2022 21st International Symposium on Parallel and Distributed Computing (ISPDC) | 978-1-6654-8802-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISPDC55340.2022.00013

Cuckoo Node Hashing on GPUs

Muhammad Javed
Department of Computer and Information Science
The University of Mississippi
University, Mississippi
mhjaved@go.olemiss.edu

Hao Zhou
Department of Computer Science and Engineering
Pennsylvania State University
State College, Pennsylvania
hfz5190@psu.edu

David Troendle
Department of Computer and Information Science
The University of Mississippi
University, Mississippi
david@cs.olemiss.edu

Byunghyun Jang
Department of Computer and Information Science
The University of Mississippi
University, Mississippi
bjang@cs.olemiss.edu

***Abstract*— The hash table finds numerous applications in many different domains, but its potential for non-coalesced memory accesses and execution divergence characteristics impose optimization challenges on GPUs. We propose a novel hash table design, referred to as *Cuckoo Node Hashing*, which aims to better exploit the massive data parallelism offered by GPUs. At the core of its design, we leverage Cuckoo Hashing, one of known hash table design schemes, in a closed-address manner, which, to our knowledge, is the first attempt on GPUs. We also propose an architecture-aware warp-cooperative reordering algorithm that improves the memory performance and thread divergence of Cuckoo Node Hashing and efficiently increases the likelihood of coalesced memory accesses in hash table operations. Our experiments show that Cuckoo Node Hashing outperforms and scales better than existing state-of-the-art GPU hash table designs such as DACHash and Slab Hash with a peak performance of 5.03 billion queries/second in static searching and 4.34 billion insertions/second in static building.**

I. Introduction

Basic data structure semantics are typically specified for a single-threaded environment, but require a concurrent implementation to perform well in modern, highly parallel execution environments. Designing and optimizing concurrent data structures for modern multicore processors, however, have proven to be a challenging task. The primary difficulty is concurrency. Threads executing concurrently may interleave their operations in many different ways with potentially unexpected outcomes. Along with correctness, there are numerous challenges related to performance and scalability. Hardware thread execution model, the layout of data in memory, and the communication mechanism across processors all influence performance.

GPUs have become an accelerator of choice for data and compute intensive tasks. Their unique throughput oriented architecture that trades memory subsystem (e.g., effective cache lines per thread) for ALU units requires the following new considerations when designing concurrent data structures: First, the GPU implements a thread model that a group of threads execute the same single instruction in lock step. This model, called the Single Instruction Multiple Thread (SIMT) model [1], can cause thread divergence where threads in a group remain idle while others take the control path. Second, a SIMT model cannot execute an instruction until the data for all threads in the SIMT model is available to the execution unit. Thus, designs with good spatial data locality within a SIMT model are desirable. Third, the thread execution order is out of the programmer's control. GPU threads are scheduled as a group by the hardware in a two-level hierarchical fashion based on a certain policy. Such fixed hardware scheduling policies and thread execution models can cause frequent lock-oblivious thread switching and worsen under increased concurrency.

A hash table implements a dictionary of <key, value> pairs [2]. Dictionary objects can be inserted, deleted, updated, or searched. While there are many different hash table design schemes, Cuckoo Hashing [3] is one scheme that guarantees worst-case $\mathcal{O}(1)$ lookup time by evicting an existing key to a different location in the table when a new key's hash collides with the existing key. However, Cuckoo Hashing can make this guarantee at the cost of making the table static and by using open-addressing techniques.

This paper presents the design and implementation of a concurrent hash table for GPUs that uses a dynamic Cuckoo Hashing scheme. We call the proposed hash table design *Cuckoo Node Hashing*. Cuckoo Node Hashing is the first design to use Cuckoo Hashing in a closed-addressing scheme and may open the door to other such schemes. The novel features of our Cuckoo Node Hashing can be summarized as follows:

First, the proposed base structure of the hash table enables Cuckoo hashing on the entries within a Cuckoo node. This unique Cuckoo hashing scheme does not guarantee a constant search lookup time as original Cuckoo Hashing does, but, does minimize the addresses probed within the table and improves lookup efficiency. Next, a GPU architecture-aware warp-cooperative reordering algorithm is proposed to improve the performance of the hash table. The algorithm leverages memory coalescing and warp-level primitive functions to reorder data in a way that does not strain the GPU's memory

978-1-6654-8802-0/22/$31.00 ©2022 IEEE
DOI 10.1109/ISPDC55340.2022.00013

subsystem. The efficiency of reordering is very important for such an algorithm as it is a memory intensive task that can easily lose its benefits.

