Rapids GroupBy

Aaron Nightingale
Christopher Patterson
Jersin Nguetio
Menglu Liang
Tonglin Chen

Design Overview

Data structure

Provided data set is in column major.

Matrix is flattened to 1D array for easy memory copying

Input: 1) Key matrix : A matrix with M rows and N columns; rows of this matrix are the “keys” of the reduce-by-key operation

2) Value matrix : A matrix with M rows and J columns

3) Operation array: An array of J reduction operations

Output: 1) Keys: U rows and N columns, where U is the number of unique keys from the input Keys matrix; Row “i” of the output Keys matrix corresponds to row “i” of the output Values matrix.

2) Values: U rows and J columns, where U is the number of unique keys from the input Keys matrix;Row “i” of the output Values matrix corresponds to row “i” of the output Keys matrix;

Data structure example

Sample key and value matrix

key\entries	0	1	2	3	4	5
0	1	1	4	9	8	4
1	2	2	5	2	9	5
2	3	3	6	4	1	6
value						
0	1	3	5	7	1	8
1	2	4	9	3	1	9

Expected output on (max, count)

Output Key and Value Matrix

key\entries	0	1	2	3
0	1	4	9	8
1	2	5	2	9
2	3	6	4	1
value				
0	3	5	7	8
1	2	1	1	2

Implementation

Thrust

- Library providing parallel data structures and algorithms
- Reduce, scan, copy, etc.
    - Customizable for different data representations
    - Reduce by key for sorted key input
    - Special reduction operations for add, min, max, etc.
- Allows for quick implementation for our case of many value operations
- Less control over more advanced CUDA features
    - Shared memory
    - Streams
    - etc.

Data Structures

- thrust::device_vector
- thrust::host_vector
- thrust::device_ptr
- Etc.

Algorithms

- thrust::sort
- thrust::reduce
- thrust::exclusive_scan
- Etc.

GroupBy

Input:

- mxn matrix
  - Each row corresponding to a key
  - Unknown number of unique keys

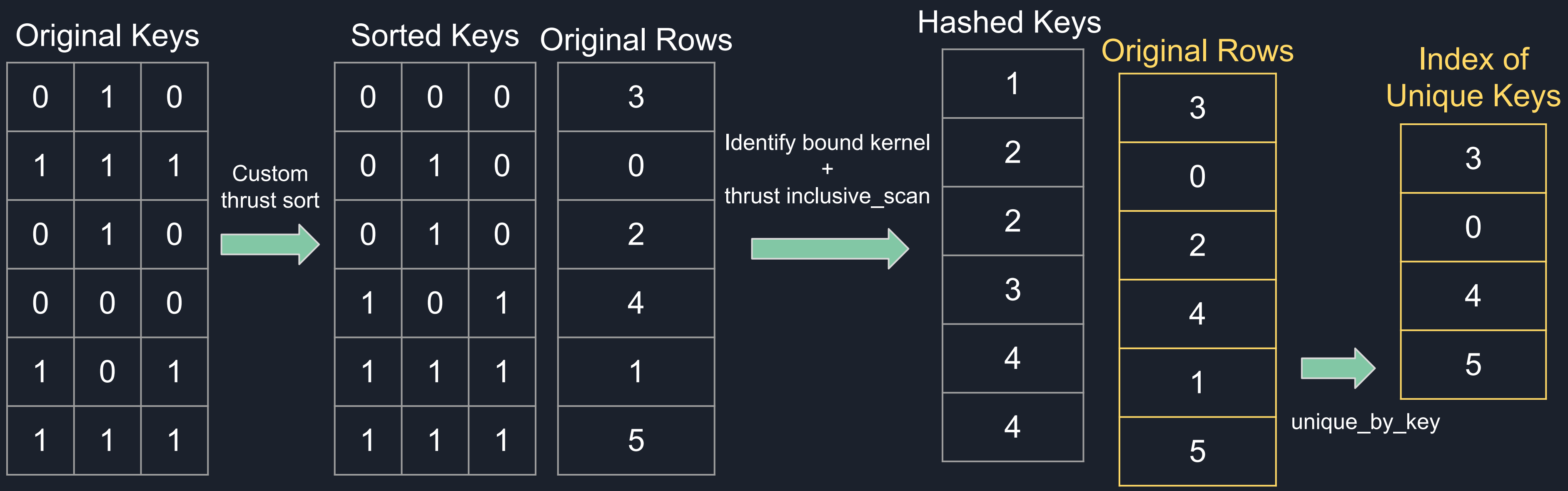
Output:

