Optimizing GPU-accelerated Group-By and Aggregation

Tomas Karnagel[*]
Technische Universität Dresden
Dresden, Germany
tomas.karnagel@tu-dresden.de

Rene Mueller Guy M. Lohman
IBM Research–Almaden
San Jose, CA USA
{muellerr, lohmang}@us.ibm.com

ABSTRACT

The massive parallelism and faster random memory access of Graphics Processing Units (GPUs) promise to further accelerate complex analytics operations such as joins and grouping, but also provide additional challenges to optimizing their performance. There are more implementation alternatives to consider on the GPU, such as exploiting different types of memory on the device and the division of work among processor clusters and threads, and additional performance parameters, such as the size of the kernel grid and the trade-off between the number of threads and the resulting share of resources each thread will get.

In this paper, we study in depth offloading to a GPU the grouping and aggregation operator, often the dominant operation in analytics queries after joins. We primarily focus on the design implications of a hash-based implementation, although we also compare it against a sort-based approach. Our study provides (1) a detailed performance analysis of grouping and aggregation on the GPU as the number of groups in the result varies, (2) an analysis of the truncation effects of hash functions commonly used in hash-based grouping, and (3) a simple parametric model for a wide range of workloads with a heuristic optimizer to automatically pick the best implementation and performance parameters at execution time.

1. INTRODUCTION

Despite the recent performance gains that in-memory database systems have brought to the relatively mature technology for processing complex SQL analytics queries, user requirements for ever-faster performance over ever-larger databases has sparked increasing interest in exploiting Graphics Processing Units (GPUs) for further accelerating these queries. GPUs promise massive parallelism and faster memory access, particularly for the random accesses that are so prevalent in joins and grouping operations that dominate the execution time in analytical queries.

As is true for traditional CPU-based database processing, the best implementation and parameter settings for GPU processing depend upon (a) the given SQL query, (b) the data distribution (such as cardinality and skew), and (c) the hardware it is run on.

But building database engines for execution on GPUs presents many additional challenges. Often entirely new approaches and algorithms are necessary to adequately exploit the massive parallelism GPUs offer. There are more implementation alternatives to consider on the GPU, such as exploiting different types of memory on the device (global memory and local scratchpad memory) versus the CPU's memory, and the division of work among processor clusters and threads. To make matters worse, there are also more performance parameters, such as the size of the kernel grid and the trade-off between the number of threads and the resulting share of resources each thread will get. Grouping and aggregation, e.g., for the SQL "GROUP BY" clause, is one of the most time-consuming operators in any database system, especially when performing cubing in On-Line Analytic Processing (OLAP) systems, and dominates the performance time in systems that encourage de-normalized (pre-joined) schemas for performance reasons [9, 27] or that even do not support joins at all, such as "NoSQL" systems, e.g., MongoDB.

In this paper, we study in depth offloading the grouping operator to a GPU. We primarily focus on the design implications of a hash-based implementation, although we also compare it against a sort-based approach in Section 5.2. Our study provides (1) a detailed performance analysis of grouping and aggregation on the GPU as the number of groups in the result varies, (2) an analysis of the truncation effects of hash functions commonly used in hash-based grouping, and (3) a simple parametric model for a wide range of workloads with a heuristic optimizer to automatically pick the best implementation and performance parameters at execution time. We make two simplifying assumptions. First, we assume that the intermediate data structures, such as the hash tables, fit into the device memory of the GPU so that no spilling to main memory or disk occurs. Second, we make the simplifying assumption that there are no queries executing concurrently on the GPU. We do not think that these simplifications are too restrictive in terms of the workloads that can be run. First of all, GPUs today come with significant memory – up to 12 or 24 GB. Second, since OLAP workloads seek to minimize the response time, it makes sense

[*]Work done while author was at IBM Research Almaden.

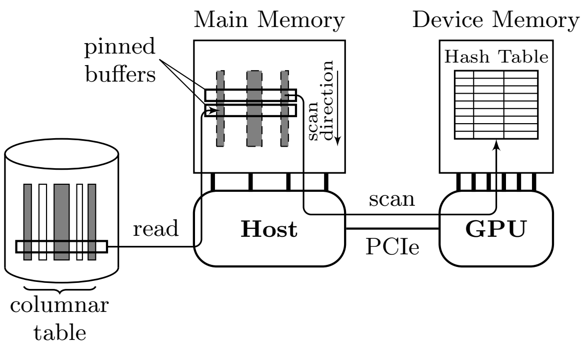


Figure 1: System Setup.

to allocate all available resources in order to finish a given query quickly.

