Data Science and Engineering (2021) 6:265–279
https://doi.org/10.1007/s41019-021-00161-5

GPU-Based Dynamic Hyperspace Hash with Full Concurrency

Zhuo Ren[1] · Yu Gu[1] · Chuanwen Li[1] · FangFang Li[1] · Ge Yu[1]

Received: 19 January 2021 / Revised: 16 March 2021 / Accepted: 18 April 2021 / Published online: 17 June 2021

Abstract
Hyperspace hashing which is often applied to NoSQL data-bases builds indexes by mapping objects with multiple attributes to a multidimensional space. It can accelerate processing queries of some secondary attributes in addition to just primary keys. In recent years, the rich computing resources of GPU provide opportunities for implementing high-performance HyperSpace Hash. In this study, we construct a fully concurrent dynamic hyperspace hash table for GPU. By using atomic operations instead of locking, we make our approach highly parallel and lock-free. We propose a special concurrency control strategy that ensures wait-free read operations. Our data structure is designed considering GPU specific hardware characteristics. We also propose a warp-level pre-combinations data sharing strategy to obtain high parallel acceleration. Experiments on an Nvidia RTX2080Ti GPU suggest that GHSH performs about 20-100X faster than its counterpart on CPU. Specifically, GHSH performs updates with up to 396 M updates/s and processes search queries with up to 995 M queries/s. Compared to other GPU hashes that cannot conduct queries on non-key attributes, GHSH demonstrates comparable building and retrieval performance.

Keywords Hyperspace hashing · Merge access · Warp-level pre-combinations data sharing · Atomic operations

1 Introduction

NoSQL databases such as key-value stores are increasingly prevalent in big data applications for their high throughput and efficient lookup on primary keys. However, many applications also require queries on non-primary attributes. For instance, if a tweet has attributes such as tweet id, user id, and text, then it would be useful to be able to return all tweets of a user. But supporting secondary indexes in NoSQL databases is challenging because secondary indexing structures must be maintained during writes , while also managing the consistency between secondary indexes and data tables, which is a commonly supported feature in SQL databases but deficiently supported by NoSQL. For instance, if a tweet has attributes such as tweet id, user id, and text, then it would be useful to be able to return all (or the most recent) tweets of a user. However, supporting secondary indexes in NoSQL databases is challenging, because secondary indexing structures must be maintained during writes, while also managing the consistency between secondary indexes and data tables. To solve this issue, hyperspace hashing is proposed in HyperDex system [1] for distributed key-value stores that supports retrieving partially-specified secondary attribute searches in addition to primary keys. Compared to the method of stand-alone secondary indexes (e.g. table-based secondary index in Hbase [2]), hyperspace hashing can greatly save storage space, which is particularly important for in-memory databases. Compared to the method of embedded secondary indexes like KD-tree [3], hyperspace hashing can quickly locate the hash bucket where the data is located, without judging each layer in order. Hyperspace hashing represents each table as an independent multidimensional space, where the dimension axis directly corresponds to the attributes of the table. An object is mapped to a deterministic coordinate

✉ Zhuo Ren
1801807@stu.neu.edu.cn

Yu Gu
guyu@mail.neu.edu.cn

Chuanwen Li
lichuanwen@mail.neu.edu.cn

FangFang Li
lifangfasng@mail.neu.edu.cn

Ge Yu
yuge@mail.neu.edu.cn

[1] School of Computer Science and Engineering, Northeastern University, Shenyang 110819, Liaoning, China

in space by hashing each attribute value of the object to a location on the corresponding axis. As shown in Fig. 1, one plane represents the plane perpendicular to the axis of the query for a single attribute through all points of last name = 'Smith', and the other plane through all points of first name ='John'. Together, they represent a line formed by the intersection of two search criteria, meaning "John Smith". So one can find "John Smith" by looking up the number of John Smith in the hash bucket that intersects this line. Nuno Diegues et al. [4] explore a performance model and propose a method for adaptively selecting the best solutions based on changing workloads. But it is also designed for the distributed implementation. However, in a centralized environment, GPU-accelerated implementation is imperative, since the multicore of CPU cannot meet the demand of high data parallelism and high memory bandwidth. As a promising solution, GPU can greatly speed up key-value storage (and query) operations in memory due to inherent hardware features.

HyperDex is a distributed system which can relieve the performance issue of hyperspace hashing. But in a centralized environment, GPU-accelerated implementation is imperative. In this paper, we aim to crack the nut of improving the performance of hyperspace hashing on GPU for the first time. First, we perform a comprehensive analysis on Hyperspace hash and GPUs, and identify several gaps between characteristics of Hyperspace hash and the features of GPUs. By using the traditional hyperspace hash structure (Fig. 2), it is difficult to maximize memory throughput on GPU as the number of queried attributes can not be previously determined. Two concurrently executed queries may need to be performed in different hash buckets or need to query different attributes. It will lead to branch divergence, which will decrease the query performance tremendously when processed in the same GPU warp. Moreover, updating indexable attribute values will cause data relocation, which will further increase the complexity of concurrency. All these characteristics of HyperSpace hash mismatch the features of GPU, which impedes the performance of Hyperspace hash on GPU.

Based on this observation, we propose a new hyperspace hash data structure (GHSH) to make hyperspace hashing better adapted to GPU. In GHSH, we use structure-of-arrays instead of array-of-structures data layouts, in which keys, second attributes, and values are stored separately. The novel data structure is more suitable for the memory hierarchy of GPU and has good cache locality, which can avoid access to unrelated second attributes. Furthermore, for batch queries, we devise a warp-level pre-combination data sharing strategy that uses query classifications to reduce branch divergence. To further improve the performance of GHSH, we explore two other tailored optimizations, i.e., atomic operations instead of locking and a new concurrency control strategy.

The experiments performed on an Nvidia RTX2080Ti (Turing) GPU suggest that GHSH has advantages in both flexibility and performance. Compared with the CPU version hyperspace hashing, GHSH outperforms it by 20-100 times. Compared to other GPU hashes that cannot conduct queries on non-key attributes, GHSH demonstrates comparable building and retrieval performance and achieves full concurrency. Our contributions are summarized as follows:

- We focus on the GPU-based hyperspace hashing technique for the first time and propose a novel hyperspace hash data structure that is well adaptive with the GPU memory hierarchy with the superior locality.
- We propose a warp pre-combination data sharing strategy to minimizes divergences, and we also propose a method of using atomic operations instead of locking and a temporary repeated read strategy to improve the performance of GHSH to achieve lock-free full concurrency.
- Based on the above design, we further describe how GHSH handles common operations in a batch update scenario, including bulk-build, search by key, search by secondary attribute, modify, insert and delete.

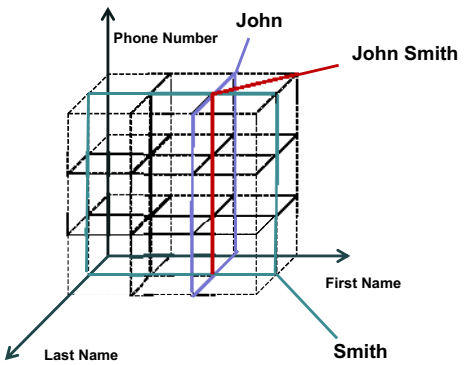


Fig. 1 Three-dimensional space hash

2 Background and Related Work

2.1 General-Purpose GPUs

Executing general-purpose programs on heterogeneous GPU/CPU architectures is implemented through various application programming interfaces (APIs), such as CUDA (NVIDIA) [5] and OpenCL [6]. GPUs are massively parallel processors with thousands of active threads. The threads in a warp are executed in a single instruction multiple thread manner, and thus any branch statements that cause threads to

run different instructions are serialized (branch divergence). A group of threads (multiple warps) is called a thread block and is scheduled to be run on different streaming processors (SMs) on the GPU. Maximizing achieved memory bandwidth requires accessing consecutive memory indices within a warp (coalesced access). NVIDIA GPUs support a set of warp-wide instructions (e.g., shuffles and ballots) so that all threads within a warp can communicate with each other.

2.2 HyperSpace Hasing

*Secondary Index*Several NoSQL databases have added support for secondary indexes. Mohiuddin et al. [7] perform a comparative study of secondary indexing techniques in NoSQL databases showing that the stand-alone indexes have higher maintenance cost in terms of time and space, and embedded indexes have slower query time. But hyperspace hashing devised in HyperDex [1] can better balance the maintenance cost and query efficiency as an embedded secondary index. Nuno Diegues et al. [4] explore a performance model and propose a method for adaptively selecting best solutions based on changing workloads. But it is also designed for the distributed implementation.

