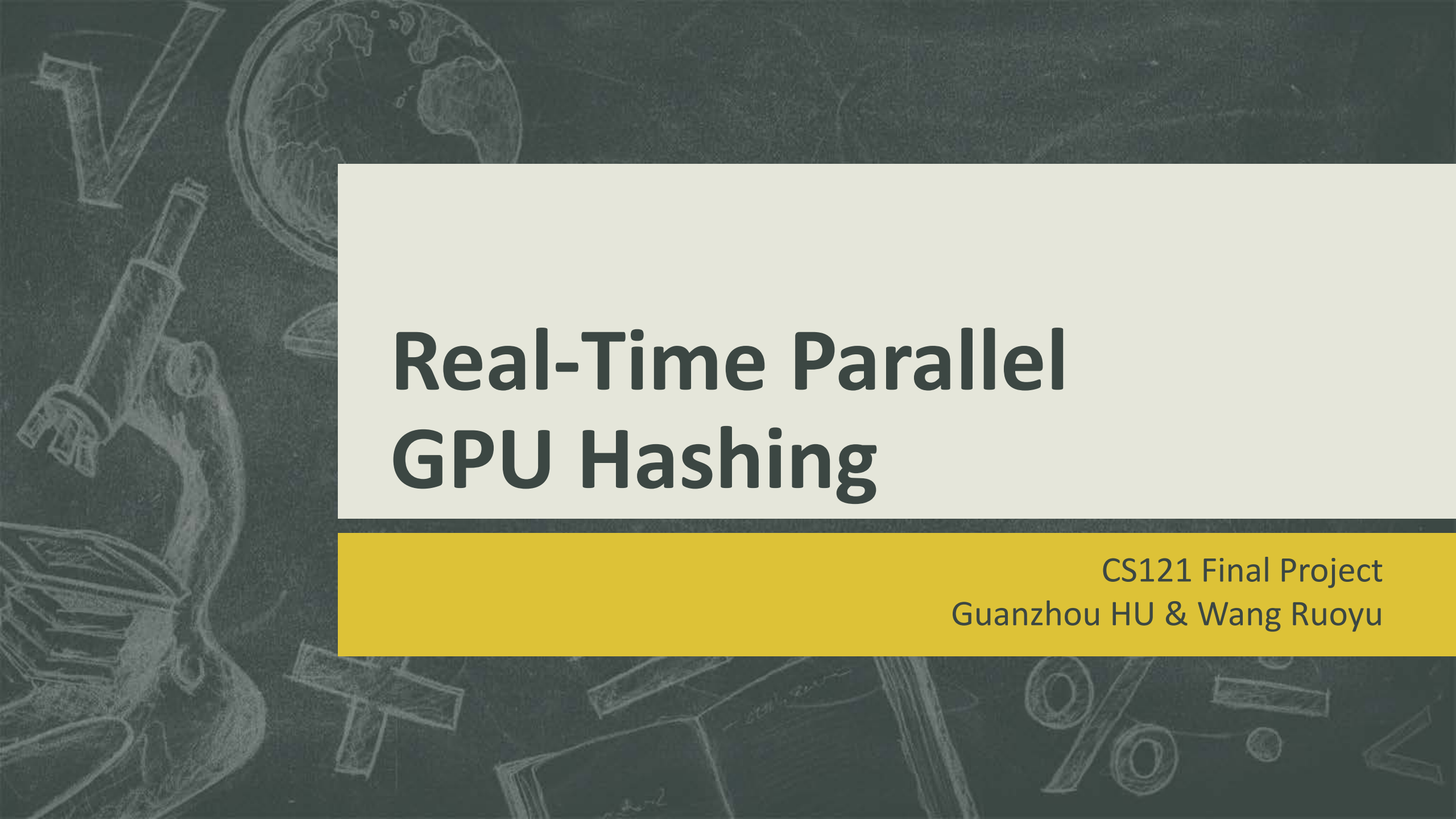


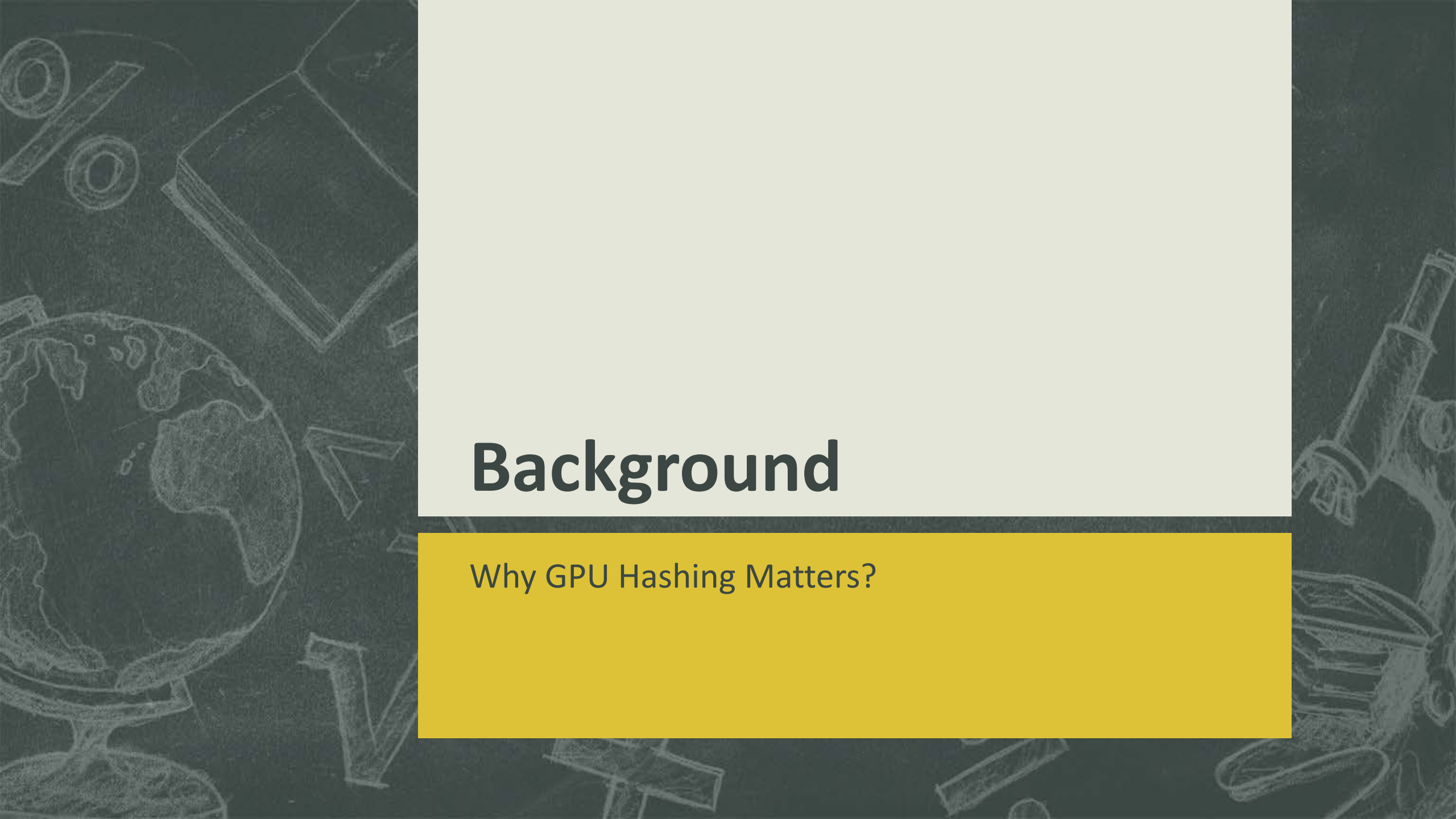
Real-Time Parallel GPU Hashing

CS121 Final Project

Guanzhou HU & Wang Ruoyu

Outline

1. Background
2. Paper Reading: “Real-Time Parallel Hashing on the GPU”, SIGGRAPH’09
   - Motivation
   - Design of the 2-level Hashing Scheme
   - Experiments & Results
3. Our Implementation: A Simplified Version
4. Command-Line Demo

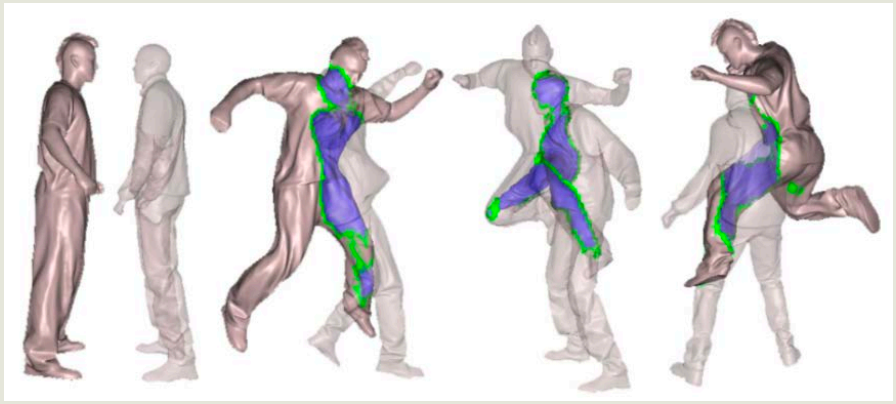


Background

Why GPU Hashing Matters?

Background

