Dynamic Hash Tables on GPUs

ABSTRACT

Hash table, one of the most fundamental data structure, have been implemented on Graphics Processing Units (GPUs) to accelerate a wide range of data analytics workloads. Most of the existing works focus on the static scenario and try to occupy as much GPU device memory as possible for maximizing the insertion efficiency. In many cases, the data stored in the hash table gets updated dynamically and existing approaches takes unnecessarily large memory resources, which tend to exclude data from concurrent programs to coexist on the device memory. In this paper, we design and implement a dynamic hash table on GPUs with special consideration for space efficiency. To ensure the insertion performance under high filled factor, a novel coordination strategy is proposed for resolving massive thread conflicts. Furthermore, We devise an efficient resizing strategy for the dynamic scenario without rehashing the entire table and the strategy ensures a guaranteed filled factor. Extensive experiments have validated the effectiveness of the proposed design over several state-of-the-art hash table implementations on GPUs. For bounded filled factor, our hash table design achieves up to 5x speedups over the compared approaches against dynamic workloads.

PVLDB Reference Format:
xxx. . *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

The exceptional advances of General-Purpose Graphics Processing Units (GPGPUs) in recent years have completely revolutionized the computing paradigm across multiple fields, including cryptocurrency mining [**?**, **?**], machine learning [**?**, **?**], and database technologies [**?**, **?**]. GPUs bring astonishing computational power that is only available from supercomputers in the past. The state-of-the-art commercial GPU equipment (GV100) is capable to operate at the speed of 14.8 TFLOPS for single precision arithmetic and 870 GB/s peak memory bandwidth [1]. Meanwhile, as the computing resources of GPUs continue to explode, it makes rooms for more applications to run on a single GPU equipment concurrently. For example, GV100 is built with 5120 CUDA cores and 32GB device memory. Nevertheless, most GPU programs try to occupy as much resources as possible to maximize their performance individually. The egoism renders inefficiency when the GPUs run multiple programs simultaneously. Imaging a number of concurrent programs requesting a total memory size larger than the device memory. Interleaving the program executions leads to redundant data transfers between CPUs and GPUs through PCIe, which is expensive in nature.

In this paper, we investigate a fundamental data structure, i.e., *hash table*, which has been implemented on GPUs to accelerate numerous applications, ranging from relational hash join [**?**, **?**, **?**], data mining [**?**, **?**, **?**], key value store [**?**, **?**, **?**] and many others [**?**, **?**, **?**, **?**, **?**]. Existing works [**?**, **?**, **?**, **?**, **?**] focus on the static scenario: they know the data size in advance and allocate a sufficiently large hash table to insert all data entries efficiently. In many cases, the data size varies and static allocation leads to poor device memory utilization. To fill this gap, we design a dynamic GPU hash table that adaptively adjusts to the size of active entries in the table. The hash table supports efficient memory management by sustaining a guaranteed *filled factor* of the table when the data size changes. There are two major challenges for maintaining a high filled factor for hash tables on GPUs:

- It leads to a smaller hash table size with less distinct keys, hence triggers additional conflicts as multiple threads trying to insert/delete the data, which is particularly expensive under the GPU architecture;
- It also means that the table needs to be frequently adjusted to the active data size and we incur costly rehashing to move the entries to a new table when the old table cannot accommodate the adjusted data.

To overcome the aforementioned challenges, we propose a dynamic cuckoo hash table on GPUs. Cuckoo hashing [**?**] uses a number of hash functions to provide each key with multiple locations instead of one. When a location is occupied, existing key is relocated to make room for the new one. Existing works [**?**, **?**, **?**, **?**] have demonstrated great success in speeding up their respective applications by paralleling cuckoo hash on GPUs. However, most of these works require the size of a Key-Value (KV) pair to fit a single

[1]Quadro GV100 launched by NVIDIA in March 2018

Proceedings of the VLDB Endowment, Vol. 12, No. xxx

DOI: https://doi.org/TBD

atomic transaction on GPUs (64 bits wide) to handle conflicts when multiple threads trying to update keys hashed to the same value. In addition, a complete relocation of the entire hash table is required when the data cannot be completely inserted. It thus calls for a general hash table design to support larger KV size as well as efficient relocation strategy against dynamic updates. In this work, we offer two novel designs for implementing dynamic cuckoo hash tables on GPUs.

First, we propose a voter-based coordination scheme among massive GPU threads to support efficient locking without assuming the size of the key-value pairs. For each hash value, we allocate a bucket of b locations to store key-value pairs. Each thread is assigned to an insertion operation on one key. Instead of immediately acquiring a lock on the corresponding bucket to be updated, a thread will first propose a vote among its warp group and all threads in that same warp collaborate to join the winner thread for its update task. There are three distinguishing advantages for the voter-based coordination: (a) once a conflict is detected on one bucket, instead of spinning, the warp instantly revotes and switches to another bucket; (b) a near-optimal load balancing is achieved as a thread will assist other warp-aligned threads, even when the thread finishes its assigned tasks; (c) locking the bucket exclusively allows to update KV pairs without assuming their length below 64 bits.

Second, we employ the cuckoo hashing scheme with d subtables specified by d hash functions, and introduce a resizing policy to maintain filled factor in a bounded range, while minimizing entries in all subtables relocated at the same time. Insertions and deletions would trigger the hash tables to grow and shrink, if the filled factor falls out of the specified range. Our proposed policy only lock one subtable for resizing and it always ensures no subtable can be more than twice as large as any other for handling subsequent resizing efficiently. Meanwhile, the entries in the hash table are distributed so that each subtable has near-equivalent filled factor. In this way, we drastically reduce the cost of resizing the hash tables and provide better system availability compared with existing works that needs to relocate all data for resizing. Our theoretical analysis demonstrates the optimality of the scheduling policy in terms of processing updates. Empirically, the proposed hash table design is capable to operate efficiently at filled factors exceeding 90%.

Hereby, we summarize our contributions as follows:

- We propose a general dynamic hash table without assuming the size of the key-value pair and devise a novel voter-based coordination scheme to support the locking mechanism under massive GPU threads.
- We introduce an efficient policy to resize the hash tables and our theoretical analysis has demonstrated the near-optimality of the resizing policy.
- We conduct extensive experiments on both synthetic and real datasets to showcase the superiority of the proposed approach over several state-of-the-art baselines on GPU hashing. For guaranteed filled factor, our hash table design achieves up to 5x speedups over the baselines against dynamic workloads. The profiling results have also revealed that our approach efficiently leverages GPU resources and demonstrates near-optimal load balancing.

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	filled factor of the hash table
loc	a bucket in the hash table

The remaining part of this paper is organized as follows. Section **??** introduces the preliminaries and the background on GPUs, followed by the related work in Section **??**. Section **??** presents our hash table designs as well as the voter-based coordination scheme. Section **??** introduces our resizing policy against dynamic updates on hash tables. The experimental results are reported in Section **??**. Finally, we conclude the paper in Section **??**.

2. PRELIMINARIES

In this section, we first introduce some preliminaries on general hash tables and then present the background on the GPU architecture.

2.1 Hash Table

A hash table is a fundamental data structure to store KV pairs (k, v) and the value could refer to either the actual data or a reference to the data. The hash table offers the following functionalities: **insert** (k, v) - stores (k, v) in the hash table; **find** (k) - given k returns the associated values if they exist, and NULL otherwise; and **delete** (k) - removes existing KVs that match k if they 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. Contrary to these schemes, cuckoo hashing [**?**] 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 in *one* of the hash tables. When inserting a (k, v), we store the pair to $loc = h^1(k)$ 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 will be reinserted into another hash table, e.g., $loc' = h^2(k')$. We repeat this process until an empty location is encountered.

