will trigger additional upsize operations to accommodate the failed insertions. The overhead of such upsizes results in the outlier. We will develop a better failure handling mechanism as future work. 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 13. We only show the run time of DyCuckoo since Slab is unable to control the filled factor because of Slab's symbolic deletion approach. We profile the run-time by examining three components: upsize, downsize, and the rest of the computation. One can observe that downsize is more expensive than upsize. When performing downsize, the vacant locations are reduced and it becomes harder to insert the remaining KV pairs back into the reduced space. Another interesting observation is that the run-time for downsize increases slightly for a larger α. A better memory efficiency is guaranteed by a larger α but a smaller memory size will result in a slight degradation for insertions. Nevertheless, The overall performance of DyCuckoo is not affected significantly due to the incremental resizing approach by updating only one subtable at a time.

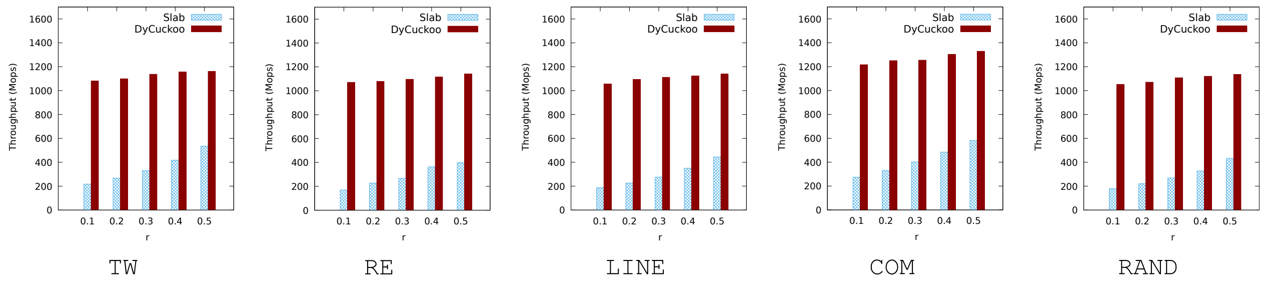

Fig. 10. Throughput for varying the ratio r.

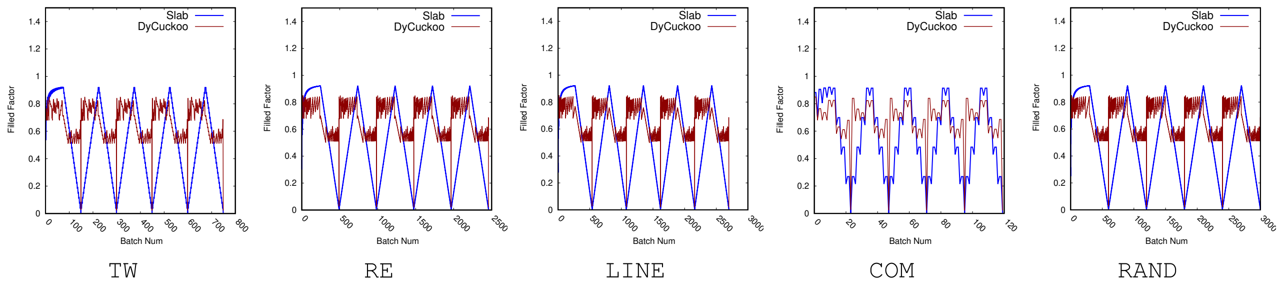

Fig. 11. Tracking the filled factor.

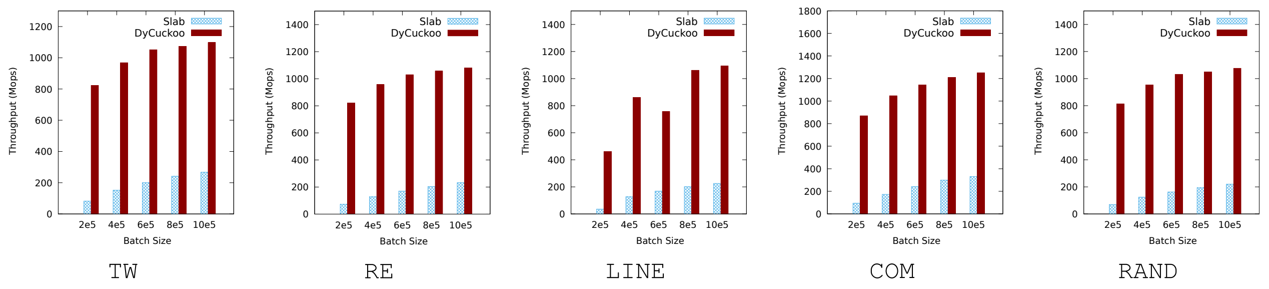

Fig. 12. Throughput for varying batch Size.

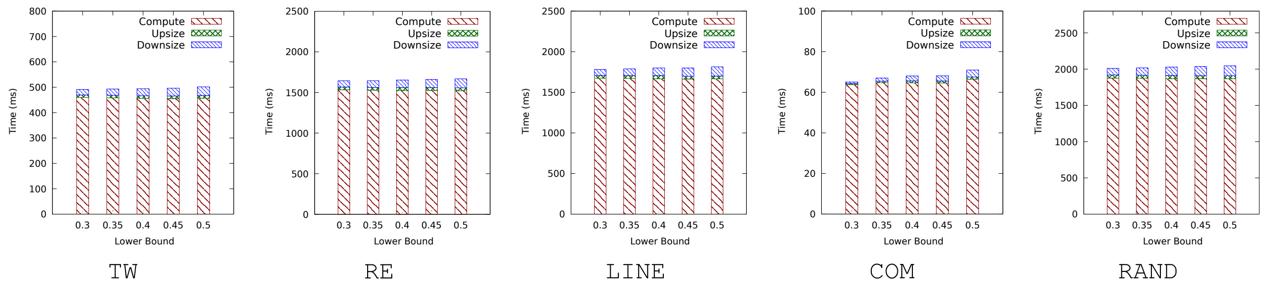

Fig. 13. Throughput for varying α.

