University of Mississippi

eGrove

Electronic Theses and Dissertations | Graduate School

1-1-2021

DACHash: A Dynamic, Cache-Aware and Concurrent Hash Table on GPUs

Hao Zhou
University of Mississippi

Follow this and additional works at: https://egrove.olemiss.edu/etd

 Part of the Computer Engineering Commons

Recommended Citation

Zhou, Hao, "DACHash: A Dynamic, Cache-Aware and Concurrent Hash Table on GPUs" (2021). *Electronic Theses and Dissertations*. 2080.
https://egrove.olemiss.edu/etd/2080

This Thesis is brought to you for free and open access by the Graduate School at eGrove. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of eGrove. For more information, please contact egrove@olemiss.edu.

DACHASH: A DYNAMIC, CACHE-AWARE AND CONCURRENT HASH TABLE ON GPUS

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Hao Zhou

May 2021

Copyright Hao Zhou 2021
ALL RIGHTS RESERVED

ABSTRACT

GPU acceleration of hash tables in high-volume transaction applications such as computational geometry and bio-informatics are emerging. Recently, several hash table designs have been proposed on GPUs, but our analysis shows that they still do not adequately factor in several important aspects of a GPU's execution environment, leaving large room for further optimization.

To that end, we present a dynamic, cache-aware, concurrent hash table named DACHash. It is specifically designed to improve memory efficiency and reduce thread divergence on GPUs. We propose several novel techniques including a GPU-friendly data structure, a reorder algorithm, and dynamic thread-data mapping schemes that make the operations of hash table more amendable to a GPU architecture. Testing DACHash on an NVIDIA GTX 3090 achieves a peak performance of *8.65 billion queries/second* in static searching and *5.54 billion operations/second* in concurrent operation execution. It outperforms the state-of-the-art SlabHash by 41.53% and 19.92% respectively. We also verify that our proposed technique improves L2 cache bandwidth and L2 cache hit rate by *9.18*× and *2.68*× respectively.

ACKNOWLEDGEMENTS

Firstly, I would like to express my grateful thanks to my supervisor Dr. Byunghyun Jang for his supports of my graduate study and research. I was encouraged by his hard-working and passion. I also would like to thank Dr. Yixin Chen and Dr. David Troendle for their technical advice. With their help, I was able to make progress in my research.

I would also like to thank Dr. Conrad Cunningham and everyone else who helped me in many ways in the department. I was lucky to study in the department as an undergraduate working with Dr. Cunningham and to start my graduate study in the HEROES lab along with Dr. Jang, Dr. Troendle, and other lab members.

Lastly, I would like to thank my father Zhou Honghai, my mother Han Yumei, my sister Zhou Wen, and my girlfriend Wu Yujie for their selfless mental supports.

Special thanks go to Dr. Byunghyun Jang, Dr. Yixin Chen, Dr. Feng Wang, and Dr. David Troendle for serving on my thesis committee.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vi

INTRODUCTION 1

TECHNICAL BACKGROUND 4

2.1 General-Purpose Computing on Graphics Processing Unit 4

2.2 Hash Table 7

RELATED WORK 9

BASIC DESIGN AND IMPLEMENTATION 11

4.1 Base data structure and organization 11

4.2 Operations supported 12

4.3 Memory stack 14

REORDER 15

DYNAMIC MAPPING SCHEMES 19

PERFORMANCE EVALUATION 23

7.1 Impact of Parameters 24

7.2 Contribution of Reorder Algorithm 27

7.3 Build Rate Comparison 30

7.4 Static Search Comparison 31
7.5 Concurrent Operations Comparison 34

CONCLUSIONS 36

BIBLIOGRAPHY 37

VITA 40

LIST OF FIGURES

4.1 Basic structure of DACHash. Bucket *0* ($\mathbf{B_0}$) has two super nodes and other buckets have one super node. Each super node has a small array of key-value pairs as well as a next pointer. 12

4.2 DACHash uses a concurrent stack to support dynamic memory allocation. All pre-allocated super nodes are pushed into the stack at the beginning. Two threads t_0 and t_1 compete for the same stack *top* index in this example. . . . 14

5.1 Example of poor data reuse. A warp of 4 threads processes keys 0, 3, 7, 8. The shaded nodes indicate the visiting super nodes by the warp. Initial state. 16

7.1 The performance impact of super node size. Expected lengths 0.5 and 1 indicate that there are 1 and 2 super nodes per bucket respectively. Note that when each bucket owns more than one node, dynamic memory allocation is necessary. 25

7.2 The performance impact of the length of combined buckets. The total number of buckets is 2^{19}. 26

7.3 The performance impact of threshold across different settings. The total number of elements is fixed at 2^{22} and the expected length varies. 28

7.4 The performance impact of the proposed reorder algorithm. The total number of elements stored in the table is 2^{22}. 29

7.5 Build rate comparison. The total number of elements inserted in the table is 2^{22}. 31

7.6 Static search comparison. 33

7.7 Concurrent operation comparison. The total number of operations is 2^{22} and node size is 16*4 bytes. Note that we use the dynamic mapping schemes in this experiment. 35

Chapter 1

INTRODUCTION

GPUs have become the platform of choice for many compute and data intensive applications in various fields. Traditionally CPU centric data structures are finding GPU solution. Hash table offering fast data access in near constant time is an important data structure in the fields of computational geometry and bio-informatics, but not well researched. Designing a high-performance hash table on massively multi-threaded GPUs is a challenging task. Tens of thousands of active threads attempting simultaneous hash table access can cause severe performance degradation unless carefully designed. Traditional lock-based implementations suffer from high thread contention [1], leaving non-blocking methods a better choice for the GPU environment [2], [3], [4]. Nonetheless, any approach must accommodate and address the fact that GPUs are very sensitive to memory access patterns and thread divergence [5].

In this thesis, we present a hash table specifically designed and optimized for a GPU architecture. We propose several novel techniques to address two major sources of GPU inefficiency - memory access patterns and thread divergence.

First, we introduce a GPU-friendly chaining structure to support hash collisions. This enables mutability via dynamic memory management for new data to be stored or old data to be deleted, while avoiding the need for repeated rebuilds from scratch. We optimize the chaining structure into a GPU-friendly linked-list of *super nodes*, where each super node

is a small array of key-value pairs. This improves memory access patterns, which is an important design consideration for a GPU's SIMT (Single Instruction Multiple Threads) execution model.

Second, we improve the efficiency of dynamic memory management by pre-allocating a large memory pool and using a concurrent stack to manage memory buffer allocation and deallocation dynamically. This helps reduce the overhead of searching candidates to delete, and the cost of memory allocation and deallocation on GPUs.

Third, we reorder input data elements based on their hash values to improve cache performance. Rather than using expensive traditional sorting, our proposed reorder algorithm efficiently groups operations on the fly, increasing the likelihood of data reuse and coalesced memory transactions. To our knowledge, this is the first attempt to study and improve the locality of hash table data structures on GPUs.

Lastly, we design a novel dynamic mapping scheme that can switch between two different thread-data mapping schemes depending on the shape of hash table: A *one-to-one mapping scheme* maps each thread to a key so that threads process their keys individually; and a *many-to-one mapping scheme* maps each thread to a key, but an entire warp (32 threads) cooperatively processes 32 keys sequentially. Our proposed dynamic mapping scheme automatically switches between these two mapping schemes to achieve better performance.

Our experiments show that on a latest NVIDIA GPU, GTX 3090, our proposed DACHash achieves a static searching throughput and concurrent operations throughput of *8.65 billion queries/second* and *5.54 billion operations/second* respectively. It outperforms the state-of-the-art SlabHash [6] (*7.55 billion queries/second* and *4.41 billion operations/second*). On average, DACHash is 41.53% and 19.92% faster than SlabHash under these two categories. We also profile and verify the cache performance of DACHash using the NVIDIA Visual Profiler. It shows our proposed technique improves L2 cache bandwidth and hit rate by *9.18*× and *2.68*×, demonstrating that the improved cache performance can yield a

significant overall performance boost.

Chapter 2

TECHNICAL BACKGROUND

2.1 General-Purpose Computing on Graphics Processing Unit

Graphics processing unit (GPU) is a specialized electronic circuit made to firstly accelerate computer graphics application. GPUs are virtually in every computer systems including phones, embedded systems and severs. Modern GPUs are very powerful at processing compute and data intensive workloads since their special-designed parallel structure makes them more efficient than CPUs in processing algorithms in parallel.

GPUs employ *Single Instruction, Multiple Threads* (SIMT) execution model where single instruction stream operates on multiple data streams (SIMD) by a group of multiple threads. However, SIMT is still different from SIMD in that instructions are executed by a group of threads in a lock-step fashion. SIMT execution model has been implemented on modern GPUs and is relevant for general-purpose computing on graphics processing units (GPGPU). Essentially, GPUs perform instructions efficiently in a parallel manner but operate at lower frequencies with a large number of cores. Thus, they typically target high throughput.

General-purpose computing on graphics processing units (GPGPU) is a computing

model that uses GPU for non-graphics processing. In other words, GPU processes workloads that are traditionally performed by CPU.

Two main parallel computing platforms are available nowadays, i.e., Compute Unified Device Architecture (CUDA) by NVDIA and Open Computing Language (OpenCL). In this thesis, we use CUDA as our platform.

2.1.1 CUDA

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model that take advantages of the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a traditional CPU [7]. CUDA comes with a software environment where programmers are allowed to use a high-level programming language such as C/C++. On hardware side, CUDA schedules thread blocks onto its available Streaming Multiprocessors (SMs). More specifically, a thread block consists of a certain number of warps where a warp is a set of 32 threads in a thread block. All 32 threads in a warp execute the same instruction in a lock-step fashion.

