Optimizing GPU-Accelerated Group-By and Aggregation
Tomas Karnagel*, René Müller, Guy Lohman
IBM Research–Almaden

Rene Mueller, Research Staff Member
August 31st 2015, ADMS 2015, Kahola Coast, Hawaii

*) Technische Universität Dresden, work was done while intern at IBM Research–Almaden
Introduction

- Group-By operator offloaded to a GPU
- Group-By aggregation is an important operator in OLAP workloads
- Dominant operator in scenarios with denormalized schemas

Challenge

- Parameter selection for non-benchmark workloads
  - Size of hash table (fill factor)
  - Number of threads and thread blocks
  - Hash function and mapping
- Given device properties
  - Device memory
  - Processor count and architecture
- and workload characteristics
  - Cardinality
  - Data types
GPU Offload of Group-By Aggregation

Host

Main Memory

Device Memory

pinned buffers

read

columnar table

Hash Table

PCIe

scan direction
Textbook Implementation of Hash Table

Hash Function

Map

Hash Table

32-bit word

bucket k

key 95120

count 63

sum 5215.11

<empty>

bucket k+1

count 0

sum 0.0

bucket k+2

key 95123

count 23

sum 1123.42

key 95120

count 23

sum 1123.42

<empty>

bucket k+1

count 0

sum 0.0

bucket k+2

key 95123

count 23

sum 1123.42

key 95120

count 63

sum 5215.11

<empty>

bucket k

key 95120

count 63

sum 5215.11

<empty>

bucket k+1

count 0

sum 0.0

key 95123

count 23

sum 1123.42

key 95120

count 63

sum 5215.11

<empty>

bucket k

key 95120

count 63

sum 5215.11

<empty>

bucket k+1

count 0

sum 0.0

key 95123

count 23

sum 1123.42

key 95120

count 63

sum 5215.11

<empty>

bucket k

key 95120

count 63

sum 5215.11

<empty>

bucket k+1

count 0

sum 0.0

key 95123

count 23

sum 1123.42

key 95120

count 63

sum 5215.11

<empty>

bucket k

key 95120

count 63

sum 5215.11

<empty>

bucket k+1

count 0

sum 0.0

key 95123

count 23

sum 1123.42

key 95120

count 63

sum 5215.11

<empty>

bucket k

key 95120

count 63

sum 5215.11

<empty>

bucket k+1

count 0

sum 0.0

key 95123

count 23

sum 1123.42

key 95120

count 63

sum 5215.11

<empty>

bucket k

key 95120

count 63

sum 5215.11

<empty>

bucket k+1

count 0

sum 0.0

key 95123

count 23

sum 1123.42

key 95120

count 63

sum 5215.11

<empty>

bucket k

key 95120

count 63

sum 5215.11

<empty>

bucket k+1

count 0

sum 0.0

key 95123

count 23

sum 1123.42

key 95120

count 63

sum 5215.11

<empty>

bucket k
Experiment Setup

Hardware

- NVIDIA GTX TITAN
  - 14 SMX processors
  - 6 GB device memory
  - PCIe 3.0 (11.8 GB/s)

Implementation

- Hash function: 32-bit FNV-1a
- Hash table fill factor: 50%
- GPU Grid: 14 blocks with 1024 threads

Data

CREATE TABLE xyz (  
  col1 INTEGER NOT NULL,  
  col2 INTEGER NOT NULL,  
  col3 INTEGER NOT NULL,  
  col4 INTEGER NOT NULL  
);

- Columns are random values in $[0, 10^9]$  
- Table has 355 million rows (5.5 GB in total)

Queries

SELECT MOD(col1, ?), COUNT(*) FROM xyz GROUP BY MOD(col1, ?)  
SELECT MOD(col1, ?), SUM(col2) FROM xyz GROUP BY MOD(col1, ?)  
SELECT MOD(col1, ?), SUM(col2+col3) FROM xyz GROUP BY MOD(col1, ?)  
SELECT MOD(col1, ?), SUM(col2+col3+col4) FROM xyz GROUP BY MOD(col1, ?)
Expected Performance Behavior

![Image of a graph showing expected performance behavior. The x-axis represents the number of groups, ranging from 1 to 10^9. The y-axis represents execution time in seconds, ranging from 0 to 1. The graph includes lines for different queries: MOD(col1,?), SUM(col2+col3+col4), MOD(col1,?), SUM(col2+col3), MOD(col1,?), SUM(col2), and MOD(col1,?), COUNT(*).]
Actual Performance Behavior

![Graph showing execution time vs. number of groups for different MOD(SUM(col_n)) queries.](image)

- Red: MOD(col1,?),SUM(col2+col3+col4)
- Orange: MOD(col1,?),SUM(col2+col3)
- Green: MOD(col1,?),SUM(col2)
- Cyan: MOD(col1,?),COUNT(*)
5 Regions with Different Behavior

Region I  Contention on atomics. Updates to same cache line are sent to same ALU in L2.

Region II  Spikes from collisions of the hash mapping and their effects with linear probing.

Region III  Hash table > L2 cache size (1.5 MB).

Region IV  Looks like a cache issue (Hash table > 200 MB). Might be first-level TLB?