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 $\theta_i = m_i/n_i$.

2.2 GPU Architecture

We focus on introducing the background of NVIDIA GPU architecture in this section due to its popularity and wide adoption of the CUDA programming language. It is noted that our proposed approaches are not unique to NVIDIA GPUs and can be implemented on other GPU architectures as well. Figure **??** presents a high-level layout of a NVIDIA GPU. An application written in CUDA executes on GPUs

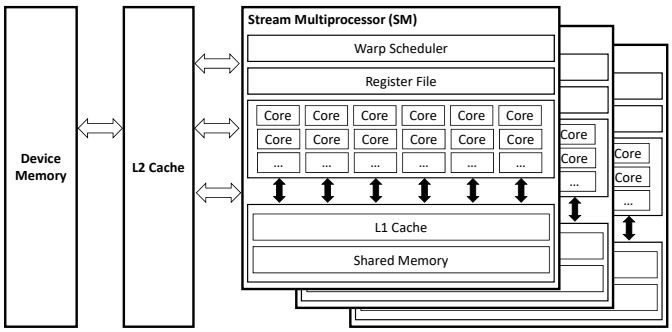


Figure 1: Layout of an NVIDIA GPU architecture

through invoking the *kernel* function. The kernel is organized as a number of *thread blocks*, and one block executes all its threads on a *streaming multiprocessor*, which contains a number of CUDA cores as depicted in the figure. Within a block, threads are divided into *warps* of 32 threads each. A CUDA core executes the same instruction of a warp in a lockstep[2]. Each warp runs independently but can collaborate through different memory types discussed as the following.

Optimizing GPU Programs. When programming a GPU device, there are several important guidelines to harness GPUs' massive parallelism.

- *Minimize Warp Divergence.* Threads in a warp will be serialized if they execute different instructions. To enable maximum parallelism, one needs to minimize branching statements executed within a warp.
- *Coalesced Memory Access.* Warps have a wide cache line size (128 bytes for NVIDIA GPU). The threads are better off to read consecutive memory locations for fully utilizing the device memory bandwidth, otherwise to trigger multiple random accesses for a single read instruction by a warp.
- *Control Resource Usage.* Registers and shared memory are valuable resources to enable fast local memory accesses. Nevertheless, each SM has limited resources (GTX 1080 has 98 KB shared memory and 256KB register files per SM). Overdosing register files or shared memory leads to reduced parallelism on a SM.
- *Atomic Operations.* When facing thread conflicts, an improper locking implementation leads to serious performance degradation. One can leverage the native support of atomic operations [**?**] on GPUs to carefully resolve the conflicts and minimize thread spinning.

3. RELATED WORKS

Alcantara *et al.* present a seminar work on GPU-based cuckoo hashing to accelerate computer graphics workloads [**?**]. This work has inspired a number of applications from diverse fields. Wu *et al.* investigate the use of GPU-based cuckoo hashing for on-the-fly model checking [**?**]. A proposal of accelerating the nearest neighbor search is presented in [**?**]. Due to the success of cuckoo hashing on GPUs, the implementation of [**?**] has been adopted in the CUDPP library[3]. To improve from [**?**], stadium hash is proposed in [**?**] to support out-of-core GPU parallel hashing. However it uses double hashing which needs to rebuild the entire table for any deletions. Zhang *et al.* propose another efficient design of GPU-based cuckoo hashing, named MegaKV, to boost the performance for KV store [**?**]. Subsequently, Horton table [**?**] improves the efficiency of **find** over MegaKV by trading with the cost of introducing a KV remapping mechanism. Meanwhile, in the database domain, several SIMD hash table implementations have been proposed to facilitate relation join and graph processing, including cuckoo hash [**?**] and linear probing [**?**].

It is noted that the aforementioned works focus on the static case: the size of data that needs to be hashed is known in advance. Thus, the static designs would prepare a sufficiently large memory to store the hash table. In this way, the hash table operations are fast since the collision rarely happens. However, the static approach wastes memory resources and, to some extent, it prohibits data from other applications to coexist on the device memory. Moreover, existing designs rely on atomicExch operation to avoid conflicts when transacting a KV pair on GPUs, which only supports up to 64 bits for KV altogether. This design, albeit being efficient, has severe limitation as the value component exceeds 64 bits in many real-world scenarios. This motivates us to develop a general dynamic hash table on GPUs that not only support larger key-value pairs but also actively makes adjustments according to the data size to preserve space efficiency. In this paper, we propose a dynamic cuckoo hash table on GPUs, which maintains high filled factor to minimize memory footprint. In order to support efficient concurrent hash updates, we introduce a novel voter-based coordination scheme which reduces thread conflicts.

To the best of our knowledge, there is only one existing work for building dynamic hash tables on GPUs [**?**]. This proposed approach presents a concurrent linked list structure, called *slab list*, to construct the dynamic hash table with *chaining*. However, parallel chaining incurs two major issues. First, it could frequently invoke concurrent memory allocation requests, especially when the data keeps inserting. Efficient concurrent memory allocation is difficult to implement under the GPU architecture due to its massive parallelism. Although a dedicated memory management strategy is proposed in [**?**] to alleviate this allocation cost, it is not transparent to other applications. More specifically, the dedicated allocator still needs to reserve a large piece of memory in advance to prepare for efficient dynamic allocation, and the occupied space cannot be readily accessed by other concurrent applications. Second, 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**, but also triggers more overheads in resolving conflicts when multiple **insert** and **delete** operations occur at the same key. In contrast, the cuckoo hashing table adopted in this work guarantees $\Theta(1)$ worst case complexity for **find** and **delete**, $O(1)$ amortized **insert** performance. In this paper, we do not introduce extra complication in implementing our own version of memory manager but only reply on the default memory allocator provided by CUDA.

[2]Although the latest Volta architecture supports independent thread scheduling [], it is still more efficient to run with the lockstep execution model.

[3]https://github.com/cudpp/cudpp

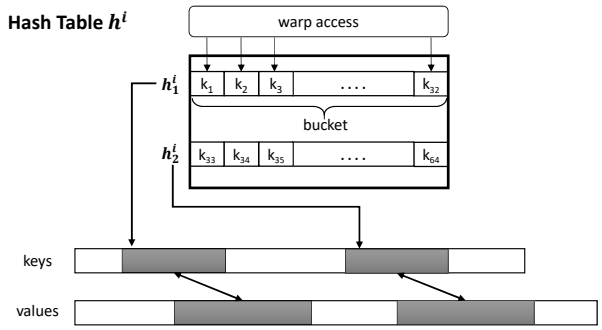


Figure 2: The hash table structure

4. VOTER-BASED GPU CUCKOO HASH

In this section, we first present a overview of the hash table structure and pinpoint a number of important design principles in Section **??**. Subsequently, in Section **??**, we give details on how to perform updates with the *voter* coordination, i.e., **find**, **insert** and **delete**, for the case where hash table resizing is not required. We leave how to handle the fully dynamic scenario in Section **??** for further discussions.

4.1 Hash Table Structure

Adopted from cuckoo hashing [**?**], 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 function such as $h^i(k) = (a_i \cdot k + b_i \mod p) \mod |h^i|$. a_i, b_i are random integers, p is a large prime. Note that the proposed approaches in this paper also apply to other hash functions as well. There are three major advantages for adopting cuckoo hashing on GPUs. First, it avoids chaining by inserting the elements into alternative locations if collision happens. As discussed in Section **??**, chaining present several issues which are not friendly to the GPU architecture. Second, to lookup a KV pair, one only needs to search d locations specified by d unique hash functions. Thus, the data could be stored contiguously in the same location and thus enable the preferred coalesced memory access. Third, cuckoo hashing can maintain high filled factor, which is ideal for memory saving in the dynamic scenario. For $d = 3$, cuckoo hashing achieves over 90% filled factor and still process **insert** operations efficiently [**?**].

