# Parallel Computing: Project Report - Real-Time Hashing on GPU Architecture with CUDA

Ruoyu Wang
Computer Science and Technology
ShanghaiTech University
wangry@shanghaitech.edu.cn
30401619

Guanzhou Hu
Computer Science and Technology
ShanghaiTech University
hugzh1@shanghaitech.edu.cn
36136477

***Abstract*—Hashing is a frequently used technology in the field of computer graphics. Applications like interactive frame intersection detection and photo pattern matching require real-time hashing on millions of voxels. Efficient hash tables are usually considered not suitable for GPU architecture. Dan A. Alcantara, et al., proposed a multi-level shared-memory solution that achieves real-time graphic hashing on GPUs [1]. In this report, we summarize their solution and introduce our simplified implementation for numerical hashing. Multi-level shared-memory hashing achieves 10x speedup compared to traditional global-memory GPU hashing at insertions, meanwhile lookups are not affected.**

## I. Introduction

Computer graphics applications frequently use hashing techniques. In frame intersection detection (Figure 1, top), voxels in the based frame are inserted into a hash table. We then query the table for voxels in subsequent frames, to check whether these two frames intersect at those positions. In photo pattern matching (Figure 1, bottom), similar procedures are conducted. With real-time embedded systems become popular, these applications are made interactive, which requires hashing to be real-time as well. Dan A. Alcantara, et al., proposed a multi-level shared memory hashing scheme on GPU architecture that can hash millions of voxels in milliseconds time [1].

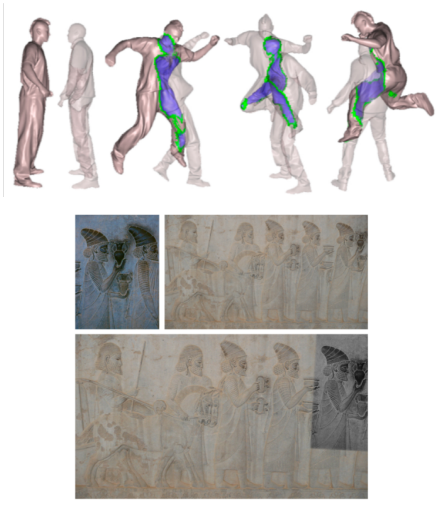

Fig. 1. Examples From the SIGGRAPH'09 Paper [1]: Frame Intersection Detection (top), Photo Pattern Matching (bottom)

Dan's solution contains redundant stages in order to serve graphics applications. We modified their algorithm and implemented a simplified version for purely numerical hashing. The following sections will cover:

- Summary of the aforementioned paper "Real-Time Parallel Hashing on the GPU" (see Section II).
- Our simplified implementation [2] (see Section III).
- Experiments and results (see Section IV).

Then Section V will conclude.

## II. Paper Reading

We will summarize the main ideas of SIGGRAPH'09 paper "Real-Time Parallel Hashing on the GPU" [1] in this section.

### A. Motivation

Their solution is motivated by two important hashing techniques: *FKS perfect hashing* and *cuckoo hashing*.

Traditional perfect hashing states that mapping $n$ input keys onto a table of size $n^2$ is likely to introduce no collisions. FKS perfect hashing is a space-optimized version of perfect hashing by using 2 levels. On the first level, keys are hashed into buckets with a hash function chosen to be uniform enough. On the second level, keys in each bucket are then hashed using traditional perfect hashing locally (therefore a bucket with $n'$ keys inside will be of size $n'^2$).

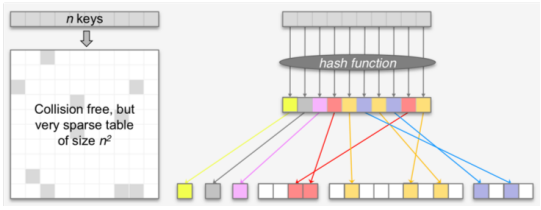


Fig. 2. From the SIGGRAPH'09 Poster [3]: FKS Perfect Hashing

Choosing collision-free perfect hash functions is not practical in real-world applications. Cuckoo hashing is a modified version of perfect hashing that makes it practical. It allows a key $k_1$ to be hashed into multiple slots with multiple hash functions, and when all slots are occupied, evicts the key in its first slot $k_2$ to the next slot on $k_2$'s chain. Lookup operations

are ensured constant-time, while insertion operations depend on the choice of hash functions and the input scale.

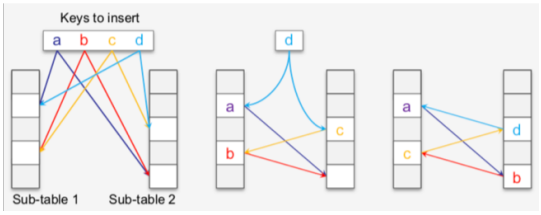


Fig. 3. From the SIGGRAPH'09 Poster [3]: Cuckoo Hashing

Cuckoo hashing can be parallelized under GPU computation model, where each thread tries to insert a unique key, and each warp stage executes a round of eviction. However, in one stage, all threads communicate with the global device memory in relatively random (thus non-coalesced) manner. Memory accesses become a serious bottleneck of GPU cuckoo hashing. Multi-level hashing can potentially help with this situation.