- pxn matrix
  - Each row corresponding to a unique key

Process:

1. Sort keys
2. Hash keys for easier compaction
3. Compact hashed keys to obtain just unique keys
4. Copy back original n columns for each unique key

GroupBy


Original Keys
0 1 0
1 1 1
0 1 0
0 0 0
1 0 1
1 1 1
Custom
thrust sort
Sorted Keys
0 0 0
0 1 0
0 1 0
1 0 1
1 1 1
1 1 1
Original Rows
3
0
2
4
1
5
Identify bound kernel
+
thrust inclusive_scan
Hashed Keys
1
2
2
3
4
4
Original Rows
3
0
2
4
1
5
unique_by_key
Index of
Unique Keys
3
0
4
5

Reduction Operations

Input:

- mxj matrix of values
  - Each row maps to the corresponding row in key matrix
- j reduction ops

Output:

- pxj matrix of reduced values
  - Each row now maps to the unique key matrix rows
  - Columns are values for a given reduction in the reduction op array

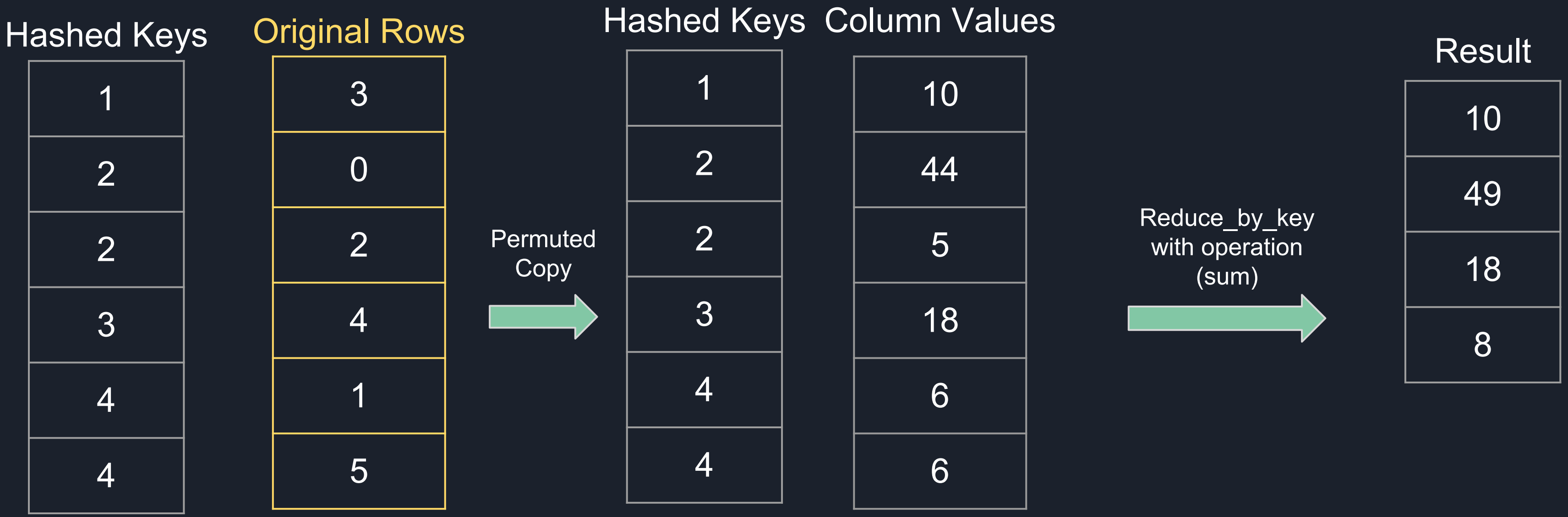
Process:

