https://doi.org/10.2352/ISSN.2470-1173.2020.1.VDA-376

HashFight: A Platform-Portable Hash Table for Multi-Core and Many-Core Architectures

Brenton Lessley; Verb Surgical Inc.; Santa Clara, CA
Shaomeng Li; National Center for Atmospheric Research; Boulder, CO
Hank Childs; University of Oregon; Eugene, OR

Abstract

We introduce a new platform-portable hash table and collision-resolution approach, HashFight, for use in visualization and data analysis algorithms. Designed entirely in terms of data-parallel primitives (DPPs), HashFight is atomics-free and consists of a single code base that can be invoked across a diverse range of architectures. To evaluate its hashing performance, we compare the single-node insert and query throughput of HashFight to that of two best-in-class GPU and CPU hash table implementations, using several experimental configurations and factors. Overall, HashFight maintains competitive performance across both modern and older generation GPU and CPU devices, which differ in computational and memory abilities. In particular, HashFight achieves stable performance across all hash table sizes, and has leading query throughput for the largest sets of queries, while remaining within a factor of 1.5*X of the comparator GPU implementation on all smaller query sets. Moreover, HashFight performs better than the comparator CPU implementation across all configurations. Our findings reveal that our platform-agnostic implementation can perform as well as optimized, platform-specific implementations, which demonstrates the portable performance of our DPP-based design.*

Introduction

Many-core architectures are ubiquitous on modern supercomputers, involving diverse architectures from vendors such as NVIDIA, AMD, and Intel. As visualization software adapts to support this hardware, some efforts have focused on a hardware-agnostic approach. In effect, the idea is a "write once, use anywhere" code base that is future-proofed for evolving hardware.

The VTK-m project is a popular open source effort advocating a hardware-agnostic approach [26]. VTK-m operates by having algorithm developers utilize data-parallel primitives (DPPs) as building blocks. Instead of iterating over data with traditional "while" and "for" loops, developers use DPPs such as "map," "reduce," "gather," "scatter," etc. This approach reduces overall development time, since it allows for porting to a new architecture by writing architecture-specific code for DPPs (of which there are approximately 20) rather than for algorithms (of which there are currently dozens, with plans for hundreds more). Further, VTK-m offers services not found in approaches such as RAJA [16] and KOKKOS [8], since it manages mesh topology so that algorithm developers can access the data they need to operate on a given point or cell.

While VTK-m was initially designed for traditional visualization and data analysis operations such as isosurfacing, particle advection, and volume rendering, it also needs to support traditional data structures. With this work, we consider the topic of hashing, which is needed to efficiently implement algorithms such external facelist calculation [18], 3D volume reconstruction [27], and surface collision detection [17]. In effect, the research associated with this work is mapping a traditional data structure (hash table) into a visualization-centric framework (VTK-m). This is challenging because the mapping must be composed of DPPs and must avoid architecture-specific operations, such as hardware atomics and specialized memory accesses.

In this study, we address this challenge by introducing a new hash table and collision-handling technique called *HashFight*. Our approach is platform-portable and composed entirely of DPPs within the VTK-m code base. The collision-resolution routine does not use any hardware-specific locking mechanisms or hardware atomics (e.g. compare-and-swap) to synchronize hash collisions at a hash table address. Instead, our technique permits race conditions and "winner-takes-all" writes by parallel threads. Unsuccessful insertions by losing threads (i.e., insertions that were overwritten by a race condition) are accounted for in subsequent "hash-fighting" iterations.

Over a wide-variety of experimental configurations, HashFight achieves insertion and query throughput that is competitive to state-of-the-art CPU and GPU hash table implementations from open-source parallel algorithm libraries. For configurations with the smallest and largest hash table sizes on the GPU, HashFight performs comparably or better than the comparator GPU implementation. For configurations with the largest batches of unsigned integer query keys, HashFight attains leading GPU query throughput performance, while remaining within a factor of 1.5*X* of the comparator implementation on smaller batch sizes. The query throughput of HashFight is also shown to not be significantly affected by the percentage of unsuccessful queries (i.e., query keys that do not reside within the hash table). Moreover, HashFight achieves better insert and query throughput than the comparator CPU implementation across all experimental configurations.

Our findings demonstrate that HashFight is an effective solution for large hashing workloads and the need to achieve competitive portable performance across multiple platforms using a single DPP-based implementation. We demonstrate that the platform-agnostic design of HashFight performs comparably to the platform-specific GPU and CPU hash table implementations. This design also enables HashFight to be future-proof to new platforms as they emerge, without the need to re-implement HashFight for each new platform. We are in the process of contributing our HashFight implementation to the VTK-m library for open-source use in visualization algorithms.

Background

The following section reviews hashing fundamentals, summarizes existing parallel hashing implementations, describes data-parallel primitives which serve as the foundation of our tech-

nique, and surveys the use of hashing in visualization algorithms.

Searching via Hashing

Searching via hashing reorganizes an array of key-value pairs (e.g., unsigned integers keys and values) in an indexable hash table data structure H, such that only a constant number of direct, random-access lookups are needed per query on average. Each pair is inserted into the hash table at the index determined by a hash function, which generally maps keys (e.g., integers or spatial coordinate vectors) to arbitrary unsigned integers, modulo the size of the table. The hash function can take a variety of forms, but should be efficient to compute and distribute pairs as evenly as possible throughout the table, reducing the occurrence of *hash collisions*, i.e., two or more unique keys are hashed to the same location in the table.

For a larger number of pairs and smaller hash table size (i.e., fewer empty locations into which to hash), hash collisions become highly probable. Collisions are a result of both the choice of hash function and the size of the hash table. The hash table size can be described via a multiplicative factor of the number of keys $|K|$ to be inserted: $|H| = |K| \times f$, where f is a *load factor* that increases the available slots into which keys can be hashed. Typically, $1.0 < f \leq 2.0$, with larger factors reducing the chance of hash collisions, but requiring a larger memory allocation for the hash table.

Hash collisions are typically handled using one of two approaches: separate chaining and open-addressing. In separate chaining, colliding keys are stored in a linked list data structure at each hash table location. Query keys might need to make a number of probes through the linked list at its hashed location. In open-addressing, colliding keys are re-hashed to different locations in the hash table until they can be inserted into unoccupied locations. To query a key, the sequence of hashed locations, or probes, is followed in order until the key is found.

A variation of open-addressing, *cuckoo hashing*, allows currently-residing keys to be evicted at probe locations and then immediately re-hashed to different locations, making room for colliding keys. Each key is assigned two or more probe locations in the hash table, each computed with a different hash function. When a key is evicted from its location by a colliding key, it is inserted into the next location of its probe sequence. If another key already occupies this next location, then that key is evicted and inserted into its next probe location, and so on until no further collisions occur or a maximum number of evictions is made.

Parallel Hashing

Since the emergence of multi- and many-core CPUs and general-purpose computing on GPUs, a large body of research has investigated the design of parallel hashing techniques. In this parallel setting, multiple threads each simultaneously perform one or more hash table operations (e.g., insertion, deletion, or query), with one operation per assigned key. Typically, each concurrent operation at a hash table location must be explicitly synchronized to handle hash collisions and prevent race conditions. Our HashFight technique does not try to prevent race conditions, but instead uses them as a positive feature of the parallel hashing.

CPU-based Implementations

Single-node, CPU-based approaches have primarily focused on the design of dynamic, concurrent hash tables within shared memory [24, 29, 31]. These tables synchronize concurrent operations with either lock-based methods (e.g., mutexes or spin-locks) or lock-free hardware atomics (e.g., compare-and-swap (CAS)). Many of these tables are implemented as linked list data structures to support resizing and separate-chaining collision-resolution.

Notable open-source library implementations of CPU-based concurrent hash tables are provided within the Intel Thread Building Blocks (TBB) library and the Microsoft Parallel Patterns Library (PPL) [11, 25]. Both libraries include a *concurrent unordered map* hash table that supports lock-free (non-blocking) insertions, queries, and updates, using an underlying linked list, with CAS atomics over nodes [24, 31]. These unordered maps are extensible and hash key-value pairs into buckets, or segments of the linked list, similar to the unordered map provided by the C++ Standard Library. However, both maps do not support concurrent-safe deletions and the hash table size is expected to be a power of 2, which may affect the choice of hash function used.