The paper is organized as follows. The next section describes the setup and the hash-based implementation of the grouping operator. Section 3 describes a list of unexpected and surprising observations made from a number of experiments, and presents a deep analysis for the observed effects. Section 4 contains an evaluation of the parameter space and introduces a heuristic approach to select "good" parameters at execution time for different queries and data distributions. We provide a comparison of different placements for the hash table as well as a comparison of hash grouping with a sort-based approach in Section 5. The last two sections discuss related work and the paper's conclusions.

2. GPU GROUPING AGGREGATION

This section describes the system set-up and our implementation of the grouping hash tables.

2.1 System Setup

Figure 1 shows the setup used throughout the paper. Unlike other earlier work, we *do not assume that the entire data set fits into the device memory of the GPU* [12]. Instead, the only restrictive assumption we make is that the intermediate data structures, i.e., the hash table for the hash-based grouping operator, fits into the device memory. The base table is stored on the host system. More precisely, it is loaded from stable storage such as disks, or in our case, SSDs, into main memory. Both the on-disk and in-memory representations are in columnar form.

Data transfers from the SSDs into the host memory and into the GPU have to be overlapped for maximum efficiency. This is achieved by first splitting the rows to be scanned into *strides* with a fixed number of rows. A stride of an `INTEGER` column usually amounts to $\approx$ 32 MB. In a first stage, the columns of the next stride are read from the SDDs into pinned buffers in the host memory, one buffer per column. The current stride that was read in the previous step is available in a second set of column buffers. Since all column buffers are backed by pinned memory, they are directly accessible from the GPU in the second stage via *zero-copy access* (available via *Universal Virtual Addressing (UVA)* in NVIDIA GPUs [22]) as shown in Figure 1. The load accesses that the GPU performs to fetch the column values during the table scan are aligned and *coalesced*, i.e., neighboring threads in the GPU load adjacent words in memory. This permits a high ingest bandwidth. We achieved $> 90\,\%$ of the maximum theoretical bandwidth of the 16-lane PCI Express 3.0 (PCIe) link at the transaction layer.

Alternatively, we could have used a three-stage approach: (1) SSD to host memory, (2) explicit host to device memory copy, and (3) column scan out of device memory. While this approach is recommended for non-coalesced memory access patterns (but non-coalesced accesses to host memory are prohibitively slow!), we found the bandwidth of the zero-copy access sufficient for our application setup. Furthermore, it reduces the pipeline from three to two stages and does not require column buffers in the device memory.

2.2 Group-By Implementation

In this section, we first provide a high-level overview of the implementation of the group-by operator. The implementation is illustrated using the following simple group-by query over a table of orders in a store:

```
Query A:
  SELECT zip, 1-AVG((chargedamount-tax)/ordertotal)
    FROM orders
GROUP BY zip
```

This query calculates the average discount offered to customers grouped by zip code. Here, we assume that `zip` is an `INTEGER` while the other three columns are of type `REAL`. The query consists of the following elements:

$$\begin{aligned}
\text{expr} &:= (\texttt{chargedamount} - \texttt{tax})/\texttt{ordertotal}\\
\text{aggregate} &:= (c, s)\\
\text{initial aggregate} &:= (0, 0)\\
\text{merge}\big((c, s), v\big) &:= (c + 1, s + v)\\
\text{final}\big((c, s)\big) &:= 1 - s/c
\end{aligned}$$

expr is the argument expression in the `AVG` aggregate. The state for this aggregate is a tuple (c, s) consisting of a count c of values and sum s of values. The initial state of the aggregate is $(0, 0)$. The merge function adds a new value v to the aggregate. The value v is the result of evaluating expr. At the end of the aggregation, the finalizer function calculates the result `1-AVG(..)` specified in the SELECT clause of the query from the final aggregation state (c, s). We limit our discussion in this paper to algebraic aggregates with constant aggregate state, and leave the DISTINCT aggregate case for future work.

The implementation of the group-by operator and the data flow within the operator are illustrated in Figure 2. The operator consists of five steps and involves three different GPU kernels. The INIT kernel in step ① in the figure initializes the hash table, i.e., it sets all keys equal to EMPTY and initializes the aggregation state. For the query shown above, the aggregation state is set to $(0, 0)$. The INIT kernel only accesses the device memory and so completes very quickly.

