Real-Time Parallel Hashing on the GPU

Dan A. Alcantara   Andrei Sharf   Fatemeh Abbasinejad   Shubhabrata Sengupta   Michael Mitzenmacher*
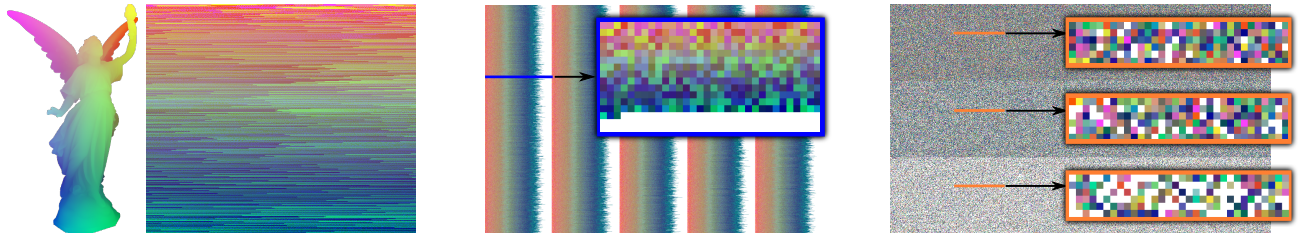
John D. Owens   Nina Amenta

University of California, Davis   *Harvard University

Figure 1: *Overview of our construction for a voxelized Lucy model, colored by mapping x, y, and z coordinates to red, green, and blue respectively (far left). The 3.5 million voxels (left) are input as 32-bit keys and placed into buckets of ≤ 512 items, averaging 409 each (center). Each bucket then builds a cuckoo hash with three sub-tables and stores them in a larger structure with 5 million entries (right). Close-ups follow the progress of a single bucket, showing the keys allocated to it (center; the bucket is linear and wraps around left to right) and each of its completed cuckoo sub-tables (right). Finding any key requires checking only three possible locations.*

Abstract

We demonstrate an efficient data-parallel algorithm for building large hash tables of millions of elements in real-time. We consider two parallel algorithms for the construction: a classical sparse perfect hashing approach, and cuckoo hashing, which packs elements densely by allowing an element to be stored in one of multiple possible locations. Our construction is a hybrid approach that uses both algorithms. We measure the construction time, access time, and memory usage of our implementations and demonstrate real-time performance on large datasets: for 5 million key-value pairs, we construct a hash table in 35.7 ms using 1.42 times as much memory as the input data itself, and we can access all the elements in that hash table in 15.3 ms. For comparison, sorting the same data requires 36.6 ms, but accessing all the elements via binary search requires 79.5 ms. Furthermore, we show how our hashing methods can be applied to two graphics applications: 3D surface intersection for moving data and geometric hashing for image matching.

Keywords: GPU computing, hash tables, cuckoo hashing, parallel hash tables, parallel data structures

1 Introduction

The advent of programmable graphics hardware allows highly parallel graphics processors (GPUs) to compute and use data representations that diverge from the traditional list of triangles. For instance, researchers have recently demonstrated efficient parallel constructions for hierarchical spatial data structures such as k-d

trees [Zhou et al. 2008b] and octrees [DeCoro and Tatarchuk 2007; Sun et al. 2008; Zhou et al. 2008a]. In general, the problem of defining parallel-friendly data structures that can be efficiently created, updated, and accessed is a significant research challenge [Lefohn et al. 2006]. The toolbox of efficient data structures and their associated algorithms on scalar architectures like the CPU remains significantly larger than on parallel architectures like the GPU.

In this paper we concentrate on the problem of implementing a parallel-friendly data structure that allows efficient random access of millions of elements and can be both constructed and accessed at interactive rates. Such a data structure has numerous applications in computer graphics, centered on applications that need to store a sparse set of items in a dense representation. On the CPU, the most common data structure for such a task is a *hash table*. However, the usual serial algorithms for building and accessing hash tables— such as chaining, in which collisions are resolved by storing a linked list of items per bucket—do not translate naturally to the highly parallel environment of the GPU, for three reasons:

Synchronization Algorithms for populating a traditional hash table tend to involve sequential operations. Chaining, for instance, requires multiple items to be added to each linked list, which would require serialization of access to the list structure on the GPU.

Variable work per access The number of probes required to look up an item in typical sequential hash tables varies per query, e.g. chaining, requires traversing the linked lists, which vary in length. This would lead to inefficiency on the GPU, where the SIMD cores force all threads to wait for the worst-case number of probes.

Sparse storage A hash table by nature exhibits little locality in either construction or access, so caching and computational hierarchies have little ability to improve performance.

While common sequential hash table constructions such as chaining have *expected* constant look-up time, the lookup time for *some* item in the table is $\Omega(\lg \lg n)$ with high probability. The influential work of Lefebvre and Hoppe [2006], among the first to use the GPU to access a hash table, addressed the issue of variable lookup time by using a *perfect hash table*. In this paper we define a perfect hash table to be one in which an item can be accessed in worst-

case $O(1)$ time (not to be confused with *perfect hash functions*, which simultaneously store a set of keys into a table with no collisions). Perfect hash tables in the past have been constructed on the CPU, using inherently sequential algorithms, and then downloaded, a cumbersome process that limits their application to static scenes.

Our major contribution in this work is a parallel, GPU-friendly hash table design that allows construction as well as lookup at interactive rates. We address the three problems above using hashing strategies that are both new to the GPU. The first is the classical FKS perfect hashing scheme [Fredman et al. 1984], which is simple and fast but not space-efficient. The second is the recently developed *cuckoo hashing* method [Pagh and Rodler 2001]. Cuckoo hashing is a so-called "multiple-choice" perfect hashing algorithm that achieves high occupancy at the cost of greater construction time. Our solution is a novel hybrid of these two methods for which construction and access times, in practice, grow linearly with the number of elements.

One big advantage of the hash table in sequential computing is that it is an inherently dynamic data structure; deletions and insertions have $O(1)$ expected time. But it is often unnecessary to have a dynamic data structure on the GPU: if any or all of the data items in the hash table change at every step, we simply rebuild the table from scratch. This is a heavyweight approach, but one congruent with a parallel processor with massive, structured compute capability; it is the approach we adopt here.

