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

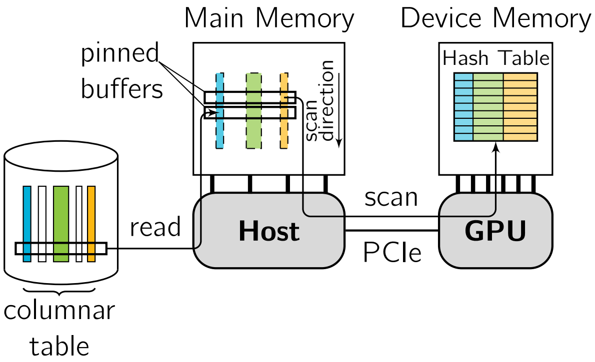
Introduction

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

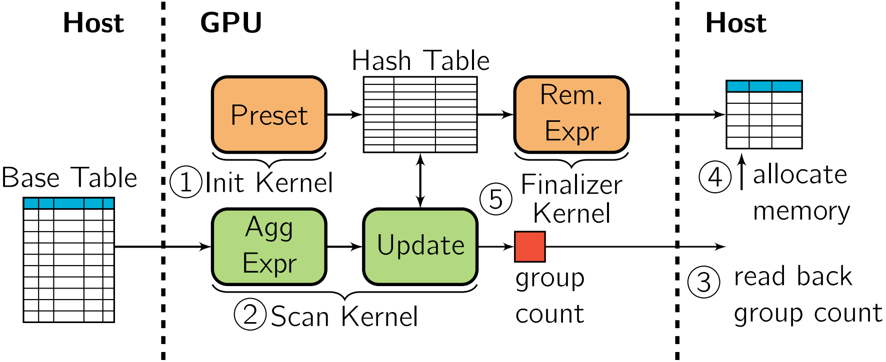
Challenge

- 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

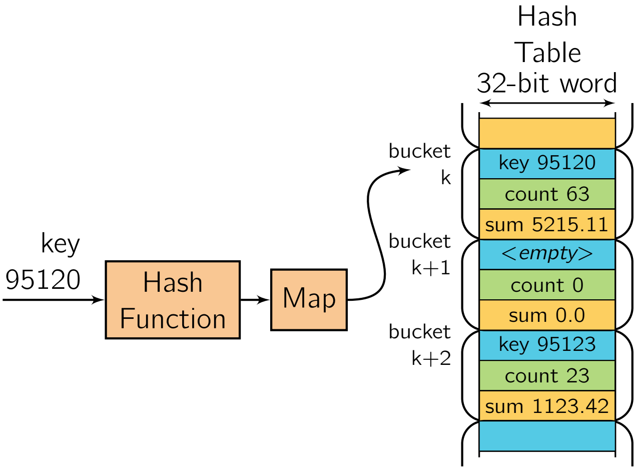
GPU Offload of Group-By Aggregation



Group-By Process Flow



Textbook Implementation of Hash Table



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 bocks with 1024 threads

Data

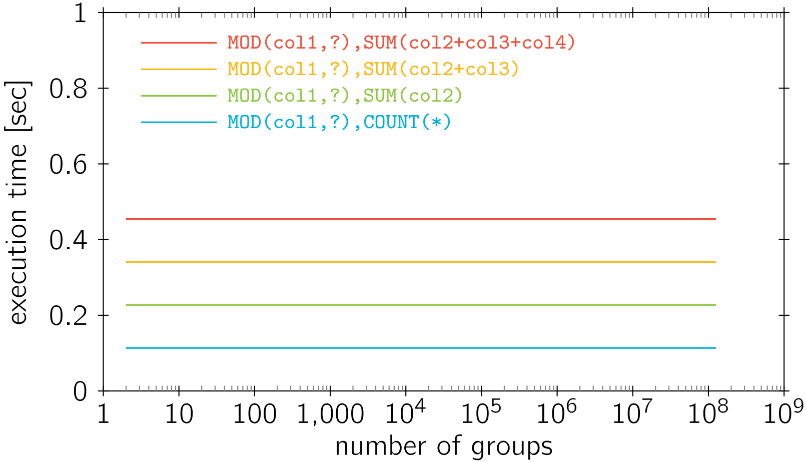
```
CREATE TABLE xyz (
   col1 INTEGER NOT NULL,
   col2 INTEGER NOT NULL,
   col3 INTEGER NOT NULL,
   col4 INTEGER NOT NULL
);
```