GPU-based Techniques

GPUs are specifically designed for data-parallel processing, whereby a single instruction is performed over multiple data elements (SIMD) in parallel. The massive thread- and instruction-level parallelism available on a modern GPUs has motivated a large body of GPU-specific hashing techniques [17, 1, 2, 9, 10, 27, 5, 3, 7]. These techniques identify and address various performance challenges (e.g., irregular random memory accesses, thread control flow divergence, host-to-device data transfer, etc.) that arise while performing data-parallel hashing on the GPU. Many of these techniques maintain a static, on-device hash table with open-addressing collision-handling. These tables are efficient to construct and use fine-grained, hardware atomic primitives to synchronize table accesses and modifications.

The best-in-class open-source library implementation of GPU parallel hashing is based on cuckoo hashing and is packaged within the CUDA Data Parallel Primitives Library (CUDPP) [22], which contains top-performing algorithms and data structures written in NVIDIA CUDA. Introduced by Alcantara et al. [2], this general-purpose, cuckoo hash table supports a variable number of hash functions, hash table size, and maximum length of a probing sequence. The hash table resides in global memory and is constructed in parallel by threads each inserting key-value pairs into locations specified by the cuckoo hash functions. Insertions and evictions are synchronized using CAS atomic primitives, and each thread manages the re-insertion of any key-value pair that it evicts along the eviction chain, until a pair is finally placed in an empty table location. If a thread exceeds its maximum eviction, or probe, chain length during the insertion phase, then the table is reconstructed.

Heimel et al. [10] introduce an OpenCL-based hashing approach for databases that is most closest to our own approach, in the desire to implement hash tables on diverse architectures. However, uur work differs in that it considers this approach in the context of a visualization framework using DPPs.

A traditional benchmark for hashing key-value pairs, which we consider in this study, is to first sort the pairs by key and then perform queries via binary search. The CUDA Thrust library of data-parallel algorithms and data structures [28] provides fast and high-throughput data-parallel implementations of mergesort [30] and radix sort [23] for arrays of custom or numerical data types,

respectively. Additionally, Thrust includes a data-parallel, vectorized binary search primitive to efficiently search within a sorted array. The combination of sorting an array and searching within it has been widely-used as a benchmark for search-based tasks, particularly hashing [2, 4]. As a platform-portable library, Thrust also provides implementations of these algorithms and data structures in TBB and OpenMP for CPU execution.

Data-Parallel Primitives

Our HashFight approach is designed entirely in terms of data-parallel primitives (DPPs), which are building blocks that can be combined together to compose a larger algorithm. By providing highly-optimized implementations of each DPP for each platform architecture (e.g., implemented in CUDA for GPUs and TBB for CPUs), an algorithm composed of DPPs can be executed efficiently across multiple platforms with varying level of parallelism using only a single high-level code base, as seen in a variety of algorithms [19, 15, 20, 21]. This use of DPPs eliminates the combinatorial (cross-product) problem of implementing a different version of HashFight for each different platform of execution. Thus, only a single code base is needed to execute HashFight across varying platforms. As new platforms emerge, only the underlying DPPs of HashFight need to be re-implemented for these platforms, enabling a future-proof design. Example DPPs that are used in our approach include the following:

- *Map*: Applies an operation on all elements of the input array, storing the result in an output array of the same size, at the same index;
- *Reduce*: Applies a summary binary operation (e.g., summation or maximum) on all elements of an input array, yielding a single output value.
- *Gather*: Reads each value of an input data array into an index in an output array, as specified in the array of indices;
- *Scatter*: Writes each value of an input data array into an index in an output array, as specified in the array of indices;

Additional DPPs that we use in our algorithm include *Binary_Search*, *Sort*, and *Copy*.

Hashing in Visualization Algorithms

Hash tables are used in visualization algorithms to quickly store and query spatial data elements, and identify spatially-similar or duplicate elements. The following are recent examples from related research that apply hash tables to visualization tasks:

- For external facelist calculation of 3D unstructured meshes, a hash table is used to identify and discard the internal, or duplicate, cell faces that hash to the same value and are not needed in the output rendering or isosurface [18].
- For surface collision detection, hash tables are used to quickly detect intersecting geometrical surfaces that hash to the same table location [9].
- For real-time 3D volume reconstruction, hash tables are used to compactly store only the spatial regions, or voxel cells, in which surface data has been observed. This approach achieves similar performance as grid-based hierarchical data structures that store the entire 3D space [27].
- For merging coincident points in a space, hash tables can be used to quickly identify points that reside in, or hash to, the same spatial bin, as defined by a similarity metric. These spatially-coincident points are then merged together into a single, representative point [32].

Algorithm Overview and Discussion

The following section introduces our *HashFight* hashing approach, which is designed for data-parallel, platform-portable execution on both CPU and GPU platforms. In this approach, keys are inserted into and queried from a multi-level hash table via an iterative routine, which we refer to as *hash fighting*. During hash collisions at table locations, no explicit synchronization or hardware atomics are used. Instead, all colliding keys are inserted into the location in a winner-takes-all fashion, with the winner of the hash fight being the last key written into the location. Then, after all hash fights have completed, each thread checks its hash table location to see its key is currently residing in the table; that is, whether the key was the winner of its hash fight. All winning keys are marked as *inactive* and the remaining non-winning keys (marked as "*active*") proceed to the next round, or iteration, where they hash fight again, but into a smaller hash table. The size of the hash table at each iteration is equal to the number of active keys times the pre-specified hash table load factor. As in the first iteration, all active keys attempt to insert themselves into the subtable, and then check to see whether they won the fight into the table or need to remain active for another iteration of hash fighting.

The hash fighting routine continues until a specified number, or threshold, of keys have become inactive and are successfully placed into a location in one of the subtables. After this threshold, the remaining active key-value pairs are sorted and then contiguously inserted into a buffer region at the end of the hash table. Since the hash table size decreases each iteration, new sets of colliding keys may arise at hash locations, leading to a variable number of iterations necessary to insert all keys. In all of our experimental configurations, particularly those with a small hash table load factor, the number of iterations rarely exceeds 6 and never reaches 10. In terms of computation time, only the first one or two iterations account for most of the overall runtime.

For querying keys within the hash subtables, hash fighting is performed in a similar fashion to the insertion phase. Each iteration, all active keys hash to their location in the current subtable and then read the key-value pair residing in that location. If the residing key is equal to the query key, then the value is returned and the query key is set to inactive. If not equal, then the key remains active and performs a query again within the subtable of the subsequent iteration. Since each table location is initially populated with an "empty" key-value pair prior to the insertion phase, the residing key will remain empty if no input pairs were hashed and inserted into that location. For this latter case, a query key immediately returns an empty value and is set to inactive. The querying continues until all subtables have been searched, after which the remaining active keys each binary search for their query matches within the ending buffer region of the hash table. This region contains the keys from the insertion phase that were sorted and contiguously inserted.

Design Goals and Limitations

HashFight is designed for static hashing in which key-value pairs are first inserted into a table and subsequently queried. Thus, our approach does not currently support inter-mixed modifica-

```
1 void Insert(const uint32 *keys,
2        const uint32 *vals,
3        HashTable &ht, uint32 numKeys)
4 {
5   //All keys start active
6   uint32 activeKeys = numKeys;
7   uint8 *isActive[numKeys] = {1};
8
9   //Hash into subtables
10  uint32 tableStart = 0;
11  uint32 tableSize =
12    activeKeys*ht.loadFactor;
13
14  //Hash until a lower limit
15  //of active keys is reached.
16  while (activeKeys > HASHING_LIMIT) {
17    ht.subTableSizes.push_back(tableSize);
18
19    //Active keys hash into subtable
20    Fight(keys,vals, ht.entries,
21      isActive, tableStart, tableSize);
22
23    //Active keys see if they won.
24    //If a winner, a key is deactived.
25    CheckWinner(keys, ht.entries,
26      isActive, tableStart, tableSize);
27
28    tableStart = tableSize;
29    activeKeys = Reduce(isActive);
30    tableSize = activeKeys*ht.loadFactor;
31  }
32
33  //Sort remaining active keys and
34  //insert them into end of table.
35  uint32 *tempKeys, *tempVals;
36  CopyIf(keys, isActive, tempKeys);
37  CopyIf(vals, isActive, tempVals);
38  SortByKey(tempKeys, tempVals);
39  CopyToTable(tempKeys, tempVals,
40    ht.entries, tableStart, HASHING_LIMIT);
41 }
```

Listing 1. *Pseudocode of HashFight Insertion*

tions, such as deleting pairs or changing the values of keys. The hash table must be reconstructed in order to insert new entries or delete existing entries.