Our experiments conducted on an NVIDIA RTX 3090 show that, for random input datasets, our Cuckoo Node Hashing performs at a peak of *5 billion queries per second* for both when all searched keys exist in the table and when they do not exist. It translates to a peak speedup of 3.88× over DACHash [4] and 7.61× over Slab Hash [5] for when all keys exist, and 15.41× over DACHash and 14.10× over Slab Hash when all queries fail. Additionally, we find that our data reordering algorithm performs at a peak reordering rate of *6.31 billion elements reordered per second* which translates to a peak speedup of 2.23× over DACHash's reordering algorithm. Our profiling shows that our reordering algorithm is more efficient than DACHash's in L1 and L2 cache utilization, demonstrating that the proposed algorithm is more amendable to the GPU's architecture.

II. Related Works

Cuckoo Hashing [3] was a seminal paper introducing a novel hash table scheme. Cuckoo Hashing guarantees constant search performance because keys that are searched for could only be found in one of two hash buckets. Two hash functions determine the two possible buckets for each key. As a result, Cuckoo Hashing has an elegant search function which simply compared a key to two locations and checked for any matches. It has an insertion algorithm that places keys only in buckets to which they hash to. The algorithm evicts keys that are already situated in the table (similarly to cuckoo birds in a nest) and keeps rehashing old keys until a configuration is found where every key is situated in the table at one of its hashed locations. However, such a configuration is not guaranteed to always exist. Because of collisions with the two hash functions, it is possible that a set of keys could not be hashed to the table in any configuration. Cuckoo Hashing realizes this when the insertion process has iterated past a threshold. In this case, the table will be rehashed to an up-sized table.

Khorasani et al. [6] proposed Stadium Hashing that introduces a unique approach to sustain a very large table size. This is particularly useful in scenarios where the hash table can not reside in GPU memory. They achieve it by keeping the hash table in system main memory and a supplemental data structure, referred to as a ticket board, in GPU memory. The ticket board manages all buckets in the hash table by keeping track of bucket availability and hints as to what keys may reside in the buckets. They also introduce a SIMT-aware version of their table which allows threads in a warp to collaborate on operations.

Ashkiani et al. [5] proposed a dynamic hash table using chaining on GPUs referred to as Slab Hash. They found that the linked list element of traditional chaining methods was excellent for a dynamic hash table on GPUs. However, traditional linked lists exhibited a large amount of random memory requests. In order to mitigate that, they introduce the Slab List: every node in the linked list now contains a contiguous array of elements and one pointer rather than just one element and one pointer. This design was derived from Braginsky and Petrank's idea of a linked list that exploited spatial locality [7].

DACHash [4] introduced another chaining approach that was oriented towards cache awareness. The table featured buckets with chains of Super Nodes. These Super Nodes are composed of contiguous memory chunks to be more favorable to cache, similarly to Slab Hash. However, DACHash attaches a stack to the table which contains Super Nodes. Thus, when a thread needs to attach a Super Node to a chain, it can refer to the stack as a source of Super Nodes. When the table needs to get rid of unused Super Nodes due to deletions, the stack becomes replenished with those Super Nodes. In order to further improve cache performance, DACHash implements data reordering in their table. When an operation on the table is about to occur, the input data is reordered such that keys which hash to the same bucket are likely to be found closer together in the input data. DACHash also introduces a dynamic thread-to-data mapping scheme that can change operation techniques based on a certain threshold for better performance.

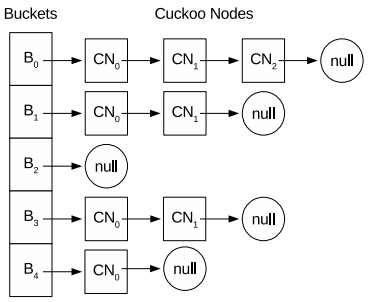

Fig. 1: A Cuckoo Node Hashing with 5 buckets. The base structure resembles traditional hashtable with linked lists of Cuckoo Nodes.

III. Basic Design And Implementation

Modern applications execute in a highly threaded environment, and require high-performance supporting data structures consistent with that environment. GPUs offer an inexpensive, power efficient, massively parallel execution environment, but inject challenging algorithm design considerations. Linked lists, which are essential to our algorithm, are especially challenging because their scattered memory access patterns cannot be processed efficiently on GPU architecture.

A. Basic Data Structure

Our Cuckoo Node Hashing adopts a chaining technique as a basic structure. The hash table consists of a set of B buckets (Figure 1) such that the set of buckets can be indexed via the integers between 0 and $B-1$. Each bucket contains only a pointer which points to the first node of a linked list. Traditionally, the nodes that compose a linked list contains just a single element and a pointer to another node. This allowed

for modifications of the linked list to occur at a fine granularity. However, at the hardware level, linked list traversal can cause irregular memory access patterns, which are unfavorable to cache memory. To mitigate this, our proposed hash table contains buckets which point to a linked list of nodes, which we refer to as *Cuckoo Nodes*, where each node consists of a bounded array of multiple elements. This approach was inspired by Slab Hash [5] and DACHash [4], and sacrifices fine grained node traversal and modification in favor of improved memory performance.

