CHAPTER

Building an Efficient Hash Table on the GPU

4

Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta

Hash tables provide fast random access to compactly stored data. In this chapter, we describe a fast parallel method for building hash tables. It is both simpler and faster than previous methods: on a GTX 470, our implementation can achieve hash insertion rates of around 250 million pairs per second; a table containing 32 million items can be built in 130 ms and have all items retrieved in parallel in 66.8 ms. The key to this performance is minimizing the total number of uncoalesced memory accesses. We also discuss how to specialize the hash table for different situations and analyze our performance with respect to the hardware bottlenecks for uncoalesced memory access.

4.1 INTRODUCTION

Hash tables are one of the most basic data structures used to provide fast access and compact storage for sparse data. Data can typically be retrieved from a hash table using $O(1)$ memory accesses per item, on average. But this performance is achieved by using randomness to distribute the items in memory, which poses a problem on the GPU: uncoalesced memory accesses are an order of magnitude slower than sequential memory accesses. This forms a bottleneck for both hash table construction and retrieval.

In this chapter, we describe a fairly straightforward algorithm for parallel hash table construction on the GPU. We construct the table in global memory and use atomic operations to detect and resolve collisions. Construction and retrieval performance are limited almost entirely by the time required for these uncoalesced memory accesses, which are linear in the total number of accesses; so the design goal is to minimize the average number of accesses per insertion or lookup. In fact, we guarantee a constant *worst-case* bound on the number of accesses per lookup. Although this was thought to be essential in earlier work (such as our previous construction [1], or in Perfect Spatial Hashing [2]), it turns out to be important only in that it helps keep the average down, especially when many lookups are misses. Our previous construction also factors the work into independent thread blocks, makes use of shared memory, and reduces thread divergence during construction, but as this required more total uncoalesced memory accesses, it was not as effective a design.

One alternative to using a hash table is to store the data in a sorted array and access it via binary search. Sorted arrays can be built very quickly using radix sort because the memory access pattern of radix sort is very localized, allowing the GPU to coalesce many memory accesses and reduce their cost

GPU Computing Gems. DOI: 10.1016/B978-0-12-385963-1.00004-6

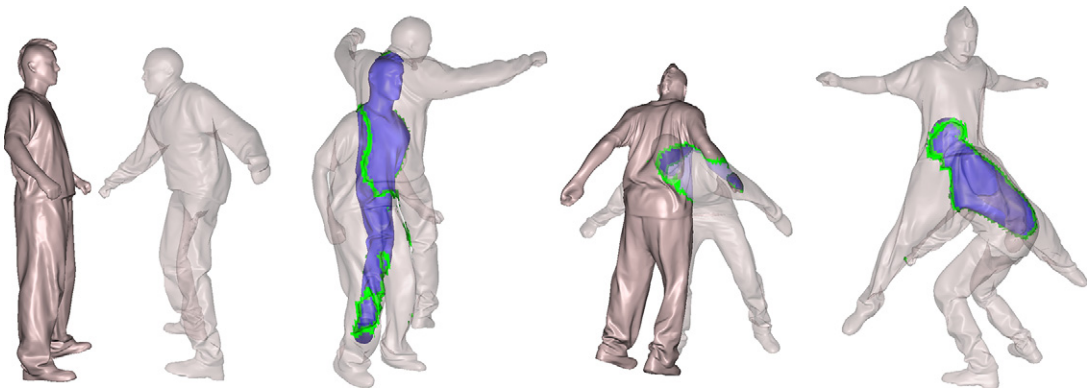

FIGURE 4.1

GPU hash tables are useful for dynamic spatial data. Shown here are four frames from an application performing Boolean intersections between two animated point clouds at interactive rates. Hash tables are built for both models' voxels every frame, then queried with the voxels of the other model to find surface intersections. A floodfill then determines which portions of each model are inside the other.

significantly. However, binary search, which incurs as many as $\lg(N)$ probes in the worst case, is much less efficient than hash table lookup.

GPU hash tables are useful for interactive graphics applications, where they are used to store sparse spatial data — usually 3D models that are voxelized on a uniform grid. Rather than store the entire voxel grid, which is mostly empty, a hash table is built to hold just the occupied voxels. Querying the table with a voxel location then returns a positive result only when the voxel is occupied. This method is much more space-efficient and still allows all the lookups to be performed quickly. Example applications include volumetric painting [2] and collision detection between deforming models [1] (Figure 4.1). Hash tables are also useful in a GPGPU context for time-consuming matching algorithms. Many applications use hash tables as databases to find potential matches between different objects; two examples include image recognition and DNA alignment.

We provide an overview of our algorithm in Section 4.2, then go into the details of the construction of a basic hash table that stores a single value for each key in Section 4.3. We briefly describe how to extend our construction algorithm for more specialized hash tables in Section 4.4, then analyze the hash table's performance in Section 4.5. We conclude with future directions in Section 4.6.

