International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(2): 195–216, doi:10.21655/ijsi.1673-7288.00249

International Journal of Software and Informatics

Research Article

Hybrid Access Cache Indexing Framework Adapted to GPU

Hongjun Zhang (张鸿骏)[1,2], Yanjun Wu (武延军)[1], Heng Zhang (张珩)[1], Libo Zhang (张立波)[1]

[1] (Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2] (University of Chinese Academy of Sciences, Beijing 100049, China)

Corresponding author: Hongjun Zhang, hongjun@iscas.ac.cn

Abstract Hash tables, as a type of data indexing structure that provides efficient data access based on key values, are widely used in various computer applications, especially in system software, databases, and high-performance computing field that requires extremely high performance. In network, cloud computing and IoT services, hash tables have become the core system components of cache systems. However, with the large-scale increase in the amount of large-scale data, performance bottlenecks have gradually emerged in systems designed with a multi-core CPU as the core of the hash table structure. There is an urgent need to further improve the high performance and scalability of the hash tables. With the increasing popularity of general-purpose Graphic Processing Units (GPUs) and the substantial improvement of hardware computing capabilities and concurrency performance, various types of system software tasks with parallel computing as the core have been optimized on the GPU and have achieved considerable performance improvements. Due to the sparseness and randomness, using the existing parallel structure of the hash tables directly on the GPUs will inevitably bring high-frequency memory access and frequent bus data transmission, which affects the performance of the hash tables on the GPUs. This study focuses on the analysis of memory access, hit ratio, and index overhead of hash table indexes in the cache system. The hybrid access cache indexing framework CCHT (Cache Cuckoo Hash Table) adapted to GPU is proposed and provided. The cache strategy suitable to different requirements of hit ratios and index overheads allows concurrent execution of write and query operations, maximizing the use of the computing performance and concurrency characteristics of GPU hardware, reducing memory access and bus transferring overhead. Through GPU hardware implementation and experimental verification, CCHT is shown to have better performance than other cache indexing hash tables while ensuring cache hit ratios.

Keywords system software; cache index; hash table; GPU

Citation Zhang HJ, Wu YJ, Zhang H, Zhang LB. Hybrid access cache indexing framework adapted to GPU, *International Journal of Software and Informatics*, 2021, 11(2): 195–216. http://www.ijsi.org/1673-7288/00249.htm

---

This is the English version of the Chinese article "一种适应 GPU 的混合访问缓存索引框架. 软件学报, 2020, 31(10): 3038–3055. doi: 10.13328/j.cnki.jos.006069".

Funding items: Strategic Priority Research Program of the Chinese Academy of Sciences (Y8XD373105); Key Research Program of Frontier Sciences, Chinese Academy of Sciences (ZDBS-LY-JSC038)

Received 2020-02-10; Revised 2020-04-04; Accepted 2020-05-09; IJSI published online 2021-06-22

Hash tables are an important and common data indexing structure used in system software and high-performance data applications. By mapping the key value to a content location in the table to access the corresponding data record, we can greatly accelerate the speed of the data query. In the context of in-memory relational databases[1–3] and key-value databases[4, 5], the hash table acts as a core system component, provides the key data indexing structure, and indexes the data to the corresponding location through the mapping function. Specifically, as an important system component of the network, cloud computing, and IoT services[6], in-memory cache systems based on key values such as memcached[5], Ramcloud[7] and MemC3[8] all adopt hash tables to provide high-performance in-memory data indexing services. In-memory cache systems improve data access performance by loading data into memory from a storage medium such as a disk that is relatively slow to access. Since memory usually cannot hold all the data, when the memory space becomes full, the cache replacement system deletes part of the cached data by means of the cache replacement algorithm to make room for new data.

In a recent work on hash tables, the Cuckoo Hash Table (CHT)[9] has been proposed; it is based on the cuckoo algorithm and it maps the original single data into multiple buckets. CHT replaces the mapping and the linked list operations of the traditional hash tables and has the characteristics of fast access and saving space, which has been widely applied and studied in the hash table management[10–12] and data storage systems[4, 8, 13–16].

Typical CHT does not support concurrent data insert and data query operations, which maps data into two different buckets using two different hashing algorithms. Four slots in each bucket are allowed to store data and each data corresponds to a set of key-value pairs. During the insert operation, we should first determine whether there is a free slot in the mapping bucket, and if any, the data will be inserted. If not, the free slot is obtained by hashing and replacing the data stored in the existing unit, and then the data is inserted. Furthermore, if the data in the slot needs to be replaced when the key-value pair data is inserted, CHT will randomly select a Key Value (KV) to be replaced to replace KV′, calculate another optional bucket corresponding to KV′ through the hashing algorithm, and put KV into the slot where the original KV′ was located. If another optional bucket has a free slot, KV′ is put into this free slot. If the bucket does not have a free slot, the data is randomly selected from a slot in the bucket and replaced in the same way until a free slot is found. The load rate is the ratio of the number of stored data to the number of slots on the hash table, which is used to measure the current load status of the hash table. When the load rate is high, CHT frequently accesses memory due to the small number of remaining free slots and the frequent data replacements during insert operations. At the same time, a large number of temporary caches occur in the CPU, reducing the performance of the hash table.

In key-value in-memory cache systems, the main workload characteristic of such systems is that read operations are more frequent than write operations[17, 18]. When the request of a read operation results in a cache miss, the system writes the data requested by the read operation into the memory cache[19]. As the number of missed read requests increases, more replacement operations are introduced to the workload of key-value in-memory cache systems due to the frequent writing of the data missed by the cache. When the load rate of the hash table is high, CHT frequently occurs in the key-value in-memory cache system, which leads to many memory accesses and reduces the overall performance of the cache system. Therefore, the hash indexing method of key-value in-memory cache systems should reduce the memory access of write operations, ensure the hit ratio of the system, and reduce the number of write operations. With the large-scale increase in memory cache data, performance bottlenecks have gradually emerged in indexing systems with parallel hash table structures in a multi-core CPU. There is an urgent need to further improve the high performance and scalability of the hash table.

The development and change of hardware promote the evolution of system software[20]. With the improvement of computing power and concurrency performance of GPU hardware, more system tasks are run on the GPU. CHT-related work in key-value storage cache systems and relational databases has taken advantage of the performance improvements of hardware such as GPUs[10, 21–24]. However, the existing parallel optimization methods of CHT on multi-core CPU and GPU hardware do not operate efficiently when acquiring data, and the performance of the computing system is still limited by the memory bandwidth[25–28]. Firstly, due to the sparsity and randomness of the hash table, after the accessed data is stored in the chip cache, it may be replaced and expelled from the chip cache by other accessed data before the next access, which will result in inefficient utilization of the chip cache and a large amount of memory access to obtain data, thus affecting the system performance. Xie *et al.*[29, 30] proposed a cache bypassing method for GPUs to reduce the inefficient utilization of chip cache. Secondly, when GPU hardware is adopted as the computing core unit, frequent data transfer between CPU and GPU memory introduces more system interrupts, process switches, driver calls, and other additional overhead. Therefore, reducing the number of memory access times of the hash table and the frequent transfer of data between CPU and GPU memory is still an important way to optimize the system performance of the hash table.

In this paper, we propose the hybrid access cache indexing framework CCHT adapted to GPUs. This indexing framework can be applied to a cache replacement system for index management. CCHT achieves few memory access times through a multi-level cache index data structure and provides cache insertion and query algorithms that support concurrent execution of read and write operations. The multi-level index data structure not only provides the index position information and the priority queue information to be evicted from all caches, but also provides the priority queue to be evicted from the bucket of the hash table. The cache inserting and querying algorithms which support concurrent execution of read and write operations presents the hash table schemes, including the evicting algorithm when the index storage space in the hash table bucket and the global storage space are full and the updating algorithm of the evicted queue in the bucket and the global evicted queue when inserting and querying data. Based on the hit ratio and the index space overhead of the in-memory cache system, a double LRU CCHT with a relatively higher hit ratio and a coarse LRU CCHT with relatively lower memory space overhead are proposed, respectively. Double LRU CCHT can be used independently in global eviction and bucket eviction through two cache eviction priority queues to achieve a high cache hit ratio. The coarse LRU CCHT method only uses one cache eviction priority queue in both global and bucket eviction processes, which ensures the cache hit ratio and further reduces the index space overhead of the priority queues. In this paper, CCHT is implemented in the heterogeneous environment of GPUCPU. To reduce the occupation of memory bandwidth and the access times of GPU, we propose a multi-level index data structure based on the out-of-core computing system. Through this structure, only key data can be transferred between CPU and GPU memory in the request processing of CCHT, which avoids the GPU memory space and bandwidth resources to be occupated by value data. Experimental results show that when the ratio of cache space size to hash table size is 80%, the average access times of insertion and global load are reduced by 30.39% and 32.91% respectively. When the ratio of cache space size to hash table size is 90%, they are decreased by 94.63% and 97.29% respectively. This indicates that CCHT can provide little memory access even in the case of the high load rates of the hash table, and guarantee the performance of the cache replacement system. Experiments on the heterogeneous CPUGPU environment show that compared with other data indexing methods, CCHT improves the throughput performance of the system, which verifies the effectiveness of the multi-level index data structure and its implementation method.