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

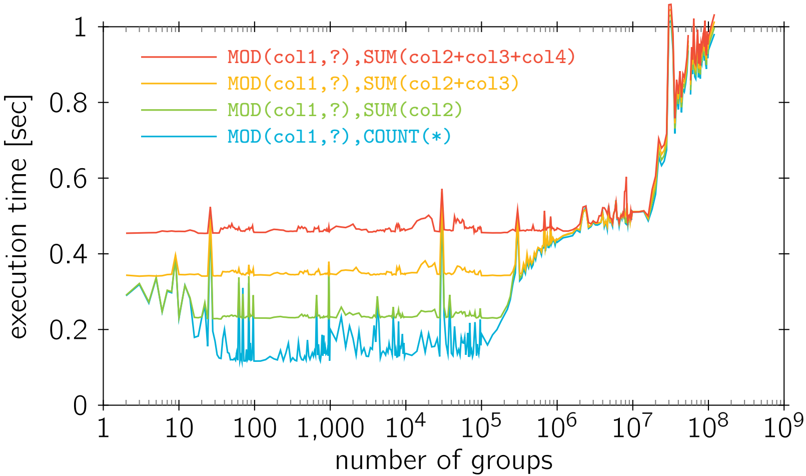
Queries

```
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

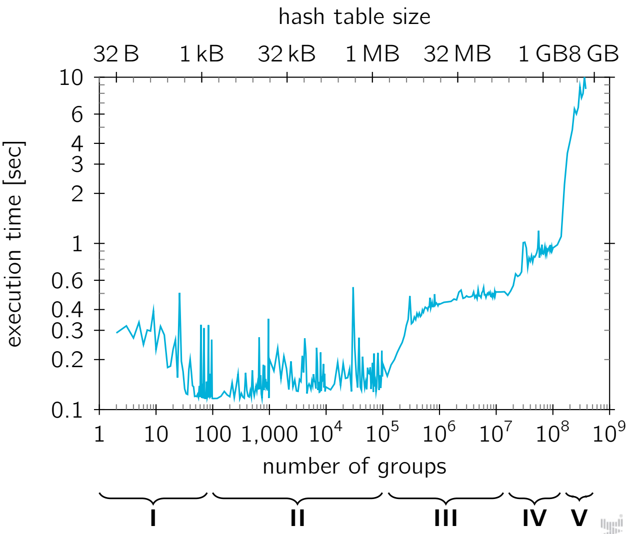


Actual Performance Behavior

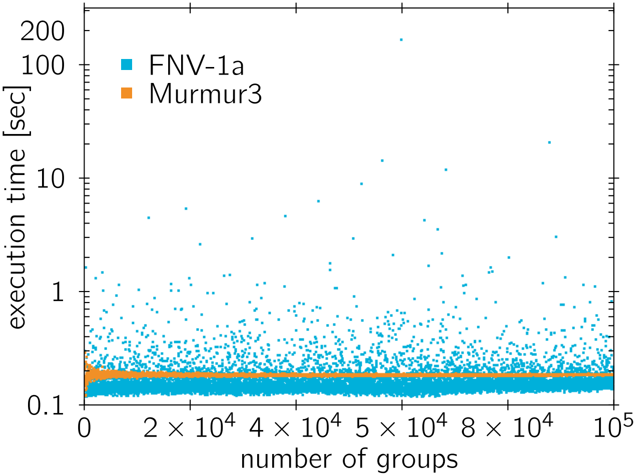


5 Regions with Different Behavior

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 → up to 1,000× 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:

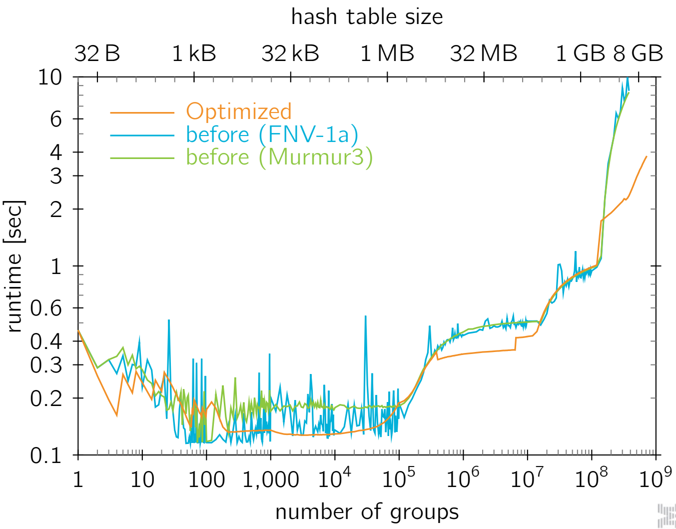
- Always Murmur3

Hash Table Size (Fill Factor):

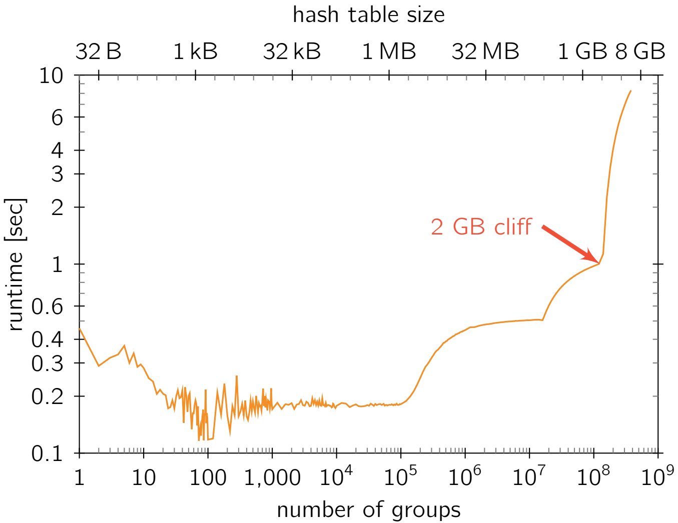
- Use L2 size if fill factor is below 50%.
- 50% fill factor if hash table < 2 GB.
