Real-Time Parallel Hashing on the GPU

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

UC DAVIS UNIVERSITY OF CALIFORNIA

* HARVARD School of Engineering and Applied Sciences

To appear in SIGGRAPH Asia 2009

Problem

Goal: Parallel-friendly data structure allowing efficient random access to millions of items and buildable at interactive rates.

On the CPU, hash tables are generally used. Collisions are handled using techniques like *chaining,* where items hashing to the same location are put in a linked list.

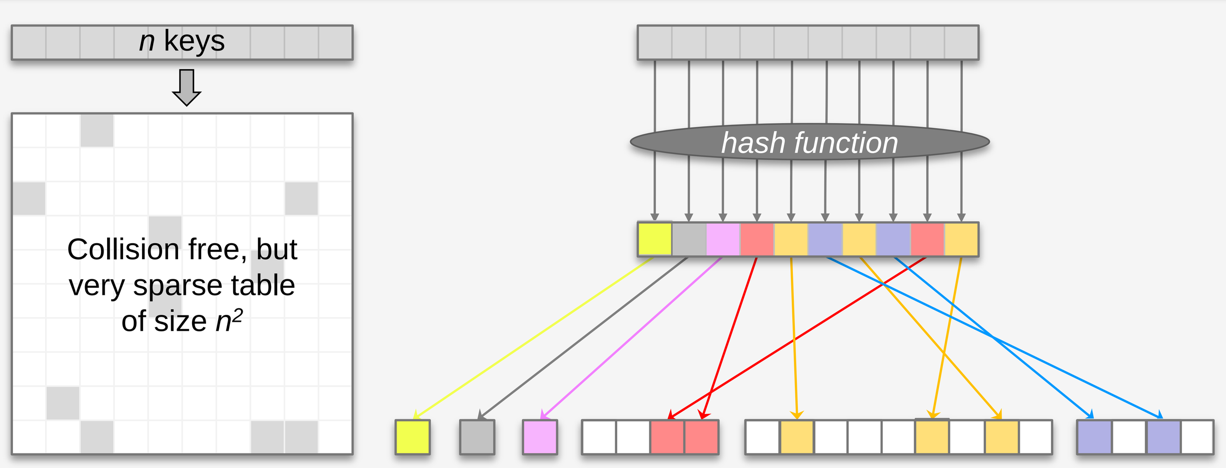
Hash techniques don't parallelize well for three reasons:

- *Synchronization*: Insertions usually involve sequential operations
- *Variable work per access*: Number of probes varies per query, forcing threads to wait for worst-case number of probles
- *Sparse storage*: Little locality exhibited in either construction or access, so caching and computational hierarchies have little ability to improve performance.

Can be addressed with *perfect hash tables*, where each element can be accessed in *O(1)*, but previously required CPU construction.

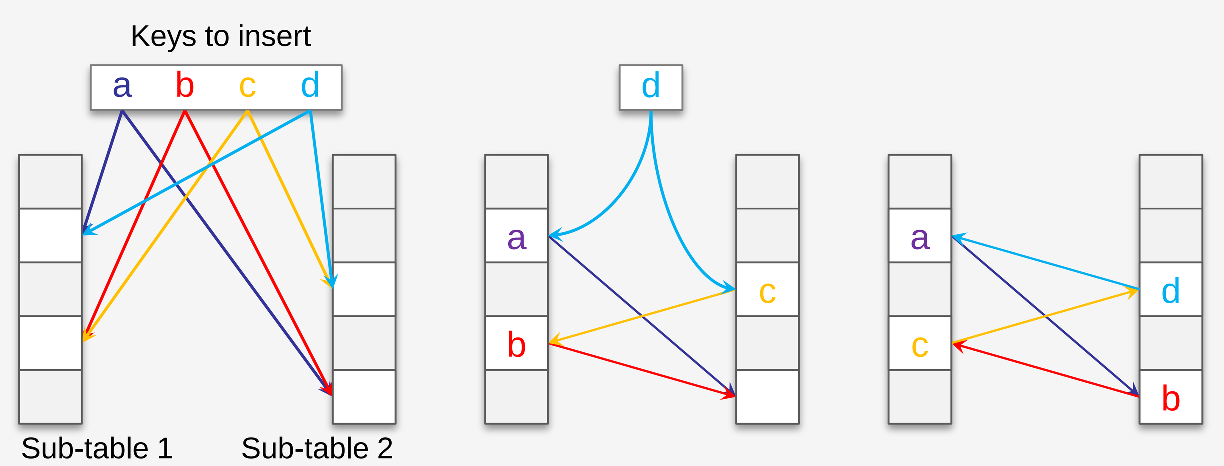
Get efficient GPU construction by combining two strategies: classical FKS perfect hashing and relatively new **cuckoo hashing**.