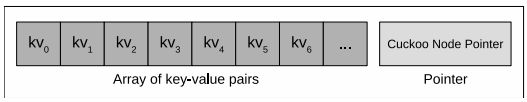

Fig. 2: The internal structure of a Cuckoo Node. It consists of a certain number of key-value pairs and a pointer to another Cuckoo Node.

B. Cuckoo Node

The Cuckoo Node (Figure 2) consists of two components: a bounded array that holds key-value pairs and a pointer to the next Cuckoo Node in the linked list. This differs from the traditional structure of a linked list where every node contains a single element rather than an array. This unique structure of a Cuckoo Node brings benefits of better per-warp memory accesses over scattered memory access patterns of traditional linked list nodes.

This node structure also introduces an opportunity for the Cuckoo Hashing scheme. With the increasing of key-value pairs per node, every Cuckoo Node within a bucket can now be treated as its own Cuckoo Hash table. To our knowledge, this is the first time Cuckoo Hashing is used in a closed addressing hash table scheme.

The pointer within each Cuckoo Node holds the address of the next Cuckoo Node. This allows for concurrent modification of a chain of Cuckoo Nodes through the use of atomic functions such as compare-and-swap (CAS) [1]. The result is a table that can be dynamically resized with no need for rebuilds. The original Cuckoo Hashing scheme encountered significant performance degradation because each failed insert triggered a rebuild, while in our scheme a failed insert simply expands the Cuckoo Node list.

The allocation system for Cuckoo Node Hashing is the same as the one in DACHash [4], in that we pre-allocate a large amount of Cuckoo Nodes and place them onto a stack.

The first hash function associated with the hash table is $h_B(key)$. In a table with B buckets, this function will hash the key to some value from 0 to $B-1$. Effectively, this function determines which bucket the key will be placed into. Since this is a closed addressing hash table scheme, keys can only be found in the bucket to which this function outputs.

There is also a set of hash functions $H = \{h_i(key), 1 \leq i \leq k\}$, where k is a configurable integer value which denotes the size of set of hash functions H. Each h_i is a Cuckoo Node hash function, and, ideally, $h_i(key) \neq h_j(key)$ when $i \neq j$. $h_i(key)$ determines the key's placement within a Cuckoo Node. Thus, for each $h_i(key)$ we must have $0 \leq h_i(key) < |S| - 1$, where S is the number of maximum keys a Cuckoo Node can hold.

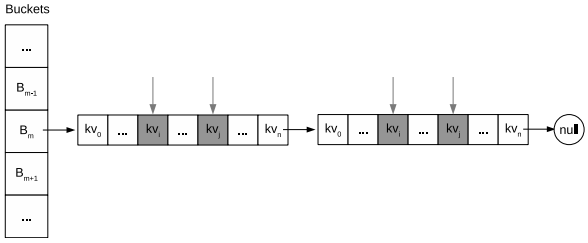

Fig. 3: An example of the search algorithm. The key being searched is hashed to the bucket B_m and the two hash functions associated with the table tell us that the key could be in the *i* or *j* spot of any of the 2 Cuckoo Nodes in the chain. The search algorithm then performs 2 probes per Cuckoo Node for a total of 4 probes, regardless for any node size *n*.

Algorithm 1: Search Operation

```
   Input: given_key
 1 H = set_of_hash_functions();
 2 cuckoo_node = hB(given_key);
 3 while cuckoo_node != null do
 4     foreach h in H do
 5         index = h(given_key);
 6         if cuckoo_node[index].key == given_key then
 7             return cuckoo_node[index].value;
 8         end
 9     end
10     cuckoo_node = cuckoo_node.next;
11 end
12 return KNF;
```

C. Hash Table Operations

We implement the four standard hash table operations: search, update, insert, and delete. Additionally, a clean function is triggered when an insert allocation fails.

The ***search*** operation (Figure 3 and Algorithm 1) takes in a key (*given_key*) as input and attempts to retrieve its value from the hash table. However, if the key does not exist in the hash table, then the search operation returns a Key-Not-Found (*KNF*) sentinel. The search operation starts by hashing *given_key* using the $h_B(key)$ hash function. Then, for every Cuckoo Node within bucket $h_B(given\_key)$, the search operation compares the keys located at every position in the set $\{x \mid (\forall h \in H)[x = h(given\_key)]\}$ with *given_key*. If there is a match, the search operation returns the associated value of that key. Otherwise, the search operation returns *KNF*. As a

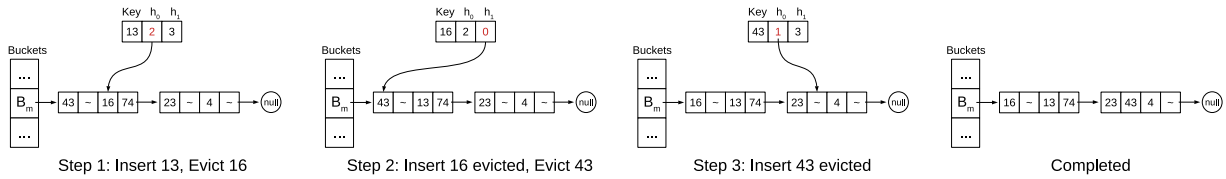

Fig. 4: An example insertion of the key 13 which has hashed to the bucket B_m and the hash table is configured with a *max_loop* value of 2.

result, Cuckoo Node Hashing only performs $|H|$ comparisons for every Cuckoo Node in a bucket because it needs to check the index given by every hash function in the set H.