4.2 OVERVIEW

We aim for a hash table that provides both fast construction and lookup of millions of key-value pairs on the GPU. Our previous work on GPU hash tables [1] had the same goals, but the method we present in this chapter is more flexible and efficient for modern GPUs, with superior construction time, lookup time, and memory requirements. Our solution is based on *cuckoo hashing* [3], which assigns each item a small, random set of slot choices for insertion (see Figure 4.2); each choice is determined by a different hash function. Items are moved between their choices to accommodate other items, but will

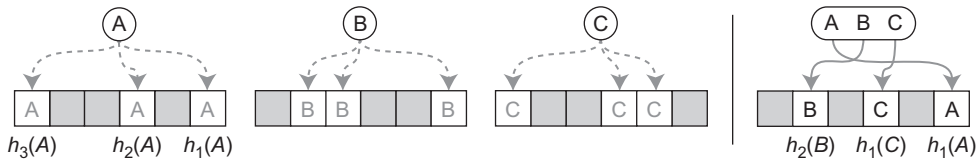

FIGURE 4.2

Cuckoo hashing uses multiple hash functions $h_i(k)$ to give keys a restricted set of insertion locations; in this case, each key may be placed in one of three locations indicated by the arrows (left). The algorithm finds a conflict-free configuration where each key is assigned its own slot (right), guaranteeing that any query can be answered within a constant number of probes.

always be located in exactly one location. The number of choices available for each item (typically three or four) is fixed for all items, guaranteeing that any item can be found after checking a constant number of slots.

This guarantee is one reason cuckoo hashing is a good fit for the GPU: when many threads each request an item from our hash table, we know that all of those requests will complete in constant time, keeping the average number of memory accesses per retrieval low. Another reason is that cuckoo hash table construction is efficient, allowing simultaneous insertion of many keys at the same time without the need for serialization. Moreover, it can be tailored for different situations: its parameters can be changed to balance between faster construction rates, higher retrieval efficiency, or tighter memory usage.

Although it is a probabilistic algorithm, cuckoo hashing has a high success rate for finding conflict-free placements; this counterintuitive result comes from its use of multiple hash functions. Essentially, moving the items during an insertion allows items to flow toward less contested slots of the table. Although these chains can be long once there are fewer slots available, we found that the median number of iterations required to insert any item was only one or two for our test cases.

Our hash table implementation inserts all items into the hash table in parallel. With our base configuration, the hash table uses $1.25N$ space for N items and allows each item four choices, guaranteeing that any item can be found after at most four probes.

4.3 BUILDING AND QUERYING A BASIC HASH TABLE

We begin by discussing the implementation of a basic hash table, which stores a single value for each unique key. The input consists of N pairs of 32-bit keys and values, where none of the keys are repeated. The value could represent indices into another structure, allowing a key to reference more than just 32 bits of data; we briefly discuss this in Section 4.5. In this section we describe an implementation that has an equal emphasis on construction time, lookup time, and memory usage, but Section 4.3.1 discusses trade-offs in our parameters that would allow more refined optimizations.

4.3.1 Construction

The construction process for a cuckoo hash table finds a configuration where each input item is inserted into a unique slot in the hash table. Each item is assigned a set of possible insertion locations, each of

which is determined by a different hash function, h_i. To insert an item, we choose one hash function and insert the item into the location determined by that function. If that location is already occupied, we replace the item already there with our item, and then use the same process to reinsert the evicted item into the hash table.

To determine where to reinsert an evicted item, a serial algorithm would either search all possibilities until it finds an empty slot, or randomly select one of the hash functions. We choose a simpler approach, deterministically using each item's hash functions in round-robin order. An item evicted from slot $h_i(k)$ is reinserted into slot $h_{i+1}(k)$. When moved from its last slot, or when being inserted for the first time, the item uses $h_1(k)$.

This can result in chains of evictions, which terminate either when an empty slot is found (Figure 4.3) or when we determine it is too unlikely that one will be found. We allow several attempts to build the table in case of a failure. Three main parameters affect how often this occurs and how the table performs: the hash functions chosen, the size of the hash table, and the maximum number of evictions a single thread will allow. The failure rate can be driven arbitrarily low by tweaking these parameters, giving different tradeoffs in construction and access times.

4.3.1.1 Parameters

The hash functions $h_i(k)$ should ideally be tailored for the application so that they distribute all of the items as evenly as possible throughout the table. However, it is difficult to guarantee this for every given input. To achieve fast construction times, we instead rely on randomly generated hash functions of the form:

$$h_i(k) = (a_i \cdot k + b_i) \bmod p \bmod tableSize$$

Here, $p = 334214459$ is a prime number, and *tableSize* is the number of slots in the table. Each h_i uses its own randomly generated constants a_i and b_i. These constants produce a set of weak hash functions, but they limit the number of slots under heavy contention and allow the table to be built successfully in most of our trials. We found that using an XOR operation worked better than multiplication