Hashing is, of course, not the only way to solve the problem of random access to sparse data. Another possible solution is to represent the sparse elements as a dense array. Construction requires sorting these elements; lookup requires a binary search on the sorted array. We compare our hashing scheme to a sorting scheme based on the GPU radix sort of Satish et al. [2009], the current highest-performing key-value sort on any commodity processor. Our hash-based solution is comparable in construction time and superior in lookup time, but uses, on the average, 40% more space. We also compare our hash table performance to our own implementation of the spatial hashing scheme of Lefebvre and Hoppe [2006]; lookup times between the two are similar, while our construction times are much shorter.

Hash tables with real-time construction and access times are useful in a variety of applications that require sparse data. We exercise our hashing scheme in two graphics applications: spatial hashing for intersection detection of moving data, and geometric hashing for feature matching. We demonstrate that a spatial hash on a virtual 128^3 grid can be constructed at each frame at around 27 fps. Geometric hashing is a GP-GPU application requiring intensive random access to a table of pairs of feature points. We demonstrate a performance improvement over an implementation based on a sorted list.

2 Related Work

Related Work on Hashing As a fundamental data structure, hash tables are a standard tool in computer graphics. Parallel hashing has been studied from a theoretical perspective, mostly in the early nineties [Matias and Vishkin 1990; Matias and Vishkin 1991; Bast and Hagerup 1991; Gil and Matias 1991; Gil and Matias 1998]. While these approaches achieve impressive results (guaranteeing $O(1)$ lookup time, $O(\lg \lg n)$ parallel table construction time, and $O(n)$ size) the implicit constant factors, especially on space, are unreasonable, and the constructions use features of the theoretical PRAM model not available on actual GPUs. In sequential hashing, there has been terrific progress since the mid-nineties, mostly centered around the study of "multiple-choice" hashing algorithms. We review this work in Section 3, since we build on these

ideas.

Related Work in Applications We use our parallel hash table construction in two applications on the GPU. The first is intersection detection using spatial hashing. The spatial hash representation is useful in computations that only accesses the neighborhood of the surface, e.g. the surface reconstruction method of Curless and Levoy [1996] or the narrow-band level-set method [Adalsteinsson and Sethian 1995]. Lefebvre and Hoppe [2006] advocated the use of spatial hashing as a surface representation on the GPU, an approach that has recently been adopted by others as well [Zhou et al. 2008c; Nehab and Hoppe 2007; Bastos and Celes 2008; Qin et al. 2008].

A second classic application of hash tables in graphics and computer vision is matching. Geometric hashing [Lamdan et al. 1988; Lamdan and Wolfson 1988; Lamdan et al. 1990] is the most basic of this family of techniques, which also includes the generalized Hough transform. It has been applied to a variety of matching scenarios over the years, including medical imaging [Guéziec et al. 1997], image mosaicking [Bhosle et al. 2002], fingerprint matching [Germain et al. 1997], repair of CAD models [Barequet 1997], mesh alignment and retrieval [Gal and Cohen-Or 2006], protein sequence matching [Nussinov and Wolfson 1991], and the construction of collages [Kim and Pellacini 2002]. Given a parallel hash table implementation, geometric hashing is highly parallelizable.

3 Background and our Approach

3.1 Perfect Hashing

Like prior work on GPU hash tables, we concentrate on perfect hash table constructions. The classic perfect hash table construction of Fredman et al. [1984] (the "FKS" construction) relies on the observation that if the size of the table, m, is much larger than the number of items n, specifically $m = \Theta(n^2)$, then with some constant probability a randomly chosen hash function will have no collisions at all, giving constant lookup time. To obtain constant lookup time with linear space, the FKS construction uses a two-level table. The top level hashes the items into buckets, and the bottom level hashes a bucket of k items into a buffer of size $O(k^2)$. As long as each bucket has only $O(1)$ expected items, the expected size of the hash table is $O(n)$ and the time to find each item $O(1)$. The main drawback is that to achieve reasonable table occupancy, say 1/4, requires many very small buckets.

Much subsequent work focused on constructing *minimal* perfect hash tables, which store n items in exactly n locations. Minimal or even near-minimal perfect hash tables reduce the space overhead at the cost of increased construction time. The spatial hash table construction used by Lefebvre and Hoppe [2006] was based on one of these approaches [Fox et al. 1992]. Such constructions are not only expensive, but they seem to be inherently sequential, since the location of an item depends on the locations already taken by earlier items. The FKS construction, on the other hand, is purely random and straightforward to implement in parallel; theoretically it requires $O(\lg n)$ time on a CRCW PRAM [Matias and Vishkin 1990].

3.2 Multiple-choice Hashing and Cuckoo Hashing

The key result behind multiple-choice hashing was presented in a seminal work by Azar et al. [2000], who considered the usual chaining construction and showed that using $d \geqslant 2$ hash functions and storing an item into the bucket containing the smallest number of items reduces the expected size of the longest list from

$O(\log n / \log \log n)$ to $O(\log \log n / \log d)$. Vöcking [2003] used multiple sub-tables, each with its own hash function, and reduced the expected size of the longest list to $O(\log \log n / d)$.

Cuckoo hashing [Pagh and Rodler 2001; Devroye and Morin 2003] places at most *one* item at each location in the hash table by allowing items to be moved after their initial placement. As with multiple-choice hashing, a small constant number $d \geqslant 2$ of hash functions are used, and it is convenient to think of the d tables as being split into sub-tables. The sequential construction algorithm inserts items one by one. An item first checks its d buckets to see if any of them are empty. (Here we assume each bucket can hold just one item; in settings where buckets hold multiple items, we check if any bucket has space.) If not, it "kicks out" one of the items, replacing that item with itself. This explains the name of the algorithm: in nature, the cuckoo chick kicks its step-siblings out of the nest. The evicted item checks its other possible buckets, and recursively kicks out an item if necessary, and so on. Although with an unfortunate choice of hash functions this process might continue forever without finding a location for every item, the probability of this is provably small. As long as every item finds a location, lookups take only constant time, as only d buckets must be checked.

For $d = 2$, the expected maximum number of steps required to insert an item is $O(\lg n)$, but unfortunately we can only achieve an occupancy of just less than one half. For $d = 3$, cuckoo hashing can achieve about 90% occupancy [Fotakis et al. 2005], but the theoretical upper bounds on insertion time have proven much more difficult for $d \geqslant 3$. A recent result shows that the expected maximum number of steps required to insert an item can be polylogarithmic for sufficiently large d, and it is believed that the expectation is actually logarithmic; Frieze et al. [2009] have more background and details.