### B. Algorithm Design

Their solution contains two major phases:

1) *Tier-1*: Distribute $n$ keys into $b$ buckets, trying to be as uniform as possible.
2) *Tier-2*: Inside each bucket, do local cuckoo hashing using shared device memory.

The overall insertion algorithm is summarized in Algorithm 1. It launches 4 kernels for every insertion operation. For simplicity, we omit satellite values and only consider the keys.

**Algorithm 1** Insert($T, K$)

**input:** $T$, Hash table; $K$, Input keys
**output:** Inserts all keys in $K$ into table $T$

```
    for every key k in parallel do
        c = bucket[k] = h1(k);
3:      atomicAdd(count[c]);
        offset[k] = old value of count;
    end for
6:  parallel prefix sum on count[] to get start[];
    for every key k in parallel do
        c = bucket[k];
9:      buf[start[c] + offset[k]] = k;
    end for
    for every bucket c in parallel do
12:     retrieve its keys from buf[];
        cuckoo_hash(start[c], count[c]);
        write back to global memory;
15: end for
```

In graphics applications, input values are usually uniform, therefore level-1 hash function $h_1$ can be chosen as a simple modulo function. Experiments show that the following hash function gives a more balanced distribution across buckets:

$$h_1(k) = ((c_0 + c_1 k) \bmod 1900813) \bmod |\text{buckets}|,$$

where $c_0$ and $c_1$ are randomly chosen integers, and 1900813 is a prime number which gives the best distribution in their experiments.

Their work also includes optimization techniques for multiple satellite values (for example RGB values for a voxel). This is not the most important part of our concern, therefore we will not discuss them in this report.

### C. Results

On Amazon AWS GPU servers, they applied multi-level shared-memory GPU hashing for the 3D Lucy model voxels. Performance results are shown in Figure 4 below. "GPU Hash: Construction" means insertion and "GPU Hash: Retrieval" means lookup. "Radix sort" and "Binary search" correspond to the time performance of not using hash tables to lookup all values after they have been sorted.

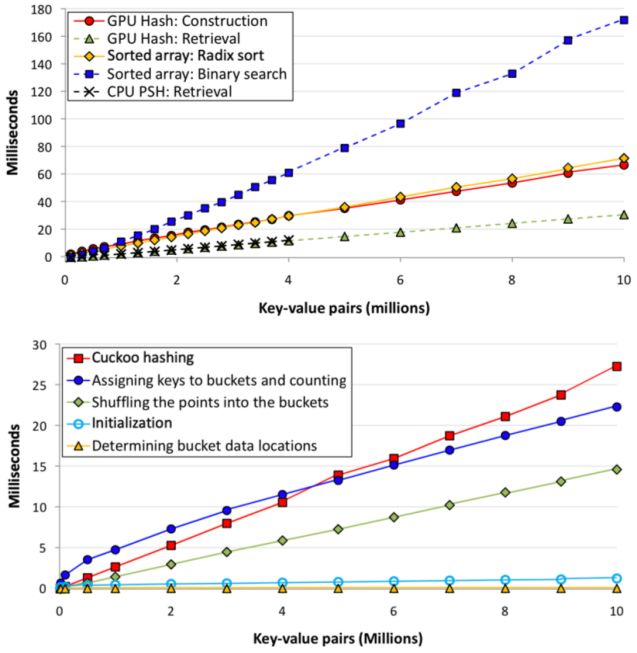


Fig. 4. Results From the SIGGRAPH'09 Paper [1]: Overall Time Performance Comparison (top), Insertion Time Breakdown (bottom)

## III. Our Implementation

We found that for purely numerical and uniform input keys, their solution can be simplified into 2 kernels per insertion, by sacrificing slightly more global memory space. When partitioning a table of size $m$ into $b$ buckets and inserting $n$ keys into the table, we assume that every bucket will not receive more than $\frac{m}{b}$ keys. Thus, instead of using a buffer array of size $n$ to store distribution results, we use a table buffer array of size $m$, and write every key $k$ directly into its bucket during distribution phase. Then, we assign a CUDA thread block for each bucket. It retrieves keys in the bucket from the table buffer, and conducts local cuckoo hashing.

The overall insertion procedure of our simplified algorithm is summarized in Algorithm 2. Bucket size is normally chosen

as 256 / 512, since they are the most reasonable sizes for a CUDA thread block [2].