- In computer graphics, many applications need to store a **sparse data set** into a **dense representation**, and requires **super-fast lookups**.

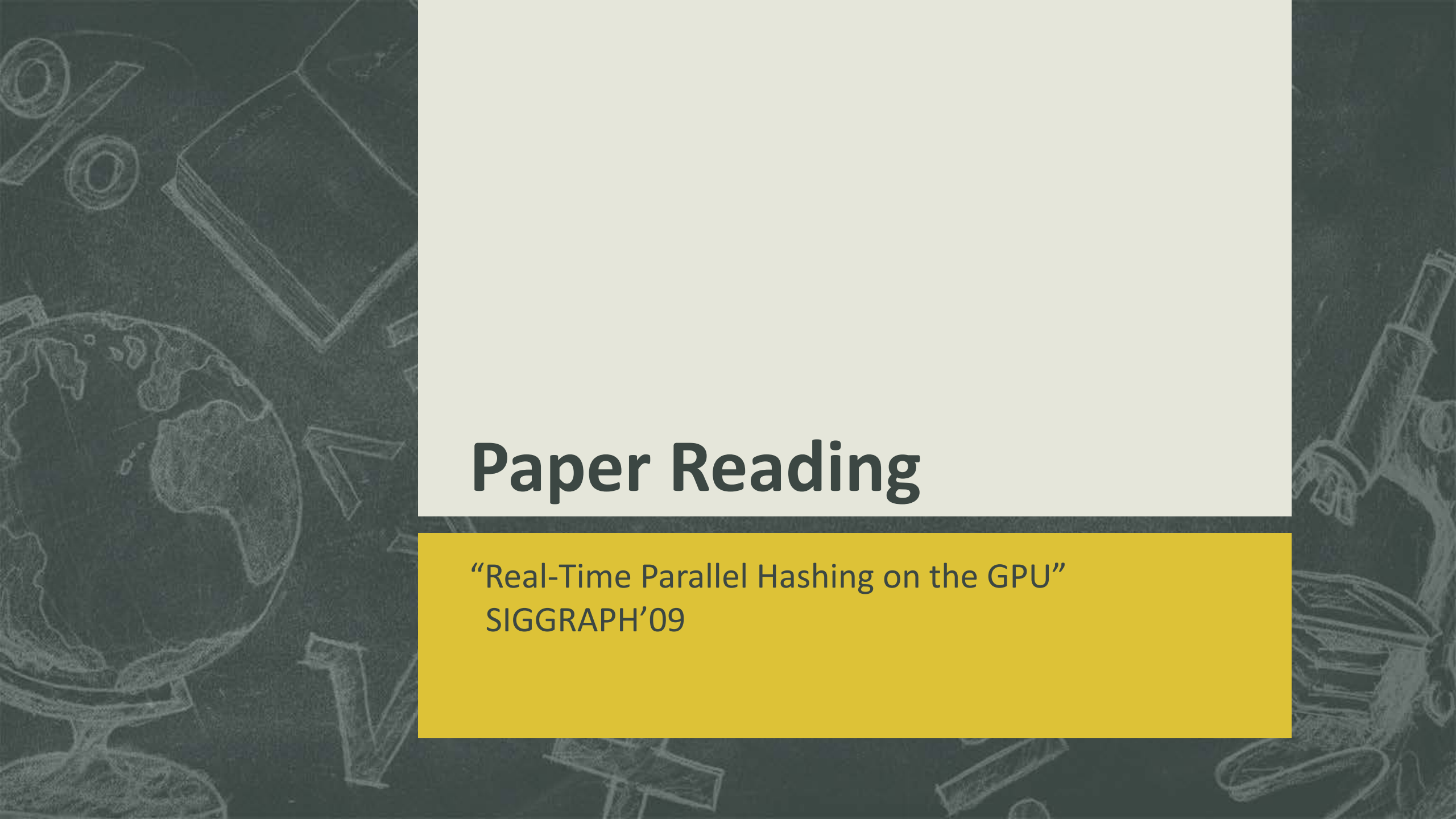


Frame Surface Intersection Detection

Photo Segments Alignment

Background

- Solution: **Hash Tables**
- On GPUs, efficient hash tables are hard to implement:
  - Synchronization
    - Closed addressing: synchronization on *linked lists*
    - Open addressing: synchronization on the *chaining* process
  - Big table size requires global memory
  - …