Background



FKS perfect hashing

- For n keys and a table with n^2 slots, randomly chosen hash functions will be likely be collision-free (left)
- Can shrink to *O(n)* slots by first hashing the items into small buckets, each needing much less space (right)
- Requires many small and empty buckets, still using over $3n$ space



Cuckoo hashing

- Items can hash into one of multiple locations (left)
- Items inserted into any empty slots if possible (middle)
- If no slots available, item is evicted to make room, forcing recursive insertion & evictions until convergence (right)
- Get better space usage with more sub-tables (up to 90% with three)

Our approach

FKS and cuckoo hashing are problematic for GPU:

- FKS hashing is fast, but requires too much space
- Cuckoo hashing can be space-efficient, but requires slow global memory and global synchronization for parallel insertion

Get efficient algorithm by combining both:

- Like with FKS hashing, first partition into small buckets of at most 512 items
- Build parallel cuckoo hash on smaller buckets in fast on-chip memory
- Retrieval takes at most 3 probes: one for each cuckoo sub-table of the bucket item hashes into

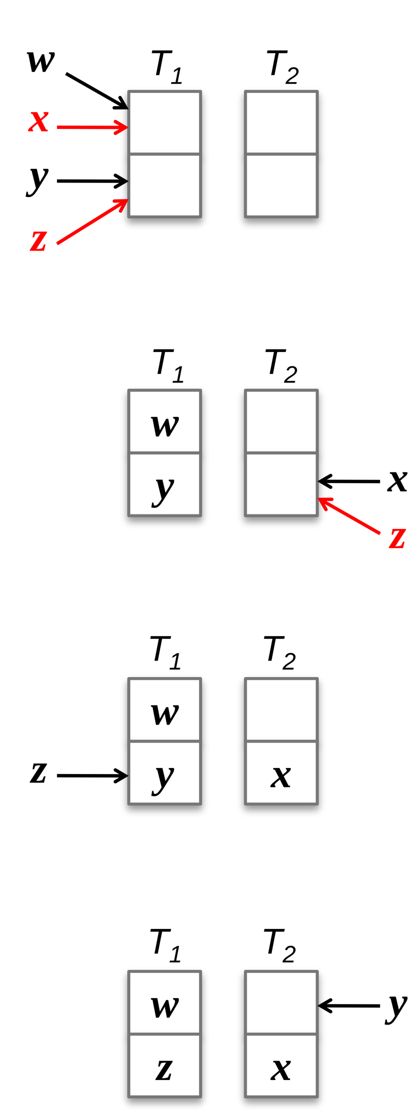
Algorithm can be generalized to handle multiple values per key, or to generate two-way index between keys and unique IDs.

We parallelized cuckoo hashing insertion for the GPU. **Parallel cuckoo hashing**:

- Inserts all items simultaneously, iterating through sub-tables in round-robin fashion
- Assumes that exactly one write will succeed for colliding items
- Typically completes in *O(lg n)* iterations for two sub-tables

	H_1	H_2
w	0	0
x	0	1
y	1	0
z	1	1

Hash functions H_1 and H_2 map each item (w, x, y, z) to two places in the sub-tables T_1 and T_2