3.3 Our Approach : Parallel Cuckoo Hashing

The main issue in parallelizing the cuckoo hashing algorithm is to define the semantics of parallel updates correctly, making sure that collisions between items are properly handled. We describe our algorithm using three sub-tables ($d = 3$), each of size $n(1 + \gamma)/3$, where γ is a suitably large constant. In the first iteration, we attempt to store every item into the first sub-table, T_1, by writing each item into its position in the table simultaneously. Our semantics require that exactly one write succeeds when collisions occur, and that every thread should be able to tell which one succeeded. We implement this by writing all items independently and then invoking a thread synchronization primitive, ensuring the contents of the table don't change while checking for success. Roughly $2n/3$ of the items fail; these attempt to write themselves into T_2 in the second iteration. Those that fail proceed to T_3. Finally, those that fail in T_3 return to T_1, evict the item occupying the location they want to occupy, and again try to write themselves into the now-empty locations. The evicted items, and those items which failed to find a location in T_1, continue to T_2. The iterations continue until all items are stored, or until a maximum number of iterations occur, in which case we decide that we have had an unfortunate choice of hash functions and we restart the process. Although for $d = 2$ we can adapt the proof for maximum expected insertion time for the sequential algorithm to show that the maximum expected number of iterations is $O(\lg n)$, it again appears much more difficult to get a theoretical upper bound when $d \geqslant 3$. Nonetheless we find that the number of iterations is reasonable in practice (see Section 5).

However, the parallel cuckoo hash would not be a good choice for building a large hash table on a GPU, since each iteration would require shuffling items in global memory with global synchronization at each iteration. But it is very useful for tables which can be com-

---

Algorithm 1 Basic hash table implementation

Phase 1: Distribute into buckets of size ≤ 512
 1: compute number of buckets required
 2: allocate output space and working buffers
 3: **for each** $k \in$ keys in parallel **do**
 4:   compute $h(k)$ to determine bucket b_k containing k
 5:   atomically increment $count[b_k]$, learning internal $offset[k]$
 6: **end for**
 7: perform prefix sum on $count[\,]$ to determine $start[\,]$
 8: **for each** key-value pair (k, v) in parallel **do**
 9:   store (k, v) in buffer at $start[b_k] + offset[k]$
10: **end for**
Phase 2: Parallel cuckoo hash each bucket
 1: **for each** bucket b in parallel, using one block per bucket **do**
 2:   build parallel cuckoo hash containing the items in b
 3:   write out tables T_1, T_2, T_3 and hash functions g_1, g_2, g_3
 4: **end for**

---

puted in a small and fast on-chip memory. Thus we adopt a hybrid method. As in the FKS scheme, we find a first-level hash function to divide the items into smaller buckets. Within each bucket, we then use parallel cuckoo hashing to place the items into three sub-tables. Since the cuckoo hashing is performed entirely within the on-chip memory, it is very fast in practice.

4 Implementation

In this section we describe our implementation, which uses NVIDIA's CUDA programming environment and targets GPUs that feature atomic global memory operations.

4.1 Basic hash table

We assume the input is a set of n integer key-value pairs (items), where all of the keys are unique. Our build process is summarized in Algorithm 1, and consists of two phases. Phase 1 distributes the items into buckets using a hash function h, shuffling the data so that each bucket's items are in a contiguous area of memory; this improves memory access patterns for phase 2. Phase 2 then builds a separate cuckoo hash table for each bucket in parallel using fast shared memory, then writes out its tables to global memory into a single array with the keys and values interleaved.

Phase 1 The main goal of the first phase is to distribute the items into a set of buckets each containing at most 512 items; this limit allows each thread to manage at most one item and keeps each bucket's cuckoo table small enough to fit in shared memory. We begin in Step 1 by estimating the number of buckets necessary. To lower the risk of overfilling any buckets, the number of buckets is set to $\lceil n/409 \rceil$ so that each is filled to 80% capacity on average. This is based on our estimation of the binomial distribution for a large number of keys using a Poisson distribution, which says that the probability of a given bucket having more than 512 items is less than one in a million (empirical numbers are shown in section 7). We then allocate all memory required for both phases in Step 2. The rearranged data will reside in a single buffer, with the items of each bucket b beginning at index $start[b]$.

In Steps 3–6, we launch a kernel assigning each item to a bucket according to h. We set $h(k) = k \bmod |buckets|$ for our first attempt, based on the observation of Lefebvre and Hoppe [2006] that even though it is not random it seems to work and improves locality; there was a small, but noticeable, improvement in our retrieval

times. If needed, subsequent attempts set

$$h(k) = [(c_0 + c_1 k) \bmod 1900813] \bmod |buckets|$$

where the c_i are random integers and 1900813 is a prime number.

Each key k computes $h(k)$ to determine the bucket b_k that it falls in. Next, we simultaneously count the number of items that fall into each bucket, and compute an offset into the bucket where each item will be stored. The $count[\,]$ array stores the size of each bucket, with each k atomically incrementing $count[b_k]$. The thread stores the value before the increment as $offset[k]$; this will be the offset at which k is stored in the area of memory allotted for bucket b_k. Although these atomic increments are serialized, we expect at most 512 of them per counter regardless of the input size. If any bucket is found to have more than 512 items in it, we pick a new h, and repeat these steps.

In step 7, we determine the addresses of each item in the working storage buffer that will hold the rearranged data. Using a prefix sum operation from the CUDA Data Parallel Primitives library [Sengupta et al. 2007], we compute the starting location for each bucket. Finally, in step 8, a kernel copies the key-value pair (k, v) into $start[b_k] + offset[k]$. We interleave the key and value for each pair in the array to take advantage of write coalescing.

