Hash-Based Algorithms for Discretized Data

Presented at:

Colorado State University

April 6th, 2012

Authors: Rachel Robey[1], David Nicholaeff[2], Bob Robey[2]

[1]Los Alamos High School

[2]Los Alamos National Laboratory

LA-UR-12-01565

Hash-based Algorithms for Discretized Data

Hash-based algorithms are shown to have the potential to speed-up many algorithms by over a factor of a hundred on the CPU. These algorithms include sorting, neighbor searching, remapping, and data table lookup, all common operations for mesh-based calculations. In addition, hash-based algorithms are straight-forward to port to the GPU. The GPU algorithms can give additional speed-ups of around 50-100x faster yielding a total speed-up over existing methods of over 10,000x. The cumulative effect of these algorithms is to enable calculations to scale to the exascale regime.

Presented by:Rachel Robey, David Nicholaeff, and Robert Robey

Los Alamos High School and Los Alamos National Laboratory

April 6th

Colorado State University

Rachel Robey is a junior at Los Alamos High School and is considering CSU for her undergraduate degree in a technical field such as applied math, science or engineering.

David Nicholaeff is a recent graduate of UCLA with a BS in Mathematics and Physics. Currently, he is working at Los Alamos National Lab. He will be attending Oxford for Mathematics and the Foundations of Computer Science in the Fall of 2012.

Robert Robey is a staff member at Los Alamos National Laboratory in the Eulerian Applications group.

Hashing and Spatial Data

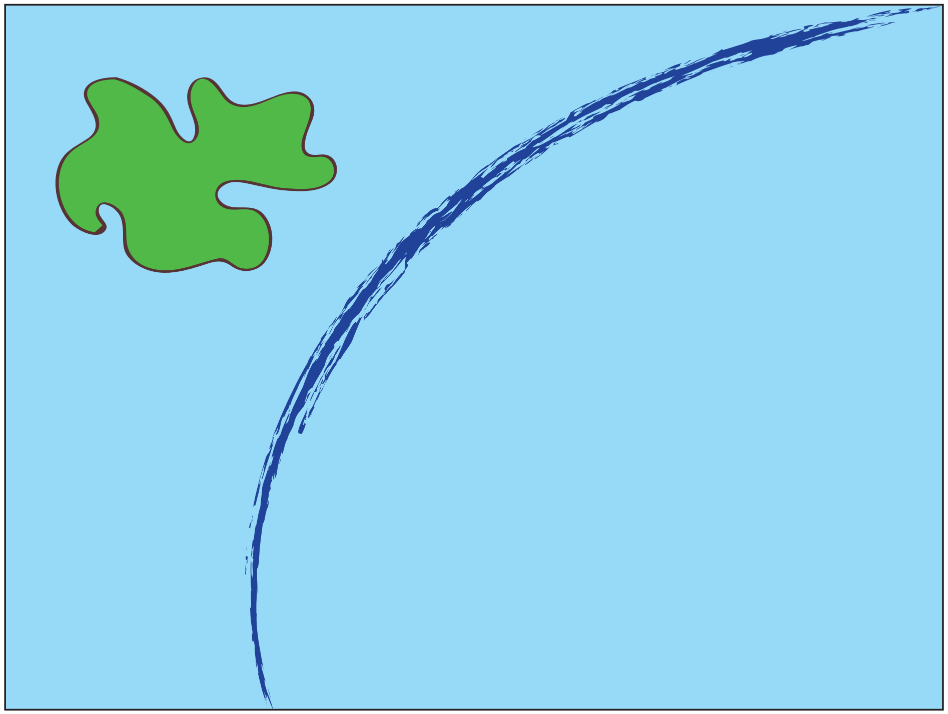
- Themes:
  - $O(n \log n) \rightarrow O(n)$
  - Trade space for time
  - Exploit underlying data structure
  - Parallel characteristics suitable for the GPU

Spatial Hashing

- Applications of Spatial Hashing:
  - Sorting
  - Remap
  - Neighbor calculations
  - Spatial queries
  - Table lookup
- Applies for any discretized data – values spread out relatively uniformly (vs. clustered)

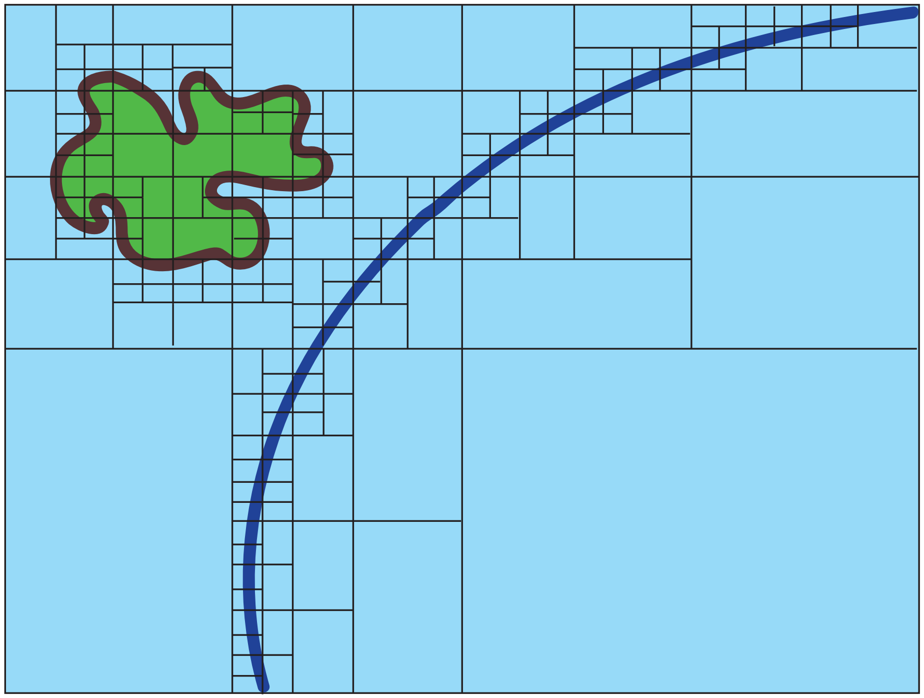
Numerical Methods Problem

- A model is governed by differential equations in a continuous domain
- Here is an example of a shock/ water wave approaching an island



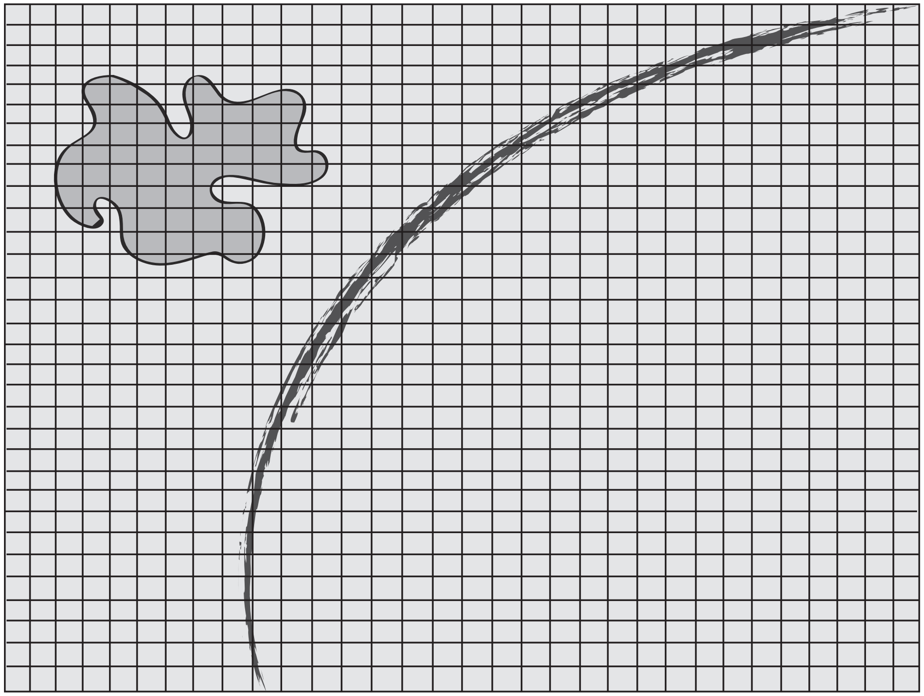
Computational Mesh

- Discretize the space to allow for a finite representation on the computer and find a representative value for the continuous function in each cell
- In discretized data, there exists a δ such that $d(A_i, A_j) \geq \delta$
- Discretize to optimize the modeling
  - Maximize benefit (reduce error) for the work done on the mesh
- In this example, there is higher resolution in areas of greater interest and high gradients (wave front and shoreline)

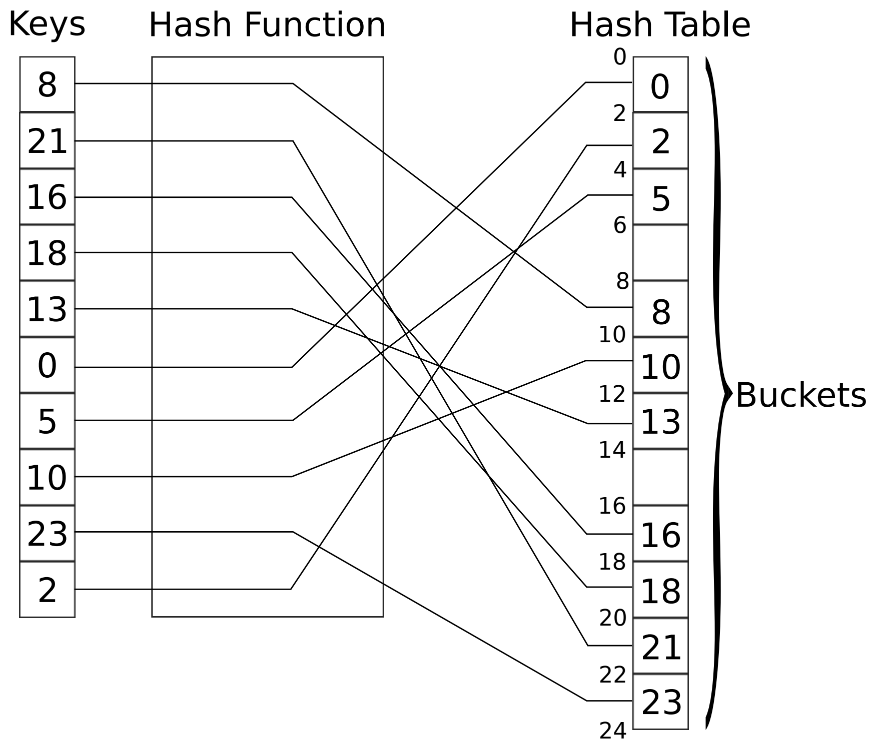


Hash Abstraction

- Abstract the discrete space to another, finer mesh to allow for certain spatially-dependent operations to be performed with hash-based methods efficiently
- In this example, each cell from the discrete space can be mapped to a unique refined hash cell, or refined bucket
- Results in data independence (and hence parallelism) for cell operations such as neighbor lookup and remapping, vs. comparison-based or tree-based methods

