1

Dynamic Hash Tables on GPUs

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

Abstract—~~Hash~~**The** hash table~~, one of the most~~ is a fundamental data ~~structures, have~~ structure that has been implemented on ~~Graphics Processing Units~~graphics processing units (GPUs) to accelerate a wide range of data analytics workloads. Most existing works ~~focus~~have focused on ~~the~~ static scenarios and ~~try to occupy~~occupying large GPU memory ~~for maximizing the~~to maximize insertion efficiency. In many cases, ~~the~~ data stored in ~~the~~ hash ~~table gets~~tables get updated dynamically and existing approaches ~~take~~use unnecessarily large memory resources. One naive solution is to rebuild a hash table (~~a.k.a~~known as rehashing) whenever it is either filled or mostly empty. However, this approach renders significant overheads for rehashing. In this paper, we propose ~~*DyCuckoo,*~~ a novel dynamic cuckoo hash table technique on GPUs~~.~~, known as *DyCuckoo*. We devise an efficient resizing strategy for ~~the~~a dynamic scenario without rehashing the entire table ~~and the strategy~~that ensures a guaranteed filled factor. The strategy trades search performance with resizing efficiency, and th~~e~~is tradeoff can be configured by ~~the~~ users. To further improve efficiency, we further propose a two-layer cuckoo hashing scheme that ensures ~~at most~~a maximum of two lookups for find and delete operations, while ~~still retains~~retaining similar performance for insertion as ~~that of~~ general cuckoo hash tables. Extensive experiments have validated the proposed design's effectiveness ~~of the proposed design~~ over several state-of-the-art hash table implementations on GPUs. *DyCuckoo* achieves superior performance while sav~~es~~ing up to ~~4x~~four times the memory over the state-of-the-art approaches against dynamic workloads.

- 4 -

1 INTRODUCTION

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

In this paper, we investigate a fundamental data structure~~, i.e.,~~ known as the hash table, which has been implemented on GPUs to accelerate ~~numerous~~ applications~~,~~ ranging from relational hash joins [15], [14], [16], data mining [30], [39], [38], key

Comment [Author1]: Remark: Journals usually recommend numbering the in-text citations in the sequence of their appearance in the text. The reference citations in this document are not numbered sequentially. Check and revise accordingly.

value storage [36], [17], [9~~] and many~~], among others [8], [29], [13], [26], [35]. Existing works [2], [36], [18], [17], [9] ~~focus~~have focused on

the static scenario~~: they know~~ in which the size of the data ~~size~~is known in advance and ~~allocate~~ a ~~sufficiently large~~ hash table large enough to efficiently insert all data entries ~~efficiently~~is allocated. However, ~~the~~ data size varies in ~~many~~different application scenarios~~, e.g.,~~ such as sensor data processing, Internet traffic analysis~~,~~ and analysis of ~~various~~ transaction logic such as in web server logs and telephone calls. When ~~the~~ data size varies, the static allocation strategy leads to poor memory utilization [4]. The static strategy ~~renders inefficiency~~is thus inefficient when an application requires multiple data structures to coexist on ~~the~~ GPUs. One ~~has to~~must then resort to expensive PCIe data transfer between CPUs and GPUs, as the hash table takes up an unnecessarily large memory space. ~~To fill~~Addressing this ~~gap, it~~shortcoming calls for a dynamic GPU hash table that adjusts to the size of active entries in the table. ~~The~~Such a hash table should support efficient memory management by sustaining a guaranteed *filled factor*

• *Y. Li is with the School of Information Systems, Singapore Management University. E-mail: yuchenli@smu.edu.sg*

• *J. Zhang, Y. Liu, Z. Huang and J. Sun are with the College of Computer Science and Technology, Zhejiang University. E-mail: {zhangjing000, liuyue1013, hzd, sunjl}@zju.edu.cn*

• *Z. Lyu is with the Alibaba Group.*

~~of~~for 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~~, i.e.,~~ such as find, delete~~,~~ and insert. Although dynamically~~-~~ sized hash tables have been studied across ~~the~~ academia [25], [40] and ~~the~~ industry [23], [11] for CPUs, GPU-based dynamic hash ~~table has been~~tables have largely been overlooked.

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

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

TABLE 1 Frequently Used Notations

(k,v)	a key value pair
d	the number of hash functions
h	the ith hash table
$\backslash h^t \backslash, Ui, mi$	range, table size and data size of h
wid, l	a warp ID and the Ith lane of the warp
e	filled factor of the entire hash table
$0t$	filled factor of hash table i
loc	a bucket in the hash table

Comment [Author2]: **Remark:** Please cite table 1 and 4 in the text at appropriate instances such that the citations appear in sequential order.

indicates d lookup positions for find and delete operations~~. Apparently,~~, and a larger d indicates less workload for resizing but more lookups for find and delete operations. To mitigate th~~e~~is tradeoff, we devise a two-layer approach that first hashes any key to a pair of hash tables, where the key can be further hashed and stored in one of the ~~pair. The~~two hash tables. This design ensures that there are at most two lookups for any find and deletion operation~~s~~. Furthermore, the two-layer approach retains the general cuckoo hash tables' performance guarantee ~~of general cuckoo hash tables.~~. Empirically, the proposed hash table design ~~is capable to~~can operate efficiently at filled factors exceeding 90%. ~~Hereby~~Thus, we summarize our contributions as follows:

• We propose an efficient strategy ~~to resize the~~ for resizing hash tables and ~~our theoretical analysis has~~ demonstrated the near-~~-~~optimality of the resizing strategy through theoretical analysis.

• We devise a two-layer cuckoo hash scheme that ensures ~~at most~~a maximum of two lookups for find and deletion operations, while still retain~~s~~ing similar performance for insertion 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~~ on GPU hash tables.~~. For dynamic workloads, the proposed approach demonstrates superior~~ity~~ performance and reduces memory usage ~~of~~by up to ~~3x~~a factor of three over the compared baselines.

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

2 PRELIMINARIES

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

2.1 —Hash Table

A hash table is a fundamental data structure ~~to store~~that stores KV pairs (k,v~~)~~), and the value could refer to either ~~the~~ actual data or a reference to the data. ~~The hash table offers~~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~~if they do not; and **DELETE** (k~~)~~ –), which removes existing ~~KVs~~KV pairs that match k if they are present in the table.

Given a hash function with range 0 ~~...~~… h~~—~~ – 1, collisions must happen when we insert m > h keys into the table.

There are many schemes to resolve collisions~~:~~ such as linear probing, quadratic probing, and chaining ~~and etc.~~ ~~Contrary to~~. Unlike these schemes, cuckoo hashing [28] guarantees a worst case constant complexity for **FIND** and **DELETE**, and an amortized constant complexity for **INSERT**. A cuckoo hash uses multiple (i.e., d) hash tables with independent hash functions (h^1, h^2,~~...~~…~~,~~, h^d) and stores a KV pair in one of the hash tables. When inserting a ~~(k, v),~~KV pair, we store the pair ~~to~~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 ~~will be~~then reinserted into another hash table, e.g., loc' = $h^2(k')$. We repeat this process until encountering an empty location ~~is encountered~~.