The majority of the time is spent in the SCAN kernel during step ②, in which the group-by aggregation is performed. In the SCAN kernel, the GPU threads first load the column values, each one from a separate row, and compute the argument expressions of the aggregate functions. For the example query above, the expression expr is evaluated to a value v. Then the threads update the aggregate state that corresponds to the `zip` value they loaded earlier, as determined by the merge$\big((c, s), v\big)$ function. If a thread cannot find an entry for its zip code in the hash table, it inserts a

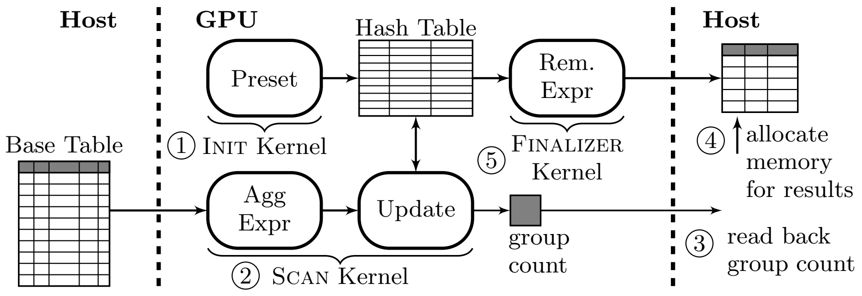


Figure 2: Group-By data flow.

new value and merges the value v into the initialized aggregation state. During this insert, the thread also increments a global group counter (shown in Figure 2) via one global `atomicInc` operation. Although this global atomic counter represents a potential contention point for the threads, an atomic operation is only performed when a key is encountered for the first time and inserted into the hash table. No atomic writes to this global counter are performed during the subsequent updates for this key.

After the scan over the input data is complete and the Scan kernel has terminated, the group counter is read back into the host (Step ③ in Figure 2). This result count is used in Step ④ to allocate a sufficiently large result buffer in the host memory that can hold the results of the group-by operation. Afterwards, the Finalizer kernel is launched in step ⑤. It scans the hash table and the remaining expressions on all non-empty buckets. For example Query A, shown above, this remaining expression is $1 - s/c$. The results are written back into host memory, again via zero-copy access.

Generation of Group-By Kernels. The three kernels for the grouping operator are query dependent. The Init kernel depends on grouping keys and the aggregate functions, the Scan kernel on the aggregate function and expressions, and the Finalizer kernel on the aggregate function and ultimately on the SELECT clause[1]. The kernels need to be generated ahead of time as "prepared SQL statements" or as dynamic statements at execution time. The discussion of the kernel generation is beyond the scope of the paper. Here, we assume that the three kernels from Figure 2 are generated by some means and only discuss the performance impact from an algorithmic and parametric point of view. There are various ways by which these kernels can be generated at execution time. One possibility is by using the OpenCL `clCreateProgramWithSource(..)` API call [19]. Another approach could be based on NVRTC [23], the C++ run-time compilation of CUDA 7.0, or directly on NVIDIA's PTX driver API [24].

Hash Table: Open-Addressing with Linear Probing. We begin our discussion with the simplest possible hash table implementation, shown in Figure 3. The figure depicts the textbook implementation of a hash table with open addressing. The hash table is an array of hash buckets that

[1]If the query also contains a HAVING clause, the corresponding predicate needs to be evaluated in the Finalizer kernel.

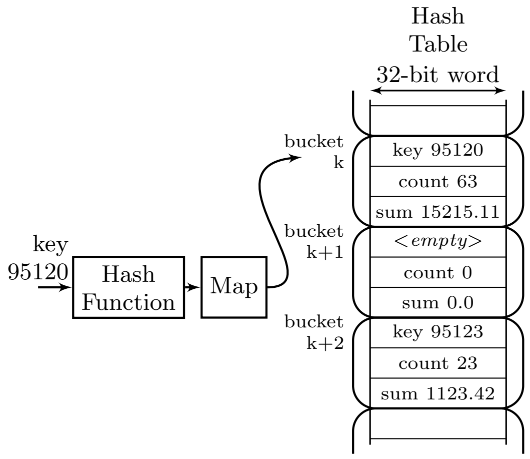


Figure 3: Hash table implementations with a globally shared hash table in device memory.

contain three 32-bit elements for the example query: the key and the two aggregates, `count` and `sum`. During an update for a given key, the hash value of the key is computed and mapped (by remainder division or by selecting a subset of the bits from the hash value) to one of the hash buckets k. In this simple scheme, collisions are resolved with linear probing, i.e., if the bucket is already occupied by another key, the next free buckets $k+1$, $k+2$, etc., are searched until either that key or an empty bucket is found.