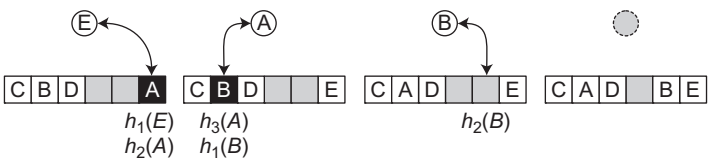

FIGURE 4.3

Insertion into a cuckoo hash table is a recursive process, where the table's contents are moved around to accommodate items being inserted. In this example, E needs to be inserted into slot $h_1(E)$ of the hash table (left), but the slot is already occupied by A. Cuckoo hashing swaps A and E and moves A to its next slot; in this case, A was evicted from slot $h_2(A)$, so it will move into slot $h_3(A)$. This process repeats until either an empty slot is found, or the algorithm determines that it is unlikely to find one (right).

at distributing the items for some datasets; the question of what implementable hash functions to use for cuckoo hashing for provable performance guarantees is a current research topic in theoretical computer science.[1]

Using four hash functions yields a good balance between all three metrics, though the hash table can be constructed with other numbers of hash functions. Using more functions allows each key more choices for insertion, making it easier to build the table and decreasing the construction time. However, this increases the average number of probes required to find an item in the table, increasing retrieval times. This is more obvious when querying for items not in the structure.

The size of the table affects how easily the construction algorithm can find a conflict-free configuration of the items. Given a fixed input dataset, bigger tables have less contention for each slot, reducing both the average length of an eviction chain and the average number of probes required to retrieve an item. Our experiments show that the fastest construction time is achieved using around $4N$ space, but this can be impractical for space-critical applications.

When memory usage is important, using more hash functions makes it easier to pack more items into a smaller table. Theoretical bounds on the minimum table size for differing numbers of hash functions were calculated by Dietzfelbinger et al. [5], but we found the smallest practical table sizes with our hash functions were slightly larger at $\sim 2.1N$, $1.1N$, $1.03N$, and $1.02N$ for two through five hash functions, respectively; our hash table uses $1.25N$ space, which limited the number of restarts and balanced construction and retrieval times.

The maximum number of iterations tells the algorithm when the insertion process is unlikely to terminate. Threads usually finish well before reaching a chain of this length, but the limit must be set carefully as it is the only way for the algorithm to terminate with a bad set of hash functions. Setting this number too high results in very expensive restarts, but setting it too low will cause the algorithm to declare failure too early.

There is no theoretical value for the limit, but it depends on both the table size and the number of items being inserted: cramming a large number of items into a small hash table greatly increases slot contention, resulting in longer eviction chains. For four hash functions and $1.25N$ space, we let threads follow chains of at most $7\lg(N)$ evictions; for 32 million items, we saw an empirical restart rate of 0.56%.

4.3.1.2 Building the Table

Our hash table stores keys and their values interleaved, allowing key-value pairs to be retrieved with a single memory access. Each slot of the cuckoo hash table stores a 64-bit entry, with the key and its value stored in the upper and lower 32 bits, respectively.

Construction may require several attempts if the algorithm cannot find a conflict-free configuration of the items. Each attempt to build the table first initializes *cuckoo*[] with a special entry $\varnothing$, which signifies that the slot is empty; we used an item with a key of `0xffffffff` and value of zero. Random constants are then generated for all of the hash functions being used.

[1] See the original paper [3] for more on the theory and practice of the choice of hash function for cuckoo hashing, and subsequent work [4] for an analysis of why weak hash functions generally perform well in practice.

```
#define get_key(entry)        ((unsigned)((entry) >> 32))
#define make_entry(key,value) ((((unsigned long long)key) << 32) + (value))

// Load up the key–value pair into a 64–bit entry.
unsigned key             = keys[thread_index];
unsigned value           = values[thread_index];
unsigned long long entry = make_entry(key, value)

// New items are always inserted using their first hash function.
unsigned location        = hash_function_1(key);

// Repeat the insertion process while the thread still has an item.
for (int its = 0; its < max_iterations; its++) {
  // Insert the new item and check for an eviction.
  entry = atomicExch(&table[location], entry);
  key = get_key(entry);
  if (key == KEY_EMPTY) return true;

  // If an item was evicted, figure out where to reinsert the entry.
  unsigned location_1 = hash_function_1(key);
  unsigned location_2 = hash_function_2(key);
  unsigned location_3 = hash_function_3(key);
  unsigned location_4 = hash_function_4(key);
       if (location == location_1) location = location_2;
  else if (location == location_2) location = location_3;
  else if (location == location_3) location = location_4;
  else                             location = location_1;
};

// The eviction chain was too long; report the failure.
return false;
```

Listing 4.1. Inserting a new item into the hash table using four hash functions.

The cuckoo hashing algorithm is performed by a single kernel (Listing 4.1). The kernel is launched with N threads, each managing one item at a time from the input. Recall that threads complete their work when they successfully place their item (or the item(s) displaced by their item), but a thread block will not complete until all of its threads are done. Thus we choose a relatively small thread block size — 64 threads — that minimizes the number of threads within a block kept alive by a single thread's long eviction chain.