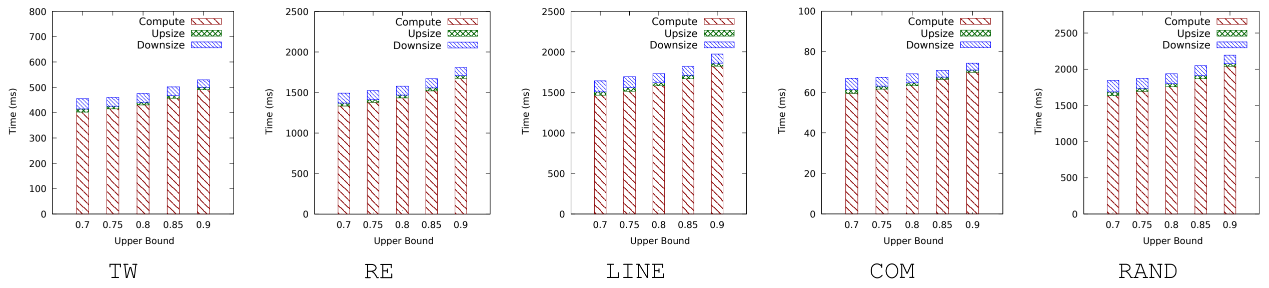

Fig. 14. Throughput for varying β.

Varying the filled factor upper bound β. The results for varying β is reported in Figure 14. When a higher filled factor is allowed, there are fewer vacant locations for insertions, which causes an increasing run-time. The time for both upsize and downsize is again small comparing with the rest of the computation. Furthermore, the resizing components show a stable performance with increasing upper bound values.

Scalability. We generate various samples of the `RAND` dataset by varying the data size for scalability test. The results are reported in Table IV. `DyCuckoo` shows sable and superior performance over `Slab`.

TABLE IV
THROUGHPUT (MOPS) WHEN VARYING THE DATASIZE OF RAND.

KV pairs	$2e7$	$4e7$	$6e7$	$8e7$	$10e7$
`Slab`	265	255	250	230	218
`DyCuckoo`	1085	1086	1074	1076	1082

VII. CONCLUSION

In this paper, we contribute a number of novel designs for the 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 2-in-d cuckoo hash 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 efficiency while enables fine-grained memory control, which is not available in existing GPU hash table approaches.

Acknowledgment. This work was supported by Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies.

REFERENCES

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

[2] M. B. Taylor, "Bitcoin and the age of bespoke silicon," in *CASES*, p. 16, IEEE Press, 2013.

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

[4] 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*, pp. 265–283, 2016.

[5] W. Guo, Y. Li, M. Sha, and K.-L. Tan, "Parallel personalized pagerank on dynamic graphs," *PVLDB*, vol. 11, no. 1, pp. 93–106, 2017.

[6] M. Sha, Y. Li, and K.-L. Tan, "Gpu-based graph traversal on compressed graphs," in *SIGMOD*, pp. 775–792, 2019.

[7] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *GPGPU*, pp. 94–103, ACM, 2010.

[8] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in *DaMoN*, pp. 55–62, ACM, 2012.

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

[10] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *TODS*, vol. 34, no. 4, p. 21, 2009.

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

[11] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *PVLDB*, vol. 6, no. 9, pp. 709–720, 2013.

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

[14] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *TPDS*, vol. 25, no. 6, pp. 1543–1552, 2014.

[15] 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*, vol. 8, no. 11, pp. 1226–1237, 2015.

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

[17] 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*, pp. 281–294, 2016.

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

[19] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3d reconstruction at scale using voxel hashing," *TOG*, vol. 32, no. 6, p. 169, 2013.

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

[21] M. Sha, Y. Li, B. He, and K. Tan, "Accelerating dynamic graph analytics on gpus," *PVLDB*, vol. 11, no. 1, pp. 107–120, 2017.

[22] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *SIGMOD*, pp. 1067–1082, 2020.

[23] 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*, vol. 28, no. 5, p. 154, 2009.

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

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

[26] Y. Liu, K. Zhang, and M. Spear, "Dynamic-sized nonblocking hash tables," in *PODC*, pp. 242–251, ACM, 2014.

[27] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *OSDI*, pp. 461–476, 2018.

[28] 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.

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

[30] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[31] 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*, pp. 39–53, Elsevier, 2011.

[32] K. A. Ross, "Efficient hash probes on modern processors," in *ICDE*, pp. 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] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan, "Stadium hashing: Scalable and flexible hashing on gpus," in *PACT*, pp. 63–74, IEEE, 2015.

[35] D. Jünger, C. Hundt, and B. Schmidt, "Warpdrive: Massively parallel hashing on multi-gpu nodes," in *IPDPS*, pp. 441–450, IEEE, 2018.

[36] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," *Theory of Computing Systems*, vol. 38, no. 2, pp. 229–248, 2005.

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

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

[39] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *EuroSys*, p. 27, ACM, 2014.