For a hash table with the hash function $h^\%$, ¥$h^\%$ ¥ is defined to be the number of unique hash values for h^j and n_j to be the total memory size allocated for the hash table. A location or a hash value for h^l is represented as loc = hj, where j G [0, ¥ h^i¥~~—~~ – 1]. If the occupied space of the hash table is m_j, the filled factor of h^l is denoted as $0_j = m_j/n_j$. The overall filled factor of the cuckoo hash table is thus denoted as 0 =(£i mj)/(£j nj).

2.2 –GPU Architecture

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

Comment [A3]: Remark: Please clarify. Which figure is this referring to?

Memory Hierarchy. Compared to CPUs, GPUs are built with large register files ~~to~~that enable massive parallelism. Furthermore, the shared memory, which has similar performance ~~with the~~to L1 cache, can be programmed within a block to facilitate efficient memory access inside ~~a~~an SM. The L2 cache is shared among all SMs to ~~speedup memory~~accelerate access to the device memory~~. The device memory~~, which has the ~~lar~~highest capacity and ~~the~~ lowest bandwidth in the memory hierarchy.

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

• *Minimize Warp Divergence.* Threads in a warp will be serialized if ~~they execute~~executing different instructions. To enable maximum parallelism, one ~~needs to~~just 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~~reading consecutive memory locations ~~for~~to fully ~~utilizing~~use the device memory bandwidth, otherwise ~~to~~a single read instruction by a warp will trigger multiple random accesses ~~for a single read instruction by a warp~~.

• *Control Resource Usage.* Registers and shared memory are valuable resources ~~to enable~~for enabling fast local memory accesses. Nevertheless, each SM has limited resources.

~~(~~For example, the 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~~causes serious performance degradation. One can leverage the native support of atomic operations [33] on GPUs to carefully resolve these conflicts and minimize thread spinning.

3 RELATED WORKS

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

It is noted that ~~the aforementioned~~these works ~~focus~~have focused on the static case~~:~~ in which the data size for insertion is known in advance. ~~Thus, the~~The static designs would thus prepare a ~~sufficiently~~ large enough memory amount to store the hash table. In this ~~way, the~~manner, hash table operations are fast ~~since the collision~~as collisions rarely happen~~s~~. However, the static approach wastes memory resources and, to some extent, ~~it~~ prohibits coexistence with other data structures for the same application ~~to coexist on~~in the device memory. This motivates us to develop a general dynamic hash table ~~on~~for GPUs that actively ~~makes adjustments according to the~~adjusts based on data size to preserve space efficiency.

To the best of our knowledge, there is only one existing work ~~for~~on building dynamic hash tables on GPUs [4]. This proposed approach presents a concurrent linked list structure, ~~called~~known as a *slab list*, to construct the dynamic hash table with *chaining*. However, there are three major issues for slab lists. First, ~~it could~~they can frequently invoke concurrent memory allocation requests, especially when ~~the~~ data keep~~s~~ inserting. Efficient concurrent memory allocation is difficult to implement ~~under the~~in a GPU architecture ~~due to~~because of its massive parallelism. Although a dedicated memory management strategy ~~is proposed in [4]~~ to alleviate this allocation cost~~, it~~ is proposed in [4], the strategy is not transparent to other data structures. More specifically, the dedicated allocator still ~~needs to~~must reserve a large ~~piece~~amount of memory in advance to prepare for efficient dynamic allocation, and th~~e~~at occupied memory space cannot be readily accessed by other GPU~~-~~-resident data structures. Second, a slab list

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

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~~ they were~~ occupied by deleted entries. Third, the chaining approach has a lookup time of *il(log(log(m)))* for some KVs with high probability. This not only results in degraded performance for **FIND**, ~~but~~it also triggers more ~~overheads in~~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(~~O(1) worst case complexity for **FIND** and **DELETE**~~,~~ and O(1) amortized **INSERT** performance. Moreover, we ~~do~~did not introduce extra complication in implementing a customized memory manager, but ~~only reply~~rather rely on the default memory allocator provided by CUDA, ~~and,~~while at the same time~~, ensure~~ ensuring a fixed filled ratio for the hash table.

4 DYNAMIC HASH TABLE

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

4.1 Hash Table Structure

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

dynamic scenarios. For d = 3, cuckoo hashing achieves ~~over 90%~~a filled factor of more than 90% and still efficiently processes **INSERT** operations ~~efficiently~~ [12].

Figure 1 depicts the design of a single hash table h^j on GPUs. ~~Assuming the~~The keys are assumed to be 4-byte integers~~. A~~ and a bucket of 32 keys, which are all hashed to the same value hj, are stored consecutively in the memory. The design of buckets maximizes the ~~utilization of~~ memory bandwidth utilization in GPUs. Consider that the L1 cache line size is 128 bytes~~,~~. ~~o~~Only a single access is required when one warp is assigned to access a bucket. The values associated with the keys in the same bucket are also stored consecutively, but in a separate array. In other words, we use two arrays, one to store the keys and one to store the values ~~respectively. The~~ . However, the values ~~could~~can take up a much larger memory space than the keys~~. Thus~~; therefore, storing keys and

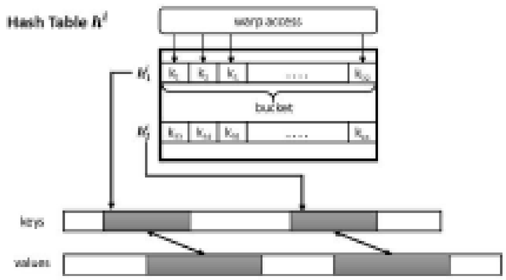


Fig. 1. The hash table structure

values separately avoids ~~the overhead of~~ memory access overhead when ~~accessing the values~~it is not necessary~~,~~ ~~e.g.,~~ to access the values, such as when 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~~fewer KV pairs in a bucket. ~~Suppose the~~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 insertion~~/~~ and 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 would occupy one bucket alone, which is unnecessarily large in practice.

4.2~~-~~ 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., [a, //]. One possible strategy to address this 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~~hash table size ~~of the hash tables~~ results in the filled factor being immediately cut ~~to~~in half, ~~while~~whereas downsizing ~~the~~ hash tables to half the original size followed by rehashing ~~could~~is only~~ be~~ efficient when the filled factor is significantly low (e.g., 40%)~~,~~. ~~b~~Both of ~~which~~these scenarios are not resource friendly. Second, rehashing all KV pairs is expensive and ~~it hurts~~harms the performance stability for most ~~of the~~ streaming applications ~~since~~as the entire hash table is subject to locking.

~~We~~Thus, we propose an alternative strategy. Given d hash tables, we always double the smallest subtable or chop the largest subtable in~~to~~ half for upsizing or downsizing, respectively, when filled factor falls out of ~~[a,/].~~the desired range. In other words, no subtable will be more than ~~double~~twice the size ~~as~~of others. Th~~e~~is strategy implies that we ~~do~~must not ~~need to~~ lock all hash tables ~~and only~~ to resize only one, thus achieving better performance stability ~~compared with~~than the aforementioned simple strategy.