Due to its data-parallel processing, HashFight is best-suited to perform insertions and queries in large batches of keys, with each key being assigned to one of many threads. Thus, the execution of HashFight on small, or even individual, workloads of keys can result in sub-optimal performance, as the overhead time of the DPP kernel invocations exceeds the time needed to perform the actual hash table operation.

Listings 1 through 5 provide algorithm pseudocode for the HashFight insertion and query phases, along with the subroutines that compose the majority of the overall computation. These phases are discussed in more detail as follows.

```
1 void Fight(const uint32 *keys,
2        const uint32 *vals, uint64 *entries,
3        const uint8 *isActive,
4        uint32 tableStart, uint32 tableSize)
5 {
6   //Thread index
7   uint32 tid = getGlobalIndex();
8   if (isActive[tid]) {
9     uint32 key = keys[tid];
10    uint32 value = vals[tid];
11    uint64 entry =
12      ((uint64)(key) << 32)+value;
13    uint32 hash = Hash(key);
14    hash = (hash%tableSize)+tableStart;
15
16    //Non-atomic write
17    entries[hash] = entry;
18  }
19 }
```

Listing 2. *Pseudocode of the Fight Kernel*

Insertion Phase

In the `Insert` function, keys are hashed to random locations in the hash table and then inserted as pairs with their accompanying values (Listing 1). Given an array of 32-bit unsigned integer keys and an array of 32-bit unsigned integer values, each value is appended to its corresponding key to form a 64-bit unsigned integer pair. The hash table structure (Line 3) has pre-allocated sufficient memory to store these pairs, and each location in the table is initialized to an "empty" key-value pair `UINT_MAX << 32`. Initially, all keys are considered "active" and marked as such with an `isActive` array of bit indicators (Line 7).

Next, the hash fighting routine begins, consisting of multiple iterations (Line 16). Each iteration is assigned a separate contiguous partition, or subtable, of the larger hash table, into which the currently-active keys are hashed and inserted. This subtable is specified by a start location and a size (Line 10 – 11), the latter of which is equal to the number of active keys times a pre-specified hash table load factor. As the number of active keys decreases each iteration, the subtable sizes decrease proportionally.

Given a subtable, the active keys proceed to insert, or fight, themselves into the subtable.

In the `Fight` kernel (Listing 2), a thread is assigned to each key, and only threads with active keys (Line 8) perform computation. Each thread computes the hash value of its assigned key (Line 13) and takes the modulo of the subtable size to determine the write location (Line 14). We use a randomly-generated, non-cryptographic hash function that is a variant of the MurmurHash, which is efficient to compute and maintains strong hash properties to minimize collisions between 32-bit unsigned integers. Then, the threads concurrently write their key-value entries into the subtable (Line 17). During this scatter process, no locking mechanism or hardware atomics are used to synchronize simultaneous writes. Instead, race conditions are a fundamental and non-detrimental feature of handling hash collisions. Multiple threads may contend for the same location and overwrite each other (one after the other), but the final thread to write its pair into the location is declared the "winner" of the hash fight.

After all threads have finished hash fighting their keys, they

```
1 void CheckWinner(const uint32 *keys,
2         const uint64 *entries,
3         uint8 *isActive,
4         uint32 tableStart, uint32 tableSize)
5 {
6   //Thread index
7   uint32 tid = getGlobalIndex();
8
9   if (isActive[tid]) {
10    uint32 hash = Hash(keys[tid]);
11    hash = (hash%tableSize)+tableStart;
12    uint64 entry = entries[hash];
13    uint32 winningKey =
14      (uint32)(entry >> 32);
15    if (winningKey == keys[tid])
16      isActive[tid] = 0;
17  }
18 }
```

Listing 3. *Pseudocode of the CheckWinner Kernel*

return to the Insert function and proceed to determine whether they have "won" the fight, or successfully inserted the keys into the subtable (Listing 1, Line 28).

In the `CheckWinner` kernel (Listing 3), each thread recomputes the hash location of its assigned (and active) key (Line 14) and reads the currently-residing, or "winning," key-value pair at the location in the subtable (Line 12). If the thread's key is equal to the winning key (Line 15), then the thread won the hash fight and marks the key as inactive (Line 16). Otherwise, the key remains active and another thread will attempt to insert the key again in the next iteration of the insertion phase.

Finally, the threads return to the `Insert` function (Listing 1), where the local function variables are updated for the next iteration, including a Reduce data-parallel operation (Line 29) that counts and updates the number of active keys (or set bits).

The hash fighting continues until a minimum number of active keys remain; in this work, we used a minimum of 1 million keys (Line 16). After this point, all the active key-value pairs are sorted in ascending order by key and then contiguously written into a new subtable, or buffer region (Line 35 – 40). This ending sort and write procedure is meant to be faster and simpler to perform than hash fighting a small number of active keys into smaller-sized subtables, because with a small number of active keys, new collisions induce extra hash fight iterations and overhead of kernel invocations on rather small amounts of data.

Query Phase

In the `Query` function (Listing 4), a batch of input query keys (Line 1) is searched within the multiple subtables of a hash table (Line 2), which was previously constructed in the insertion phase (Listing 1). The result of this function is an output array of values (Line 3) corresponding to the query keys. Since the query keys are independent from the keys inserted into the hash table, a certain percentage of the query keys may not exist within any of the subtables, prompting an "empty," or failed, query value to be returned. Initially, all query values are set to an empty value of `UINT_MAX`, which is the largest 32-bit unsigned integer value.

Similar to the `Insert` function, all input query keys are

```
1 void Query(const uint32 *queryKeys,
2         const HashTable &ht,
3         uint32 *queryVals, uint32 numKeys)
4 {
5   //All query keys start active
6   uint32 activeKeys = numKeys;
7   uint8 *isActive[numKeys] = {1};
8
9   //Hash into subtables
10  uint32 iter = 0, tableStart = 0;
11  uint32 numTables=ht.subTableSizes.size();
12  uint32 tableSize=ht.subTableSizes[0];
13
14  while (iter < numTables) {
15    //Probe the subtable for the
16    //active query keys.
17    Probe(queryKeys, ht.entries, queryVals,
18      isActive, tableStart, tableSize);
19    tableStart = tableSize;
20    tableSize = ht.subTableSizes[iter++];
21  }
22
23  //Binary search the end of hash table
24  //for remaining active query keys.
25  BinarySearch(queryKeys, queryVals,
26               isActive, ht.entries,
27               tableStart, HASHING_LIMIT);
28 }
```

Listing 4. *Pseudocode of HashFight Querying*

marked as active (Lines 7) and local parameters are initialized to record the start index and size of a subtable (Lines 10 – 12). Then, `numTables` iterations of querying are conducted, with each iteration searching for active query keys within a different subtable. A data-parallel `Probe` kernel is invoked (Line 17) to look-up, or probe, the query keys within the subtable. Within this kernel (Listing 5), each query key is assigned to a thread (Line 7), which computes the hash table location of the key (if active) and reads the residing key-value pair at that location (Lines 10 – 13). If the query key is equal to residing key, then the residing value is returned as the query value and the query key is marked inactive (Lines 23, 28). If the table location contains the empty key-value pair, then no key was ever hashed to that location, and the query key is marked inactive (Line 22). Since the query value is set to empty by default, a query value does not need to be returned. Finally, if the residing key is neither empty nor equal to the query key, then the query key *may* exist within another subtable and, thus, remains active.