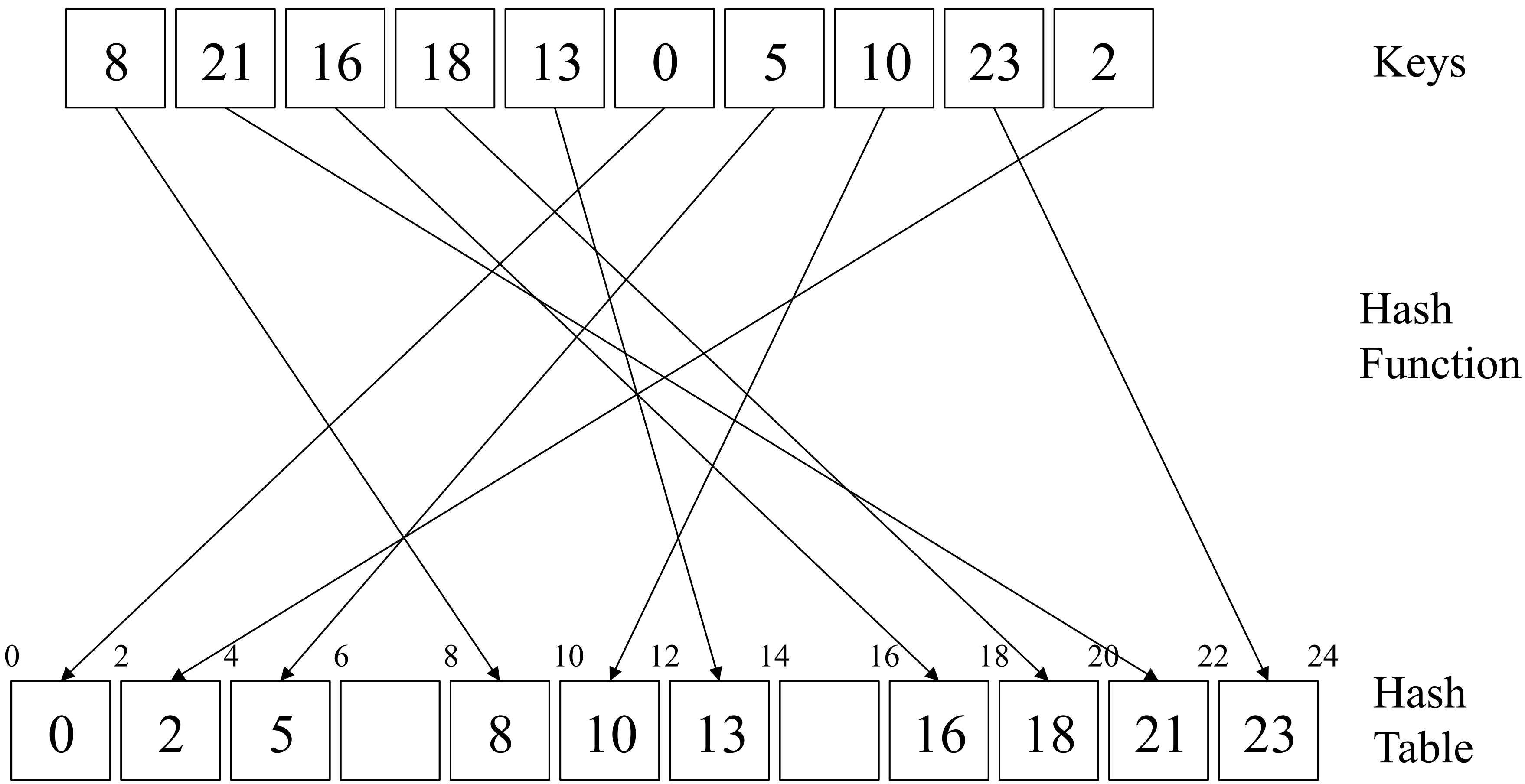


Hash Table for Sort



The data (keys) are mapped into a hash table that covers the range with buckets the size of the minimum difference. In this example, the minimum difference is 2, the minimum value is 0, and the range is 23. Each key is mapped using a ratio index=(int)((key-val_{min})/$diff_{min}$). Empty buckets are left in the hash table, which are removed in another pass.

Hash Sort

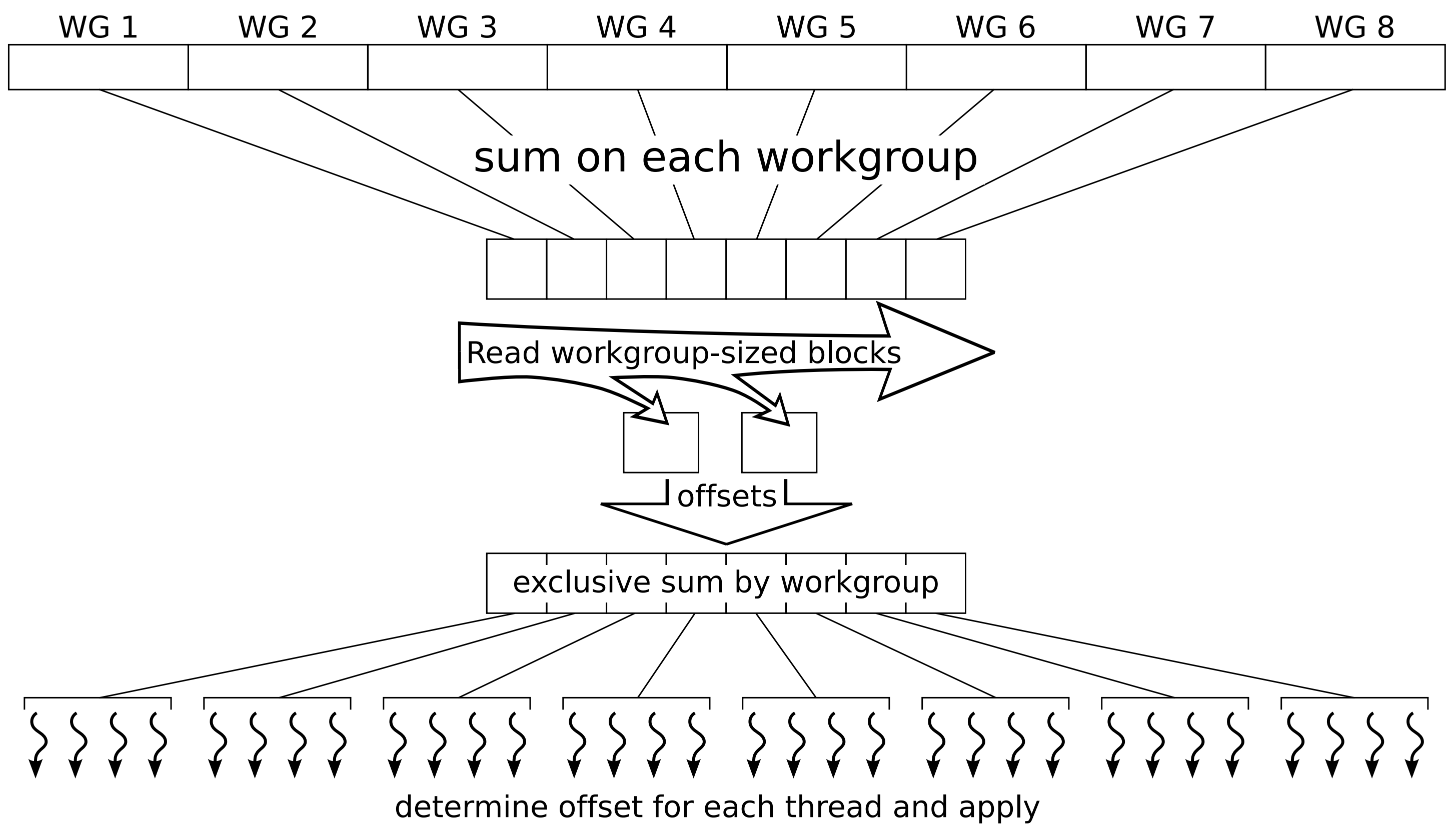


8
21
16
18
13
0
5
10
23
2
Keys
Hash
Function
0
2
4
6
8
10
12
14
16
18
20
22
24
0
2
5
8
10
13
16
18
21
23
Hash
Table