The ***update*** operation takes in a key (*given_key*) and value (*given_value*) as input and attempts to replace the value associated with *given_key* in the hash table with *given_value*. This operation behaves similarly to search and, as a consequence, has the same procedure except it updates the value once the key is found.

The ***insert*** operation takes in a key (*given_key*) and value (*given_value*) and inserts the key and value in the hash table. Since we do not allow duplicate keys in the table, we first search for *given_key* to ensure it does not exist within the hash table. If *given_key* does exist in the hash table, then we simply update its value with *given_value*. The insertion operation starts by hashing *given_key* using the $h_B(key)$ hash function that determines which bucket *given_key* is inserted into ($h_B(given\_key)$). We then access the head Cuckoo Node of the bucket $h_B(given\_key)$ and we pick some hash function h from H to apply unto *given_key* to find a point of insertion for *given_key*. If there is no other key at $h(given\_key)$, we simply insert *given_key* at this position and finish the insertion process. However, if there is already a key at $h(given\_key)$, then we simply evict the old key (*evicted_key*) and replace it with *given_key*. Next, we replace h with a different function from the set H and apply it to the evicted key to find its next point of insertion ($h(evicted\_key)$). Again, if another key is already at $h(evicted\_key)$ then we simply evict it, replace *evicted_key* with this newly evicted key, and repeat this process of picking some different hash function from H and applying it to *evicted_key* until we either find an empty spot for *evicted_key* or we conduct the evicting process at most *max_loop* times. Once *max_loop* is reached and and an insertion without an eviction has not occurred yet, we allocate and attach another Cuckoo Node if the current one does not point to another one already, and we begin the eviction loop again until we find an empty spot for *evicted_key* in the next Cuckoo Node. This process repeats until an insertion that does not cause an eviction occurs. An example scenario of insertion can be found in Figure 4 and the pseudocode can be found in Algorithm 2.

The ***delete*** operation takes in a key (*given_key*) and searches for it. If successful, it marks the *given_key* as logically deleted. This operation also behaves similarly to search and has the same procedure.

Algorithm 2: Insert Operation

```
Input: given_key
       given_value
1  cuckoo_node = hB(given_key);
   // Start hash function at h1
2  hash_function = h1;
3  evicted_kv = key_value(given_key, given_value);
4  do
       // Eviction loop
5      for i ← 0 to max_loop do
6          hash_function = alternate_hash(hash_function);
7          index = hash_function(evicted_kv.key);
8          evicted_kv = atomicExch(cuckoo_node[index],
            evicted_kv);
9          if evicted_kv == null then
10             break;
11         end
12     end
13     if cuckoo_node.next == null then
14         attach_cuckoo_node(cuckoo_node);
15     end
16     cuckoo_node = cuckoo_node.next;
17 while evicted_kv != null;
```

When an insert operation fails to allocate a new node, it calls the ***clean*** kernel, which frees and compresses nodes. Empty nodes are deallocated and become available for allocation. All normal operations are suspended while a clean is in progress.

IV. Data Reordering

GPU performance is known to be very sensitive to the memory access patterns [8]. Thus, improving memory access patterns is one of the most important and effective optimizations. With the modern GPU programming model, memory access patterns are a function of thread-data mapping and the hardware thread scheduler. We propose a data reordering algorithm to address the dynamic nature of thread-data mapping in hash table operations.

A. Warp-Cooperative Reordering Algorithm

To remove potential non-coalesced memory accesses and thread divergence found in hash table operations, we propose

Algorithm 3: Warp-Cooperative Reorder Algorithm

```
   Input: key_set
          reordered_key_set
          reorder_bucket_sizes
          max_bucket_size
          combining_factor
 1 N = key_set.length;
 2 R = reorder_bucket_sizes.length;
 3 lane_key = key_set[thread_id];
 4 lane_bucket = H_R(lane_key) / combining_factor;
 5 has_reordered_key = false;
   // Work-sharing loop
 6 while (work_queue = ballot(has_reordered_key == false)) do
 7     chosen_lane = ffs(work_queue);
 8     chosen_key = shfl(lane_key, chosen_lane);
 9     chosen_bucket = (shfl(lane_bucket, chosen_lane) +
         lane_id) % R;
10     chosen_inserter_lane =
         ffs(ballot(reorder_bucket_sizes[chosen_bucket] <
         max_bucket_size));
11     completed_reorder = false;
12     if lane_id == chosen_inserter_lane then
13         index =
             atomicInc(reorder_bucket_sizes[chosen_bucket]);
14         if index < max_bucket_size then
               // Key can be reordered
15             reordered_key_set[chosen_bucket *
                 max_bucket_size + index] = chosen_key;
16             completed_reorder = true;
17         end
18     end
19     if any(completed_reorder) then
20         if lane_id == chosen_lane then
21             has_reordered_key = true;
22         end
23     end
24     else
25         if lane_id == chosen_lane then
26             lane_bucket = (lane_bucket + warp_size) % B;
27         end
28     end
29 end
```