We treat key-value pairs as a single 64-bit entity, which keeps each value with its key throughout the entire process. An alternative would be to build the table using only the keys, but this would require a second pass to insert values into their proper locations. Using `atomicExch()` operations, each thread

repeatedly swaps its current item with the contents of the slot determined by the current hash function. The atomic operations guarantee that no items are lost when threads simultaneously write into the same slot.

If the swap returned an $\varnothing$ item, the thread declares success and stops. Otherwise, the thread must reinsert the item it evicted. Because hash functions are used in round-robin order, the thread must figure out which hash function was previously used to insert it. It calculates the value of $h_i(k)$ for all of the hash functions, then checks which of those functions points to the slot it was evicted from. The evicted item is then reinserted using the next hash function in the series.

Our code snippet short-circuits after finding the earliest match, preventing the item from utilizing its full set of assigned slots when multiple hash functions return the same value. In practice, the rate of failure is unaffected, but it is an issue for tables containing only a few hundred items. This issue can be addressed either by advancing to the next function that doesn't point to the same location or by randomly picking a hash function to use.

We designate a failure by setting a flag when we reach the maximum eviction chain length, indicating that the table must be rebuilt from scratch. Once successfully built, we store the contents of the table, the number of slots in the table, and the hash function constants.

4.3.2 Retrieval

Retrieval of a query key can be performed by checking all of the locations identified by the hash functions (Listing 4.2). Each of the locations is then checked in order, either immediately returning the key's value as soon as it is found, or returning a "not found" value after all possible locations have been checked.

```
#define get_value(entry)  ((unsigned)((entry) & 0xffffffff))

// Compute all possible locations for the key.
unsigned location_1 = hash_function_1(key);
unsigned location_2 = hash_function_2(key);
unsigned location_3 = hash_function_3(key);
unsigned location_4 = hash_function_4(key);

// Keep checking locations until the key is found or all are checked.
unsigned long long entry;
if (get_key(entry = table[location_1]) != key)
  if (get_key(entry = table[location_2]) != key)
    if (get_key(entry = table[location_3]) != key)
      if (get_key(entry = table[location_4]) != key)
        entry = make_entry( 0, NOT_FOUND );

return get_value(entry);
```

Listing 4.2. Retrieving an item from the table using four hash functions.

4.4 SPECIALIZING THE HASH TABLE

The basic hash table can be specialized for different applications. In this section, we briefly describe some variations on the structure.

Sets are basic data structures for storing lists of nonrepeated elements; a typical operation is to check if some element is a member of the set. A hash table can be used to implement this because querying it with a key will indicate whether it was part of the original input. Construction of the hash table is modified slightly to work with just 32-bit keys so that each slot of the table contains only a key.

The input consists of 32-bit keys, which may or may not be unique. Similarly to the basic hash table construction, each thread manages a single key and uses 32-bit `atomicExch()` operations to swap their key with the contents of the table. The thread now stops if the slot was empty, or if the thread exchanged two copies of the same key. Once all threads have stopped, multiple copies of the same key may still be inserted in the table, so a cleanup phase must occur afterward to erase all copies of a key except the first.

Compacting hash tables extend sets, counting the number of unique keys in the input and assigning a unique ID to each as their value, effectively "compacting" the input. The compacting hash table consists of a hash table and a compacted list of the unique keys; this combination allows $O(1)$ translation between the IDs and the keys in both directions. It is most useful for memory-intensive applications where a lot of data is stored on a per-key basis: it determines how big an array is needed to store the data and directs each unique key to a different location in the array.

The cuckoo hash table is built just as they are for sets, but during the cleanup phase the first copy of each key in the table is assigned a value of one, with all other slots in the table filled with $\varnothing$. A prefix sum is then performed over the values, creating a unique value from 0 to $n-1$ for every unique key stored in the hash table, where n is the number of unique keys. We use another pass to interleave the keys with their values in the table, and finally create the compacted list by writing out each key to a separate array into the slot identified by its ID.

Multivalue hash tables are generalizations of the basic hash table that store multiple values for each key. As input, they take in one array of keys and one array of values, where each of a key's values is represented as different key-value pairs with the same key. Intuitively, the main goal during construction is to boil down the input to a single value for each key so that it can be stored using a basic hash table. The value here is a unique ID that indexes into an auxiliary array that stores more information about the key. This process can be adapted to store data other than 32-bit integers.

The multivalue hash table stores two auxiliary arrays: *sortedValues*[ ], which stores each key's values contiguously, and *keyInfo*[ ], which stores the location of every key's values and the number of values each has. Construction of the table is detailed in Algorithm 1; the majority of the algorithm is spent preprocessing the input. We first use a radix sort on the input pairs to rearrange the data, placing all copies of the same key adjacent in memory. Additionally, all values of each key end up in a contiguous location within their array; this array is stored as *sortedValues*[ ].