*HyperSpace Hasing in HyperDex*One of the main goals of HyperDex is to support efficient partial searches by secondary attributes, mainly by reducing substantially the number of servers involved in each query. The main idea is to use hyperspace hashing, in which the system can deterministically calculate the smallest set of servers that may contain data matching a given query. Consider that the objects to be stored have ς distinct attributes. A hyperspace in HyperDex is an Euclidean space with ς dimensions, such that each dimension i is associated with an attribute $A_i \in \{A_1, \ldots, A_\varsigma\}$. Hyperspace hashing maps an object in the hyperspace by applying a hashing function to the value of each attribute A_i of the object. In this way, we obtain a vector of ς coordinates that correspond to the point in the hyperspace where the object is located. But for this paper, we only study the hyperspatial hash index in the centralized form, and we treat disjoint regions in the multidimensional space as many corresponding hash buckets. Depending on the query criteria, the query involves different hash buckets.

A naive hyperspace construction, however, may suffer from a well-known problem with multi-attribute data known as "curse of dimensionality [8]." With each additional secondary attribute, the hyperspace increases in volume exponentially. If constructed in this fashion, each server would be responsible for a large volume of the resulting hyperspace, which would in turn force search operations to contact a large number of servers, counteracting the benefits of hyperspace hashing. HyperDex addresses this problem by partitioning the data into smaller, limited size subspaces of fewer dimensions. In Fig. 3 we show three possible subspaces with the corresponding regions (distributed to servers) and some points representing employees. Considering a query for employees in Beijing: using the subspace of Fig. 3a it is necessary to contact only 1 region, whereas in Fig. 3b it is necessary to contact 3. If the query also specifies an additional requirement of salary 5000, only one region is contacted in both cases. Furthermore, if we consider a three-dimensional subspace (see Fig. 3c), we need to specify three attributes in the query to have an efficient operation that contacts only one region. Note that, independently of the number of dimensions of a subspace, the strategy adopted in HyperDex is to divide each dimension of a subspace such that the total number of regions per subspace is close to a predefined value R. In our study, we have also realized the subspace mode named adaptive indexing in the centralized GPU environment. It can predict query tasks based on historical query records, periodically adjust the hyperspace hash index automatically, and always maintain a high query rate.

2.3 GPU Hash

With the popularity of parallel hardware, significant efforts have been made to improve the insert and query performance of hash. There are multiple GPU-based static hash tables. Alcantara et al. [9] propose Cuckoo hashing which has good performance in batch construction and retrieval stages. It is adopted in the implementation of the CUDA data-parallel primitives library (CUDPP) [10]. However, for a large load factor requirements, the batch construction is likely to fail. Garcia et al. [11] propose a Robin hood-based hashing, focusing on higher load factors and take advantage of the spatial locality of graphics applications, but the performance of this method is impacted. Khorasani et al. [12] propose Stadium Hashing (Stash), by extending the Cuckoo hashing for large hash tables. The key focus of Stash is to design an out-of-core hash table for the case that the data cannot be completely accommodated into a single GPU memory. In the research of GPU's fully concurrent and dynamically updateable hash table, Misra and Chaudhuri [13] evaluate the acceleration of several lock-free data structures transplanted from CPU to GPU. However, the implementation is not completely dynamic. We can see from the experiments that node resource arrays are pre-allocated for future insertions (i.e., it must be known at compiling time), and cannot be dynamically allocated and released at runtime. One main objective of our proposed GHSH is to conquer this problem. Recently, Ashkiani et al. [14] propose a fully concurrent dynamic lock-free chained hash table on GPU named Slab Hash.

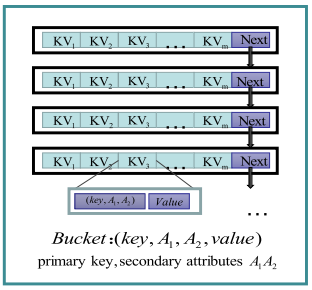


Fig. 2 Traditional data structure

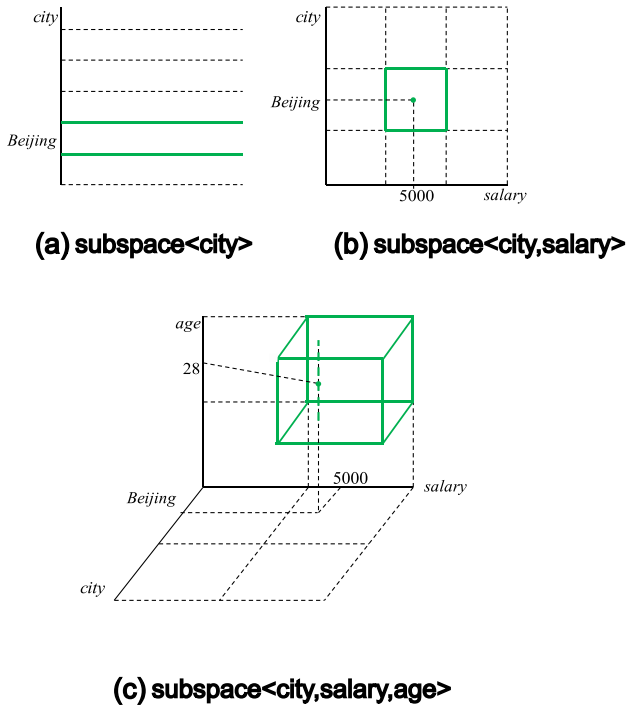


Fig. 3 Three different configurations and corresponding visualization of a search specifying values for all attributes indexed by the subspace

2.4 Other structures on GPU

Cederman et al. [15] perform experiments on various known lock-based and lock-free Queue implementations. They find that the parallel optimization of Queues on GPU is beneficial to performance improvement. Multi-core GPU technology can further improve data parallelism. Maksudul Alam et al [16] propose a novel parallel algorithms for fast Multi-GPU-Based generation of massive networks. Inspired by [13], Moscovici et al. [17] propose a GPU-friendly skip list (GFSL) based on fine-grained locks, mainly considering the preferred coalesced memory accesses of the GPU. In addition, Maksudul Alam et al. [18–20] propose a novel parallel algorithms for fast Multi-GPU-Based generation of massive networks. Brandon Tran et al. [21] present three GPU algorithm enhancement strategies (data structure reuse, metadata creation with various type alignment and a preallocated memory pool) for executing queries of bitmap indices compressed using Word Aligned Hybrid compression. Harish Doraiswamy et al. [22] propose a new model that represents spatial data as geometric objects and define an algebra consisting of GPU-friendly composable operators that operate over these objects. These structures and models can not directly applied to GPU hyperspace hash.

3 GHSH Structure

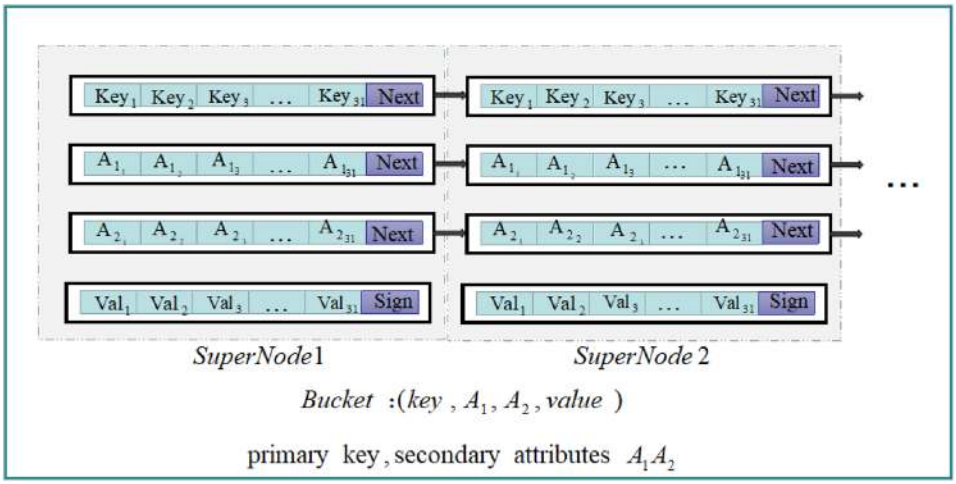
In this study, we represent the element N as $(key, A_1, A_2, \ldots, A_p, value)$, where key is the primary key, $A_1, A_2, \ldots, A_p$ are indexable secondary attributes with the number of p, and $value$ is a location ID or value. We build a hyperspace hash to assist the query processing of secondary attributes. Here, p secondary attributes require a hyperspace hash of $p + 1$ dimensions.