Filled factor analysis: Assuming there are d' hash tables with size 2n, d~~—~~ – d' tables with size n, and ~~the~~a current filled factor of 0, one upsizing process when 0 > // lowers the filled factor to ~~d+d++1 —~~ – dTT. ~~Since~~Because the filled factor is always lower bounded by a, we can deduce that a < dzpr. Apparently, a higher lower bound can be achieved by adding more hash tables, ~~while~~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.

11

4.3 –KV distribution

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

Theorem 1. *The amortized conflicts for inserting* m *unique KV pairs to* d *hash tables ~~is~~are minimized when* ("2¹) /n₁ = ~~. . . .~~… = ("2ᵈ)/nd. m_j *and* n_j *denote the elements inserted to table* i *and the size of table i, respectively.*

Proof. ~~It is noted that the~~The amortized insertion complexity of a cuckoo hash is O(1). Thus, ~~similar to~~like a balls and bins analysis, the expected number of conflicts ~~occurred for~~occurring when inserting m_j elements in table i ~~is~~can be estimated as /n_j. Minimizing the amortized conflicts among all hash tables can be modeled as the following optimization problem:

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

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

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

~~By~~Following Jensen's inequality, the following inequality holds:

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

where ~~the~~ equality holds when /n_j = ("2³) /nj Vi, j = 1,~~...~~…, d and we obtain the minimum. ~~⊟~~

~~According to~~Based on our resizing strategy, one hash table can only be ~~as~~ 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., 0_j = 0j if n_j = nj, while 0_j ≠2 • 0j if n_j = 2nj. Thus, larger tables should have a higher filled factor. ~~Guided by~~Following Theorem 1, we employ a randomized approach~~:~~ in which a KV pair (k, v) ~~will be firstly~~is first assigned to table i with a ~~probability~~ proportional ~~to~~probability to ensure the distribution of KVs.

4.4 Rehashing

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

Upsizing. ~~We~~Here, we introduce a conflict-free rehashing strategy for the upsizing scenario. Figure 2 ~~presents an illustration for~~ illustrates the upsizing of a hash table h^j. As we always double the size for h^j, a KV pair ~~which~~that originally resides in bucket loc could be rehashed to bucket loc+ ¥ h^j ¥ 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 ~~corporately~~ takes a KV pair in the bucket and, if necessary, relocates th~~e~~at KV pair ~~if necessary.~~. Moreover, ~~the~~ rehashing does not trigger any ~~conflict since~~conflicts as KV pairs from two distinct

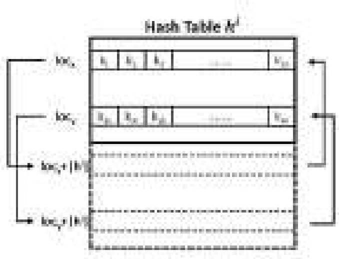


Fig. 2. Illustration for upsizing and downsizing.

buckets before upsizing cannot be rehashed to the same bucket. Thus, ~~the~~ locking ~~of a~~the bucket is not required, ~~and~~meaning we can make use of the device's full ~~device~~ memory bandwidth for the upsizing process.

~~Note that after~~After upsizing ~~hash table~~ h^j, its filled factor θ_j is cut ~~to~~in half, which could break the balancing condition emphasized in Theorem 1. Nevertheless, we use ~~the~~a sampling strategy for subsequent KV insertions, ~~where~~in which each insertion is allocated to table i with a probability proportional to "2*), to recover the balancing condition. In particular, mj remains the same ~~while~~but nj doubles after upsizing, and the scenario leads to doubl~~e~~ing the probability of inserting subsequent KV pairs to h^j.

Downsizing. Downsizing ~~the hash table~~ h^j is ~~a~~the reverse process of upsizing h^j. ~~We note that there~~There is always room to relocate KV pairs in the same table for upsizing~~. In contrast~~; however, downsizing may rehash some KV pairs to other hash tables, especially when θ_j > 50%. ~~Since~~Because the KV pairs located in loc and loc + ¥hj¥ are hashed to loc in the new table, there could be cases in which the KV pairs exceed~~s~~ 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. ~~Similar as~~Like upsizing, ~~it~~downsizing does require locking ~~since~~as there will be no thread conflict on any bucket. For the remaining KV pairs ~~which~~that cannot fit in the downsized table, ~~called~~known as *residuals,* we insert them into other subtables. To ~~make sure~~ensure no conflict occurs between inserting residuals and processing the downsizing subtable, both of which are ~~both~~ executed in a single kernel, we exclude the downsizing subtable when inserting the residuals. ~~Take~~As an example, when we have three subtables and one of them is being downsized~~.~~, ~~W~~we only insert the residuals to the remaining two subtables.

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

5 TWO-LAYER CUCKOO HASH

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

5.1 ~~—~~The Two-~~l~~Layer Approach

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

One possible approach to reduce the number of lookups is to first hash all KV pairs into d' partitions. For each partition, we employ a cuckoo hash with two hash functions. In this ~~way~~manner, one must only ~~needs to~~ perform two lookups for any **FIND** and **DELETE** operation~~s~~. However, this approach cannot prevent ~~the~~ skewness ~~issue~~ across the d' partitions, especially ~~against~~with frequent delete operations. It is possible that the deleted KVs all fall into one partition cj, which results in low filled factor for the table~~/s~~ or tables allocated to c_j. Furthermore, when inserting KVs into other partitions, e.g., cj = c_j, ~~the~~ efficiency could be severely degraded due to the high filled factor of the table~~/s~~ or tables allocated to cj.

Hence, we propose a two-layer approach to resolve the skewness ~~issue. The two-layer approach~~problem, which is inspired by ~~the~~ data partitioning techniques [6], [7], [37], [40]. Given d hash tables, we first hash all KV pairs into Q partitions~~. Each partition~~, each of which refers to a unique pair of hash tables. Then~~,~~ each KV pair is hashed and stored in only one of the corresponding pair. ~~In this way, it~~This only requires ~~at most~~a maximum of two lookups for **FIND** and **DELETE**. The advantage of this approach is that each KV pair could appear in any of the d hash tables, which ~~offers~~provides opportunities to balance a skewed distribution. The following example illustrates a scenario where ~~the~~ skewness is mitigated during the insertion process.

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

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

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

Proof. Assuming d hash tables for the two-layer approach, ~~W.L.O.G.~~without loss of generality we set ~~[0, n) to be~~ the range for each hash function~~.~~ to be [0, n). Given a KV pair (k,v), we denote ~~that~~ hash function hp as the one ~~which~~that hashes (k, v) to a pair of hash tables.