When an empty bucket is found, it will be initialized by (1) placing the key into the table, (2) incrementing the count from the initial value (zero), and adding the argument value to the sum field. Since the hash table is globally shared by all threads running on the different processors, the update of that bucket's fields have to be performed atomically, as illustrated by this pseudo-code:

```
uint32_t* hashtable = ...
update(bucket k, uint32_t key, float arg)
{
BEGIN TRANSACTION
  uint32_t x = hashtable[3*k]; // current key
  if (x == EMPTY_MARKER)
    hashtable[3*k]    = key;   // insert key
  else if (x != key) {
    // bucket k not empty and keys do not match
    ABORT TRANSACTION and try bucket k+1
  }
  // update aggregate state
  hashtable[3*k+1] += 1;         // update count
  hashtable[3*k+2] += arg;       // update sum
  COMMIT
END TRANSACTION
}
```

Unfortunately, the GPU does not provide hardware support for transactional memory, and latching the entire hash bucket for the duration of every attempted update would severely limit parallelism and hence performance. Instead, we can decompose the single atomic operation into atomic updates of each of the bucket's three fields. For the latter, modern GPUs do provide some hardware support. Hence, we can re-write the update code for our simple group-by as follows:

```
uint32_t* hashtable = ...
update(bucket k, uint32_t key, float arg)
{
  uint32_t x = hashtable[3*k]; // current key
  if (x == EMPTY) {
    // try inserting key
    old = atomicCAS(&hashtable[3*k], key, EMPTY);
    if ((old != EMPTY) || (old != key))
      return failure; // lost race for bucket k
  } else if (x != key) {
      return failure; // key collision on bucket k
  }
  // update aggregate state (count, sum)
  atomicAdd(&hashtable[3*k+1], 1);
  atomicAdd(&hashtable[3*k+2], arg);
  return success;
}
```

The update code first loads the key of the bucket and checks whether it is equal to the EMPTY marker. If so, it tries to overwrite the value with the key value. The operation is implemented using an atomic Compare-And-Swap (CAS). The operation can have three possible outcomes: (1) old == EMPTY indicates that no other thread has updated this memory location since the load into x. (2) The value returned by CAS is equal to the value of the key that the thread is trying to write. This means that another thread successfully inserted the same key. (3) The returned value corresponds to a different key. In this case, the calling thread lost the race for the insert. In the other two cases, however, it can proceed with updating the aggregation state. This is implemented again via atomic operations. The order of these updates does not matter.

While this code correctly implements example Query A, there are several questions and concerns whether this approach would also work for a broader range of group-by queries. We discuss these questions below:

What is the value of the EMPTY marker? — Clearly, the empty marker corresponds to one value out of the key domain, such that there cannot be any key with the same value as the EMPTY marker. Typically, it is assumed that a marker value is chosen that does not occur in the key space. This is also an option here, and can easily be guaranteed if the value for key is an encoded value instead of the actual key itself (e.g., in systems like DB2 BLU [26]). The encoded space is thus just reduced by the one value used for the EMPTY marker. If operating on encoded keys is not possible and a special value cannot be set aside for EMPTY, the hash mapping can be modified such that a key that has the same value as the EMPTY marker is mapped to a special bucket "outside" of the hash table.

Table 1: Hardware support [24] for aggregation functions and data types (aggregate implementation: HW atomic = in hardware, CAS = in software using Compare-and-Swap, ext32 = sign-extend to 32-bit and HW atomic).

	MAX	MIN	SUM
SMALLINT	ext32	ext32	ext32
INTEGER	HW atomic	HW atomic	HW atomic
BIGINT	HW atomic	HW atomic	HW atomic
FLOAT	CAS	CAS	HW atomic
DOUBLE	CAS	CAS	CAS
DECIMAL	CAS	CAS	CAS

Which are the supported types and aggregate functions? — Our prototype system supports the data types and aggregation functions listed in Table 1. Not all atomic functions are available for all data types on NVIDIA GPUs. The missing functions have to be implemented "by hand" using CAS operations in a loop. SMALLINT types must be extended to the next larger data type that is supported by the hardware, which unfortunately increases the size of the hash table.

How is the hash table sized? — We assume that we obtain an estimate of the group-by cardinality ahead of time such that we can size the hash table accordingly. Modern commercial systems employ quite sophisticated query optimizers that can provide these estimates. Unfortunately, correlation between columns and expressions on columns make it very difficult for the optimizer to provide accurate estimates. For DB2, we observe that correlated columns always lead to an over-estimate of the cardinality, in the worst case the product of the cardinality of the individual columns. Although over-estimation avoids an overflow of the hash table, it could create hash tables that are too big to fit within the GPU's device memory. We are treating this case in future work.