2.1.1.1 CUDA Functions and Primitives

In the thesis, atomic functions and warp-level primitives [7] are leveraged to solve issues in hash tables.

1. Atomic functions perform a read-modify-write atomic operation on one 32-bit or 64-bit word stored in global or shared memory.

    - *atomicAdd(address, val)*: reads the *old* value located at the *address* in global or shared memory, computes (*old* + *val*), and stores the result back to memory at the *address*. All three operations are performed in one atomic transaction. The function will return *old* value.

- *atomicSub(address, val)*: reads the *old* value located at the *address* in global or shared memory, computes (*old* - *val*), and stores the result back to memory at the *address*. All three operations are performed in one atomic transaction. The function will return *old* value.
- *atomicExch(address, val)*: reads the *old* value located at the *address* in global or shared memory and stores *val* back to memory at the *address*. Two operations are performed in one atomic transaction. The function returns *old* value.
- *atomicCAS(address, compare, val)*: reads the *old* value located at the *address* in global or shared memory, computes ($old == compare\ ?\ val\ :\ old$) , and stores the result back to memory at the *address*. These three operations are performed in one atomic transaction. The function returns *old* (Compare And Swap) value.

2. NVIDIA GPUs execute warps of 32 parallel threads using SIMT where each thread accesses its own registers, loads and stores from memory addresses, and follows divergent control flow paths. The CUDA compiler and the GPU hardware work together (explicitly or implicitly) to ensure all threads of a warp execute the same instructions as frequently as possible to minimize thread divergence and maximize execution performance.
   - *shfl(data, lane)*: moves *data* at the $lane^{th}$ of the warp to other threads in the same warp. It is a fast mechanism for simultaneously exchanging data between threads in the same warp without use of shared memory.
   - *ballot(predicate)*: evaluates *predicate* for all non-exited threads in a warp and returns an integer whose N^{th} bit is set if and only if *predicate* evaluates to non-zero for the N^{th} thread of the warp and the N^{th} thread is active. It is a fast voting mechanism in CUDA.

2.2 Hash Table

A hash table is an efficient data structure that maps keys to values. A hash table uses a hash function $h()$ to compute an index, also called a *hash value*, into a list of buckets, where the wanted value will be found. In general, the key k is hashed by $h()$ and the resulting hash value tells where the corresponding value is stored.

Ideally, the hash function generates unique hash values for different keys, also called perfect hash function, but most of hash table designs adopt a non-perfect hash function design, which causes *collisions* where hash function generates the same hash value for more than one keys, that is $h(k_1) = h(k_2)$, where k_1 is not equal to k_2. It is known that *collisions* could be solved in multiple ways, such as open addressing and separate chaining. Another well-known issue for hash tables is dynamic resizing, where either all entries (key-value pairs) are re-hashed into another hash table with larger size or dynamic allocation is in play.

A well-designed hash table is efficient in terms of lookup (or search) since each lookup is independent of the number of key-value pairs stored in the hash table. Usually, the average cost for searches is $O(1)$, which is much efficient compared to search trees $O(logN)$ and array structures $O(N)$. For this reason, hash tables are widely used in various applications such as computer graphics and database applications.

The prposed DACHash in this thesis uses separate chaining and dynamic allocation techniques to support hash *collisions* and dynamic resizing, respectively.

2.2.1 Concurrent Hash Table

A concurrent hash table is a hash table that allows concurrent accesses by multiple threads. Contention therefore becomes an issue in such hash tables. More specically, concurrent hash tables suffer from a variety of contention problems such as *ABA problem*, *race conditions*, and *deadlocks*. Accordingly, there is a variety of ways to solve contention problems such as atomic instructions, locking, and etc.

2.2.2 Challenges on GPUs

There are two major challenges when implementing a hash table on GPUs - memory access pattern and thread divergence.

2.2.2.1 Memory Access Pattern

A traditional hash table presents a sparse data (key-value pairs) storage. The property of this kind of hash tables makes the hash table inefficient on GPUs. For instance, when GPU threads search keys in the hash table, threads could be mapped to different locations of the hash table due to the sparse data storage, which can decrease the GPU's SIMT throughput of memory accesses since adjacent threads access sparse memory location which results in non-coalesced memory accesses.

2.2.2.2 Thread Divergence

Some hash tables support different operations such as search, update, delete, and insert, where times to finish these operations are different. In this case, when implementing such hash tables on GPUs, thread divergence cannot be ignored because CUDA schedules every 32 threads into a warp which execute instructions in a lock-step fashion. If some of threads in a warp take longer time to execute instructions, the other threads in the same warp have to wait until all 32 threads converge.

Chapter 3

RELATED WORK

Several hash table designs and implementations have recently been reported for GPUs in the literature.

Alcantara et al. [8] built a hash table on GPUs, which performs parallel insertions and retrievals. Their work is based on Cuckoo Hashing [9] and relies on atomic operations during multi-threads table construction. The authors use a set of hash functions to find a key in multiple candidate locations for insertion as Cuckoo Hashing does. Evicted keys need to be inserted into another location until no more evicted keys exist. A careful design of a set of hash functions is required since hash functions determine the frequency of rebuilding from scratch. The order of hash functions also matters.

Garcia et al. [10] presented a parallel hashing method where their hashing could reach high load factor but with a low rebuilding failure rate. The authors designed a coherent hash function to leverage coherence in memory and further increase locality in memory. In addition, coherent hashing also makes groups of threads execute consistent paths.

Khorasani et al. [11] proposed a hashing method called Stadium Hashing (Stash) and Stash with collaborative lanes (clStash). Stash Hashing has two basic structures: a table for keeping all keys and values, and a compact auxiliary structure called a ticket-board to maintain a ticket (consists of the availability bit and the info bits) for every bucket in the table.

The availability bit determines if the bucket is occupied and the info bits store information of the key. This design reduces unnecessary accesses to the actual table content according to the availability bit and the info bits, which speeds up retrievals. By solving collisions via double-hashing (primary and secondary hash functions), Stash allows concurrent execution of mixed insertions and retrievals. The secondary hash function generates a step size that could hurt the memory performance on GPUs. clStash improves warp execution efficiency by redistributing tasks to early-finished threads in a warp.

SlabHash [6] proposes further improvements to the efficiency of warp execution and memory coalescing. The authors proposed a warp-cooperative work-sharing (WCWS) strategy, where all threads in a warp process one operation at a time by utilizing warp-synchronous programming and warp-wide communications. This design presents less thread divergence when compared to other hash tables. The authors also take advantage of array and linked-list structures to further serve their WCWS strategy. The SlabHash designs slabs which are arrays with key-value pairs stored. Each slab has the size of 128 bytes that matches the size of a cache line on GPUs. The SlabHash also designs a specialized memory pool to implement dynamic allocation.

Gao et al. [12] adopted a structure similar to SlabHash. In addition, the paper discusses the throughput of the WCWS strategy. They show that when the number of elements stored in the table is large, it could achieve higher throughput. Otherwise, the throughput is relatively low. Gao et al. solves the problem by proposing an adaptive model. In addition, the authors present a reader-writer lock based synchronization and bucket-level synchronization to ensure atomicity of hash operations (individual hash operations or groups of hash operations).

WarpCore [13] proposed a fast hash table on GPUs. They proposes a memory-compact bucket list to support flexible multi-value storage, a hashing scheme to improve global memory access patterns by leveraging CUDA cooperative groups, and an efficient techniques to support multi-GPUs.

Chapter 4

BASIC DESIGN AND IMPLEMENTATION

In this chapter, we introduce the basic design and implementation of DACHash, including base data structure, organization, supported operations, and memory management.

4.1 Base data structure and organization

Each DACHash bucket is designed as a linked-list of small arrays consisting of key-value pairs as shown in Fig. 4.1. The array offers contiguous, linear memory access patterns, while a linked-list chain offers easy, concurrent modification.

In our design, each node in the linked-list chain holds multiple interleaved key-value pairs. We call these nodes *super nodes*. The first super node connected to a bucket head is pre-allocated. Subsequent super nodes are dynamically allocated or deallocated at run time as needed.

The base data structure and organization of DACHash offer several optimization opportunities. First, the combined array/linked-list structure enables a natural way to support collisions. Second, a chaining technique allows dynamic allocation, instead of needing to

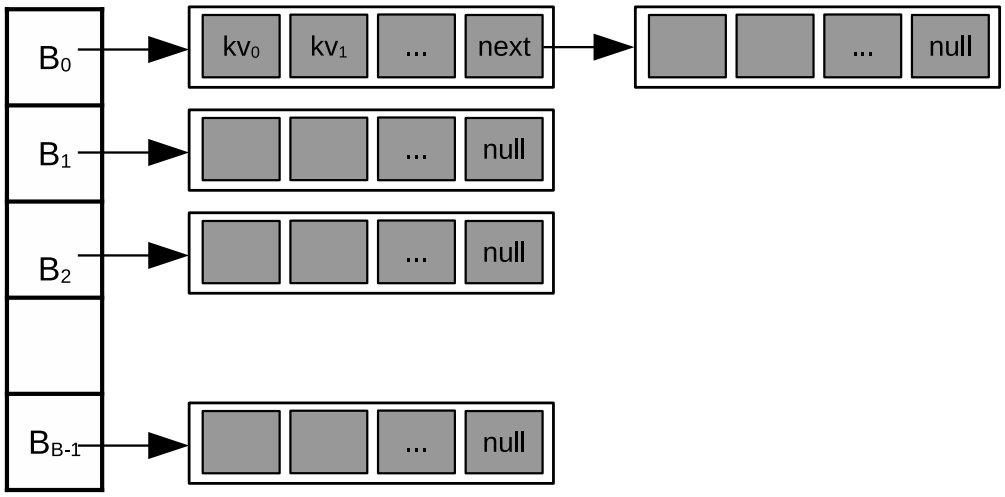


Figure 4.1. Basic structure of DACHash. Bucket *0* ($\mathbf{B_0}$) has two super nodes and other buckets have one super node. Each super node has a small array of key-value pairs as well as a next pointer.

rebuild the hash table from scratch. Third, array structures achieve GPU-friendly memory access patterns compared to a linked-list's scattered memory accesses. Fourth, super nodes offer flexible thread-data mapping scheme options, e.g., *one-to-one* or *many-to-one mapping* schemes.