3.1 Data Structure

We adopt a linked list to handle hash collisions. Due to the variability of p, it is difficult for the accesses to be as coalesced as in traditional methods (see Fig. 2). In this paper, we propose a new data structure—super node. As shown in Fig. 4, a super node contains a key node, p attribute nodes, and a value node. Each node in a super node stores corresponding portions of multiple data for data alignment. We set pointers for each queryable attribute node and key node so that we can quickly traverse between them to find querying targets. For a query task, we search in the hash buckets where the target may be stored based on our hash function. Each thread traverses the corresponding attribute chain in a hash bucket that it is responsible for, and finds the corresponding value from the corresponding value node after finding the target. The peak memory bandwidth is achieved when threads within each SIMD unit (i.e., a warp in GPUs) access consecutive memory indices with a certain fixed alignment (e.g., on NVIDIA GPUs, each thread fetches a 32-bit word per memory access, i.e.,128 bytes per warp). To maximize memory throughput, the size of each node is set as 128 bytes. Thus a warp of 32 threads can access the entire contents of a node at once. We assume that each element and pointer take a memory space with a size of $(p + 2)x$ and y bytes. Therefore, the number of elements stored in each super node is $M = \lfloor \frac{128-y}{x} \rfloor$.

We use a universal hash function on each dimension, $h(k;a, b) = ((ak + b) \bmod q) \bmod B$, where a, b are random arbitrary integers and q is a random prime number. As a result, elements are expected to be distributed uniformly among B^{p+1}

Fig. 4 Data structure in GHSH



buckets with an *average super node count* of $\beta = \lceil \frac{n}{MB^{p+1}} \rceil$ super nodes per bucket, where n is the total number of elements in GHSH. When searching a secondary attribute, we perform $\beta + F (0 \leq F \leq \beta)$ memory accesses, where F is the number of fetches required to read the values in a bucket. Processing queries of keys (recall that keys are unique) can achieve slightly better performance, but have similar asymptotic behavior. In that case, we perform $\lceil \frac{n}{B^{p+1}} / \lfloor \frac{128-y}{(p+2)x} \rfloor \rceil$ memory accesses with the traditional data structure. It can be deduced from Eq. 1 that when $n\ B^{p+1} \cdot (p+1)$, our data structure can reduce the number of memory accesses. In our application scenarios, $n \gg B^{p+1} \cdot (p+1)$, so the proposed data structure is theoretically effective, in that it maximizes the throughput.

$$\zeta_{\text{optimized}}(x, y, p, B, n, F) = \lceil \frac{n/B^{p+1}}{\lfloor \frac{128-y}{x} \rfloor} \rceil + F < \lceil \frac{n/B^{p+1}}{\lfloor \frac{128-y}{(p+2)x} \rfloor} \rceil \quad (1)$$

For open addressing hash tables, the memory utilization is equal to the load factor, i.e., the number of stored elements divided by the table size. In order to be able to compare our memory usage with open-addressing hash tables that do not use any pointers, we define the memory utilization to be the amount of memory actually used to store the data over the total amount of used memory (including pointers and unused empty slots), which is shown in Eq. 2. Assume k_i denotes the number of super nodes for bucket i. The maximum memory utilization can reach $\frac{Mx}{(Mx+y)}$. Intuitively, this case happens when all nodes in GHSH are full. According to Eq. 2, we can calculate the memory utilization of GHSH by the number of buckets, and set various memory utilization in the experiments in Sect. 5.

$$\eta(x, y, B, p) = \frac{(p+2)x}{\lfloor \frac{128-y}{x} \rfloor (p+2)x + (p+2)y} \cdot \frac{n}{\sum_{i=1}^{B^{p+1}} k_i} \leq \frac{\lfloor \frac{128-y}{x} \rfloor (p+2)x}{\lfloor \frac{128-y}{x} \rfloor (p+2)x + (p+2)y} \quad (2)$$

3.2 Supported Operations in GHSH

Suppose our GHSH maintains a set of tuples represented by S. We allow the secondary attributes to be non-unique, but the primary key is unique. We support the following operations. More details will be introduced in Sect. 4.1.1.

- Insert $(key, A_1, A_2, \ldots, A_p, value) : S \leftarrow S \cup < key, A_1, A_2, \ldots, A_p, value >$,which represents inserting a tuple into GHSH.
- Search(key, $val(key)$): Returning $< key, A_1, A_2, \ldots, A_p, value > \in S$, or $\emptyset$ if not found.
- Search($A_i, val(A_i)$): Returning all found instances of A_i in the data structure ($\{<*, A_i = val(A_i), *>\} \in S$), or $\emptyset$ if not found.($1 \leq i \leq p$)
- Modify $(key, A_1, A_2, \ldots, A_p, value) : S \leftarrow (S - \{< key, *>\}) \cup \{< key, A_1, A_2, \ldots, A_p, value >\}$, which represents inserting a new tuple and deleting the old one.
- Delete(key) : $S \leftarrow S - \{< key, A_1, A_2, \ldots, A_p, value >\}$, which represents deleting the tuple $< key, A_1, A_2, \ldots, A_p, value >$.

4 Implementation Details

4.1 Optimizations

4.1.1 Warp Pre-combination Data Sharing Strategy

Assume that the batch size of queries is b and there are $p+1$ query types. We denote queries as $Q = \{key : val(key), A_1 : val(A1), \ldots, A_p : val(A_p)\}$, where $val(key)$ denotes the value of the key and $val(A_i)$ denotes the value of the secondary attribute A_i. GPU organizes threads in units of warps. Different query paths may incur warp divergence (see Fig. 5). A traditional solution is the warp-cooperative work sharing (WCWS) strategy [14], which forms a work queue of arbitrary requested operations from different threads within a warp. All threads within a warp cooperate to process these operations one at a time. However, a query operation of a hyperspace hash maps multiple query paths, which results in a sharp increase in the number of query tasks. The serialization of threads in a warp in WCWS severely hinders operation efficiency in hyperspace hashing. Considering the characteristics of our new hyperspace hash data structure, in this paper, we propose a new approach where threads in a warp read corresponding data of nodes to shared memory. All threads in a warp can compare in parallel whether the current node has its target. We call this strategy warp pre-combination data sharing (WPDS). WPDS is particularly suitable in following scenarios: (1) threads are assigned to independent and different tasks, which can avoid divergence within a warp; (2) each task requires an arbitrarily placed but vectorized memory access (accessing consecutive memory units); (3) it is possible to process each task in parallel within a warp using warp-wide communication (warp friendly). It is worth mentioning that WPDS is not suitable for situations where the branch divergence in a warp cannot be avoided through previous operations.

In our data structure context, query classifications are combined to reduce branch divergence. Take a secondary attribute query as an example. A query $(A_i, val(A_i))$ may exist in B^p buckets. Since the results are not unique, we need to traverse all nodes in these hash buckets. For query tasks of number b, classification is performed in two steps: *classify by BucketID* (CBB) and *pre-combining by query type* (PBQ). Each bucket i maintains a task queue TQ_i. In the CBB step, each task is parsed into p subtasks and added to the corresponding TQ_i (see Fig. 6). This can be performed in parallel on GPU. The variety of query types in a warp makes memory divergence (see Fig. 7a), which would greatly impede the GPU performance. To eliminate the divergence, in the PBQ step, threads in each TQ_i are grouped in a warp according to A_i (see Fig. 7b).

The total amount of time required for searching a batch of b elements in GHSH (n elements are stored in GHSH) is $T^b_{\text{Search}}(n,p) = T^b_{\text{CBB}} + T^b_{\text{PBQ}} + T^b_{\text{Tra}}$,where T^b_{CBB} is the time spent to classify a size b batch by *bucketID*, T^b_{PBQ} is the time spent to pre-combinate a size b batch by query type, and T^b_{Tra} is the time spent to traversal in buckets. The time complexity is $O(\frac{n/B^{p+1}}{\lfloor(128-y)/x\rfloor})$. Recall that the time complexity is $O(n/B^{p+1})$ without using WPDS strategy.