- Use 83% fill factor for the rest.

GPU Kernel Grid:

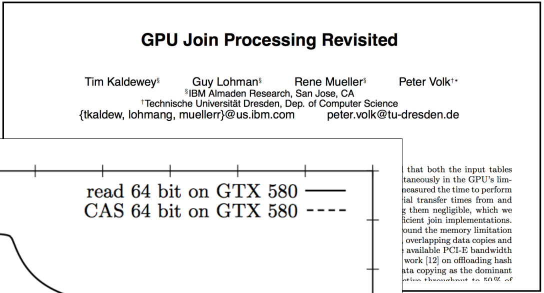
- NSMX: # multiprocessors on GPU (14 for GTX Titan).
- NSMX $\times 1,024$ if hash table $< 4\times$ L2.
- 8NSMX $\times$ 64 if hash table < 2 GB.
- 8NSMX $\times$ 8 for the rest.



Region V: Cliff at 2 GB



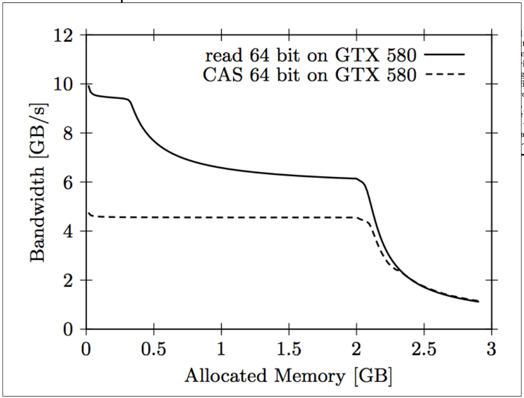
We’ve seen that before…



GPU Join Processing Revisited

Tim Kaldewey§ Guy Lohman§ Rene Mueller§ Peter Volk†*
§IBM Almaden Research, San Jose, CA
†Technische Universität Dresden, Dep. of Computer Science
{tkaldew, lohmang, muellerr}@us.ibm.com peter.volk@tu-dresden.de

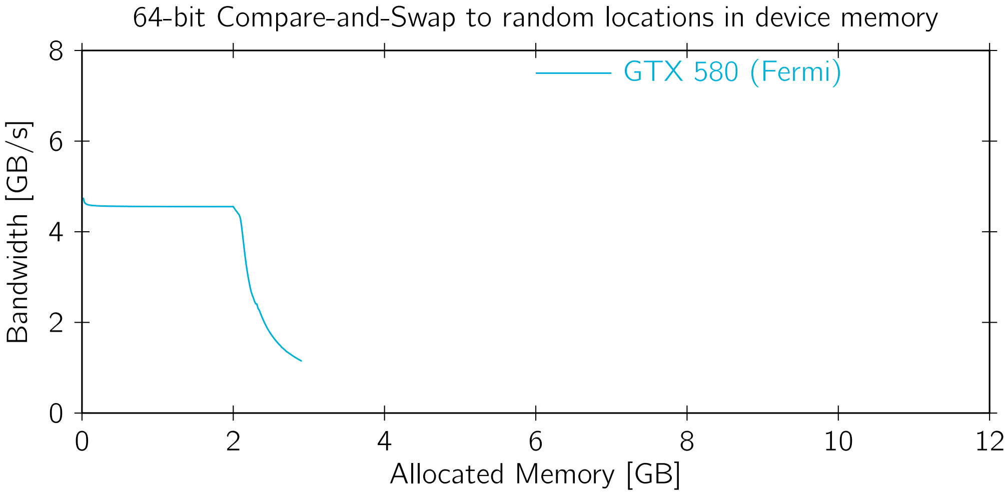
l that both the input tables