1. Use sorted index array to copy in a sorted column of value matrix
2. Perform reduce by key on column based on corresponding reduction operation
3. Copy reduction result to output matrix column
4. Repeat for all columns in matrix

Reduction Operations

For each value column we perform reduction

- Use correct operation from the op input array
- Copy result back to host



Hash-based Implementation

Hash-based Implementation

- GPU-based sorting algorithms still present on average O(nlogn) complexity
- We wanted to explore alternative approaches that hover around O(n) complexity
- Karnagel et al. have proposed a hash-based group-by implementation that eliminates multiple passes over input data
- Inspired by their work, we came up with a custom hash-based implementation of a group-by operator

➔ Step 1
  - INIT kernel

➔ Step 2
  - SCAN kernel

➔ Step 3 & Step 4
  - Transfer results to host memory

➔ Step 5
  - FINALIZER kernel

⇩

➔ Setup phase
➔ Reduction phase
➔ Compilation phase
➔ Housekeeping phase

Hash-based Implementation
Setup phase

- Create hash table on host memory
  - Initialize to height of key matrix
  - Use statistical sampling to get a somewhat accurate estimate of unique keys and dynamically update hash table size if estimate was conservative. Dynamic update of hash table size can be costly.
- Transfer hash table, key and value data to device memory
- Run device kernel to get a hash value for each key.