Hash CPU Sort Code

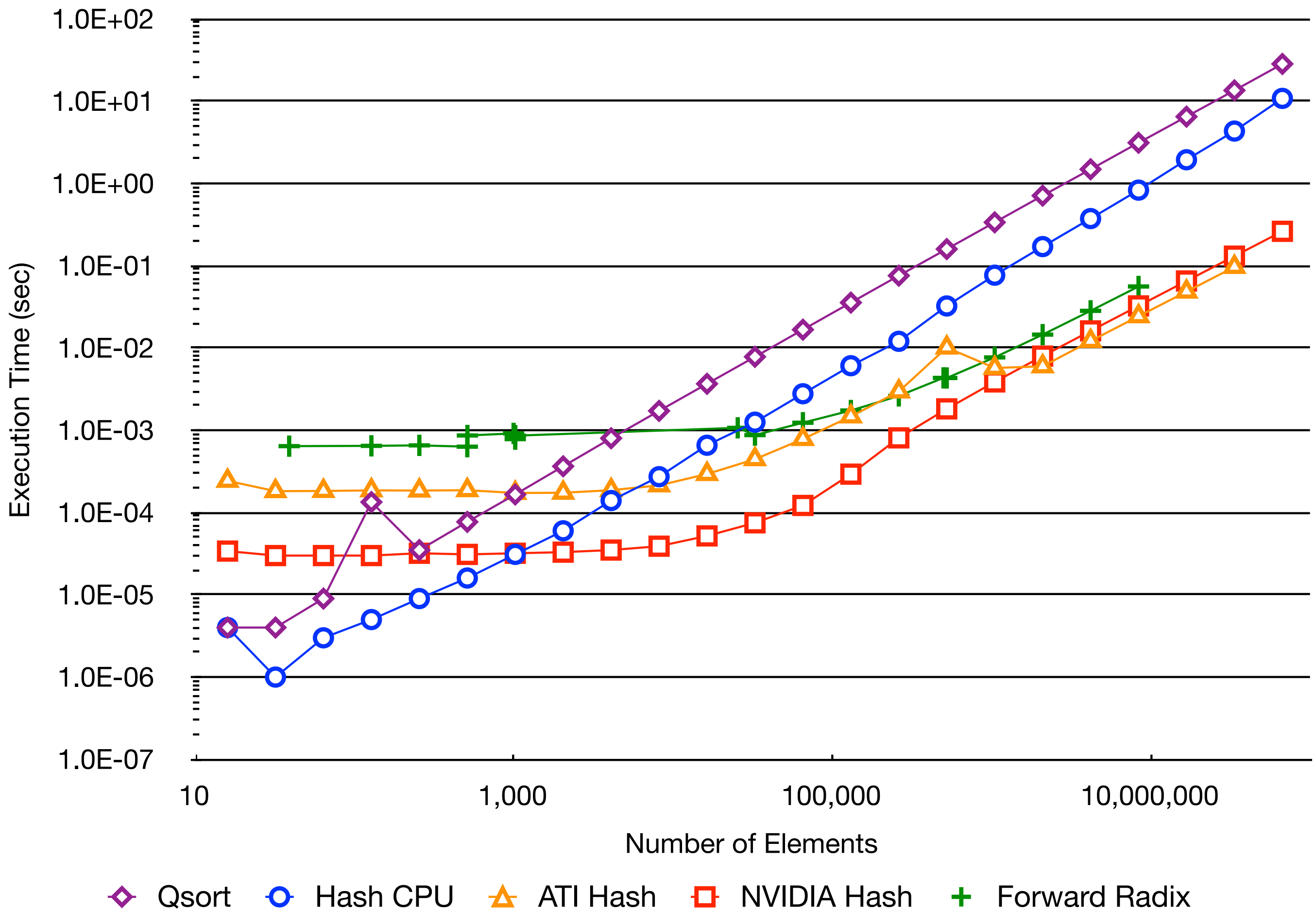
```
//create hash table with buckets of size mindx and add one for truncation
hash_table_size = (int)((max_val-min_val) / mindx + 1);

//set all hash table elements to -1 to tag empty buckets
memset(hash_table, -1, hash_table_size*sizeof(int));

for( i = 0; i < unsorted_array_length; i++ ) {
    hash_table[(int)((arr[i]-min_val)/mindx)] = i;
    //place the index of each array element into its respective bucket
}

//remove empty buckets and condense the sorted array
int count = 0;
for( i = 0; i < hash_table_size; i++ ) {
    if(hash_table[i] >= 0) {
        sorted[count] = array[hash_table[i]];
        count++;
    }
}
```