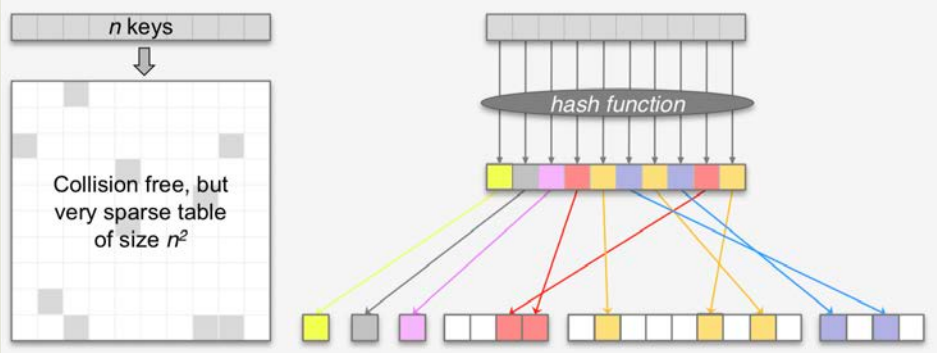
Paper Reading

"Real-Time Parallel Hashing on the GPU"
SIGGRAPH'09

Motivation

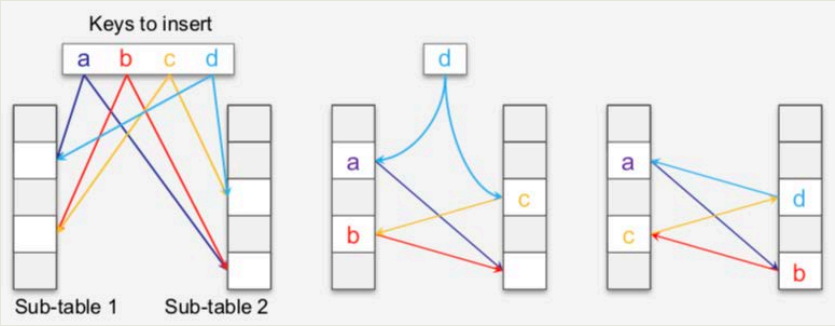
- FKS Perfect Hashing

Key idea: **Multi-level**



- Cuckoo Hashing

Key idea: **Eviction chain**



Algorithm: 2-Level Hashing

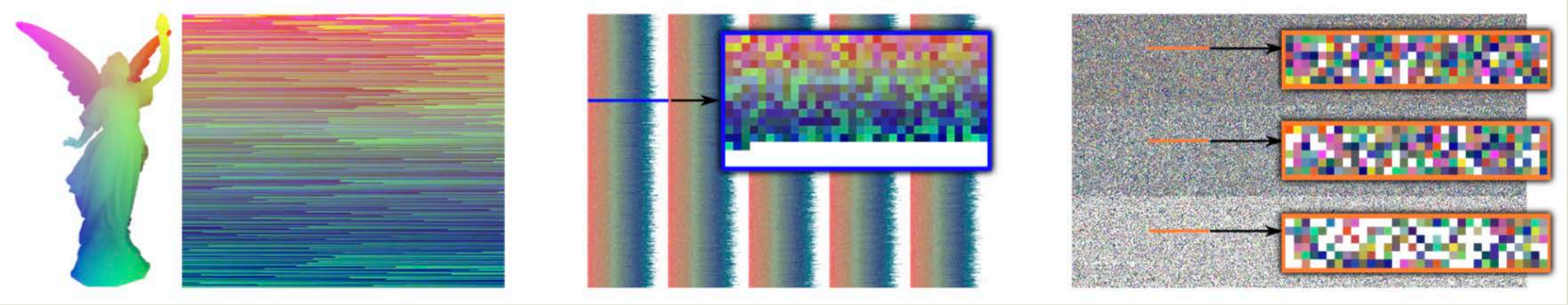
- Combine **multi-level hashing** with **cuckoo evictions**
- Use 2 hash phases:

Choose bucket size = *512*, suppose inserting *n* keys into a table of size *512 * b*

Distribute *n* keys into *b* **buckets**

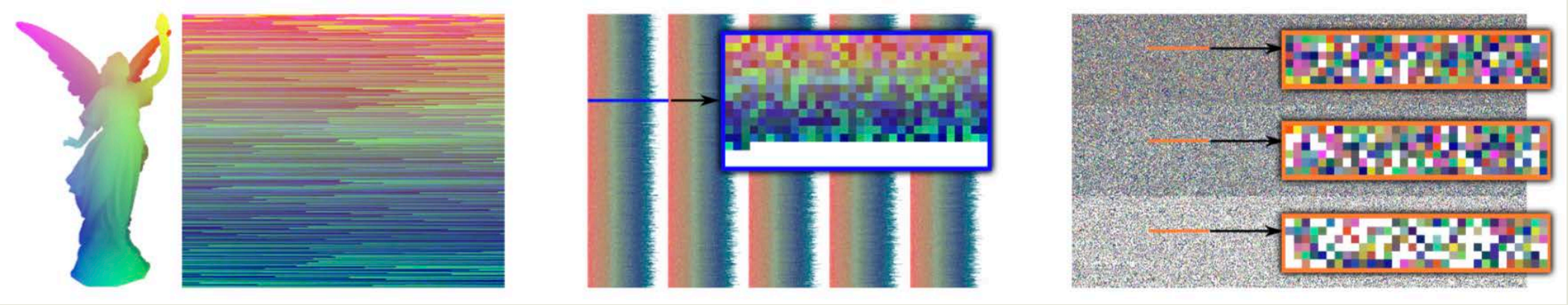
Do **local** cuckoo hashing on each bucket

Algorithm: Procedure



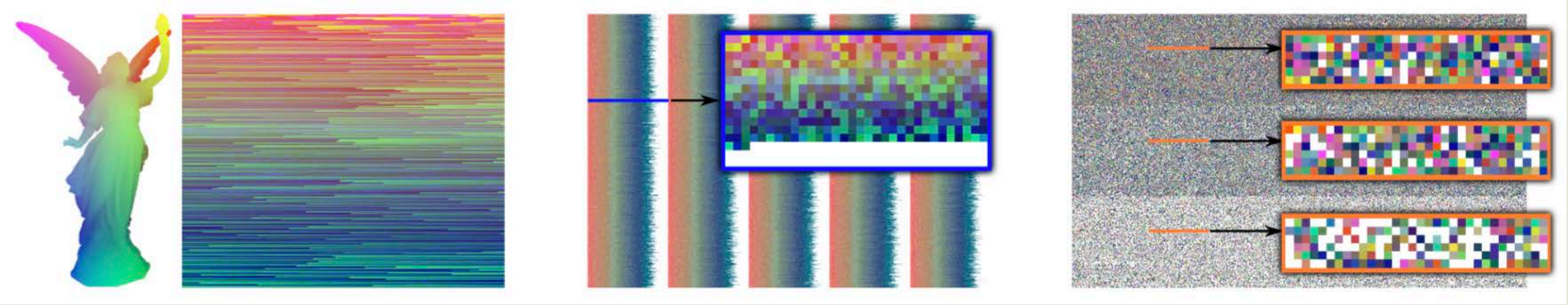
```
PARALLEL FOR each key k:
        bucket_number[k] = h1(k);
        bucket_offset[k] = atomicAdd(count[bucket_number[k]]);
```