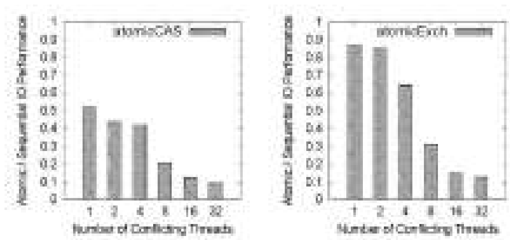


Fig. 3. The performance of atomic operations for increasing conflicts

Now, we transform the two-layer approach to the plain cuckoo hash ~~as the following: we construct~~by constructing two new hash functions, $H_1(k) = i \cdot n \cdot h_i(k)$ and $H_2(k) = j \cdot n \cdot hj\ (k$~~)~~)), where hp(k) = ($h_j$, hj). ~~Apparently, the~~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 ~~the~~ U ~~are~~represents the buckets for H_1, V ~~are~~represents the buckets for H_2, and E ~~are~~represents the KV pairs connecting the two buckets from H_1 and H_2 ~~respectively. We note that each~~. Each KV pair is independently hashed to a random edge e G E with the same probability ~~independently~~, i.e., $1/(n^2d^2)$. Hence, we can follow ~~the~~a similar proof procedure~~, which~~ that utilizes random bipartite graph analysis to show the amortized complexity ~~for~~of a cuckoo hash with two tables [22], to prove Theorem 2.

5.2 ~~–~~Parallel Hash Table Operations

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

Find. It is relative straightforward to parallelize **FIND** operations ~~since~~as only read access is required. Given a batch of size m, we launch w warps ~~in total (which means launching~~(meaning we launch 32w threads ~~in total)~~ ~~and~~), with each warp ~~is~~being responsible for |_J**FIND** operations. To locate a KV pair, we ~~need to~~must hash the key to a hash table pair (~~-~~hj, hj) and perform ~~at most~~a maximum of two lookups in the corresponding buckets of hj and hj, respectively.

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~~a warp-centric approach to access a bucket. On the other hand, when updating a bucket, a warp requires a mutex ~~when updating a bucket~~ to avoid corruption, and ~~the~~on GPUs locking is expensive ~~on GPUs~~. In the literature, it is a common practice to use atomic operations for implementing a mutex under ~~the~~a warp-centric approach [36]. ~~In particular, we~~We can still invoke a warp to insert a KV pair~~. The~~; however, the warp ~~is required to~~must 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~~to this direct warp-centric approach. First, the conflicting warps ~~are spinning~~spin while locking, thus wasting computing resources. Second, although atomic operations are natively supported by recent GPU architectures ~~natively~~, they become costly when the number of atomic operations issu~~ing~~ed at the same location increases. In Figure 3, we show the profiling statistics for two atomic operations ~~which~~that are often used to lock and

Algorithm 1 **Insert**(lane l, warp wid)

```
active ← 1
while true do
    l' ← ballot(active == 1)
    if l' is invalid then
        break
    [(k', v'), i'] ← broadcast(l')
    loc = h^{i'}(k')
    if l' == l then
        success ← lock(loc)
    if broadcast(success, l') == failure then
        continue
    l* ← ballot(loc[l].key == k' || loc[l].key == ∅)
    if l* is valid and l' == l then
        loc[l*].(key, val) ← (k', v')
        unlock(loc)
        active ← 0;
        continue
    l* ← ballot(loc[l].key ≠ ∅)
    if l* is valid and l' == l then
        swap(loc[l*].(key, val), (k', v'))
        unlock(loc)
```

unlock a mutex~~:~~, atomicCAS, and ~~and~~ atomicExch, respectively. We compare the throughputs of the atomic operations ~~vs.~~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 ~~has~~ seriously degrade~~d~~s when a larger number of conflicts occur. Thus, it will be expensive for the direct warp-centric approach in ~~the~~ contention critical cases. ~~Imaging~~Suppose that one wants to track the number of retweets posted to active ~~t~~Twitter accounts in the current month~~, via~~ by storing the ~~t~~Twitter ID and the obtained retweet counts as KV pairs. In this ~~particular~~ scenario, certain ~~t~~Twitter celebrities could receive thousands of retweets in a very short period. This ~~triggers~~causes the same ~~t~~Twitter ID ~~gets~~to get updated frequently ~~and~~, thus a ~~serious~~large number of conflicts would happen.

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

Parallel insertion with ~~the~~a voter coordination scheme is presented in Algorithm 1. The ~~pseudo code~~pseudocode in Algorithm 1 demonstrates how a thread (with lane l) from warp wid inserts a KV pair. The warp first conducts a vote ~~with~~among active threads using the ballot function ~~among active threads~~ and the process ~~will terminate~~terminates if all threads finish their tasks (lines 1~~-~~–5). This achieves better resource utilization ~~since~~as no thread will be idle when ~~one of the threads~~another thread in the same warp is active. The leader l' then broadcasts its KV pair (k',v') ~~as well as the~~ and hash table h_j/ to the warp and ~~tries~~attempts to lock the inserting bucket (lines 69). ~~Note that the~~The ballot and broadcast functions are implemented ~~with~~using the CUDA primitives _ballot and _shfl. The

broadcast function ensures that all threads in the warp receive the locking result, and ~~the warp revotes~~ if l' fails to obtain ~~the~~a lock, the warp revotes. Otherwise, the warp follows l' and proceeds to update the bucket for (k', v') with a warp-centric approach ~~similar to~~like FIND. Once a thread finds k' or an empty space in the bucket, l' ~~will add~~adds or ~~update~~updates it with (k', v') (lines 12-–17). ~~When there is~~ If no empty slot is found, l' swaps (k', v') with another KV pair (k*, v*) in the bucket and inserts the evicted KV pair to hash table j in the next round. The warp finishes the ~~work~~process when all KV pairs ~~are~~have been inserted.

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

~~We give the~~The following example ~~to demonstrate~~demonstrates the parallel insertion process.

Example 2. *In Figure 4, we visualize ~~the~~a scenario for three threads~~:~~, l_x, l_y, l_z, from warp a and warp b, ~~inserting~~which insert KV pairs* (k_1, v_1), (k_{33}, v_{33}), and ($k65$, v_{65}) *independently. Suppose that* l_y *and* l_z *become the leaders ~~ofwarp~~of warp* a *and warp* b, *respectively. Both threads will compete for ~~the~~ bucket* y, *and* l_z *wins the battle~~. l_z will~~ and then leads warp* b *to inspect the bucket and evict KV pair* ($k64$, v 64) *by replacing it with* (k_{65}, v_{65}). *In the mean~~wh~~t~~i~~ime,* l_y *does not lock ~~on~~ bucket* y *and ~~joins~~ 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 ~~get back~~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. ~~The DELETE operation, in~~In contrast ~~to~~with INSERT, the DELETE operation does not require locking with a warp-centric approach. ~~Similar to~~As with FIND, we assign a warp to process ~~a~~ key k on deletion. The warp iterates through the buckets ~~among~~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 ~~only erase~~erasing the key only if k is found, thus caus~~es~~ing no conflict.