Phase 2 Phase 2 works on each bucket b independently, using a block of 512 threads to parallel-cuckoo-hash all items that fall within b. This phase uses the shared memory extensively to build the hash table. We seed the construction with pseudo-random numbers generated on the CPU for the hash functions g_1, g_2, g_3, each associated with a sub-table T_1, T_2, T_3 and of the form
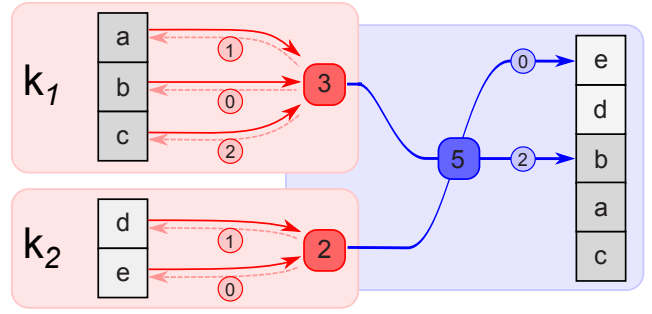
$$g_i(k) = [(c_{i0} + c_{i1}k) \bmod 1900813] \bmod |T_i|$$

All 6 constants c_{ij} are generated by XORing a single random number seed with different fixed numbers.

As described in Section 3.3, all threads begin by simultaneously writing their keys into T_1 using g_1, disregarding any item already in the slot. After synchronizing, each key checks whether it was evicted, or failed to write, by comparing itself to the key occupying its slot in T_1. All unplaced keys perform the same operation again, iteratively trying each sub-table in order until either all items place themselves in the table or too many iterations (more than 25) are taken. In the latter case, we assume that we have made an unlucky choice of g_1, g_2, g_3, and that a cycle of items prevents completion. In this case we choose new hash functions for the bucket, and begin again. Empirically, a bucket will loop through the sub-tables an average of 5.5 times before succeeding. Only a few buckets fail to build with the first hash functions, and these typically succeed after restarting once.

The size of our tables at the second level is $3 \times 192 = 576$ for at most 512 items; this is sufficient to ensure that up to 512 items can be successfully hashed using three choices with sufficiently high probability (based on CPU simulations, since the theoretical bounds here are not at all tight). The overall load factor of our structure is thus $409/576 \approx 71\%$.

Finally, in step 3, we copy the completed hash table out to global memory from shared memory, interleaving the values with their associated keys. The completed hash tables are written so that the T_1 for all buckets are contiguous, followed by all of T_2, then all of T_3 (see Figure 1). This encourages parallel retrievals to concentrate on the same area of memory rather than search across the entire array. We also store the seed each bucket uses to create g_1, g_2, and g_3.

Figure 2: *Laying out the array of values for multi-value hashing. Each key reserves space for its values by learning how many values c_k it has. Its values atomically increment c_k, simultaneously giving them unique offsets o_v within their key's space (shown in red). Space is then allotted by atomically adding each c_k to a counter tracking the number of currently reserved slots, returning the index i_k where a key's values should begin (blue). Values finally get copied into the location $i_k + o_v$ (right).*

Retrieval Retrieval for a key k requires calculating b_k using $h(k)$. The bucket's seed is read to recreate g_1, g_2, g_3, which are then used to determine where k could be stored. The three lookups are performed by the same thread, which stops looking as soon as k is found.
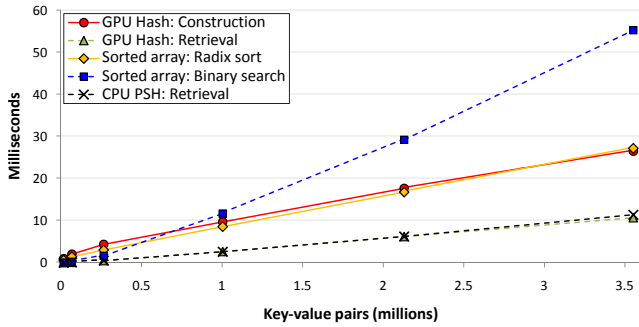
4.2 Multi-value hashing

Many applications require adding the same key into the hash table multiple times with different values. For example, in geometric hashing (Section 6), we compute a collection of signatures from an image, where a specific image signature can be found multiple times in the same image. Using the signature as a key, we want to store a list of the places in which it is found as the value.

Using the shared memory atomic operations available on NVIDIA GPUs with compute capability 1.2, we can construct such a multi-value hash table. Specifically, our parallel multi-value construction produces a hash table in which a key k is associated with a count c_k of the number of values with key k, and an index i_k into a data table in which the values are stored in locations $i_k \ldots i_k + c_k - 1$.

Phase 1 of the construction algorithm proceeds similarly to the case of the basic hash table, with all values sharing the same key directed into the same bucket. While it is still likely that each bucket contains at most 512 unique keys, the actual number of items may now exceed 512. We handle this by allowing each thread to manage more than one item in phase 2. Note that unlike for the basic hash table, we cannot determine how many unique keys there are until attempting to build the cuckoo tables.

Phase 2 begins with each thread attempting to write all of its keys into T_1, then synchronizing and checking which of its keys were successfully stored. When multiple attempts are made to store the same key, the key is defined to be stored if any one of them succeeds. The process then proceeds as before, cycling through each sub-table. Repeated failure to build the cuckoo hash tables at this point likely indicates an overfilling of the bucket, which causes a complete restart from phase 1. Once cuckoo hashing has established a place for every key k, we rearrange the values so that each key's data is contiguous. The process is analogous to the one followed in phase 1, steps 3–10 of the basic algorithm, but uses shared memory atomics in place of the global atomics and the prefix sum; see Figure 2. Afterward, each k is associated with its i_k and c_k in the cuckoo hash table, which then gets written to global memory.

Figure 3: *Timings generated for increasingly larger voxelized representations of the Lucy model. Note that our retrievals are consistently faster than using a binary search, with our construction time edging ahead of radix sort for the 3.5 million item case.*

Figure 4: *Timings generated for increasingly large sets of randomly sampled voxels from a 1024^3 grid. Note that our hash build times are comparable with a radix sort. For increasingly larger datasets, our retrieval times remain on par with PSH retrievals while the binary search gets worse; this reflects the smaller number of trips to memory that a hash retrieval must perform.*

4.3 Compacting unique keys

Like sorting, hashing can be used to aggregate and compact data. One typical scenario is first generating or selecting keys in parallel, and then counting the number of unique keys created and compacting them into an array. To accomplish this, we use a simplified version of our multi-value hashing procedure that avoids additional atomics. Everything proceeds the same until the cuckoo step is completed, after which each key is associated with a value of one; slots without keys receive values of zero. We write the cuckoo tables out to global memory and then perform a prefix sum on the values, simultaneously producing a unique index for each key and counting the number of unique keys. The keys are then compacted into a new array using these unique indices.
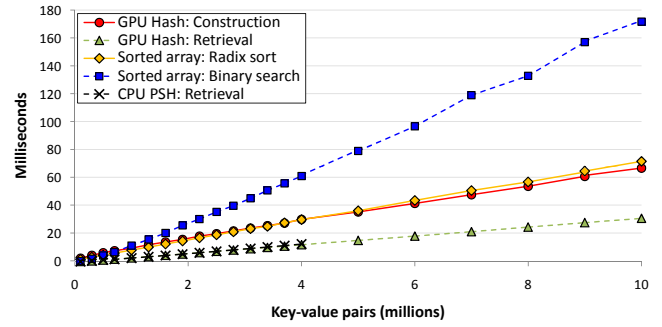
Together, the array of unique keys and the hash table gives a mechanism for translating between keys and their indices in constant time, in both directions. We call this a *two-way index*. Of course, a two-way index could also be constructed by compacting a sorted list of keys. But while the sorted list version can also translate indices to keys in constant time, translating keys to indices would require a binary search.