Figure **??** depicts the design of a single hash table h^i on GPUs. Assuming the keys are 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 the utilization of memory bandwidth in GPUs. Since the 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 to store the keys and the values respectively. The values could take much larger memory space than the keys. Thus, storing keys and values separately avoid the overhead of memory access when accessing the values are not necessary, e.g., finding a nonexistent KV pair or deleting a KV pair.

For keys having size larger than 4 bytes, a simple strategy is to store less KV pairs in a bucket. Suppose the keys are 8-types, a bucket can then accommodate 16 KV pairs. As we will discussed later in this section, we lock the entire bucket exclusively for a warp to perform insertion/deletion 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 occupy one bucket alone, which is unnecessary large in practice.

Algorithm 1 Find(lane l, warp wid, key k)

```
found ← ∅
for i = 0, ..., d − 1 do
    loc ← h^i(k)
    found ← ballot(loc[l].key == k)
    if found ≠ ∅ then
        return loc[l]
return false
```

4.2 Parallel Hash Table Operations

In the reminder of this session, we discuss how to utilize GPUs' threads to execute hash table operations in parallel. Following existing works [**?**, **?**, **?**], we assume the **find**, **insert** and **delete** operations are batched and each batch only contains one type of operations. A batch with mixed type of operations can also be supported but the semantic is ambiguous.

Find. It is relative straightforward to parallelize **find** operations since only read access is required. Given a batch of size m, we launch w warps in total (which means launching $32w$ threads in total) and each warp is responsible for $\lfloor \frac{m}{w} \rfloor$ **find** operations. To locate a KV, we need to hash the key with d hash functions and look for the corresponding locations. Algorithm **??** presents the pseudo code for a KV lookup by a warp. First, the bucket of the ith hash table is located, i.e., loc_i. Then, each thread in the warp (a thread lane l) simultaneously searches the key and the ballot function return the lane for which the key is found.

Insert. Contention occurs when multiple **insert** operations target at the same bucket. There are two contrasting objectives for resolving the contention. On one hand, we want to utilize the warp-centric approach to access a bucket. On the other hand, a warp requires a mutex when updating a bucket to avoid corruption, and the locking is expensive on GPUs. In the literature, it is a common practice to use atomic operations for implementing a mutex under the warp-centric approach [**?**]. In particular, we can still invoke a warp to insert a KV pair. The warp is required to acquire a lock before updating the corresponding bucket. The warp will keep trying to acquire the lock before successfully obtain the control. There are two drawbacks for this direct warp-centric approach. First, the conflicting warps are spinning while locking, thus wasting computing resources. Second, although atomic operations are supported by recent GPU architectures natively, they become costly when the number of atomic operations issuing at the same location increases. In Figure **??**, we show the profiling statistics for two atomic operations which are often used to lock and unlock a mutex: atomicCAS and atomicExch respectively. We compare the throughputs of the atomic operations vs. 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 perfor-

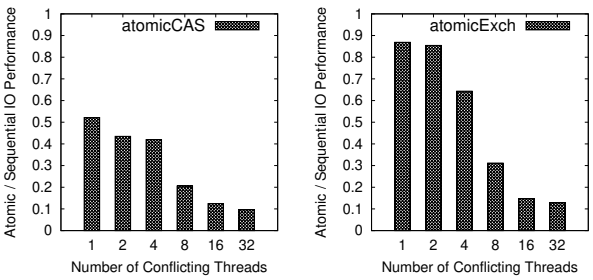


Figure 3: The performance of atomic operations for varying number of conflicts

mance has seriously degraded when a larger number of conflicts occur. Thus, it will be expensive for the direct warp-centric approach in the contention critical cases. Imaging one wants to track the number of retweets posted to active twitter accounts in the current month, via storing the twitter ID and the obtained retweet counts as KV pairs. In this particular scenario, certain twitter celerities could receive thousands of retweets in a very short period. This triggers the same twitter ID gets updated frequently and a serious number of conflicts would happen.

To alleviate the cost of spinning, we propose the voter-coordination scheme. We assign an **insert** to a thread rather than using a warp to handle the operation. Before submitting a locking request and updating the corresponding bucket, the thread will participate in a vote among the threads in the same warp. The winner thread l becomes the leader of the warp and takes control. Subsequent, the warp inspects the bucket and insert the KV for l if there are spaces left, upon l successfully obtaining the lock. If l fails to get the lock, the warp revotes another leader to avoid locking on the same bucket. Compared with locking on atomic operations, the cost of warp voting is almost negligible since it is heavily optimized in the GPU architecture.

Parallel insertion with the voter coordination scheme is presented in Algorithm **??**. The pseudo code demonstrates how a thread (with lane l) from warp *wid* inserts a KV (k, v) to the ith hash table. The warp will first propose a vote among active threads and the process will terminate if all threads finish their tasks (lines **??**-**??**). This achieves better resource utilization since no thread will be idle when one of the threads in the same warp is active. The leader l' then broadcasts its KV pair (k', v') as well as the hash table index i' to the warp and tries to lock the inserting bucket (lines **??**-**??**). The broadcast function ensures all threads in the warp receive the locking result and the warp revotes if l' fails to obtain the lock. Otherwise, the warp follows l' and proceeds to update the bucket for (k', v') with a warp-centric approach similar to **find**. Once a thread finds k' or an empty space in the bucket, l' will add or update with (k', v') (lines **??**-**??**). Otherwise there is no space left, l' swaps (k', v') with a random KV in the bucket and inserts the evicted KV to hash table $i + 1$ in the next round. In summary, each iteration of the loop presented in Algorithm **??** issues 1 atomic operation and at most 1 device memory read/write (lines **??**-**??**).

Note that we have yet covered how to choose the hash table index i for each insertion (Algorithm **??**). Additional number of conflicts would happen if all threads attempting

Algorithm 2 Insert(lane l, warp *wid*, kv (k, v), table i)

```
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)
22:        i ← i + 1
```

to insert to the same table. For the flow of presentation, we leave the discussion to Section **??** since choosing the hash table is more coherent to the load balancing problem for resizing hash tables.

We give the following example to demonstrate the parallel insertion process.

EXAMPLE 1. *In Figure* **??**, *we visualize the scenario for three threads: l_x, l_y, l_z from warp a and warp b, inserting KV pairs (k_1, v_1),(k_{33}, v_{33}),(k_{65}, v_{65}) independently. Suppose l_y and l_z become the leaders of warp a and b respectively. Both threads will compete for the bucket y and l_z wins the battle. l_z will then leads warp b to inspect the bucket and evict (k_{64}, v_{64}) by replacing with (k_{65}, v_{65}). In the meanwhile, l_y does not lock on bucket y and joins the new leader l_x voted in warp a. l_x locks bucket x and inserts (k_1, v_1) in place. Subsequently, l_y may get back 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. The **delete** operation, in contrast to **insert**, does not require locking with a warp-centric approach. Similar to **find**, we assign a warp to process a key k on deletion. The warp iterates through the buckets among 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 only erase the key if k is found, thus causes no conflict.