Keys

0	1	0
1	1	1
0	1	0
0	0	0
1	0	1
1	1	1

crc_x64_32_hash kernel →

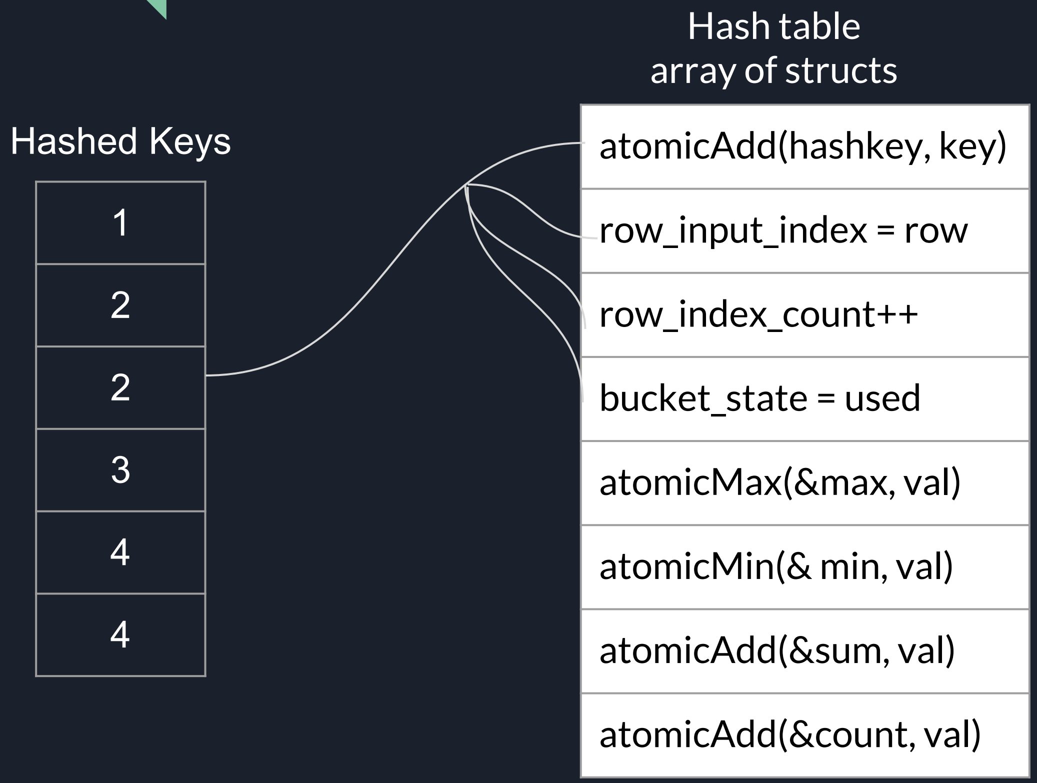
Hashed Keys

1
2
2
3
4
4

Hash table array of structs

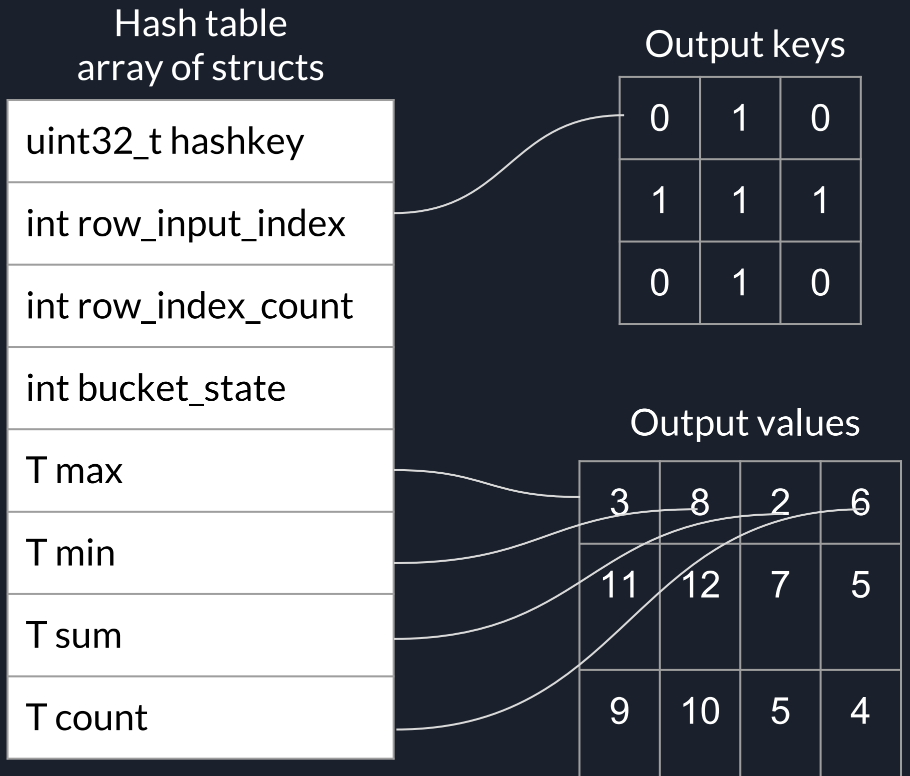
uint32_t hashkey
int row_input_index
int row_index_count
int bucket_state
T max
T min
T sum
T count

Hash-based Implementation
Reduction phase



- ➔ Launch kernel for group-by phase
- ➔ Each thread is mapped to one key
- ➔ Map hashed key to hash bucket
- ➔ Conflict resolution
  - ◆ Open addressing with linear probing
- ➔ If bucket is empty, occupy it, update bucket state to used, increment row count, cache key index, and update aggregator fields
- ➔ If bucket is filled, hashed keys match, and cached keys match, update row count and aggregator fields
- ➔ If bucket is filled and hashed keys do not match or hashed keys match and cached key does not match, move on to next bucket until you find an empty bucket