After all probing has completed, the threads return to the `Query` function, and local parameter values are updated for the next iteration of querying (Lines 19 – 20). Once all subtables have been searched via hashing, any remaining active query keys are binary-searched (Line 25) within the sorted buffer region of keys that were inserted at the end of the hash table during the insertion phase. This binary search is performed in data-parallel fashion by threads assigned to the remaining query keys. As in the `Probe` kernel, if the query key is not found, then an empty query value is returned; otherwise, the residing value in the table is returned.

```
1 void Probe(const uint32 *keys,
2         const uint64 *entries, uint32 *vals,
3         uint8 *isActive, uint32 tableStart,
4         uint32 tableSize)
5 {
6   //Thread index
7   uint32 tid = getGlobalIndex();
8
9   if (isActive[tid]) {
10    uint32 queryKey = keys[tid];
11    uint32 hash = Hash(queryKey);
12    hash = (hash%tableSize)+tableStart;
13    uint64 entry = entries[hash];
14    uint32 residingKey =
15      (uint32)(entry >> 32);
16
17    bool isEqual = queryKey==residingKey;
18    bool isEmpty = residingKey==UINT_MAX;
19
20    //If keys match or residing key is
21    //empty, then deactivate query key.
22    if (isEqual || isEmpty)
23      isActive[tid] = 0;
24
25    //Query is successful, so return
26    //the residing value.
27    if (isEqual)
28      vals[tid] = (uint32)entry;
29  }
30 }
```

Listing 5. *Pseudocode of the Probe Kernel*

Peak Memory Footprint

Assuming 4-byte (32-bit) keys and values, and 8-byte (64-bit) hash table entries, the peak CPU memory allocation (in bytes) required for both the insertion and querying phases is approximated by the following equation:

$$mem_{cpu} = (9 + 12.6f)|K| + 8|Q| + 8|B|, \quad (1)$$

where f is the hash table load factor, $|K|$ is the number of input keys, $|Q|$ is the number of query keys, and B is the buffer region at the end of the hash table (Listing 1, Line 43), whose size $|B|$ is set equal to the number of active keys that are sorted and copied directly into the buffer. Note that $|K|$ dominates this memory footprint, since $|B| << |K|$.

On GPU devices, the memory allocated on-device is released following the completion of each phase and kernel, except for the hash table. Since copies of the keys and values are inserted into the hash table as pairs, the original arrays can be released from GPU memory following the insertion phase. Then, during the querying phase, the query keys and values arrays must be transferred into GPU memory. Due to the `isActive`, `tempKeys`, and `tempVals` arrays (Listing 1), the insertion phase has a peak GPU memory usage of (in bytes):

$$mem_{gpu} = (9 + 12.6f)|K| + 8|B|, \quad (2)$$

which is similar to mem_{cpu} minus the allocation for the query keys and values arrays.

Experimental Overview

In this study, we assess the performance of HashFight across several different factors, comparing its performance to that of best-in-class comparator implementations. Our primary measure of hashing performance is the throughput of insertions and queries, which is calculated as the number of key-value pairs inserted or queried per second. The different experimental factors are outlined as follows, each factor consisting of multiple options.

- Algorithm (5 options)
- Platform (3 options)
- Dataset size (29 options)
- Hash table load factor (10 options)
- Query failure rate (10 options)

Since some of the configurations are not compatible together, We do not test the cross product of all configurations. For example, some algorithms cannot be executed on a CPU platform, and several dataset size and load factor combinations would exceed available on-device memory of certain GPU platforms. The details of each factor and configuration are discussed in the following subsections.

Algorithms

We compare the performance of HashFight with that of four different parallel hashing and search-based implementations from well-known open-source libraries:

- **HashFight** (CPU+GPU): DPP-based implementation with a single code base for both CPU and GPU platforms.
- **Thrust-Sort/Search** (CPU): Quick sort and vectorized binary search implementations written in TBB and contained within the Thrust library.
- **CUDPP** (GPU): Cuckoo hash table implementation written in CUDA and packaged within the CUDPP library.
- **Thrust-Sort/Search** (GPU): Radix sort and vectorized binary search implementations written in CUDA and provided in the Thrust library (Section).
- **TBB-Map** (CPU): Lock-free, unordered map hash table implementation written in TBB and packaged within the TBB library of parallel algorithms (Section).

HashFight is written with the open-source VTK-m library (v1.2), which is C++-11/14 compliant and provides a set of generic data-parallel primitives (e.g., Reduce, Sort, Scan, Copy, and LowerBounds) that can be invoked on both GPU and CPU devices with a single algorithm code base [26]. For NVIDIA GPU execution, primitives from the CUDA Thrust library are invoked; for CPU execution, primitives from the Intel TBB parallel algorithms library are invoked. Thus, HashFight can be mapped to either CUDA- or TBB-compliant code, via the back-end implementations of the primitives inside of VTK-m.

The Thrust Sort and Search comparators are meant to provide a baseline measure of throughput performance for a search-based task such as hashing. The combination of sorting key-value pairs and then querying them via binary search is an alternative to constructing and querying a hash table.

Platforms

To assess the cross-platform performance of HashFight, we conduct experiments on the following two GPU devices and one

CPU device, each residing on a single node:

- **K40 GPU**: NVIDIA Tesla K40 accelerator with 11.4 GB on-board memory.
- **V100 GPU**: NVIDIA Tesla V100 accelerator with 32.5 GB on-board memory.
- **Xeon Gold CPU**: 2.3 GHz Intel Xeon Gold 6140 (Skylake generation) CPU with 36 physical cores (72 logical) and 370 GB DDR4 memory.

All CPU code was compiled using GCC with flags for `-O3` optimization and the C++-11 standard. Additionally, the TBB scalable allocator was used for dynamic memory allocation with the TBB concurrent unordered map, which resizes itself as new key-value pairs are inserted. All GPU code was compiled using NVCC with GCC as the host compiler.

Dataset Sizes

In this study we focus on the task of hashing unique, unsigned 32-bit integer key-value pairs into the hash table. Each key and value is randomly-generated by the Mersenne Twister pseudo-random number generator, using a state size of 19937 bits (mt19937). This generator has been extensively used in the simulation domain and theoretically-proven to possess a long period ($2^{19937} - 1$ generated values without repetition) and high equidistribution. Any duplicate keys are removed, and the remaining unique keys are shuffled. To construct a hash table, a batch of k unsigned integer keys and k corresponding unsigned integer values are provided as input from two separate datasets. To query the hash table, a batch of k randomly-generated, unsigned integer keys is provided as input from a separate dataset. These query keys may contain duplicates and are not necessarily equal to any of the keys previously inserted into the table.

Among the four tested GPU and CPU platforms, we generate and experiment with datasets containing between 50 million and 1.45 billion unsigned integers (k), in increments of 50 million. This results in 29 different sizes of key-value pairs and query keys. The maximum size that can be executed on a given platform is a function of the maximum on-device memory of the platform and the hash table load factor (see Equations 1 and 2). Thus, holding the load factor constant, we are able to insert and query larger sets of key-value pairs on platforms with larger memory.

Hash Table Load Factors

We measure the effect of the insertion and query throughput as the hash table load factor, f, is varied between 1.03 and 2.0, inclusive. Overall, 10 different values of f are tested: 1.03, 1.10, 1.15, 1.25, 1.40, 1.50, 1.60, 1.75, 1.90, 2.0. A load factor of 1.03 was selected as the minimum value because the CUDPP cuckoo hash table implementation is only designed and tested for load factors of at least this value. Traditionally, a load factor of 2.0 has served as the conservative upper-bound for constructing a hash table [6]. The smaller load factors reduce the memory footprint of the hash table, but at the expense of an increase in the number of hash collisions.

Query Failure Rates

For a dataset size of k query keys, we randomly-generate 10 different sets of k query keys, each with a different percentage of keys that are not contained within the hash table; that is, "failed" queries that return empty query values. The rate of failure of query keys is varied, in increments of 10 percent, from 0 percent (all query keys exist in the table) up to 90 percent. This failure rate factor is meant to assess the worst-case querying ability of hash table implementations.

Results

In this section, we present and analyze the findings of our GPU and CPU hashing experiments. For each experiment, we assess the insertion and query throughput of three search-based techniques (HashFight compared to a benchmark hash table and sorting-based technique) as the dataset size, load factor, and query failure rate are varied. Each configuration, or data point, is run 10 trials, with each trial using a different randomly-generated data set of unique, 32-bit unsigned integer keys and values. The result of each configuration is reported as the average throughput of the 10 different runs, with the operation being an insertion or query. For a given configuration, the same 10 data sets are used by each hashing implementation, in order to provide a fair comparison.

