Analyzing and Implementing GPU Hash Tables

Muhammad A. Awad* Saman Ashkiani† Serban D. Porumbescu*
Martín Farach-Colton‡ John D. Owens*

Abstract

We revisit the problem of building static hash tables on the GPU and present an efficient implementation of bucketed hash tables. By decoupling the probing scheme from the hash table in-memory representation, we offer an implementation where the number of probes and the bucket size are the only factors limiting performance. Our analysis sweeps through the hash table parameter space for two probing schemes: cuckoo and iceberg hashing. We show that a bucketed cuckoo hash table (BCHT) that uses three hash functions outperforms alternative methods that use iceberg hashing and a cuckoo hash table that uses a bucket size of one. At load factors as high as 0.99, BCHT enjoys an average probe count of 1.43 during insertion. Using three hash functions only, positive and negative queries require at most 1.39 and 2.8 average probes per key, respectively.

1 Introduction

Designing high-performance hash tables is a challenge in part because the design space is broad. First, the hash table designer must choose a design goal, perhaps prioritizing or balancing among query performance, build performance, and memory efficiency. Second, the designer must also decide between numerous alternatives in the hash table parameter space, which may include probing schemes, bucket sizes, probe complexity, and placement strategy.

In our work we do not propose new priorities or new dimensions in the hash table parameter space; we use existing techniques, such as bucketing and different probing strategies, albeit ones that have historically been understudied in the GPU space. Instead, we perform a full exploration of the performance landscape, across the entire parameter space, for static GPU hash tables. We begin by implementing an efficient generic hash table that successfully decouples the implementation of the hash table from the probing scheme. We show that we can implement different probing schemes (e.g., Listings 3, 4, and 5) atop our hash table implementation with no GPU-specific code.

The reason this decoupling is important is that we show, at least for static GPU hash tables, that hash table performance is (very) strongly correlated with the number of probes per item ("probe count") and thus primarily influenced by the choice of probing strategy. Our best implementation, a bucketed cuckoo hash table with 3 hash functions and 16-element buckets, requires 1.43 probes per insertion and achieves best-of-class performance on both insertions and queries at load factors as high as 0.99. We thus submit that any future hash table with superior performance must necessarily implement a probing scheme that sustains fewer probes per operation for a particular bucket size. Table 1 shows a summary of our recommendations for different hash table design space criteria.

Our contributions are:

- An efficient bucketed hash table implementation that decouples the probing scheme from the hash table and the GPU-specific details (Section 5);
- An analysis and sweep of the parameter space of two probing schemes: iceberg and cuckoo hashing on NVIDIA GPUs (Section 7.1);

*UC Davis
†NVIDIA
‡Rutgers University

	Load factor	Insertion	Query
Insertion	BCHT, $b = 16$	—	—
Query	BCHT, $b = 16$	BCHT, $b = 16$	—
Stability	IHT, $b = 32$	IHT, $b = 16$	IHT, $b = 16$

Table 1: Our hash table recommendations. If the application does not require stability, bucketed cuckoo hash with a bucket size of 16 and 3 hash functions is the best method whether we prioritize insertion rate, query rate, or load factor. With BCHT, we can achieve up to a load factor of 0.98. If we do require stability, then an iceberg hash table is the best choice. For an application that requires stability and prioritizes load factor, choose IHT with a bucket size of 32, which allows a load factor of up to 0.91. If instead the application prioritizes query or insertion rate, use a bucket size of 16 instead, which allows a load factor of up to 0.82. Each of these recommendations guarantees a success rate of at least 99% in building the hash table.

- Analysis-driven hash table recommendations for different use cases (Section 6); and
- Hardware-agnostic analysis of the two probing schemes to guide hash table designers when designing hash tables for future and current hardware (Section 7.4).

The remainder of the paper describes our hash-table implementation and performance analysis that enables us to make this probe-count conclusion and to construct the fastest NVIDIA GPU static hash tables in the literature.

2 Background and Previous Work

2.1 Performance and parameter space landscape

Hash table performance The primary design tradeoff in any hash table implementation is between three metrics:

1. **Query performance** is measured in queries per second and reflects the throughput of membership queries or key-value lookups into the hash table. Queries can either be positive (exist in the hash table) or negative (do not exist in the hash table).
2. **Build performance** is measured in keys per second and indicates the throughput of building a hash table from an unordered list of key-value pairs.
3. **Memory efficiency** expresses how well the implementation uses its allocated memory. In particular, the metric of *load factor* is the ratio between the amount of storage used to store key-value pairs and the total amount of storage allocated for the hash table.

The parameter space landscape We consider four parameters:

Probing scheme Probing schemes offer different techniques to resolve collisions between multiple keys mapping to the same bucket in the hash table. These schemes include linear probing, quadratic probing, double hashing, and cuckoo hashing. Compared to double hashing and cuckoo hashing, linear and quadratic probing schemes are prone to clustering.

Bucket size Using larger buckets helps to reduce the variance of the load between the buckets. As the bucket size increases, the achievable load factor for the hash table increases. The choice of the bucket size is influenced by hardware constraints and the key-value pair size; for instance, we can choose the bucket size to be the same as the cache line size.

Probe complexity A hash table can place key-value pairs in different buckets; each of these buckets is associated with a different hash function. Increasing the number of hash functions improves the achievable load factor but also increases the number of buckets we need to inspect while inserting or querying a key.

Placement strategy Since there are potentially multiple buckets where a given key-value pair might be placed, a hash table with multiple hash functions needs to decide on a placement strategy. One choice is to always insert in the first hash function's bucket, avoiding any placement strategy overhead. A different choice is to place a pair into the least loaded bucket out of the possible choices. This adds the additional overhead of inspecting all possible buckets.

2.2 Previous work in GPU hash tables Cuckoo hashing Cuckoo hashing [18] differs from open-addressing hash schemes in that a key-value pair may only be inserted into one out of two possible locations, making queries fast. If existing pairs occupy both of the locations, then we have a collision. We resolve the collision by performing the "cuckooing" operation in which the new pair pushes one of the existing pairs to a new location. Fotakis et al. [9] generalized cuckoo hashing and introduced d-ary cuckoo hashing, where the number of possible locations per key is d (in this paper, we use h). Using more than one hash function, they achieved 0.97 and 0.99 load factors for 4 and 5 hash functions, respectively. Panigrahy [19] generalized the bucket size b to achieve higher load factors introducing a "bucketized" cuckoo hash table (BCHT) while using $b = 2$. Later, Erlingsson et al. [8] experimented with a cuckoo hash table parameterized using both h and b.

The first hash table that could be built on the GPU [1] used a cuckoo hashing hash table formulation, and was succeeded by a simpler cuckoo hashing implementation (CUDPP [2]) that stored the entire hash table as a single table in global memory. CUDPP's implementation uses $h = 4$ and $b = 1$. Subsequent work has explored both different performance tradeoffs and a broader hash-table feature set; rather than summarize this work here, we direct the reader to the comprehensive 2020 survey by Lessley and Childs [15].

Mega-KV's 8-entry, two-hash-function BCHT targets a hybrid CPU-GPU in-memory key-value store [20]. The differences between our BCHT with $b = 8$ and Mega-KV are that we use three hash functions instead of only two and a single hash table instead of multiple partitions. Using three hash functions allows our BCHT to achieve higher load factors and insertion throughput. However, the drawback of using an additional hash function is that the query throughput becomes lower (1.38× at load factor 0.96) when all queries are negative. Mega-KV chose one point in the much larger implementation space that we survey in this work. As we shall see in our results (Section 7), a bucket of size 16 is the optimal choice for the GPU hardware.

Double hashing WarpDrive [13] and WarpCore [14] introduced an open-addressing static hash table that uses a double hashing scheme. They achieve up to 2.84× and 1.3× faster insertion and query performance than CUDPP. The main difference between our approaches and WarpCore's is the probing scheme choice. Our bucketed cuckoo hash table (BCHT with $b = 16$) achieves similar insertion performance and slightly faster ($\leq 1.2\times$) positive query performance. For negative queries, BCHT achieves speedups of {2.3×, 1.6×} compared to WarpCore with probing windows of {4, 8}. BCHT negative query performance significantly outperforms WarpCore at high load factors, reaching up to 6.4× speedups.[1] As the load factor increases, double hashing techniques search the entire hash table until finding the key or an empty location. In contrast, cuckoo hashing only reads as many buckets as there are hash functions. While performance may be comparable for some operations, WarpCore's unacceptably low performance for negative queries motivates our BCHT recommendation.