4.1.2 Global Memory with CUDA Atomic Operations

The GPU memory structure is divided into three levels: global memory that can be accessed by all threads in the device, smaller but faster shared memory per thread block, and local registers for each thread in the thread block. Although GHSH allows the entire thread of the warp to

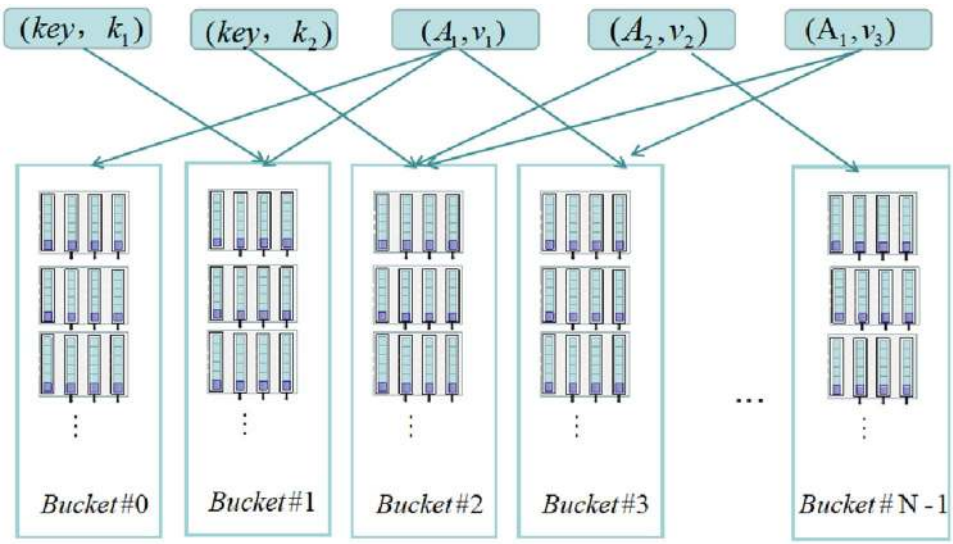


Fig. 5 Query divergence

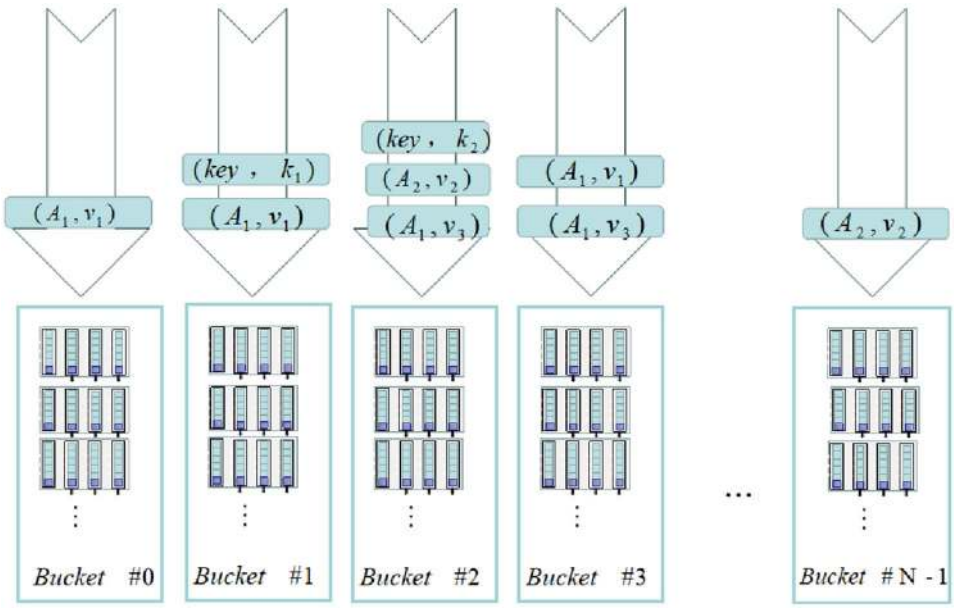


Fig. 6 Classify by BucketID

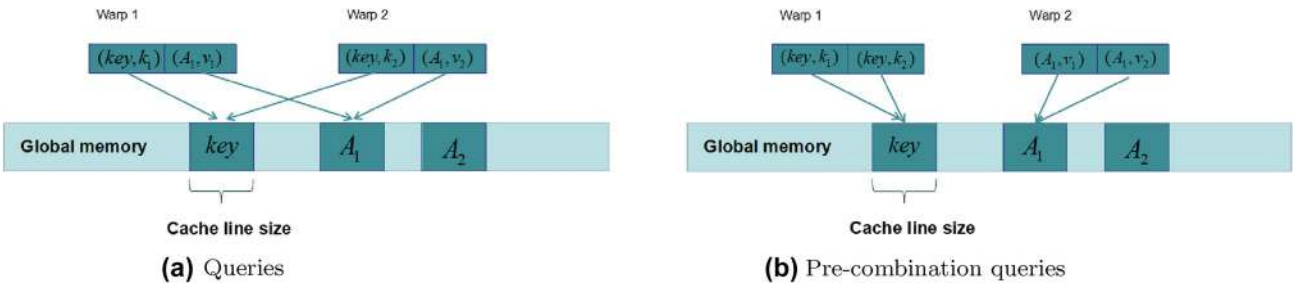


Fig. 7 An example of memory access pattern for queries

cooperatively handle the same operation task, the operation between different thread blocks is still independent and completely concurrent. The shared memory is small and partitioned, so threads in different blocks cannot access the shared memory of other blocks. The GPU's global memory capacity is large and can be accessed by all threads. Since millions of threads can execute GPU kernel functions simultaneously, but only a limited number of SMs exist, thread blocks need to be queued for SMs. Therefore, there is no way to synchronize all threads globally except when the kernel function ends. In order to achieve full concurrency between warps, GHSH uses global memory to ensure the sharing of all data states by each thread. GHSH controls concurrency optimistically. Common lock-free algorithms are generally based on atomic operations. For multiple atomic operations in GHSH, we set a lock tag on data items, and use atomic operations to change the lock tag to make the locking of data items atomic.

4.1.3 Temporary Repeated Read Strategy

The existing method employs reader-writer locks to allow concurrent read to hash table, which may cause conflicts to increase latency. In order to make GHSH free of structural locks, we design a "temporary repeated read" strategy. We split the bucket change operation and specifies a strict order. We first put new data into a new hash bucket, and then delete the data in the old bucket. This ensures all existing data will be read. Although it will cause a short-term duplication, it guarantees the valid values stored in the table can be read. If the query result for a key returns two values, either can be used. The "temporary repeated read" strategy guarantees the no-wait feature of reading operations, which can greatly improve the efficiency of reading operations in a concurrent situation. For read-intensive applications or applications with far more read operations than write operations. Improving read operation efficiency is very meaningful to improve overall operational efficiency.

4.2 Operation Details

We use WPDS for bulk query and deletion concurrency. But we still use WCWS [14] for full concurrency of all operations of different types. Because branch divergence here is caused not only by query path divergence but also by different instruction types. In WCWS, each thread has its assignment, that is, update (insertion, modification, or deletion) or query (by key or by secondary attributes). We design the node size as 128 B, so that when a warp accesses a node each thread has exactly 1/32 of the node's content. We use the term "lane" to denote the portion of a node that is read by the corresponding warp's thread, and we denote *lane 31* as *ADDRESS_LANE*. We assume that the key and the secondary attributes each take 4 bytes, and the value takes 4 bytes. For a node, we also need a 4-byte space to store a pointer. The remaining 124 bytes can be used to store a total number of 31 of keys or secondary attributes. Obviously, the spatial complexity of these operations is $O(n)$.

4.2.1 Insertion

Bulk-Build. The batch building operation constructs a hyperspace hash directly from the batch input of tuples. We first sort tuples by *bucketID* to ensure that data from a warp is most likely to be inserted into the same hash bucket. The first 31 threads in a warp are responsible for writing the corresponding part of the data, and the 32nd thread is responsible for putting the node into the linked list of the corresponding bucket. We assign a lock tag for each bucket. Only when a thread modifies the lock tag through atomic operations can it chain its data block into the bucket by header interpolation. The different tasks of the last thread will cause divisions, which is inevitable because the link operation must be executed after applying for memory. The first 31 threads can be parallelized, which is faster than the 32 thread tasks serialized in Slab hash.

Incremental Insertion. As shown in Alg. 1, any thread that has an inserting operation to perform will set *is_active* to true. Following WCWS introduced earlier, all threads read the corresponding part of the target super node, and search for the empty point in the key node of the super node. If found, the thread uses an atomicCAS operation to insert its key, secondary attributes and value address into the corresponding node of the super node (Line 6). If the insertion succeeds, the thread marks its operation as resolved. If fails, it means that some other warps have been inserted into that blank spot. Then the whole process should be restarted. If no empty point is found, all threads will obtain the address of the next super node from the ADDRESS_LANE. If the address is not empty, we should read a new super node and repeat the insertion process. Otherwise, a new super node should be allocated. Then, the source thread uses `atomicCAS` to update the pointer of the previous super node. If `atomicCAS` succeeds, the entire insertion process is repeated using the newly allocated super node. If not, it means that another warp has allocated and inserted a new super node. Then the super node allocated by this warp should be reassigned and the process should be restarted with the new super node.