Complexity. Since a **find**, **insert** and **delete** operation is independently executed by a thread. The analysis of a single thread complexity is the same as the sequential version of cuckoo hashing [**?**]: $O(1)$ worst case complexity for **find** and **delete**, $O(1)$ expected time for **insert** for the case of 2 hash tables. It has been pointed out that analyzing the theoretical upper bound complexity of insertion in $d \geq 3$ hash tables is hard [**?**]. Nevertheless, empirical results have shown that increasing the number of tables leads to better

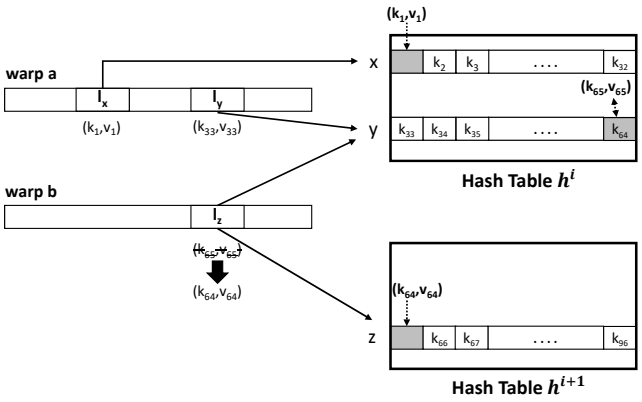


Figure 4: Example for parallel insertions

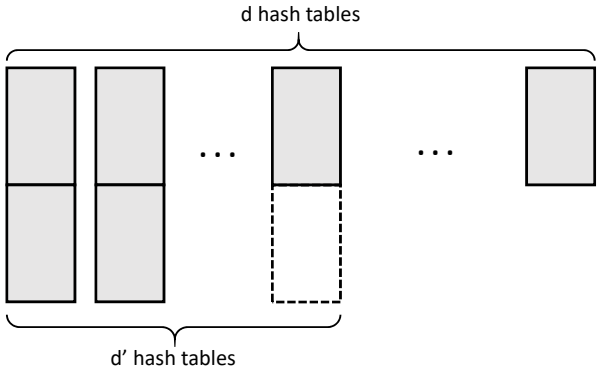


Figure 5: Resizing strategy

insertion performance, see our experiments in Section **??**. Thus, we assume the complexity for inserting to d hash tables is the same as that of 2 has tables, as long as d keeps constant.

We then analyze the number of possible thread conflicts. Assuming we launch m threads in parallel, each of them is assigned to an 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 [**?**], which is $O(\binom{m}{2}/H)$. Given GPUs have a large number of threads, there could be a significant amount of conflicts. Thus, we propose the voter coordination scheme to reduce the cost of spinning on locks. Note that the analysis is done for the cases where the key to update is unique. More conflicts could occur in reality when the same key is updated in parallel.

5. DYNAMIC HASH TABLE

In this section, we focus on how to dynamically update the hash tables according to the actual data size. We introduce our strategy how to resize the table in Section **??**. In section **??**, we discuss how to distribute the KV pairs for better load balancing with theoretical guarantees. Finally, we present how to efficiently rehash and relocate the data after the tables have been resized in Section **??**.

5.1 Structure Resizing

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

We propose an alternative strategy, illustrated in Figure **??**. Given d hash tables depicted in Figure **??**, we always double the smallest subtable or chop the largest subtable into half for upsizing or downsizing respectively, when filled factor falls out of $[\alpha, \beta]$. In other words, no subtable will be more than double the size as others as shown in the figure. This strategy implies that we do not need to lock all hash tables and only to resize one, thus achieving better performance stability compared with the aforementioned simple strategy. Assuming there are d' hash tables with size $2n$, $d - d'$ tables with size n and a current filled factor θ, one upsizing process when $\theta > \beta$ lowers the filled factor to $\frac{\theta \cdot (d+d')}{d+d'+1} \geq \frac{\beta \cdot d}{d+1}$. Since 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, while it leads to less efficient **find** and **delete**. We allow the user to configure the number of hash tables to trade off between memory and query processing efficiency.

5.2 KV distribution

Recall Section **??** where we discuss how to insert a KV pair attached to a thread, we assign a hash table ID for inserting the KV pair. Here, we present how to choose the hash table ID and distribute the KV pairs among all hash tables to minimize the number of conflicts occurred. We have the following theorem to guide us for allocating insertions.

THEOREM 1. *The amortized conflicts for inserting m unique KV pairs to d hash tables is 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. It is noted that the amortized insertion complexity of cuckoo hash is $O(1)$. Thus, similar to balls and bins analysis, the expected number of conflicts occurred for inserting m_i elements in table i is 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:

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

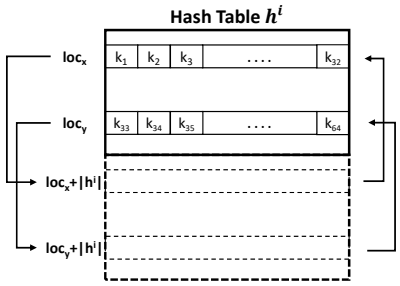


Figure 6: Illustration for upsizing and downsizing.

By Jensen's inequality, the following inequality holds:

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

where the 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. □

According to our design, one hash table can only be as twice large as the other tables. This implies 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 have a higher filled factor. To maintain the balancing condition in Theorem **??** efficiently, we employ a randomized approach: a KV pair (k,v) is assigned to table i with a probability proportional to $n_i/\binom{m_i}{2}$, given there are m_i elements in table i before the batch of insertion containing (k,v).

5.3 Rehashing

Rehashing relocates the KV pairs after the hash tables are resized. An efficient relocation process maximizes the utilization of device memory bandwidth and minimizes thread conflicts. We discuss two scenarios for rehashing: *upsizing* and *downsizing*. Both scenarios are processed in one single kernel.

Upsizing. We introduce a conflict-free rehashing strategy for the upsizing scenario. Figure **??** presents an illustration for upsizing a hash table h^i. As we always double the size for h^i, a KV pair which 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 bucket to fully utilize the cache line size. Each thread in the warp corporately takes a KV pair in the bucket and relocates the KV pair if necessary. Moreover, the rehashing does not trigger any conflict since KV pairs from two distinct buckets before upsizing cannot be rehashed to the same bucket. Thus, the locking of a bucket is not required and we can make use of the full device memory bandwidth for the upsizing process.

Note that after upsizing hash table h^i, its filled factor θ_i is cut to half, which could break the balancing condition emphasized in Theorem **??**. Nevertheless, we use the sampling strategy for subsequent KV insertions, where 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 while n_i doubles after upsizing, the scenario leads to double the probability of inserting subsequent KV pairs to h^i.

Downsizing. Downsizing the hash table h^i is a reverse process of upsizing h^i. There is always room to relocate KV pairs in the same table for upsizing. In contrast, downsizing may rehash some KV pairs to other hash tables, especially when $\theta_i > 50\%$. Since the KV pairs located in loc and $loc+|h^i|$ are hashed to loc in the new table, there could be cases the KV pairs exceeds the size of a single bucket. Thus, we first assign a warp to accommodate KV pairs that can fit the size of a single bucket. Similarly as upsizing, it does require locking since there will be no thread conflict on any bucket. For the remaining KV pairs which cannot fit in the downsized table, called residuals, we insert them into other subtables using Algorithm **??**. To make sure no conflict occurs between inserting residuals and processing the downsizing subtable, which are both executed in a single kernel, we exclude the downsizing subtable when inserting the residuals. Take 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/downsizing rehashes at most m/d KV pairs. For inserting/deleting these m elements, the number of rehashes is bounded by $2m$. Thus, the amortized complexity for inserting m elements is still $O(1)$.

6. EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments by comparing the proposed hash table designs with several state-of-the-art approaches for GPU-based hash tables. We will introduce the experimental setup and the discuss the results.

6.1 Experiment Setup

Baselines. We compare the proposed approach in this paper with several state-of-the-art hash table implementations on GPUs and CPUs.