Dynamic cuckoo hashing GPU implementations of cuckoo hashing do not typically target dynamic scenarios where new items are inserted into or deleted from an existing hash table. However, Li et al. [16] recently proposed a dynamic cuckoo hash table implementation (DyCuckoo). In static scenarios, DyCuckoo achieves similar performance to Mega-KV and WarpDrive. On insertions, BCHT with $b = 16$ is 1.1× faster than DyCuckoo. Positive queries in BCHT achieve higher speedups with an average speedup[1] over DyCuckoo of 1.2×. For negative queries, BCHT achieves significant speedups {2.1×, 1.1×} at load factors between {0.6, 0.93}. However, at higher load factors, DyCuckoo starts to outperform our BCHT, reaching a speedup of 1.1× at a load factor of 0.97 since BCHT performs up to three probes when the key does not exist on the hash table; on the other hand, DyCuckoo only performs up to two lookups.

2.3 Graphics processing units (GPUs) GPUs are composed of many cores grouped to form a streaming multiprocessor (SM). SMs are coupled with a high-bandwidth device memory to feed the SM cores with data. Each SM has a private L1 cache, and all SMs share an L2 cache. NVIDIA GPUs use a 128-byte cache line equivalent to four 32-byte sectors. Depending on the GPU microarchitecture, load and store granularity in the memory system hierarchy may use one or multiple sectors [12].

A block of threads of user-defined size runs on the GPU cores in groups of 32 threads (warp) that execute in SIMD style. To efficiently access GPU memory, adjacent threads must access memory in a coalesced fashion, which requires avoiding branch divergence within a warp. A typical design decision for GPU data structures is to use the warp-cooperative work-sharing (WCWS) strategy [3] to eliminate branch divergence and achieve

[1] Averaged results over different load factors for inserting and querying 50M unique keys.

coalesced memory access. CUDA cooperative groups[2] simplify implementing WCWS and enable fine control over the number of threads that cooperatively execute a data structure task. In our implementation (Section 5), we heavily use cooperative groups and extend WCWS to use a sub-warp group of threads (i.e., tile of threads).

3 Probing Schemes

In this section, we describe the implementation of two different hash table approaches: bucketed cuckoo hash table (BCHT) and iceberg hash table (IHT). BCHT is a generalization of CUDPP's cuckoo hash table implementation (1CHT), where BCHT's bucket size can be greater than one. Next, we discuss the insertion and query algorithms for each of these approaches. A hash table is represented using a fixed size array split into a number of buckets m of size b (i.e., a bucket contains b locations for b pairs). The array has a capacity C of key-value pairs where $C = m \times b$. A hash table containing n keys has a load factor of n/C. For simplicity, we only discuss storing keys (values are typically handled by bundling them with their associated key).

3.1 Bucketed cuckoo hash table We first begin by specifying h hash functions $H_0, H_1 \dots H_{h-1}$. We then allocate a table of m buckets, each with b valid locations in each bucket. A number of hash functions h maps a key k to the buckets $H_0(k) \dots H_{h-1}(k)$. Thus, the lookup process only requires checking these buckets. The key is present in the hash table only if it exists in one of these buckets. We use a constant number of hash functions, so queries have an $O(1)$ complexity.

Construction is (slightly) more complex and demonstrates the unique idea behind cuckoo hashing. We initialize all buckets of the hash table to empty and then insert each key into the hash table. For key k, we begin with the first hash function H_0 and attempt to insert k into bucket $H_0(k)$. If the bucket is empty, we insert the key and are done. If it is full, we *exchange* our item k with $\hat{k}$, a random key already in $H_0(k)$. It is this exchange that is the characteristic feature of cuckoo hashing, and we say that k *cuckoos* $\hat{k}$. The key k is now stored in the hash table at $H_0(k)$ and we now must insert $\hat{k}$. If $\hat{k}$ was previously stored with H_i, we (typically) begin this new insertion with the next sequential hash function, $H_{(i+1) \bmod h}$.[3] An advantage of using an increasing order when picking a hash function is that if a bucket contains an empty key during queries, the query can exit early. We finish when we successfully store our key into an empty bucket or reach a maximum number of exchanges, in which case we have failed to construct our hash table. The theoretical analysis of cuckoo hashing requires a fairly low load factor in order to guarantee that the construction fails with low probability, but in practice, it can achieve high load factors.

Another variation of bucketed cuckoo hash tables uses a load-balancing scheme when picking a bucket out of the possible h buckets. Instead of always inserting a key into the first bucket as we discussed previously, we insert the key into the least loaded bucket out of all the possible buckets. A load-balancing technique improves the achievable load factor at the cost of the additional memory transfers required for evaluating the load of all possible h buckets. Since insertion could pick any bucket out of the possible buckets, during queries early-exit strategies cannot be used. This results in an overhead similar to the insertion's overhead (unless the key is found in an early bucket).

3.2 Iceberg hashing In a stable hash table, larger buckets increase query and build throughput but do not allow reaching high load factors. "Power-of-two-choices" hashing can improve the load factor at the cost of increased work for queries and insertions. This technique is a powerful one: "even a small amount of choice can lead to drastically different results in load balancing" [17]. Given h choices, the maximum load for n keys now decreases from $O(\log n / \log \log n)$ to $O(\log \log n / \log h)$ [4]. While we can directly combine the two techniques, doing so does not yield both the performance gains and memory efficiency gains that we target.

What we like about larger buckets is the common-case behavior: we most likely find the key we are searching for, or succeed with an insertion, in the first bucket. What we like about power-of-two-choices is that it gives us better memory utilization for the extraordinary cases. For these cases, we are willing to pay the extra cost of the multiple choices. Iceberg hashing [6] combines the common and extraordinary cases [4] in a novel way. First,

[2] `https://developer.nvidia.com/blog/cooperative-groups/`

[3] We can determine which of our h hash functions was used to store a particular key at a particular address $H_i(k)$ by hashing our key with all h of our hash functions and seeing which of those hash functions yields the hash table address $H_i(k)$, breaking ties by favoring the lowest index hash function.

[4] Separating a compute task into common and extraordinary components has been previously explored in other GPU application

we establish the intuition. Consider filling up most of the hash table cheaply by placing keys in the first bucket where they fit, then fill the rest of the table by optimizing placement (i.e., using power-of-two-choices). The term "iceberg hashing" reflects the two strategies: the common case (corresponding to the largest part of the iceberg under the water) and the extraordinary case (the small part of the iceberg above the water). Iceberg hashing is stable (i.e., once we insert a key, we never move it).

Preliminaries. Choose a threshold t for buckets. t marks the boundary between the aforementioned "common" (buckets have fewer than t keys) and "extraordinary" (at least t) cases. If each bucket has on average b_{avg} keys, then make t a little larger than b_{avg}. Choose 3 hash functions (H_p, H_{s0}, H_{s1}). H_p is the hash function for the "primary" bucket; the two $H_{\{s0,s1\}}$ are for "secondary" buckets.

Insertion. To insert a key k we compute its primary hash function $H_p(k)$. If the resulting bucket has fewer than t keys, insert the item there and return. Otherwise, compute *both* secondary hash functions $H_{s0}(k)$ and $H_{s1}(k)$. Insert the key into the less full of those two buckets. If all of these buckets are full, our build has failed. Choose another set of hash functions or enlarge the table and try again. We expect this scenario will only occur with a very low probability if the threshold and the bucket size are properly chosen.

Query. To look up a key k we first check location $H_p(k)$ and return the key if found. Otherwise, check both locations $H_{s0}(k)$ and $H_{s1}(k)$. If the key is not found in any of those locations, it is not in the hash table.

4 Common Parameter Space Decisions

When implementing the aforementioned hash table strategies, we face some common design decisions: how many key-value pairs we allocate per bucket, the number of hash functions we use in the hash table, and how we decide to place pairs given multiple choices. We now discuss these parameters and how they influence the performance of hash table operations.

4.1 Bucket size

Effect of bucket size on the number of probes Consider a cuckoo hash table that uses two hash functions, $h = 2$, and where the bucket size is one, $b = 1$. In this case, as the load increases, the percentage of key-value pairs in their first location decreases, though the asymptotic behavior of such a system is not well understood. As the bucket size increases, the percentage of items in their first position increases, and when $b = \omega(\log n)$, all items are in their first position, w.h.p, which we can see via a balls and bins analysis.

Effect of bucket size on the memory access pattern Not only does using larger bucket sizes improve the achievable load factor, it also satisfies the GPU optimal memory access pattern. In general, we would like adjacent threads to access adjacent memory locations achieving a *coalesced* memory access. With 4-byte keys and 4-byte values and a GPU cache-line size of 128 bytes, we expect the optimal bucket size from the memory-system perspective to be 16.