GPU Experiments

We conduct three different GPU experiments, each run on both the NVIDIA K40 and V100 GPU devices. The results and analysis of these experiments are as follows.

Vary Data Size

Our first GPU experiment assesses the throughput performance of each hash table as the number of key-value pairs is increased and the load factor is held constant. We display results for constant load factors of 1.03 and 1.50, since they reveal the performance of both higher-capacity and lower-capacity hash tables.

Figure 1 displays the results for both the K40 and V100 GPUs, each testing a different maximum number of key-value pairs due to on-device memory limitations. From these results, we observe that HashFight achieves scalable and leading query throughput for the largest numbers of key-value pairs on both GPU devices, while remaining within a factor of 1.5 of the CUDPP cuckoo hash table for all other data sizes. For insertions, both hash table approaches maintain comparable throughput, with HashFight demonstrating more-stable throughput for both of the tested load factors, and CUDPP performing its best for the 1.50 factor. Overall, both HashFight and CUDPP achieve higher throughput for queries than for insertions, which is a desirable property for hash tables that are used primarily as look-up structures, particularly in real-time applications.

With the 1.03 load factor, HashFight attains a consistently-higher insertion and query throughput than CUDPP on both devices, and matches the query throughput of CUDPP for the smallest and largest data set sizes on the V100 device. With the 1.50 load factor, CUDPP sees an increase in query throughput and outperforms HashFight within a factor of 1.5 until 950 million key-value pairs. However, when the number of key-value pairs exceeds 1 billion, CUDPP experiences a drop in throughput and nearly matches the throughput of HashFight at 1.45 billion pairs. This trend can also be observed for CUDPP queries on the K40 device at both 150 million key-value pairs (1.03 load factor) and 250 million pairs (1.50 load factor).

A further analysis reveals that these dropoff points directly coincide with the points at which the total memory of the hash

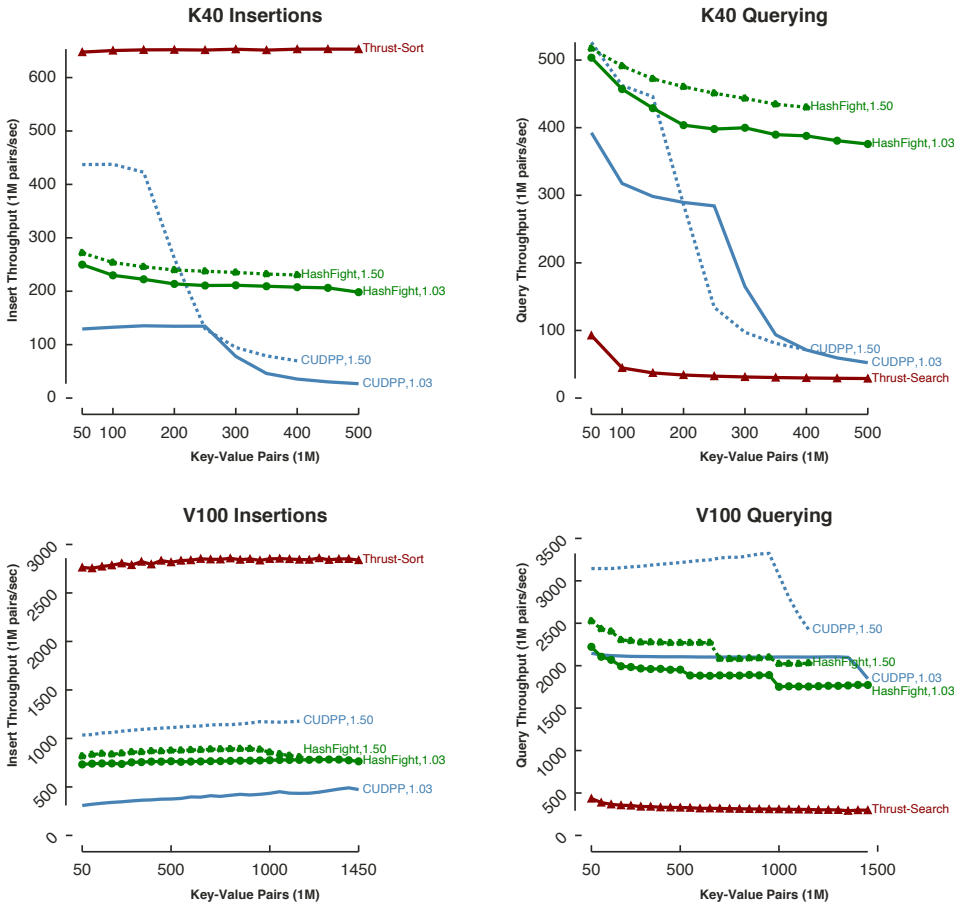


Figure 1. *GPU insertion and query throughput as the number of key-value pairs is varied on the K40 and V100 devices. For both HashFight and CUDPP, hash table load factors of* 1.03 *and* 1.50 *are presented separately.*

table exceeds the size, or coverage, of the translation lookaside buffer (TLB) of the last-level cache (L3 cache on K40 and L2 cache on V100). According to the micro benchmarking of Jia et al., the V100 and K40 have TLB sizes of approximately 8.2 GB and 2 GB, respectively [12]. On the K40 and V100, these last-level caches use physical memory addresses and so the virtual addresses of thread read and write requests must first be translated into physical addresses via one of the page tables cached in the TLB. Once the pages tables of the TLB are full, TLB misses induce page faults (or swaps) that increase the latency of memory transactions, regardless of whether the memory requests hit or miss the last-level cache. Recently, Karnagel et al. microbenchmarked a suite of modern NVIDIA GPUs and discovered that irregular memory accesses for data sets larger than 2 GB on the K40 result in latency increases, due to inefficient accesses to the L3 TLB [13]. Lai et al. expand upon this finding by modeling a multi-pass scatter and gather scheme that splits a batch of TLB-exceeding memory accesses into smaller chunks of accesses that each fit within the size of the TLB [14].

HashFight draws from these findings and performs the `Fight`, `CheckWinner`, and `Probe` kernels (Listings 2, 3, and 5) in a multi-pass fashion that, as seen in Figure 1, does not suffer a drop in throughput after the TLB size is exceeded. Once the hash table memory size reaches the TLB size, the hash table is logically broken into chunks of locations, each of roughly equal size and smaller than the TLB size. Then, a kernel is invoked for each chunk in order, and only the threads with memory accesses into the current chunk are allowed to insert or query their key. The minimization of TLB page faults more than offsets the overhead of invoking each kernel multiple times per HashFight iteration, resulting in more stable throughput than CUDPP for large data sizes and hash tables. This multi-pass feature maintains platform-portability, since TLB-specific constant values are only used when HashFight is compiled in GPU (CUDA) mode.

On the V100 GPU device, both tables realize a considerable improvement in insertion and query throughput, reaching approximately 1 billion pairs per second for insertions and at least 2 billion pairs per second for queries. This increase in throughput can be attributed to several hardware improvements over the older-generation K40, including increased global memory bandwidth and decreased latency for global memory atomic instructions. These improvements play a role in the absolute throughput differences observed between the devices, as CUDPP uses CAS atomic operations during insertions and both CUDPP and HashFight resolve a very high percentage of their memory transactions from global memory.