Next, we examine the sorted keys array to find the indices of the first instance of each key; these indices also indicate where in *sortedValues*[ ] each key's values start, and they can be differenced to determine how many values each key has. This information is stored in *keyInfo*[ ] as uint2s.

We then create a scratch array of flags indicating whether each key was unique, then perform a prefix sum to assign each key a unique ID. These unique IDs are used to index into *keyInfo*[ ] and to

Algorithm 1: Multivalue hash table construction

1: radix sort the input key and value arrays
2: determine what and where the unique keys are in the sorted key array
3: determine how many values each key has by differencing the locations of consecutive unique keys
4: perform a prefix sum to assign each key a unique ID
5: create an auxiliary array *keyInfo*[ ] to store the location and number of each key's values
6: create a basic hash table using the unique keys and their IDs

compact down the unique keys. The compacted list and their IDs serve as the input for the creation of a cuckoo hash table using the basic algorithm.

Querying the structure with a valid key first performs a retrieval using the hash table. The result is then used as an index into *keyInfo*[ ] to return information about the key's values.

4.5 ANALYSIS

4.5.1 Performance of the Basic Hash Table

To measure the performance of our hash table, we repeatedly measured the time taken to build and query it using the parameters we chose in Section 4.3. Our inputs consisted of increasingly larger sets of random 32-bit keys and values. Retrieval rates measured how long it took to retrieve all of the input data, with each retrieval performed by a different thread to mimic applications randomly accessing the data. We ran the same tests with two alternative GPU data structures: a array sorted with Merrill's radix sort implementation [6], and the two-level hash tables produced using our previous method [1]. Our tests were performed on a machine using CUDA 3.2 and the NVIDIA 260.24 drivers on an EVGA GTX 470 SC (Figure 4.4).

The sorted array uses exactly N space for N pairs and can be built at much higher rates than either hash table for large inputs. However, it is queried using $O(\lg N)$ binary searches; performance depends greatly on how well ordered the queries are. Retrieval times grow increasingly worse for larger sets because the memory reads were uncoalesced and many threads required the full set of probes.

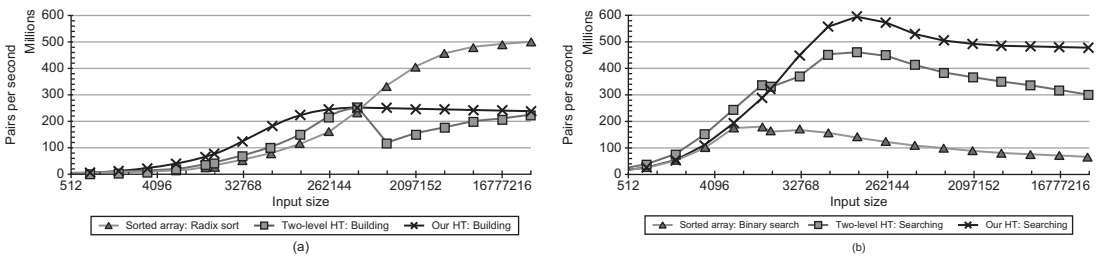

FIGURE 4.4

Comparison between insertion rates (left) and retrieval rates (right) for increasingly larger sets of random pairs of keys and values on the GTX 470. Higher rates are better.

The two-level hash table uses a total of $1.42N$ space. Its construction is faster than a radix sort for smaller inputs, but it becomes increasingly slower with larger inputs. However, it guarantees $O(1)$ random access into the structure, keeping the average number of probes required to retrieve an item below three. This method provides retrieval rates that are much faster than using binary searches, even though the probes are all uncoalesced.

Our single-level hash table has similar construction rates to the two-level hash table for large N, but uses only $1.25N$ space. However, ours is able to build at higher rates than the two-level hash table for smaller inputs (Figure 4.4). Moreover, when both tables occupy $1.42N$ space, the single-level hash table has a significantly higher construction rate for all input sizes since the algorithm encounters shorter eviction chains.

Except for inputs of size 16K or less, our retrievals complete more quickly than two-level hash table retrievals when the query keys all exist within the hash table. The likeliest reason is that the average number of probes for our retrieval is lower: the two-level hash table artificially limits the number of items that can be found with fewer probes, raising the average number of probes required to find an item.

The higher average is advantageous for the two-level hash table as its retrieval performance does not degrade significantly for failed queries. To test this performance, we built hash tables containing 10 million items then queried them with increasingly higher percentages of keys not in the table. The two-level hash table's retrieval timings linearly increase from 30 ms (when all queries are found) to 33.7 ms (when none are found). In contrast, our retrievals linearly increase from 20.5 ms to 41.6 ms because we use four probes in the worst case. When we use the same number of hash functions as the two-level hash table, our retrievals linearly increase from 19 ms to only 31.4 ms.