taneously in the GPU’s lim-
neasured the time to perform
ial transfer times from and
g them negligible, which we
ficient join implementations.
ound the memory limitation
, overlapping data copies and
e available PCI-E bandwidth
work [12] on offloading hash
ata copying as the dominant



Tim Kaldewey, Rene Mueller, Guy Lohman, Peter Volk, *GPU Hash Join Revisited.* In DaMoN 2012

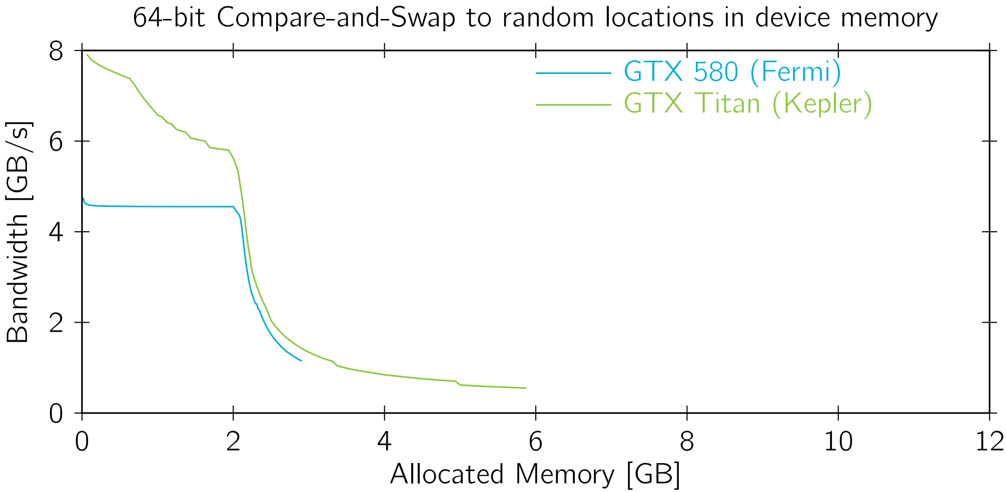
GPUs got better. . .

. . . or didn't they?



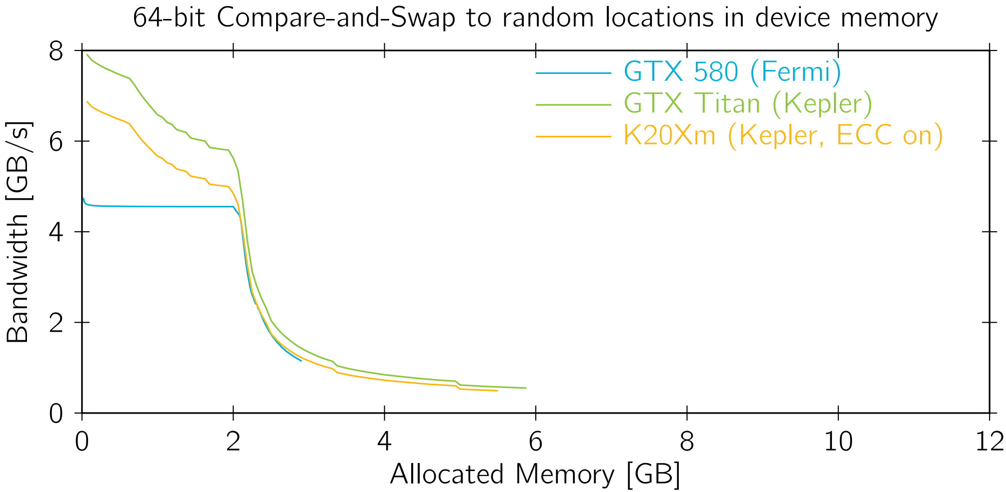
GPUs got better. . .

. . . or didn't they?



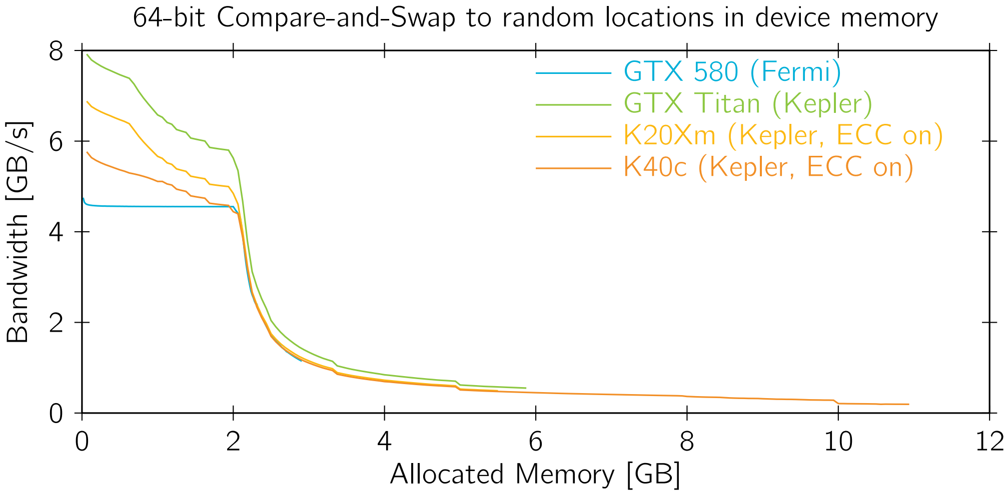
GPUs got better. . .

. . . or didn't they?