**Algorithm 2** Insert($T, K$)

**input:** $T$, Hash table; $K$, Input keys
**output:** Inserts all keys in $K$ into table $T$

```
    choose bucket size = m/b = 512;
    for every key k in parallel do
3:      c = bucket[k] = h_1(k);
        atomicAdd(count[c]);
        offset[k] = old value of count;
6:      tab_buf[512 · c + offset[k]] = k;
    end for
    for every bucket c in parallel do
9:      retrieve its keys from tab_buf[];
        cuckoo_hash(512 · c, count[c]);
        write back to global memory;
12: end for
```

Notice that in the second kernel, data retrieval from global memory and write back towards global memory are all memory coalesced.

## IV. Performance Evaluation

We conducted experiments on insertion and lookup time performance of multi-level shared-memory GPU hashing implementation on SIST-AI cluster, node 14. Its hardware configuration is listed in Table I. Our experiments only include numerical keys without satellite values, therefore they can clearly demonstrate the speedup brought by utilizing shared memory.

| | |
|---|---|
| **GPU** | Nvidia Tesla K80, 12 GB memory |
| **CPU** | Intel Xeon E5-2690 v4 @ 2.60GHz, 28 threads |
| **Main Memory** | 128 GB, DDR4 |

TABLE I
Experiment Hardware Configuration

### A. Insertion Performance

We insert $2^s$ uniformly random integer keys into a table of size $2^{s+1}$. Compared to traditional global-memory cuckoo hashing, multi-level shared-memory implementation achieves nearly 10x speedup, as shown in Figure 5.

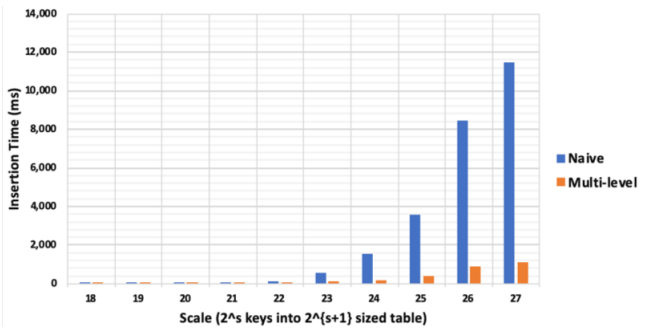


Fig. 5. Insertion Time Performance Comparison

### B. Lookup Performance

We lookup $2^s$ unique integer keys from a table of size $2^{s+1}$ (initially containing $2^s$ keys inside), where half of the keys to lookup are picked from existing keys, and another half purely random. Results are shown in Figure 6. We can see that though we added an extra level of hashing into buckets, its effect towards lookup performance is too small to be observed.

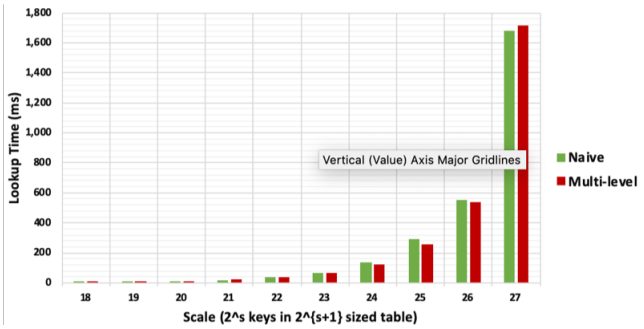


Fig. 6. Lookup Time Performance Comparison

## V. Conclusion & Further Optimizations

In summary, combining multi-level hashing with shared-memory cuckoo hashing significantly accelerates GPU hashing performance. Dan A. Alcantara, et al., brought this idea into scope in their work "Real-Time Parallel Hashing on the GPU" [1]. They achieved graphical hashing on millions of voxels in milliseconds time. We simplified their algorithm to adapt for purely numerical hashing. Our implementation achieves nearly 10x speedup for insertion operations, meanwhile not affecting lookup performance. With multi-level shared-memory GPU hashing, it is possible to process hundreds of millions of input queries within real-time constraint.

Interestingly, we failed to find any further work related to parallel real-time GPU hashing within the recent 10 years. We suppose that multi-level shared-memory hashing algorithm has already pushed the performance of GPU hashing to an extreme. Further improvements might be brought by a better choice of hash function set, based on the actual application scenery.

## References


[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the gpu," in *ACM SIGGRAPH Asia 2009 Papers*, ser. SIGGRAPH Asia '09. New York, NY, USA: ACM, 2009, pp. 154:1–154:9. [Online]. Available: http://doi.acm.org/10.1145/1661412.1618500

[2] Our implementation on github. [Online]. Available: https://github.com/hgz12345ssdlh/cuckoo-hashing-CUDA

[3] Real-time parallel hashing on the gpu. [Online]. Available: https://www.cs.bgu.ac.il/~asharf/Projects/RealTimeParallelHashingontheGPU.pdf