Hash-based Implementation
Compilation phase & Housekeeping phase



- ➔ Transfer hash table to host memory
- ➔ Scan hash table, map each bucket to one row of output key and update matching row of output values
- ➔ Delete hash table, hashed table
- ➔ Free memory allocated for input data on on host and device memory
- ➔ Calculate and log performance stats

Hash-based Implementation Optimization

- Calculate hash value in-flight
- Overlap memory transfers with computations
    - CUDA streams
- Atomic updates of aggregator fields are serialized. Performance of hash-based implementations are thus bounded by spread of unique keys. Explore an adaptive algorithm that uses distribution of keys to select hash-based or sort/reduction kernels
- Accesses to hash buckets need to be synchronized to avoid race conditions

Verification

Early Verification Process

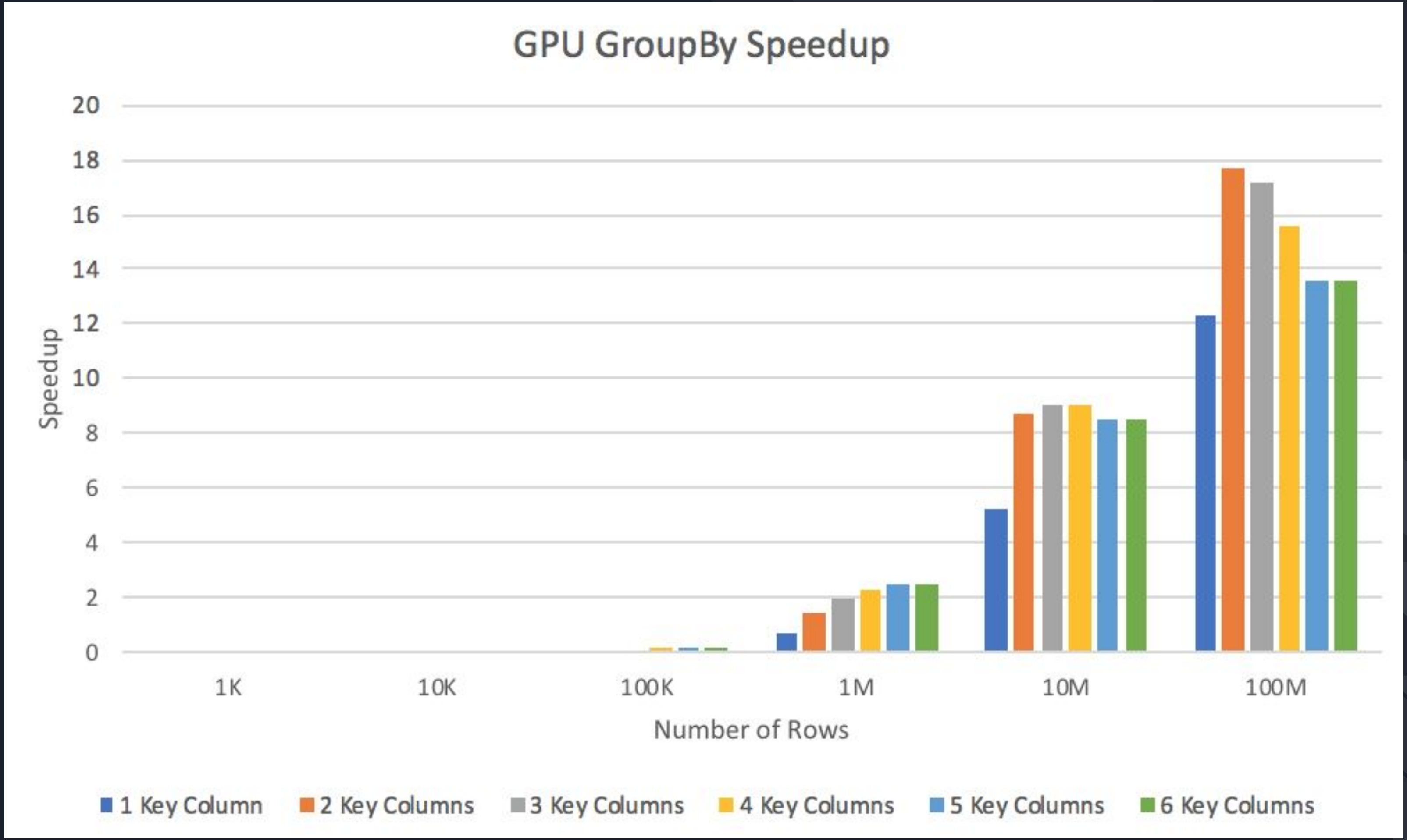
Input Data

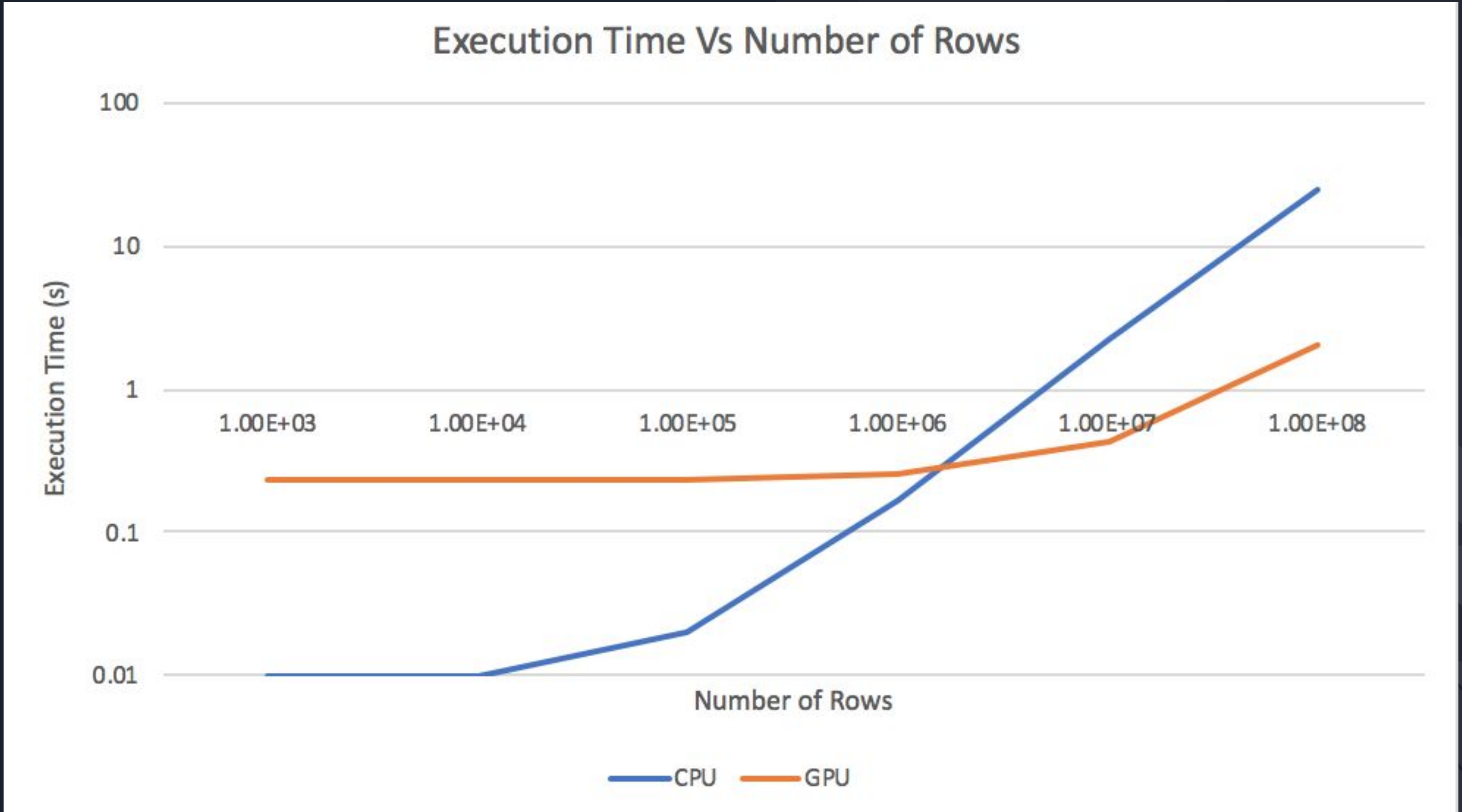
```
numGroups: 94
Printing Data...
{2:1:1}:{59:73:77}
{2:3:2}:{51:18:75}
{0:0:3}:{54:60:4}
{0:2:2}:{75:70:27}
{1:3:1}:{30:98:60}
{0:0:3}:{15:27:67}
{3:0:0}:{63:80:60}
{0:0:2}:{74:73:19}
{0:1:2}:{4:13:60}
{2:3:1}:{96:16:39}
{2:3:3}:{20:70:34}
{3:1:0}:{57:59:37}
{3:3:2}:{93:43:83}
{2:2:0}:{56:78:96}
{1:1:0}:{24:89:40}
```