- `CUDPP` is a popular CUDA primitive library[4] which contains the cuckoo hash table implementation published in [**?**]. For `CUDPP`, each hash value stores only one KV pair with 64 bits size (32 bits for key and 32 bits for value). `CUDPP` only supports **insert** and **find** operations.
- `MegaKV` is a state-of-the-art approach for GPU-based key value store published in [**?**]. `MegaKV` employs a cuckoo hash as well and it allocates a bucket for each hash value. However, it does not lock a bucket when performing update. Instead, it uses intra-block synchronization to resolve race condition.
- `Linear` is a GPU-based hash table which uses linear open addressing to resolve conflicts [**?**].
- `GoogleHash` is an efficient CPU-based hash table implementation[5]. We choose the *dense_hash_map* implementation since it provides the best efficiency.
- `DyHash` is the approach proposed in this paper.

We adopt the original implementations of `CUDPP` and `GoogleHash` from their corresponding inventors. For `MegaKV`, we note

[4]https://github.com/cudpp/cudpp
[5]https://github.com/sparsehash/sparsehash-c11/

Table 2: The datasets used in the experiments.

Datasets	KV pairs	Unique keys	Max Duplicates
TW	50,876,784	44,523,684	4
RE	48,104,875	41,466,682	2
LINE	50,000,000	45,159,880	4
COM	10,000,000	4,583,941	14
RAND	100,000,000	100,000,000	1

Table 3: Parameters in the experiments

Parameter	Settings	Default
α	25%, 30%, 35%, 40%, 45%	40%
β	75%, 80%, 85%, 90%, 95%	90%
r	0.1, 0.2, 0.3, 0.4, 0.5	0.4

that its intra-block synchronization could lead to inconsistency issue as well as a large number of insertion failures, especially when the filled factor is high. Thus, we revise its code by replacing the intra-block synchronization with atomicExch to resolve the race condition. The adoption of atomicExch preserves the design principle of MegaKV for *not* locking the entire bucket for updates. Moreover, it leads to less insertion failures and similar performance against its original implementation. For Linear, we implement the CUDA version of the algorithm proposed in [**?**].

Note that we do not compare with the dynamic GPU hash approach proposed in [**?**] for two major reasons. First, we cannot obtain the original implementation from its authors. Second, the approach devises a dedicated memory allocator other than cudaMalloc. A dedicated allocator will improve the performance but add complexity the system. Additionally, it needs to occupy a large memory in advance and is not transparent to other GPU applications. In contrast, our proposed approach only use native allocator supported. We do not compare with [**?**] since it only improves MegaKV marginally using a more costly insertion process.

Datasets. We evaluate all compared approaches using several real world and synthetic datasets described as follows:

- TW: Twitter is an online social network where users perform actions include *tweet*, *retweet*, *quote* and *reply*. We crawl these actions for one week via Twitter stream API[6] on 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 actions include *post* and *comment*. We collect all Reddit *comment* actions in May 2015 from *kaggle*[7] and query the Reddit API for the *post* actions the same period. The dataset contains 48,104,875 actions as KV pairs.

- LINE: Lineitem is a synthetic table generated by the TPC-H benchmark[8]. We generate 100,000,000 rows of the lineitem table and combine the *orderkey,linenumber* and *partkey* column as keys.

[6]https://dev.twitter.com/streaming/overview
[7]https://www.kaggle.com.reddit/reddit-comments-may-2015
[8]https://github.com/electrum/tpch-dbgen

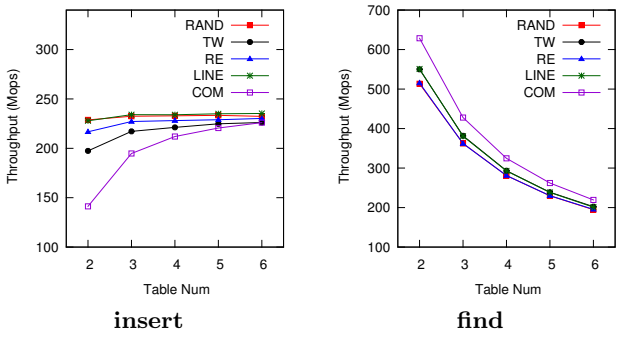


Figure 7: Throughput of DyHash when varying the number of hash tables.

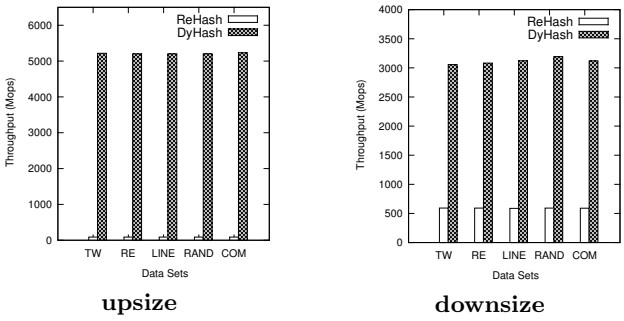


Figure 8: Throughput of subtable resize.

- RAND: Random is a synthetic dataset generated from a normal distribution. We have deduplicated the data and generate 100,000,000 KV pairs.

- COM: Databank is a PB-scale data warehouse that stores Alibaba's customer behavioral data for the year 2017. Due to confidentiality, we sample 10,000,000 transactions and the dataset contains 4,583,941 encrypted customer IDs as keys.

The summary of the datasets can be found in Table **??**. Since all GPU approaches (except for DyHash) only supports 32 bit key and 32 bit value, we hash longer keys to 32 bits and truncate all values to 32 bits across all datasets.

Static Hashing Comparison (Section ??). Under the static setting, we evaluate **insert** and **find** performance among all compared approaches. In particular, we insert all KV pairs from the datasets followed by issuing 1 million random search queries.

Dynamic Hashing Comparison (Section ??). Under the dynamic setting, we generate the workloads by batching the hash table operations. We 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 to balance insertions and deletions. After we exhaust all the batches obtained from the datasets, we rerun these batches by swapping the **insert** and **delete** operations in any batch. We evaluate the performance of all compared approaches except CUDPP as it does not support deletions. Since all approaches other than DyHash are static hash tables, we double/half the memory usage followed by rehashing all KV pairs as their resizing strategy, if the corresponding filled factor falls out of the specified

htp

Table 4: The failure of TW (Percentages).

	0.75	0.8	0.85	0.9	0.95
Linear	0.0	0.0	4.6e-5	1.1e-3	3.9e-1
CUDPP	0.0	0.0	0.0	0.0	3.8e-5
MegaKV	3.4e-1	5.6e-1	8.9e-1	1.4	2.0
DyHash	0.0	0.0	0.0	0.0	0.0

Table 5: The failure of RE (Percentages).

	0.75	0.8	0.85	0.9	0.95
Linear	5.6e-3	1.5e-2	1.8e-2	2.0e-2	2.0
CUDPP	0.0	0.0	0.0	0.0	0.0
MegaKV	5.1e-1	8.8e-1	1.5	2.3	3.5
DyHash	0.0	0.0	0.0	1.0e-6	1.0e-6

Table 6: The failure of LINE (Percentages).

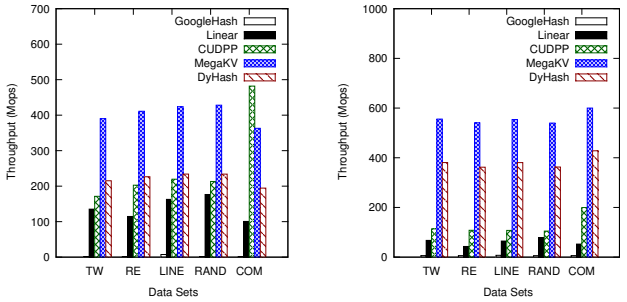
	0.75	0.8	0.85	0.9	0.95
Linear	0.0	0.0	1.1e-4	1.8e-3	3.9e-1
CUDPP	0.0	0.0	0.0	0.0	0.0
MegaKV	4.6e-1	8.4e-1	1.5	2.5	3.9
DyHash	0.0	0.0	0.0	2.0e-6	4.0e-6

Table 7: The failure of RAND (Percentages).