Algorithm: Procedure



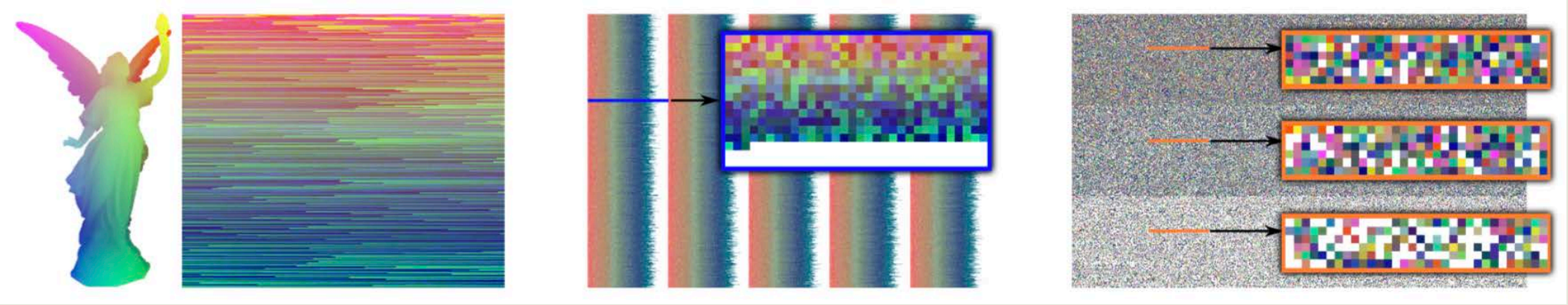
```
PARALLEL FOR each key k:
        bucket_number[k] = h1(k);
        bucket_offset[k] = atomicAdd(count[bucket_number[k]]);
PARALLEL prefix sum on count[];  // Gets start[];
```

Algorithm: Procedure



```
PARALLEL FOR each key k:
        bucket_number[k] = h1(k);
        bucket_offset[k] = atomicAdd(count[bucket_number[k]]);
PARALLEL prefix sum on count[];  // Gets start[];
PARALLEL FOR each key k:
        store k into buffer[start[bucket_number[k]] + offset[k]];
```

Algorithm: Procedure



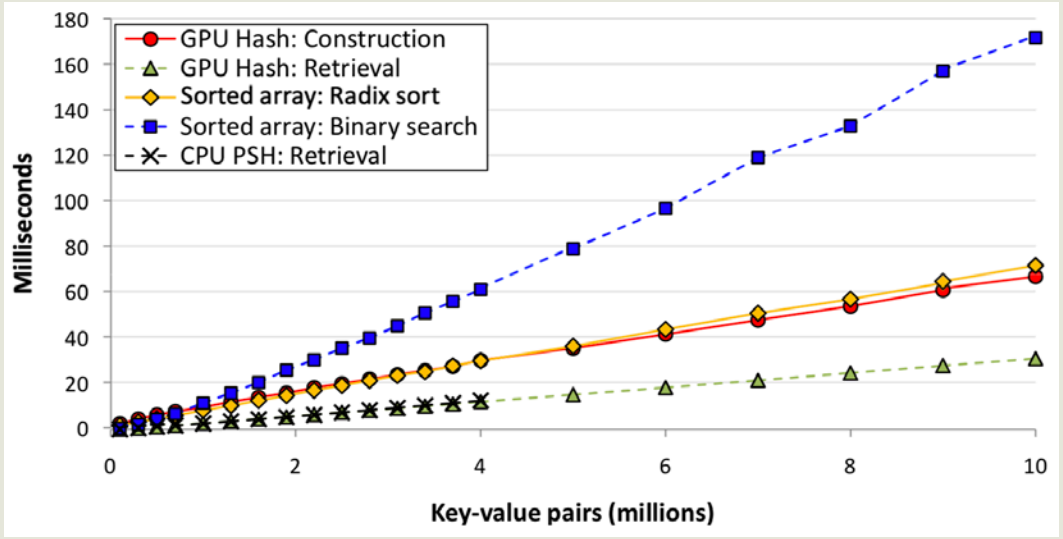
4 Kernels

```
PARALLEL FOR each key k:
        bucket_number[k] = h1(k);
        bucket_offset[k] = atomicAdd(count[bucket_number[k]]);
PARALLEL prefix sum on count[];  // Gets start[];
PARALLEL FOR each key k:
        store k into buffer[start[bucket_number[k]] + offset[k]];
PARALLEL FOR each bucket b:
        do local cuckoo hash inside shared memory;
        write back results to global table;
```

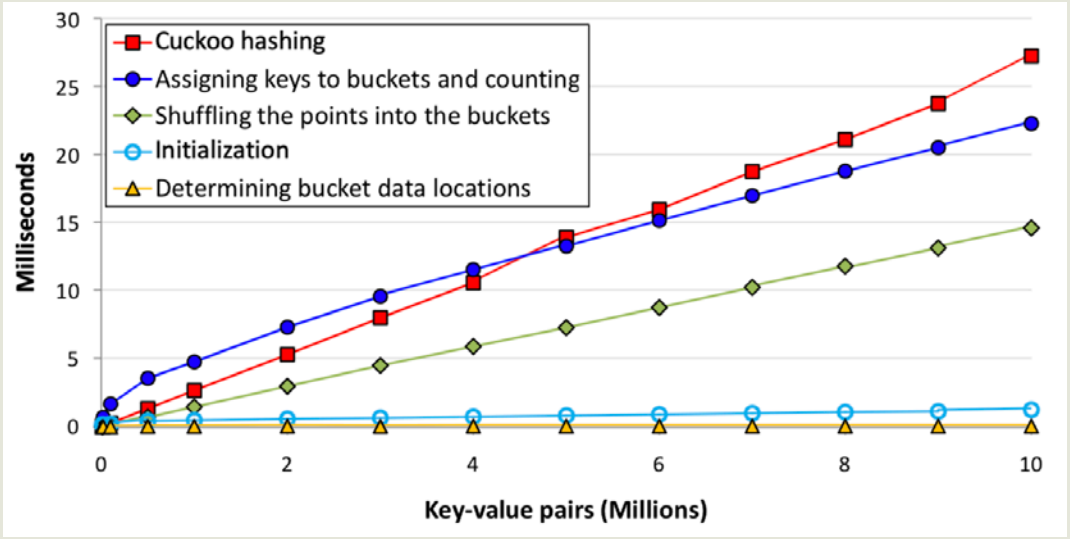
Algorithm: Details

- Choice of $h_1(k)$: $k \rightarrow$ bucket number
  1. Naive: $h_1(k) = k \bmod |\text{buckets}|$
  2. Better: $h_1(k) = \big((c_0 + c_1 k) \bmod 1900813\big) \bmod |\text{buckets}|$
- In the 4th phase, assign 1 thread block / bucket
- Hashing with (multiple) satellite values…
- Key compaction…