We conclude that bucketizing hash tables is a good idea, in terms of decreasing expensive memory accesses in exchange for inexpensive computation. Increasing the bucket size improves both the achievable load factor and the number of probes per key. However, increasing the bucket size beyond the cache-line size reduces the achievable insertion and query throughput.

4.2 Number of hash functions The number of hash functions, corresponding to the number of possible locations for any pair, has both positive and negative effects on hash table performance. Increasing the number of hash functions enables achieving high load factors. For instance, in a standard cuckoo hash table, using two hash functions can only achieve load factors up to only 0.5. Adding a single additional possible location increases this achievable load factor up to 0.92 [7]. However, the benefits of adding more locations are marginal beyond three hash functions. The reason behind these achievable higher load factors is that adding more possible locations reduces the possibility of ending up in a cycle while moving keys between buckets when a collision happens and decreases the number of probes required for insertion. However, increasing the number of hash functions also increases the work necessary to query a hash table, particularly when a queried key is not present in the hash table. In this case, having h hash functions requires h probes into the hash table before concluding the queried key is not present.

In a bucketed cuckoo hash table, when the bucket size is large, using two hash functions is enough to achieve

domains, e.g., sparse-matrix dense-vector multiplication [5].

```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table){
  cg::thread_block thb = cg::this_thread_block();
  auto tile = cg::tiled_partition<bucket_size>(thb);
  auto thread_rank = tile.thread_rank();
  bool success = true;
  while(auto work_queue = tile.ballot(to_insert)){
    auto cur_lane = __ffs(work_queue) - 1;
    auto cur_pair = tile.shfl(pair, cur_lane);
    auto cur_result = insert(tile, cur_pair,table);
    if(tile.thread_rank() == cur_lane){
      to_insert = false;
      success = cur_result;
  }}
  return success;}
```

Listing 1: Tile-wide cooperative insertion.

high load factors, but this requires balancing the load between the two possible buckets. A simpler approach where we insert a pair into the first bucket using only two hash functions reduces the achievable load factor, but adding a single hash function to this simple approach allows achieving higher load factors. Moreover, using a simple insertion approach improves the insertion throughput (over the load-balanced technique), but it reduces the query throughput. Specifically, when the number of positive queries is low, using more hash functions reduces the performance of queries since a query requires loading a number of buckets equal to the number of hash functions. In general, we opt to use three hash functions and a simple insertion strategy; however, if the primary goal is to improve the query rate, using a load-balanced insertion strategy and only two hash functions is a better choice.

4.3 Placement strategies While increasing the number of hash functions improves achievable load factors, it also increases the number of choices when placing a pair. The simplest choice is to always pick the first bucket and only perform cuckooing when it is full. A more sophisticated approach would be to pick the least loaded bucket (i.e., power-of-two-choices hashing). This increases the cost of the insertion since it requires loading all possible buckets before deciding on which bucket to perform the insertion into. Iceberg hashing is a third strategy for placement.

5 GPU Implementation Details

We first describe how to support arbitrary bucket sizes, a feature common to the hash tables we build. Next we discuss specific implementation details for insertion and query using two probing schemes. Our implementation optimizes for 32-bit keys and values. We discuss how to extend our implementation to different sizes in Section 5.4. We use the largest number an unsigned integer can hold as our sentinel key (or value).

5.1 Supporting arbitrary bucket sizes In a bucketed hash table GPU implementation, we balance two considerations: larger buckets reduce the number of bucket accesses required for queries and insertions, but larger buckets are also more complex to search and manage. Nonetheless, a larger bucket is a sensible idea for a GPU implementation for the following two reasons: 1. The cost of accessing a single word in a GPU cache line, to first order, is the same as accessing the entire cache line. Thus, if a bucket size is no larger than a cache line, we can access the entire bucket with the same memory cost as accessing a single pair. 2. Even while we incur more work per query or insertion with a larger bucket size, we are still memory-bound, so the additional work does not affect our performance.

GPU implementation details Our implementation uses CUDA's cooperative groups to achieve flexible control over the bucket size while taking advantage of the tile-wide communication intrinsics between threads. The challenge with buckets of size $b > 1$ is to efficiently implement operations involving the entire bucket (e.g., search for a particular key in a bucket) without the use of additional metadata. The cooperative groups API, specifically the `tiled_partition` explicit groups, offers communication within the tile (e.g., using `ballot` or `shfl`) to address this challenge.

We first discuss the implementation details for performing insertion in a tile-cooperative fashion. Performing insertion in a cooperative fashion allows serializing all operations within a tile and then each serial operation can read a bucket in a coalesced fashion (a generalized version of Ashkiani's warp-cooperative work sharing strategy [3]). Listing 1 shows the construction of a tile in a cooperative insertion function.

In lines 2–3 we divide a thread block into tiles with a size equal to the bucket size (known at compile time).

```
struct bucket{
  bucket(pair_type* ptr, tile_type& tile) : ptr_(ptr), tile_(tile){
    lane_pair_ = ptr[tile_.thread_rank()];
  }
  int compute_load() {
    auto load_bitmap = tile_.ballot(lane_pair_.key == EMPTY_KEY);
    return __popc(load_bitmap);
  }
  value_type find_key_value(const key_type key) {
    bool key_exist = (key == lane_pair_.key);
    int key_lane = __ffs(tile.ballot(key_exist));
    if(key_lane == 0) return EMPTY_VALUE;
    return tile_.shfl(lane_pair_.value, key_lane - 1);
  }
  bool cas_at_location(pair_type& pair, int location) {
    pair_type old_pair;
    if(tile_.thread_rank() == elected_lane_)
      old_pair = atomicCAS(ptr_ + location, EMPTY_PAIR, pair);
    return tile_.shfl(old_pair, elected_lane_) == EMPTY_PAIR;
  }
  pair_type exch_at_location(pair_type& pair, int location) {
    pair_type old_pair;
    if(tile_.thread_rank() == elected_lane_)
      old_pair = atomicExch(ptr_ + location, pair);
    return tile_.shfl(old_pair, elected_lane_);
  }
  private:
  pair_type* ptr_; pair_type lane_pair_;
  tile_type tile_; int elected_lane_ = 0;}
```

Listing 2: Tile-sized bucket implementation.

We utilize the tile and perform the various operations to either query the rank of the thread in the tile (line 4) or evaluate a predicate to build a queue of all the threads that will perform an operation later (line 6). Line 7 finds the next item in the queue. We broadcast the next pair in the queue across all threads in the tile (line 8). In line 9, the pair is inserted into the hash table. Finally, in line 10 only the current lane removes the item from the queue by setting `to_insert` to `false` (line 11) and writes back its result to its own register (line 12). Performing a query is similar, with the only differences that (a) the result contains the value of the key and (b) calling the find operation instead of the insertion.

Now we discuss the common operations on a bucket shown in Listing 2. In line 2 we construct the bucket structure using a pointer to the bucket and the cooperative-group tile. To compute the load of the bucket (line 5) we perform a `ballot` operation to populate an unsigned integer with bits (for each thread in the tile) that are set whenever the bucket's key is not a sentinel key. Using the populated integer, we can count the number of valid keys in the bucket using the population count (`__popc`) intrinsic. Additionally, we can use the load to find the next available insertion location in the bucket.

To perform a query (line 9), we first compare the query key with the keys present in the bucket followed up by a `ballot` instruction to compact the comparison result into a single unsigned integer. By finding the first-set bit in the the result (using an `__ffs`) we can determine the location of the key. Note that the index of the least-significant bit is 1 while a result of zero indicates that the key is not present. Finally, we broadcast the result to all tile threads.

One way to perform insertion is to swap an empty sentinel pair with a new pair (line 15) using the compare-and-swap (CAS) instruction. Only one thread performs the CAS operation and attempts to swap an empty sentinel pair with the new pair, then we broadcast the swapped-out pair to all threads in the tile. Only when the swapped out pair is indeed an empty pair we can conclude that the insertion succeeded. Another way to perform insertion is to exchange an old pair with the new one (line 21). Similarly, only one thread performs the exchange operation then we broadcast the exchanged pair to all threads in the tile.