	0.75	0.8	0.85	0.9	0.95
Linear	8.4e-5	4.1e-4	5.7e-4	8.7e-4	2.0
CUDPP	0.0	0.0	0.0	0.0	0.0
MegaKV	4.2e-1	7.9e-1	1.4	2.4	3.9
DyHash	0.0	0.0	0.0	0.0	0.0

Table 8: The failure of COM (Percentages).

	0.75	0.8	0.85	0.9	0.95
Linear	0.0	0.0	2.0e-5	6.5e-4	2.5e-2
CUDPP	5.4e-3	5.3e-3	5.1e-3	5.0e-3	5.3e-3
MegaKV	6.1e-2	8.0e-1	1.0	1.3	1.5
DyHash	0.0	0.0	0.0	0.0	7.1e-4



insert **find**

Figure 9: Throughput of all compared approaches under the static setting.

range. Moreover, if an insertion failure is found for a compared approach, we trigger its resizing strategy.

Parameters. We vary the parameters when comparing `DyHash` with the baselines. α is the lower bound on the filled factor θ for all compared approaches, whereas β is the respective upper bound. r is the ratio of insertions over deletions in a processing batch. The settings of the aforementioned parameters could be found in Table **??**.

Experiment Environment. We conduct all experiments on an Intel Xeon E5-2620 Server equipped with NVIDIA GeForce GTX 1080.The GTX 1080 is built on Pascal architecture with 20 SMs and 128 SPs per SM. The GTX 1080 has 8 GB of GDDR5 memory. Evaluations are performed using CUDA 8.0 on Ubuntu 16.04.3. The optimization level (-O3) is applied for compiling all programs.

6.2 Tunning Parameters

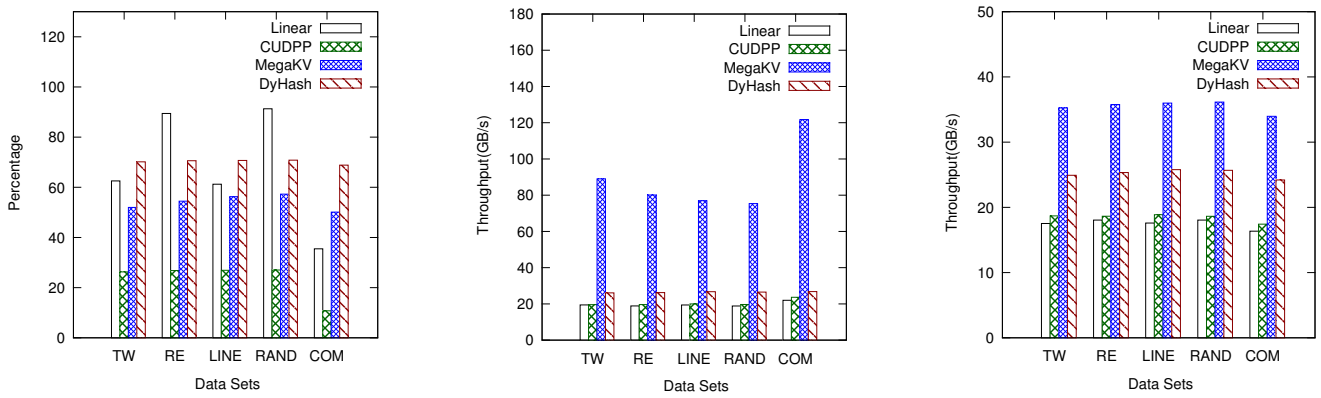
A key parameter that affects the performance of `DyHash` is the number of hash table chosen. For the static scenario, we present the throughput performance of **insert** and **find** for varying number of hash tables in Figure **??**, while fixing the memory space of the entire structure to ensure the default filled factor. The throughput of **insert** increases with more hash tables, since there are more alternative locations for inserting a KV pair. However, the marginal improvement drops for a larger number of hash tables. The throughput of **find** falls with more hash tables as additional locations need to be scanned to locate a KV pair. Thus, one can tradeoff the performance between **insert** and **find** operations by varying the number of tables. In this paper, we choose three hash tables as our default implementation as it provides the best balance.

6.3 Dynamic Resizing

To validate the effectiveness of our resizing strategy proposed in Section **??**, we compare it with rehashing. For evaluating upsizing, we initialize `DyHash` with all the data and the filled factor as the default upper bound 0.9. Then, we perform one time upsizing, i.e., upsize one subtable, and compare our resizing strategy against rehashing all the entries in the subtable with Algorithm **??**. For evaluating downsizing, the setup is a mirror image of upsizing evaluation with an initializing filled factor as the default lower bound 0.4. The throughputs are reported in Figure **??**. The throughput of rehashing for the upsizing scenario is severely limited, since the remaining subtables not being upsized are almost filled ($\theta = 0.9$) and inserting KV pairs resulting frequent evictions. In comparison, the downsizing throughput of rehashing is significantly faster due to a low filled ratio. Our resizing strategy achieves compelling speedups over rehashing. Besides, it only lock the subtable being resized and supports concurrent updates for the remaining subtables.

6.4 Static Hashing Comparison

Throughput Analysis. We present the throughput of all compared approaches in Figure **??** when varying the filled factor. First of all, GPU-based approaches are orders of magnitude faster than `GoogleHash` (2 million ops for **insert** and 6 million ops for **find** on average), which validates the motivation of designing hash tables on GPUs. Among GPU hash tables, `MegaKV` delivers the best performance across all datasets except for `COM`. Nevertheless, for **insert**, the drawback of `MegaKV` is that it fails to insert some of the KV



Warp Efficiency Cache Utilization Memory Bandwidth Utilization

Figure 10: GPU profiling results for static hashing comparison.

pairs. Tables **??**-**??** present the percentages of insertion failure for varying filled factor across all datasets. `MegaKV` has up to 2.5% failure rate for the default filled factor, which is the highest among all approaches. Our proposed `DyHash` has near-zero failure rate and achieves the 2nd best performance behind `MegaKV`. We note that `CUDPP` obtains a significant advantage over all approaches in the `COM` dataset. This is because `CUDPP` will immediately terminate its GPU kernel once it finds some KV pairs cannot be inserted after a number of lookups. This explains why `CUDPP` has a superior performance since it early terminates on `COM` due to failed insertions (Table **??**). For **find**, `MegaKV` achieves the best performance as it only has two hash functions. This explains why `DyHash` is slower since we use three hash tables and additional IO lookups are required for each **find**. Both `MegaKV` and `DyHash` are faster than `CUDPP` and `Linear` since they employs the bucket mechanism for storing multiple KV pairs contiguously under the same hash value, which exhibits better coalesced memory access.

For interested readers, please find the throughput results for varying the filled factor across all datasets in the appendix (Figure **??** and **??**).

GPU Profiling. To further study the behavior of the approaches, we present three types of profiling results for all **insert** GPU kernels in Figure **??**. For *warp efficiency*, `DyHash` maintains a stable rate at around 70%, which is significantly higher than the other two cuckoo hash approaches: `MegaKV` and `CUDPP`. We attribute this phenomenon to the voter mechanism proposed in Section **??**, which yields better overall load balancing. `Linear` could achieve higher warp efficiency than `DyHash`, but is very volatile across different datasets. This is because each **insert** in `Linear` may require scanning varying number of hash values for distinct data distributions. For example, in the `COM` dataset, its warp efficiency drops below 40% since `COM` contains more duplicated keys than other datasets.