a warp-cooperative reordering algorithm. Reordering occurs right before a hash table operation takes place and operates upon the input data for that hash table operation. The algorithm aims to reorder keys such that keys which hash to the same bucket are more likely to be found contiguously in memory. Perfectly reordering the keys so that they are sorted by which bucket they belong to can be a very costly operation. Therefore, we take a heuristic approach that gives an approximate reordering that still reaps significant speedup. The algorithm does not need to make any assumptions about how the keys will be distributed among the buckets and is meant to reorder keys on-the-fly. Also, data reordering can be quite taxing on the memory subsystem. Therefore, we leverage memory coalescing to reduce the strain on the memory subsystem. The algorithm also features work-sharing within a warp in order to exploit the SIMT execution model and eliminate thread divergence for most of the algorithm.

The algorithm's first input is a set of keys (*key_set*) of size *N* to be reordered. The algorithm also takes in the memory space where the reordered keys will be placed in (*reordered_key_set*), which is of size *N*. Also, *reordered_key_set* is split up into equally sized chunks referred to as *reorder buckets*, such that every reorder bucket has size *max_bucket_size* (Figure 5). *reorder_bucket_sizes*, another input for the algorithm, keeps track of the amount of keys inserted into these equally sized buckets. If a bucket's size is equal to or exceeds *max_bucket_size* then that bucket is treated as full and cannot have anymore keys inserted into it.

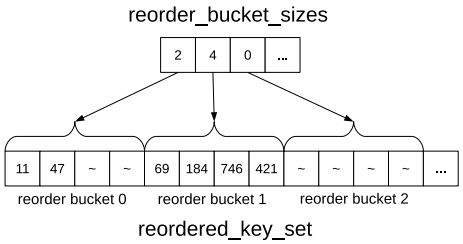

Fig. 5: *reordered_key_set* split into reorder buckets with *max_bucket_size* being 4.

The goal of the algorithm is to place keys that hash to the same bucket, or a neighborhood of buckets, into the same reorder bucket. The size of the neighborhood of buckets is determined by one of the inputs of the algorithm which is called *combining_factor*. As a result, the size of *reorder_bucket_sizes* is *R*, which is equivalent to the number of buckets in the hash table divided by *combining_factor*. If *combining_factor* is 1 then for every bucket in the hash table there is a corresponding reorder bucket. However, if it is greater than 1, then a corresponding number of buckets are mapped to a single reorder bucket. As a result, each reorder bucket in *reordered_key_set* should contain a contiguous set of keys which hash to the same neighborhood of *combining_factor* hash table buckets. If a neighborhood of buckets contains more keys than *max_bucket_size*, then the overflow keys will be placed in some other arbitrary reorder bucket.

The algorithm begins by instructing every thread to grab its relevant key into (*lane_key*), then hash *lane_key* and apply the combining factor to find the corresponding reorder bucket (*lane_key*), and finally create a boolean that represents whether the individual thread has finished their reordering job (*has_reordered_key*). Then, the work-sharing loop begins. At the beginning of every iteration of the loop, a $ballot$ [1] warp-primitive function is performed by all threads in the warp. The ballot function evaluates the predicate, which it receives as input, for every thread in the warp and returns a 32-bit value. This 32-bit value indicates which threads of the warp evaluated true for the predicate and which evaluated false. In the case of the work-sharing loop, the ballot at every iteration is to create an ever-updating list of the threads in the warp who have not finished reordering their key (*work_queue*).

Within the work-sharing loop, every thread performs the CUDA *ffs* (find first bit set) [1] function on *work_queue* to pluck a thread from their warp as the chosen lane (a thread in a warp) whose reordering task will be completed by the entire warp. Once the lane has been chosen (*chosen_lane*) this lane will have its relevant data copied from it by every other lane in the warp. Every lane will do this by performing the *shfl* (shuffle) [1] function on the chosen lane's key and bucket (*chosen_key* and *chosen_bucket* respectively). Once every lane has copied the chosen bucket, we offset this value for every lane by the value of the lane's id. Thus, when the lanes begin their search for a reorder bucket which has room for *chosen_key*, they will access *reorder_bucket_sizes* with *chosen_bucket* and the reading of *reorder_bucket_sizes* will be a coalesced memory access. The algorithm uses the *ballot* and *ffs* function again to determine which lane will be chosen to insert *chosen_key* into *reorder_bucket_sizes* as long as that lane reported to have found a reorder bucket that had room (*chosen_inserter_lane*).