4.2 Operations supported

We implement five basic hash table operations on unique keys. CUDA atomic funtions ensure correctness. Note that, although not implemented, duplicate keys can be accommodated without significant design changes. The supported operations are:

Search is responsible for finding a key in the hash table and returning its value. If no key is found, it returns null. The operation starts by hashing a key to a bucket. Searching begins at the bucket's first super node. If no key matches, it continues traversing the bucket's super nodes until a matching key is either found or it reaches the end of the bucket list.

Insert adds a key-value pair to the hash table. Since keys must be unique, we must first ensure the key exists in the hash table. If it does, the operation acts as *update*, replacing

the old value with a new value. If it does not exist, it is inserted into the hash table. A new super node may be dynamically allocated if needed. When inserting a new key-value pair into the table, it first looks for an empty slot in the first super node of the bucket. If an empty slot exists, an *atomicCAS()* (a CUDA atomic function) ensures a correct insertion. If the first super node is full, the thread traverses the super node list until it finds an empty slot. If it reaches the last node in the bucket, the thread dynamically allocates a new super node and connects the new node after the bucket's last super node using an *atomicCAS()*. Although multiple threads may try to connect their super nodes after the last super node simultaneously, only one thread will succeed. The failing threads deallocate their nodes and retry until they succeed. Once successful, the threads redo their *insert* operation using the bucket's new last super node.

Update finds the key to update its value. If the key is found, it replaces its value with the new value using an *atomicExch()*. Otherwise, the new pair is inserted.

Delete is similar to the *search* operation, but returns no value. It starts its traversal at the first super node in the bucket the key maps to. If found, it marks the key as logically *deleted*. If not, it continues traversing super nodes looking for the matching key until it reaches the last super node in the bucket. This operation does not deallocate empty super node. Deallocation is done by the clean operation (see below).

Clean compacts the bucket's super node linked-list, ensuring only the last super node has any empty slots. We implement the *clean* operation as a separate kernel, so no other operations interfere when cleaning the hash table. Deallocated super nodes are pushed back to our memory stack for later use. The *clean* operation is only required when the memory stack is empty.

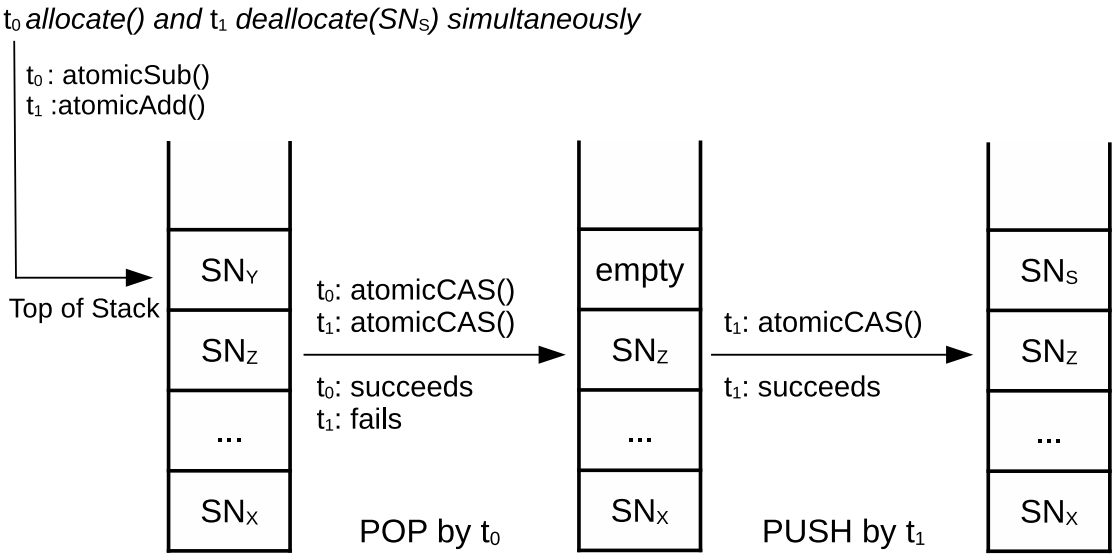


Figure 4.2. DACHash uses a concurrent stack to support dynamic memory allocation. All pre-allocated super nodes are pushed into the stack at the beginning. Two threads t_0 and t_1 compete for the same stack *top* index in this example.

4.3 Memory stack

The *insert*, *update*, *delete* and *clean* operations may require dynamic super node allocation or deallocation. To support this, we pre-allocate a large number of super nodes and place them on a concurrent stack. A pop allocates a super node and a push deallocates a super node concurrently as shown in Fig. 4.2. This is a simple, fast, GPU friendly alternative to a CPU-side *malloc()* or *free()*.

Chapter 5

REORDER

Input keys are hashed to different buckets. When they are mapped to threads, a warp suffers from poor locality because the super nodes within and across buckets are likely scattered in memory. Memory requests from threads in a warp are highly likely to reside in different cache lines (uncoalesced) rather than a single line (coalesced). Such poor spatial locality causes multiple memory transactions, which in turn, significantly increases memory traffic. SlabHash [6] proposed a work-sharing strategy within warps to increase coalesced memory accesses. However, it still suffers repeated linked-list traversals, which is another source of poor locality. Suppose a warp processes a list of query keys {*0, 3, 7, 8*} and the size of warp is 4, as visualized in Fig. 5.1. The warp processes *key* 0, finds the key in *bucket* 0, and loads the first super node. With this access, other keys stored in that super node are loaded together but not used. The same issue occurs for *key* 3 and *key* 7. In the mean time, the super node loaded for *key* 0 may have been evicted from the cache. When processing *key* 8, the warp starts over from the beginning, and loads the first super node of *bucket* 0, then it loads the second super node that contains *key* 8. In this case, the first super node is loaded twice. Even if the warp issues coalesced memory requests, it may not end up being an efficient use of data as it can still cause extra global memory transactions and waste potential use of other keys in the loaded super nodes.

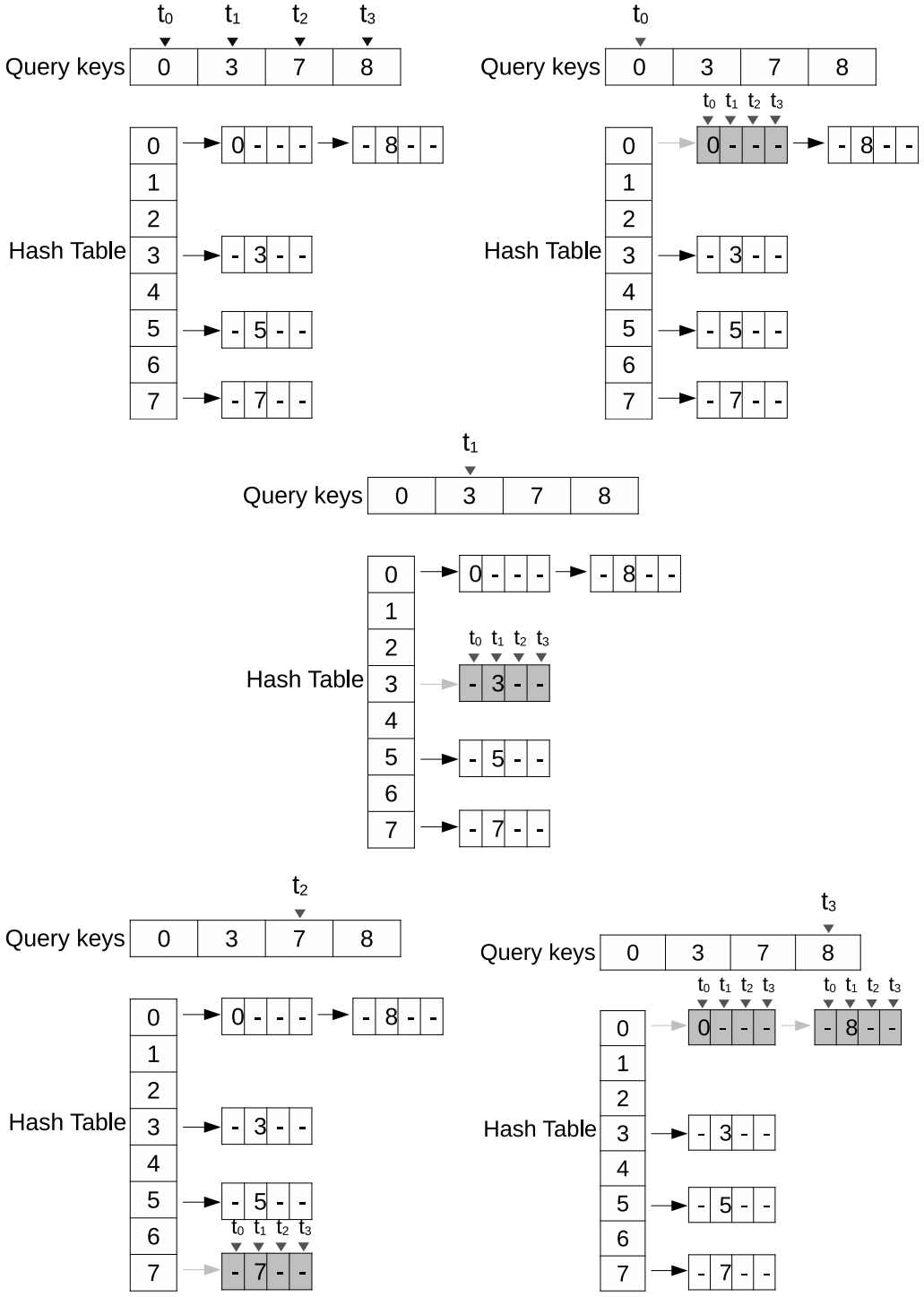


Figure 5.1. Example of poor data reuse. A warp of 4 threads processes keys 0, 3, 7, 8. The shaded nodes indicate the visiting super nodes by the warp. Initial state.