Complexity. ~~Since a~~Because FIND, INSERT, and DELETE ~~operation is~~operations are independently executed by ~~a thread. The~~ threads, the analysis of a single thread's complexity is the same as in the sequential version of cuckoo hashing [28~~].~~], which is an $O(1)$ worst case complexity for FIND and DELETE~~,~~ and an $O(1)$ expected time for INSERT for the case of ~~2~~two hash tables. It has been pointed out that analyzing the theoretical upper bound complexity of insertion in d ~~—~~ ≥ 3 hash tables is ~~hard~~difficult [2]. Nevertheless, empirical results have shown that increasing the number of tables leads to better

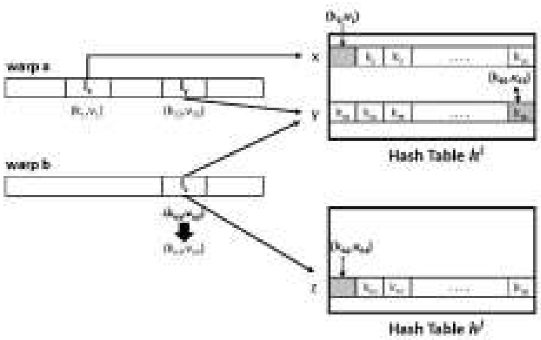


Fig. 4. Example for parallel insertions

insertion performance. Please refer to ~~our~~the experimental results presented in Section 6.

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

6 EXPERIMENTAL EVALUATION

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

6.1 Experimental Setup

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

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

TABLE 2

The datasets used in the experiments.

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

• Slab is a state-of-the-art GPU-based dynamic hash table [4~~], which~~] that 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 ~~codes of~~code for DyCuckoo ~~are~~is released[2]. ~~For~~Performance numbers for GPU-based solutions~~, all performance numbers~~ are calculated based purely ~~based~~ on ~~the~~GPU run-time ~~on GPUs~~. The overhead of data transfer between CPUs and GPUs can be hidden by overlapping ~~the~~ data transfer and GPU computation, as proposed ~~in~~by MegaKV [36]. ~~Since~~Because this technique is orthogonal to the approaches proposed in our paper, we ~~thus~~ focus solely on GPU computation ~~only~~.

Datasets. We evaluate all compared approaches using several real world datasets, and ~~the~~a summary of those datasets can be found in Table 2.

• TW: Twitter is an online social network where users perform the actions ~~include~~ *tweet, retweet, quote,* and *reply*. We crawl these actions for one week ~~via~~through the Twitter stream API[3] ~~on~~for the following trending topics ~~US president~~: U.S. presidential election, 2016 NBA finals, and Euro 2016. The TW dataset contains 50,876,784 KV pairs.

• RE: Reddit is an online forum where users perform the actions ~~include~~ *post* and *comment*. We collect all Reddit *comment* actions in May 2015 from *kaggle*[4] and query the Reddit API for ~~the~~ *post* actions during the same period. The RE dataset contains 48,104,875 ~~actions as~~ KV pairs.

• LINE: Lineitem is a synthetic table generated by the TPC-H benchmark[5]. We generate 100,000,000 rows of data in the ~~lineitem table~~LINE dataset and combine the *orderkey, linenumber,* and *partkey* column as ~~keys~~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.

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

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

Dynamic Hashing Comparison (Section 6.4). ~~Under~~We generate workloads under the dynamic setting~~, we generate the workloads~~ by batching ~~the~~ hash table operations. We then partition the datasets into

2. [[ADD link to the open source repository.]]

3. https://dev.twitter.com/streaming/overview

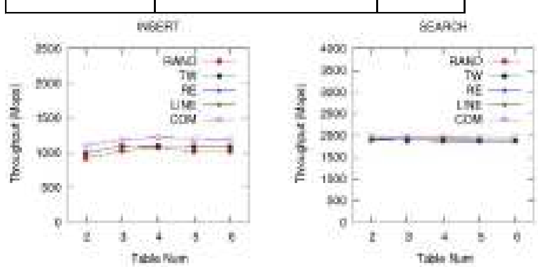
4. https://www.kaggle.com.reddit/reddit-comments-may-2015

5. https://github.com/electrum/tpch-dbgen

TABLE 3

Parameters in the experiments

Parameter	Settings	Default
Filled Factor e	70%, 75%, 80%, 85%, 90%	85%
Lower Bound *a*	20%, 25%, 30%, 35%, 40%	30%
Upper Bound ///	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



INSERT FIND

Fig. 5. Throughput of DyCuckoo for varying the number of hash tables.

batches of 1 million insertions. For each batch, we augment 1 million **FIND** operations and 1 • r million **DELETE** operations, where r is a parameter ~~to balance~~for balancing insertions and deletions. After ~~we exhaust~~exhausting all the batches, we rerun the~~se~~ batches by swapping the **INSERT** and **DELETE** operations in each batch. We then evaluate the performance of all compared GPU approaches except for CUDPP and Warp as they do not support deletions. We also exclude the CPU approach as it is significantly slower than the GPU-based approaches. ~~Since~~Because `MegaKV` does not provide dynamic resizing, we double/half the memory usage followed by rehashing all KV pairs as ~~its~~a resizing strategy~~,~~ if the corresponding filled factor falls out of the specified range. Moreover, if an insertion failure is found for a compared approach, we trigger its resizing strategy.

Parameters. We vary the parameters when comparing `DyCuckoo` with the baselines. Here, a ~~is~~represents the lower bound ~~on the~~for filled factor 0 for all compared approaches, ~~whereas~~ // is the respective upper bound~~.~~ and r is the ratio of insertions over deletions in a processing batch. The parameter settings ~~of the aforementioned parameters could be found~~are listed in Table 3. For all experiments, we use *million operations/seconds* (Mops) as a metric to measure the performance of all compared approaches.

Experiment Environment. [[Update the environment specs.]] We conduct all experiments on an Intel Xeon E5-2620 Server equipped with an NVIDIA GeForce GTX 1080~~. Evaluations~~, and evaluations are performed using CUDA 9.1 on Ubuntu 16.04.3. The optimization level (-O3) is applied for compiling all programs.

TABLE 4

The number bucket accessed for subtable resizing.

	UPSIZE		UPSIZE	
	DyCuckoo	Rehash	DyCuckoo	Rehash
TW	1133	314,444	701,055	1,386,990
RE	1024	314,485	704,030	1,388,938
LINE	1062	314,528	707,677	1,390,672
COM	1081	314,326	702,058	1,387547
RAND	1021	314,453	706,265	1,387, 034

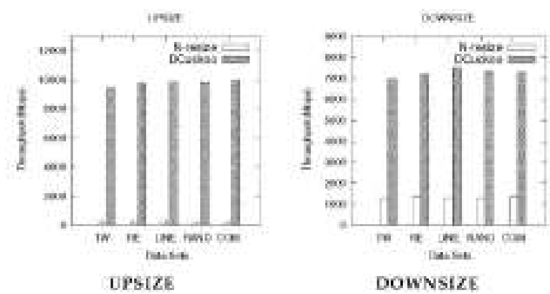


Fig. 6. Throughput of subtable resize.