5 Results

We compare the performance of our hash table against two other methods: a binary search on a radix sorted array [Satish et al. 2009], and an implementation of perfect spatial hashing (PSH) [Lefebvre and Hoppe 2006]. Times are reported using CUDA 2.2 under Ubuntu Linux for an EVGA GeForce GTX 280 SSC, with a core clock speed of 648 MHz. Due to the inherent non-predictability of GPU execution times and the randomness of our hash construction, timings are averaged over 100 runs for all data points.

Our space-optimized implementation of PSH performs retrievals on the GPU, but builds on the CPU. Binary search is used to find the optimal size of the offset table, resulting in a typical space usage of $1.16n$. This leads to construction times several orders of magnitude slower than the other methods, which are omitted from our graphs.

Voxelized Lucy In Figure 3, we examine performance against increasingly finer voxel representations of the Lucy dataset. Each voxel is represented as 30 bits of a 32-bit integer, with the z-coordinate taking the lowest 10 bits. We note that both radix sort and our hash have very similar construction times, with hashing nudging ahead of radix sort as the number of voxels increases. However, this comes at the cost of increased storage for hash tables. In no case does this overhead exceed $1.42n$ of input data, arising from our 71% overall load.
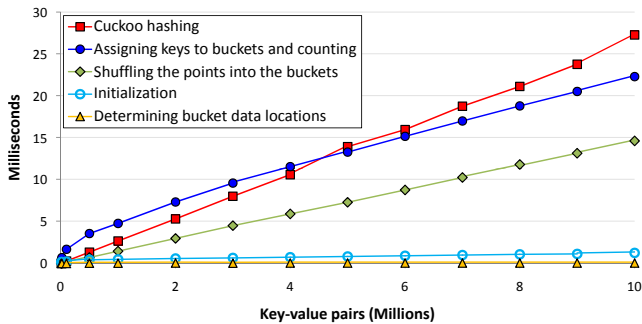
Radix sort greatly reduces uncoalesced input/output by first sorting in shared memory and then writing out data to global memory. While our hash algorithm reads data in a perfectly coalesced fashion, the writes at both levels are highly uncoalesced. Our final cuckoo tables could actually be written out in a coalesced manner, by having each thread write out contiguous elements of the subtables. However, we found that it was actually faster to write out only the occupied entries uncoalesced.

Search times are compared by searching for all voxels in a random order. A binary search is performed to find a key in the sorted array, which takes $\log(n)$ steps on average. Looking up a key in a hash table takes a constant number of steps and computation of three hash functions, though it is uncoalesced (again due to the randomizing nature of the hash function), while there exists a fair degree of coalescence in the initial steps of the binary search. As the number of keys searched for increases, the constant cost of the hash table lookup improves in comparison to the binary search. There is a point around 300k items after which the combined time of our construction and retrieval is shorter than combined time of the radix sort and the binary search.
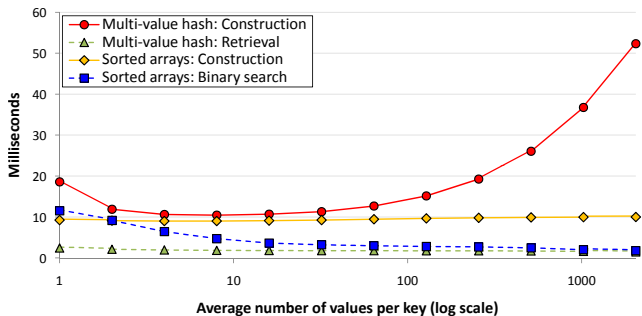
Randomized voxels Figure 4 shows the performance on random voxels drawn from a 1024^3 grid, with each voxel represented as a 32-bit integer. We search for all the voxels in a randomized order. We compare with PSH up to only 4 million items, as PSH's construction time becomes impractical beyond this point. Our retrieval times are on par with PSH retrievals, which trades off our faster construction time for our increased storage requirements. The binary search consistently does worse than our retrievals, while our construction time is comparable to that of the radix sort, overtaking it after 5 million items.

To investigate this behavior, we break down the cost of our hash construction in Figure 5, which shows that most steps grow linearly with input size. There is an initial ramping up cost with the bucket assignment performed in step 3, causing the construction to be slower for cases under 5 million items, as seen in Figures 3 and 4. Afterward, its relative effect on the construction time lessens, making the cuckoo hashing step the most expensive.

Multi-value hash We also analyzed the performance of our multi-value hash on random sets of 1 million items, plotting the

Figure 5: *Timing for various steps of the hash table construction generated on random voxel data. Note that timings are roughly linear for most stages.*

Figure 6: *Construction and retrieval times for the multi-value hash as a function of the average number of values per key, shown with a logarithmic x-axis. The data consisted of randomly sampled voxels from a 1024^3 domain. The build times for high multiplicity represent the overhead incurred by the extra work incurred to organize the key-value pairs within each bucket.*
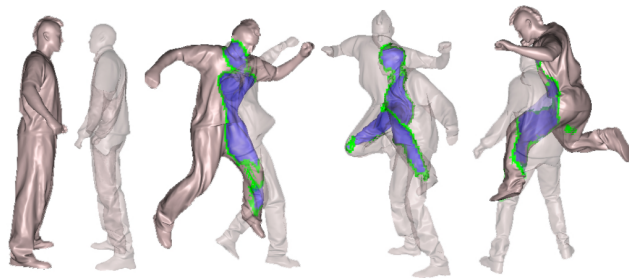
construction and retrieval times against the average number of values per key (Figure 6). One query is performed on each key for each of its values. For comparison, we generate an equivalent data structure utilizing a radix sort and compaction of the unique keys to enable binary search. Retrieval performs a binary search over the compacted keys, giving both the location of the key's first value as well as the number of values it has.