We observed that such inefficiencies are caused by the randomly arranged input data list. Reordering can improve the data locality and cache performance (i.e., hit rate and bandwidth). Sorting is the traditional way of reorganizing data. However, not only is a sort expensive on GPUs, but also a strictly ordered sort is not required for our purpose. We need only group the keys with the same hash value together. To this end, we propose a *reorder* algorithm. In our reorder algorithm, we take advantage of a pre-allocated memory buffer on GPUs to partition data according to their hash values. Keys with the same hash value compute their indexes in this pre-allocated memory buffer so that keys with the same hash value are physically close and adjacent to each other in memory. By doing so, when each thread claims its key, the adjacent threads are more likely mapped to the same bucket. In this way, when traversing bucket's super nodes, threads in a warp probably request the same or nearby super nodes, resulting in fewer memory transactions. Compared to scattered memory requests, the total number of global memory transactions can decrease significantly. Since the input keys are partitioned based on their hash values, when warps process these keys, the same or nearby super nodes are likely loaded from the cache directly, so that the cache performance increases as well.

Algorithm 1 details our proposed reorder algorithm. It utilizes a pre-allocated GPU memory buffer named *ReorderSpace* with the same or larger size than *keysList* but with an additional dimension. Note that the size of *ReorderSpace* is $B * M$, where B is the number of buckets and M is calculated by $\lceil N/B \rceil$ (line 1-4) where N is the total number of keys in *keysList*, and the size of *Record* is B in order to keep track of the latest index of buckets. The row index is defined by bucket (hash) value, and an *atomicAdd()* is used to find the corresponding index of that key in that row (line 8). By doing so, keys with the same hash value will be mapped to the same row in *ReorderSpace*. However, for our input keys, we cannot ensure that keys have a perfect uniform distribution across buckets. In this case, we have to arrange keys with the indexes that are greater than M to a nearby location (line 9-11). It checks whether the next row has an empty spot to add. If so, it updates *index*. If

not, check the row after the next row until an empty spot is found. Lastly, we add keys to *ReorderSpace* (line 12). Since the size of *ReorderSpace* is equal to or greater than the size of *keysList*, it guarantees all keys will have a spot to be added.

However, the reorder algorithm delays operations until the reorder process completes. This causes a synchronization problem. To this end, we utilize host-side device-synchronization via CUDA API. In order to speed up the reorder algorithm, we also improve the algorithm. Instead of grouping keys with the same hash value, we group keys with nearby hash values. Originally, B is equal to the total number of buckets in our hash table. By decreasing B (line 3), we can group the keys with nearby hash values into one row. For instance, if we decrease B to $B/4$, every four hash values will be grouped together, e.g., *0*, *1*, *2* and *3* (*4* values). In *keysList*, the probability of adjacent data elements that can be mapped to the same row of *ReorderSpace* increases, so the throughput of reorder is further improved.

Algorithm 1: Pseudocode for the proposed reorder algorithm.

```
   input: ReorderSpace
          keysList
          Record
 1 N = lengthOf(keysList);
 2 B = totalNumberOfBucket();
 3 // B = totalNumberOfBucket() / lengthOfCombinedBuckets;
 4 M = ⌈N/B⌉;
 5 key = keysList[threadId];
 6 bucket = hash(key);
 7 // bucket = hash(key) / lengthOfCombinedBuckets;
 8 index = atomicAdd(Record[bucket], 1);
 9 while index >= M do
10 |   index = atomicAdd(Record[(++bucket) mod B], 1);
11 end
12 ReorderSpace[bucket][index] = key;
```

Chapter 6

DYNAMIC MAPPING SCHEMES

There are two common thread-data mapping schemes used in hash table design: 1) a *one-to-one mapping* scheme, which maps each thread to a single key so that threads can process their 32 keys individually. Note that threads in a warp could possibly exit at different times due to the different operations they may have. The threads tend to have scattered memory access patterns when reading from or writing to GPU memory; and 2) a *many-to-one mapping* scheme, which maps a warp to 32 keys but 32 threads in a warp work and communicate with each other to cooperatively process a single key at a time. Threads in a warp will converge at the same time because all 32 threads in the warp will execute the same low-level instructions and the memory access pattern is likely to be coalesced. In general, each scheme shows different performance characteristics. The *one-to-one mapping* scheme outperforms when thread divergence is not an issue (e.g., threads in a warp process same type of operations), while the *many-to-one mapping* scheme is a better choice when execution efficiency matters (e.g., threads within a warp process different types of operations).

In order to better utilize two different schemes, we propose our dynamic mapping schemes where DACHash selects one scheme over the other based on, so-called *expected length*. The expected length ϵ, is defined as the average number of super nodes per buckets. Therefore the selection of the proposed dynamic mapping schemes D, is expressed as the

following.

$$D = \begin{cases} O & \text{if } \epsilon < \tau \\ M & \text{otherwise} \end{cases}$$

where O is *one-to-one mapping* scheme, M is *many-to-one mapping* scheme, and τ is a threshold.

Algorithm 2: Pseudocode for *one-to-one mapping* scheme.

```
1  key = keysList[threadId];
2  value = valuesList[threadId];
3  bucket = hash(key);
4  superNode = getNextSuperNode(bucket);
5  do
6  |  for i ← 0 to NODE_SIZE-1 do
7  |  |  if superNode[i].key == key then
8  |  |  |  SEARCH();
9  |  |  |  or DELETE();
10 |  |  |  or UPDATE();
11 |  |  end
12 |  end
13 |  superNode = getNextSuperNode(superNode);
14 while superNode != nullptr;
15 if UPDATE() failed && isUPDATE then
16 |  INSERT();
17 end
```

Our implementations of these mapping schemes are shown in Algorithm 2 and 3 respectively. In the *one-to-one mapping* scheme, groups of threads may have different operations to perform and their finishing times are likely to be different, resulting in low execution efficiency. The many-to-one mapping schemes partitions thread blocks into tiles and the size of tile is specified by *CG_SIZE* which can be {*1, 2, 4, 8, 16, 32*}. It enables threads communication by using CUDA primitives such as *shfl()* and *ballot()*, and lastly tiles would perform different hash operations, i.e., *search*, *delete*, *update*, and *insert*. In the *many-to-one mapping* scheme, tiles are more likely to execute the same low-level instructions so that thread

Algorithm 3: Pseudocode for *many-to-one mapping* scheme.

```
1  key = keysList[threadId];
2  value = valuesList[threadId];
3  bucket = hash(key);
4  operation = operationsList[threadId];
5  superNode = getNextSuperNode(bucket);
6  tile = tiled_partition⟨CG_SIZE⟩(this_thread_block()); laneId = tile.thread_rank();
7  for lane ← 0 to CG_SIZE-1 do
8      // a tile processes operations from 0 to CG_SIZE-1;
9      share data in the lane^th of tile by tile.shfl();
10     do
11         tile reads superNode and finds target key by tile.ballot();
12         if found then
13             SEARCH();
14             or DELETE();
15             or UPDATE();
16         else
17             share the next superNode by tile.shfl();
18         end
19     while superNode != nullptr;
20     if UPDATE() failed && isUPDATE then
21         INSERT();
22     end
23 end
```

divergence is minimized, and the memory access pattern is also improved.

Chapter 7

PERFORMANCE EVALUATION

We evaluated our proposed DACHash on an NVIDIA GTX 3090, which is based on the *AMPERE* microarchitecture [14] with compute capability of 8.6, 10496 CUDA cores, and 24 GB off-chip global memory. We compiled our code with the CUDA 11.2 compiler. We divide our performance evaluation into three parts. First, we show the performance impact of various parameter values for the super node size, the length of combined buckets in our proposed reorder algorithm, and the threshold τ for dynamic mapping schemes. Second, we demonstrate the effectiveness of our reorder algorithm. Lastly, we compare our results using the state-of-the-art SlabHash [6] as a baseline for different categories, such as build rate, static search throughput, and concurrent operation throughput. In our experiments, we assume all keys are unique.

7.1 Impact of Parameters

7.1.1 Super Node Size

We measured the impact of the number of elements per super node[1] on performance while keeping the total input keys size fixed. We built a hash table with randomly generated keys and different super node sizes. Then we created the shuffled query list with the same keys existing in the hash table to perform *search* operations. We kept our reorder algorithm enabled for this experiment. Fig. 7.1 shows the results of super node sizes of 4*4, 8*4, 16*4, and 32*4 bytes.