Looking at *cache* and *memory bandwidth* profiling results, `MegaKV` and `DyHash` demonstrate better utilization as they employ the bucket mechanism. As `DyHash` always needs to use atomicCAS to lock a bucket before accessing it, its utilization is inferior than that of `MegaKV` due to the additional IO when inserting a KV pair to a bucket. In addition, atomicCAS is less efficient than atomicExch as it involves more workload per operation (see Table **??**). Nevertheless, implementations based on atomicExch only supports KV pair with 64 bits, whereas adopting atomicCAS in `DyHash` can support arbitrary length as we lock the bucket for exclusive update. It is also noted that, even though `MegaKV` has great cache utilization due to the use of atomicExch accessing buckets directly, the overall performance is eventually bounded by device memory IOs.

In summary, under the static environment, `DyHash` achieves competitive efficiency against the compared GPU baselines. Moreover, although `DyHash` does not deliver the best performance over existing approaches, it has very low insertion failure rate and supports more general hash tables.

6.5 Dynamic Hashing Comparison

Varying the filled factor lower bound α. We vary the lower bound of the filled factor and report the results in Figure **??**. The performance is measured by the total running time for processing all batches described in Section **??**. Furthermore, we indicate the time taken for all components involved in the processing: **insert**, **find**, **delete** and **resize**. Apparently, the resizing strategy adopted by `Linear` and `MegaKV` incur significant overhead. Such overhead grows dramatically for a higher α since the number of downsizings are executed. It is noted that the impact of resizing is the more significant for `MegaKV` than `Linear`. We interpret the phenomenon as the following. `MegaKV` has a higher insertion failure rate than that of `Linear` (see Table **??**-**??**). Thus `MegaKV` triggers more upsizings leading to a lower filled ratio when resized. As each processing batch contains deletions, `MegaKV` is prone to downsize when the filled factor falls below α after deletions materialize. The overhead of repeated resizings severely degrades the performance of `MegaKV`, for the major reason of maintaining the filled factor above α. The only exception is where α is very small (i.e., $\alpha = 0.25$) and `MegaKV` does not need to perform downsizing regularly. However, small α means the hash table is mostly empty and thus wastes GPU memory resources. In contrast, `DyHash` achieves significant speedups over `Linear` and `MegaKV` (up to 6.62 and 5.05 faster respectively under the default setting). `DyHash` can support even larger α (i.e., $\beta \cdot \frac{d}{d+1}$) but we omit the results since the baselines cannot deliver reasonable performance beyond $\alpha > 0.5$. There are two reasons behind why `DyHash`'s performance. First, `DyHash` has low failure rate and thus avoids unnecessary resizing operations.

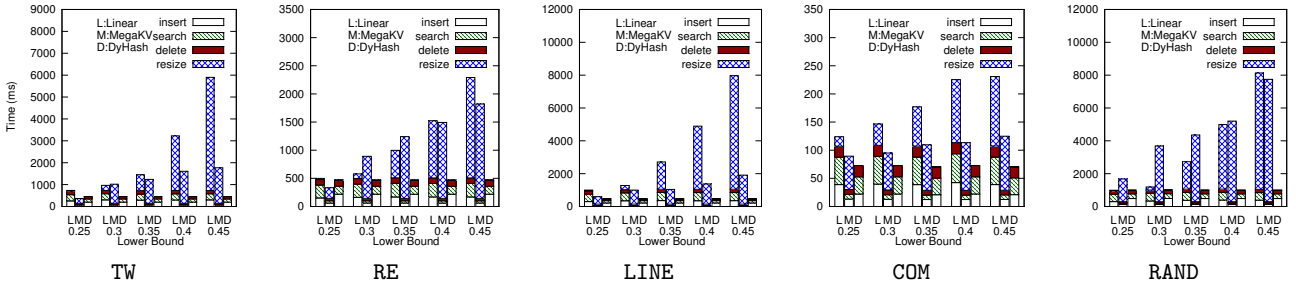


Figure 11: Run time for varying α.

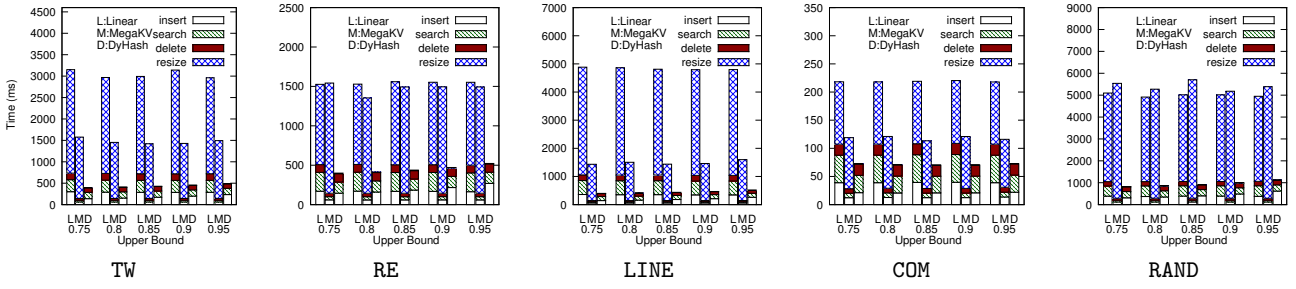


Figure 12: Run time for varying β.

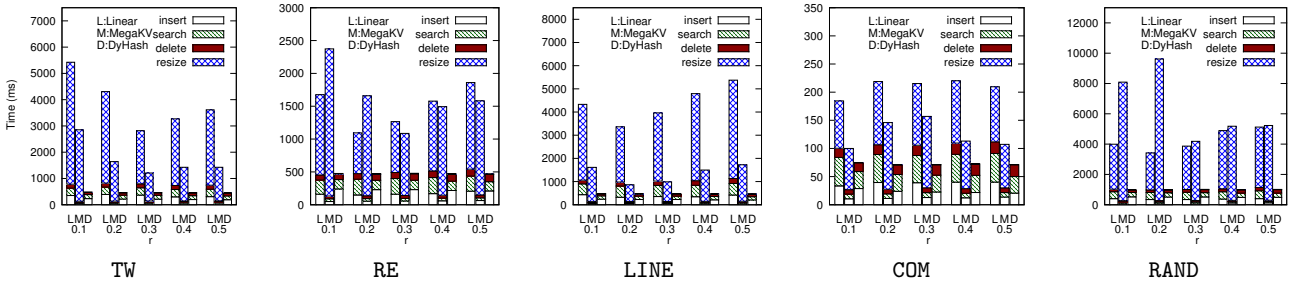


Figure 13: Run time for varying r.

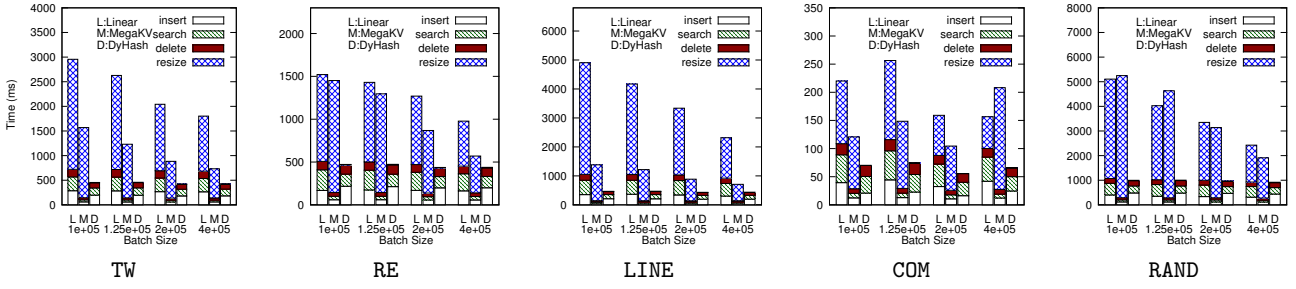


Figure 14: Run time for varying the batch size.