4.2.2 Search and Deletion

We describe how queries are processed using hyperspace hashing. We define a search query Q as the set of attributes that the query accesses (and respective values). In the general case, to execute a query it is necessary to obtain the information of the buckets(regions) where the data is located according to the subspace. For instance, in the example of Fig. 1b, a search $Q =<$ city = Beijing, salary = 5000 $>$ results in contacting only one bucket(region), but in a query for $Q =<$ city = Beijing $>$ in the subspace $<$ city, salary $>$ all three regions are contacted. In order to obtain the best throughput possible, HyperDex always executes a query on the subspace $S_i \in S$ which yields the minimum number of regions. Note that HyperDex maintains a full copy of each object in each configured subspace.

*Search by Key.*Algorithm. 2 shows the pseudocode of searching in GHSH. We use WPDS to reduce query divergence with high concurrency. During the query process, we first parse the query tasks and distribute the results to the queue to be queried for the corresponding hash bucket (Line 1). Query tasks with the same query attributes are aggregated (Line 3), and organized into warps, and query paths are shared to avoid warp divergence. Each thread in a warp determines the part of the data it should read according to the *laneID* it carries. Although the reading positions are different, each thread needs to make a conditional judgment on the data in the node, and thus it uses `shuffle` instruction. The address read by the first thread is distributed to other threads (Lines 5-6). When GHSH performs a search operation, it reads the corresponding position of the linked list node. When checking whether there is data equal to the target key, the `ballots` and `ffs` instructions are used to make a parallel judgment on the data held by all threads in the warp (Line 7). If the target is found, we read the corresponding value into *myValue* and mark the corresponding task as resolved (Lines 8-12). Otherwise,each thread reads the pointer marked by ADDRESS_LANE and finds the next key node until the pointer is empty (Lines 14-19).

*Search by Secondary Attribute.*The secondary attribute query is similar to the key query, except that since the second attribute is not unique, a query may correspond to multiple values. Therefore, we need to traversal all the super nodes stored in a bucket. In our data structure, attributes are stored separately, and Each attribute node stores a pointer

to the next super node. Therefore, without accessing other attribute nodes, we can just query a linked list of the query attribute.

*Deletion.*Deletion is similar to search. If valid (matching found), we use *DELETE_DKEY* to overwrite the corresponding element. If not found, the next pointer is updated. If the end of the list is reached, the target does not exist and the operation terminates successfully. Otherwise, we load the key node of the next super node and restart the process. In particular, if the data being modified is found, there may be two values returned, and both are deleted.

4.2.3 Modification

In GHSH, the modification is divided into two types. One is to change the non-queryable attribute value, and the other is to change the secondary attribute value. The former is relatively simple, whose operation process is similar to a query, except that when the key value is found, the corresponding value is modified by using an atomic operation. The latter is more complicated. According to the updated value, the modified data may be or not be in the same hash bucket. If it involves a bucket change operation, we use the temporary repeated read strategy introduced earlier. Specifically, the data in the original bucket needs to be deleted, and the new value needs to be inserted into a new hash bucket. These two operations should be performed in an atomic fashion. Otherwise, data errors will occur. So we design a lock tag on the data item, namely *swap_lock*, and modify it through atomic operations to ensure that data is not modified by other tasks while it is being modified. As shown in Alg. 3, for a modification $(key, A_1, A_2, \ldots, A_p, value)$, we first search for *key* (lines 6), and if not found, it returns invalid task (line 23). If found without *swap_lock*, we mark *swap_lock* before inserting the new value with *swap_lock* (lines 5-8). Afterward, we delete the found data and erase the lock tag (lines 9-21). Otherwise, we need to wait until *key*'s lock tag is erased.

5 Adaptive Indexing

In the high-dimensional Euclidean space constructed by the hyperspace hash index, our query needs to find some related buckets in parallel, and the number of these related buckets is closely related to our hyperspace dimension. However, for the case with many searchable attributes, the hyperspace may be very large because its capacity increases exponentially with the number of dimensions. Even for large data center designs, covering a large space with a large number of hash buckets is not feasible. For example, a table with 9 secondary attributes may require B^9 (B is the number of hash buckets per dimension) Buckets. Due to the limited GPU computing resources, this shortcoming greatly affects the search efficiency. In view of the above problems, inspired by the data partition strategy in Hyperdex [1], this paper proposes an adaptive indexing strategy. We group the attributes to be queried to build a hyperspace hash. That is, the original large hyperspace is divided into several low-dimensional spaces for simultaneous maintenance. The attribute division is shown in Fig. 8a. But how to store the divided search space hash has become our new problem. One method is to store only a part of the attributes of each hyperspace hash, although this method can minimize the storage space of each hyperspace hash because the attributes are not stored across space, which will lead to expensive search costs because Constructing data requires collaboration across hyperspaces. Instead, another design choice is to store the full value address of each key-value in each hyperspatial hash index, which will lead to faster search speeds. Because we store the value address, the basic operation of the super hash is based on the in-place update of the cuda atomic operation, so it does not involve data consistency issues.

Basic hyperspace hashing does not distinguish the key of data from its minor parts. This leads to two important issues when dealing with actual key-value stores. First, a key lookup is equivalent to a single property search. A single attribute search in a multidimensional space requires querying multiple hash buckets. In this hypothetical scenario, the cost of a key operation would be strictly higher than that of a traditional key-value store. HyperDex provides efficient key-based operations that use one-dimensional subspaces for keys. Inspired by this, when we create a hyperspatial hash index in groups, we create a one-dimensional index of the keys separately, and this one-dimensional index will not change with the change of the data, because the keys are immutable.

When building in batches, considering the application scenario of the application, we can build a hyperspace hash index of the key and some auxiliary indexes, as shown in Fig. 8b. Considering the search locality of the application, our index can be adaptively adjusted based on historical queries. Eradicate historical query records, calculate the attribute sets with more queries, combine attributes with high query attributes, and reconstruct the remaining hyperspatial hash indexes based on the one-dimensional hash index of the key.

6 Evaluation

We use an Nvidia RTX2080Ti (Turing) GPU for all experiments on an 8-core server (Intel Xeon CPU Silver 4110 @ 2.1 GHz, 64 G host memory). It has 68 SMs and an 11 GB GDDR6 memory with a peak bandwidth of 616 GB/s. The size of the share memory on each SM is 64 KB. All results are harmonic averages of rate or throughput. Our

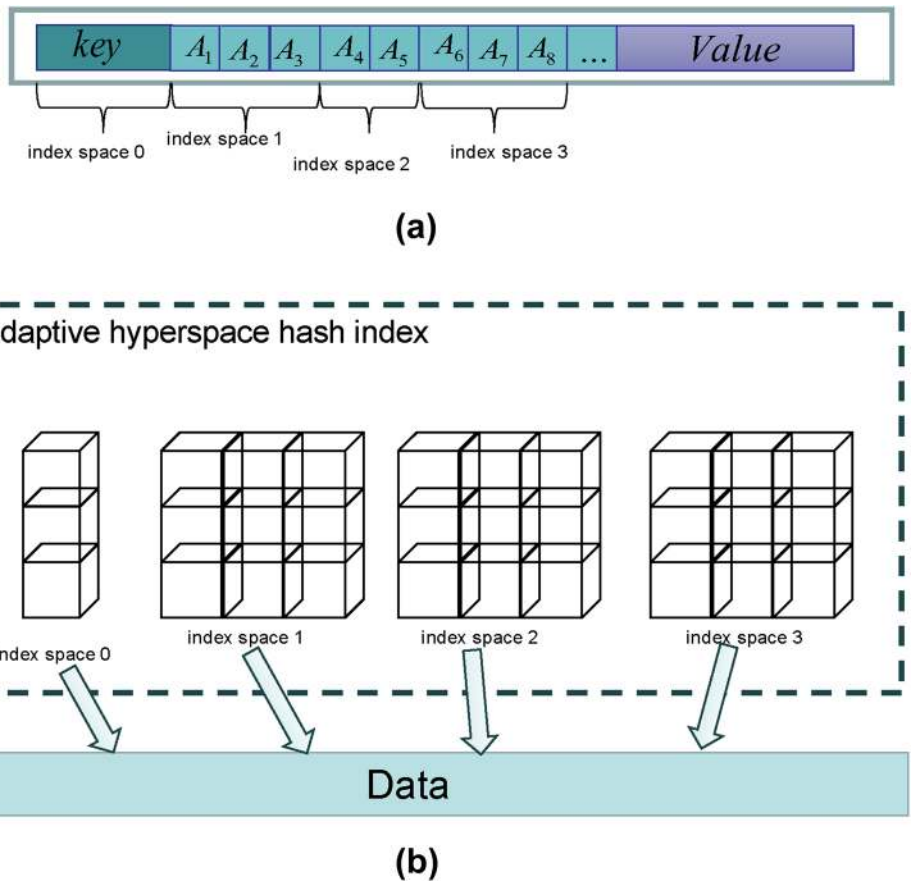


Fig. 8 Adaptive Indexing

implementation is compiled by CUDA 10 on Ubuntu 16.04 using O3 optimization option. Because the existing GPU-based index studies 4-byte keys and values for benchmarking and large batches for scalability tests [3, 14], we follow the convention and employ similar simulation patterns and parameter settings. We use a simple universal hash function such as $h(k;a,b) = ((ak + b)\ mod\ p)\ mod\ B$, where a, b are random arbitrary integers and p is a random prime number. As a result, on average, keys are distributed uniformly among all buckets. Our experiment additionally sets two secondary attributes. As a result, GHSH can achieve a maximum memory utilization of 97%. For static hash tables, there are two main operations: (1) building the data structure given a fixed load factor (i.e., memory utilization) and an input array of tuples, and (2) searching for an array of queries and returning values. By providing GHSH with the same set of inputs (each thread reads a key-value pair and dynamically inserts it into the data structure), we can compare GHSH with other static methods.