4.5.2 Hash Table Specializations

We tested the performance of our compacting and multivalue hash tables by repeatedly building them with datasets of a fixed size, but with an increasingly higher average number of times a key is repeated.

The compacting hash table was compared against two equivalent data structures: a structure based around sorted arrays and compacted lists, and our previous two-level compacting hash table [1] (Figure 4.5).

Both the single-level and two-level compacting hash tables take longer to build than the radix sorted structure; the biggest gap occurs when there are no duplicates in the list. These differences arise from the extra work required to process the input keys. Once there is some key repetition, construction times for both methods drop sharply. Although our single-level compacting hash table uses nearly the same algorithm, our faster cuckoo hashing procedure results in significantly smaller build times. Our retrievals also complete faster than the two-level version; until each key is repeated an average of 512 times, the combination of our construction and retrieval times is faster than using the sorted structure equivalent.

The multivalue hash table was compared against the equivalent two-level multivalue hash table, and a sorted structure based around sorted and compacted lists (Figure 4.6). To build the latter, we follow the construction procedure for the multivalue hash table until the construction of the hash table itself. Accessing the information for each key can then be done by employing binary searches rather than querying our hash table.

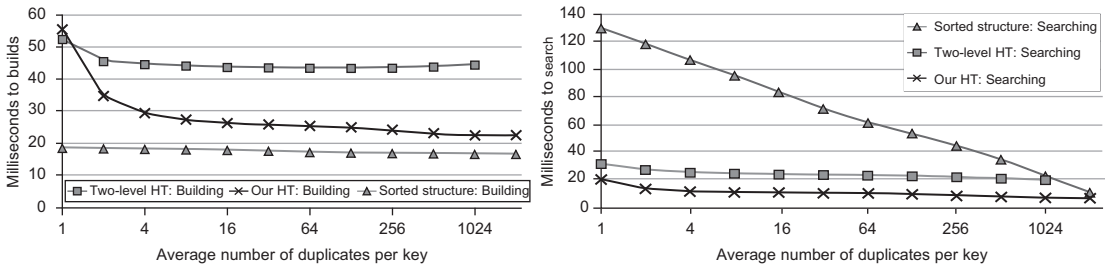

FIGURE 4.5

Compacting hash table timing comparison of construction (left) and retrievals (right) for inputs consisting of 10 million keys with increasing multiplicity on the keys. Each key was queried once for every time the key appeared in the input. Lower times are better.

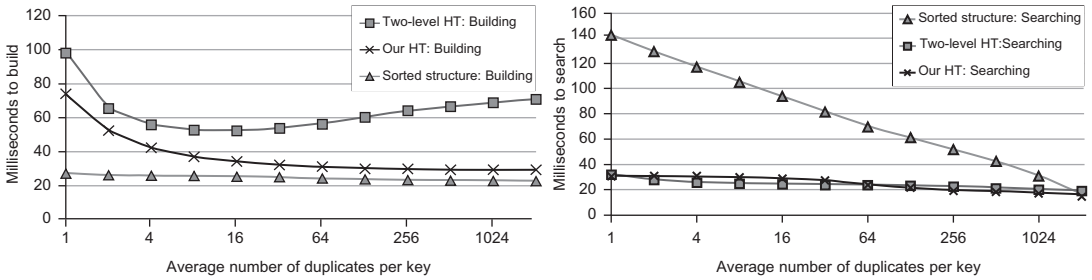

FIGURE 4.6

Multivalue hash table timing comparison of construction (left) and retrievals (right) for inputs consisting of 10 million pairs with increasing multiplicity on the keys. Each key was queried once for every time the key appeared in the input. Lower times are better.

Because construction of our hash table requires an additional step beyond construction of the sorted structure, our construction times are always slower. However, our retrievals are consistently faster than binary-searching the structure. Moreover, our hash table's size is dependent on the number of unique input keys, while the two-level version's size depends on the total number of input keys; as the average multiplicity of each key increases, the two-level multivalue hash table becomes more and more sparse.

4.5.3 Analysis of Performance Limits

We now focus our attention on the performance limits of our cuckoo hashing implementation. Because our implementation generates memory traffic that is largely random, we begin by discussing the performance bounds of random memory accesses on a GPU, then apply those lessons to our implementation. Finally, we compare our construction performance against that of radix sort.

How Fast Is Random Memory Access?

Modern dynamic random access memory (DRAM) and memory systems are optimized for workloads with substantial amounts of data access locality. However, good hash functions scatter input keys independently across the entire hash table, yielding very little locality. Given little locality, what are the limits to the throughput of our memory system?

Our hash tables are much larger than the GPU's cache, so given a random address stream, cache hits are unlikely. Thus (to first order), all memory accesses must go to DRAM. The pin bandwidth of the GTX 480 is 177.4 GB/s, so we could access 64-bit data no faster than 22.2 Gaccesses/s. However, with no spatial locality every access must be wrapped into an individual 32-byte transaction, which limits throughput to 5.5 Gaccesses/s.