1 Related Work

Cuckoo hash is an open addressing approach with an efficient space utilization[9], which specifies multiple candidate hash bucket positions for each object and allows the stored objects to be moved to other candidate bucket positions. SILT[13] achieved a high space occupation through two hashing methods, but it cannot be applied to memory cache with large storage space due to the limited hash table size. MemC3[8] not only ensured the high space utilization but also eliminated the limitation of the hash table size through labeling and an optimized locking mechanism. Hopscotch hashing[14] and Cache-oblivious hashing[15] improved the throughput performance by increasing the index pointer space to ensure access concurrency. Li *et al*.[31] achieved the high concurrency of access through Hardware Transaction Memory (HTM) and a coarse locking mechanism. FlashStore[15] mapped an object to 16 bucket positions with Cuckoo hash, and increased the probability of looking for a free slot when inserting the object. Kirsch *et al*.[16] used additional hidden space to reduce the overhead of looking for free space.

In the research process of CHT implementation in the CPUGPU heterogeneous environment, Horton tables[10] adopted the additional space and additional hash mapping function to replace the original replacement method in the case there are no free slots. Stadium hashing[11] used the combination of CPU and GPU to improve the performance; i.e., it stores the keys in the GPU and the values in the CPU memory. The operations on CPU and GPU are distinguished by adding an auxiliary data structure (ticket board), which allows concurrent execution of read and write operations on the same hash table. Barber[12] implemented two compressed hash tables using a similar bit-mapping structure. Xie *et al*.[29, 30] statically or dynamically specified the temporary or infrequently used data to skip a part of chip caches by using program compilation and instrumentation on the GPU, to reduce the frequent replacement of chip caches.

In practical applications, the hash table is widely used in various types of system software and data processing programs. Many memory key-value storage systems speed up the query of key data through a hash table applied to the GPU[23, 32, 33]. Early implementations of the hash table on the GPU were used for database management, graphics processing, and computer vision[34–38]. In memory databases, a lot of work has been done to optimize the hash tables on the CPU platform[39–42], GPU platform[43–45], and Xeon Phi platform[21, 46].

2 Research Background

This section introduces the basis for our work. Specifically, Section 2.1 describes the basic concepts of CHT; Section 2.2 presents the typical implementation of CHT; Section 2.3 describes the cache replacement strategy based on LRU; Section 2.4 presents the basic concepts of the CUDA programming teachniques.

2.1 Basic concepts of CHT

Typical CHT contains two hashing methods to map the Key-Value data (KV)[9]. Figure 1 shows two typical insert scenarios where two different KV KV_1 and KV_2 are inserted into CHT. The row number indicates the location of the hash index, and four slots are set in the bucket at each location to store KV according to typical CHT. For example, H_1 and H_2 represent two independent hashing methods, through which each KV can be mapped to two different candidate positions. KV_1 is mapped to positions 0 and 2 and KV_2 is mapped to positions 3 and 5. When KV_1 is inserted, the corresponding candidate positions 0 and 2 can be found by means of the two hashing methods, both of which have free slots in their buckets for inserting new KV. Based on the specific CHT algorithm, the bucket at position 0 or 2 will be selected for insertion.

When KV_2 is inserted, the corresponding positions 3 and 5 are found through the two hashing methods, and there are no free slots in the buckets at these positions.. Under such circumstances, CHT will choose and evict the key-value data from one slot in the bucket at candidate locations, add the new data to the slot, and meanwhile calculate another candidate position for the evicted KV through the hashing method. If there is any free slot, the evicted KV is inserted, otherwise the eviction and replacement operation will be repeated until any free slot can be found. Typical CHT sets a threshold for the number of eviction and replacement operations, and when this threshold is reached, the eviction and replacement operations stop and the CHT expansion starts. The introduction of eviction and replacement operations to CHT allows more free slots to be used, thus resulting in a CHT load rate greater than 95%. In the insert scenario of KV_2, since there is no free slot in the bucket corresponding to candidate positions 3 and 5, KV with the value of o in the slot at position 5 is randomly selected for the eviction and replacement operation, and KV_2 is inserted in the slot corresponding to o in the bucket at original position 5. The other candidate bucket position of o is calculated as 1 by the hashing method, and no free slot is found after querying the bucket at position 1. Then, a KV d is selected for the eviction and replacement operation, and o is inserted in the slot space where the d locates. The hash value of d is further calculated, and a free slot is found after querying the bucket at the other candidate position 4. Finally, d is inserted into the free slot and the insert operation is completed. Li *et al*.[29] proved that using a breadth-first strategy is an effective way for finding a replacement path during an eviction and replacement operation.

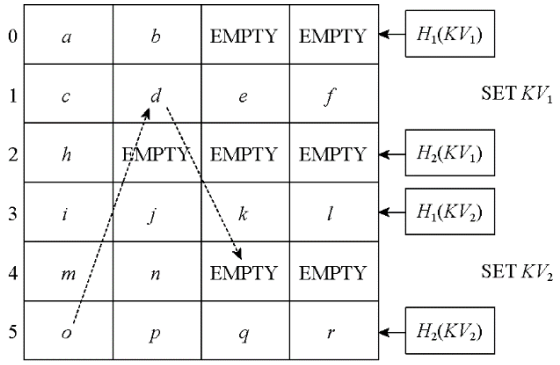

Figure 1 Setting KV_1 and KV_2 on CHT

During the query, the hash values are calculated through H_1 and H_2 to find KV in the slot in the bucket at the corresponding positions, and the stored value data is obtained by traversing and comparing the key data in the slot.

CHT only supports concurrent tasks of the same type, such as concurrent insert operations or concurrent query operations. When the insert and query operations occur at the same time, the insert operation may be replaced because there is no free slot. At this point, if the target data being replaced is queried or the target data to be queried is not returned correctly, the target data is replaced into a slot in another bucket and the query of the indexed data will fail. The hash table loses the data integrity when the above two conditions result in the concurrent execution of the insert and query operations. Compared with CHT, CCHT removes the replacement operations when free slots are not available and supports the concurrent execution of insert and query operations, thus improving the concurrency performance of the hash table.

2.2 Typical implementation of CHT

The performance of CHT in practical applications is affected by a large number of parameters. In CHT, the number of key hash functions and the number of storage spaces in each location affect the load rate and the access latency. In the case of more hash functions in CHT and KV are allowed to be mapped to more locations, the chance of finding a free slot increases because there are more candidate addresses for the requested data. This multi-hash-function approach can effectively improve the load rate of CHT. However, when the query operations are performed, more buckets and slots need to be traversed due to the increase in hash functions, which leads to more frequent access latency and more loading times of cache lines in the processor. In practical applications, the hash function of CHT is set to reduce the query time and the access latency while ensuring the full utilization of slots and the high load rates.

In CHT, increasing the number of slot spaces per bucket can effectively improve the load rate. As the number of slots in the bucket increases, more slots in a single bucket can be accessed during a single insert and read operation, thereby reducing the number of second or more hashing operations and effectively increasing the load rate of CHT. In the practical application of CHT, most designs employ 4 or 8 slots per bucket[9]. The reason for this configuration is that the cache line in the processor can cache data in one or two locations at a time. If more slots are used in each bucket, more cache lines in the processor will be loaded when KV is compared, thereby aggravating the access latency.

The main feature of the CHT algorithm is the ability to provide fast queries. In response to a read request, the maximum number of search spaces is $n \times s$, where n is the number of hash functions and s is the number of slot spaces in a single bucket. About the storage area used for the slots in the bucket, the allocation mode of fixed-length storage space is adopted to facilitate access to the data through arrays or vectors. This approach makes the CHT algorithm easier to implement in different processor architectures, such as CPU, GPU, and Xeon Phi.