6.1 Memory Utilization

In order to study the impact of memory utilization, we compared GHSH with CUDPP (the most representative static hash) and Slab Hash (the most advanced dynamic hash). For static hash tables, such as CUDPP's cuckoo hash, there are two main operations: (1) batch construction stage, given a fixed load factor (which can be simply expressed in terms of pre-designed memory usage) and one An input array of key-value pairs is used to construct the entire data structure by batch insertion operations. If the insertion failure occurs during the construction phase, it needs to be reconstructed from scratch. (2) In the retrieval phase, after the end of the batch construction phase, the key array is used as the input, and the batch search operation is performed in the data structure, and the corresponding value returned is stored in the output array. By providing GHSH with the same set of inputs (each thread reads a key-value pair and dynamically inserts it into the data structure), we can build a hash table. Similarly, after building the hash table, each thread can read a query from the input array, search it dynamically in GHSH, and then store the search results in the output array. By doing this, we can compare GHSH with other static methods. Slab hash does the same.

For CUDPP, we can easily obtain a predetermined memory utilization, which is equal to the load factor, i.e., the number of stored elements divided by the table size. In contrast, we define the memory utilization of dynamic hash as the total amount of memory actually used to store data divided by the total amount of memory used. For Slab

hash [14], If each element and pointer take x and y bytes of memory respectively, then each slab requires $Mx + y$ bytes. As a result, the memory utilization of the Slab hash can be calculated according to Eq. 3, where k_i denotes the number of slabs for bucket i. Also, we give the definition and calculation of GHSH memory utilization in the previous Sect. 3.1.

$$\eta\_\text{Slab}(x, y, B, M) = \frac{x}{Mx + y} \cdot \frac{n}{\sum_{i=0}^{B-1} k_i} \leq \frac{Mx}{Mx + y} \tag{3}$$

Figures 9 and 10 respectively shows the building rate and query rate of several hash methods for various memory utilization. $n = 2^{22}$ elements are stored in the table. At about 60% memory utilization there is a sudden drop in performance for both bulk building and searching operations. Our experimental scenarios below using uniform distributions as input data set with 60% memory utilization based on this result and existing work [14]. In the concurrent benchmark, we set the initial memory utilization to 50% because of the insertions in concurrent workload.

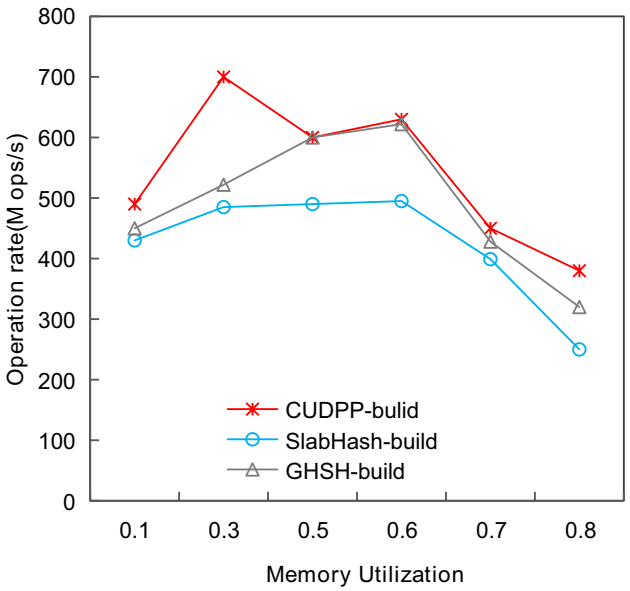


Fig. 9 Build rate vars memory utilization

6.2 Baseline: CPU Hyperspace Hash

We compare the speed of GHSH with its counterpart on CPU. Since there is no existing open-sourced concurrent CPU hyperspace hashing in the centralized environment , we implemented one based on openMP [23]. Experimental results show that the building performs best with 8 threads and the searching for secondary attributes performs best with 12 threads on CPU. Thus, we use these settings as benchmarks to compare with GHSH. To evaluate bulk building and searching operations for HyperSpace hash on CPU and GPU, we evaluate multiple data structure sizes incrementally. We choose an initial number of buckets so that its final memory utilization is 60%. The means of all operation rates for a given batch size of b is reported in Table 1. The mean of bulk building rates on GPU is 20.24 M elements/s, which is 20x of CPU. The mean of searching rates on GPU is 28.17 M elements/s, which is 100x of CPU.

Table 1 Rates(M operation/s)for different bath-sized put and get on GPU and CPU

Bath size	GPU (put)	CPU (put)	GPU (get)	CPU (get)
2^{10}	6.81	1.02	9.72	0.30
2^{11}	14	1.23	17	0.27
2^{12}	26	1.23	40	0.23
2^{13}	47	1.17	69	0.25
2^{14}	86	1.05	146	0.27
2^{15}	151	0.9	211	0.28
Mean	20.24	1.09	28.17	0.26

6.3 Impact of Different Design Choices

We evaluated the basic query operation with WCWS, CBB, and WPDS (CBB+PBQ). The results are shown in Fig. 15. We find that search efficiency can be greatly enhanced with CBB, but CBB is more time consuming as a preprocessing. Only with PBQ can a better optimization

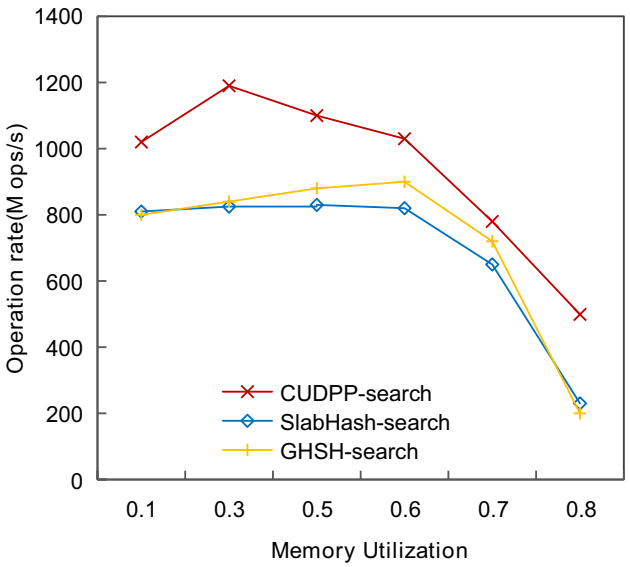


Fig. 10 Search rate vars memory utilization

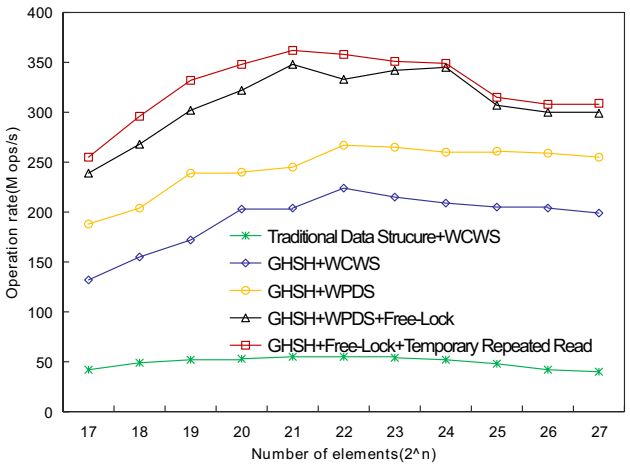


Fig. 11 Modification rate

effect be achieved, which accelerates 1.25x on average. It is worth mentioning that branch divergence in a warp cannot be avoided, if directly perform PBQ without CBB and hence threads in a warp cannot be parallelized.