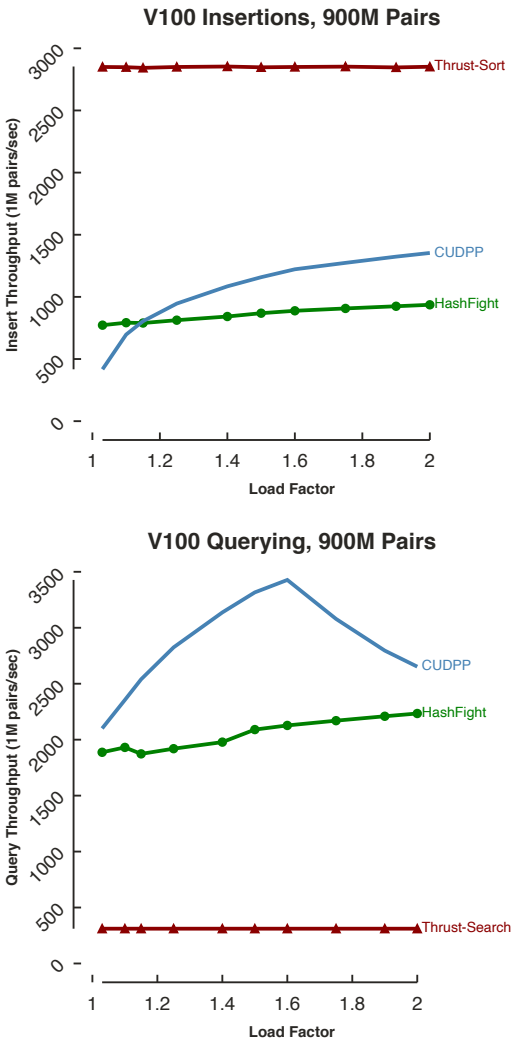


Figure 2. *GPU insertion and query throughput as the hash table load factor, or capacity, is varied on the V100 device. The number of key-value pairs inserted and queried is set equal to the maximum number that can satisfy on-device memory constraints for all load factors.*

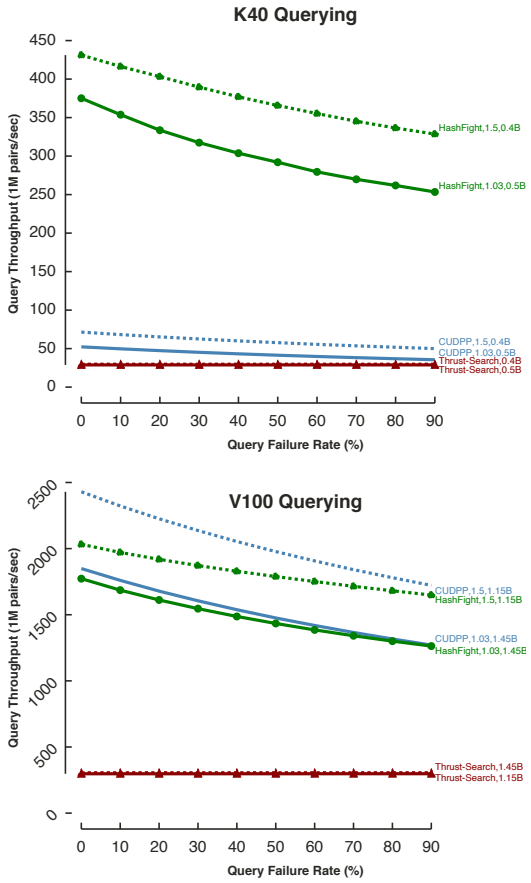


Figure 3. *GPU query throughput as the percentage of failed, or unsuccessful, queries is varied on the K40 and V100 devices. For both HashFight and CUDPP, separate plots are presented for hash table load factors of* 1.03 *and* 1.50*, which are queried with the maximum number of query keys (millions) permitted within on-device memory limits.*

Finally, Figure 1 reveals that, for both K40 and V100, sorting the input key-value pairs with the Thrust radix sort is significantly faster than inserting the pairs into either of the two hash tables. However, this comes with a tradeoff of significantly slower queries via the Thrust binary search, which suffers from uncoalesced and random-access query patterns into the sorted array. The hash tables enable each query to complete within a constant fixed number of uncoalesced probes through the hash table, whereas the binary search must make a logarithmic number of probes in the worst case.

Vary Load Factor

Our second GPU experiment measures the performance of the hash table approaches as the load factor is varied between 1.03 and 2.0, and the number of key-value pairs is held constant. We hold 300 million and 900 million pairs constant for the K40 and V100 devices, respectively, as these are the maximum numbers for which all load factors can be tested within on-device memory limits. We also plot the throughput of the Thrust radix sort and binary search for both of the tested number of key-value pairs; since the load factor is hash table-specific, the Thrust performance does not vary.

From Figure 2 we see that HashFight maintains very stable throughput performance across all load factors, never deviating by more than 200 million pairs per second. CUDPP, as an open-addressing method, achieves faster insertions, as the load factor and hash table capacity increase, particularly on the V100 GPU. On the K40 GPU, CUDPP fails to increase its insertion throughput for 300 million pairs beyond a load factor of 1.10 (the K40 plot is not shown due to similar findings as in V100 chart). At this point, the hash table memory usage reaches the maximum L3 TLB capacity of the K40 and, as noted before, CUDPP is unable to further increase its insertion throughput due to excessive TLB page faults. Also, as seen in Figure 1, HashFight performs comparably or better than CUDPP for the smallest load factors and highest-capacity hash table sizes.

Vary Query Failure Rate

Our third and final GPU experiment assesses the query throughput of HashFight and CUDPP as the percentage of failed, or unsuccessful, query keys is varied between 0 and 90 percent, while holding the number of query keys and load factor (1.03 and 1.50) constant.

Figure 3 reveals that CUDPP and HashFight are modestly affected by an increase in unsuccessful queries on both the K40 and V100. On the V100, CUDPP begins with a slightly higher query throughput than HashFight for the 1.50 load factor, but then sees a 43 percent decrease in throughput until it nearly matches the throughput of HashFight at the 90 percent failure rate. A similar pattern appears for the 1.03 load factor. On the K40, both hash table approaches realize slightly larger decreases in query throughput as compared to the V100 runs. For the 90 percent query failure rate and 1.50 load factor, HashFight and CUDPP realize decreases in throughput by 33 and 43 percent, respectively. For the 1.03 load factor, HashFight and CUDPP observe decreases in throughput by 48 and 47 percent, respectively, until the 90 percent failure rate, at which point the throughput of both tables is equivalent.

The K40 findings can be largely explained by the TLB caching limit and HashFight's TLB-oblivious design, as observed in the previous experiments, whereas the V100 findings are more indicative of algorithmic properties, such as increase in worst-case probes per thread and memory load transactions per warp. As an optimized cuckoo hash table, CUDPP only requires at most h lookup probes per query, where h is the number of cuckoo hash functions or possible table locations for a key to be inserted. Since this value is typically small (4 in this study), CUDPP does not have to perform too many extra global memory loads to determine that a query key does not exist within the hash table. HashFight is even less affected by the failed queries, as most threads can determine in the first iteration whether their active query key resides in the table, and rarely will need to exceed 6 iterations to determine failure.

CPU Experiments

We conduct three different experiments on the Intel Xeon Gold CPU, comparing the throughput of HashFight with that of the TBB concurrent unordered map (TBB-Map) and the Thrust sort and binary search primitives. HashFight and Thrust are compiled and run in TBB mode, without any code changes from the equivalent GPU experiments. The results and analysis of the CPU experiments are presented as follows.

Figure 4 shows that HashFight achieves a significantly higher throughput than the TBB-Map for both insertions and queries across all data sizes. In particular, for the largest batch of key-value pairs, 1.45 billion, and smallest load factor, 1.03, the throughput of HashFight exceeds that of TBB-Map by approximately 30X and 3X for insertions and queries, respectively. From Figure 5, as the load factor, or table capacity, is increased from 1.03 up to 2.0, HashFight continues to increase its insertion and query throughput by 1.3X and 1.4X, respectively. However, TBB-Map maintains relatively the same throughput for insertions (8.5 million pairs/sec) and queries (137.5 million pairs/sec) until a load factor of 1.60. After this point, TBB-Map realizes a noticeable 6.3X decrease in insertion throughput and a 1.35X decrease in query throughput, instead of expected increases. Also, the aggregate runtime of performing a Thrust sort followed by a vectorized