2.3 Cache replacement strategy based on LRU

In an in-memory cache system, the system caches a part of the data from the persistent storage medium into the memory to reduce the data access latency. Since the memory storage space is limited and smaller than the storage space required by the whole data, when the memory cache space is full, it is necessary to evict the cached data through the cache replacement strategy and obtain the free space to store the newly cached data. The cache replacement strategy ensures that the most valuable cache data remains in the memory; its effectiveness is usually measured by the hit ratio. In system implementation, increasing access throughput and latency is generally considered as a comprehensive measure indicator. In practice, it is necessary to choose a suitable replacement strategy according to different application loads. For example, in the case of Web applications, recently popular data is frequently accessed, and the Least-Recently-Used (LRU) replacement algorithm is applicable[7, 8].

The LRU algorithm is a typical cache replacement algorithm. The core idea is to take the recently accessed data item as the most valuable item and evict the data item with the longest time since its last access when the replacement operation is required. The flow of the LRU algorithm is shown in Figure 2. The upper limit of the storage space of the data item in the LRU algorithm is 4, and the right side of the storage space is the recently accessed data. Meanwhile, the time label of operation is marked for each data item, including the insert and read operations. During the execution of steps 1–4, when the storage space is not full, data items A, B, C, and D are successively inserted into the storage space. In step 5, since the storage space is full, the LRU algorithm evicts the data item A with the longest time since its last access and inserts a new data item E. In step 6, since data item B is accessed, data item B is moved to the far right

of the storage space and the time label is updated at the same time. In step 7, a new data item F needs to be inserted. Since the storage space is full, the leftmost data item C is evicted and data item F is inserted in the far right of the storage space, which is the same as step 5. Due to the features of simple algorithm implementation, high hit ratios in certain scenarios, and small impact on application performance, LRU is widely used by cache replacement systems, such as memcached[5], memC3[8], and MICA[4].

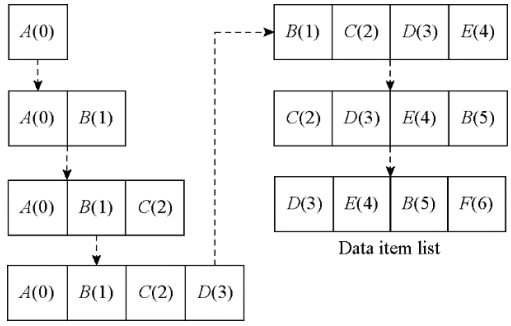

Figure 2 Management of data items by LRU replacement strategy

Linked lists and timestamps are usually adopted to implement the LRU algorithm. When the linked list is adopted, it is usually combined with adjacency hash tables[5] to quickly access the hash table and the data structure managed by the LRU algorithm through the pointer inside the data item. The timestamp method is often applied to the index structure with a fixed item size. By comparing the timestamps, the system evicts the data item with the earliest access[47].

2.4 CUDA programming techniques

Compute Unified Device Architecture (CUDA) is a parallel computing architecture launched by graphics card manufacturer NVIDIA and has gradually developed into the programming specification for NVIDIA GPU.

CUDA provides a programming model based on standard C language and supports the keywords and structures related to GPU operations. The CUDA-based GPU program contains the CPU Host-side execution code and the GPU Device-side code, and the two parts are automatically divided by the CUDA-based compiler for compilation and linking[48].

CUDA enables the developers to write device method code, which is called the kernel. The kernel is executed by the GPU in the form of Single Instruction Multiple Thread (SIMT). During the execution of a program, loading a kernel approach is equivalent to calling a kernel approach; the program developer also needs to specify the corresponding GPU space grid for execution. Each grid contains multiple thread blocks, forming a two-dimensional space, and each block contains multiple threads, forming a three-dimensional space. The threads in each GPU are identified by their ID and the threads in the same block can be synchronized through barriers.

GPU threads gain access to GPU memory during the execution of the kernel approach and the storage operations of each thread include reading/writing private registers and local memory. The local variables in the kernel approach are automatically allocated to the registers or memory. The variables in other GPU memory can be created and managed through interfaces. A CUDA program may contain multiple kernels and all operations can be applied to each kernel. In the design and implementation of CCHT described below, all CHT and replacement operations are implemented in the kernel to improve the execution efficiency of the program; also the implementations in a single processor can be ported to other platforms more conveniently.

3 CCHT Based on Different Index Space Occupations

In this section, the design and implementation of cache indexing methods based on different index space occupations are mainly introduced, including the design of the index structure and the cache replacement algorithm. Specifically, Subsection 3.1 presents the hash index analysis concerning memory cache; Subsection 3.2 describes the double LRU index access method, and Subsection 3.3 describes the coarse LRU index access method.

3.1 Hash index analysis concerning memory cache

In in-memory cache systems, the hash index and the cache replacement strategy are generally implemented separately[5, 8]. The hash index is used to index the cached data in memory, which allows quick access to the target data, while the cache system processes the query requests. Common hash indexes include the CHT algorithm applied in MemC3[8] and the open hashing method used in memcached[5]. The replacement strategy selects and replaces the data to be evicted from the cached data by a certain strategy when the memory cache is full and the new data needs to be cached. Commonly used cache indexing methods include the LRU algorithm mentioned in this paper, FIFO (First-In First-Out), LFU (Least Frequently Used), and CLOCK[49] algorithm. Among them, LRU is widely applied due to its simple implementation, easy maintenance, and ability to meet the requirements of the general workload.

When the CHT algorithm is used as data index, each data corresponds to two optional buckets. If there are no free slots in these two buckets, one data entry will be randomly selected for replacement from these two locations. After that, the hash value is calculated according to the replaced KV, which may lead to further memory access operations, as described in Section 2.1. In an in-memory cache system, there is no persistent storage requirement for the cached data. The cache replacement strategy can evict the cached data based on multi-dimensional indicators such as the hit ratio, the access latency, the space utilization, and the throughput. Therefore, CCHT with the built-in cache strategy is designed in this paper. Specifically, in the in-memory cache system, when the index hash table needs the operations of eviction and replacement, the cache replacement strategy is called based on CHT to evict the data. Two approaches are implemented based on the requirements of index overheads and hit ratios, which will be respectively described in the following sections.

3.2 Double LRU CCHT

The core idea of CCHT design is to reduce the memory access and support the concurrent execution of insert and query operations by adding the cache queue operation in the bucket to remove the replacement operation of CHT when there are no free slots.

The structure of double LRU CCHT is similar to that of the typical LRU CHT. As shown in Figure 3, KV is indexed through a hash table in the structure. Each hash value corresponds to one bucket, and each bucket contains a fixed number of slots. Each slot contains KV and occupies the markers, including two slot pointers for the LRU in the bucket and two slot pointers for the global LRU. Among them, unlike the typical LRU CHT structure, two slot pointers are added for the LRU in the bucket. For the convenience of understanding and subsequent description, a timestamp label is added to each bucket in Figure 3, which is equivalent to the timestamp of the tail unit of the LRU queue in each bucket, and marks the earliest time when a data element is inserted in each bucket.

When the key data and the value data are inserted into the memory cache through CCHT, as shown in Algorithm 1, if there is free cache space in the memory cache, we can obtain the corresponding hash value H_1 through the hash function $Hash_1(key)$ and check whether there is a free slot in the bucket corresponding to H_1. If there is a free slot, KV is inserted into the slot, the

occupation sign is updated, and this slot is placed at the front of the global LRU queue and the bucket LRU queue. If there is no free slot, the hash value H_2 is calculated through $Hash_2(key)$ to check whether there is any free slot in the corresponding bucket. If there is a free slot, the above same operations are performed. If there is no free slot in the buckets corresponding to H_1 and H_2, the bucket with a smaller *timestamp* will be selected after comparison. If the timestamp of H_2 is smaller, the slot at the rear of the LRU queue in the bucket corresponding to the hash value H_2 has not been accessed for a longer time than that corresponding to H_1. The slot at the rear of the LRU queue in the bucket corresponding to the hash value H_2 is evicted, including the release of the corresponding KV and the evictions from the global LRU queue and the LRU queue in the bucket. KV is inserted into the slot and the slot is placed at the front of the global LRU queue and the LRU queue in the bucket. If there is no free cache space in the memory cache when KV is inserted, KV corresponding to the rear slot and the slot pointer of LRU queue need to be evicted through the global LRU queue.