How are overflows of the hash table handled? — In open addressing, the hash table is full when a thread cycles around the entire hash table and arrives back at the starting bucket into which the key was initially hashed, without ever finding that key or an EMPTY bucket. The thread then sets a special global trap flag and terminates. To prevent other threads from unnecessarily repeating this expensive traversal, they should periodically poll the global trap flag. We tried to use the special kernel trap instruction to intentionally abort the entire kernel, but we found that this damages the GPU context, preventing us from re-executing the kernel once we allocated a larger hash table. If the hash table overflows in our prototype, we double the size of the hash table and simply rerun the query.

What is the support for wider and combined keys? — Current NVIDIA GPUs support CAS for 32- and 64-bit words only. Multi-column keys can be implemented as long as they fit the largest data type (64-bit). Wider keys that take up more than one 64-bit word have to be treated differently than shown above. The relaxed memory ordering model of modern NVIDIA [22, §B.5] and AMD [3] GPUs further exacerbate the problem. Our current solution for these cases is to revert to latching the entire bucket.

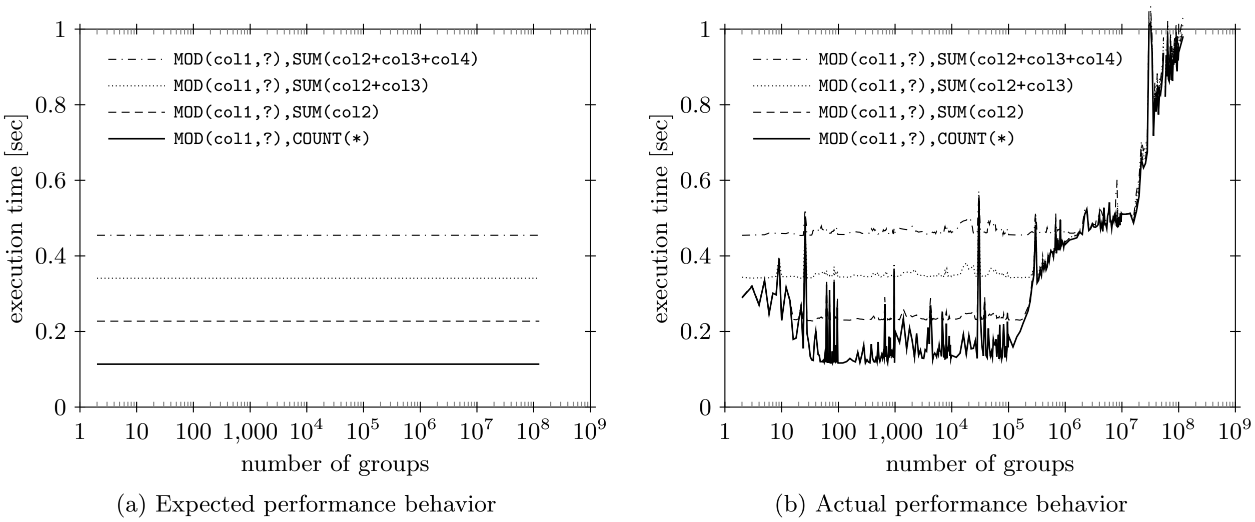


Figure 4: Performance of initial group-by implementation, measured as execution time of a table scan over 335 million rows, on a GTX Titan, hash table with 50 % fill factor, and 14×1024 threads.

3. OBSERVATIONS

We tested the simple hash-based group-by described in Section 2 on an NVIDIA GTX Titan GPU using 14 thread blocks (the number of SMX processors on the GPU) and 1024 threads/block. The allocated hash tables were twice the size of the group count, providing the desired fill factor of 50 %. We use the 32-bit FNV-1a hash [10] as a hash function. We map the hash value into the bucket via modulo division. The group-by was performed on a table with 335 million rows consisting of four INTEGER columns:

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

The values in `col1` are independent, uniformly distributed and random in the range of $[0, 10^9)$. This distribution results in scattered memory accesses with minimal caching opportunities. By contrast, non-randomized or skewed distributions generally have more localized memory accesses and make better use of the GPU cache and, thus, lead to better performance. Our data distribution can thereby be considered as the worst-case scenario. To analyze the performance as a function of the number of groups, we use the following query that uses the `MOD` parameter to limit the number of groups output:

```
Query B:
  SELECT MOD(col1, ?), COUNT(*)
    FROM atable
GROUP BY MOD(col1, ?);
```

We study the impact of the data transfers on the overall performance by extending the SELECT clause of Query B with expressions that reference more columns. Ideally, we would expect the execution time for these variants of Query B to be determined by how many columns are referenced, as shown in Figure 4(a). The GPU can read from the host memory via zero-copy access at a peak speed of 11.8 GB/s in our setup. Hence, we would expect an execution time that is inversely proportional to the total size of the accessed columns, i.e., the four equi-spaced lines shown in Figure 4(a). However, the actual performance we observed is very different, as shown in Figure 4(b). We observe that: (1) the performance does not remain constant as we increase the number of groups by adjusting the `MOD` parameter in the query. The resulting curves have the shape of a bath tub. (2) The runtime has a high variability when only one column is accessed. The execution time can have large jumps for certain group combinations. This is not a measurement artifact – the numbers are in fact repeatable. (3) Accessing more than one column appears to hide some of this variability. (4) The execution time increases for few groups and (5) sharply increases for > 100 million groups.