Although there is an overhead incurred by this version when each key has only 1 value, construction time steeply drops as keys have 2 values on average. At this point, values with the same key end up in the same bucket and have a simpler parallel cuckoo build, with fewer iterations occurring due to the looser definition of "eviction" (Section 4.2). Construction time increases after a key has 32 values on average, at which point the extra work and shared memory atomics used in phase 2 (and to a much lesser extent, the global memory atomics used in phase 1) become prohibitively expensive. Our retrievals are competitive with binary searches up to this point; it is an open problem to do better with higher multiplicities.

6 Applications

In order to evaluate our hash table constructions in practice, we have implemented two classic geometric applications of hash tables. Prior work on both applications was described in Section 2.

Spatial hashing When placed in a voxel grid, a typical surface occupies only a small fraction of the cells. Storing the data for every voxel in the grid would then result in an extremely sparse

Figure 7: *Our hash tables are built and queried every frame for two animated point clouds (left), allowing interactive Boolean operations between the two surfaces. Blue parts of one model represent voxels inside the other model, while green parts mark surface intersections.*

data structure. A more space-efficient approach would store only the information on the occupied cells. Spatial hashing does this using a hash table, which still allows voxel lookup in constant time; failure to find a voxel implies that it does not contain any part of the surface.

To intersect two surfaces, we look up each occupied voxel of one surface to see if it is also occupied for the other surface. Because we can build the hash table on the GPU, we can construct and intersect the spatial hashes in real time. Furthermore, we can detect which parts of a surface are inside or outside of the other surface (see Figure 7). We first determine the inside-outside status of voxels near the intersection, then propagate this information along the surface voxels to determine their status. This allows us to display Boolean combinations of two animated surfaces in real-time, while the user modifies the viewpoint interactively.

We upload two point clouds with normals, representing the two models, to the GPU at each frame. For both models, we assign a thread to each input point, perform any user-specified rotation and translation, and then round the resulting coordinates to identify the voxel containing the point. Since there are multiple points per voxel, we use the compacting hash table as described in Section 4.3 to assign a unique index to each voxel; these indices point into an array of information gathered for each voxel, including its current status and normal.

The next step intersects the voxelized surface representations and robustly finds voxels inside or outside of the other surface. A thread for each occupied voxel x first checks which of its 6 neighboring voxels are also occupied by looking in its surface's hash table. It then looks itself up in the other spatial hash. If the corresponding voxel x' on the other surface is also occupied, x is marked as an intersection. After all intersections are marked, we examine their unmarked neighbors to see which are likely inside or outside the other surface. For a neighbor voxel y, we take the dot product between the vector from x to y and the normal at x'. Intuitively, this compares the tangent of one surface with the normal of the other. Negative values show the surface is entering the other surface towards y, while positive values indicate it is exiting. Because the coarseness of the voxelization only allows approximation of the tangent, we set two thresholds representing our confidence: y is initialized as inside if the dot product is below the first threshold and outside if above the second. Let M be the set of these initialized voxels.

We follow with a flood-fill kernel to propagate the inside-outside status to the rest of the surface, again using one thread per occupied voxel. Each unmarked voxel repeatedly checks the status of its neighbors. If the voxel finds an inside or outside neighbor, it

Figure 8: *Matching two photographs of a carved relief image from Persepolis. The images differ in lighting, scale, and by a slight rotation. Our entirely GPU geometric hashing algorithm aligns them in 2.38 seconds; the same algorithm using sorted lists requires 2.86 seconds.*

takes on that status, and also copies from that neighbor the ID of the initialized voxel from M which initialized the chain of propagation. The kernel iterates and is re-run until all threads stabilize, indicating that no further propagations will occur.

As is typical with flood-filling, if the initialization was incorrect the results might be wrong. We do a final check, using another kernel, to see if any inside cell is a neighbor of an outside cell; if so the initialized cells in M causing the conflict are cleared and the flood-fill is restarted from the remaining cells of M. This has produced good results in the examples we tried.

We normally use a 128^3 voxel grid for point clouds of approximately 160k points, though we use a 256^3 voxel grid for one of the datasets in the accompanying video. We achieve frame rates between 25–29 fps, with the actual computation of the intersection and flood-fill requiring between 15–19 ms. Most of the time per frame is devoted to actual rendering of the meshes.

Geometric hashing Geometric hashing is a well-known image matching technique from computer vision, which tries to find a query image within a reference image. A set of feature points is chosen in both images, with pairs of different points called *bases*. Bases from the two images form a match if, when the images are rotated, translated and scaled so as to align the two bases, many other feature points are aligned as well. Essentially, geometric hashing is a brute-force search for matches that uses large hash tables to store the accumulating information.

We begin by applying a Sobel edge detection filter to the reference image. Pixels detected as edges (using a threshold) are marked, and then compacted into a contiguous array R of k feature points. Any basis $r_i, r_j \in R$ defines a 2D coordinate system where a rescaled $\overrightarrow{r_i, r_j}$ forms the x-axis: r_i is placed at the origin and r_j is given a fixed, positive x-coordinate. The coordinate system is quantized, so that it assigns integer coordinates (u, v) of every other point p of R, with all points within the same unit square receiving the same coordinates. Note that any labeled triple of points in *either* image that is similar (in the geometric sense) to (r_i, r_j, p) produces the same code (u, v).

Any match between the sets of feature points will be represented by multiple pairs of matching bases, so we need not consider all k^2 possible bases in each image. Instead, we take each pair of feature points $r_i, r_j \in R$ as a basis with probability $\frac{1}{5}$ (the constant is somewhat arbitrary; we found that this choice did not degrade quality and greatly reduced the size of the hash tables). We further ignore bases for which the distance between r_i and r_j is very small or very large, since the information they provide is not very salient: the accompanying transformations would either spread the points out too thinly or concentrate them into the same few (u, v) coordinates. The n remaining bases are compiled into a two-way index by using a compacting hash table (Section 4.3); this allows us to find either the location of a given basis, or the basis at a given location, in constant time.