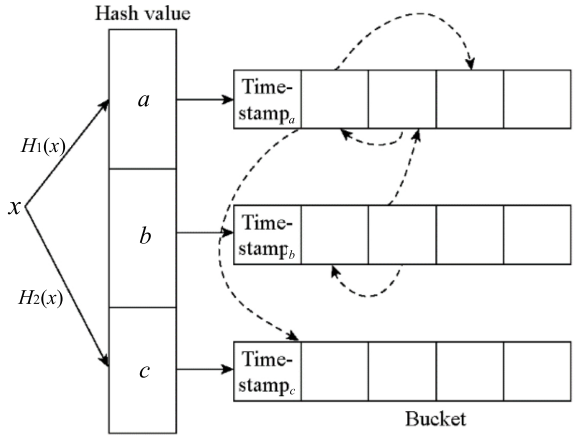

Figure 3 Double LRU CCHT structure

Similar to CHT query method described in Section 2.1, CCHT query method is shown in Algorithm 2. When the query result is returned, if CCHT contains KV for the desired query, the slot corresponding to KV is placed at the front of the global LRU queue and the LRU queue in the bucket.

Through the double LRU CCHT, the operations of eviction and replacement in the original insert process of KV are changed into cache replacement, which eliminates the memory access times and the query time of the free slots caused by the replacement operation. The double LRU CCHT ensures the hit ratio of the in-memory cache system through the independent use of the global LRU queue and the in-bucket LRU queue. Compared with LRU CHT, this method increases the overhead of the pointer storage space for the LRU queue in the bucket. Therefore, we propose a coarse LRU CCHT structure, which saves the storage space occupied by the pointers when compared with double LRU CCHT.

3.3 Coarse LRU CCHT method

To save the storage space occupied by pointers in CCHT, we propose a coarse LRU CCHT method, as shown in Figure 4. In contrast to the double LRU CCHT indexing structure, the slot pointer for the global LRU queues is removed and the bucket-based LRU queue operation is added. Each bucket contains two bucket pointers to maintain a coarse LRU queue.

When the operations of insertion and query are successfully performed through the coarse

LRU CCHT method, the corresponding bucket is placed at the front of the bucket LRU queue, and the corresponding index unit is placed at the front of the LRU queue in the bucket. When the data is inserted, if there is no free cache space in the memory cache, the bucket at the rear of the LRU queue is selected, and the slot at the rear of the corresponding LRU queue is selected to evict and release KV.

Algorithm 1. Double LRU CCHT insert algorithm.

Input: key, value;
Output: insert completion state.
1. **if** $Existfreespace()$!= true **then** //if there exist free space
2. $Evictglobaltail()$ //evict global LRU tail unit
3. $UpdateLRU(H, i)$
4. **end if**
5. H_1 = $Hash_1(key)$//compute the hash value of key with hash function 1
6. **if** $i = Findfree(H_1)$ **then** //find a free space in the Bucket H_1
7. $Set(H_1, i, key, value)$ //insert key and value into free space
8. $UpdateLRU(H, i)$ //update LRU queue
9. **return** success
10. **end if**
11. H_2 = $Hash_2(key)$ //compute the hash value of key with hash function 2
12. **if** $i = Findfree(H_2)$ **then**
13. $Set(H_2, i, key, value)$
14. $UpdateLRU(H, i)$
15. **return** success
16. **end if**
17. $H = Comparetimestamp(H_1, H_2)$ //find a hash value through comparing the timestamps of H_1 and H_2
18. $i = EvictBucketTail(H)$ //evict the tail unit of bucket H LRU queue
19. $Set(H, i, key, value)$
20. $updateLRU(H, i)$
21. **return** success
22. Procedure $UpdateLRU(H, i)$:
23. $UpdateBucketLRU(H, i)$ //update bucket internal LRU queue
24. $UpdateGlobalLRU(H, i)$ //update global LRU queue

Algorithm 2. Double LRU CCHT query algorithm.

Input: key;
Output: value.
1. $H = hash_1(key)$
2. **for** $i = 0; i < BucketInternalMax; i++$ **do**
3. **if** $CompareKey(Bucket[H][i] \rightarrow key, key)$ **then** //compare the value of key
4. $UpdateLRU(H, i)$
5. **return** $Bucket[H][i] \rightarrow value$
6. **end if**
7. **end for**
8. $H = Hash_2(key)$
9. **for** $i = 0; i < BucketInternalMax; i++$ **do**
10. **if** $CompareKey(Bucket[H][i] \rightarrow key, key)$ **then**
11. $UpdateLRU(H, i)$
12. **return** $Bucket[H][i] \rightarrow value$
13. **end if**
14. **end for**
15. **return** NULL

The coarse LRU CCHT method replaces the global LRU index in the double LRU CCHT

method and the LRU CHT method by the bucket LRU algorithm. In comparison with the double LRU CCHT, $2 \times m \times (s-1)$ slot pointers are released, where m is the number of buckets in CCHT and s is the number of slots in a single bucket, thereby saving the index storage for CCHT.

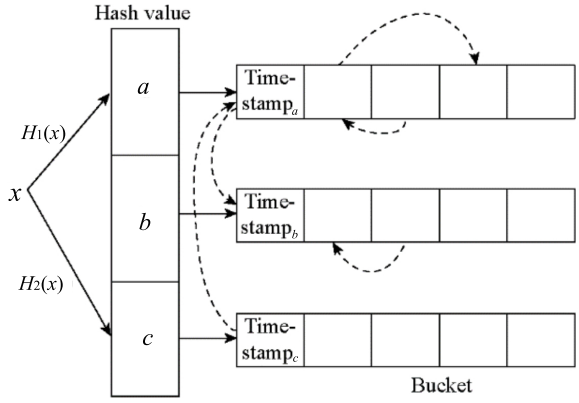

Figure 4 Coarse LRU CCHT structure

4 Implementation of CCHT in CPUGPU-Oriented Heterogeneous Environment

According to CCHT described in Section 3, this section mainly introduces the implementation of CCHT in the CPUGPU heterogeneous environment. Subsection 4.1 describes the multi-level data indexing structure of the CPUGPU heterogeneous storage; Subsection 4.2 presents the application interface functions of CCHT in a heterogeneous environment; Subsection 4.3 describes the implementation details of CCHT; and Subsection 4.4 shows the optimization of CCHT in the implementation process.

4.1 Multi-level data indexing structure of heterogeneous storage

In real applications, the hash tables with graph data, log data, and video data as the value data have a large storage space overhead. The design and performance of the processing with the GPU memory used for storing the hash table are limited by the bus bandwidth of the system PCI and the size of the GPU memory. Therefore, we design a multi-level data indexing structure based on an out-of-core system. When the storage space is initialized, contiguous storage is allocated in the CPU memory. When the hash table is operated on the GPU, it can be represented by the index sequence value corresponding to the original value data on the CPU, which reduces the memory occupation of the GPU and the transmission cost of the PCI-E bandwidth.

The distribution of KV on the CPU and GPU is shown in Figure 5. For example, if KV pair KV_1 is inserted, the value data V_1 is stored in the storage area of continuous value in the CPU memory, with the position identifier ID set to 0. The key data K_1 and the index position $ID = 0$ are transferred to the GPU memory through the system bus and stored in the free slots in the corresponding hash bucket. When querying KV_3, K_3 will be transferred to the GPU memory through the system bus. After the completion of the query, the corresponding position identifier $ID = 2$ will be returned through the bus. With the multi-level heterogeneous index structure of CPUGPU, the memory occupation and bus bandwidth in the GPU are reduced effectively. In the GPU memory, in addition to the necessary key data for query and the position identification only containing the value data, the memory occupation in the limited GPU space is reduced

relative to the original value data. During the request processing, the value data to be inserted and queried is not transmitted to the GPU memory through the bus, which effectively avoids the bus bandwidth occupation and the performance loss caused by too large value data.

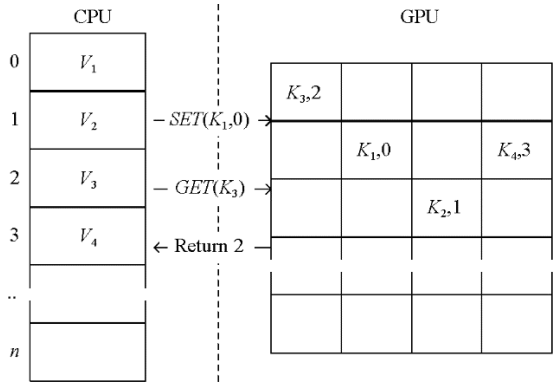

Figure 5 CCHT data structure under CPUGPU

4.2 Application interface functions of CCHT in a heterogeneous environment

The implementation methods of CCHT mainly include the library access functions, initialization functions of the data storage area, and GPU kernel functions for handling hash tables. The GPU programming framework of CUDA version 9.1 is adopted in this paper, with a total of 1 385 lines of code, including 324 lines of library access functions and initialization functions of the data storage area and 953 lines for the GPU kernel function, and the rest being the macro definitions and the global variable indexes.