In order to avoid race conditions, *chosen_inserter_lane* will atomically increment the size of the reorder bucket they found and ensure that the bucket has room. If the reorder bucket does have room, the lane will insert *chosen_key* into that bucket and the loop will continue on to the next lane which still has to reorder their bucket. However, if *ffs* returned 0 (i.e. no thread was able to find a suitable reorder bucket), then *chosen_lane* will offset its *lane_bucket* by the size of a warp. Thus, when the loop executes again, it will pick the same lane as the previous iteration and the search for a suitable reorder bucket will be offset by the warp size. Also, the warp-cooperative reordering algorithm can be found in Algorithm 3.

V. Performance Evaluation

In this section, we evaluate the performance of our proposed data reordering algorithm and Cuckoo Node Hashing. We performed all experiments on an NVIDIA GeForce RTX 3090 GPU, which is based on NVIDIA's Ampere architecture and features 10,496 CUDA Cores with a clock rate between 1.40 GHz and 1.70 GHz. The GPU is also equipped with 24 GB of GDDR6X memory. All code is written in CUDA v11.4 running on Ubuntu 20.04.

A. Data Reordering Performance

We begin by evaluating the performance of our warp-cooperative reordering algorithm against the reordering algorithm offered by DACHash [4]. Both algorithms use a similar parameter known as the combining factor. This parameter effectively measures the strictness of the reordering algorithms. When the combining factor is low, fewer hash table buckets are mapped to a reorder bucket. However, when the combining factor is high, more hash table buckets can be mapped to a single reorder bucket, which can make it easier for the reordering algorithms to find a suitable spot for their data. In our experiments we reorder 2^{25} random elements for a hash table that would have a bucket count of 2^{21}. The performance

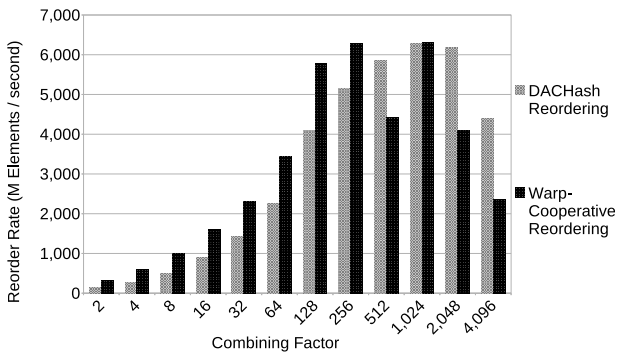

Fig. 6: Reordering algorithm performance: the proposed warp-cooperative vs. DACHash's.

is measured in millions of elements reordered per second and can be found in Figure 6. The evaluation demonstrates that our warp-cooperative reordering algorithm performs better up until a combining factor of 2^{11}. At that point, DACHash's reordering algorithm starts to perform better. This trend highlights that for lower combining factors, the warp-cooperative reordering algorithm performs significantly better. As a consequence, the warp-cooperative reordering algorithm can perform under a much stricter reordering scenario and efficiently produces a much more favorable reordering. However, we did notice a considerable dip in performance when the combining factor is set to 512, and this issue is a subject of ongoing investigation. Additionally, we found a peak reordering rate for our table resides at a value between 2^7 and 2^{10} for the combining factor. We consider this range of values for the combining factor to be more favorable because we noticed a trend such that lower combining factors consistently guaranteed faster search speeds.

Finally, data reordering is known to be a memory intensive task. Therefore, we profile the utilization of various memory components using the NVIDIA CUDA toolkit profiler. We compare the L1 and L2 cache utilization of the proposed warp-cooperative algorithm with DACHash's and these results are shown in Figure 7. We found that our warp-cooperative reordering algorithm has a higher peak utilization for all 3 memory types compared to DACHash's reordering algorithm. This is consistent with the trend of reordering performance discussed earlier. The cooperative nature of the algorithm helps explain the higher peak utilization in both caches. Since all threads in a warp are working on the same task, it is more likely that the data they need has already been loaded into the cache by another thread in the warp.

B. Hash Table Performance

Search and build rates are fundamental performance metrics which should be reported for all hash table designs. A lower bound on search performance can be estimated by searching for keys that exist in the hash table while an upper bound can be estimated by searching for keys that do not. For estimating

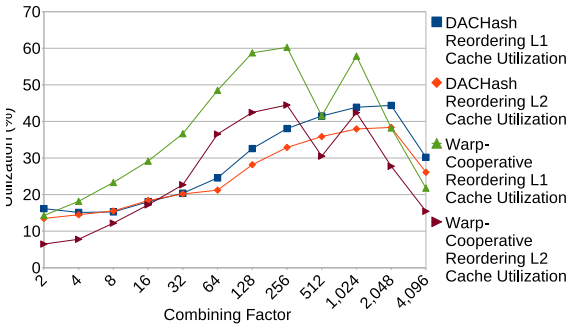

Fig. 7: DACHash's data reordering algorithm's L1 cache utilization vs. the warp-cooperative reordering algorithm's L1 cache utilization.

build performance, we time how fast a given set of keys can be inserted into the table.