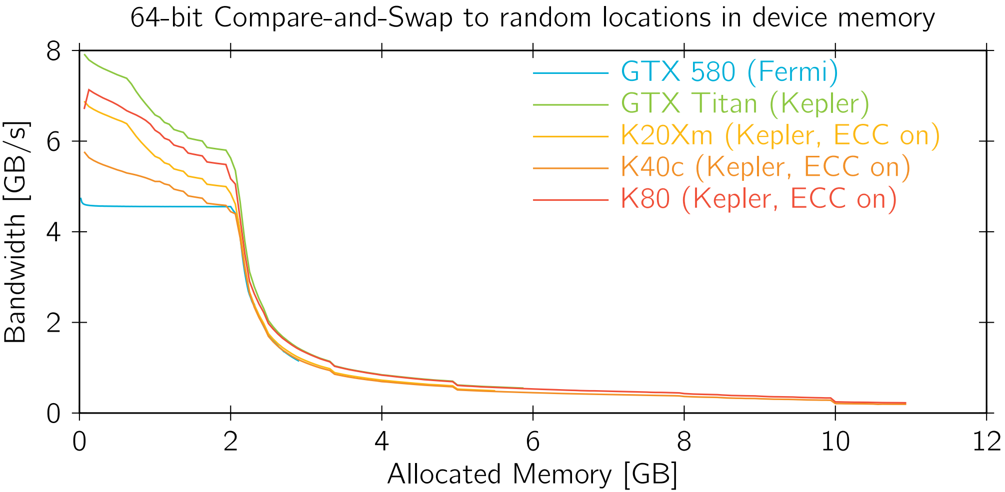
GPUs got better. . .

. . . or didn't they?



GPUs got better. . .

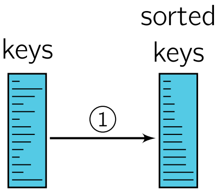
. . . or didn't they?



Sort-based Group-By Aggregation

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

Using sort function from CUB library:

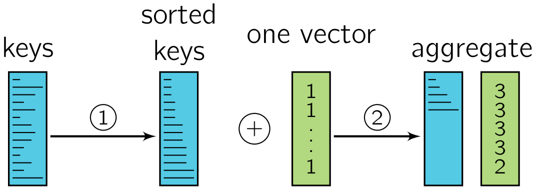


1 `cub::DeviceRadixSort::SortKeys`

Sort-based Group-By Aggregation

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

Using sort function from CUB library:



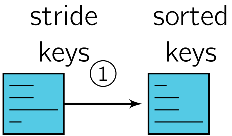
1 `cub::DeviceRadixSort::SortKeys`

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:

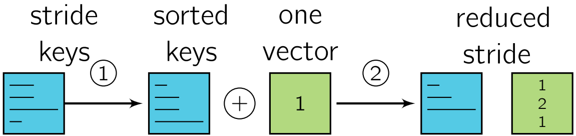