Next, we compute all of the possible (u, v) pairs for the image. Using one thread per basis, we iterate through the set R of feature points. For each, we compute and store an item with key (u, v) and the value (r_i, r_j) in a multi-value hash table (Section 4.2). Each of these key-value pairs represents the image transformation causing a feature point to appear at (u, v); notice that each key will likely have many values.

We then similarly compute a set Q of feature points for the query image, select m random bases from the points, and again build a two-way index for the set of chosen bases using the compacting hash table. Afterward, we assign a thread to each chosen basis (q_k, q_l) from the query image and iterate through the points of Q. For each point, we compute coordinates (u, v) with respect to the basis and look them up in the multi-value hash table. If we find that (u, v) is present as a key, we iterate through the set of bases from the reference image stored as its value list, casting a vote for the correspondence between (q_k, q_l) and each (r_i, r_j). Correspondences with many votes are candidates for the transformation needed to align the two images.

Since almost every pair of bases receives some votes, we store the votes in an array T of size $n \times m$ in global memory. To address the location for a vote in T, we need the index for (r_i, r_j) and the index for (q_k, q_l), which we get using the two-way indexing hash tables. Votes are then cast by performing an atomic addition on the correct element in T. The atomic is required because multiple threads can vote for the same basis pair, but contributes to making this the most expensive step of the algorithm. After all votes have been cast, we use a reduce operation on T to find the pair of bases that received the most votes. If the number of votes is above a threshold, we output the transformation aligning the two bases as a possible correspondence between the reference and query images.

One aligned example is shown in Figure 8, with timings in Table 1. The Table also shows average timings for matching each frame of a video to an image, as well as comparisons between a hash table and sorted list based approaches. Aligned images for these are shown in our accompanying video. While geometric hashing successfully reports the correct match as the best correspondence in these examples, it can fail to find a good match even when one exists if there are regions where edge pixels are densely distributed (such regions match any curve in the query image).

This application stresses the hash tables in ways different than for spatial hashing. First, the multiplicity for the multi-value hash table is highly variable: the Kremlin dataset has an average of 105 values per key with a maximum of 5813, while for Figure 8 the average is 18 with a maximum of 947. Building the hash table for the query

in Figure 8 required 44 ms for 3.02 million items, roughly twice the time required for the same number of unique random items.

The extra time required for construction is offset by our retrievals, the number of which is much higher for both hash table types. For a query image with k feature points, $O(k^3)$ queries are passed to the multi-value hash for voting, with each vote needing to use the two-way index to address into T. The advantage of the hash table implementation over the sorted list implementation becomes more prominent as the number of items increases.

Dataset	# Items	Avg. Matching Time Per Frame (ms)	
		Hash Table	Binary Search
Persepolis Images	$\sim$3M	2377	2858
Matt Video—Kremlin	$\sim$4M	3025	4969
Matt Video—Taj Mahal	$\sim$440K	1437	1735
Matt Video—Easter Island	$\sim$550K	1721	2095

Table 1: *Times required to match an image to a pre-processed query; most of the time is spent voting. The number of items reported is the number of key-value pairs inserted into the multi-value hash table. The time to build the data structures for the reference image are three orders of magnitude smaller.*

7 Limitations

While our work establishes that the hash table is a viable data structure on the GPU, we believe that there is much more work to be done in this area.

Space usage As noted in section 4.1, we set the number of unique keys falling within a bucket to 512, leading to an overall load factor of 71%. While one limiting factor here is the amount of shared memory available, it is still possible to allow more than 512 per bucket. Doing so allows tighter packing of the data while maintaining a low restart probability. In preliminary experiments, we found that increasing the number of items managed by a bucket to 1024 allowed raising the overall load factor to 75%. This had a minor effect on our retrieval timings, but did increase our construction times, reflecting the extra work for building the denser cuckoo tables.

Related to this is the fixed cuckoo table size for all buckets. Regardless of the number of items contained within each bucket, the tables must account for the maximum number of unique keys possible. This is more problematic in our multi-value and compacting hashing implementations, where higher multiplicities lead to sparser tables. Future work will explore dynamically sized tables to increase the overall load factor. The trade-off is the extra bookkeeping that must be done in both the construction and retrieval stages to determine where each bucket's tables are stored in memory.

Restarts Empirically, we have seen buckets overfilled on 22 out of 25000 trial runs (0.088%) for 1 million random items of the basic hash table, with an increase of average construction time from 9.78 to 14.7 ms. For the 5 million item case, restarts occurred for 125 out of 25000 runs (0.5%), with an increase of time from 35.7 to 49.4 ms. It is possible to decrease restarts in the first phase by lowering the number of average number of items per bucket, at the expense of further space overhead. Alternatively, one could use the power of multiple choices for this first phase to both improve space utilization and reduce restarts. For example, one could allow items to choose from the less loaded of two different buckets in this first phase. While this would double the number of places to check when performing retrievals (as both buckets would need to be

considered), it would lead to much tighter space usage and tighter concentration of the maximum bucket load.

The penalty is higher for the other hash table types, as we can't know how many unique keys fall into any given bucket until the cuckoo tables are built. If a bucket fails to build its tables, the whole process must restart from Phase 1. However, the chance of this happening decreases as multiplicity increases: because the amount of buckets allocated is dependent on the total number of items and not the number of unique keys, it becomes more difficult to overfill a bucket.

Ordered retrievals Hash tables in general have problems with locality; items that are logically sequential might be stored very far apart. Thus hash tables are often not a good choice for retrieving an ordered set of items. Parallel binary searches on an ordered set of items are more branch-friendly and their reads can be coalesced well on current hardware. As a rule of thumb, if the data is to be accessed sequentially, a sorted list may be the better data structure, while a hash table may be the better choice if the data is to be accessed randomly.

8 Conclusions

We have demonstrated that hash tables with millions of elements can be successfully constructed and accessed on the GPU at interactive rates. Our hybrid hash table builds on modern ideas from the theory of hashing. We see similar construction times, and better lookup times, compared to the approach of representing sparse data in an array using a state-of-the art sort. One of the most interesting outcomes of our work is the tradeoff between construction time, access time, and space requirements. The implementation we describe here balances these three metrics, but applications that (for example) only require infrequent updates, or that are not limited by storage space, may choose to make different design decisions.

Different applications also require different hash table features. The most efficient implementation of multiple-value hashing is not necessarily going to be the most efficient implementation when the multiple values can be aggregated (i.e. by counting or averaging), and yet another implementation might be more efficient when unique keys are guaranteed. A standard data structures library may have to include all of these specialized variants, and perhaps others.