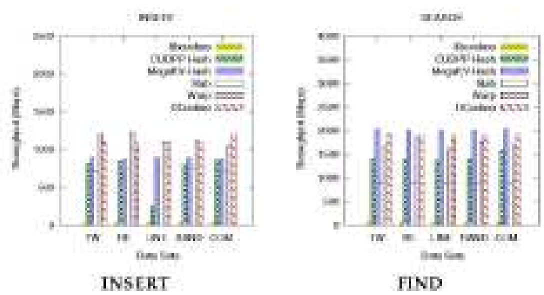


Fig. 7. Throughput of all compared approaches under the static setting.

6.2 Sensitivity Analysis

Varying the number of tables. A key parameter ~~that affects~~affecting the performance of `DyCuckoo` is the number of hash tables chosen. For the static scenario, we present the throughput performances of **INSERT** and **FIND** for a varying number of hash tables, as shown in Figure 5, while fixing the entire structure's memory space ~~of the entire structure~~ to ensure ~~the~~a default filled factor of 0. ~~The throughput of~~ **INSERT** increases its throughput with more hash tables~~, since~~ because there are more alternative locations ~~for relocating a~~to relocate KV ~~pair. The~~pairs. However, performance slightly degrades with more than four tables~~ beyond 4. As~~. This is because the total allocated memory is fixed, ~~the size of~~and thus each table becomes smaller for more tables. This leads to more evictions for some overly occupied tables~~ and degrades~~, thus slightly degrading the performance~~ slightly~~. Another interesting observation is that we achieve the best performance ~~on~~with the `COM` dataset. This is because `COM` has the smallest ratio of unique keys (Table 2). Inserting an existing key is equivalent to an update operation, which ~~results in~~has better performance compared ~~with that when~~than inserting a new key. The throughput of **FIND** remains constant for additional hash tables, as ~~the~~ two-layer cuckoo hashing guarantees ~~at most~~a maximum of two look ups for **FIND**. In the ~~remaining part~~remainder of this section, we fix the number of hash tables ~~to be 4~~at four.

Resizing analysis. To validate the proposed resizing strategy's effectiveness~~ of our resizing strategy proposed~~, we compare it with rehashing. For upsizing~~,~~ we initialize DyCuckoo with all ~~the data~~0data and set the filled factor as the default upper bound of 85%. ~~Then, we~~We perform a one-time upsizing, i.e., we upsize one subtable, and then compare our resizing strategy against rehashing all ~~the entries in the~~ subtable entries by reinserting the entries ~~with~~using Algorithm 1. ~~For~~ The setup for downsizing~~, the setup~~ is a mirror image of ~~the~~ upsizing~~ evaluation~~, except with with an initial~~izing~~ filled factor set as the default lower bound of 30%. ~~The~~ Figure 6 shows the throughput~~ is reported in Figure 6. The~~. Rehashing throughput ~~of rehashing~~ for the upsizing scenario is severely limited~~, since~~ as the remaining subtables that are not being upsized are almost filled and inserting KV pairs ~~resulting~~results in frequent evictions. In comparison, the downsizing throughput of reinsertion is significantly faster ~~due to~~because of a low filled ratio. Our resizing strategy also achieves dramatic speedups over rehashing for downsizing~~ as well. Besides, it~~. Additionally, our resizing strategy only locks the subtable being resized and supports concurrent updates for the remaining subtables.

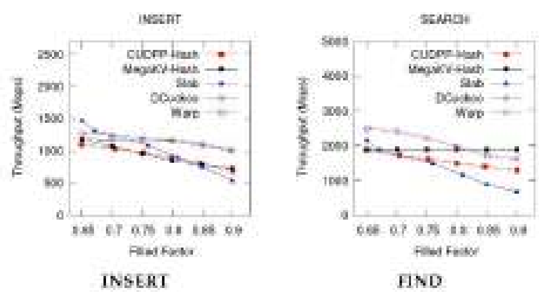


Fig. 8. Throughput of all compared approaches for varying the filled factor against the RAND dataset.

6.3 Static Hashing Comparison

Throughput Analysis. ~~In~~ Figure 7~~, we present~~ shows the throughput of all compared approaches over all datasets under ~~the~~ default ~~setting.~~settings. The GPU-based approaches significantly outperform the CPU-based baseline ~~significantly.~~. For **INSERT**, Warp demonstrates the best performance overall~~. This is~~, because Warp employs ~~the~~a linear probing strategy~~, which~~ that achieves better cache locality than the cuckoo strategy [19]. Nevertheless, DyCuckoo shows competitive throughput. For **FIND**, MegaKV shows the best performance at the default filled factor ~~since~~as it simply checks two buckets ~~for locating~~to locate a KV pair. Although DyCuckoo also checks two buckets, it has slightly inferior performance ~~than~~to MegaKV ~~as~~because DyCuckoo employs another layer of hashing that adds cost to the overall performance. ~~As~~ Slab employs a chaining approach; therefore, it requires more random accesses to locate a KV pair along the chain when a high filled factor is required. Hence, `Slab` has inferior performance ~~than~~to other GPU-based solutions.

Varying filled factor 0. We vary the filled factor 0 and show the performance of all GPU-based approaches against the `RAND` dataset in Figure 8. The other datasets show similar ~~trend and~~trends, thus we omit those results in theis paper. For `Slab`, the filled factor dramatically affects the performance of both **INSERT** and **FIND** ~~is dramatically affected by the filled factor.~~. This is because `Slab` employs ~~the~~a chaining approach~~, where~~ in which a high filled factor leads to long chains and poor performance. Overall, `Warp` demonstrates superior performance for ~~the~~a static setting. As ~~aforementioned~~previously mentioned, `Warp` has better cache locality than the other approaches. `DyCuckoo` shows competitive performance and could outperform `Warp` ~~at~~for high filled factors~~,~~ (e.g., 0 ~~—~~ = 0.85~~.~~).

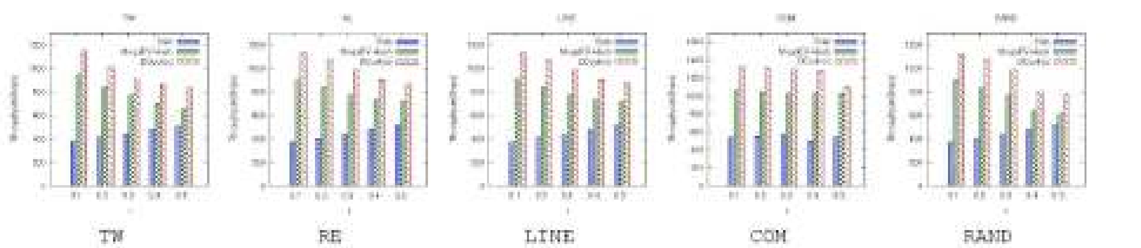


Fig. 9. Throughput for varying the ratio r.