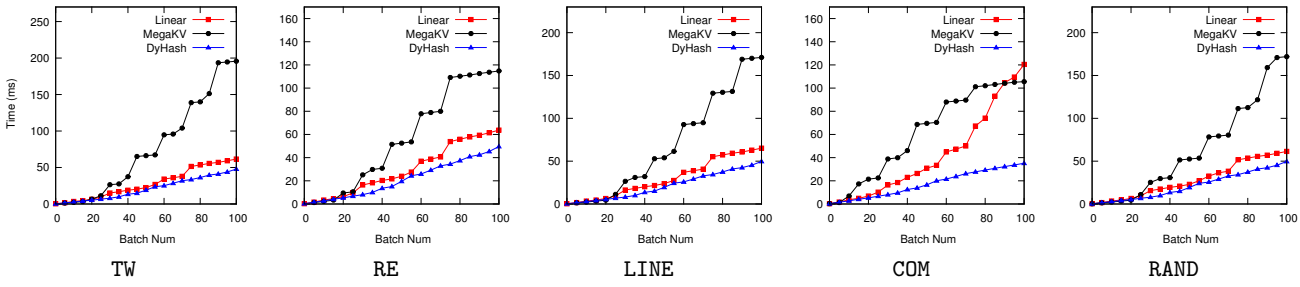


Figure 15: System stability.

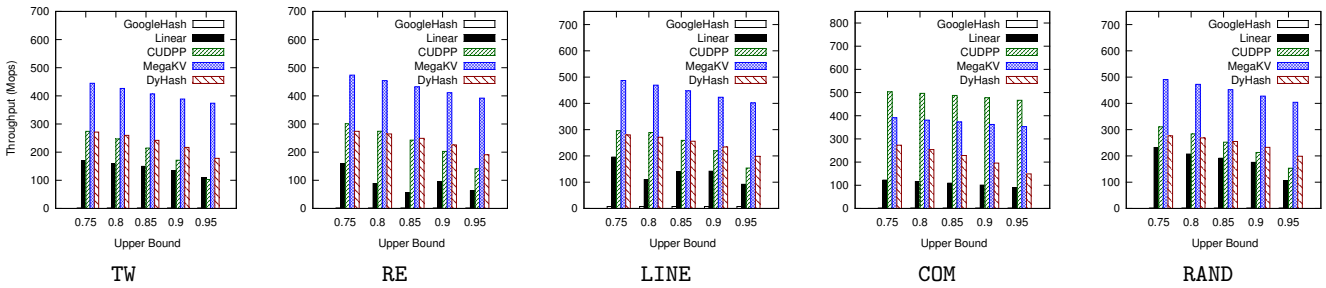


Figure 16: Throughputs of insert for varying β.

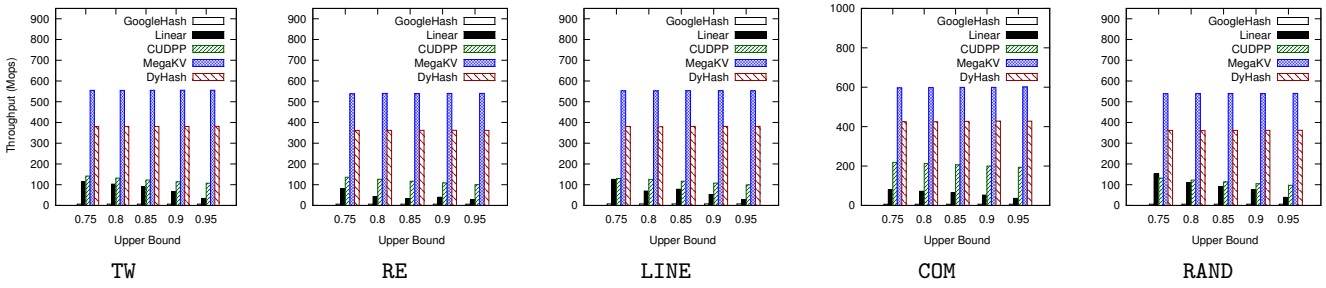


Figure 17: Throughputs of search for varying β.

Second, `DyHash` employs the resizing strategy (proposed in Section **??**) by only relocating the entries in one subtable efficiently. Compared with time taken by **insert**, **delete** and **find**, the cost of **resize** for `DyHash` is almost negligible.

Varying the filled factor upper bound β. The results for varying β is reported in Figure **??**. It is interesting to see that the upper bound does not significantly affect the overall performance for all methods. For `Linear` and `MegaKV`, it is very hard for them to achieve high filled factor since upsizing will be triggered once there are failed insertions. Although `DyHash` also incurs insertion failure, such cases only occur for filled factor beyond 0.9 and thus the resizing operations are rarely invoked for the sake of handling failed insertions. For `DyHash`, it is noted that there is a slight increasing trend for a larger β, especially in `RE` and `RAND` datasets. The increase is mostly attributed to the additional processing workload of the insertions at a higher filled factor, rather than the resizing cost.

Varying insert vs. delete ratio r. In Figure **??**, we report the results for varying the ratio r. `Linear` and `MegaKV` remains inefficient as they incur expensive overheads of resizing. An interesting observation for `Linear` and `MegaKV` is that there exists a sweet spot where the resizing cost is the minimal. It is because the workload composition of insertions and deletions should be just right so that it does not trigger unnecessary upsizings or downsizings. However, in reality, the workloads could vary significantly and existing methods cannot be easily adapted while maintaining guaranteed filled factor. This has again validated the effectiveness of `DyHash` against dynamic workloads.

Varying the batch size. We have also varied the size of each processing batch. The results are reported in Figure **??**. `DyHash` remains the most efficient method over `Linear` and `MegaKV`. It is not surprised to find that the performance of `Linear` and `MegaKV` is the worst for the smallest batch size, since a fine grained processing batch triggers additional resizing for `Linear` and `MegaKV`.

Performance stability. We evaluate the performance stability of the compared approaches in Figure **??**. In particular, we track the elapsed time for running the first 100 update batches. `MegaKV` is the most unstable approach since frequent failed insertions lead to expensive resizing invoked repeatedly. Although `Linear` achieve shorter running time and better stability than `MegaKV`, it is still significantly worse than `DyHash`. Finally, the trend demonstrates the superior stability of `DyHash` as there is no sign of run time bump even through there are resizing operations involved.

7. CONCLUSION

In this paper, we contribute a number of novel designs for dynamic hash table on GPUs. First, we equip the bucket-based hash table structure with a voter based coordinate scheme, which not only supports KV pairs with arbitrary size but also reduce the overhead of thread conflicts through the revoting mechanism. Second, we propose a resizing strategy that actively adjusts the table size to save GPU device memory. The resizing strategy only lock one subtable for efficient upsizing or downsizing, which allows concurrent updates to other subtables. Empirically, our proposed design achieves competitive performance against the state-of-the-art static GPU hash tables. Under the dynamic scenario, we gain up to 5x speedups compared with baselines that uses rehashing to adjust their table size.

Appendix

We provide additional experiments for evaluating the static scenario performance when varying the filled factor. For **insert**, the throughput of all approaches drop for a higher filled factor as it becomes harder to insert when the hash tables are almost filled. For **find**, all approaches except

`Linear` has constant performance for different filled factor because they all fall into the category of cuckoo hash and has a fixed number of locations for lookups. Overall, `DyHash` achieves the second best performance behind `MegaKV` under the static scenario, which is consistent with our discussions in Section **??**.