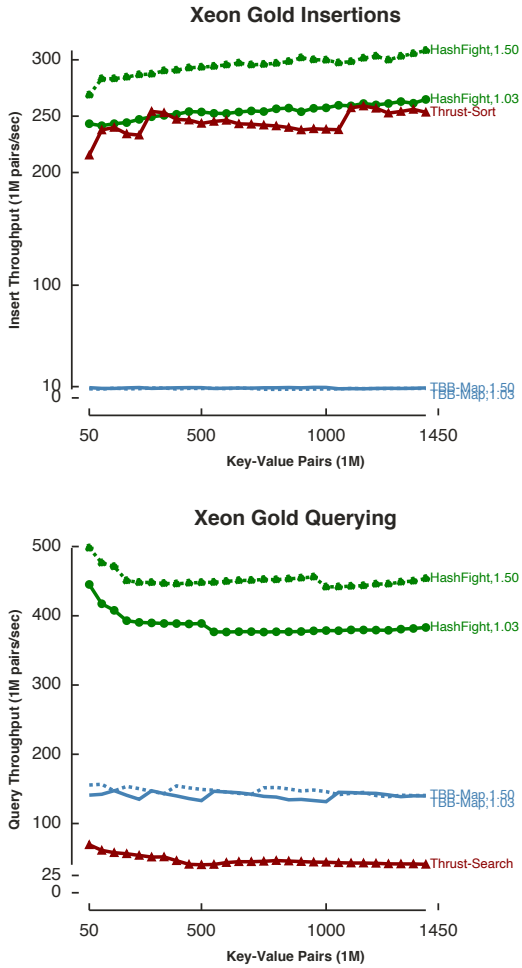


Figure 4. *CPU insertion and query throughput as the number of key-value pairs is varied on the Xeon Gold device. For both HashFight and the TBB unordered map, hash table load factors of* 1.03 *and* 1.50 *are presented separately.*

binary search is actually faster than that of TBB-Map, yet still considerably slower than the aggregate runtime of HashFight.

The significantly lower insertion throughput of TBB-Map is largely due to the underlying linked list design of the hash table and its construction under large magnitudes of key-value pairs, such as those tested in this experiment. As an extensible hash table, TBB-Map must frequently resize and allocate new memory segments as more and more key-value pairs are inserted in an unordered fashion [31]. Each insertion of a key-value pair has to follow multiple layers of pointer indirection to access a segment bucket, and the insertion must be synchronized via a lock-free atomic primitive, which adds additional overhead above that of HashFight.

Moreover, the slight drop in throughput for TBB-Map above a load factor of 1.50 is a combination of excessive TLB cache thrashing and the use of separate-chaining for collision resolution. A hash table with a load factor of 1.50 requires at least 16 GB of memory to insert 1.45 billion pairs. This memory footprint just exceeds the aggregate coverage of the L1 and L2 TLBs on the Xeon Gold CPU, and, since page entries are of smaller

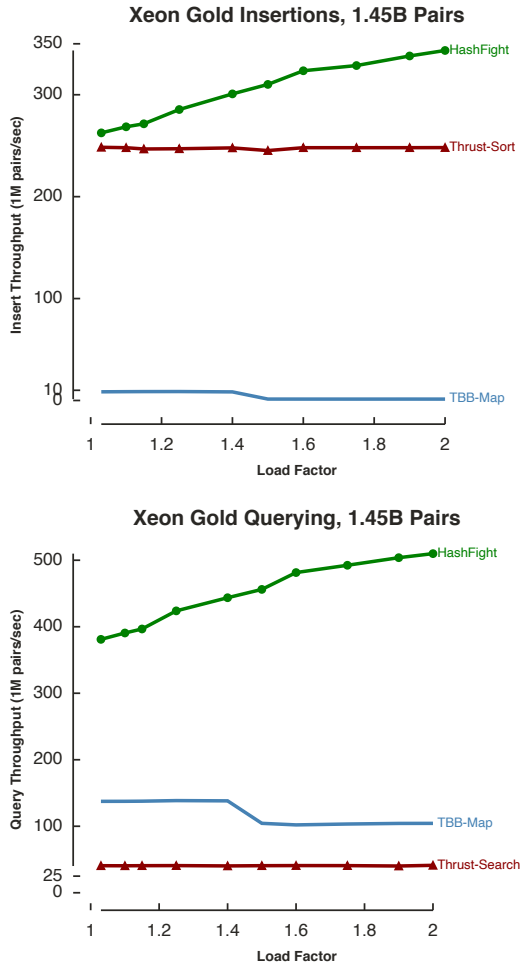


Figure 5. *CPU insertion and query throughput as the hash table load factor is varied on the Xeon Gold device.*

4KB and 2MB sizes, pages are frequently swapped in and out of a TLB by concurrent threads during random-access insertions. As observed in our GPU experiments, memory latency increases and insert throughput decreases as more hash table address mappings reside outside of the TLB. HashFight obviates this issue by means of its multi-pass gather and scatter procedure, and its use of open-addressing collision resolution, whereby the number of memory accesses typically decreases as the hash table size increases. However, TBB-Map is vulnerable to TLB memory limitations and does not benefit as well from a larger hash table due to multiple layers of pointer indirection and linear probing within buckets. A possible direction towards alleviating the TLB issue is to use huge 1 GB page entries. This permits more virtual-to-physical memory mappings in a single TLB access, but requires modifying the system configuration with root permission; due to the latter reason, we were unable to perform this experiment.

In order to validate the results of TBB-Map, we also tested the same CPU experiments with the TBB concurrent hash map, which is based on an underlying array structure as opposed to a linked-list. The concurrent hash map produced nearly identical insertion and query throughput as the concurrent unordered map. Additionally, by conducting multiple trials per configuration (10), we verified that the resulting runtimes were not affected by "cold" threads, whereby some threads are not yet active and need a warmup phase.

Finally, we conducted the third experiment of varying the query failure rate and observed that, unlike in the GPU experiments, the query throughput of HashFight and TBB-Map is only marginally decreased as the rate is increased. Since there are many buckets in the TBB-Map hash table and a relatively light load per bucket, the cost of performing an unnecessary or failed query is non-increasing, contrary to extra probing required by CUDPP cuckoo hashing for failed queries on the GPU. Due to the small variation in throughput, we have omitted the corresponding chart for this experiment.

Conclusion and Future Work

This study demonstrates the viability of HashFight as a platform-portable approach that can support the implementation of visualization and data analysis algorithms within the VTK-m framework. In particular, HashFight contributes the following:

1. Hash table data structures can be implemented in platform-portable visualization frameworks, such as VTK-m.
2. Provides important evidence that the DPP approach can still lead to competitive performance compared to hardware-specific implementations.
3. Leading insertion and query throughput to best-in-class GPU- and CPU-based implementations for compact hash tables and hash tables that exceed the TLB coverage. HashFight demonstrates comparable performance for large hash tables and the largest data sets of up to a billion integer key-value pairs on a single device.
4. Platform-portable, DPP-based design that does not use hardware atomics or locking for collision-resolution. Based on this design, HashFight maintains competitive performance across different GPU platforms, which differ in such properties as device memory size and memory bandwidth.
5. Provide evidence of the significant GPU performance penalty incurred by the CUDPP hash table implementation as the TLB cache coverage is exceeded. HashFight mitigates this performance issue via a multi-pass routine.

Future work includes adding support for 64-bit unsigned integer keys and values, and assessing the performance of HashFight with spatial point-coordinate keys. This latter experiment can be coupled with the Morton curve hash function, which seeks to map spatially-close points to similar-valued unsigned integer hash values. In this case, the hash values would be nearby indices in the hash table and could offer coalesced memory loads when queried in an ordered fashion.

References

[1] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta, *Real-time parallel hashing on the GPU*, ACM SIGGRAPH Asia 2009 Papers, ACM, pp. 154:1–154:9.

[2] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta, *Chapter 4 - Building an efficient hash table on the GPU*, GPU Computing Gems Jade Edition (Wen-mei W. Hwu, ed.), Applications of GPU Computing Series, Morgan Kaufmann, Boston, 2012, pp. 39–53.

[3] S. Ashkiani, M. Farach-Colton, and J. D. Owens, *A Dynamic Hash Table for the GPU*, Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium, pp. 419–429.

[4] Saman Ashkiani, Andrew Davidson, Ulrich Meyer, and John D. Owens, *GPU multisplit*, Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2016, pp. 12:1–12:13.

[5] Rajesh Bordawekar, *Evaluation of parallel hashing techniques*, GPU Technology Conference, March 2014.

[6] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson, *Introduction to algorithms*, 2nd ed., McGraw-Hill Higher Education, 2001.