GPU Prefix Scan

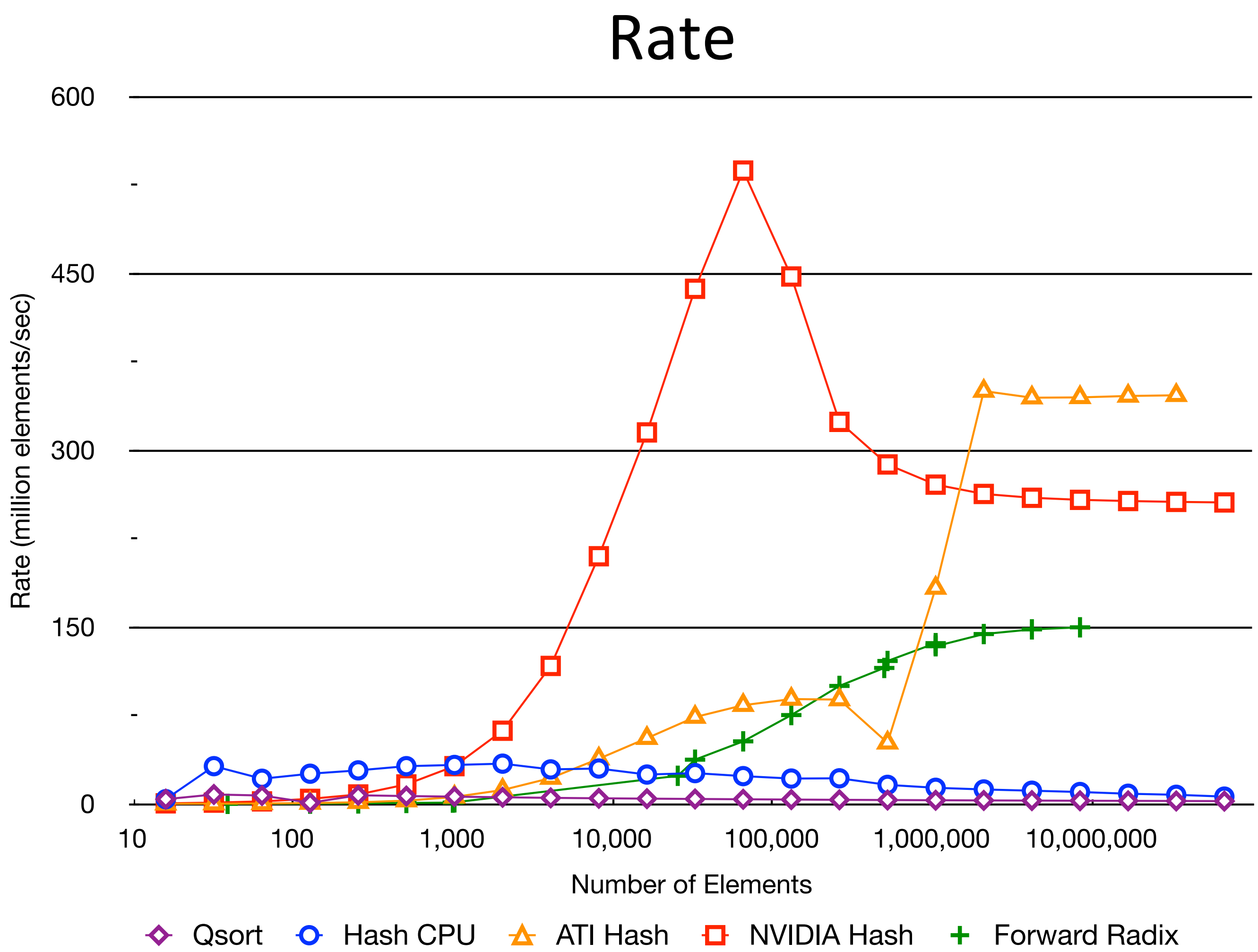


1D Sort Performance

Execution Time

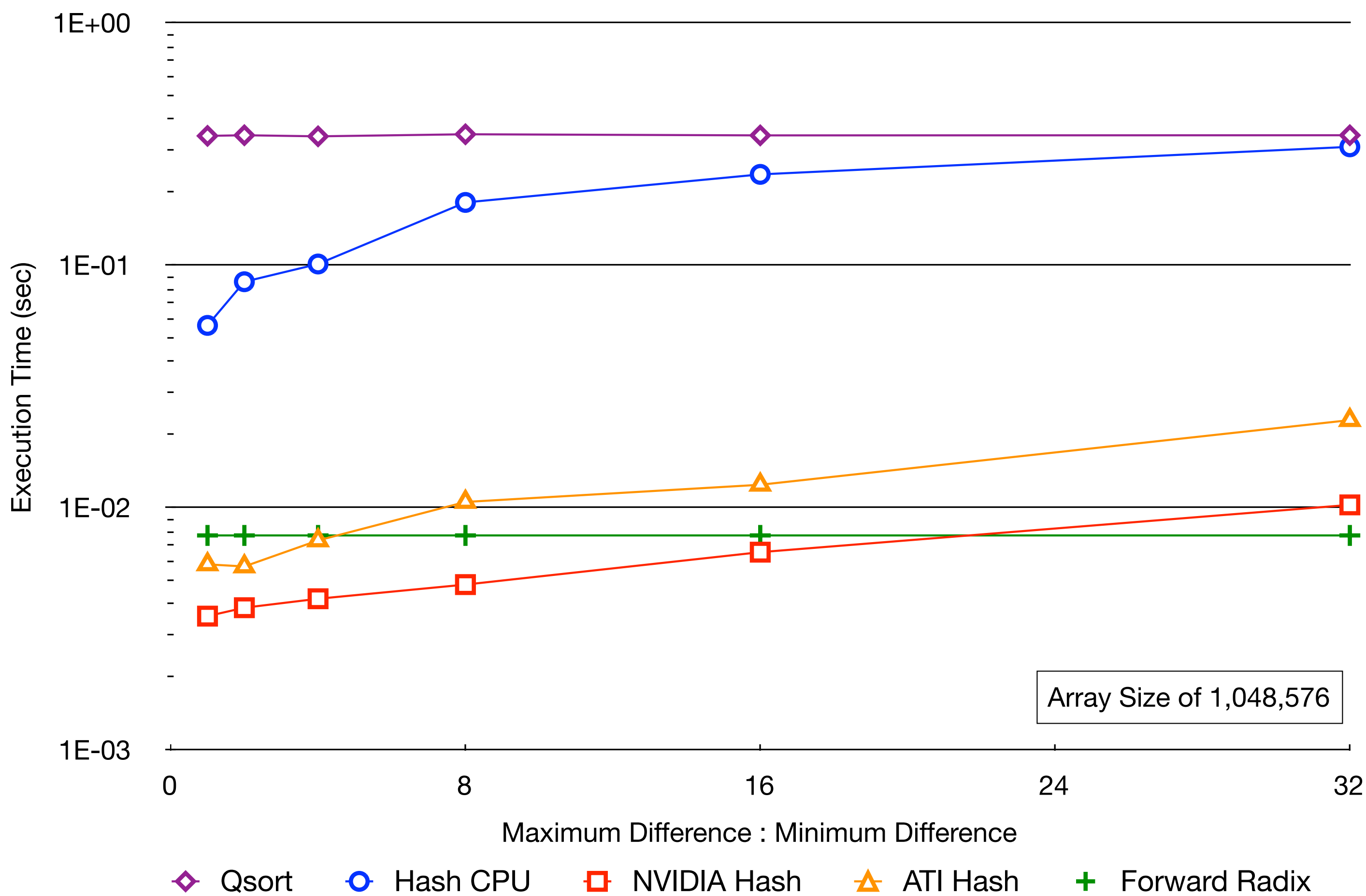


1D Sort Performance



1D Sort Performance

Growing Hash Table Size



Summary of Sort Results

Beat fastest general purpose GPU sort on record by 2.5-3.5x

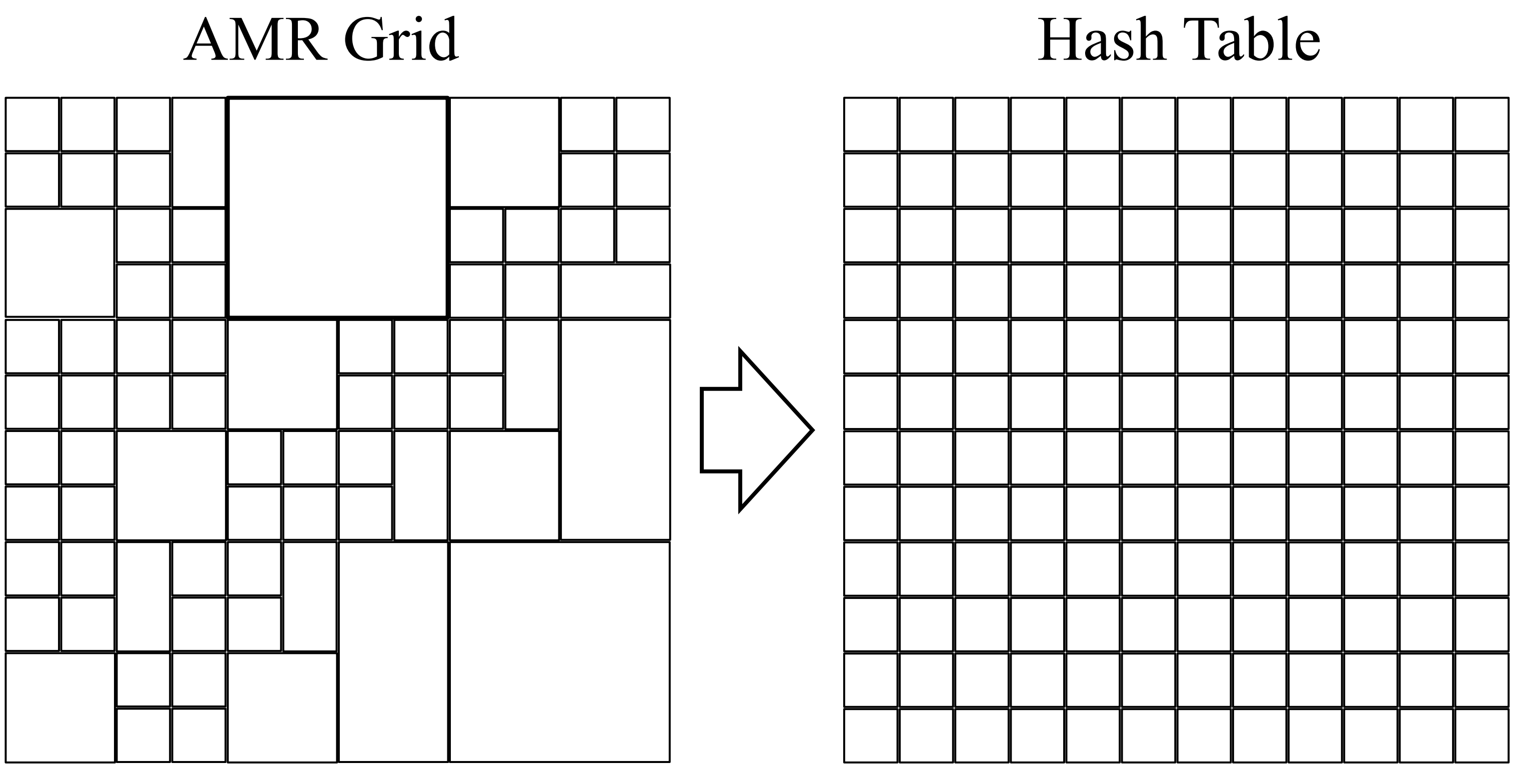
Room for improvement in implementation

Implementation is portable

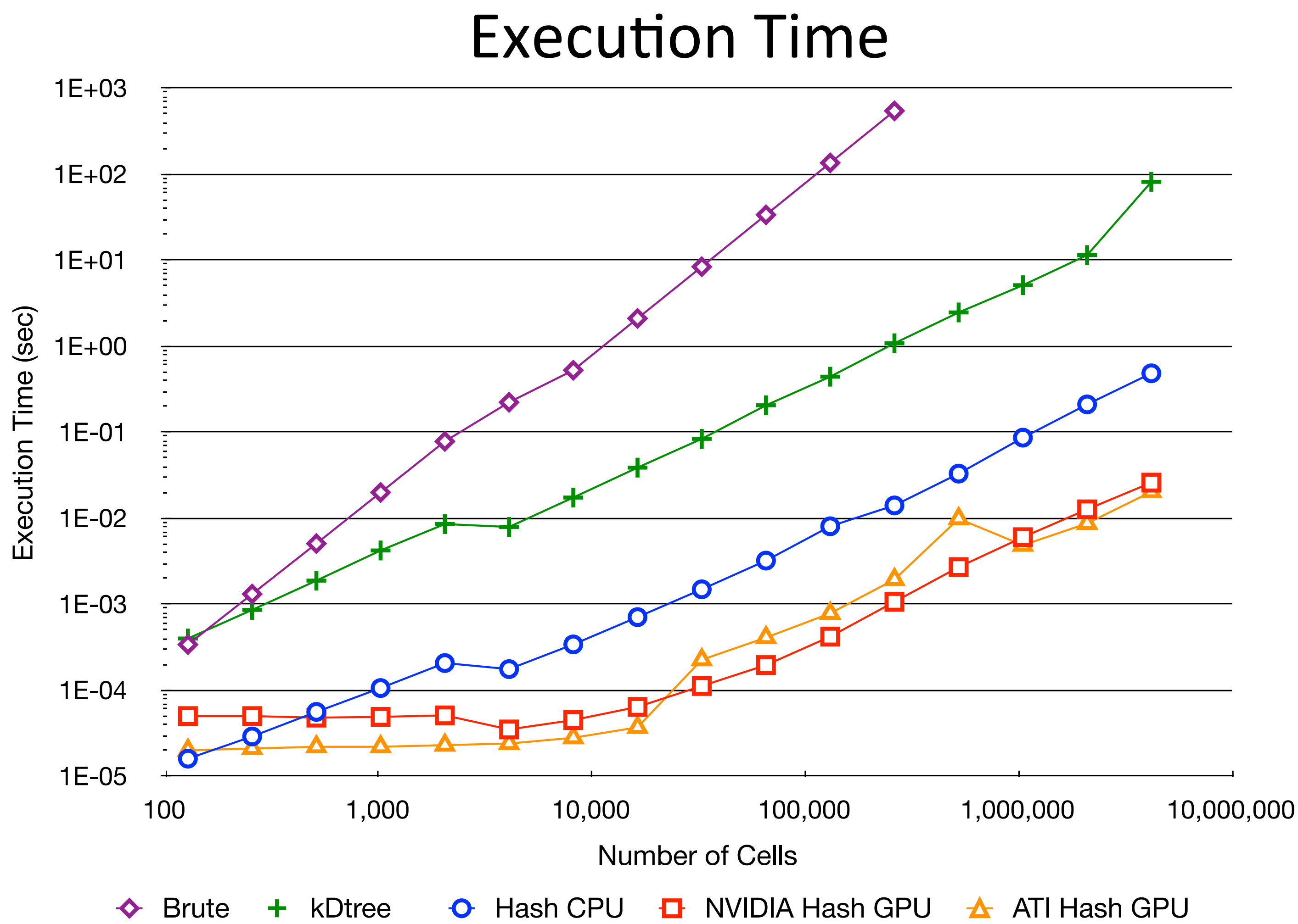
Scan is fastest OpenCL scan available

Key for irregular memory operations