The implementation methods and function descriptions are shown in Table 1 and Table 2 for the CPU and the GPU, respectively. Table 1 shows the functions implemented on the CPU, which mainly include the library access functions called by other CPU programs to manipulate KV data, the data initialization functions for the indexes and content initialization in CPU and GPU storage areas, and the functions submitting the task and KV to the GPU kernel. CPU functions mainly provide the interfaces for other programs to operate KV, encapsulate the kernel functions on the GPU, and realize the transparency of GPU operations when they are called by other programs. Table 2 shows the kernel functions on GPU, which mainly include the global function used by the GPU to receive and parse the task and data of the key-value operation submitted by the CPU, the device function for the hash table operation, and the device function for the eviction operation of the cache queue. The kernel function implemented on the GPU is used to process the corresponding KV on the hash table in the GPU storage area.

Table 1 Interface and implementation methods on CPU

Method	Function description
set	Insert the key-value data
get	Query the key-value data
del	Delete the key-value data
CCHT_init	Initialize CPU data structure and GPU storage index area
flush_ops	Submit the operations to be performed and KV in the buffer to the GPU, with the type identifier of the request operation in each task

The operations of CCHT on the GPU only include four branch kernel functions: CCHT_set, CCHT_get, CCHT_del, and CCHT_evict, which conforms to the Single Instruction Multiple

Thread (SIMT) form of GPU executions. CCHT can achieve better concurrency performance when executed in a heterogeneous CPUGPU environment.

Table 2 Kernel method on GPU

Method	Function description
CCHT_process	Global function used by CPU for submitting the key-value processing tasks to GPU, parsing the task type and executing the corresponding device functions
CCHT_set	Device function used for inserting the data into the hash table
CCHT_get	Device function used for querying and getting the target KV in the hash table
CCHT_del	Device function used for deleting the target KV in the hash table
CCHT_evict	Device function used for evicting the data at the rear of the cache queue in the hash table
hash1, hash2	Device function used for calculating the hash value of the key data

4.3 Implementation details

When initializing the data storage area, this study uses the malloc and cudaMalloc functions to allocate the required storage areas, including the storage area for executing cache operations and KV on the CPU, the storage area for the returned GPU execution results, the storage area on the GPU that receives the execution operation and KV, the storage area of KV in the hash table with the slot as a unit, the storage area of the identification in the bucket, and the storage area of the execution results of the hash table. After the allocation is completed, the memset and cudaMemset are used to initialize the data storage areas.

The functions manipulating KV on CPU copy KV and the operations passed by other programs into the buffer queue of KV and tasks in the CPU memory. All functions that manipulate KV share the same buffer queue of data and tasks, and allow multiple write and read operations in the task buffer queue. When the threshold value of the number of tasks to be submitted to the GPU is reached, the flush_ops function is called to pass the data and the corresponding operation identifier to GPU through the global function CCHT_process. When the task submitted to the GPU is completed, the result is returned to the specified storage area and further processed with the programs running on the CPU, such as determining whether the read operation is hit and updating the size of the available cache space.

When the kernel function is executed on the GPU, each kernel thread performs a single operation for the given task and KV, which is specified according to the global id of the GPU thread. CCHT_process receives KV and the task data passed into the GPU memory and parses the task data according to its identifier to obtain the operation type. CCHT_set, CCHT_get or CCHT_del are selected for different hash table operations, and the operation result is returned to the specified storage area in the GPU memory. When the memory cache space is full, the task data of the write operation passed by the CPU contains the operations of replacement and write, and the GPU calls CCHT_set to perform the write operation after calling CCHT_evict to perform the eviction operation.

4.4 Optimization details

CCHT submits the tasks by the batch processing mode. When other programs call the library access function, the passed KV and the corresponding operation are copied to the corresponding buffer. When the threshold of the number of tasks submitted through the batch mode is reached, KV and the corresponding operations are submitted to the GPU kernel function and processed concurrently by the GPU thread according to their thread ID. Through the batch processing mode, the memory access and transmission frequency between CPU and GPU are reduced, as well as the access to the PCI-E bus. At the same time, the concurrency given by

the GPU multithreading can be fully utilized to improve the processing performance of the hash table tasks.

One kind of instructions can be executed in a single warp through the dimension parameter configuration of GPU. In CCHT, instruction branches occur after the execution of the global function CCHT_process in each thread, and different device functions need to be called according to the operation type. If multiple operations are mixed in the same warp, different operation types in the same warp triggered by warp divergence will be executed in sequence. In this paper, the synchronous waiting caused by instruction branching is avoided by restricting only a single thread in the same warp. Although only executing a single thread in the same warp affects the concurrency performance of the GPU, the concurrency based on warp granularity can still achieve a good throughput effect, which is verified in Subsection 5.5.

Regarding the implementation of the buffer cache of KV and tasks, the method of mixing and reuse is adopted. KV and task for all operations share a contiguous buffer. When the result is returned, the data is returned to the data buffer that submits the task. The implementation of mixing and reuse reduces the space overhead of CPU and GPU memory and releases more memory space for the storage of KV. At the same time, compared with the multi-segment buffer-cache design, this method increases the proportion of continuous memory access operations, thereby reducing the memory access of data storage in the multi-segment buffer cache.

The CUDA-based native locking mechanism of atomic operations, in CCHT, contains the concurrent executions of write and read of slot data and the changes of LRU queues, and requires read-write lock design for each slot and LRU queue. In other words, it is necessary to support multiple threads to operate the data at the same time so to obtain better concurrency performance and prevent the long latency of single-thread tasks due to lock waiting. The lock identification data is set in the GPU memory, with the default value of 0. The write operation is implemented by the atomicMax() method, which returns the existing identification data and updates the identification data to be the the maximum value of the existing identification data and the inserted data. In the case of the inserted data value being 1, when the returned value is 0, the lock is free, which is then unlocked by zero setting with atomicExch(). When the returned value is not 0, it indicates that another read or write thread is accessing the data. The read operation is implemented by the atomicAdd() method, which returns the existing identification data and updates the identification data to be the sum of the existing identification data and the inserted data. When the returned value modulo 2 is not 0, it indicates that a writer thread is accessing the data. When the returned value modulo 2 is 0, it indicates that no other writer thread is accessing the data. After lock holding and operation completion, the lock held by the current read thread is released through the atomicSub() method. The above method allows a single write operation or multiple read operations at the same time. CUDA-based native lock mechanism of atomic operation replaces the assignment method of traditional CUDA-based atomic data operation and breaks the upper limit on the size of atomic operation data dependent on CUDA-based library functions. During the processing of a task request on the GPU side of CCHT, the thread can only acquire the lock of a single slot at the same time, and there is no deadlock caused by simultaneous preemption of multiple lock resources.

5 Experimental Results and Analysis

The experiments are carried out on the CPU +GPU heterogeneous server, with the CPU being an Intel(R) Core(TM) i7-6700 K, the dominant frequency a quad-core 4.00 GHz, the memory a DDR4 32 GB, the GPU an NVIDIA GeForce GTX 1080 TI with 11 GB of video memory.

In the experiments, the data set generated by YCSB[50] is used for testing. The length of

the key value of the data is 24 B; the value type is 100 B, and the element size of the inserted data set is 3×10^6. Data manipulation requests include three typical workloads: Zipf, latest, and uniform. The distribution skewness of the Zipf workload is 0.99; the latest workload is the most recently used data request, and the uniform workload has the same data query probability. Each workload request contains two different ratios of insert to query operations, namely, the proportion of query operations is 100% and 50% respectively. According to the characteristics of typical running environment of memory cache[12], insert operation is carried out after the query return is lost. During the experiments, data set loading and cache warm-up operations are carried out first, and then the workload request operation is carried out.

To verify the application performance of CCHT, we implement a prototype cache replacement system with CCHT as the core and another three cache indexing algorithms for comparison: LRU CHT, LRU open hash, and random CHT. Among them, LRU CHT and random CHT are compared and verified on the CPUGPU heterogeneous platform. Since LRU open hash is not suitable for the initialization of fixed storage space on the CPUGPU heterogeneous platform due to the characteristics of dynamic data space allocation, LRU open hash is implemented, compared and verified on the CPU platform. The hash value length of CCHT and CHT is set to 2^4, with four index storage units in the bucket corresponding to each hash value. The upper limit of eviction and replacement operations of CHT is 5,000. The threshold value of CCHT batch processing is set to 2,000. The load rate of the hash tables is represented by the ratio of the cache space size to the maximum index number of the hash tables.