Alternative efficient hashing techniques, utilizing different probing schemes, are straightforward to implement when based on Listing 2. We draw your attention to line 3. This line, much more than any other in our implementation, captures a crucial piece of functionality common to all hash tables we explore—this coalesced memory read equates to the number of probes during insert or query operations (see Figure 2, 4, and Section 7 for more detail).

```
bool insert(tile_type tile, pair_type pair, pair_type* table){
 auto bucket_id = hf0(pair.key) % num_buckets;
 uint32_t cuckoo_counter = 0;
 random_number_generator rng(0, bucket_size);
 bool success = false;
 do{
  auto cur_bucket = bucket(table + bucket_size * bucket_id, tile);
  auto load = cur_bucket.compute_load();
  if(load == bucket_size){
   if(cuckoo_counter == max_cuckoo_chains)
    return false;
   pair = cur_bucket.exch_at_location(pair, rng());
   auto prev_bucket = bucket;
   auto bucket0 = hf0(pair.key) % num_buckets;
   auto bucket1 = hf1(pair.key) % num_buckets;
   auto bucket2 = hf2(pair.key) % num_buckets;
   bucket = bucket0;
   bucket = prev_bucket == bucket1 ? bucket2 : bucket;
   bucket = prev_bucket == bucket0 ? bucket1 : bucket;
   cuckoo_counter++;
  }else
   success = cur_bucket.cas_at_location(pair,load);
 }while(!success);
 return false;}
```

Listing 3: Bucketed cuckoo hash table insertion.

```
value_type find(key_type key, pair_type* table){
 value_type result = EMPTY_VALUE;
 int num_hash_functions = 3;
 for(int i = 1; i < num_hash_functions && result == EMPTY_VALUE; i++){
    uint32_t bucket_id;
    if(i == 1) bucket_id = hf0(key) % num_buckets;
    else if (i == 2) bucket_id = hf1(key) % num_buckets;
    else bucket_id = hf2(key) % num_buckets;
    auto cur_bucket = bucket(table + bucket_size * bucket_id, tile);
    auto result = cur_bucket.find_key_value(key);
    if(result == EMPTY_VALUE && i != num_hash_functions){
        if(cur_bucket.compute_load() != bucket_size) break;
    }
 }
 return result;}
```

Listing 4: Bucketed cuckoo hash table find.

5.2 Bucketed cuckoo hash table

Insertion Listing 3 shows the implementation details for insertion. We begin by computing the first bucket we will attempt to insert into (line 2). Then, we compute the bucket's load, and if the bucket is full and we did not reach the maximum number of cuckoo chains, we exchange our new pair with a random pair from the bucket (line 12), then we increment the counter of the number of swaps we performed (line 20). We discussed the case when the first bucket was full, and we needed to perform cuckooing; now, if the bucket has an empty spot, we try to insert the pair into it. We attempt to insert the key-value pair using an atomic compare-and-swap (line 22). If insertion does not succeed, we attempt the insertion again until we either succeed or reach the maximum number of cuckoo chains (line 11).

Find Listing 4 shows our implementation for the find operation. To evaluate the query we serially inspect the three buckets associated with the three hash functions. We evaluate the hash function (lines 6, 7, or 8) then load the bucket (line 9). We look up the key's value (if it exists) in the bucket (line 10), and we also check if the bucket has an empty key (line 12). We terminate after we inspect all three hash functions or when one of the buckets contains an empty key (early exit).

5.3 Bucketed iceberg hash table

Insertion Listing 5 shows the implementation for insertion. We first begin by computing the primary bucket and its load (lines 2 and 5). If the load is larger than the threshold, we check the two secondary hash functions and their buckets. We calculate the buckets' indices and their load (lines 7 to 12). If the two secondary buckets are full, we insert the key into the primary bucket; otherwise, we insert it into the least loaded bucket (lines 14 and 15). Insertion fails if all three buckets are full (line 18). We attempt insertion (line 20) and repeat the process upon failure.

```
bool insert(tile_type tile, pair_type pair, pair_type* table){
  auto primary_bucket_id = hfp(pair.key) % num_buckets;
  while(true){
    auto bucket = bucket(table + primary_bucket_id * bucket_size, tile);
    auto load = bucket.compute_load();
    if(load > threshold){
      auto bucket0_id = hf0(key) % num_buckets;
      auto bucket1_id = hf1(key) % num_buckets;
      auto bucket_0 = bucket(table + bucket0_id * bucket_size, tile);
      auto bucket_1 = bucket(table + bucket1_id * bucket_size, tile);
      auto load0 = bucket0.compute_load();
      auto load1 = bucket1.compute_load();
      if(load0 != bucket_size || load1 != bucket_size){
       bucket = load0 <= load1 ? bucket0 : bucket1;
       load = load0 <= load1 ? load0 : load1;
      }
    }
    if(load == bucket_size) return false;
    if(bucket.cas_at_location(pair, load))
      return true;
  }
  return false;}
```

Listing 5: Iceberg insertion.

Find Find is similar to BCHT's find; however, in IHT, we can not perform an early exit because insertion could have taken place in any of the three buckets.

5.4 Beyond 32-bit keys and values Our implementation can be easily extended to support arbitrary key/value sizes using `libcu++`.[5] However, the atomic objects are not lock-free when the object's size is greater than 8 bytes, as NVIDIA GPUs only provide atomic operations on sizes up to 64 bits. How can we extend our design beyond 8 bytes?

Bucket size Generally, we would like the bucket size to match the cache line size, which means that the number of available key-value slots per bucket will decrease for pairs with larger sizes (e.g., a cache-line-sized bucket will only contain eight entries when the key-value pair size is 16 bytes). As we discussed in Section 4.1 and as we shall see in Section 7, decreasing the number of elements per bucket reduces the achievable load factor and increases the required number of probes during insertion. Our average probe count analysis (Section 7.4) is agnostic to the key-value pair size and the memory system cache line size and it will guide our next decisions. For instance, if the goal is to achieve high load factors at the expense of insertion and lookup rates, then we will aim to keep the number of elements inside the bucket constant (i.e., using a 256-byte bucket when the pair size is 16 bytes).

Memory layout and atomic operations When the bucket size exceeds the cache line size, we can consider laying out the bucket using a structure-of-arrays (SoA) layout. In the SoA layout, the first cache line contains keys, followed by a cache line containing values. An SoA layout will facilitate a *coalesced* read to the keys part of the bucket, followed by another coalesced read of the values when necessary (e.g., when we are looking up the key's value). During insertion, the SoA layout requires performing back-to-back atomic operations to attempt exchanging a sentinel key and value to the desired pair. When the hash table is not stable (e.g., a cuckoo hash table), back-to-back atomics are not feasible, and atomically exchanging the key-value pair requires solutions such as double compare-and-swap (DCAS) [11].

We assumed 128-byte cache lines in the previous analysis and discussion. However, we expect our average probe analysis to continue to guide hash table designers when designing hash tables on architectures with different cache-line sizes.

6 High-Level Recommendations

The designer of a hash table may prioritize peak achievable load factor, insertion rate, query rate, or some combination thereof. Table 1 summarizes our recommendations. Next, we discuss the intuition behind these recommendations.

For the *highest load factor*, choose BCHT, followed by IHT then 1CHT. BCHT can move keys between

[5] `libcu++` (part of NVIDIA's CUDA toolkit) is a freestanding standard C++ library implementing CUDA equivalent of `std::atomic`, among other C++ constructs.

Method	Load factor	Insertion Probes	Query Probes	Stability
1CHT	0.88	≈ 2.8	up to 4	no
BCHT	0.98	≈ 1.8	up to 3	no
IHT	0.92	1 or 3	up to 3	yes

Table 2: Properties for different hash tables. 1CHT uses 4 hash functions while BCHT uses only 3 hash functions. IHT uses a primary hash function and two secondary hash functions.

buckets until it succeeds or reaches the maximum allowable cuckoo chain length, but IHT only has three possible locations for insertion. 1CHT has four locations but each location can only hold a single entry. Increasing the number of hash functions for any of these techniques yields higher load factors, but decreases the throughput of insertion and query operations.

For the *highest insertion throughput*, choose BCHT, followed by IHT then 1CHT. Bucketed techniques utilize the whole cache line (in contrast to 1CHT). IHT transfers more memory since it inspects the primary bucket and if its capacity exceeds the threshold, it inspects the two secondary buckets, which is more common and hence more expensive at high load factors.

For the *highest query throughput*, choose BCHT, followed by IHT then 1CHT. As with insertion throughput, bucketed techniques make use of the entire cache line. BCHT inspects the fewest buckets on average of all techniques.

For *stability*, choose IHT; it is the only hash table that supports it. Table 2 shows a summary of the different hash table characteristics.

7 Results

Now we analyze and compare in detail the three hash table configurations that we describe in this paper and CUDPP's 1CHT. Our figures of merit are query rate, build rate, and load factor. We evaluate our implementations on an NVIDIA TITAN V (Volta) GPU with 12 GB DRAM and an Intel Xeon E5-2637 CPU. Our code[6] is complied with CUDA 11.1. The GPU has a theoretical achievable DRAM bandwidth of 652.8 GiB/s. All results are averaged over 10 successful experiments with a maximum of 50 failures unless stated otherwise. We use 32-bit keys (and values) that are randomly generated and uniformly distributed. We use the hash function $h(k; a, b) = ((ak + b) \mod p) \mod L$, where a and b are randomly generated integers and p is a random prime number and L is the number of buckets in the hash table.