Neighbor

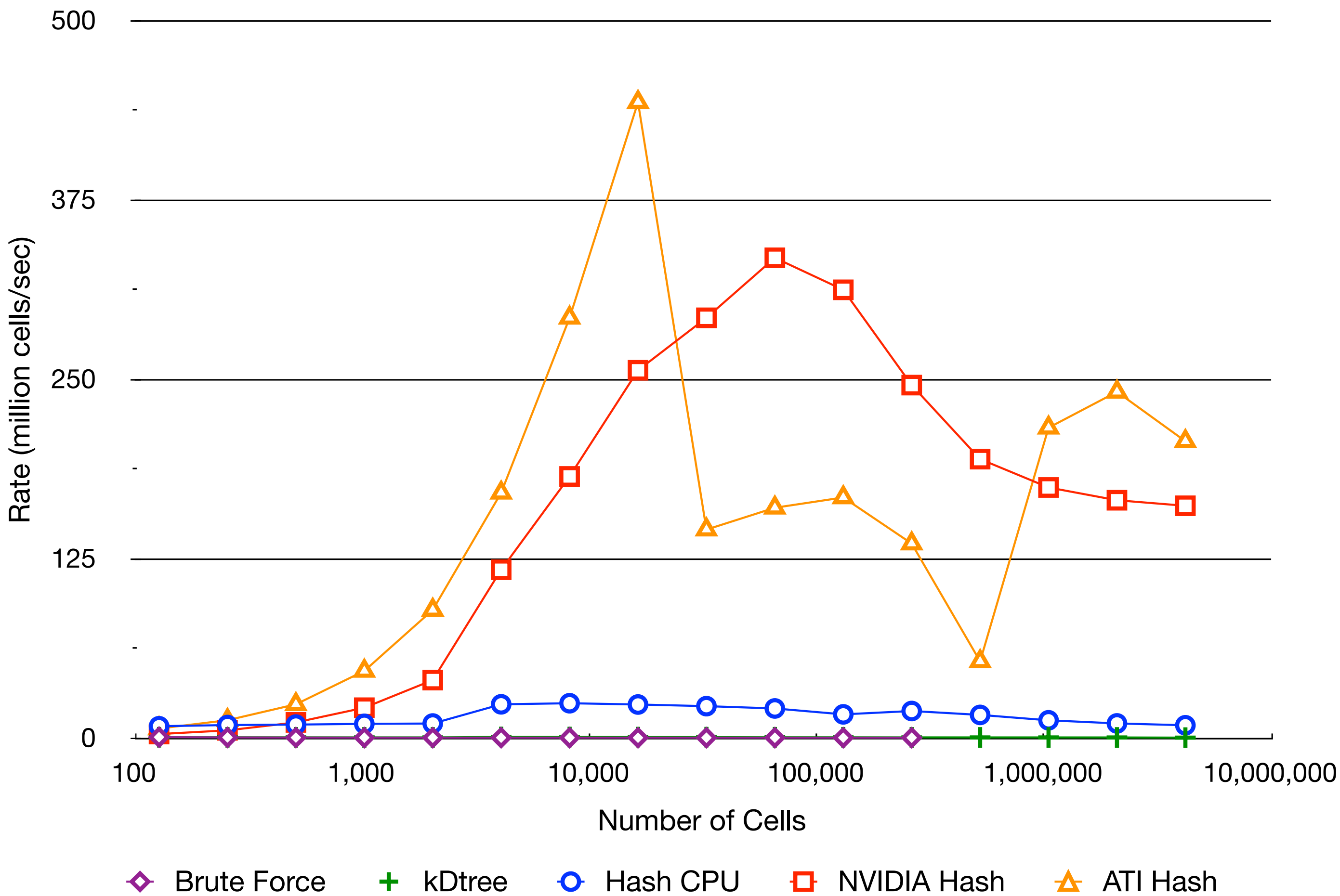


1D Neighbor Finding



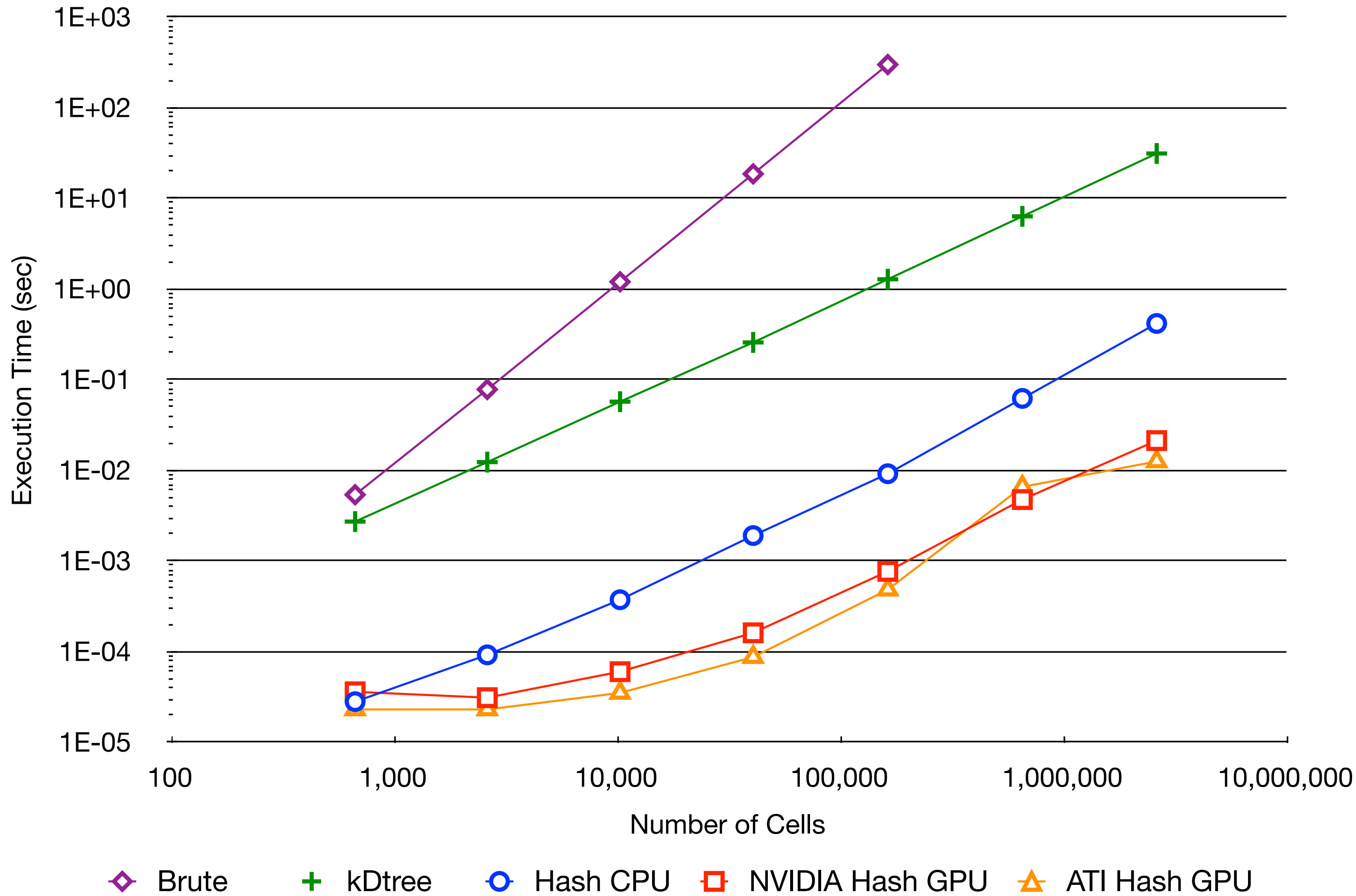
1D Neighbor Finding

Rates

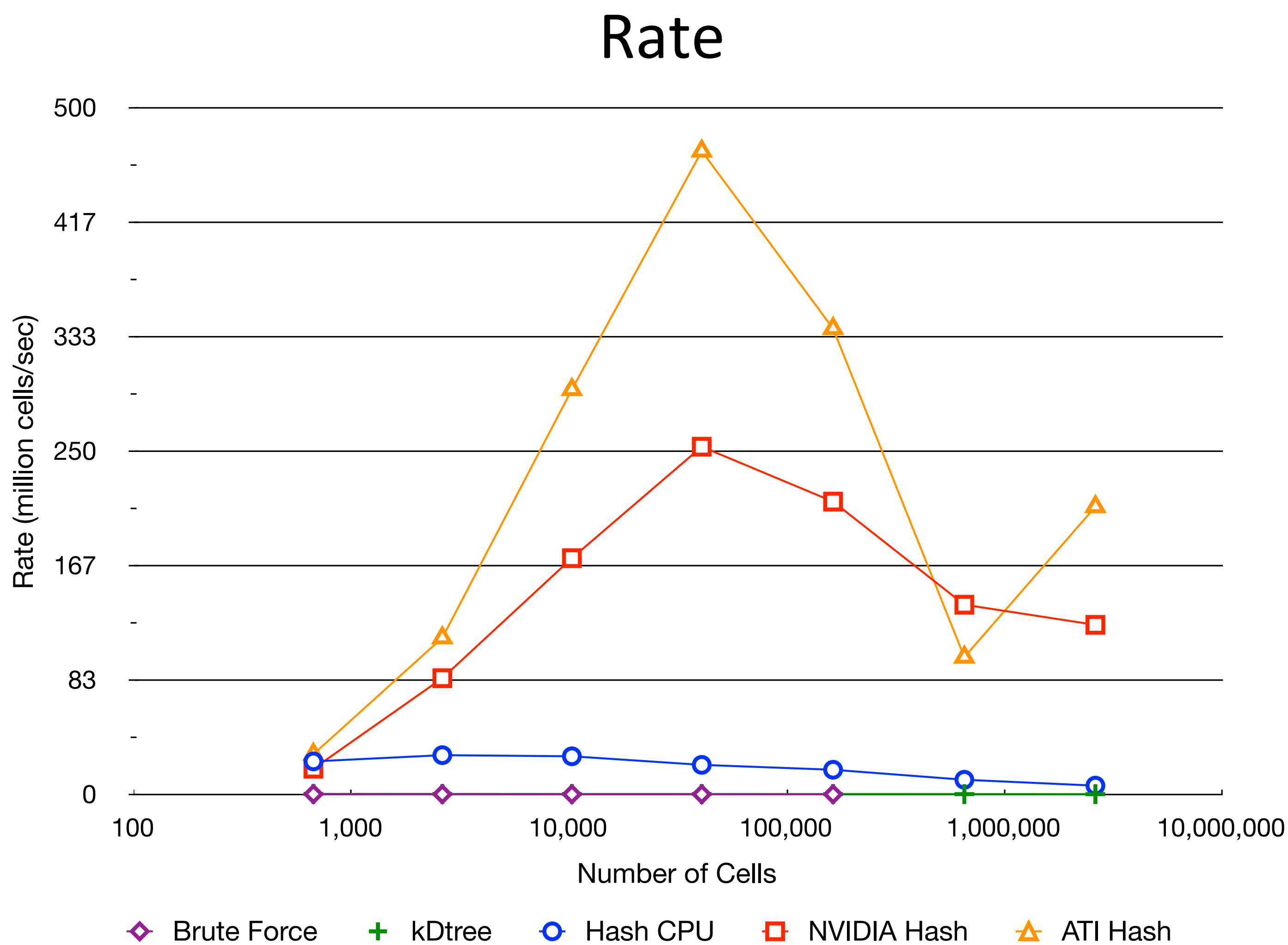


2D Neighbor Finding

Execution Time



2D Neighbor Finding



Summary of Neighbor Results

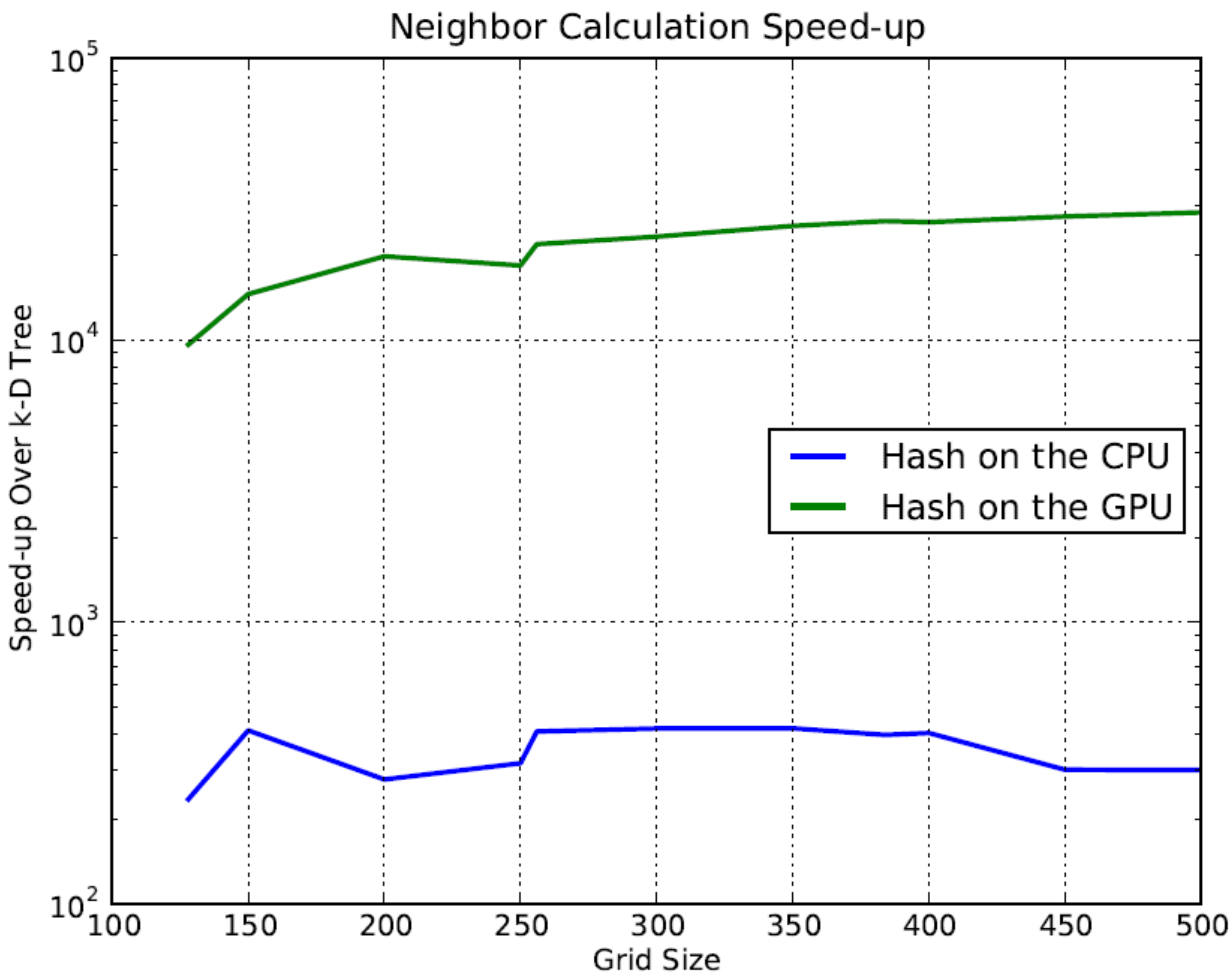
Speedup is order 100x on the CPU

Another 100x speed-up on the GPU

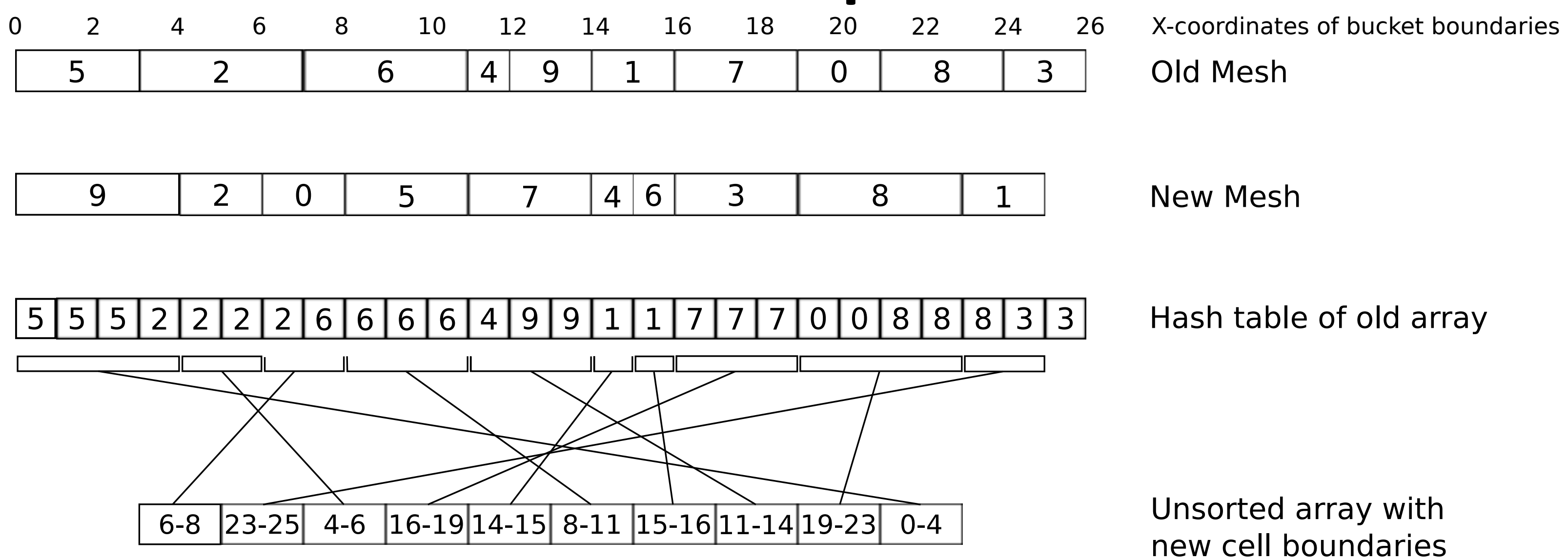
Total 10,000x speed-up

Performance can still be improved for growing hash sizes