First, we evaluate the performance of the proposed Cuckoo Node Hashing by building the table using one large batch of keys and then performing searches on the table with another large batch of keys. This building and searching is performed on 3 different hash table designs: Cuckoo Node Hashing, Slab Hash [5], and DACHash [4]. For all 3 hash tables, we evaluate how each performs with a varying number of buckets. We fix the size of the batches to be 2^{25} keys with all keys being generated with uniform randomness. For DACHash, we use a combining factor of 2^{10} and a node size of 32, which we found to be generally optimal configuration for DACHash. For our Cuckoo Node Hashing, we keep both the combining factor and node size at a constant 2^7, set the max loop value to 2^6, and have 4 hash functions belong to H. We began by building the table using a batch of insertion keys. At a lower bucket count, DACHash outperforms both Slab Hash and Cuckoo Node Hashing. However, as bucket count scales up, we begin to see that Cuckoo Node Hashing benefits from an increase in the number of buckets. Overall, we found the Cuckoo Node Hashing achieves a peak speedup of $3.20\times$ over DACHash and $2.16\times$ over Slab Hash in terms of building the table. The results can be found in Figure 8.

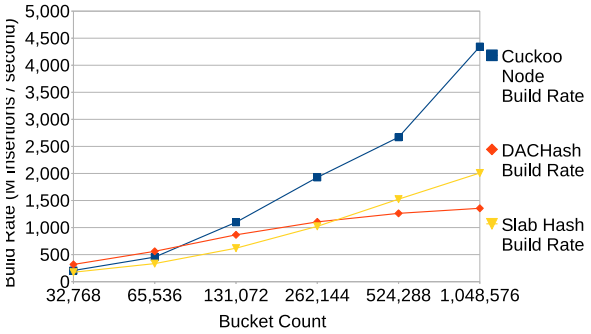

Fig. 8: Static build performance comparison.

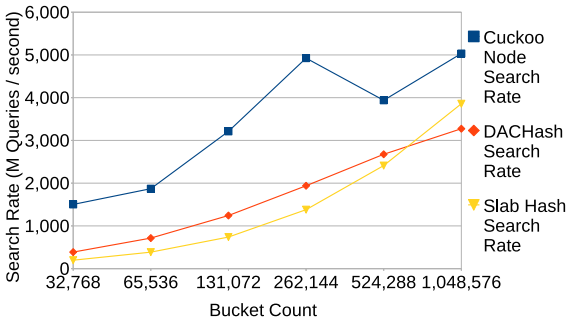

Fig. 9: Search performance when all keys searched for are guaranteed to exist in the table.

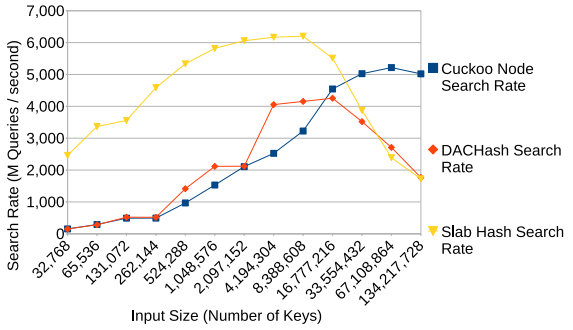

Fig. 10: Search performance across varying input sizes.

Then, we compare the performance of the 3 aforementioned hash tables in a static search setting. We use the same configuration for all tables as mentioned before and we perform a search on a prebuilt table. However, two distinct types of searches occur in our experiment. One type of search performed is on a set of keys such that all keys are guaranteed to exist in the prebuilt table (Figure 9 and Figure 10). In Figure 9 we fix the input size at 2^{25} random keys and vary the bucket count, while in Figure 10 we fix the bucket count at 2^{20} and vary the input sizes. Evaluating hash table performance under varying input sizes is critical in a concurrent setting as it helps estimate just how well the table can utilize the hardware. When varying the bucket counts, we found that Cuckoo Node Hashing attains a peak searching rate of $3.88\times$ over DACHash and a peak searching rate of $7.61\times$ over Slab Hash. However, when varying the input sizes, we found that Slab Hash and DACHash perform better than Cuckoo Node Hashing for small input sizes. However, for input sizes of 2^{25} and beyond, we find that Cuckoo Node maintains considerably better performance. This suggests that Cuckoo Node Hashing may be better for operations involving large amounts of data. The other type of search experiment has a set of keys where none of the keys exist in the prebuilt table (Figure 11). When keys do not exist, we found a peak speedup of $15.41\times$ over DACHash and $14.10\times$ over Slab Hash. We believe that this speedup over the other two hash tables is primarily due to the reduction of comparisons (probes) that occur per thread when

conducting a search, which leads to the next experiment.