From the DRAM side, the expected performance is even worse. With no spatial locality, we expect that each memory request will access a different page within each DRAM yielding an even tighter bound on the throughput of random memory accesses. For example, the 1 Gb GDDR5 DRAM from Hynix,[2] a DRAM of the same generation as the Samsung DRAMs used on the GTX 480, has a "32-bank activate window" (t_{32AW}) of 184 ns, meaning only up to 32 pages can be activated in a sliding window of 184 ns. The GTX 480 has 12 such chips, so DRAM page switches would limit performance to only 2.1 Gaccesses/s.

We constructed a benchmark to show that the actual throughput on the GTX 480 is roughly 1.3 Gaccesses/s (Figure 4.7, left). In this benchmark, we mimic successful hash insertion with no locality with strided access into an array. For an N-element array, we run N threads. Thread k reads

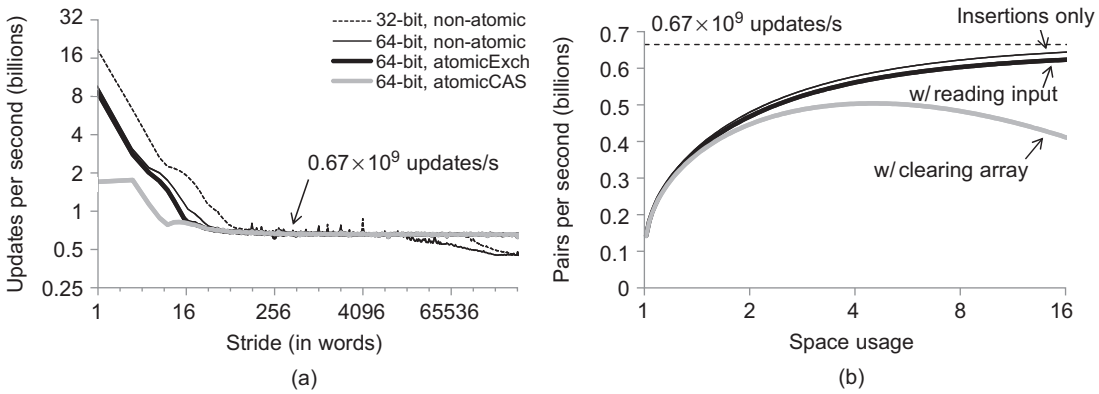

FIGURE 4.7

Left: Memory performance for strided access on the GTX 480. The array size is 512 MB. We used the `-Xptxas -cg` compiler option to enable 32 B memory transactions on reads. 32-bit `atomicInc`, `atomicExch` and `atomicCAS` performed similarly. The performance drop for nonatomic operations on the right side of the graph does not appear when the array size is under 256 MB. Right: Theoretical hash table construction updates per second on the GTX 480; our experimental results are consistent with these theoretical results.

[2]Hynix, "1Gb (32Mx32) GDDR5 SGRAM H5GQ1H24AFR", Rev. 1.0 / Nov. 2009.

an entry at location $k \cdot stride$ mod N of the array and overwrites it with a new value. Performing the read and write involves two memory accesses; we call this an "update." We choose a large stride that delivers no spatial locality and see that the access rate quickly converges to 0.67 Gupdates/s for both 32-bit/64-bit and atomic/nonatomic accesses. This GPU has an arithmetic throughput of 650 Ginstructions/s, so we can run up to 1000 instructions per memory update and remain memory bound. Consequently, arithmetic utilization and thread divergence have only marginal impact on hash performance.

How Many Random Accesses Does Our Hash Implementation Generate?

Most hashes, such as those based on chaining and open addressing, require at least one read and one write per hash-table element insertion. The read ensures that no previously inserted data is overwritten. The results of our benchmark suggests an upper performance bound of 0.67 Gpairs/s on the GTX 480.

In our cuckoo hashing implementation, we repeat insertions until no previously inserted key-value pair is evicted. How many iterations are done in total?

As an approximation, let us assume that parallel hashing does not involve more collisions than serial hashing. Then, for a table of size m, the probability of collision when inserting the kth entry is $p_k = (k-1)/m$. Using the well-known result for Bernoulli trials, we find that the expected value of the number of attempts required to find an empty slot is $1/(1-p_k)$. Summing this for $k = 1 \ldots N$ and approximating this sum with an integral, we conclude that roughly $-m\ln(1-N/m)$ attempts are required to insert N entries. For example, only $2N$ attempts are needed if running in space $m = 1.25N$, i.e., twice the minimum.

To see how well this theory matches practice, we use it to estimate the runtime of the hash construction. Our runtime also includes reading N input pairs and initializing m table entries, both with aligned, unit-stride access. We extract the access costs from Figure 4.7, left, and get $T_{\text{uncoalesced}} = 1.5\,\text{ns}$ for an update operation and $T_{\text{coalesced}} = 0.05\,\text{ns}$ for a unit-stride read or write access. The estimate is then:

$$T_{\text{construction}} = -m\ln\left(1-\frac{N}{m}\right)T_{\text{uncoalesced}} + (m+N)T_{\text{coalesced}}$$

The resulting rate $T_{\text{construction}}/N$ depends only on space usage m/N. This data is plotted in Figure 4.7, right, and is visually identical with our experimental results for N in the range of millions. The best performance (78% of the upper bound) is achieved with $4.5N$ space; half of the upper bound is achieved with $1.3N$ space.