Speed-up from AMR study



Remap

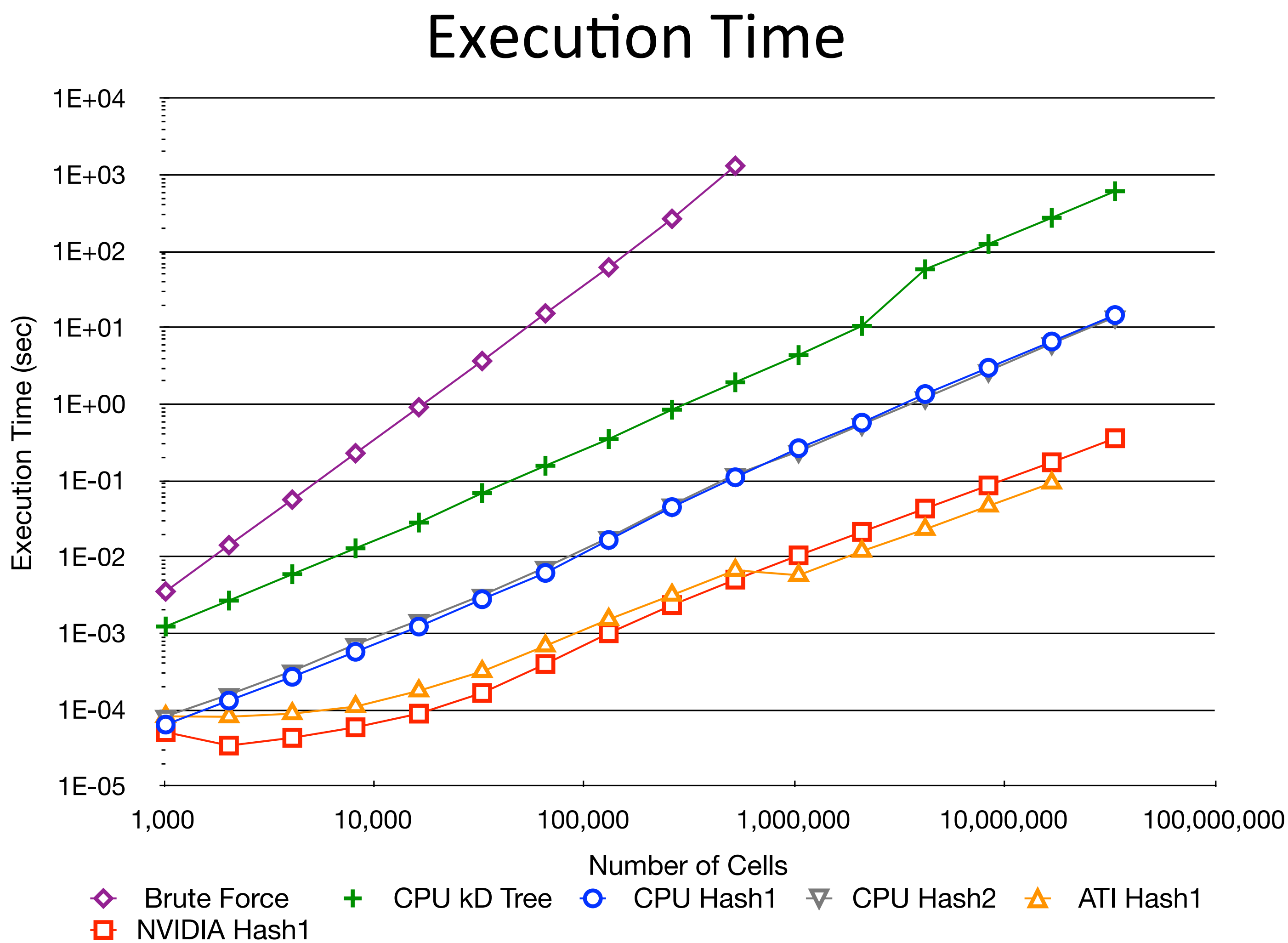


- Transfers state variables from one mesh representation to another
- Determine fractions of the old cells contained in each of the new cells
- Hash-based approach: each cell from the old mesh writes its index into each of the refined cells it contains in a hash table. Each of the new cells can directly look up which cells it contains by referencing that hash table
- Assumption that every bucket/subspace is fully contained, limits possible structures