We use the following variations of hash tables in our evaluation: **1CHT** our implementation of a cuckoo hash table with a bucket size of one, faithful to Alcantara et al. [2] and their CUDPP implementation [10]. We use four hash functions like CUDPP's default configuration. CUDPP's implementation is the most common performance baseline in the literature for GPU static hash tables. **BCHT** Bucketed cuckoo hash table where we increase b, the number of items per bucket, to larger values. **IHT** Iceberg hash table, parameterized by the number of items per bucket b and the threshold t.

Summary By showing that the number of probes is the dominant factor influencing our implementations' performance, we demonstrate that our generic hash table implementation is efficient and successful in decoupling the probing scheme from the GPU-specific implementation details (Section 7.5). Our average probe count analysis is hardware-agnostic, providing a guideline for choosing the bucket size for different probing schemes on current and future hardware (Section 7.4). Sweeping through the parameter space supports and matches our recommendations (Sections 7.1 and 7.2).

7.1 Configuring the parameters We first begin our performance analysis by determining the optimal parameters for each hash table. For each technique, we evaluate the query and insertion throughput for two scenarios: when the number of keys is constant (50M keys) with varying load factors; and a constant load factor (0.8 or 0.9) and a different number of keys. We also evaluate the query performance for three different fractions of positive queries (fraction of lookup keys that exist in the hash table): 100%, 50%, and 0%. Since our implementation uses cooperative groups as its main building block, the bucket size is always a power-of-two

[6] Our implementation is available at `https://github.com/owensgroup/BGHT`.

Fixed parameter	b	Insert	% of queries present in hash table		
			100%	50%	0%
50M keys	1	735.84	2235	1884.83	1496.21
	8	1150.60	3396.57	2645.33	2127.42
	16	**1273.86**	**3642.61**	**2840.57**	**2313.17**
	32	1140.14	2166.94	1739.29	1450.70
0.8 load factor	1	772.08	2313.14	1945.56	1530.62
	8	1309.63	3858.59	3206.12	2695.23
	16	**1425.00**	**4033.52**	**3546.12**	**3155.51**
	32	1243.22	2334.66	2181.45	2050.83
0.9 load factor	1	580.73	1987.13	1646.54	1323.49
	8	1193.83	3502.35	2643.75	2081.43
	16	**1344.38**	**3799.58**	**2921.74**	**2359.95**
	32	1206.40	2260.30	1839.16	1550.71

(a) BCHT.

Fixed parameter	b	Insert	% of queries present in hash table		
			100%	50%	0%
50M keys	16	**1297.18**	**3392.59**	**1922.64**	**1436.26**
	32	1087.01	1946.14	1088.09	803.85
0.8 load factor	16	**1319.87**	**3350.79**	**1941.20**	**1485.75**
	32	1145.62	2042.45	1131.60	825.79
0.9 load factor	32	**1073.92**	**1860.21**	**1075.96**	**829.85**

(b) IHT.

Table 3: Average insertion and query throughput (Mkey/s).

number that varies up to the warp size (i.e., 32). We avoid using larger bucket sizes (we expect they will not be performance-competitive) as the functionality of cooperative groups is more limited and it has the additional cost of using shared memory for communicating between threads in the tile when performing tile-wide operations.

7.1.1 BCHT Figure 3a shows the throughput for the fixed number of keys scenario while varying the load factor. It shows that in all of the scenarios, a BCHT with a bucket size of 16 outperforms all the alternative bucket sizes. These results match our expectation since from a hardware perspective, we strongly prefer that the bucket size matches the GPU's cache line size, i.e., 128 bytes or 16 8-byte key-value pairs. For either query or insertion, the main factor affecting the performance is the load factor. Higher load factors yield lower throughput for BCHT. This is due to the increase in the number of cuckoo chains (or evictions) we need to perform per key insertion. An additional factor that decreases the performance of query throughput is the fraction of query keys that exist in the hash table. Negative queries require inspecting more than one bucket, especially at high load factors (i.e., when most buckets are full), leading to a steeper decrease in performance for scenarios where more queries do not find a key.

Figure 3a shows a comparison between the different BCHT bucket sizes for two different load factors (0.8 and 0.9). In it, we see that when the hash table is large enough to exceed the L2 cache size (here, our L2 cache can hold up to 589k pairs), the throughput is almost constant for all the bucket sizes. Table 3a provides an averaged summary for the different bucket sizes' results.

Figure 3a also shows the average number of probes for the different hash table operations. For all operations, we see a clear advantage when using bucketed hash tables over 1CHT. For example, the average probe count for insertion drops from 2.75 to 1.23 when the bucket size increases from one to eight (at a load factor of 0.9); increasing the bucket size beyond eight yields minimal improvement resulting in an average probe count of {1.11, 1.05} for bucket sizes {16, 32}. However, as we discussed earlier, a bucket size of 16 matches the GPU's cache line

size, yielding better throughput. The number of probes per query is bounded by the number of hash functions for any bucket size. In the best-case scenario (i.e., when a query finds the key in the first bucket), the number of probes will be close to one. When all queries are positive, we see that bucketed techniques require up to 1.5 probes per key at load factors as high as 0.98. However, in the worst-case scenario (i.e., high load factor and negative query), the query will load all the possible buckets. For example, when the bucket size is 16 and at a load factor of 0.99, the average number of probes per query is 2.8.

7.1.2 IHT We experiment with the parameter space of IHT by varying both the threshold t and bucket size b. We perform the sweep of parameters for the first experiment where the number of keys is fixed in Figure 3b. We find that IHT requires larger bucket sizes to achieve high load factors; specifically, bucket sizes of {32, 16, 8} achieve load factors up to {0.93, 0.86, 0.7}, respectively. In general, as the load factor increases, more insertions (or queries) require accessing the two secondary buckets, which leads to a decrease in performance. During queries, IHT requires inspecting all three possible buckets unless the lookup operation finds the key in the first or second bucket (note that when all the queries are negative, the query needs to inspect all three buckets).

We use bucket sizes of 16 and 32 throughout our experiments as they achieve comparable load factors to our other hash tables implementations. For both bucket sizes we find that as the threshold increases, the achieved throughput, for either insertion or queries, also increases. As the threshold increases, more insertions occur into the primary bucket, thus avoiding loading the secondary buckets and saving bandwidth. The best-performing threshold for both bucket sizes is around 80% of the bucket size. Table 3b provides a summary of our results for IHT.

Figure 3b shows the average number of probes for the different IHT bucket sizes and thresholds. We find that as the threshold increases, the number of probes drops. For example, when inserting into an IHT at a load factor of 0.86 and a bucket size of 16, the average-probe counts are {1.89, 1.49} at thresholds {60%, 80%}. Increasing the threshold means performing more insertions in the primary bucket without reading the two secondary buckets, which explains the drop in the probes count. Similar to our earlier observation in BCHT's average-probe count, increasing the bucket size from 16 to 32 yields minimal improvement to the probe count. When all queries are positive, and at a load factor of 0.86, IHT with a bucket size of 16 and threshold of {60%, 80%} performs {1.6, 1.33} probes per key. As the percentage of positive queries drops, queries inspect more secondary buckets, inspecting all three possible buckets for the different load factors and bucket sizes.

7.2 Comparison across implementations Given the previous results, we choose the following three configurations of hash tables for comparisons: (1) baseline 1CHT, (2) BCHT with bucket size of 16, and (3) IHT with bucket size of 32 and threshold 80%.

7.2.1 Varying the load factor when the number of keys is constant Figure 4 shows a comparison between the recommended configurations. In it, we see that BCHT offers the best performance for both insertion and queries. BCHT has the advantage that it can use a bucket size of 16 while achieving high load factors (i.e., each bucket is a single cache line), while IHT uses a bucket size of 32 to achieve comparable load factors (i.e., each bucket is two cache lines). On the other hand, 1CHT only uses 8 bytes of data from a 128B cache line.

With respect to insertion performance, the required memory traffic for IHT (2 or 6 cache lines in the best and worst scenarios, respectively) lower its performance compared to BCHT (1 to 3 cache lines). 1CHT offers worse performance than both BCHT and IHT. On average the insertion throughputs of {BCHT, IHT, 1CHT} are {1273.86, 1087.01, 735.84} Mkey/s.