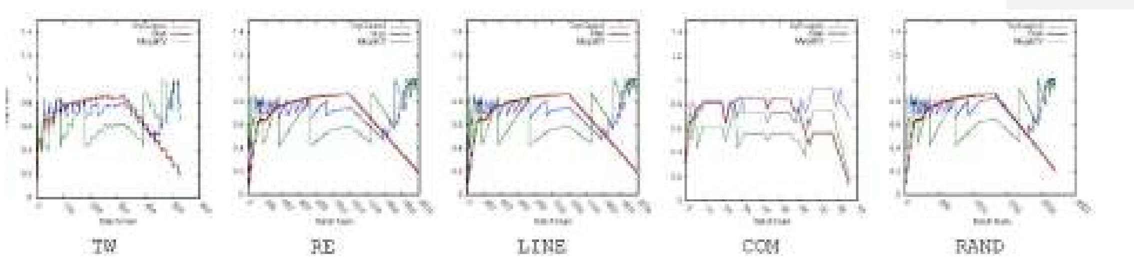


Fig. 10. Tracking the filled factor.

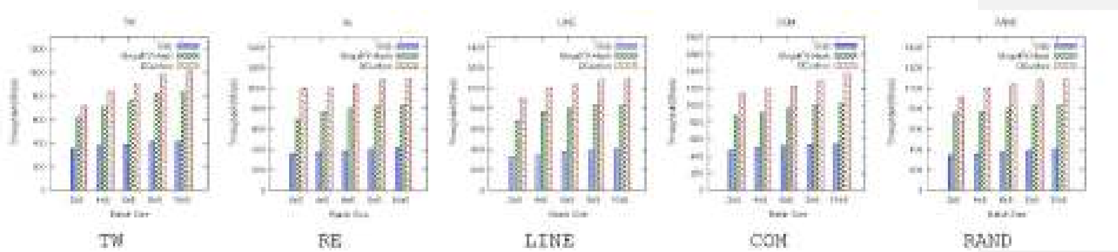


Fig. 11. Throughput for varying ~~Batch Size~~ batch size.

Furthermore, the linear probing strategy ~~adopted by~~ Warp adopts may need to scan multiple memory positions for **FIND** and **DELETE**. Although CUDPP also employs the cuckoo strategy, it automatically determines the number of hash tables for a given data size. At high filled factors, CUDPP employs more hash tables and scans more buckets~~.~~; ~~H~~hence, its performance ~~drops~~decreases for high filled factors. In contrast, DyCuckoo and MegaKV ~~with~~use the cuckoo strategy but only inspect two buckets, which ~~explain their~~explains the stable throughput shown in Figure 8 for **FIND**. Furthermore, DyCuckoo shows better insertion performance than MegaKV ~~due to~~because our proposed strategy ~~for resolving conflicts~~ efficiently resolves conflicts.

Comment [Author4]: **Tip:** Semicolon: When adverbs, such as however and therefore, join two independent sentences, use a semicolon before and a comma after them. Some other examples of the same kind include hence, thus, nevertheless, and moreover.

6.4 Dynamic Hashing Comparison

Varying insert vs. delete ratio r. In Figure 9, we report the results for varying the ratio r, ~~i.e.,~~which is number of deletions over the number of insertions in a batch. We ~~have an interesting observation~~found that~~,~~ for a larger value of r, the performance of DyCuckoo and MegaKV degrades whereas that of Slab improves. ~~As a~~A larger value of r indicates more deletions, thus resizing operations are more frequently invoked for DyCuckoo and MegaKV. In contrast, ~~more~~a greater number of deletions leads to additional vacant spaces for Slab, as ~~it~~that technique simply symbolically marks a deleted KV pair. Insertions are processed more efficiently for Slab ~~since~~because the inserted KV pairs can overwrite ~~the~~ symbolically deleted ones. Hence, Slab utilizes more GPU device memory than DyCuckoo and MegaKV. However, symbolic deletions cannot guarantee a bounded filled factor and may lead to arbitrary bad memory efficiency~~, which~~. This will be discussed with more experimental results later in this section. DyCuckoo shows the best overall performance. Furthermore, the throughput margin between DyCuckoo and MegaKV ~~grows~~increases for larger r values. As mentioned previously, a larger r~~. As aforementioned, larger r~~ triggers more resizing operations, ~~where~~thus DyCuckoo is more efficient than MegaKV ~~since~~as MegaKV ~~employs~~uses a total rehashing approach.

Performance stability. We evaluate ~~the~~compared approaches' performance ~~stability of the compared approaches~~stabilities in Figure 10~~. In particular, we track~~, tracking the filled factor after processing each batch. Slab shows good stability in terms of memory usage for the starting phases. Unfortunately, ~~due to~~because of the symbolic deletion approach employed, ~~the~~Slab's memory efficiency ~~of Slab~~ degrades significantly as more deletions are processed. In particular, its filled factor drops to less than 20% after processing ~~less~~fewer than 100 batches for the COM dataset, which deems ~~for~~ a complete rebuild. MegaKV shows an unstable trend

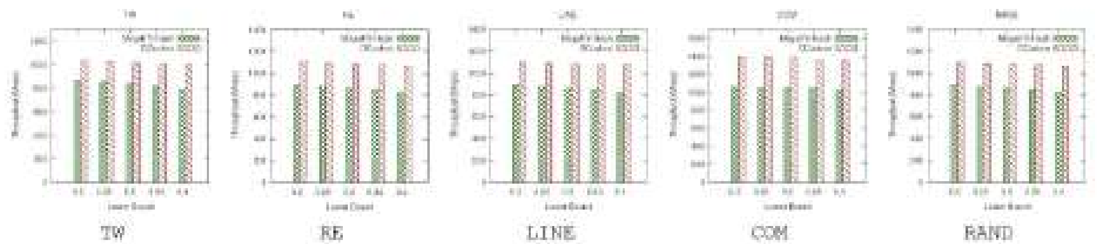


Fig. 12. Throughput for varying α~~-~~.

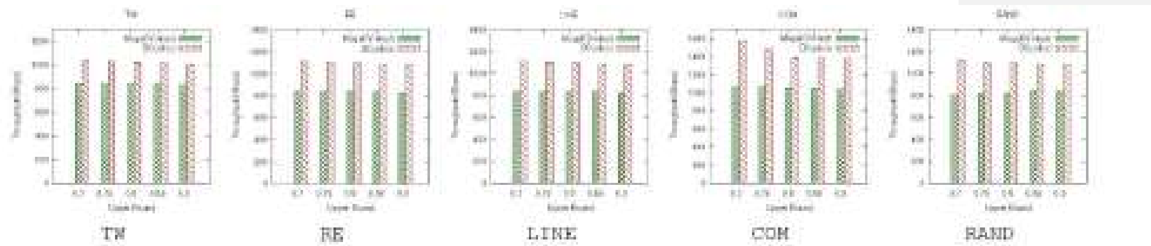


Fig. 13. Throughput for varying ~~l~~l.

~~since~~as it employs a simple double/half approach for resizing. We can see that its filled factor dramatically jumps up or down ~~dramatically~~ at the ~~points of~~ resizing points. DyCuckoo ~~demonstrates~~shows the best overall stability and ~~saves~~ has a memory savings of up to **[[check this number: 4x]]** ~~memory~~ over the compared baselines (the COM dataset). These results have validated our design~~ where~~, with only one of the subtables ~~are~~being subject to resizing. Nevertheless, there is still room~~s~~ for improvement. We note that for some datasets, i.e., TW, RE, LINE, and RAND, ~~we observe that~~ the filled factor of DyCuckoo drops sharply. This is because, even after upsizing one time~~ of upsizing~~, the insertions fail due to too many evictions~~ and it~~, which triggers another round of upsizing. We leave ~~it~~this as ~~a~~an area for future work.