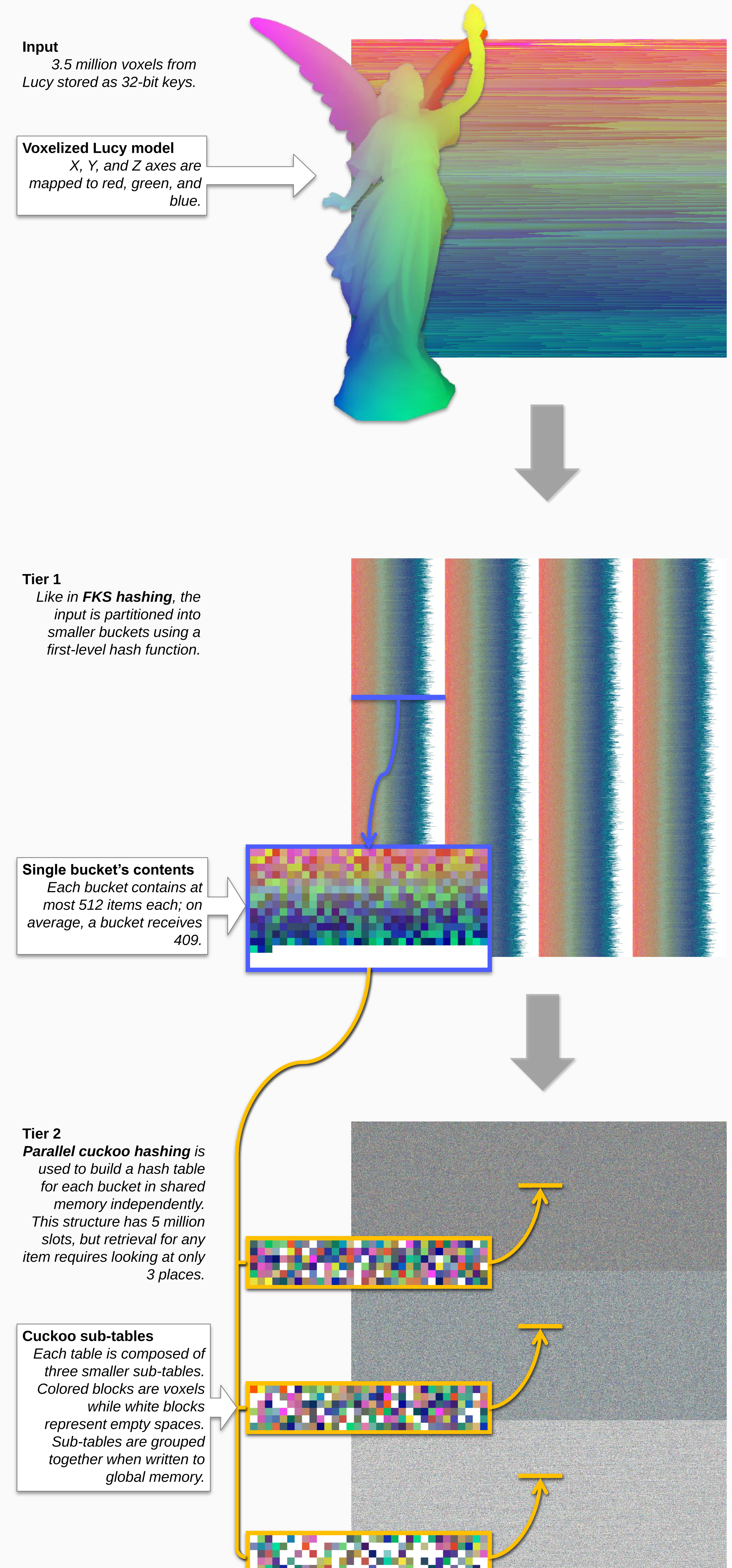


Threads simultaneously insert. Black arrows show successful insertions; x and z failed

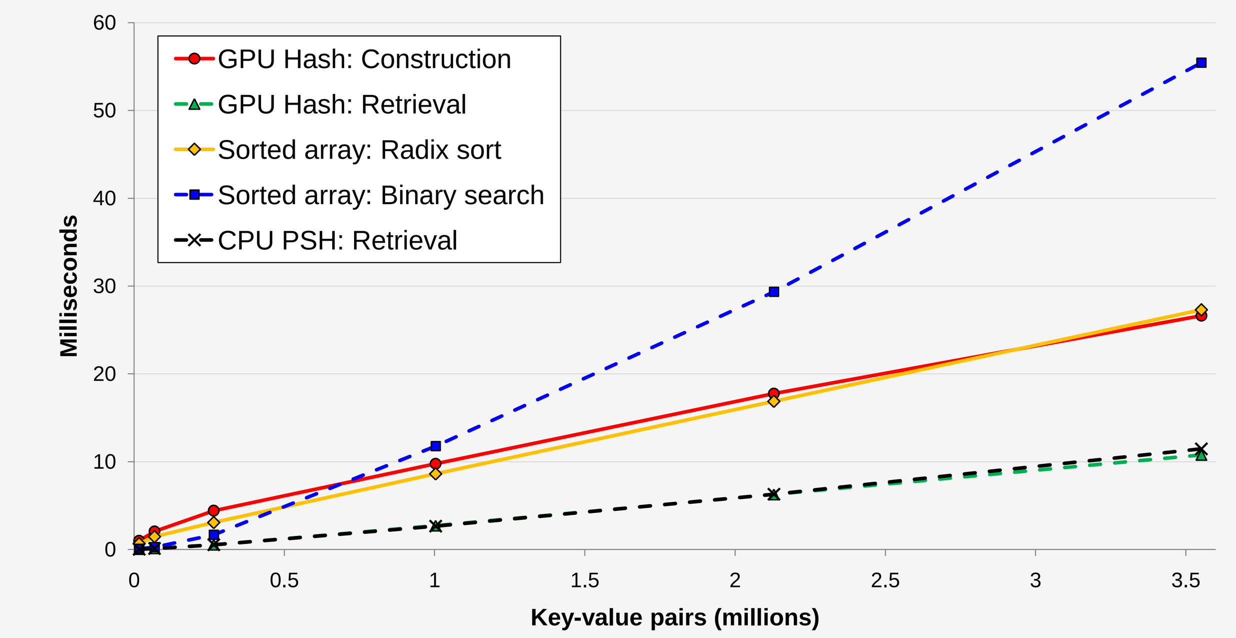
x and z failed to write to T_1 and try T_2 where they collide: x is written and z fails