Figure 11 shows the effectiveness of our novel data structure and three techniques. Assume that all modifications need bucket changing. We find that GHSH data structure can achieve performance improvement by 3.92X. Based on our data structure, after replacing the locking with global memory and atomic operation, the bucket changing rate of GHSH increases significantly, by 27% on average. This is because atomic operations greatly reduce the locking cost. Adding the temporary repeat read strategy to the lock-free version, the two techniques can improve the modification rate by 34%. However, with the increase in the number of elements, the acceleration effect is not obvious due to the limitation of the SM's number.

6.4 Baseline: Operations of GPU Hash for Keys

We compared GHSH with CUDPP (the most representative static hash) and Slab Hash (the most advanced dynamic hash). This experimental benchmark takes throughput (total number of operations/execution time) as an index to measure the performance of the data structure. The fixed memory utilization is 0.65. The hash function of each data structure also remains the same. The total number of operations is taken as the abscissa. The number of threads in the GPU data structure is equal to the total number of operations. After determining the number of threads in the GPU data structure, you need to determine the number of threads per thread block (number of thread blocks = total number of threads/number of threads per thread block)

6.4.1 Bulk-Build

For many data structures, the performance cost of supporting incremental mutable operations is very high: static data structures often provide better batch builds and query rates than similar data structures that support incremental mutable operations. However, we will see that the performance cost of supporting these additional operations in GHSH is modest. Figure 12 shows the building rate (M elements/s) versus the total number of elements (n) in table. We can see that when the table size is very small, CUDPP's building performance is particularly high, since most atomic operations can be done at cache level. Static data structures often sustain considerably better bulk-building and querying rates when compared to structures that additionally support incremental mutable operations. However, the cost of these additional operations in GHSH is modest. Slab hash and GHSH will make GPU resources reach $2^{20} \leq n \leq 2^{24}$. The build rate of GHSH is up to 1.32x that as those on Slab hash, since the basic nodes of GHSH store almost twice the data of Slab hash. Besides, data allocation in GHSH can be performed in parallel, which improves data parallelism.

6.4.2 Search Query

For key search queries, we generate two sets of random queries: (1) all queries exist in GHSH; (2) none of the queries exist. The two scenarios are important as they represent the best and worst case. The harmonic averages are 838 M/s and 732 M/s for search-all and search-null (see Fig. 13). The speedups of CUDPP over the GHSH are 1.27x, 1.16x, and 0.86x for bulk building, search-all, and search-none, respectively. The speedups of Slab hash over the GHSH are 0.84x, 1.01x, and 1.02x for bulk building, search-all, and search-none, respectively. GHSH's key query speed is slightly lower than Slab hash, which is the cost of supporting non-key queries. It is necessary for secondary attribute

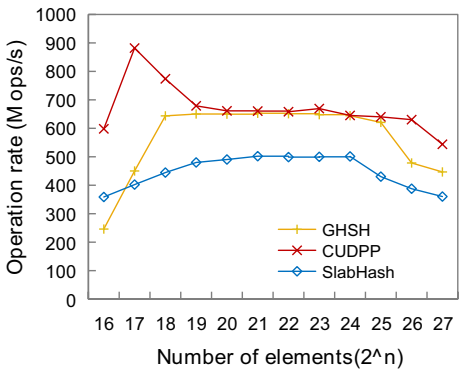


Fig. 12 Build rate

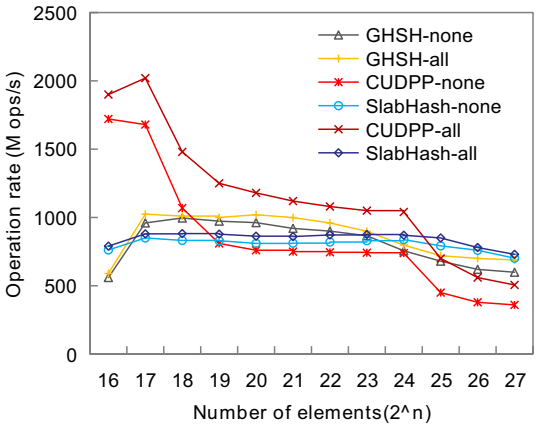


Fig. 13 Search rate

queries to traverse the complete linked list, which is equivalent to the worst case of key queries.

6.4.3 Incremental Insertion

Suppose we periodically add a new batch of elements to a hash table. For CUDPP, this means building from scratch every time. For the Slab hash and GHSH, this means dynamically inserting new elements into the same data structure. Figure 14 shows both methods in inserting new batches of different sizes (32 k, 64 k, and 128 k) until there are 2 million elements stored in the hash table. For CUDPP, we use a fixed 60% load factor. For the Slab hash and GHSH, we choose the initial number of buckets so that its final memory utilization (after inserting all batches) is 60%. As expected, the GHSH significantly outperforms cuckoo hashing by reaching the final speedup of 18.3x, 27.3x, and 32.5x for batches of size 128k, 64k, and 32k. As the number of inserted batches increases (as with smaller batches), the performance gap

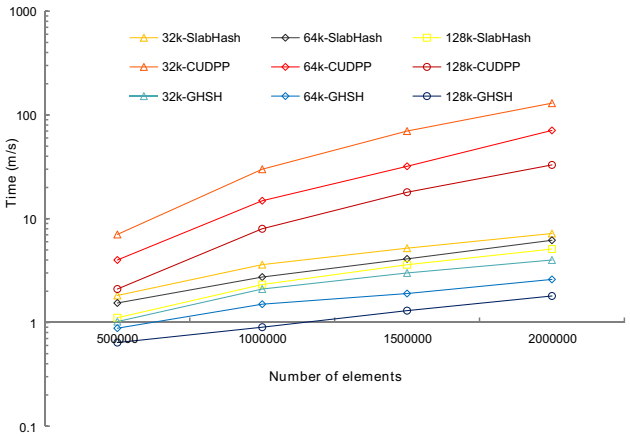


Fig. 14 Incremental insertion

increases. In contrast, the performance improvement effect of GHSH for Slab Hash is not as obvious as for CUDPP. For batch processing of 128k, 64k, and 32k, the final speedup of GHSH is 2.8x, 2.4x, and 1.8x, respectively, which is better than the Slab hash.

6.5 Concurrency Performance of GHSH

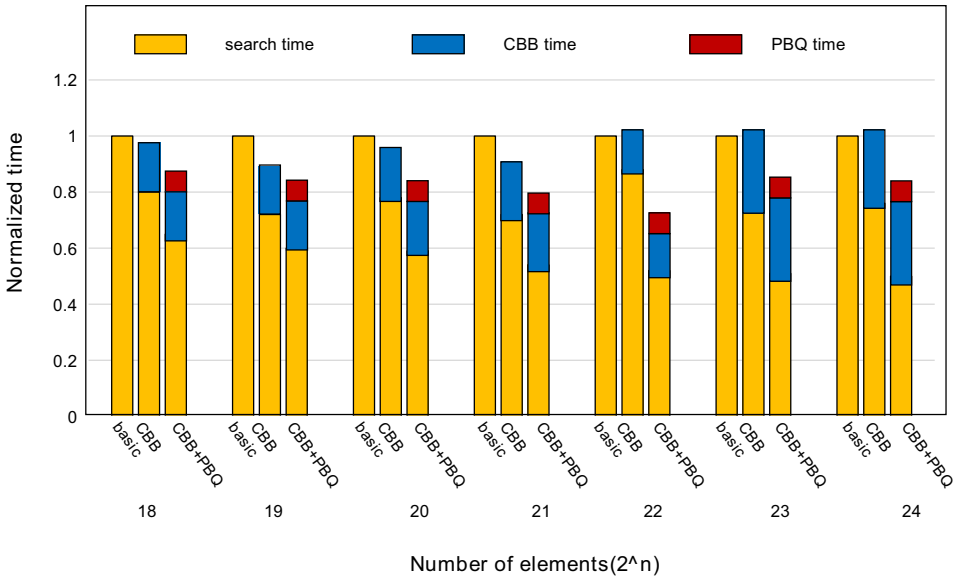
Benchmark setup A significant feature of GHSH is that it can perform true concurrent query and update operations without dividing different operations into different calculation stages. To evaluate the concurrency characteristics, we designed the following benchmarks. Suppose we build a hash table with an initial number of elements. We then proceed to perform operations in one of the following four categories: (a) inserting a new element, (b) deleting a previously inserted element, (c) searching for an existing element by secondary attribute, (d) modifying a element to a new bucket. We define an operation distribution $\Gamma = (a, b, c, d)$, such that every item is nonnegative and $a + b + c + d = 1$. Given any Γ, we can construct a random workload where, for instance, a denotes the fraction of new insertions compared to all other operations. Operations are run in parallel and randomly assigned to each thread (one operation per thread) such that all four operations may occur within a single warp. We consider three scenarios as shown in Table 2. In order to evaluate the value range of keys and secondary attributes, four different integer ranges, [0,100], [1,1000], [0,10000], and [0,100000], are designed for each operation combination. The total number of operations is fixed at 100,000. The operation sequence for each test is pre-generated based on the mix ratio and total number, and the operation keys are randomly generated from the range of keys being evaluated. The number of threads on the GPU is determined based on the total number of operations per test.