Our experiment results show that super node size significantly affects the performance of our hash table. When super node size is small, the traversal speed is faster as each thread in the operation may need to traverse the entire bucket to find its target. When a bucket holds more than one super node, the performance impact of super node size decreases relatively due to dynamic memory allocation overhead and the bucket traversal time. In the following experiments, we choose 16*4 bytes to be our super node size for the following reasons. First, the way that we optimize the reorder algorithm (e.g., combining keys with nearby hash values) leaves smaller super node more reasonable. Suppose that the size of the super node is 32*4 bytes (e.g., the size of a cache line), when some threads in a warp load a super node, other threads are not likely to be mapped to the same super node, which results in more global loads. With a smaller super node (e.g., 16*4 bytes), the reorder algorithm decreases the repeated global loads because threads in a warp may load their super nodes directly from the cache. Second, smaller super nodes (e.g., 4*4 or 8*4 bytes) may trigger more dynamic allocations as the number of total key-value pairs stored in the hash table increases, which decreases the performance.

[1]Note that the choices of super node sizes are guided by CUDA cooperative groups [7] where cooperative groups only allow finer granularity at level {1, 2, 4, 8 ,16, 32}. Expected length indicates how many super nodes exist per bucket.

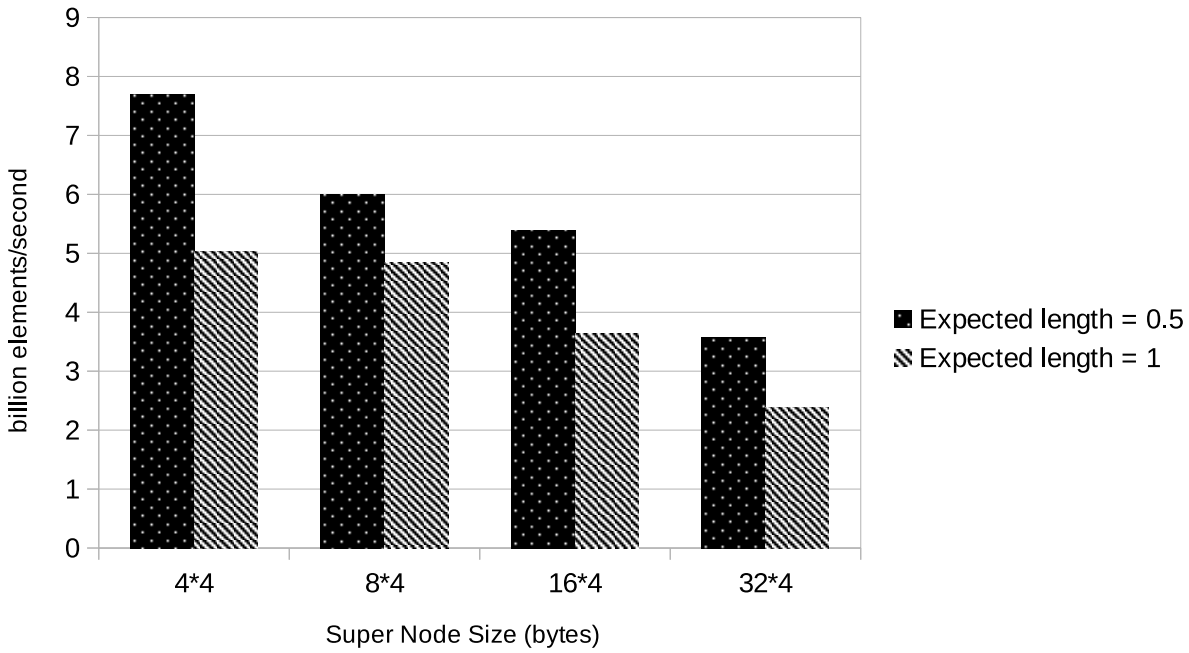


Figure 7.1. The performance impact of super node size. Expected lengths 0.5 and 1 indicate that there are 1 and 2 super nodes per bucket respectively. Note that when each bucket owns more than one node, dynamic memory allocation is necessary.

7.1.2 Length of Combined Buckets in the Reorder Algorithm

The objective of our reorder algorithm is to combine nearby buckets and partition data elements based on those combined buckets (line 7 in Algorithm 1). We measured the efficiency of our reorder algorithm on various lengths of combined buckets using *search* operations. Fig. 7.2 shows the experimental results. When the reorder algorithm is disabled (i.e., length of combined buckets is 0), we form a baseline for our *search* operation where the keys in the query list are randomly organized. Note that the total time consists of two components - the search and the reorder time. As we increase the number of combined buckets, the total time decreases because our reorder algorithm makes searching more cache-friendly. However, from the figure one can see that when too few or too many buckets are combined, the efficiency of our reorder algorithm diminishes. We observed two reasons for that. When there are too few combined buckets, the reorder algorithm suffers from poor cache performance as analyzed in Chapter 5. The input query list for reorder resides in contiguous memory but

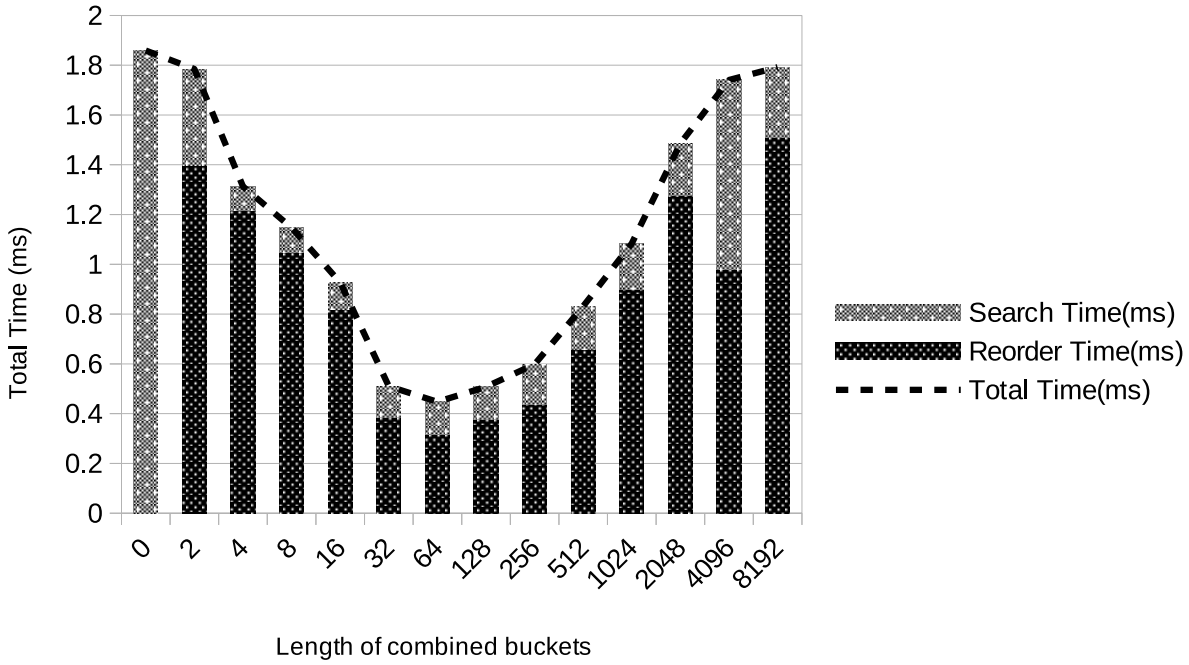


Figure 7.2. The performance impact of the length of combined buckets. The total number of buckets is 2^{19}.

the data elements are randomly located. So, every time a cooperative group or a warp reads memory (e.g., 128 bytes), it achieves coalesced memory access. But when writing to different buckets (line 12 in Algorithm 1), the threads in the same warp or group are more likely to write to different cache lines, which results in an inefficient memory access pattern. When there are too many combined buckets, contention becomes an issue even though writing to memory could be efficient. This causes many threads to modify the same memory unit (line 8 and line 10 in Algorithm 1). In Fig. 7.2, when the number of combined buckets is roughly between 32 and 256, our hash table shows best performance with the total 2^{19} buckets. In general, we find the ratio between total buckets and the number of combined buckets, 1/256 (i.e., combining 256 buckets for the different number of total buckets), is a good choice.

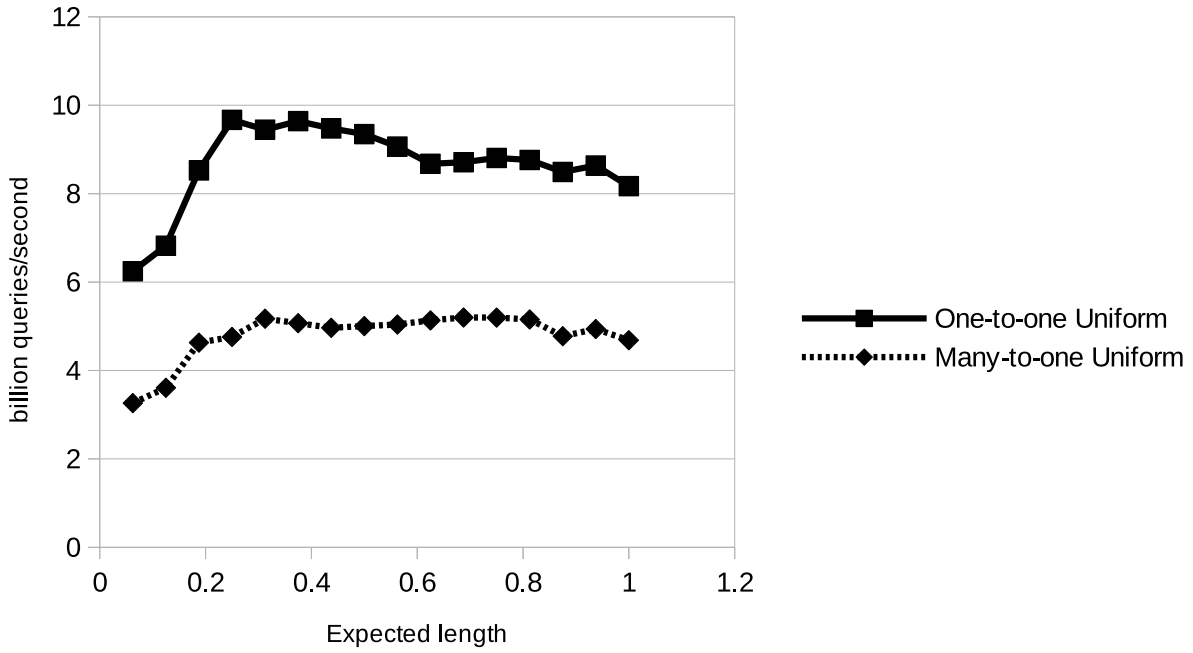
7.1.3 Threshold for Dynamic Mapping Schemes