For all-positive queries (and in the best case scenario), BCHT and 1CHT only require loading a single cache line, while IHT loads two cache lines. This results in 1CHT throughput's exceeding IHT's. BCHT's throughput continues to exceed all other alternatives.

As the ratio of positive queries goes down, each technique must check additional buckets. In the worst-case scenario, BCHT loads 3 cache lines; 1CHT loads 4 cache lines; and IHT loads 6 cache lines. This result in the performance ranking of BCHT (highest), 1CHT, and IHT (lowest). In summary the average query throughputs of {BCHT, IHT, 1CHT} are {3642.61, 1946.14, 2235}, {2840.57, 1088.09, 1884.83}, and {2313.17, 803.85, 1496.21} Mkey/s for all-positive, 50% positive, and all-negative queries, respectively.

Note that at low load factors, the performance of IHT is similar to BCHT when the bucket size is the same. However, at high load factors (bucket sizes exceed the threshold), IHT uses the power of two strategy, which

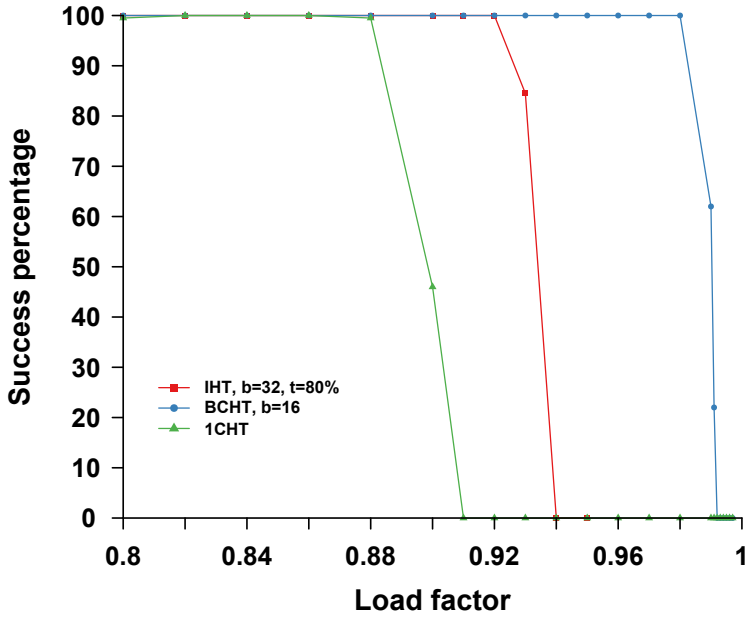


Figure 1: Comparison of success rates across the recommended hash table variants. The input is 50M keys, run over 200 experiments.

requires loading two additional buckets. In contrast, BCHT loads just enough buckets to perform cuckooing.

7.2.2 Varying the input size We now consider the scenario when we vary the input number of keys while the load factor is constant. Figure 4 shows the performance of the hash tables for load factors 0.8 and 0.9. Similar to the previous section, BCHT offers the best performance followed by IHT, then 1CHT. In summary, at 0.8 load factor, the insertion throughput of {BCHT, IHT, 1CHT} is {1425, 1145.62, 772.08} Mkey/s. For queries, the throughput of {BCHT, IHT, 1CHT} is {4033.52, 2042.45, 2313.14}, {3546.12, 1131.6, 1945.56}, and {3155.51, 825.79, 1530.62} for all-positive, 50% positive and all-negative queries, respectively.

7.3 Success rate All the hash tables we discussed could fail during building. To evaluate the success rate of each technique, we build each hash table using the same set of input keys in 200 different runs. For each run, we use different hash functions (with random constants) and we record the success or failure of each run. Figure 1 shows the achievable load factors when the success percentage is around 99%. BCHT achieves the highest load factor of 0.98, followed by IHT with a load factor of 0.91, then 1CHT with a load factor of 0.88.

7.4 Analysis of the average probes count One principal hypothesis is that *the main factor influencing the performance of insertion (or query) in a hash table is the number of probes each operation performs to complete an insertion (or query).* We prove this hypothesis by instrumenting our implementation to count the average number of probes per key each operation needs to perform and correlating that count with performance, and elaborate on it below in Section 7.5. Note that buckets have different sizes in each implementation, but here the probe is equivalent to reading a bucket regardless of its size (reading two buckets instead of one does not necessarily result in 2× slower performance). Moreover, insertion in a 1CHT requires a single atomic exchange, whereas, in bucketed techniques, two operations are required: (1) reading a bucket, (2) performing an atomic compare-and-swap operation (or an atomic exchange). We will investigate these effects in the next section. By only considering the average number of probes, we can understand the performance of each hash table in a hardware-independent way.

Figure 4 shows the average number of probes for insertion and query using different positive query ratios in a table of size 50 million keys. We build each variant for different load factors ranging from 0.6 up to the maximum load factor we can achieve.

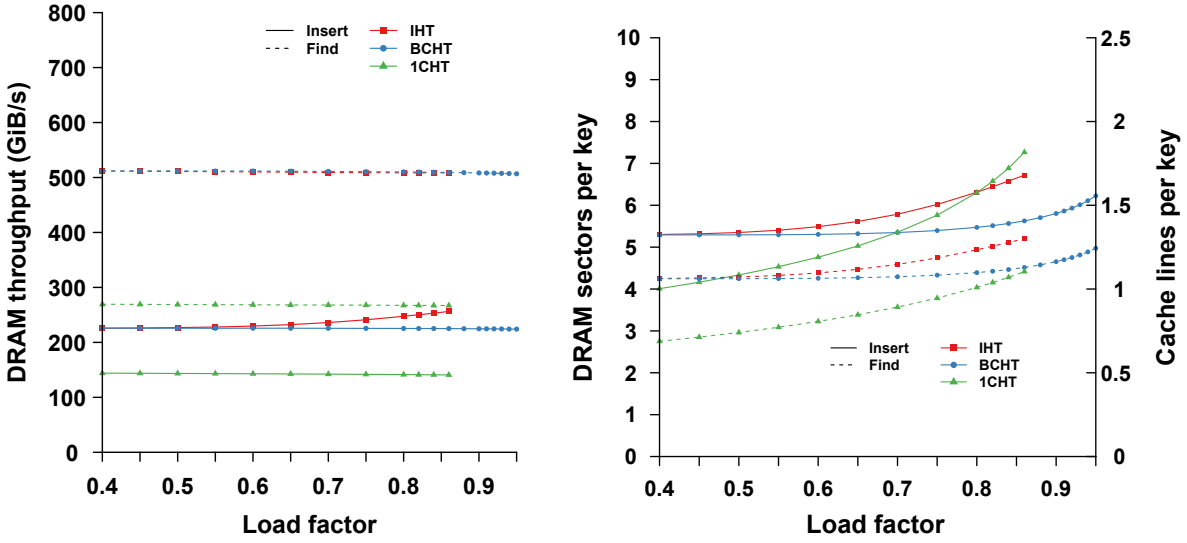


Figure 2: Achieved throughput and the average number of sectors per key (a sector is 32 bytes) for IHT and BCHT (each with a bucket of 16 and IHT uses a threshold of 13 keys per primary bucket). The number of keys is 200M keys, and all lookups exist in the hash table. Notice the relationship between the average number of probes per key (Figure 4) and the average number of sectors per key. A single probe into a 128-byte bucket corresponds to loading four sectors. Also, notice the inverse relationship between the average number of sectors per key and the average insertion or query throughput in Figure 4.

Insertion We find that BCHT enjoys a very low average number of probes per key (up to 1.43 at load factor 0.99), which will be tough to beat for any competing method. IHT, similarly, has a low number of probes (up to 1.46 at load factor 0.92), mainly because it only requires reading the primary bucket. Furthermore, IHT performs two additional buckets reads if the primary bucket exceeds the threshold. In the best-case scenario for IHT, a threshold% of the total keys will only require reading the primary bucket, whereas (100 − threshold)% will read the primary and two secondary buckets. 1CHT with 4 hash functions reads up to 2.75 buckets per insertion at 0.9 load factor.

Queries, all in table When all queries exist in the hash table, the number of probes drops for all techniques. Unlike insertions, queries do not have atomic operations that could fail, and for all hash tables, queries do not require accessing all the possible buckets. For instance, BCHT, IHT, and 1CHT's average probe count drops to 1.38, 1.32, and 2.26 at the highest achievable load factors, respectively.

Queries, not all in table However, as the ratio of positive queries drops (i.e., more keys do not exist in the table), the average number of probes increases and it is possible that they can reach up to the number of hash functions. Note that unlike 1CHT and BCHT, IHT can't perform an early exit if one of the possible buckets contains an empty spot. At a 0% positive query ratio, {BCHT, IHT, 1CHT} reach up to {2.80, 3, 3.44} probes per query.