1 `cub::DeviceRadixSort::SortKeys`

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:



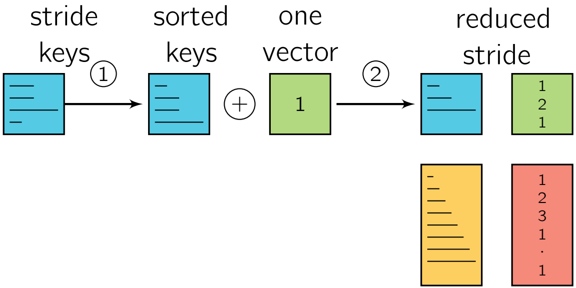
1 `cub::DeviceRadixSort::SortKeys`

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:



partial aggregate

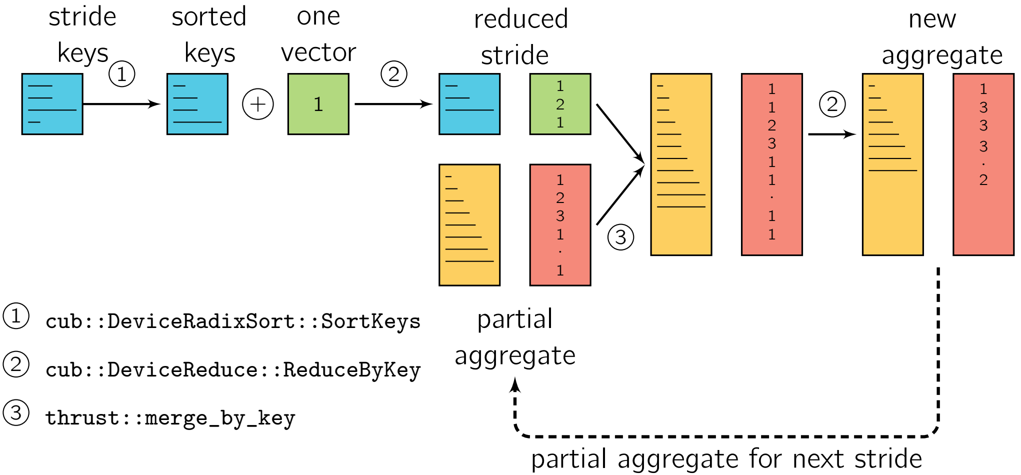
1 `cub::DeviceRadixSort::SortKeys`

2 `cub::DeviceReduce::ReduceByKey`

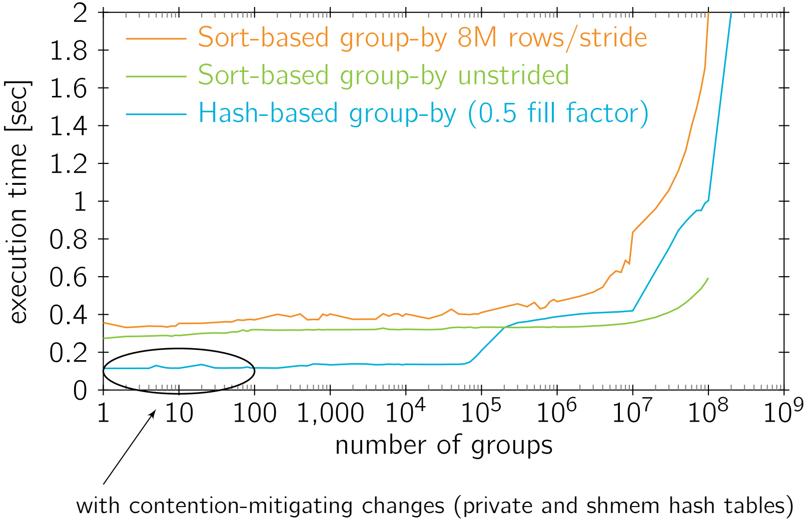
Strided Sort-based Group-By Aggregation

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

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



Hash vs. Sort-based Group-By Aggregation



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)