Comparison with Radix Sort

Thus with these assumptions, hashing cannot sustain more than 670 Mpairs/s on the GTX 480. Surprisingly, radix sort achieves a higher rate of 775 Mpairs/s. Why is sorting, which does so much more work overall than hashing, faster than our hash table?

The answer lies in better spatial locality. Radix sort runtime is dominated by two kernels: the first reads keys contiguously and builds histograms, and the second reads the entire data set in contiguous blocks of 1024 entries and writes it out into 16 bins. Given that keys and values are in separate arrays, the average contiguous write size is 256 bytes per bin. This is substantially better than the 8 contiguous bytes per access we see in our hash table construction. The result is a radix sort implementation that achieves 70% of the theoretical maximum pin bandwidth, whereas our hash insertions sustain only 6% of this maximum.

4.5.4 Limitations

The algorithm we described and the parameters we chose strike a good balance between construction rates, retrieval times, and memory usage. However, there are still some drawbacks.

Retrieving a sorted set of queries can be optimized when using a sorted list structure because threads in the same warp will be likely to follow the same branches and are likely to have coalesced memory reads. Hash tables, in general, can't be optimized for these situations because they are designed to distribute items as evenly as possible across the structure.

Restarts are uncommon, but expensive because the entire table must be rebuilt from scratch effectively multiplying construction time by the number of attempts needed to build the table. If a consistent construction time is important, the parameters can be changed to make the table easier to build.

The maximum number of iterations can be hard to pin down. As the table size approaches the theoretical minimum for the given number of hash functions, the average number of iterations performed by the construction algorithm rises very quickly.

The compacting hash table becomes extremely sparse with high key multiplicity. If space usage is an issue, a procedure similar to the one followed for the multivalue hash table can be used, where the input is preprocessed using a radix sort. However, our construction can actually beat the radix sort times with smaller datasets; performing a radix sort beforehand will guarantee that construction is always slower.

4.6 CONCLUSION

We have presented a GPU hash table with a fast and parallel construction that provides random access at higher rates than binary searches through sorted arrays, despite the uncoalesced memory accesses inherent in hashing. Our design is more flexible than previous work on GPU hashes, allowing it to be tailored for different situations; the parameters we chose work well for general cases, striking a careful balance between the constraints of retrieval efficiency, construction time, and memory usage.

We see three major avenues for future work that will make GPU-based hash tables more useful for the community:

- We hope future work will use other hashing algorithms to address different tradeoffs, particularly to address building dense hash tables at fast rates and to optimize for the fastest possible lookup times.
- Our construction procedure could be extended to allow insertions into the table after the initial construction, but we avoided this in our implementation because of cases where an insertion fails. This would require rebuilding the entire table from scratch with the new items and could cause unpredictably large insertion times. Designing complex incremental parallel data structures for GPUs remains an active and interesting problem.
- We do not know how to handle input that exceeds the memory capacity of a single graphics card. Extending the algorithm to handle out-of-core input and multiple graphics cards is a challenging problem.

Acknowledgments

This research is supported by the National Science Foundation (Award IIS-0964473), Microsoft (Award 024263), and Intel (Award 024894), with matching funding by U.C. Discovery (Award DIG07-10227), and equipment donations by NVIDIA. Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. We also thank Daniel Vlasic for the 3D models.

References

[1] D.A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J.D. Owens, N. Amenta, Real-time parallel hashing on the GPU, ACM Trans. Graph. 28 (5) (2009) 154:1–154:9.

[2] S. Lefebvre, H. Hoppe, Perfect spatial hashing, ACM Trans. Graph. 25 (3) (2006) 579–588.

[3] R. Pagh, F.F. Rodler, Cuckoo hashing, in: 9th Annual European Symposium on Algorithms, in: F. Meyer auf der Heide (Ed.), vol. 2161 of Lecture Notes in Computer Science, Springer-Verlag, London, 2001, pp. 121–133.

[4] M. Mitzenmacher, S. Vadhan, Why simple hash functions work: exploiting the entropy in a data stream, in: S.-H. Teng (Ed.), SODA '08: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2008, pp. 746–755.

[5] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, M. Rink, Tight thresholds for cuckoo hashing via XORSAT, in: P. Spirakis (Ed.), 37th International Colloquium on Automata, Languages and Programming, Springer-Verlag, Berlin, 2010, pp. 213–225.

[6] D. Merrill, A. Grimshaw, Revisiting Sorting for GPGPU Stream Architectures, Technical Report CS2010-03, Department of Computer Science, University of Virginia, 2010.