Results



Overall Time Performance



Construction Time **Breakdown**

Our Implementation

A Simplified Version to Verify This Paper

Prerequisites

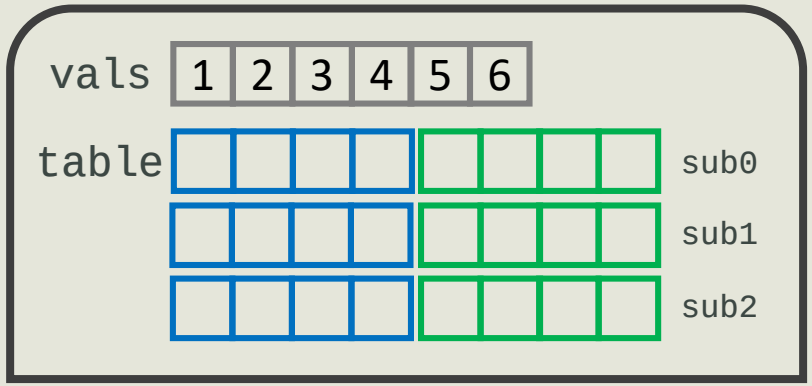
- We assume:
  - Input values are uniformly random
  - Only inserting into an empty table
  - No satellite values
  - Not considering key compression

```
J jose # ~/Dropbox/信息-CSMajor/并行计算-CS121/Project/src on git:master x
$ cloc .
       7 text files.
       7 unique files.
       1 file ignored.

github.com/AlDanial/cloc v 1.80  T=0.02 s (356.1 files/s, 60131.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
CUDA                             5            167            229            711
C/C++ Header                     1             17              6             34
make                             1              5              0             13
-------------------------------------------------------------------------------
SUM:                             7            189            235            758
-------------------------------------------------------------------------------
```

Insertion Procedure

Scenery: Inserting 6 keys into a table of size 8; bucket size = 4; 3 cuckoo sub-tables