The results that we showed in the previous sections validate and complement our high-level recommendations in Section 6.

7.5 Throughput and memory transfers analysis The number of "probes" is a (theoretical) proxy for what matters in practice and our implementation: the number of bytes transferred while performing a data structure operation. To evaluate our implementations and compare them to the ideal speed-of-light for the GPU, we measure two different metrics: (1) achieved DRAM throughput and (2) the average number of DRAM sectors (defined below) per key (insertion or lookup). Figure 2 shows the profiler results for these analyses. For this experiment, we use a large number of keys (200M keys) to avoid any caching effects, and for fairness, we use the same bucket size across the different hash tables.

Achieved throughput. For all bucketed hash tables, lookups achieve more than 500 GiB/s (over 75.5% of

the achievable bandwidth). 1CHT only achieves around 267 GiB/s. Insertion achieves lower throughput (between 220 and 300 GiB/s) in bucketed hash tables and 141 GiB/s in 1CHT.

Memory transfers. To understand the slowdown for insertions compared to lookups, we measure the average number of DRAM *sector* accesses per key. Sectors—each constituting 1/4 of a cache line, i.e., 32 bytes—are how the profiler quantifies loads and stores.

For a query operation, the dominant factor in the number of sectors per key is the number of buckets a query needs to read to find a key. For example, when BCHT finds a key in the first possible bucket (i.e., one bucket is required to evaluate the query), we expect the number of sectors per key to be four (12 in the worst-case scenario of three buckets). Similarly, IHT will require four sectors in the best-case scenario and 12 in the worst-case scenario, respectively. 1CHT only requires several sectors between 2 and 8 (note that the access granularity between the DRAM and the L2 cache is 64 bytes, i.e., when an uncoalesced access reads a single sector, it will transfer two sectors from the DRAM to the L2 cache).

On the other hand, insertion in bucketed tables requires writing back the pair into the bucket using at least one atomic compare-and-swap operation (and, in some cases, an atomic exchange operation if the bucket is full or in 1CHT). Since we perform these atomic operations on a key-value pair of 8 bytes, the atomic operation will add at least one additional sector. We see this effect for both IHT and BCHT in Figure 2. Note that these atomic operations are divided into two operations: first, moving the required sector to the L2 cache, followed by the update and writeback operations. However, since we just fetched the bucket before the atomic operation, more than 90% of the time, the bucket is already in the L2 cache.

We note that although we are using sectors here as our unit, loading a sector from the DRAM has the same cost as loading a full cache line, thus an additional sector (specifically for writing back a pair into the bucket) costs the same as reading an entire cache line.

Our conclusion from this analysis is that we effectively utilize the DRAM bandwidth, and the main performance limiter is the number of sectors per key that a hash table operation must load (or store). Our analysis here closely matches the insertion (or query) throughput results in Figure 4.

8 Conclusion

The number of probes an insertion (or query) operation performs is the primary performance limiter in a hash table given an efficient design and implementation. We efficiently implemented three different variants of static hash tables on the GPU, where the main difference between each hash table variant is the probing scheme.

For memory-bound problems such as building and querying a hash table, an efficient design fully utilizes the memory bandwidth. Bucketed hash tables where the bucket size matches the hardware cache line achieves a hash table design that fits the GPU. Our results show that resolving collisions using cuckoo hashing outperforms alternate techniques on the GPU. With around 1.43 average probes per key at load factors as high as 0.99, bucketed cuckoo hash tables are the best all-around choice among all hash tables we studied.

Acknowledgments

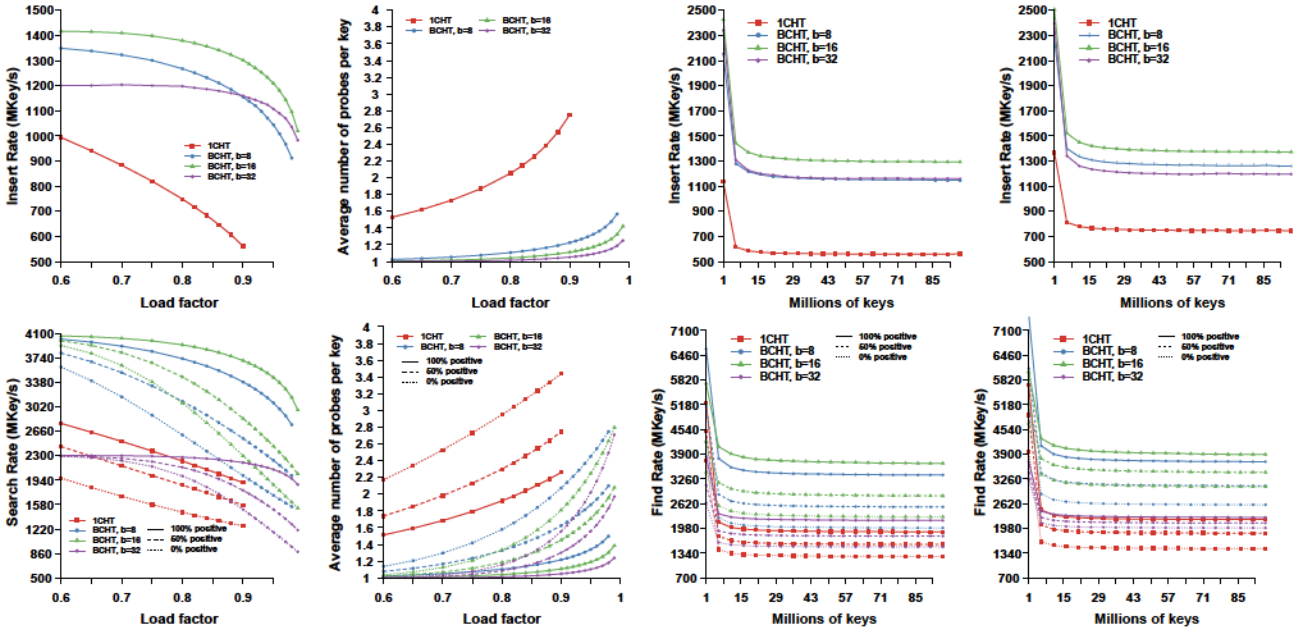
Thanks to Martin Dietzfelbinger, Lars Nyland, and Alex Conway for helpful technical suggestions.

The authors appreciate the research support of the National Science Foundation (awards # CCF-1637442 and # OAC-1740333), DARPA (AFRL awards # FA8650-18-2-7835 and # HR0011-18-3-0007), an Adobe Data Science Research Award, an NVIDIA AI Laboratory, and equipment donations from NVIDIA.

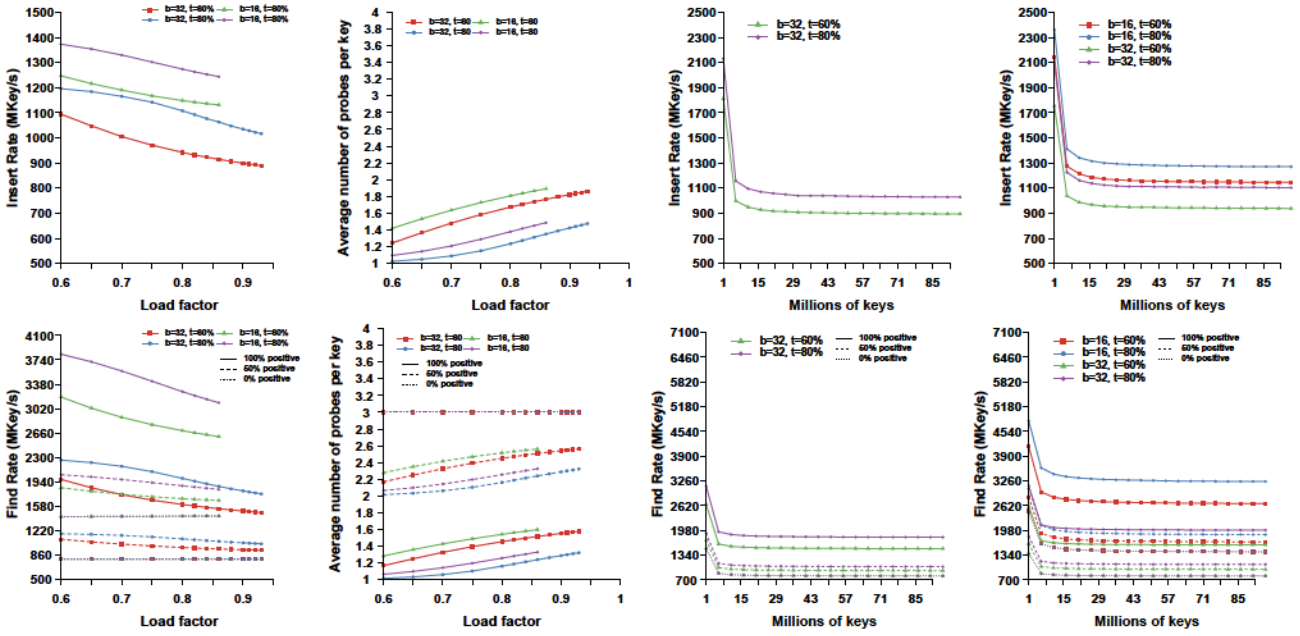
This material is based on research sponsored by the Air Force Research Lab (AFRL) and the Defense Advanced Research Projects Agency (DARPA). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Lab (AFRL) and the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, *Real-time parallel hashing on the GPU*, ACM Transactions on Graphics, 28 (2009), pp. 154:1–154:9, https://doi.org/10.1145/1661412.1618500, https://escholarship.org/uc/item/445536d6.