5.1 Average memory access per insert

Average memory access per insert refers to the average memory access during data set loading and cache warm-up operations. In this paper, the content and order of the data warmed up and loaded in the workloads of latest, zipf, and uniform are the same, as shown in Figure 6. When the ratio of cache space size to hash table size is greater than 60%, both double LRU CCHT and coarse LRU CCHT are significantly lower than other algorithms in terms of the average number of memory access per insert. When the ratio of cache space size to hash table size is 80%, the average memory access times of both double LRU CCHT and coarse LRU CCHT are reduced by 30.39% in comparison with LRU CHT. When the ratio of cache space size to hash table size is 90%, compared with that of LRU CHT, the average memory access times of both double LRU CCHT and coarse LRU CCHT are reduced by 94.63%. The results show that the two methods of CCHT can effectively reduce the memory access. When the ratio of cache space size to hash table size is high, CCHT removes the operations of eviction and replacement relative to CHT, which can significantly reduce the access to the hash table, as well as the memory access.

5.2 Average memory access per request

Average memory access per request refers to the average memory access during the loading of the workload test set. Figure 7 presents the experimental results of average memory access per request with the cache space for the five cache indexing algorithms under six different workloads.

When the ratio of cache space size to hash table size is greater than 60%, both double LRU CCHT and coarse LRU CCHT are lower than other algorithms in terms of the average memory access per request. When the proportion of query operation is 50%, average memory access is reduced more obviously by double LRU CCHT and coarse LRU CCHT than by LRU CHT. When the ratio of cache space size to hash table size is 70%, 80%, and 90% and the proportion of query operation is 50%, the average memory access per request of double LRU CCHT is decreased by 10.59%, 32.86%, and 97.25% respectively under three workloads of latest, zipf and uniform, and that of coarse LRU CCHT is decreased by 9.50%, 32.91%, and

97.29% respectively. This is because there are a lot of insert operations when the proportion of query operation in the workloads is 50%. As the cache space increases, LRU CHT and random CHT require more eviction and replacement operations to obtain free index space, resulting in a large amount of memory access. Instead, double LRU CCHT and coarse LRU CCHT do not access more index locations when the cache space increases.

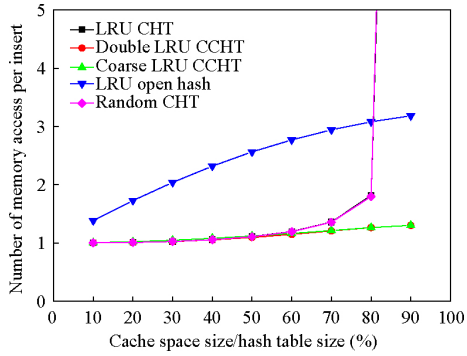

Figure 6 Average memory access per insert with hash table size

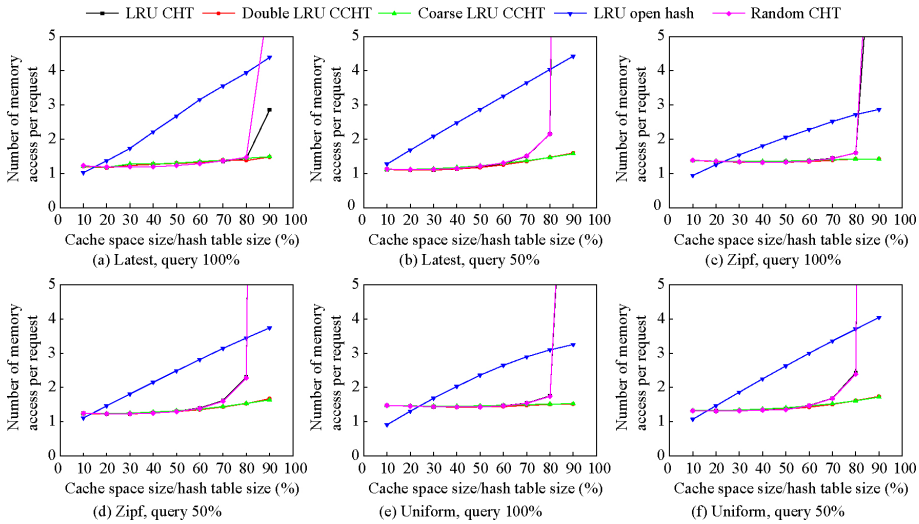

Figure 7 Average memory access per request with hash table size

5.3 Hit ratio

The hit ratio refers to the ratio of the number of successful queries to the total number of queries during the loading of the workload test set. Figure 8 shows the hit ratios of the five cache indexing algorithms under six different workloads. In the four workloads of zipf and uniform, double LRU CCHT and coarse LRU CCHT have the same hit ratio as LRU CHT, and the hit ratio increases with the increase in cache space. The maximum difference of the hit ratio between double LRU CCHT and LRU CHT is not greater than 0.12%, and that between coarse LRU CCHT and LRU CHT is not greater than 0.22%. In the two workloads of latest, the maximum difference of the hit ratio between double LRU CCHT and LRU CHT is not greater than 0.18%, and that between coarse LRU CCHT and LRU CHT is not greater than 0.56%. When the hash

table bucket introduced by CCHT has no free slots, slots are evicted from the bucket for the insert operation through the LRU algorithm. CCHT increases the number of eviction operations in cache relative to LRU CHT and LRU open hash, resulting in the deviation of the hit ratio. At the same time, because of the eviction strategy based on the LRU queue, the hit ratio of CCHT has a smaller deviation than that of CHT.

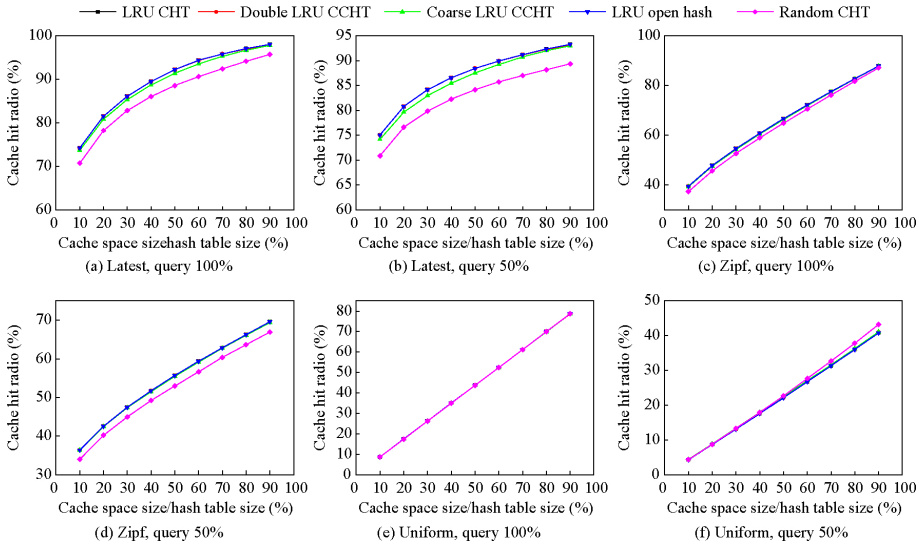

Figure 8 Cache hit ratio with hash table size

5.4 Index overhead analysis

Index overhead refers to the number of slot index pointers in the LRU queue when the cache space is full during the storage. CCHT introduces the space overhead of slot index pointers for LRU queues while removing the replacement operations of CHT. The experiments in this section analyze the index overhead of double LRU CCHT and coarse LRU CCHT under different cache space sizes, as shown in Table 3. When the ratio of cache space size to hash table size is at least 30%, the overhead of coarse LRU CCHT is significantly lower than that of double LRU CCHT. When the ratio of cache space size to hash table size is 90%, the overhead of coarse LRU CCHT is reduced by 31.71% compared with that of the double LRU CCHT. When the ratio of cache space size to hash table size is at most 20%, the index overhead of coarse LRU CCHT is greater than that of double LRU CCHT, because part of the index overhead of coarse LRU comes from the index between the buckets corresponding to the hash values in the hash table and does not vary with the cache space size. When the cache space size increases, the number of applied slots increases, and the index overhead mainly comes from the link index between slots. However, the slot index overhead of global LRU queue of double LRU CCHT is greater than that of coarse LRU CCHT, which results in the gradually increased overhead ratio between the two CCHT algorithms.