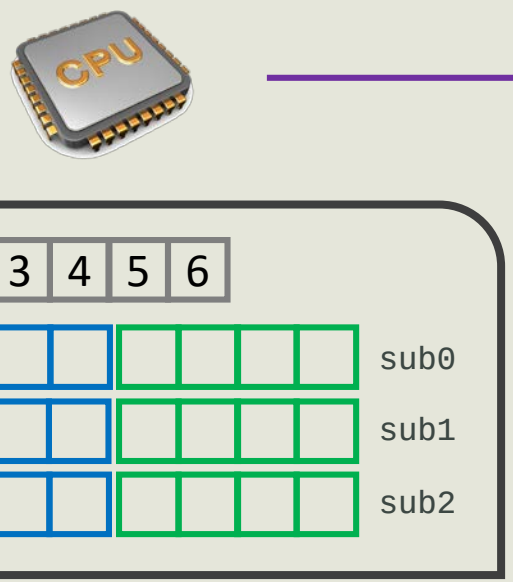
Main Memory

GPU Global Memory

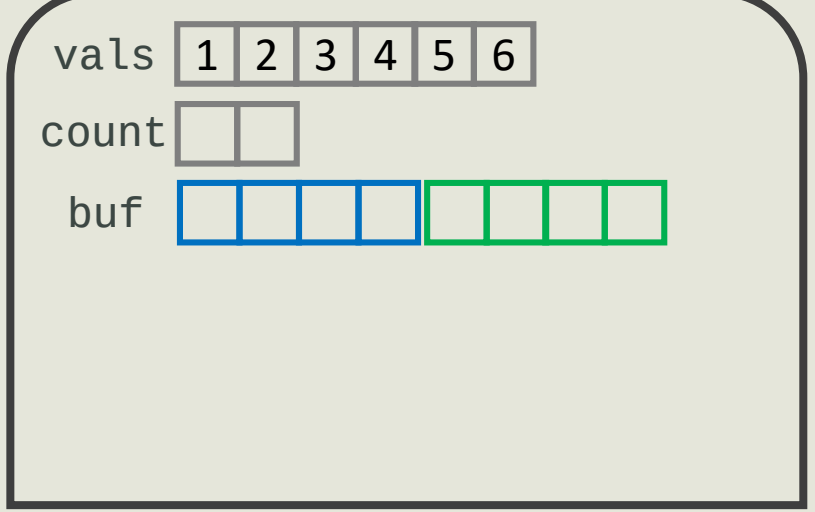
Block #1
Shared Memory

Block #2
Shared Memory

Insertion Procedure



Main Memory



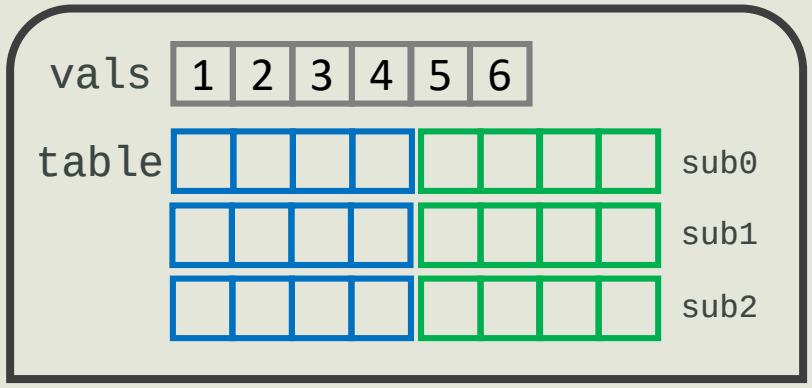
GPU Global Memory

Block #1
Shared Memory

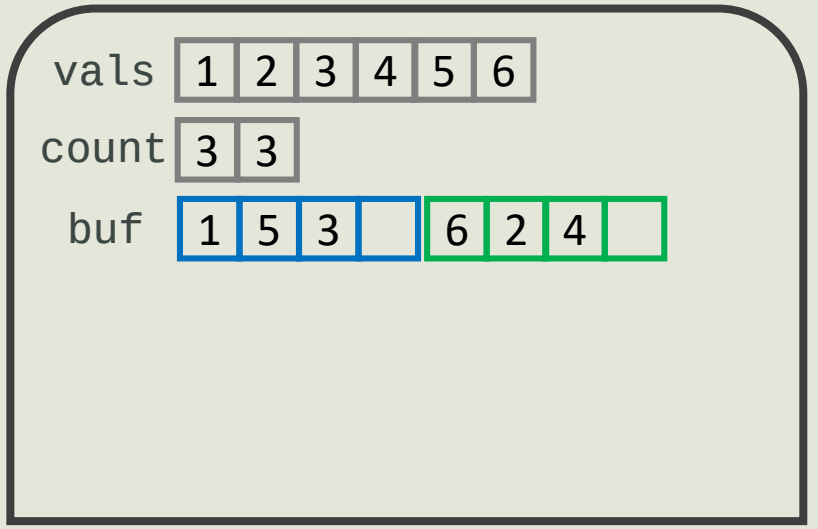
Block #2
Shared Memory

Insertion Procedure

Invoke
`BucketKernel<<<6>>>`



Main Memory

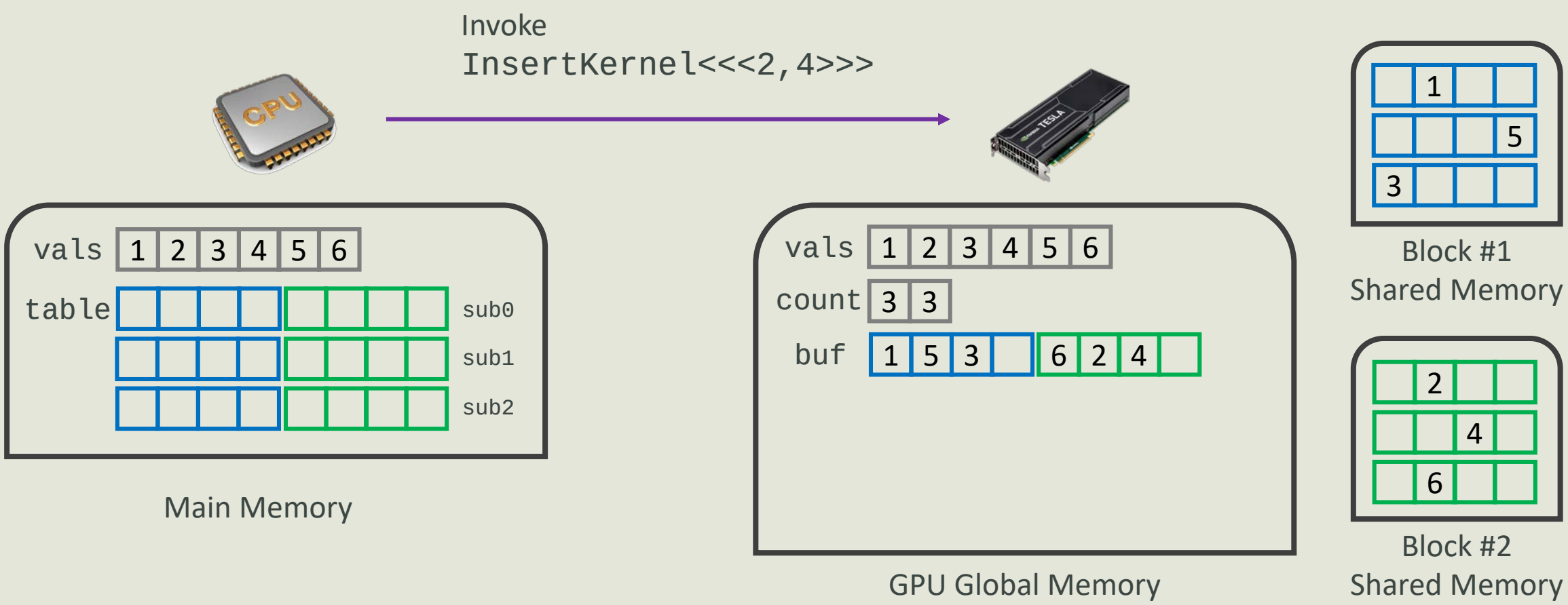


GPU Global Memory

Block #1
Shared Memory

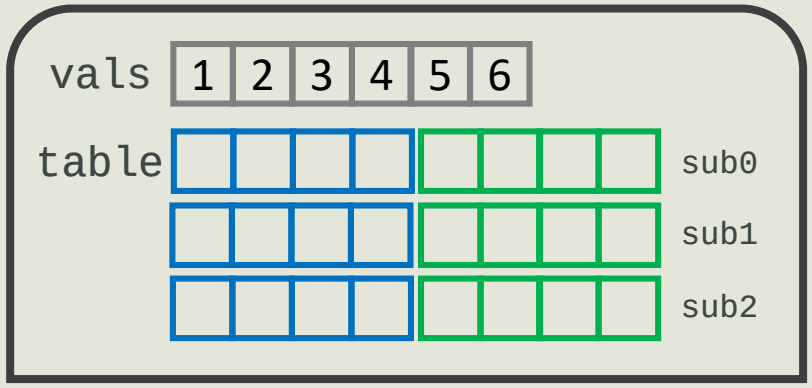
Block #2
Shared Memory

Insertion Procedure