Focusing on the 1-column query for the moment, we can distinguish five differing regions in which different phenomena appear to dominate. Figure 5 shows the ranges of these five regions enumerated as I, ..., V. Unfortunately, we do not have insight information from NVIDIA on their hardware that would provide clear explanations for the different behaviors. In order to provide an explanation, we combine the publicly available hardware descriptions with extensive profiling via performance counters.

Region I: Contention. The group count, and thus the hash table, is very small. For a globally shared hash table, this leads to contention that limits the overall performance. Though NVIDIA provides very little detail about the implementation of atomic operations in Kepler, we believe that the following is a plausible way by which atomic operations are implemented efficiently in L2. Beginning with the Kepler GPU architecture, atomic operations on device memory are performed in ALUs in the L2 cache [25], which is shared by all processors on the GPU. Atomic operations, e.g., atomic additions used for updating a SUM aggregate, are treated as store instructions. The operations are then routed to the ALUs based on their target memory address. A FIFO buffer

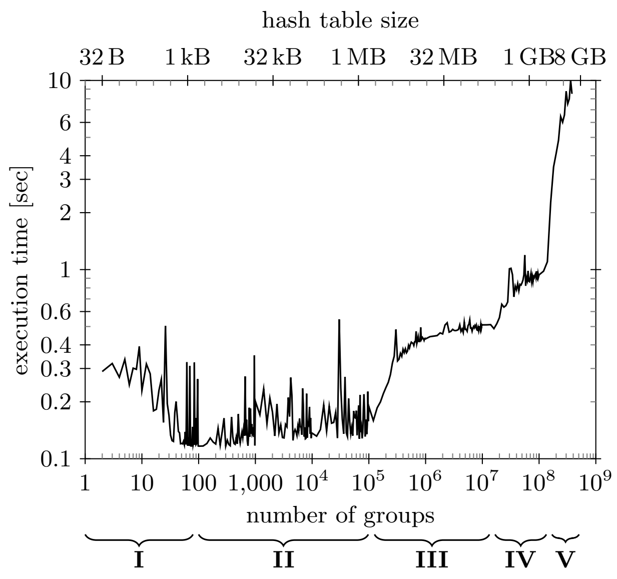


Figure 5: Regions with different behavior for Query B: `SELECT MOD(col1,?),COUNT(*) FROM atable GROUP BY MOD(col1,?)` **(GTX Titan, 50 % fill factor,** 14×1024 **threads)**

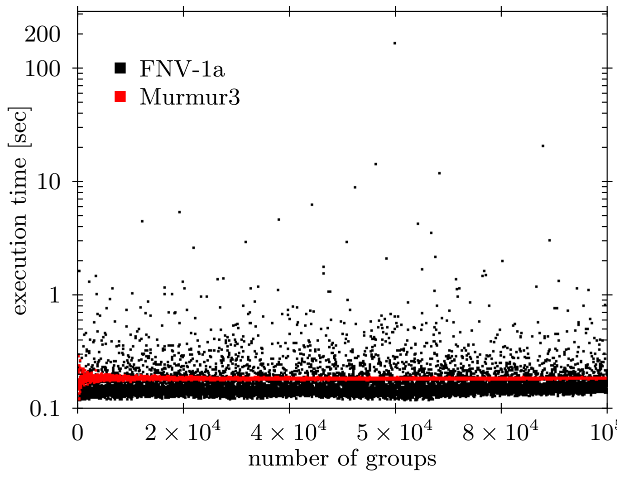


Figure 6: Scatter plot comparing execution time using FNV-1a and Murmur3 hashing for Query B: `SELECT MOD(col1,?),COUNT(*) FROM atable GROUP BY MOD(col1,?)` **(GTX Titan, 50 % fill factor,** 14×1024 **threads).**

in front of each ALU enqueues the operations before they are processed and the memory locations in L2 are updated.