Varying the batch size. We ~~have~~ also varied ~~the size of~~ each processing ~~batch. The~~ batch's size, and the results are ~~reported~~shown in Figure 11. Slab ~~remains~~continues to show inferior performance ~~than~~to MegaKV and DyCuckoo~~. This is~~ because Slab accommodates new inserted KV pairs with ~~the~~a chaining approach and does not increase the ~~range of the~~ unique hash ~~values~~value range. Hence, a stream of insertions will eventually lead to long chains, which hurts ~~the~~hash table operation performance~~ of hash table operations.~~. Furthermore, DyCuckoo ~~presents~~shows better performance than MegaKV, and the margin increases with ~~a~~ larger batch ~~size. Note that one~~sizes. One limitation of existing GPU-based approaches is that they apply updates at the ~~granularity of~~

Comment [Author5]: **Tip:** American English; Oxford comma: In American English, a comma is inserted before the coordinating conjunction preceding the last item in a list of three or more items. This comma, which was introduced by the Oxford University Press (hence called Oxford comma), is referred to as a serial comma.

~~batches~~batch level. It is an interesting direction for exploring efficient GPU hashing when a required update order is enforced.

Varying the filled factor lower bound a. We vary the filled factor's lower bound ~~of the filled factor~~ and report the results in Figure 12. We only compare MegaKV and DyCuckoo ~~since~~because Slab ~~is unable to~~cannot control the filled factor because of ~~Slab's~~its symbolic deletion approach. Apparently, the simple resizing strategy adopted by MegaKV incurs substantial overhead. Such overhead ~~grows for a~~increases with higher a ~~since~~values because the number of downsizings increases. ~~The~~DyCuckoo's performance ~~of DyCuckoo~~ is not affected significantly ~~due to~~because the incremental resizing approach ~~by updating~~updates only one subtable at a time.

Varying the filled factor upper bound //**.** The results for varying / is ~~reported~~shown in Figure 13. ~~It is interesting to see that the~~The upper bound does not significantly affect the overall performance for either MegaKV and DyCuckoo. ~~On one hand,~~A higher filled factor leads to slower INSERT performance~~. On the other hand, less number of rehashing~~; however, fewer rehashings are incurred for a higher filled factor. Thus, the overall performance remains stable for both approaches ~~as the~~because these opposing factors cancel each other. Nevertheless, DyCuckoo remains superior over MegaKV in terms of time efficiency ~~while~~but substantially saves GPU memory ~~substantially~~.

7 CONCLUSION

In this paper, we contribute ~~a number of~~several novel designs for a dynamic hash table on GPUs. First, we introduced an efficient strategy to resize ~~only~~ one ~~of the subtables~~subtable at a time. Our theoretical analysis demonstrated the resizing strategy's near-~~-~~optimality ~~of the resizing strategy.~~. Second, we devised a two-layer cuckoo ~~has~~ scheme that ensures ~~at most~~a maximum of two loops for find and deletion operations, while still retain~~s~~ing similar performance ~~for insertion as~~ to general cuckoo hash tables~~.~~ for insertion. Empirically, our proposed design achieves competitive performance against other state-of-the-art static GPU hash ~~tables.~~table techniques. Our hash table ~~design achieves~~designs achieve superior performance while sav~~es~~ing up to ~~4x~~ four times the memory over ~~the~~ state-of-the-art approaches against dynamic workloads.

REFERENCES

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

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

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

2011.

[4] S. Ashkiani, M. Farach-Colton, and J. D. Owens. A dynamic hash

table for the gpu. In *IPDPS,* pages 419-429. IEEE, 2018.

[5] P. Bakkum and K. Skadron. Accelerating sql database operations

on a gpu with cuda. In *GPGPU,* pages 94-103. ACM, 2010. [6] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: how to cache your hash on flash. *Proceedings ofthe VLDB Endowment,* 5(11):1627-1637, 2012.

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

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

2010.

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

[10] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *ICML,* pages 13371345, 2013.

[11] J. R. Douceur. Hash table expansion and contraction for use with

internal searching, May 23 2000. US Patent 6,067,547.

[12] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. *Theory ofComputing*

Systems, 38(2):229-248, 2005.

[13] I. Garcia, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel

hashing. In *TOG,* volume 30, page 161. ACM, 2011.

[14] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors.

TODS, 34(4):21, 2009. [15] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD,*

pages 511-524. ACM, 2008. [16] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB,* 6(9):709-720, 2013.

[17] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. Mem-

cachedgpu: Scaling-up scale-out key-value stores. In *SOCC,* pages

43-57. ACM, 2015.

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

217-226. ACM, 2010.

[19] D. Junger, C. Hundt, and B. Schmidt. Warpdrive: Massively parallel hashing on multi-gpu nodes. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS),* pages 441450. IEEE, 2018.

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

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

pages 63-74. IEEE, 2015.

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

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

Patent 6,578,131.

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

page 27. ACM, 2014.

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

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

2013.

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

[28] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal ofAlgorithms,*

51(2):122-144, 2004.

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

2243-2248. IEEE, 2010.

[30] J. Pan and D. Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *SIGSPATIAL,* pages 211220. ACM, 2011.

[31] M. Raab and A. Steger. "balls into bins"—a simple and tight

analysis. In *RANDOM,* pages 159-170. Springer, 1998.

[32] K. A. Ross. Efficient hash probes on modern processors. In *ICDE,*

pages 1297-1301. IEEE, 2007.

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

2010.

[34] M. B. Taylor. Bitcoin and the age of bespoke silicon. In *CASES,*

page 16. IEEE Press, 2013.

[35] Z. Wu, Y. Liu, J. Sun, J. Shi, and S. Qin. Gpu accelerated on-the-fly

reachability checking. In *ICECCS,* pages 100-109. IEEE, 2015. [36] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-

kv: a case for gpus to maximize the throughput of in-memory

key-value stores. *PVLDB,* 8(11):1226-1237, 2015.

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

2019.

[38] J. Zhong and B. He. Medusa: Simplified graph processing on gpus.

TPDS, 25(6):1543-1552, 2014.

[39] J. Zhou, K.-M. Yu, and B.-C. Wu. Parallel frequent patterns mining

algorithm on gpu. In *SMC,* pages 435-440. IEEE, 2010. [40] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18),* pages 461^76, 2018.

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

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

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

Zheng Lyu is now a staff engineer at Alibaba group, ~~and~~where he is responsible for GPU database development~~ of GPU databases.~~. He received his PhD in communication and information systems from Shanghai ~~i~~Institute of ~~m~~Microsystems and ~~information technology~~Information Technology, Chinese Academy of Sciences in 2012. He mainly works in the area of high~~-~~-performance computing and his major research interest is heterogeneous computing in database systems.

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

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