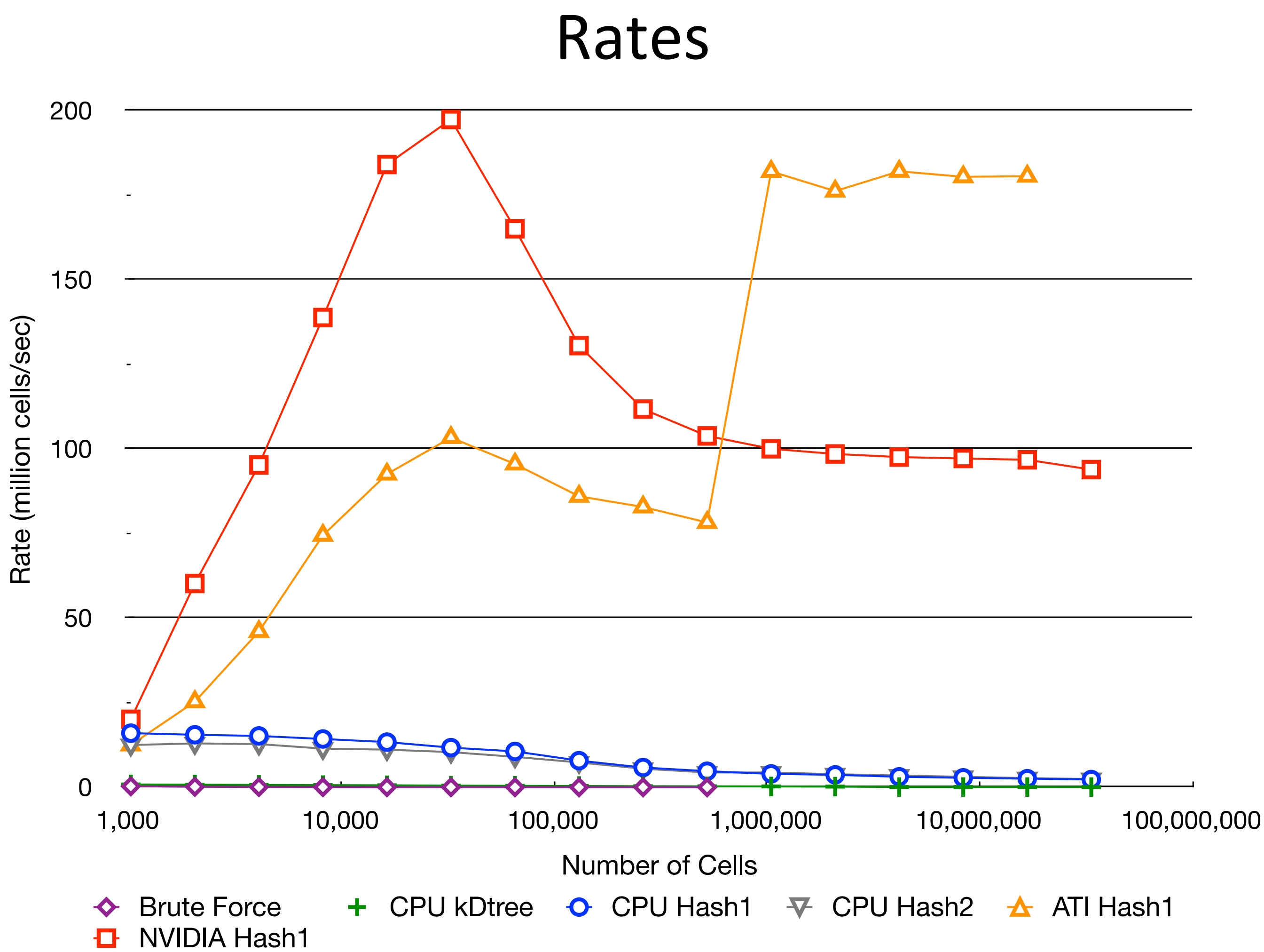
Alternate Method

- Allows more flexibility in mesh structure, works from sorts of the two meshes
- In 1-D fairly simple overlay cell boundaries of the two meshes
- Looping through both meshes in sync, each block defined by combination of boundaries from both meshes is transferred

1D Remap Performance

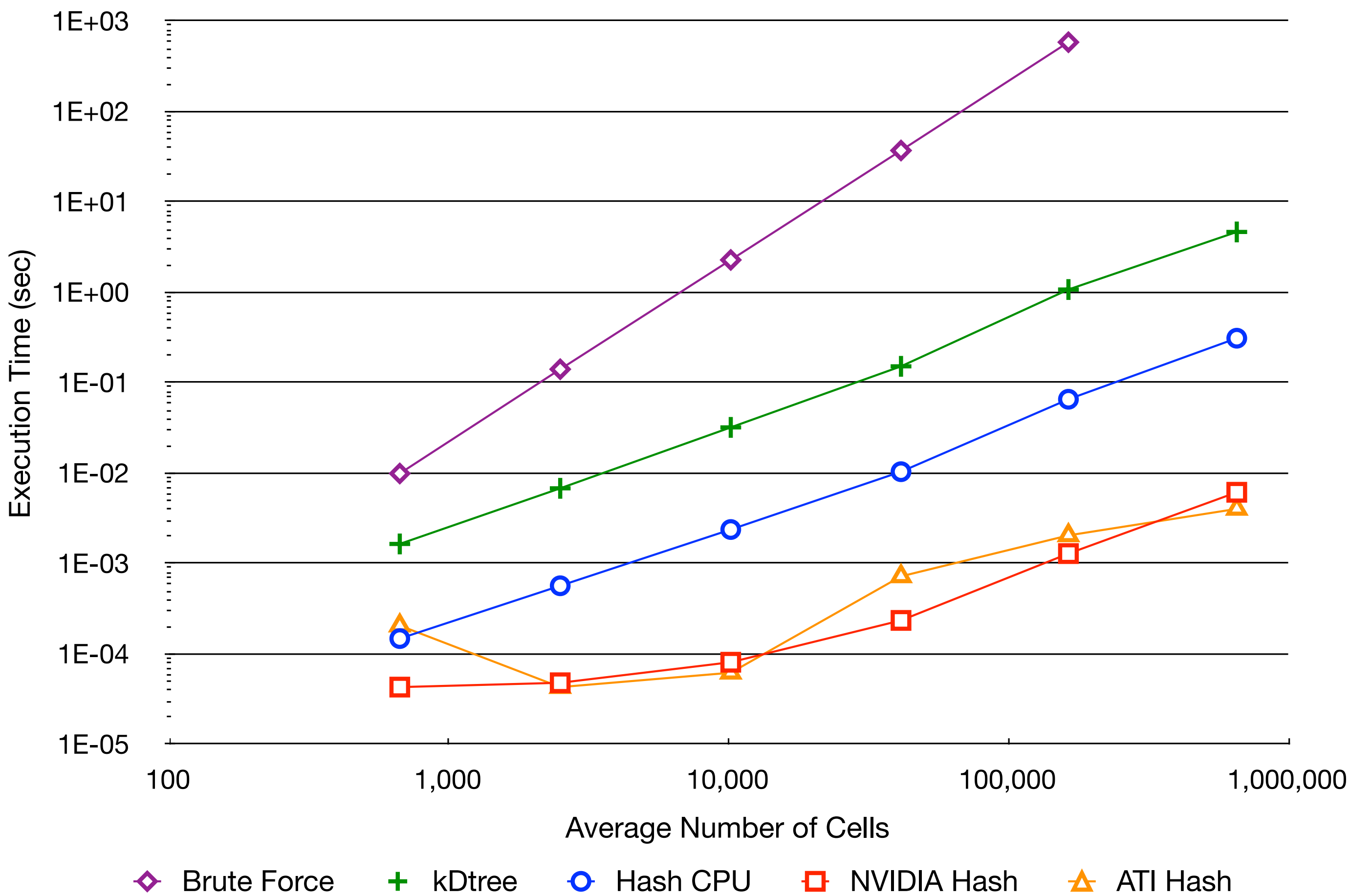


1D Remap Performance



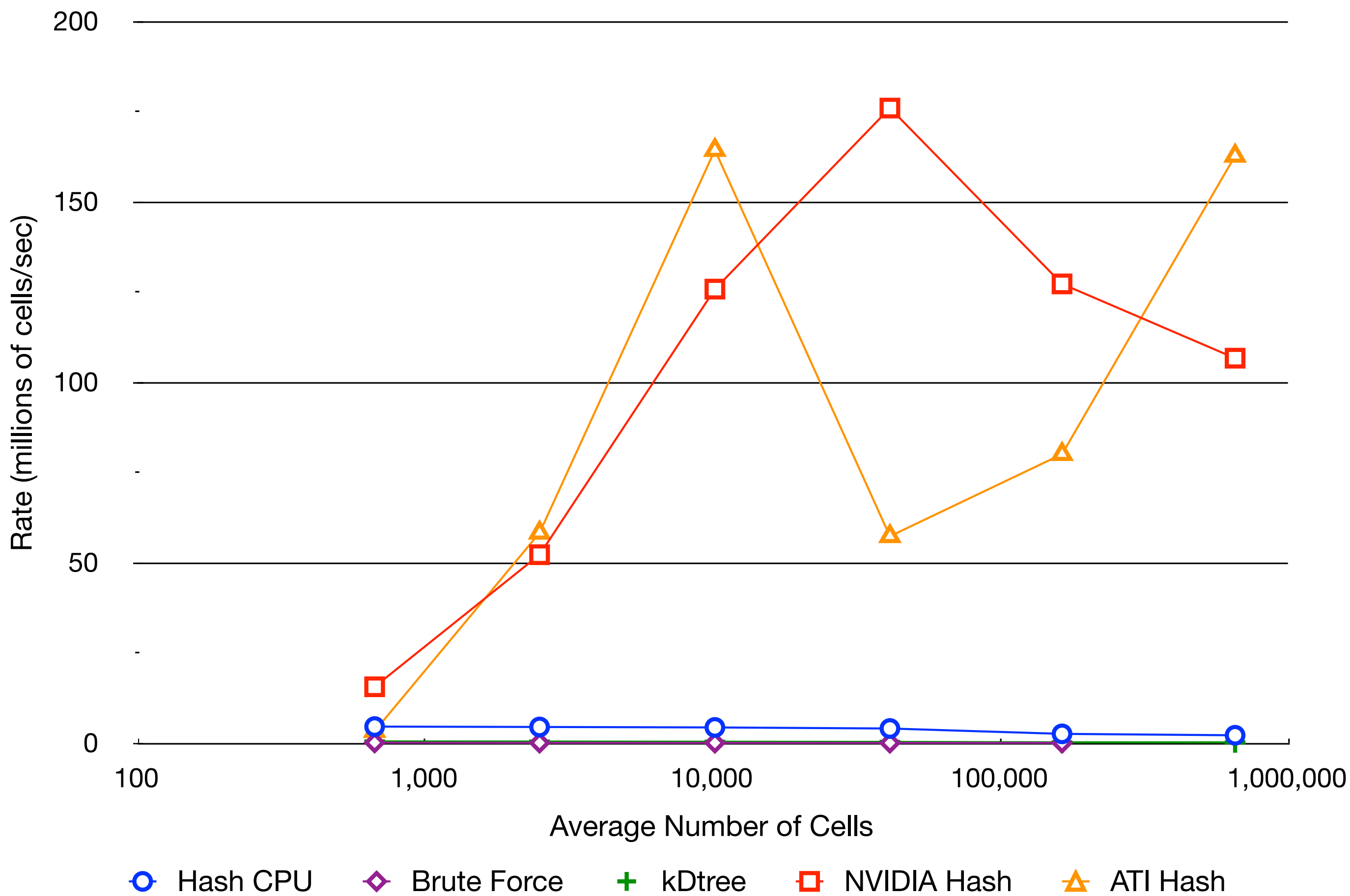
2D Remap Performance

Execution Time



2D Remap Performance

Rates



Summary of Remap Performance

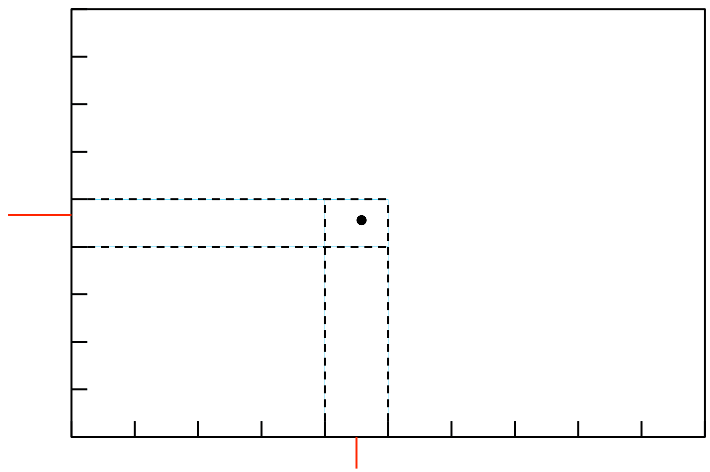
Speed-ups order 10x on the CPU (compared to k-D tree)