The downside of this approach is that updates to the same hash bucket, or to buckets that are located on the same cache line, are sent to the same ALU. For small numbers of groups or a highly skewed distribution of the grouping keys, this introduces a work imbalance on the ALUs and, thus, contention. NVIDIA states that the hardware is able to handle between 1–8 atomic operations/cycle/processor in an ideal scenario. In our example, the size of the hash table bucket is eight bytes, so there are 16 hash buckets per 128-byte cache line. We clearly need more than 16 groups to occupy more than one cache line and, presumably, ALU that handles the atomic. From Figure 5, the upper end of this region I is about 100 elements. We discuss ways to avoid this contention later in Section 5.1.

Region II: L2 & Spiky Performance. The execution time of our group-by implementation is very spiky in Region II. The execution times in this region correspond to the access time for the input columns, given the PCIe bandwidth of ≈ 11.8 GB/s. If it were not for the spikes, the performance in the second region would be entirely dominated by the available PCIe bandwidth. But what causes these spikes? In region II, the hash sizes are large enough that contention, which dominated performance in region I, no longer has a significant impact. The region contains hash tables that are ≤ 1.5 MB, i.e., that still fit into the L2 cache on the GTX Titan. Puzzled by the spikes that can be up to $4\times$ higher than the base performance limited by the PCIe bandwidth, we repeated the experiment multiple times, assuming we would observe a non-deterministic artifact. However, the spikes persisted, and at the same locations! After digging deeper, we found out that the spikes were due to collisions of the hash mapping, i.e., the FNV-1a hash function followed by the modulo division mapping, and a serialization effect due to false sharing of hash buckets located in the same cache line. FNV-1a is known as a good hash function. We confirmed that by studying the distribution of 32-bit hash values. The problem was introduced when mapping the 32-bit hash values into hash buckets. This mapping did not sufficiently preserve the uniformity of the computed hash values, especially for small and compact key domains ($\text{key}_{\text{max}} - \text{key}_{\text{min}} \ll 2^{20}$). By contrast, we did not see performance issues for values that were chosen randomly from large domains ($> 2^{20}$). The modulo operation in the grouping expression of our benchmark query effectively produces such a compact the key domain. Compact key domains are also quite common in real workloads, e.g., for sequential order keys. Curious how other commonly-used hash functions would perform, we repeated our experiment with different hash functions. Here we only show the results of the best- and the worst-performing hash functions. Figure 6 depicts a scatter plot of the different execution times using the FNV-1a and the Murmur3 [4] hash functions. We show a fine resolution in the number of groups to illustrate the variance. Indeed, FNV-1a has a variance of more than three orders of magnitude! Murmur3, however, has a lower variance but, in general, a higher minimal runtime. We conclude that the collisions introduced by the hash-value-to-bucket mapping is more uniformly distributed for Murmur3 and with fewer outliers, resulting in a more predictable performance than with FNV-1a. Having learned this lesson, we will use Murmur3 as the hash function in the remainder of this paper.

Region III: Hash Table > L2. In region III we observe the expected increase in execution time when the hash table grows beyond the L2 cache capacity. The execution time gradually increases and remains constant when almost every access results in a miss.

Region IV. We are not sure what happens in region IV. Region IV starts at a hash table size of about 200 MB. The behavior looks like missing the next level of cache; however, L2 is already the last level cache on the GPU. The translation look-aside buffer (TLB) is also organized as a cache, and we believe these effects can be influenced by the first level TLB cache. However, lacking insider information, we cannot be entirely sure.

Region V: TLB issues. The jump at the beginning of Region V corresponds to the 2 GB TLB limitation that was pointed out earlier by Kaldewey et al. [17].

4. CONFIGURATION PARAMETERS

In the previous section, we analyzed the performance as a function of the number of groups in our group-by implementation. In this section, we evaluate the impact of the chosen parameters on the execution, without changing the structure or algorithm of our group-by implementation. We identified two parameters that can be influential: the hash table size and the CUDA grid parameters.