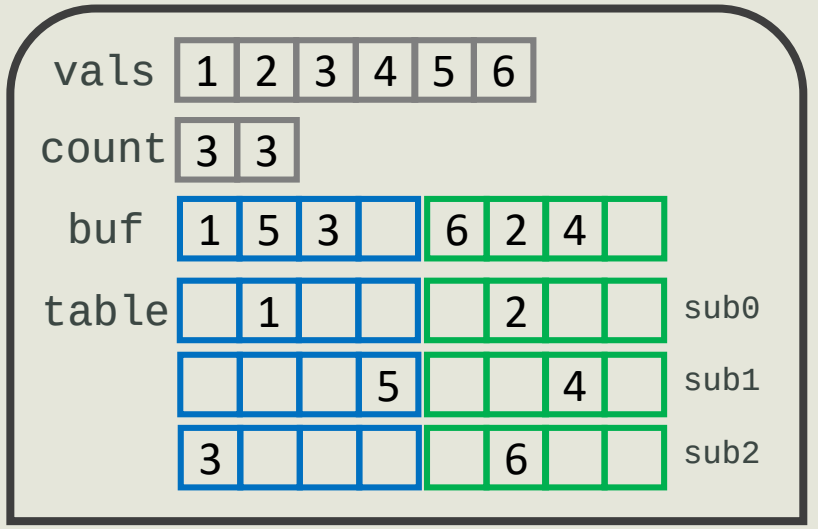


Insertion Procedure

Invoke
`InsertKernel<<<2,4>>>`



Main Memory

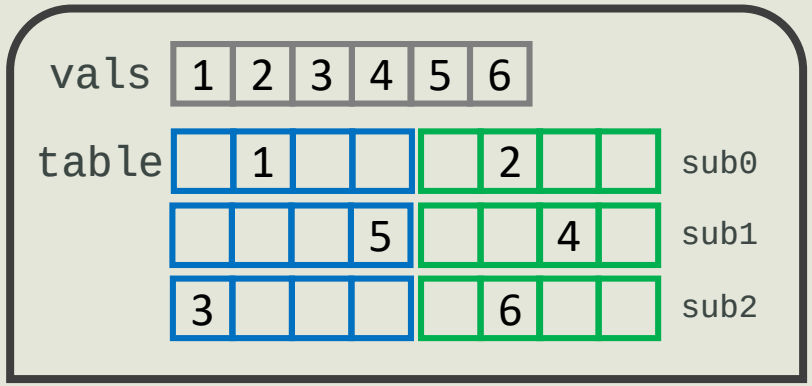


GPU Global Memory

Block #1
Shared Memory

Block #2
Shared Memory

Insertion Procedure



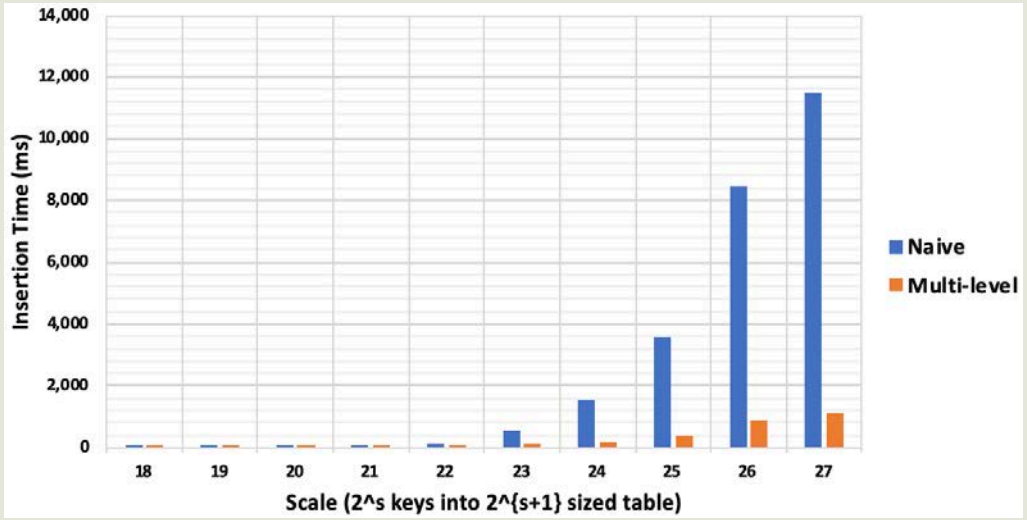
Main Memory

GPU Global Memory

Block #1
Shared Memory

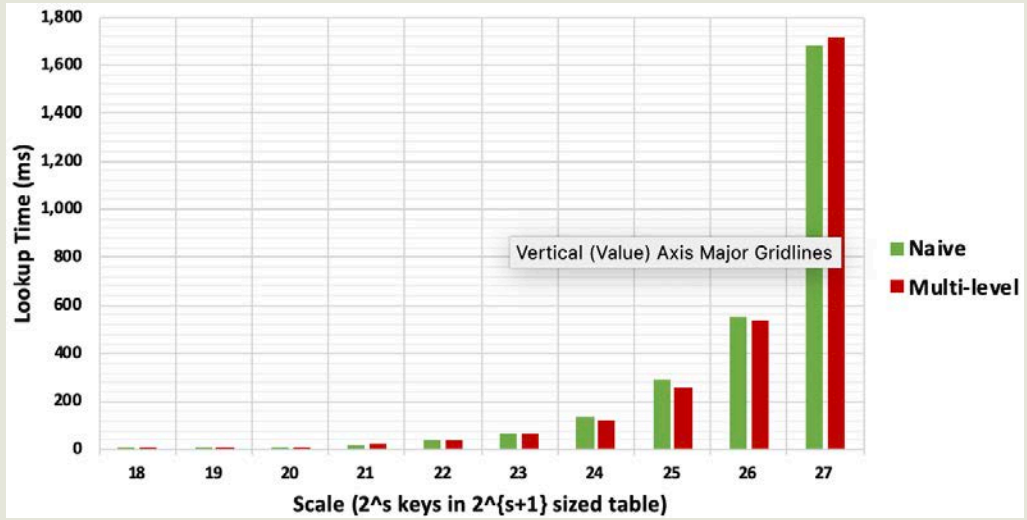
Block #2
Shared Memory

Results



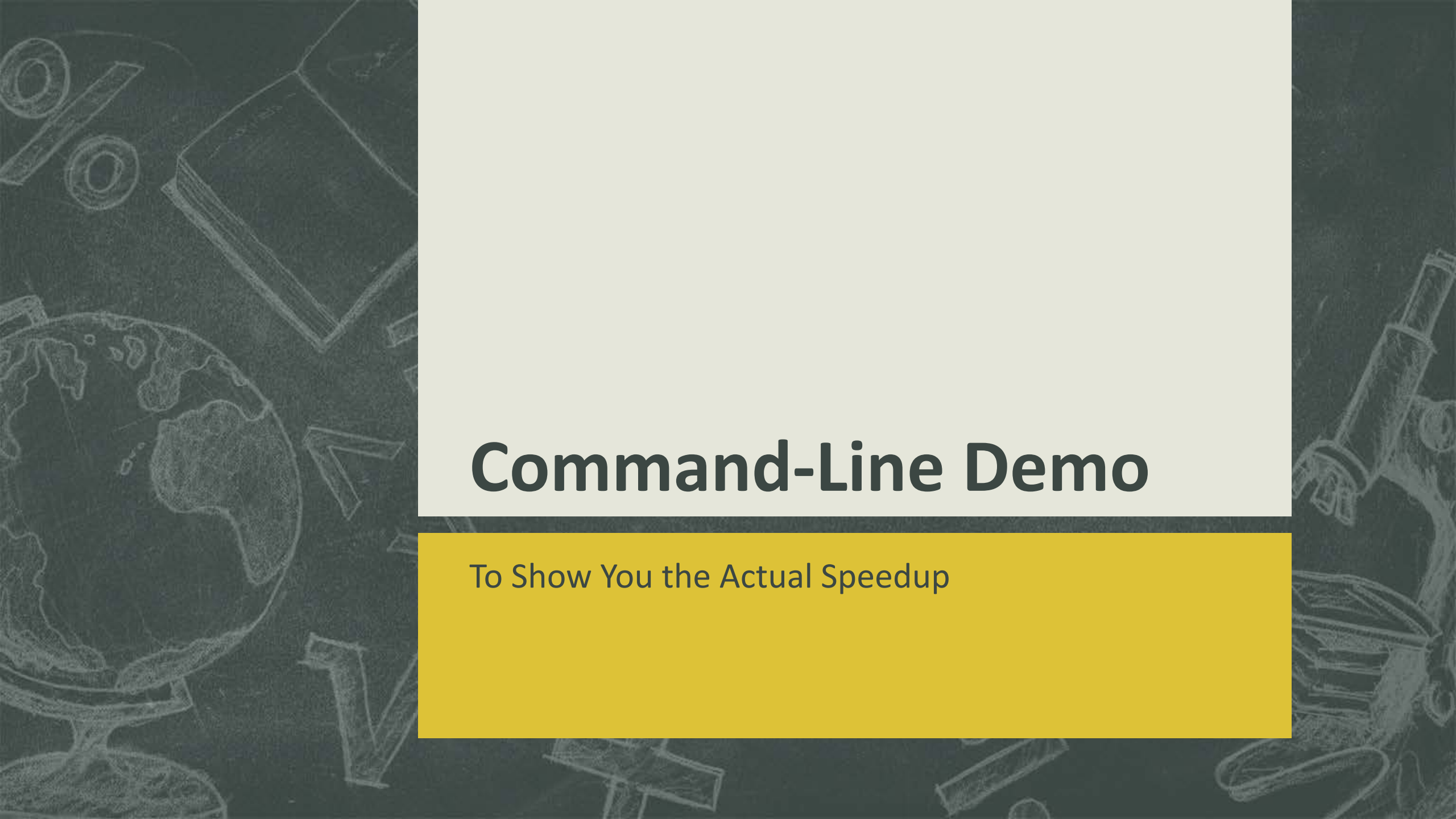
Insert 2^s keys into table of size 2^{s+1}

Achieves~10x speedup then naïve implementation



Lookup 2^s keys in a table of size 2^{s+1}