References

ADALSTEINSSON, D., AND SETHIAN, J. A. 1995. A fast level set method for propagating interfaces. *Journal of Computational Physics 118*, 2, 269–277.

AZAR, Y., BRODER, A. Z., KARLIN, A. R., AND UPFAL, E. 2000. Balanced allocations. *SIAM Journal on Computing 29*, 1 (Feb.), 180–200.

BAREQUET, G. 1997. Using geometric hashing to repair CAD objects. *IEEE Computational Science & Engineering 4*, 4 (Oct./Dec.), 22–28.

BAST, H., AND HAGERUP, T. 1991. Fast and reliable parallel hashing. In *ACM Symposium on Parallel Algorithms and Architectures*, 50–61.

BASTOS, T., AND CELES, W. 2008. GPU-accelerated adaptively sampled distance fields. In *IEEE International Conference on Shape Modeling and Applications*, 171–178.

BHOSLE, U., CHAUDHURI, S., AND ROY, S. D. 2002. The use of geometric hashing for automatic image mosaicing. In *Proceedings of the National Conference on Communication*, 533–537.

CURLESS, B., AND LEVOY, M. 1996. A volumetric method for building complex models from range images. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, 303–312.

DECORO, C., AND TATARCHUK, N. 2007. Real-time mesh simplification using the GPU. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, 161–166.

DEVROYE, L., AND MORIN, P. 2003. Cuckoo hashing: Further analysis. *Information Processing Letters 86*, 4, 215–219.

FOTAKIS, D., PAGH, R., SANDERS, P., AND SPIRAKIS, P. 2005. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems 38*, 2, 229–248.

FOX, E. A., HEATH, L. S., CHEN, Q. F., AND DAOUD, A. M. 1992. Practical minimal perfect hash functions for large databases. *Communications of the ACM 35*, 1 (Jan.), 105–121.

FREDMAN, M. L., KOMLÓS, J., AND SZEMERÉDI, E. 1984. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM 31*, 3 (July), 538–544.

FRIEZE, A., MITZENMACHER, M., AND MELSTED, P. 2009. An analysis of random-walk cuckoo hashing. In submission.

GAL, R., AND COHEN-OR, D. 2006. Salient geometric features for partial shape matching and similarity. *ACM Transactions on Graphics 25*, 1 (July), 130–150.

GERMAIN, R. S., CALIFANO, A., AND COLVILLE, S. 1997. Fingerprint matching using transformation parameter clustering. *IEEE Computational Science & Engineering 4*, 4 (Oct./Dec.), 42–49.

GIL, J., AND MATIAS, Y. 1991. Fast hashing on a PRAM—designing by expectation. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, 271–280.

GIL, J., AND MATIAS, Y. 1998. Simple fast parallel hashing by oblivious execution. *SIAM Journal of Computing 27*, 5, 1348–1375.

GUÉZIEC, A. P., PENNEC, X., AND AYACHE, N. 1997. Medical image registration using geometric hashing. *IEEE Computational Science & Engineering 4*, 4 (Oct./Dec.), 29–41.

KIM, J., AND PELLACINI, F. 2002. Jigsaw image mosaics. *ACM Transactions on Graphics 21*, 3 (July), 657–664.

LAMDAN, Y., AND WOLFSON, H. J. 1988. Geometric hashing: A general and efficient model-based recognition scheme. In *Second International Conference on Computer Vision (ICCV)*, 238–249.

LAMDAN, Y., SCHWARTZ, J. T., AND WOLFSON, H. J. 1988. On recognition of 3-D objects from 2-D images. In *IEEE International Conference on Robotics and Automation*, vol. 3, 1407–1413.

LAMDAN, Y., SCHWARTZ, J. T., AND WOLFSON, H. J. 1990. Affine invariant model-based object recognition. *IEEE Transactions on Robotics and Automation 6*, 5 (Oct.), 578–589.

LEFEBVRE, S., AND HOPPE, H. 2006. Perfect spatial hashing. *ACM Transactions on Graphics 25*, 3 (July), 579–588.

LEFOHN, A. E., KNISS, J., STRZODKA, R., SENGUPTA, S., AND OWENS, J. D. 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics 26*, 1 (Jan.), 60–99.

MATIAS, Y., AND VISHKIN, U. 1990. On parallel hashing and integer sorting. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, 729–743.

MATIAS, Y., AND VISHKIN, U. 1991. Converting high probability into nearly-constant time, with application to parallel hashing. In *ACM Symposium on the Theory of Computing (STOC)*, 307–316.

NEHAB, D., AND HOPPE, H. 2007. Texel programs for random-access antialiased vector graphics. Tech. Rep. MSR-TR-2007-95, Microsoft.

NUSSINOV, R., AND WOLFSON, H. J. 1991. Efficient detection of three-dimensional structural motifs in biological macromolecules by computer vision techniques. In *Proceedings of the National Academy of Sciences of the United States of America*, National Academy of Sciences, vol. 88, 10495–10499.

PAGH, R., AND RODLER, F. F. 2001. Cuckoo hashing. In *9th Annual European Symposium on Algorithms*, Springer, vol. 2161 of *Lecture Notes in Computer Science*, 121–133.

QIN, Z., MCCOOL, M. D., AND KAPLAN, C. 2008. Precise vector textures for real-time 3D rendering. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, 199–206.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Graphics Hardware 2007*, 97–106.

SUN, X., ZHOU, K., STOLLNITZ, E., SHI, J., AND GUO, B. 2008. Interactive relighting of dynamic refractive objects. *ACM Transactions on Graphics 27*, 3 (Aug.), 35:1–35:9.

VÖCKING, B. 2003. How asymmetry helps load balancing. *Journal of the ACM 50*, 4 (July), 568–589.

ZHOU, K., GONG, M., HUANG, X., AND GUO, B. 2008. Highly parallel surface reconstruction. Tech. Rep. MSR-TR-2008-53, Microsoft Research, 1 Apr.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics 27*, 5 (Dec.), 126:1–126:11.

ZHOU, K., REN, Z., LIN, S., BAO, H., GUO, B., AND SHUM, H.-Y. 2008. Real-time smoke rendering using compensated ray marching. *ACM Transactions on Graphics 27*, 3 (Aug.), 36:1–36:12.