Speed-ups approaching 100x on the GPU

Still some room for improvement in implementation

Table Look-up

- Hash-based look-up of two axes to find row and column
- Data ordered in hash table can be referenced directly using key and using the hash function used in mapping
- Look-up and interpolate



Hash CPU Table Look-up Code

```
//computes a constant increment for each axis data look-up
double density_increment = (density_axis[50]-density_axis[0])/50.0;
double temp_increment = (temp_axis[22]-temp_axis[0])/22.0;

for( i = 0; i < isize; i++ ) {
    //determine the interval for interpolation and the fraction in the interval
    int temp_slot = (temp_array[i]-temp_axis[0])/temp_increment;
    int density_slot = (density_array[i]-density_axis[0])/temp_increment;
    double xfrac = (density_array[i]-density_axis[density_slot] /
        (density_axis[density_slot+1] - density_axis[density_slot]);
    double yfrac = (temp_array[i]-temp_axis[temp_slot]) /
        (temp_axis[temp_slot+1]-temp_axis[temp_slot]);

    //bi-linear interpolation
    value_array[gid] =      xfrac * yfrac     * dataval(islot+1 + (jslot+1) * xstride)
                     + (1.0-xfrac)* yfrac     * dataval(islot   + (jslot+1) * xstride)
                     +      xfrac *(1.0-yfrac)* dataval(islot+1 +  jslot    * xstride)
                     + (1.0-xfrac)*(1.0-yfrac)* dataval(islot   +  jslot    * xstride);
}
```

Table Look-Up Performance

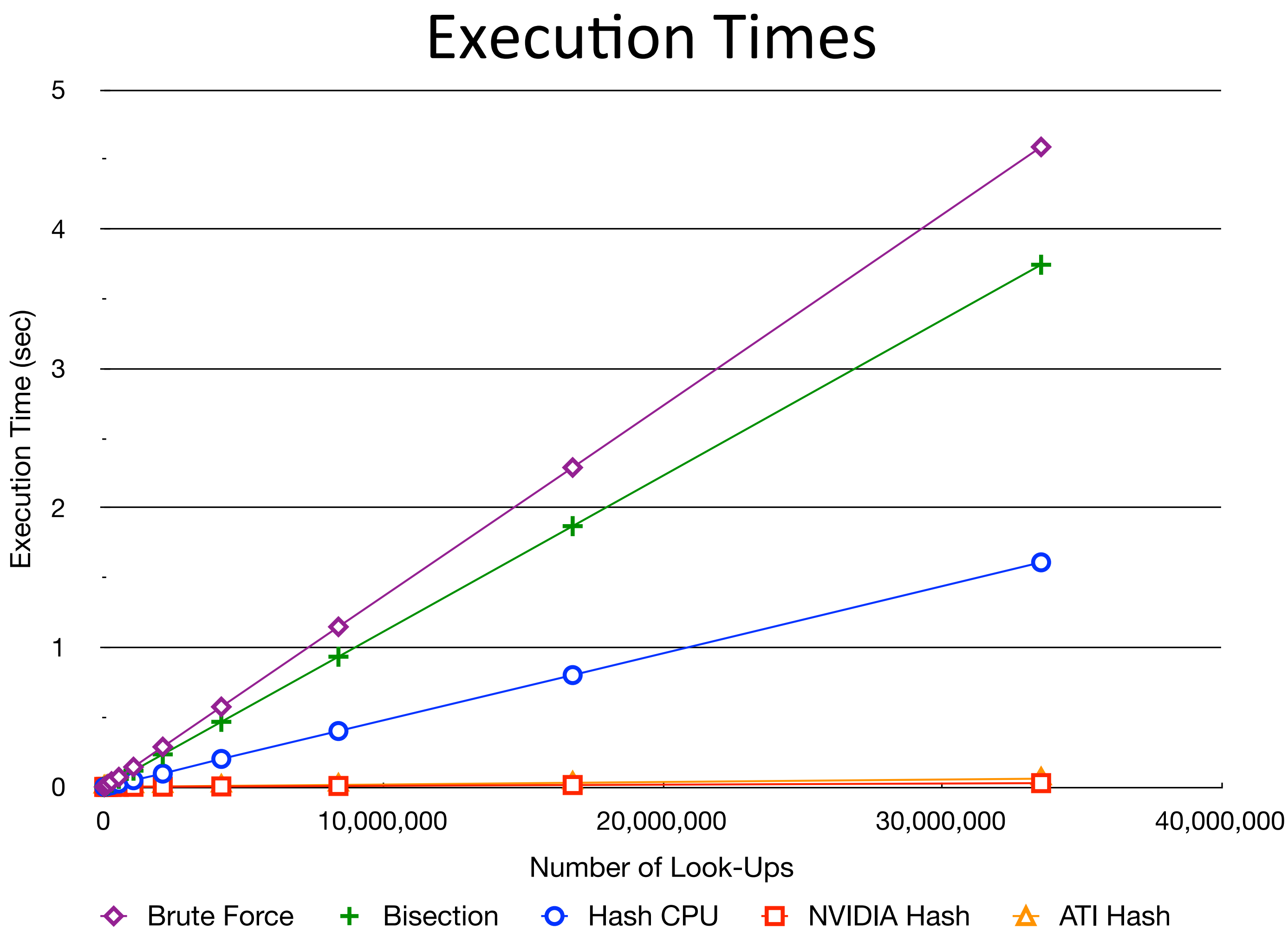


Table Look-Up Performance

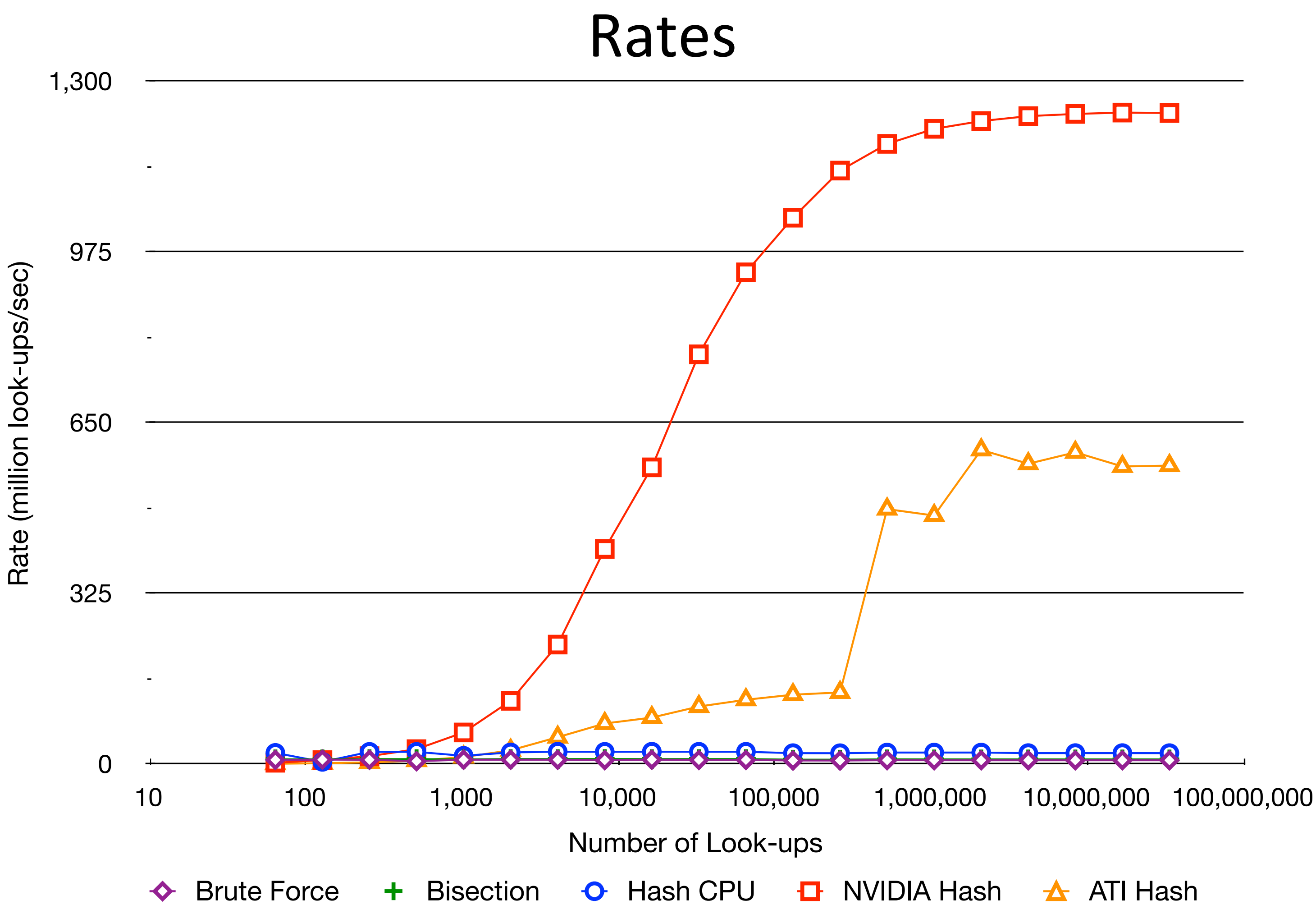


Table Look-up

Comparing to Bi-section method

Somewhat simplified case

Factor 2x speed-up on CPU

Factor 100x speed-up on GPU

Speed-Up Summary

Note: Based on problem sizes (# of elements or cells) of around 2 million. Reference CPU is generally accepted method for that operation: quicksort, kD-tree, and bisection.

	CPU Hash	NVIDIA	ATI	NVIDIA	ATI
Relative to	Reference CPU	CPU Hash		Reference CPU	
Sort	4.16	21.5	28.6	89.3	118.9
Sort 2-D	16.2	26.2	37.8	424.1	611.5
Neighbor	54.4	16.6	24.2	903.5	1316.0
Neighbor 2-D	75.5	19.1	19.1	1444.0	1445.3
Remap	18.4	26.9	48.1	495.2	885.8
Remap 2-D	13.6	42.2	61.6	574.0	837.8
Table	2.44	55.7	27.2	136.2	66.5

- Speed-ups are a combined result of:
  - replacing an O(n log n) algorithm with an O(n) algorithm
  - harnessing the massively parallel compute capability of the GPU

Conclusion

- For every spatial mesh operation there must be an efficient hash-based algorithm
- Optimizing the mesh through the iterative process formalized in the differential discretized data, we also optimize the hash operation. By optimization, we don't mean the fastest, but rather the most benefit for the work done

- You do not have to write complicated code to get fast performance -- this code is ridiculously simple
- Extensions – unstructured, MPI, collision, hash table size reduction