We conducted this experiment to determine when we should switch between the *one-to-one mapping* and the *many-to-one mapping* schemes. Note that the *many-to-one mapping* scheme is designed to solve the thread divergence issue. So in order to better understand the difference of two schemes on the thread divergence issue, we conducted the experiment on two different settings. In the uniform operation setting (e.g., all searches), Fig. 7.3a shows that the *one-to-one mapping* and *many-to-one mapping* schemes have no performance crossover point, which indicates that *one-to-one mapping* scheme consistently outperforms when all threads perform the same operation. In the mixed operation setting (e.g., searches and updates) of Fig. 7.3b, however, we see a performance crossover point roughly at an expected length of 0.6. We believe that before the crossover point, the *one-to-one mapping* scheme wins due to its higher throughput, and after the crossover point, the *many-to-one mapping* scheme presents less thread divergence by having threads in warps work together to finish an operation at a time so that the threads possibly converge at the same time. Based on our experiment, we chose an expected length of 0.6 as the value of the threshold τ for our dynamic mapping schemes. Note that the dynamic mapping schemes go in effect only when a setting of the mixed operations is required.

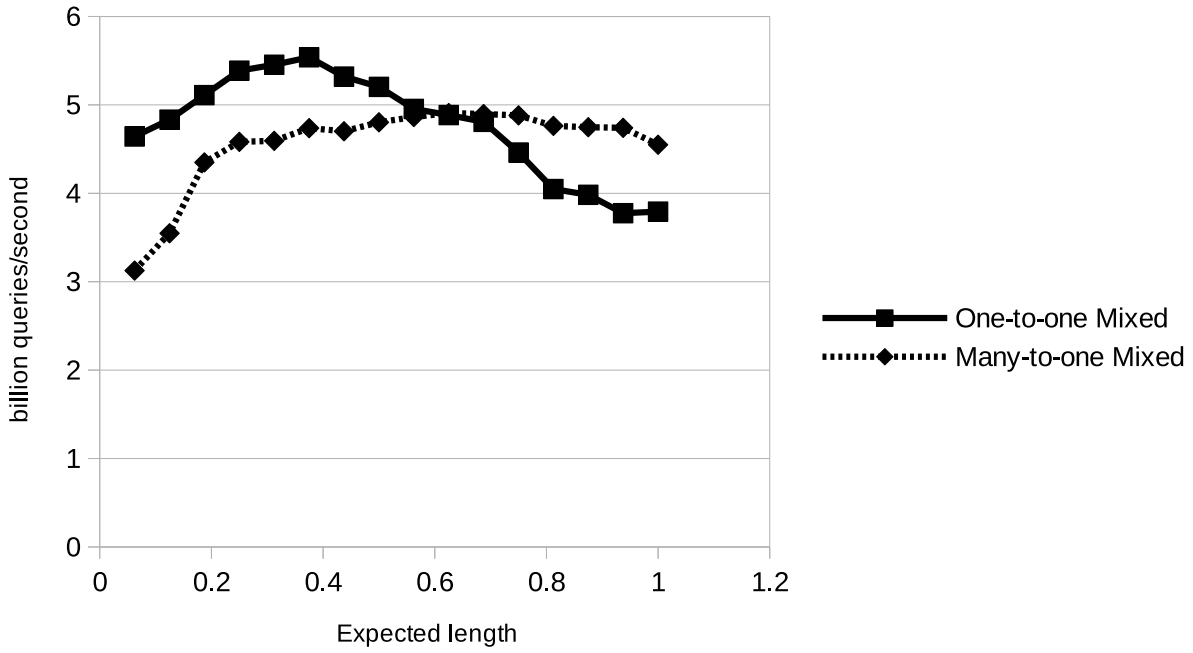
7.2 Contribution of Reorder Algorithm

We performed experiments to show the effectiveness of our proposed reorder algorithm by enabling and disabling it, and measuring changes in cache hit rate and cache bandwidth using the performance counters provided in the NVIDIA Visual Profiler [7][2]. When disabled, the keys in the input query list are randomly arranged. As shown in Fig. 7.4, reorder enabled always outperforms reorder disabled across different expected lengths. The average speedup is around *4.33*×.

[2]Note that the data is retrieved from NVIDIA GTX 1070 since either the visual profiler or the *nvprof* doesn't support GTX 3090 yet with a compute capability of 8.6 that is higher than that of 7.0.



(a) Uniform operations



(b) Mixed operation

Figure 7.3. The performance impact of threshold across different settings. The total number of elements is fixed at 2^{22} and the expected length varies.

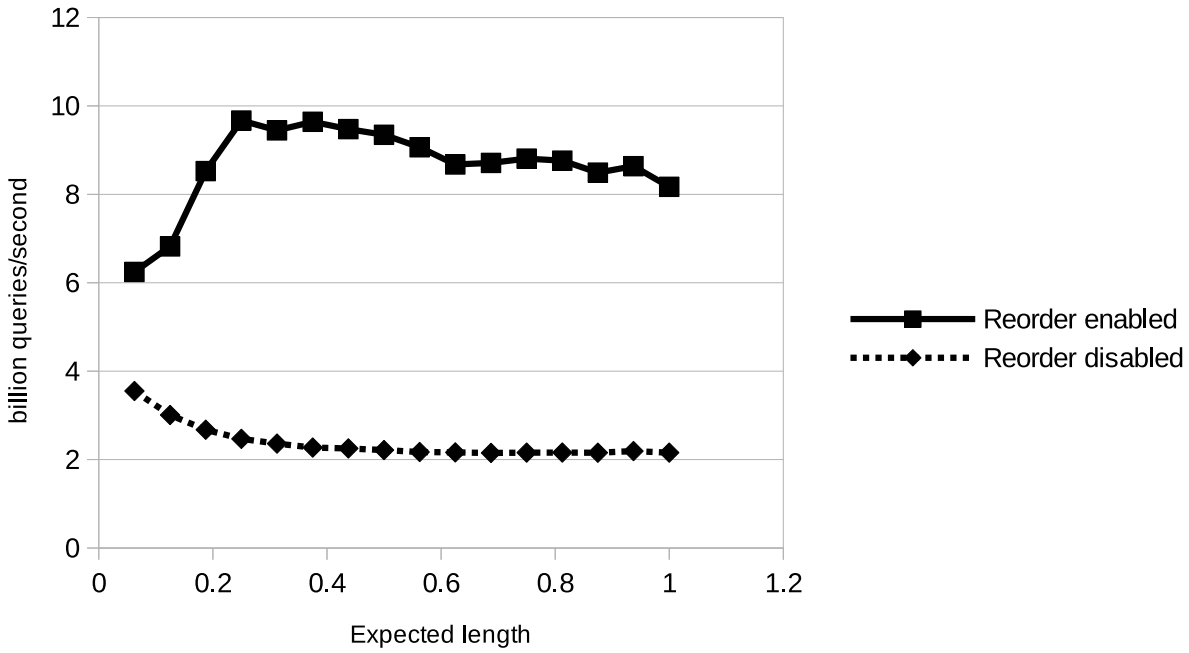


Figure 7.4. The performance impact of the proposed reorder algorithm. The total number of elements stored in the table is 2^{22}.

7.2.1 Effectiveness of Reorder

We use *search* operations in this experiment. When the reorder algorithm is disabled, the keys in the input query list remain unchanged. As shown in Fig. 7.4, given different expected lengths, enabling the reorder algorithm always gives better performance. The average speed up is around *4.33*×. It is noteworthy that our reorder algorithm achieves small speed ups only when the expected length is short. This is because when it is short (e.g., less than 0.2), the number of buckets is too many, the total number of data that each row can hold is small (i.e., $M = \lceil N/B \rceil$ in Algorithm 1). In this case, since the key distribution across buckets is not perfectly uniform, the reorder algorithm has to find available locations for some keys in other rows, which decreases the performance of the reorder algorithm.

7.2.2 Cache Performance

We also conducted a search experiment to measure the effectiveness of our reorder algorithm on L2 cache performance (data provided by the NVIDIA Visual Profiler) under different situations. From Table 7.1, one can see with reordering enabled, our proposed DACHash boosts L2 cache bandwidth by *9.18*×, and L2 cache hit rate by *2.68*×. In short, as mentioned early in Chapter 5, the effect of memory operations on GPU performance cannot be ignored. By applying our proposed reorder algorithm, we can improve data locality because adjacent threads will possibly map to the same or nearby buckets. Therefore, the overall performance is improved by increasing L2 cache bandwidth and hit rate.

	Reorder enabled	Reorder disabled
L2 Cache Bandwidth	901.25 GB/s	98.196 GB/s
L2 Cache Hit Rate	91.00 %	34.00 %

Table 7.1. L2 cache performance comparison.