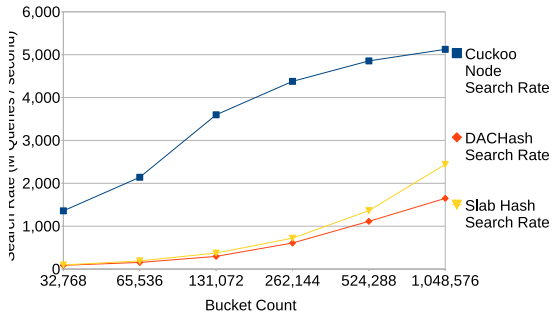

Fig. 11: Search performance when all keys searched for do not exist in the table.

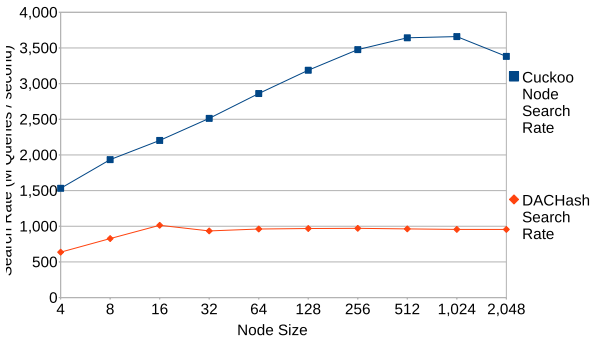

Fig. 12: Search performance for both DACHash and Cuckoo Node over varying node sizes.

In order to show that Cuckoo Node Hashing performs fewer probes when searching for a key, we conducted experiments to count and compare how many probes occur in two hash table designs. Slab Hash was not included since it has a very similar searching algorithms as DACHash which yields a very similar number of probes. During the search experiment where all keys exist in the table, we counted how many positions within the table a thread has to search for their desired key (Table I). We found that even at a low bucket count (2^{15}) our table still performs a small amount of probes. At higher bucket counts, most threads only have to probe between one and four positions in the hash table before they find their desired key.

Finally, because both DACHash and Cuckoo Node Hashing share the same design-specific parameter of node size, we compare how those two tables handle varying node sizes. We keep the number of buckets in both tables at a constant 2^{17}, and we set the Cuckoo Node hash table's combining factor to 2^7 and DACHash's to 2^{10}. We then performed a search of 2^{25} random keys that all were guaranteed to exist in the table. The results from the experiments are measured in millions of queries per second. We found that Cuckoo Node Hashing outperforms DACHash at any node size. These results can be found in Figure 12. As node size increases, Cuckoo Node Hashing continues to climb in performance. This is primarily

Probes	CNH		DACHash	
	KF	KNF	KF	KNF
32768	16.92	36.59	512.99	1039.06
65536	8.89	20.15	256.99	527.04
131072	4.82	12.03	128.99	271.03
262144	2.65	8.00	64.99	143.40
524288	1.52	5.32	32.99	79.61
1048576	1.19	4.05	16.99	47.26

TABLE I: Average number of probes performed per thread by both Cuckoo Node Hashing (CNH) and DACHash when keys were found (KF) and when keys were not found (KNF).

because as the node gets larger, the number of keys that can be held within the node increases. However, the number of queries that occur per node remains constant and depends on how many hash functions are in the set H. Thus, at large node sizes, a small number of probes occur for a large set of keys. On the other hand, DACHash plateaus at a node size of 16.

VI. Conclusion

We presented a novel, dynamic, high performance, and scalable hash table design for GPUs. Our Cuckoo Node Hashing completely eliminates the rebuild problem of conventional Cuckoo Hashing and reduces the number of probes required during hash table operations. Our experiments demonstrate that Cuckoo Node Hashing outperforms other state-of-the-art hash table designs. Additionally, we proposed a data reordering algorithm targeted specifically for GPU architectures that reduces data irregularities. This, in turn, improves the memory access patterns and minimizes thread divergence that occur during hash table operations.

References

[1] NVIDIA Corporation, "NVIDIA CUDA C++ programming guide," 2022, version 11.4.0.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[3] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms — ESA 2001*, F. M. auf der Heide, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 121–133.

[4] H. Zhou, D. Troendle, and B. Jang, "Dachash: A dynamic, cache-aware and concurrent hash table on gpus," in *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2021, pp. 1–10.

[5] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "A dynamic hash table for the GPU," *CoRR*, vol. abs/1710.11246, 2017. [Online]. Available: http://arxiv.org/abs/1710.11246

[6] 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)*, 2015, pp. 63–74.

[7] A. Braginsky and E. Petrank, "Locality-conscious lock-free linked lists," vol. 6522, 01 2011, pp. 107–118.

[8] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105–118, 2010.