(a) BCHT



(b) IHT

Figure 3: Configuring the hash tables. Left to right: insertion (top) and find (bottom) throughput, the average number of probes (for 50M keys), and throughput for a different number of keys and load factors 0.9 and 0.8.

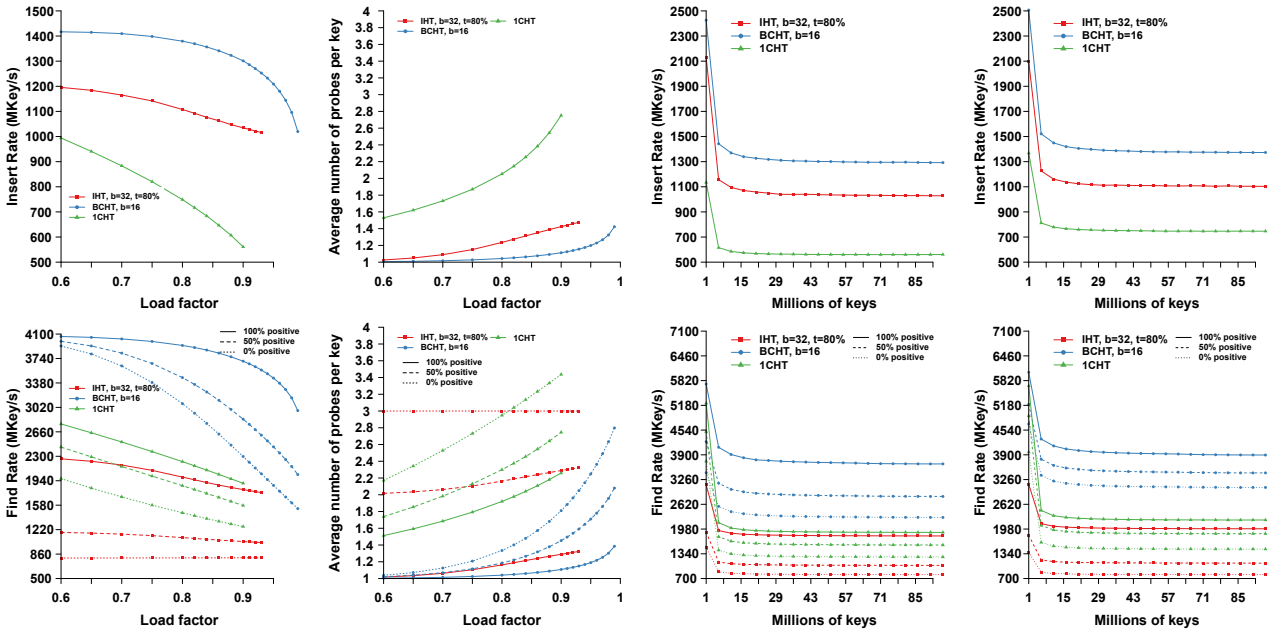


Figure 4: Hash tables recommendation. Left to right: insertion (top) and find (bottom) throughput, the average number of probes (for 50M keys), and throughput for a different number of keys and load factors 0.9 and 0.8.

[2] 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, W. W. Hwu, ed., vol. 2, Morgan Kaufmann, Oct. 2011, ch. 4, pp. 39–53, https://doi.org/10.1016/B978-0-12-385963-1.00004-6.

[3] S. Ashkiani, M. Farach-Colton, and J. D. Owens, *A dynamic hash table for the GPU*, in Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, May 2018, pp. 419–429, https://doi.org/10.1109/IPDPS.2018.00052, https://escholarship.org/uc/item/2p48q0zg.

[4] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, *Balanced allocations*, SIAM Journal on Computing, 29 (1999), pp. 180–200, https://doi.org/10.1137/s0097539795288490.

[5] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the 2009 ACM/IEEE Conference on Supercomputing, SC '09, Nov. 2009, pp. 18:1–18:11, https://doi.org/10.1145/1654059.1654078.

[6] M. A. Bender, A. Conway, M. Farach-Colton, W. Kuszmaul, and G. Tagliavini, *All-purpose hashing*, CoRR, (2021), https://arxiv.org/abs/2109.04548. arXiv:cs.DS/2109.04548v1.

[7] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink, *Tight thresholds for cuckoo hashing via XORSAT*, in International Colloquium on Automata, Languages and Programming, 2010, pp. 213–225, https://doi.org/10.1007/978-3-642-14165-2_19.

[8] Ú. Erlingsson, M. Manasse, and F. McSherry, *A cool and practical alternative to traditional hash tables*, in Proceedings of the 7th Workshop on Distributed Data and Structures, WDAS '06, Jan. 2006.

[9] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, *Space efficient hash tables with worst case constant access time*, Theory of Computing Systems, 38 (2005), pp. 229–248.

[10] M. Harris, J. D. Owens, S. Sengupta, Y. Zhang, and A. Davidson, *CUDPP: CUDA data parallel primitives library*. http://cudpp.github.io/, 2009–2017.

[11] T. L. Harris, K. Fraser, and I. A. Pratt, *A practical multi-word compare-and-swap operation*, in Proceedings of the 16th International Conference on Distributed Computing, DISC '02, Berlin, Heidelberg, 2002, Springer-Verlag, pp. 265–279, https://doi.org/10.5555/645959.676137.

[12] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, *Dissecting the NVIDIA Volta GPU architecture via microbenchmarking*, CoRR, (2018), https://arxiv.org/abs/1804.06826v1. arXiv:cs.DC/1804.06826v1.

[13] D. Jütnger, C. Hundt, and B. Schmidt, *WarpDrive: Massively parallel hashing on multi-GPU nodes*, in 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018, pp. 441–450, https://doi.org/10.1109/IPDPS.2018.00054.

[14] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, *WarpCore: A library for*

fast hash tables on GPUs, in 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC), Dec. 2020, pp. 11–20, https://doi.org/10.1109/HiPC50609.2020.00015.

[15] B. Lessley and H. Childs, *Data-parallel hashing techniques for GPU architectures*, IEEE Transactions on Parallel and Distributed Systems, 31 (2020), pp. 237–250, https://doi.org/10.1109/tpds.2019.2929768.

[16] Y. Li, Q. Zhu, Z. Lyu, Z. Huang, and J. Sun, *DyCuckoo: Dynamic hash tables on GPUs*, in 2021 IEEE 37th International Conference on Data Engineering (ICDE), IEEE Computer Society, Apr. 2021, pp. 744–755, https://doi.org/10.1109/ICDE51399.2021.00070, https://doi.ieeecomputersociety.org/10.1109/ICDE51399.2021.00070.

[17] M. Mitzenmacher, A. W. Richa, and R. Sitaraman, *The power of two random choices: A survey of techniques and results*, in Handbook of Randomized Computing, S. Rajasekaran, P. M. Pardalos, J. H. Reif, and J. Rolim, eds., vol. I of Combinatorial optimization, Kluwer Academic Publishers, June 2001, pp. 255–312.

[18] R. Pagh and F. F. Rodler, *Cuckoo hashing*, in 9th Annual European Symposium on Algorithms, vol. 2161 of Lecture Notes in Computer Science, Springer, Aug. 2001, pp. 121–133.

[19] R. Panigrahy, *Efficient hashing with lookups in two memory accesses*, in Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '05, USA, 2005, Society for Industrial and Applied Mathematics, p. 830–839.

[20] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, *Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores*, Proceedings of the VLDB Endowment, 8 (2015), pp. 1226–1237, https://doi.org/10.14778/2809974.2809984.