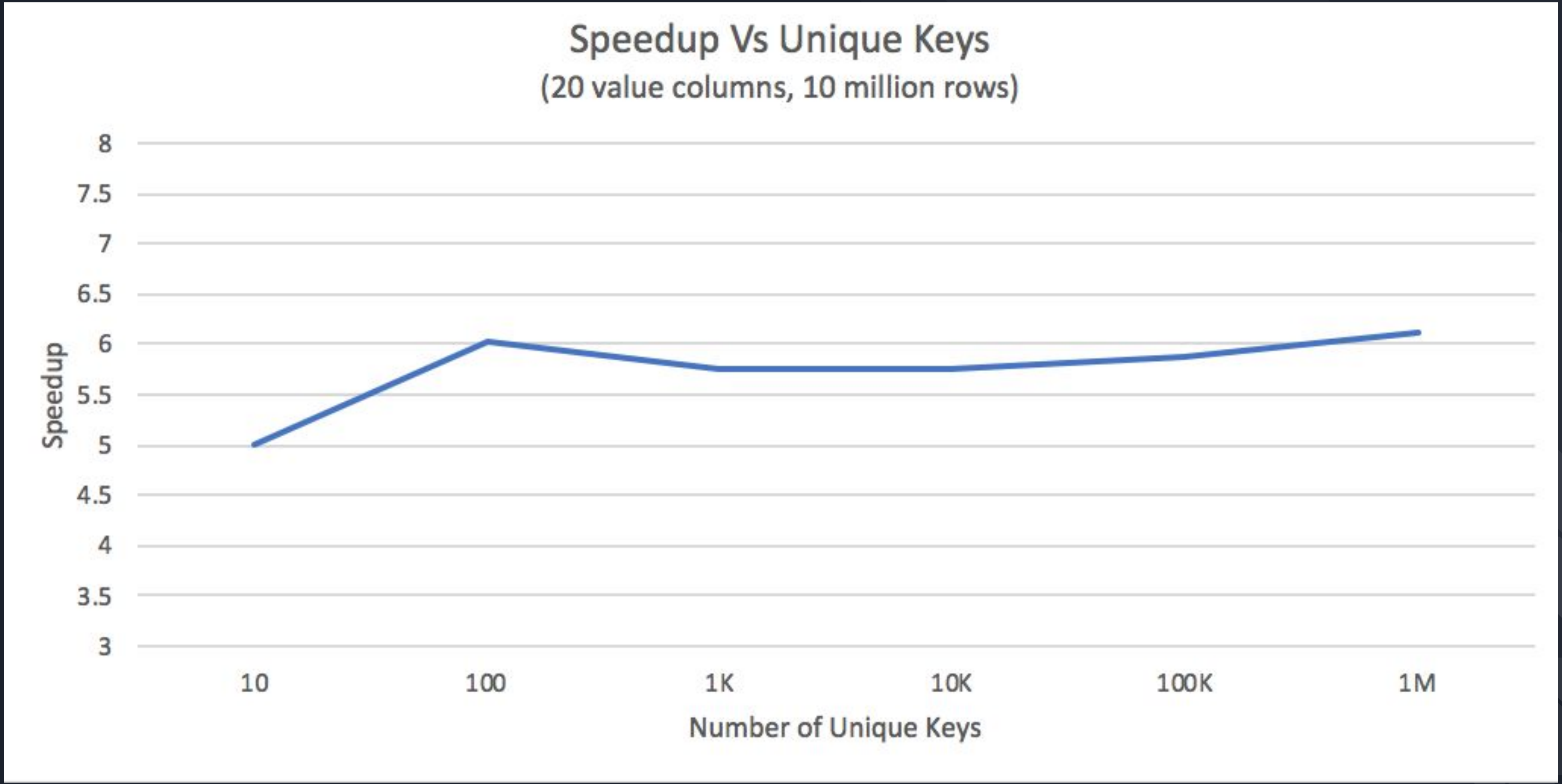
Output Data

```
Printing Results...
{0:0:0}:{32:90:33}
{0:0:1}:{140:265:254}
{0:0:2}:{9:3:93}
{0:0:3}:{155:34:97}
{0:1:0}:{248:279:219}
{0:1:1}:{64:67:39}
{0:1:2}:{13:8:24}
{0:1:3}:{140:95:130}
{0:2:2}:{40:74:139}
{0:3:0}:{167:129:97}
{0:3:2}:{107:22:82}
{0:3:3}:{99:52:101}
{1:0:0}:{37:158:103}
{1:0:1}:{291:176:159}
{1:0:2}:{65:206:193}
{1:0:3}:{190:164:151}
{1:1:0}:{128:46:92}
```