[7] Weiwei Duan, Jianxin Luo, Guiqiang Ni, Bin Tang, Qi Hu, and Yi Gao, *Exclusive grouped spatial hashing*, Computers & Graphics (2017), 71–79.

[8] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, Journal of Parallel and Distributed Computing **74** (2014), no. 12, 3202 – 3216.

[9] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram, *Coherent parallel hashing*, ACM Trans. Graph. **30** (2011), no. 6, 161:1–161:8.

[10] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl, *Hardware-oblivious parallelism for in-memory column-stores*, Proc. VLDB Endow. **6** (2013), no. 9, 709–720.

[11] Intel Corporation, *Intel Thread Building Blocks Documentation*, January 2019, https://github.com/intel/tbb.

[12] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza, *Dissecting the NVIDIA Volta GPU architecture via microbenchmarking*, CoRR **abs/1804.06826** (2018), 1–66.

[13] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner, *Big data causing big (TLB) problems: Taming random memory accesses on the GPU*, Proceedings of the 13th International Workshop on Data Management on New Hardware (New York, NY, USA), ACM, 2017, pp. 6:1–6:10.

[14] Zhuohang Lai, Qiong Luo, and Xiaoying Jia, *Revisiting multi-pass scatter and gather on GPUs*, Proceedings of the 47th International Conference on Parallel Processing, ACM, 2018, pp. 25:1–25:11.

[15] Matthew Larsen, Kenneth Moreland, Chris R. Johnson, and Hank Childs, *Optimizing Multi-Image Sort-Last Parallel Rendering*, Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV) (Baltimore, MD), October 2016, pp. 37–46.

[16] Lawrence Livermore National Laboratory, *RAJA User Guide*, October 2019, https://raja.readthedocs.io/en/master/.

[17] Sylvain Lefebvre and Hugues Hoppe, *Perfect spatial hashing*, ACM SIGGRAPH 2006 Papers (New York, NY, USA), SIGGRAPH '06, ACM, 2006, pp. 579–588.

[18] Brent Lessley, Roba Binyahib, Robert Maynard, and Hank Childs, *External Facelist Calculation with Data-Parallel Primitives*, Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (Groningen, The Netherlands), June 2016, pp. 10–20.

[19] Brenton Lessley, Talita Perciano, Manish Mathai, Hank Childs, and E. Wes Bethel, *Maximal Clique Enumeration with Data-Parallel Primitives*, Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV), October 2017, pp. 16–25.

[20] Shaomeng Li, Matthew Larsen, John Clyne, and Hank Childs, *Performance impacts of in situ wavelet compression on scientific simulations*, Proceedings of the In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization Workshop, ACM, 2017.

[21] Shaomeng Li, Nicole Marsaglia, Vincent Chen, Christopher Sewell, John Clyne, and Hank Childs, *Achieving Portable Performance For Wavelet Compression Using Data Parallel Primitives*, Eurographics Symposium on Parallel Graphics and Visualization, The Eurographics Association, 2017.

[22] CUDA Data Parallel Primitives Library, *CUDA Data Parallel Primitives Library*, http://cudpp.github.io, September 2019.

[23] Duane G. Merrill and Andrew S. Grimshaw, *Revisiting sorting for GPGPU stream architectures*, Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (New York, NY, USA), PACT '10, ACM, 2010, pp. 545–546.

[24] Maged M. Michael, *High performance dynamic lock-free hash tables and list-based sets*, Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (New York, NY, USA), ACM, 2002, pp. 73–82.

[25] Microsoft Corporation, *Parallel Patterns Library Documentation*, January 2019, https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl.

[26] Kenneth Moreland, Christopher Sewell, William Usher, Lita Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci, *VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures*, IEEE Computer Graphics and Applications (CG&A) **36** (2016), no. 3, 48–58.

[27] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, *Real-time 3D reconstruction at scale using voxel hashing*, ACM Transactions on Graphics (TOG) (2013), 1–11.

[28] Nvidia Corporation, *Thrust*, October 2019, https://thrust.github.io.

[29] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes, *Java concurrency in practice*, Addison-Wesley Professional, 2005.

[30] Nadathur Satish, Mark Harris, and Michael Garland, *Designing efficient sorting algorithms for manycore GPUs*, Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, IEEE Computer Society, 2009, pp. 1–10.

[31] Ori Shalev and Nir Shavit, *Split-ordered lists: Lock-free extensible hash tables*, J. ACM **53** (2006), no. 3, 379–405.

[32] Abhishek Yenpure, Hank Childs, and Ken Moreland, *Efficient Point Merge Using Data Parallel Techniques*, Eurographics Symposium on Parallel Graphics and Visualization (EGPGV) (Porto, Portugal), June 2019, pp. 79–88.

Author Biography

Brenton Lessley is a Senior Software Engineer at Verb Surgical Inc. He received his Ph.D. in Computer Science from the University of Oregon in 2019. His research interests include data-parallel algorithm design, index-based search techniques, platform-portable performance, and scientific visualization.

Shaomeng Li is a Project Scientist in the Computational and Information Systems Laboratory (CISL) at the National Center for Atmospheric Research (NCAR). He received his Ph.D. in Computer Science from the University of Oregon in 2017. His research focuses on high performance computing, scientific data reduction, and machine learning.

Hank Childs is an Associate Professor in the Computer and Information Science Department at the University of Oregon. He received his Ph.D. in Computer Science from the University of California at Davis in 2006. His research focuses on scientific visualization, high performance computing, and the intersection of the two.

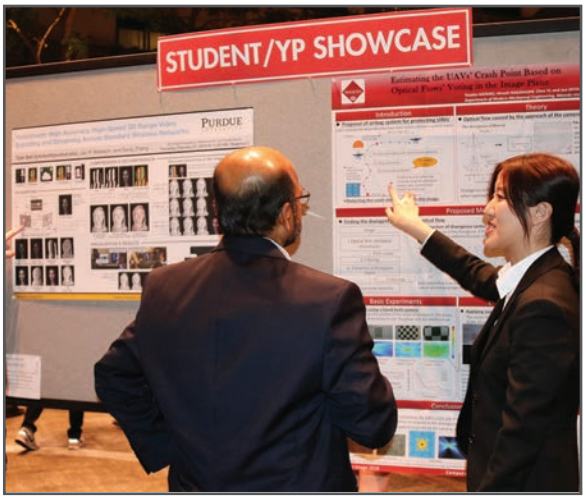
JOIN US AT THE NEXT EI!

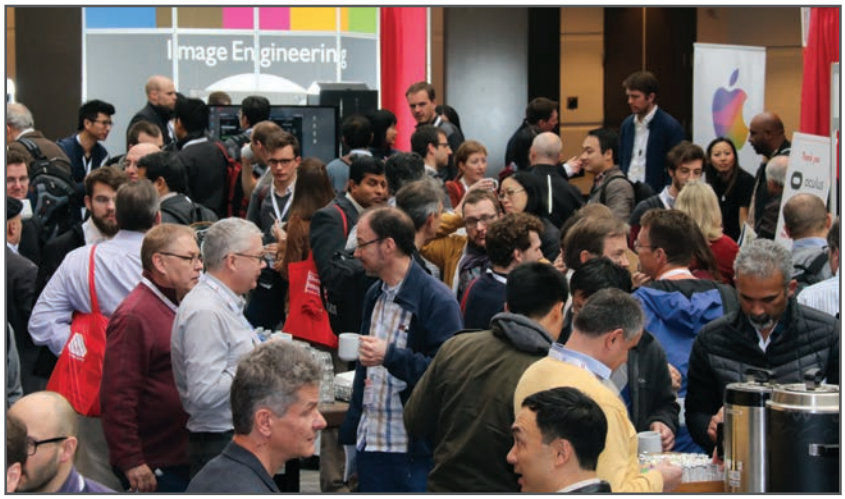
IS&T International Symposium on

Electronic Imaging

SCIENCE AND TECHNOLOGY

Imaging across applications . . . Where industry and academia meet!





• SHORT COURSES • EXHIBITS • DEMONSTRATION SESSION • PLENARY TALKS •
• INTERACTIVE PAPER SESSION • SPECIAL EVENTS • TECHNICAL SESSIONS •

www.electronicimaging.org

IS&T
imaging.org