7.3 Build Rate Comparison

We compared the build rate of DACHash to those of SlabHash. As shown in Fig. 7.5, our proposed DACHash performs better than the state-of-the-art SlabHash when the expected length roughly ranges from 0.1 to 0.6. The peak building rate of DACHash is *3.42 billion elements/second*, compared to *2.9 billion elements/second* in SlabHash with the total number of buckets of 2^{22}. However, we noticed that our hash table suffers when the expected length is very low (e.g., 0.0625). We think this is due to the limitation of the reorder algorithm as we mentioned in Section 7.2.1. The reorder algorithm has to update indexes for some data in other rows. Also, our build rates are lower when expected length is high (e.g., 0.6). We believe there are two reasons for it. The first reason is that when we have larger expected length, we also expect the time for traversing the super nodes to find empty spots longer. Increased traversal times decrease the overall build rate. The second reason is that when

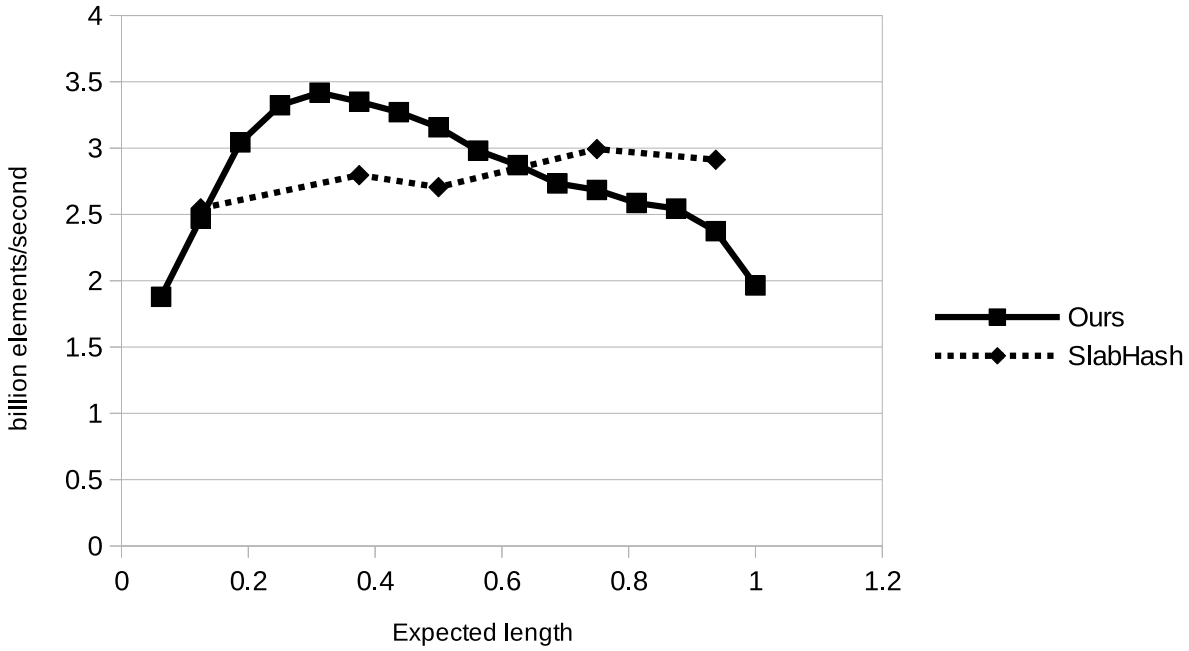


Figure 7.5. Build rate comparison. The total number of elements inserted in the table is 2^{22}.

we have fewer buckets, the contention on the same memory unit (line 8 in Algorithm 1) intensifies so that the overall throughput decreases.

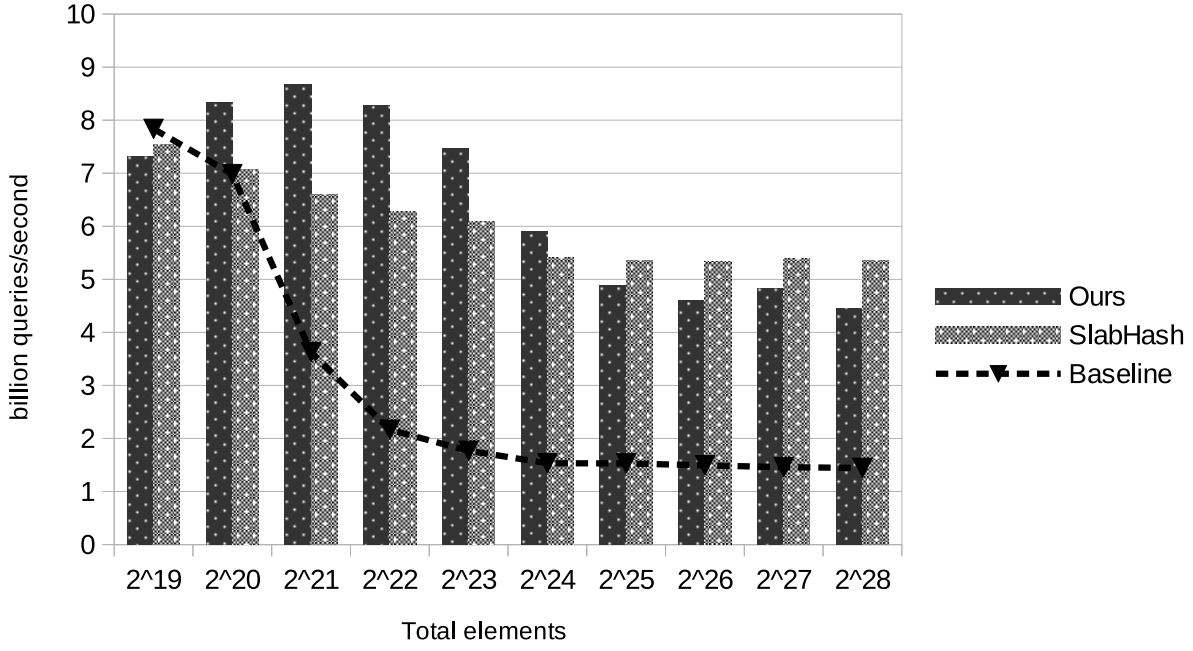
7.4 Static Search Comparison

For static search comparison, we have two different setups as shown in Fig. 7.6. For both setups, we first build a hash table with randomly generated key-value pairs, then create a query list with the same keys but re-arranged to perform *search* operations. The total number of elements varies in Fig. 7.6a, while the expected length varies in Fig. 7.6b.

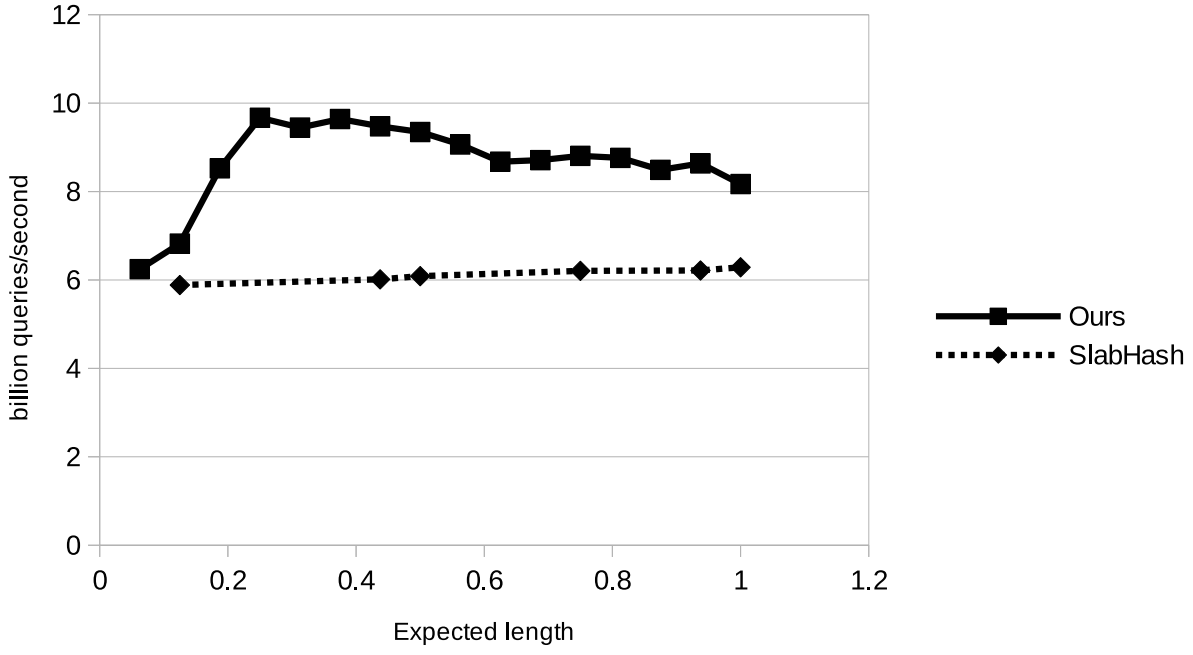
Fig. 7.6a shows the result of DACHash compared to SlabHash when the total number of queries varies. The peak performance of DACHash is *8.66 billion elements/second*, while the peak performance of SlabHash is only *7.55 billion elements/second*. On average, DACHash is improved by *7.1%* compared to SlabHash. It is noteworthy that DACHash under-performs when the total number of elements is small or large while the expected

length is fixed (e.g., 0.5). Recall that we synchronize the reorder algorithm by adopting a host-side CUDA API. So when the total number of elements is small (e.g., 2^{19}), our reorder algorithm suffers from the extra kernel launch and its extra synchronization time, which increases the overall finishing time. When the total elements ranges from 2^{25} to 2^{28}, our reorder algorithm suffers from the situation where writing to different buckets in the reorder algorithm results in an inefficient memory access pattern. Also, one can see that DACHash outperforms SlabHash when the total elements ranges from 2^{20} to 2^{24}. The reason is that our reorder algorithm indeed finds a balance point at which the benefits of reordering outweigh its drawbacks. In addition, one may notice that in Fig. 7.6a, we have an extra base line. This is a baseline measuring the performance of purely *one-to-one mapping* scheme without any other techniques involved. Note that when the total number is small (e.g., 2^{19}), either DACHash or SlabHash under-performs the baseline. We think this is due to the GPU characteristics of latency hiding. When the total number of elements is small, GPUs actually hide memory latency fairly well. This also implies why we prefer to design a reorder algorithm instead of using a sorting algorithm directly because sorting may perform well on a small data scale but will suffer on a large data scale.