*Semantics*We support concurrent operations, which guarantees that the results of the batch operation include all pre-existing keys in the hash table, as long as they are not updated in the batch. However, the result of the operation on the updated keys in the batch will depend on the hardware scheduling of the block and the switch between warps. For example, a batch process might include inserts, deletes, and queries on keys already stored in the data structure. All three operations will complete, but the order in which they complete is undefined. Many applications may choose to

Table 2 Three scenarios of concurrent benchmarks

Name	Workload	Operation distribution
A	100%updates,0%searches	$\Gamma_0 = (0.4, 0.4, 0, 0.2)$
B	50%updates,50%searches	$\Gamma_1 = (0.2, 0.2, 0.5, 0.1)$
C	30%updates,70%searches	$\Gamma_2 = (0.1, 0.1, 0.7, 0.1)$

Fig. 15 Impact of WPDS



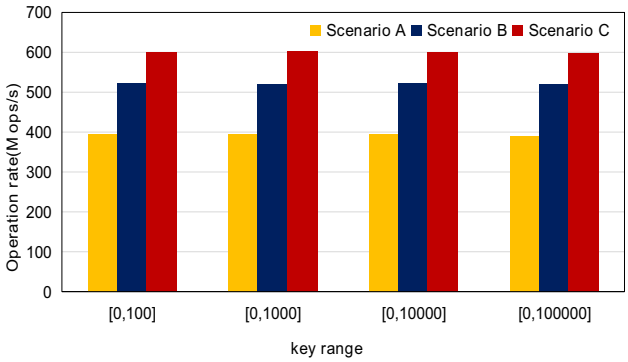


Fig. 16 Concurrency Performance of GHSH

solve this problem in a phased operation, where changes to the data structure (insert, delete) and the query of the data structure belong to different batches. However, strict serial semantics are not compatible with the implementation of hyperspace hashing.

Result Fig. 16 shows performance of GHSH for three scenarios and different key ranges. Since updates are computationally more expensive than searches, given a fixed memory utilization, performance becomes better with fewer updates ($\Gamma_0 < \Gamma_1 < \Gamma_2$). From the experimental results, we can see that the key range does not have much impact on performance. Comparing against bulk benchmark in Fig. 16, it is clear that GHSH performs slightly worse in our concurrent benchmark (e.g., Γ_0 in Figs. 12 and 16). There are two main reasons: (1) We assign multiple operations per thread and hide potential memory-related latencies, as it is assumed that in static situations all operations are available, and (2) we run four different procedures (one for each operation type) in concurrent benchmarks, but the bulk benchmark runs just one. As discussed in Sect. 2, the Slab hash is currently the most representative fully concurrent dynamic lock-free chained hash table on GPU. However, the slab hash does not support queries for secondary attributes, and the update operations involved are all updated in-situ, without data relocation to maintain the index. Therefore, the Slab hash is not suitable for the concurrency scenarios we design.

7 Conclusion

Through a comprehensive analysis of the characteristics of hyperspace hashing and the features of GPU, we identify some gaps between hyperspace hashing and GPU, such as the gap in memory access requirement, memory divergence, and query divergence. Based on the analysis, we propose a novel hyperspace hash data structure, where the query attributes are stored separately. Hence, GHSH can make full use of the GPU memory hierarchy to reduce the number of high-latency memory accesses via cache access on GPU. We also propose three optimizations in GHSH to alleviate the different divergences on GPU to improve resource utilization: warp pre-combination data sharing strategy, a method of using atomic operations instead of locking and temporary repeated read strategy. Experimental results suggest that our dynamic hyperspace hash table for GPU can gain advantages in both flexibility and performance.

Acknowledgements This work is supported by the National Key R&D Program of China (2018YFB1003400), the National Natural Science Foundation of China (62072083, 61872071), the Fundamental Research Funds for the Central Universities (N180716010,N2116008) and Liao Ning Revitalization Talents Program (XLYC1807158). The preliminary version of this article has been published in APWeb-WAIM 2020 [24].

References

1. Escriva R, Wong B, Gün Sirer E (2012) Hyperdex: a distributed, searchable key-value store. In: PrACM SIGCOMM. ACM, pp 25–36
2. D'silva JV (2017) Roger Ruiz-Carrillo, and Cong Yu. Two rings to rule them all. In: DOLAP, Secondary indexing techniques for key-value stores
3. Pedro H, Matheus N, de Almeida Eduardo C (2018) Cracking kd-tree: the first multidimensional adaptive indexing (position paper). In: EDDY
4. Diegues N, Orazov M, Paiva J, Rodrigues L, Romano P (2014) Optimizing hyperspace hashing via analytical modelling and adaptation. ACM SIGAPP Appl Comput Rev 14(2):23–35
5. Guide Design (2013) Cuda c programming guide. In: NVIDIA
6. Munshi A, Gaster B, Mattson TG, Ginsburg D (2011) OpenCL programming guide. Pearson Education, New York
7. Abdul QM, Shiwen C (2018) A comparative study of secondary indexing techniques in lsm-based nosql database. In: SIGMOD. pp 551–566
8. Kneale Samuel G (1958) Dynamic programming: by richard bellman. 342 pages, 6 ×[formula omitted] in Princeton. Princeton university press, 1957. price, \$6.75. J Franklin Inst 265(2):157–158
9. Alcantara Dan A, Andrei S, Fatemeh A, Shubhabrata S, Michael M, Owens John D (2009) Amenta Nina Real-time parallel hashing on the gpu. ACM Trans Graph (TOG) 28(5):154
10. Harris M, Owens J, Sengupta S, Zhang Y, Davidson A (2007) Cuda data parallel primitives library. Cudpp, New York
11. Ismael G, Sylvain L, Samuel H, Anass L (2011) Coherent parallel hashing. In: TOG, vol. 30. ACM, pp 161
12. Farzad K, Mehmet BE, Rajiv G (2015) Stadium hashing: scalable and flexible hashing on gpus. In: PACT. IEEE, pp 63–74
13. Prabhakar M, Mainak C (2012) Performance evaluation of concurrent lock-free data structures on gpus. In: ICPADS. IEEE, pp 53–60
14. Saman A, Martin F-C, John OD (2018) A dynamic hash table for the gpu. In: IPDPS. IEEE, pp 419–429
15. Daniel C, Bapi C, Philippas T (2012) Understanding the performance of concurrent data structures on graphics processors. In: Euro-Par. Springer, pp 883–894
16. Maksudul A, Kalyan PS, Peter S (2019) Novel parallel algorithms for fast multi-gpu-based generation of massive scale-free networks. Data Sci Eng 4(1):61–75
17. Moscovici N, Cohen N, Petrank E (2017) Poster: a gpu-friendly skiplist algorithm. PPOPP 52(8):449–450
18. Pawan H, Narayanan PJ (2007) Accelerating large graph algorithms on the gpu using cuda. In: HPCS. Springer, pp 197–208
19. Zhong J, He B (2013) Medusa: simplified graph processing on gpus. TPDS 25(6):1543–1552
20. Merrill D, Garland M, Grimshaw A (2015) High-performance and scalable gpu graph traversal. TOPC 1(2):14
21. Brandon T, Brennan S, Jason S, Joseph MM, David C (2020) Increasing the efficiency of gpu bitmap index query processing. In: International conference on database systems for advanced applications. Springer, pp 339–355
22. Harish D, Juliana F (2020) A gpu-friendly geometric data model and algebra for spatial queries. In: Proceedings of the 2020 ACM SIGMOD international conference on management of data. pp 1875–1885
23. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J (2001) Parallel programming in OpenMP. Morgan kaufmann, Ramesh Menon
24. Ren Z, Gu Y, Li C, Li F, Yu G (2020) Ghsh: dynamic hyperspace hashing on gpu. In: Wang X, Zhang R, Lee Y-K, Sun L, Moon Y-S (eds) Web and big data. Springer, Cham, pp 409–424