5.5 Throughput performance of CCHT on CPUGPU

The throughput performance of CCHT on the CPU+GPU heterogeneous server environment is verified by experiments in this study. The throughput performance on a heterogeneous CPUGPU environment refers to the number of requests per second processed by CCHT. Under the workloads with 80% cache space size/hash table size, the results of the performance

experiments are shown in Figure 9. Under different workloads, the performance of CCHT is significantly improved by 126.43 times, 143.17 times, and 1.78 times respectively when compared with LRU CHT, random CHT and LRU open hash. Here, the reason why LRU CHT and random CHT have the lowest performance is that the read and write requests in CHT can only be executed sequentially, not concurrently. Therefore, when the operation type is switched, data transmission through the PCI-E bus brings frequent kernel context switches and driver calls, and eventually leads to a significant decrease in the performance, even far below the performance of the LRU open hash algorithm on the CPU platform. Compared with LRU CHT, CCHT supports the concurrent execution of all types of operations, makes the most of concurrent hardware resources such as the stream processor on the GPU, and effectively reduces the additional overhead caused by frequent data transmission between the CPU and the GPU. In addition, the frequent replacement operation of the write operation under the high load rate is removed and the memory access times during the GPU processing are reduced, thereby improving the index access performance of the hash table.

Table 3 Pointer counts with different cache sizes for CCHT

Cache space size/hash table size (%)	Pointer count for double LRU CCHT (k)	Pointer count for coarse LRU CCHT (k)
10	235.92	314.57
20	340.78	367.00
30	445.63	419.42
40	550.49	471.85
50	655.35	524.28
60	760.21	576.71
70	865.06	629.14
80	969.91	681.56
90	1 074.77	733.99

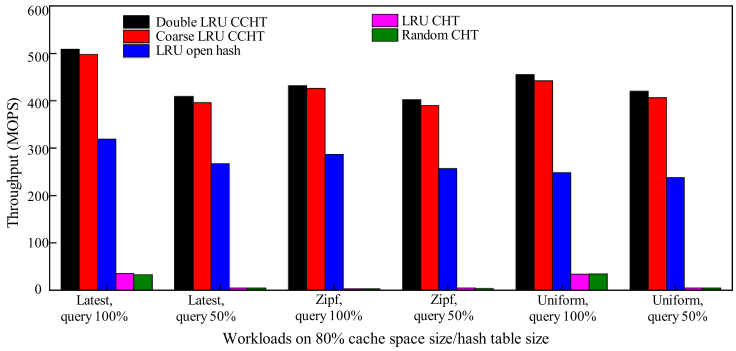

Figure 9 Performance on workloads with 80% cache space size/hash table size

5.6 Latency of CCHT on CPUGPU

The latency of CCHT on the CPU +GPU heterogeneous server environment is studied by experiments in this section. The latency experiment includes the data transfer latency and the average latency of a single operation in the CPU +GPU heterogeneous server environment. The data transfer latency represents the latency of data transfer between CPU and GPU memory when the hash table calls the GPU clarify what this should mean. In CCHT, it refers to the sum of the time copying the request data and the operation command buffer from CPU memory to GPU memory and the time copying the request results from GPU memory to CPU memory when the GPU is called to process the hash table requests. The data transfer latency is shown in

Table 4. The scale of data transfer for a single call of GPU is the same in each workload for four hash tables. The latest workload with the query proportion of 100% and the 80% cache space size/hash table size is used as a representative for experimental verification. The data transfer latency of both double LRU CCHT and coarse LRU CCHT is higher than that of LRU CHT and random CHT due to the batch processing mode of CCHT, which allows the hash table to process the large-scale requests concurrently on the GPU. With the batch processing threshold of 2 000 as an example, when CCHT calls the GPU, the single data transfer includes 2 000 operation data and operation request types, and the average transfer latency per single operation data and request type is much lower than that of LRU CHT and random CHT.

Table 4 CPUGPU data transfer latency

Hash table type	Latency (μs)
Double LRU CCHT	40
Coarse LRU CCHT	40
LRU CHT	13
Random CHT	12

Further, the average latency per operation is compared and evaluated, namely the average latency for the hash table to process a single request in the CPU or GPU environment. As to the LRU open hash, it is the average latency of a single request processing in the CPU environment, and for the remaining hash tables, it is the average latency of a single request processing in the GPU environment. The average latency for a single operation is shown in Figure 10. The latency of double LRU CCHT and coarse LRU CCHT is higher than that of other hash tables, which is because CCHT realizes the large-scale concurrent operation on GPU. Although the latency overhead of branch waiting is avoided by warp allocation, there still exists lock contention for the slot and the LRU queue among different threads and the synchronization overhead in the same-batch processing. In other words, the threads completing the operation should wait the threads that have not completed the operation yet and return together after they are completed. This kind of operation brings the additional overhead and leads to the increased latency. It can be seen from the experimental results that the latency of coarse LRU CCHT is also higher than that of double LRU CCHT. This is because coarse LRU CCHT operates the same LRU queue shared globally and within the bucket in the GPU environment, and compared with the double LRU CCHT with only the lock in the bucket operating in the GPU environment, there is more lock contention in coarse LRU CCHT. Although LRU CHT and random CHT are lower than double LRU CCHT and coarse LRU CCHT in terms of the average latency, the maximum single latency reaches 143 μs due to the frequent replacement of slot data that might occur during an insert operation under high loads, which is much higher than the average latency. LRU open hash has a latency of 0.36 μs–0.49 μs since it has no lock contention and runs in the CPU environment.

6 Conclusions

The performance of data indexing depends on the average memory access. Regarding the traditional CHT data indexing used in a cache, frequent evictions and replacements of CHT increase the memory access when the ratio of the cache space size to CHT space size is high, thus affecting the cache performance. In this paper, the hit ratio and index overhead of the in-memory cache system based on KV are analyzed. According to the overhead of index pointers, double LRU CCHT and coarse LRU CCHT are proposed. A multi-level index data structure for reducing the bus transmission and the GPU memory occupation in a CPUGPU environment is provided and implemented in a heterogeneous CPUGPU environment. The experimental results show that CCHT can effectively reduce the memory access while ensuring the hit ratio of cache,

and has good scalability and throughput performance on GPU.

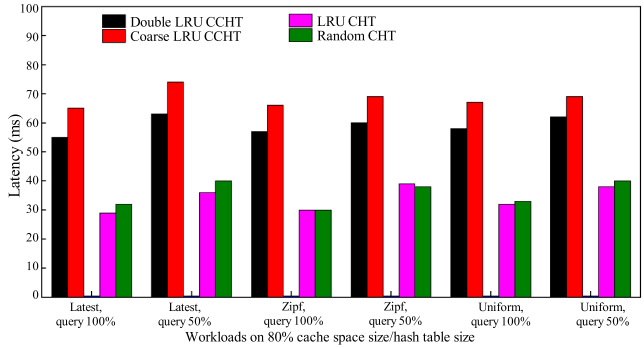

Figure 10 Latency on workloads with 80% cache space size/hash table size

In the future, we hope to further optimize the overhead of index pointers and improve the hit ratio of caches. Regarding the implementation method on GPU, the slot lock and LRU queue lock should be optimized to enhance the performance of the concurrent insert operations. The task type should be identified before the hash task submits the task queue, and the warp where the task execution locates should be pre-assigned so that the concurrent execution of multiple hash table operations can be realized in the same warp and the concurrent performance at the level of thread granularity can be achieved, thereby improving the hardware resource utilization of the GPU.

References

[1] Boncz PA, Manegold S, Kersten ML. Database architecture optimized for the new bottleneck: Memory access. VLDB, 1999, 99: 54–65.

[2] DeWitt DJ, Katz RH, Olken F, et al. Implementation techniques for main memory database systems. Proc. of the 1984 ACM SIGMOD Int'l Conf. on Management of Data. 1984. 1–8.

[3] Kemper A, Neumann T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. Proc. of the 27th IEEE Int'l Conf. on Data Engineering. 2011. 195–206.

[4] Lim H, Han D, Andersen DG, Kaminsky M. MICA: A holistic approach to fast in-memory key-value storage. Proc. of the 11th USENIX Symp. on Networked Systems Design and Implementation. 2014. 429–444.

[5] Fitzpatrick B. Distributed caching with memcached. Linux Journal, 2004, 2004(124): 5.