Region V  Bad performance beyond 2 GB.
Collisions in Hash Mapping

FNV-1a
- Considered a good hash function in general
- Problems with dense key ranges:
  - High contention $\rightarrow$ up to $1,000\times$ slower
  - Leads to high “collision” count in linear probing

Murmur-3
- No such issues
- More reliable, but worse peak performance (“reliably few” collisions).
- Slightly more compute intensive
Rule-based Parameter Selection

**Hash Function:**
- Always Murmur3

**Hash Table Size (Fill Factor):**
- Use L2 size if fill factor is below 50%.
- 50% fill factor if hash table < 2 GB.
- Use 83% fill factor for the rest.

**GPU Kernel Grid:**
- NSMX: # multiprocessors on GPU (14 for GTX Titan).
- NSMX × 1,024 if hash table < 4× L2.
- 8NSMX × 64 if hash table < 2 GB.
- 8NSMX × 8 for the rest.
Region V: Cliff at 2 GB

The graph shows the relationship between hash table size and runtime. The x-axis represents the number of groups with values ranging from 1 to $10^9$, and the y-axis represents runtime with values ranging from 0.1 to 10 seconds. The graph highlights a 'cliff' at 2 GB, indicating a significant increase in runtime as the hash table size approaches 2 GB.
We’ve seen that before...


©2015 IBM Corporation 31 August 2015
GPUs got better... 

... or didn’t they?

64-bit Compare-and-Swap to random locations in device memory

Bandwidth [GB/s]

Allocated Memory [GB]
GPUs got better... 
... or didn’t they?

64-bit Compare-and-Swap to random locations in device memory

- GTX 580 (Fermi)
- GTX Titan (Kepler)
- K20Xm (Kepler, ECC on)
- K40c (Kepler, ECC on)
- K80 (Kepler, ECC on)

Bandwidth [GB/s] vs Allocated Memory [GB]
GPUs got better...

... or didn’t they?

64-bit Compare-and-Swap to random locations in device memory

Allocated Memory [GB]

Bandwidth [GB/s]

- GTX 580 (Fermi)
- GTX Titan (Kepler)
- K20Xm (Kepler, ECC on)
GPUs got better... 

... or didn’t they?

64-bit Compare-and-Swap to random locations in device memory

- GTX 580 (Fermi)
- GTX Titan (Kepler)
- K20Xm (Kepler, ECC on)
- K40c (Kepler, ECC on)
GPUs got better... 

... or didn’t they?

64-bit Compare-and-Swap to random locations in device memory

- GTX 580 (Fermi)
- GTX Titan (Kepler)
- K20Xm (Kepler, ECC on)
- K40c (Kepler, ECC on)
- K80 (Kepler, ECC on)
Sort-based Group-By Aggregation

Example: SELECT MOD(col0, ?), COUNT(*) FROM xyz GROUP BY MOD(col,?)
Using sort function from CUB library:

\[ \text{cub::DeviceRadixSort::SortKeys} \]
Sort-based Group-By Aggregation

**Example:** SELECT MOD(col0, ?), COUNT(*) FROM xyz GROUP BY MOD(col,?)

Using sort function from CUB library:

1. cub::DeviceRadixSort::SortKeys
2. cub::DeviceReduce::ReduceByKey
**Strided Sort-based Group-By Aggregation**

**Example:** 
```
SELECT MOD(col0, ?), COUNT(*) FROM xyz GROUP BY MOD(col,?)
```

Using sort function from CUB library and merge from Thrust library:

![Diagram showing sort and merge process]

① `cub::DeviceRadixSort::SortKeys`
Strided Sort-based Group-By Aggregation

Example: SELECT MOD(col0, ?), COUNT(*) FROM xyz GROUP BY MOD(col,?)

Using sort function from CUB library and merge from Thrust library:

1. `cub::DeviceRadixSort::SortKeys`
2. `cub::DeviceReduce::ReduceByKey`
Strided Sort-based Group-By Aggregation

Example: SELECT MOD(col0, ?), COUNT(*) FROM xyz GROUP BY MOD(col,?)

Using sort function from CUB library and merge from Thrust library:

1. `cub::DeviceRadixSort::SortKeys` (partial aggregate)
2. `cub::DeviceReduce::ReduceByKey` (new aggregate)
Strided Sort-based Group-By Aggregation

Example: SELECT MOD(col0, ?), COUNT(*) FROM xyz GROUP BY MOD(col,?)

Using sort function from CUB library and merge from Thrust library:

1. cub::DeviceRadixSort::SortKeys
2. cub::DeviceReduce::ReduceByKey
3. thrust::merge_by_key
Hash vs. Sort-based Group-By Aggregation

![Graph showing comparison between Sort-based group-by and Hash-based group-by. The graph includes lines for Sort-based group-by with 8M rows/stride, Sort-based group-by unstrided, and Hash-based group-by (0.5 fill factor) with contention-mitigating changes (private and shmem hash tables).]
Conclusions

- Profile your operator
- Runtime is not always predictable, may be up to $1,000 \times$ slower
- Sort does not necessarily outperform hash grouping
- Out-of-place sort and merge operations lead to many memory moves (in particular for strided execution)