Fig. 7.6b also demonstrates the searching performance where the expected length varies. The experiment shows that DACHash outperforms SlabHash and achieves a searching throughput above *8.65 billion elements/second* and improves SlabHash by *41.53%* on average. However, there are several observations: First, the performance of our hash table decreases when the expected length increases. This is because, in this experiment, we rely on the *one-to-one mapping* scheme so that when the expected length increases, the bucket traversal time to find the matching key also increases, and the contention on the same memory unit will increase as well along with the increased expected length. Second, when the expected length is small (e.g., <0.1), SlabHash could perform better than DACHash. We believe the reason is that our reorder algorithm is less efficient when mapping to other rows.



(a) The expected length is 0.5.



(b) The total number of elements is 2^{22}.

Figure 7.6. Static search comparison.

7.5 Concurrent Operations Comparison

DACHash also supports concurrent execution of the *search*, *update*, and *delete* operations. In the experiment, we tested the performance in two groups. One is a mix of *80% search* operations and *20%* updates operations (10% *update* and 10% *delete* operations respectively). The other is a mix of *60% search* operations and *40%* updates operations (20% *update* and 20% *delete* operations respectively). Fig. 7.7 shows the DACHash and SlabHash results. It is clear that DACHash outperforms Slabhash by *19.92%* on average, and the peak performance of DACHash with the dynamic mapping schemes is *5.54 billion operations/second*, while SlabHash has the peak performance of *4.41 billion operations/second*. We believe there are mainly three reasons that account for the difference. First, the optimized structure of DACHash enables each thread to traverse its target bucket no matter what operations it has. This helps reduce thread divergence and improve warp execution efficiency. Second, our proposed reorder algorithm is more cache-friendly. Even though GPUs hide memory latency by scheduling available thread blocks, frequent memory operations still deteriorates the efficacy of latency hiding. So in our reordered input list, keys with the same or nearby hash values are grouped together. This increases the likelihood of threads in a warp being mapped to the same or nearby super nodes, so that the cache hit rate improves. Third, our dynamic mapping schemes, especially the *many-to-one mapping* scheme, minimizes the effect of thread divergence by making threads in a group process operations cooperatively with the help of CUDA primitives such as *shfl()* and *ballot()*, thus improves the overall performance. Also, note that when there are more update operations such as 20% *update* and 20% *delete*, the throughput for them is lower than that of 10% *update* and 10% *delete*. We believe that updates operations (e.g., *update* and *delete*) may trigger more memory operations so that they are more expensive than *search*.

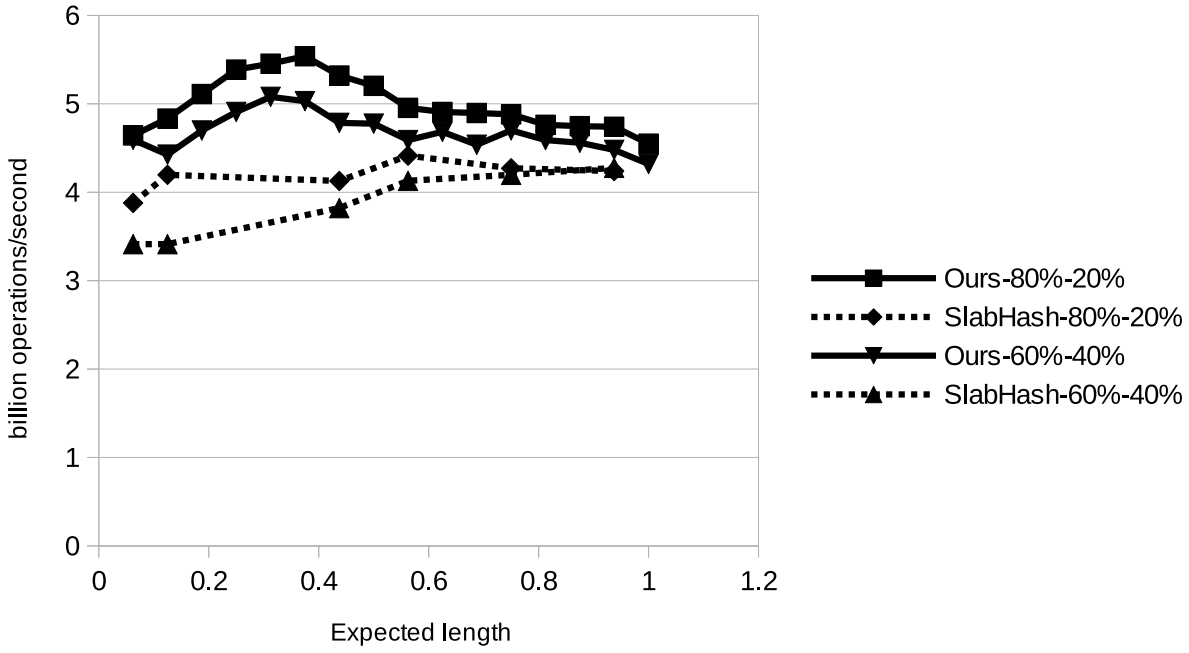


Figure 7.7. Concurrent operation comparison. The total number of operations is 2^{22} and node size is 16*4 bytes. Note that we use the dynamic mapping schemes in this experiment.

Chapter 8

CONCLUSIONS

In the past decade, rapidly growing data in numerous fields such as computational geometry and bio-informatics have given rise to research in high throughput hash tables. Hash table suffers from poor memory performance and thread divergence on GPUs. We present a dynamic, high throughput, GPU architecture-aware hash table in this thesis. Our proposed reorder algorithm and dynamic mapping schemes help improve the performance of hash table significantly, and beats the state-of-the-art implementation reported in the literature. We conducted experiments in three different categories to compare against the state-of-the-art solution SlabHash to verify our proposed DACHash. We also demonstrated the effectiveness of our proposed reorder algorithm by presenting the performance counters for L2 cache hit rate and bandwidth from the NVIDIA Visual Profiler.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Y. Xu, L. Gao, R. Wang, Z. Luan, W. Wu, and D. Qian, "Lock-based synchronization for gpu architectures," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 205–213.

[2] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 2002, pp. 73–82.

[3] P. Misra and M. Chaudhuri, "Performance evaluation of concurrent lock-free data structures on gpus," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 53–60.

[4] L. Verkleij, "Boosting shared hash tables performance on gpu," Ph.D. dissertation, University of Twente, Enschede, The Netherlands, 2016.

[5] B. Lessley and H. Childs, "Data-parallel hashing techniques for gpu architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 237–250, 2019.

[6] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "A dynamic hash table for the gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 419–429.

[7] D. Guide, "Cuda c programming guide," *NVIDIA, July*, 2013.

[8] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Building an efficient hash table on the gpu," in *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 39–53.

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

[10] I. García, S. Lefebvre, S. Hornus, and A. Lasram, "Coherent parallel hashing," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 6, pp. 1–8, 2011.

[11] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan, "Stadium hashing: Scalable and flexible hashing on gpus," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 63–74.

[12] L. Gao, Y. Xu, C. Xu, R. Wang, H. Yang, Z. Luan, and D. Qian, "Towards a general and efficient linked-list hash table on gpus," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 1452–1460.

[13] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, "Warpcore: A library for fast hash tables on gpus," *arXiv preprint arXiv:2009.07914*, 2020.

[14] NVIDIA. (2021) Nvidia ampere architecture. [Online]. Available: https://www.nvidia.com/en-us/data-center/ampere-architecture/

VITA

Hao Zhou

EDUCATION

Master of Engineering student in Computer Science at the University of Mississippi, August 2019 - May 2021. Thesis title: "DACHash: A Dynamic, Cache-Aware and Concurrent Hash Table on GPUs".

Bachelor of Science (May 2019) in Computer Science, University of Mississippi, Oxford, Mississippi.

ACADEMIC EMPLOYMENT

Graduate Research Assistant to Prof. Byunghyun Jang, Department of Computer and Information Science, University of Mississippi, Spring 2020 - Spring 2021. Research activities include: developing concurrent data structures on GPUs and solving the problem of fabric defect detection by computer vision and deep learning algorithms.

Graduate Teaching Assistant, Department of Computer and Information Science, University of Mississippi, August 2019 - Spring 2020. Responsibilities include: tutoring students taking computer science courses (e.g., Java, C/C++ and Data Structure) in their assignments and projects.

PUBLICATIONS

(In review) **H. Zhou**, D. Troendle and B. Jang, "DACHash: A Dynamic, Cache-Aware and Concurrent Hash Table on GPUs," The 30th International Conference on Parallel Architectures and Compilation Techniques (PACT'21).

H. Zhou, B. Jang, Y. Chen and D. Troendle, "Exploring Faster RCNN for Fabric Defect Detection," 2020 Third International Conference on Artificial Intelligence for Industries (AI4I), 2020, pp. 52-55, doi: 10.1109/AI4I49448.2020.00018.

ACADEMIC AWARDS

Summa Cum Laude, University of Mississippi (UM), May 2019.

National Scholarship, Ministry of Education of the P.R. China, 2015

Outstanding Freshmen, North China University of Technology, 2014