[6] Ma YZ, Meng XF. Research on indexing for cloud data management. Ruan Jian Xue Bao/Journal of Software, 2015, 26(1): 145–166. http://www.jos.org.cn/1000-9825/4688.htm [doi: 10.13328/j.cnki.jos.004688]

[7] Ousterhout J, Gopalan A, Gupta A, et al. The RAMCloud storage system. ACM Trans. on Computer Systems (TOCS), 2015, 33(3): 1–55.

[8] Fan B, Andersen DG, Kaminsky M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. Proc. of the Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation. 2013. 371–384.

[9] Pagh R, Rodler FF. Cuckoo hashing. Proc. of the European Symp. on Algorithms. Berlin, Heidelberg: Springer-Verlag, 2001. 121–133.

[10] Breslow AD, Zhang DP, Greathouse JL, et al. Horton tables: Fast hash tables for in-memory data-intensive computing. Proc. of the 2016 USENIX Annual Technical Conf. 2016. 281–294.

[11] Khorasani F, Belviranli ME, Gupta R, et al. Stadium hashing: Scalable and flexible hashing on GPUs. Proc. of the 2015 Int'l Conf. on Parallel Architecture and Compilation (PACT). IEEE, 2015. 63–74.

[12] Barber R, Lohman G, Pandis I, et al. Memory-efficient hash joins. Proc. of the VLDB Endowment, 2014, 8(4): 353–364.

[13] Lim H, Fan B, Andersen DG, et al. SILT: A memory-efficient, high-performance key-value store. Proc. of the 23rd ACM Symp. on Operating Systems Principles. 2011. 1–13.

[14] Herlihy M, Shavit N, Tzafrir M. Hopscotch hashing. Proc. of the Int'l Symp. on Distributed Computing. Berlin, Heidelberg: Springer-Verlag, 2008. 350–364.

[15] Pagh R, Wei Z, Yi K, et al. Cache-oblivious hashing. Algorithmica, 2014, 69(4): 864–883.

[16] Debnath B, Sengupta S, Li J. FlashStore: High throughput persistent key-value store. Proc. of the VLDB Endowment, 2010, 3(1-2): 1414–1425.

[17] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload analysis of a large-scale key-value store. Proc. of the 12th ACM SIGMETRICS/PERFORMANCE Joint Int'l Conf. on Measurement and Modeling of Computer Systems. 2012. 53–64.

[18] Raman V, Attaluri G, Barber R, et al. DB2 with BLU acceleration: So much more than just a column store. Proc. of the VLDB Endowment, 2013, 6(11): 1080–1091.

[19] Nishtala R, Fugal H, Grimm S, et al. Scaling memcache at facebook. Proc. of the Presented as Part of the 10th USENIX Symp. on Networked Systems Design and Implementation. 2013. 385–398.

[20] Wang HM, Mao XG, Ding B, Shen J, Luo L, Ren Y. New insights into system software. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 22–32. http://www.jos.org.cn/1000-9825/5648.htm [doi: 10.13328/j.cnki.jos.005648]

[21] Polychroniou O, Raghavan A, Ross KA. Rethinking SIMD vectorization for in-memory databases. Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. 2015. 1493–1508.

[22] Ross KA. Efficient hash probes on modern processors. Proc. of the 23rd IEEE Int'l Conf. on Data Engineering. IEEE, 2007. 1297–1301.

[23] Zhang K, Wang K, Yuan Y, et al. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. Proc. of the VLDB Endowment, 2015, 8(11): 1226–1237.

[24] Sun Y, Hua Y, Jiang S, et al. SmartCuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. Proc. of the 2017 USENIX Annual Technical Conf. 2017. 553–565.

[25] Ailamaki A, DeWitt DJ, Hill MD, et al. DBMSs on a modern processor: Where does time go. Proc. of the 25th Int'l Conf. on Very Large Data Bases. Edinburgh, 1999. 266–277.

[26] Boncz PA, Kersten ML, Manegold S. Breaking the memory wall in MonetDB. Communications of the ACM, 2008, 51(12): 77–85.

[27] McKee SA. Reflections on the memory wall. Proc. of the 1st Conf. on Computing Frontiers. 2004. 162.

[28] Wulf WA, McKee SA. Hitting the memory wall: Implications of the obvious. ACM SIGARCH Computer Architecture News, 1995, 23(1): 20–24.

[29] Xie X, Liang Y, Sun G, et al. An efficient compiler framework for cache bypassing on GPU. Proc. of the IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD). IEEE, 2013. 516–523.

[30] Xie X, Liang Y, Wang Y, et al. Coordinated static and dynamic cache bypassing for GPUs. Proc. of the 21st IEEE Int'l Symp. on High Performance Computer Architecture (HPCA). IEEE, 2015. 76–88.

[31] Li X, Andersen DG, Kaminsky M, et al. Algorithmic improvements for fast concurrent cuckoo hashing. Proc. of the 9th European Conf. on Computer Systems. 2014. 1–14.

[32] Hetherington TH, O'Connor M, Aamodt TM. Memcachedgpu: Scaling-up scale-out key-value stores. Proc. of the 6th ACM Symp. on Cloud Computing. 2015. 43–57.

[33] Hetherington TH, Rogers TG, Hsu L, et al. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. Proc. of the 2012 IEEE Int'l Symp. on Performance Analysis of Systems and Software. IEEE, 2012. 88–98.

[34] Alcantara DA, Sharf A, Abbasinejad F, et al. Real-time parallel hashing on the GPU. ACM Trans. on Graphics (TOG), 2009, 28(5): 1–9.

[35] Alcantara DA, Volkov V, Sengupta S, et al. Building an Efficient Hash Table on the GPU. GPU

Computing Gems Jade Edition, Morgan Kaufmann Publishers, 2012. 39–53.

[36] García I, Lefebvre S, Hornus S, et al. Coherent parallel hashing. ACM Trans. on Graphics (TOG), 2011, 30(6): 1–8.

[37] Korman S, Avidan S. Coherency sensitive hashing. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2015, 38(6): 1099–1112.

[38] Lefebvre S, Hoppe H. Perfect spatial hashing. ACM Trans. on Graphics (TOG), 2006, 25(3): 579–588.

[39] Metreveli Z, Zeldovich N, Kaashoek MF. CPhash: A cache-partitioned hash table. ACM SIGPLAN Notices, 2012, 47(8): 319–320.

[40] Balkesen C, Alonso G, Teubner J, et al. Multi-core, main-memory joins: Sort vs. hash revisited. Proc. of the VLDB Endowment, 2013, 7(1): 85–96.

[41] Balkesen C, Teubner J, Alonso G, et al. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. Proc. of the 29th IEEE Int'l Conf. on Data Engineering (ICDE). IEEE, 2013. 362–373.

[42] Blanas S, Li Y, Patel JM. Design and evaluation of main memory hash join algorithms for multi-core CPUs. Proc. of the 2011 ACM SIGMOD Int'l Conf. on Management of Data. 2011. 37–48.

[43] He B, Yang K, Fang R, et al. Relational joins on graphics processors. Proc. of the 2008 ACM SIGMOD Int'l Conf. on Management of Data. 2008. 511–524.

[44] He J, Lu M, He B. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. Proc. of the VLDB Endowment, 2013, 6(10): 889–900.

[45] Keckler SW, Dally WJ, Khailany B, et al. GPUs and the future of parallel computing. IEEE Micro, 2011, 31(5): 7–17.

[46] Jha S, He B, Lu M, et al. Improving main memory hash joins on intel xeon phi processors: An experimental approach. Proc. of the VLDB Endowment, 2015, 8(6): 642–653.

[47] Sanchez D, Kozyrakis C. The ZCache: Decoupling ways and associativity. Proc. of the 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture. IEEE, 2010. 187–198.

[48] NVIDIA Corporation. NVIDIA CUDA programming guide, version 10.0: Reference description. 2020. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[49] Corbato FJ. A paging experiment with the multics system. Technical Report, Massachusetts Inst of Tech Cambridge Project Mac, 1968.

[50] Cooper BF, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB. Proc. of the 1st ACM Symp. on Cloud computing. 2010. 143–154.

Hongjun Zhang, bachelor. His research interests include operating system, big data processing, distributed systesms, and parallel computing.

Heng Zhang, Ph.D., CCF student member. His research interests include distributed systems, parallel computing, big data processing, and operating systems.

Yanjun Wu, Ph.D., professor, doctoral supervisor, CCF senior member. His research interests include operating systems and system security.

Libo Zhang, Ph.D. His research interests include artificial intelligence, deep learning, and computer vision.