Basically the same performance



Command-Line Demo

To Show You the Actual Speedup

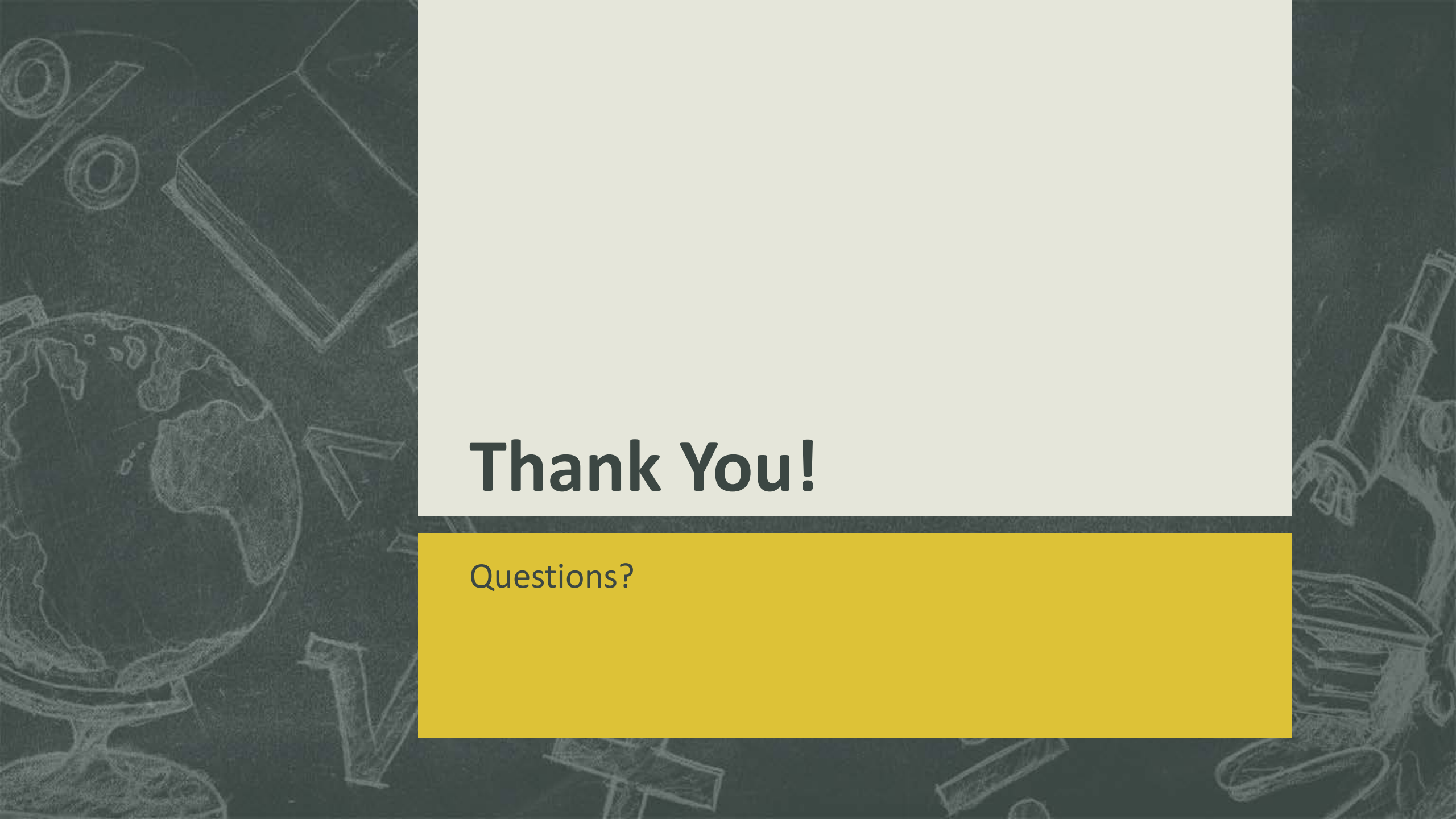
References

Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2009. **Real-time parallel hashing on the GPU.** ACM Trans. Graph. 28, 5, Article 154 (December 2009), 9 pages. DOI: https://doi.org/10.1145/1618452.1618500

URL:
https://www.cs.bgu.ac.il/~asharf/Projects/RealTimeParallelHashingontheGPU.pdf

Poster:
https://www.nvidia.com/content/GTC/posters/82_Alcantara_Real_Time_Parallel_Hashing.pdf

(Figures come from this paper and the poster)



Thank You!

Questions?