Performance


GPU GroupBy Speedup
20
18
16
14
12
10
8
6
4
2
0
Speedup
1K
10K
100K
1M
10M
100M
Number of Rows
1 Key Column
2 Key Columns
3 Key Columns
4 Key Columns
5 Key Columns
6 Key Columns


Execution Time Vs Number of Rows
100
10
1
0.1
0.01
Execution Time (s)
1.00E+03
1.00E+04
1.00E+05
1.00E+06
1.00E+07
1.00E+08
Number of Rows
CPU
GPU


Speedup Vs Unique Keys
(20 value columns, 10 million rows)
Speedup
8
7.5
7
6.5
6
5.5
5
4.5
4
3.5
3
10
100
1K
10K
100K
1M
Number of Unique Keys

Profiling Result (10M elements)

Total gpu execution time 1.4s

73.69% time of kernel calls are cudaMemcpyHtoD used by thrust (211.16ms)

98.54% time of api calls are cudaMalloc cudaMemcpyAsync and cudaFree (~1s total)

IO operations are the largest bottleneck in the implementation

Using pinned memory could help

Observed results

- The execution time of our group-by implementation on random simulated matrix

  --GPU version performs well within certain range of matrix size

- The performance varied along with different number of key columns

- The running time can have large jumps for certain combination of keys

- The execution time increases huge when groups are large

Conclusions

Take Home Message

- When matrix size is small---- C++ STL (Standard Template Library) is faster than the GPU implementation

- When matrix size is huge (Over one million rows)-- GPU code is up to 20x faster

- Thrust is a great library for accelerating GPU development time (Includes functions like scan and reduce)