z goes back to T_1 and evicts y

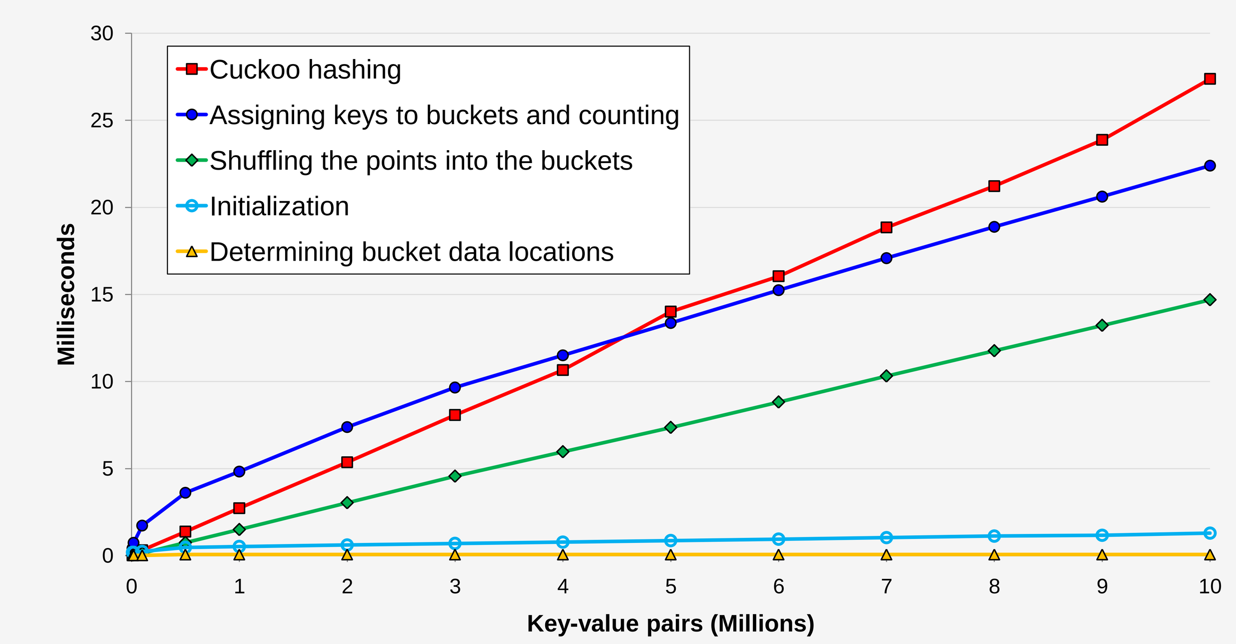
y writes itself into T_2



Results



Timings for increasingly finer voxelizations of Lucy. We compare against a sorted list, using binary searches for retrieval. Construction takes roughly the same amount of time as the radix sort, while our retrievals are consistently faster than binary searches for random access.



Timing for each stage of our algorithm for increasingly large input of random keys. We see roughly linear scaling for each stage.

In our paper, we demonstrate hash table use with two different GPU applications. G*eometric hashing* finds the image in the upper left in the image in the upper right. *Spatial hashing* finds the boolean intersection between two moving point clouds while allowing the user to interactively change the transformation.

References

- **Project page**: http://idav.ucdavis.edu/~dfalcant/
- 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), 583-544.
- Pagh, R., and Rodler F. F. 2001. **Cuckoo hashing**. In *9th Annual European Symposium of Algorithms*, Springer, vol. 2161 of *Lecture Notes in Computer Science*, 121-133.