Hash Table Size and Fill Factor. In our previous experiments, we chose a fill factor of 50 %, which results in a hash table size that has twice the number of entries of the expected number of groups. This fill factor is usually a good tradeoff between size and insert/lookup performance. A larger hash table would produce fewer conflicts on insert, and therefore a shorter lookup path, but it also consumes more space in device memory and cache. To evaluate the impact of the hash table size, we ran our initial experiment again using the Murmur3 hash function with varying fill factors. We chose three adaptive hash table sizes using the fill factors of 25 % (4x #groups), 50 % (2x #groups), and 83 % (1.2x #groups), and one fixed-size hash table, which is set to the size of the L2 cache (1.5 MB, about 200,000 entries in our test scenario). The results are shown in Figure 7. Region I is still dominated by the atomic throughput for all our test cases. Region II shows the actual impact of the fill factor on execution performance. Here all hash tables fit in the L2 cache, and the performance is bound by bucket contention and linear probing. We can see that the fixed hash table performs best because it has the smallest fill factor in most cases. The fill factor of 25 % has a slightly worse performance. The higher fill factors have an even worse performance due to much more contention in the hash table. In the passage from Region II to III, we can see that the fixed L2 version is performing worse because the fill factor and the contentions are growing until the hash table is too small for the actual data. In this part, the fill factor of 50 % performs best because it has less bucket contention than the 83 % fill factor and consumes less memory than the 25 % fill factor, where, in this case, efficient caching is not possible. The advantage is lost when the hash table no longer fits in cache, in the last part of Regions III and IV. In Region V, the 2 GB TLB problem hits every version at a different number of groups, because the sizes of the hash tables are different. There, the 83 % fill factor version performs best. These results convinced us that we do need to adapt the hash table size and fill factor in order to get the best possible performance for any number of groups.

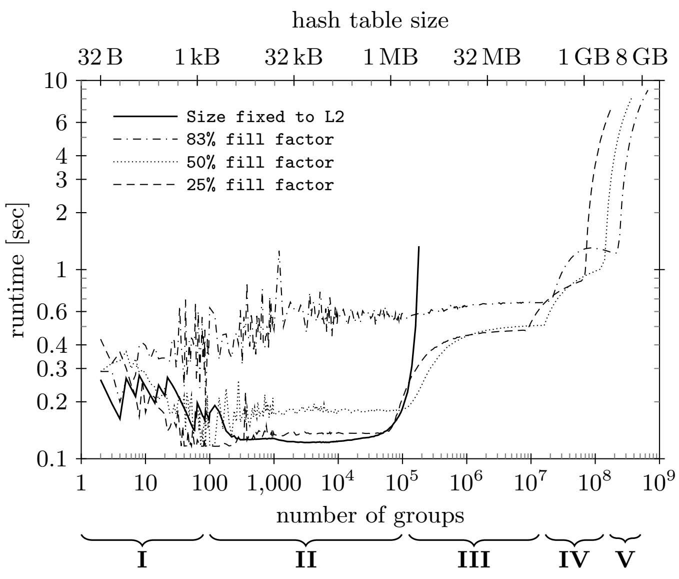


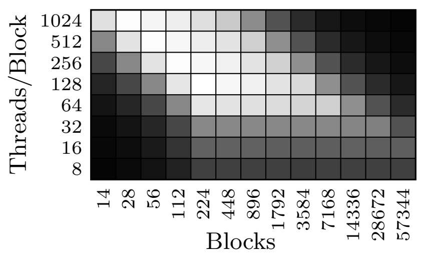
Figure 7: Different hash table fill factors and their behavior. (GTX Titan, Murmur3, 14×1024 threads)

CUDA Grid Parameters. The second parameter we can change is the CUDA grid configuration. The grid configuration specifies a *blocks* $\times$ *threads* combination:

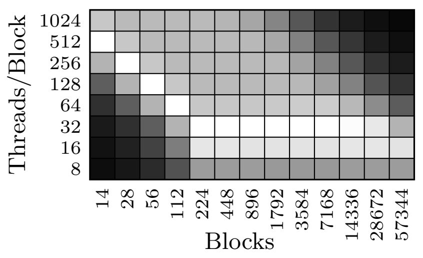
$$\text{total \#threads} = blocks * threads$$

Threads combined in a block can share resources, and they are guaranteed to be executed on the same multiprocessor. However, multiple blocks can be executed on one processor at the same time to hide memory latency. One block can contain between 1 and 1024 threads [21]. NVIDIA GPUs execute up to 32 threads simultaneously, so a block should contain a minimum of 32 threads to allow the hardware to be utilized. Usually, shared memory usage limits the number of threads per block, but since we do not use shared memory in our implementation, we are free to use any configuration of blocks and threads. In our implementation, the total work of an input data stride is divided automatically between the total number of threads. To evaluate the different grid parameters, we tested the number of threads in power-of-two steps from $2^3 = 8$ up to $2^{10} = 1024$ threads and the number of blocks in multiples of the number of multiprocessors (in our case 14). We tested the grid configurations extensively and found three different behaviors. Representatives of these behaviors are shown in Figure 8. In detail, we show our implementation with 1,000 groups (behavior similar to Regions I and II), 1 million groups (similar to Regions III and IV), and 200 million groups (Region V). For these tests we used a fill factor of 50 %.

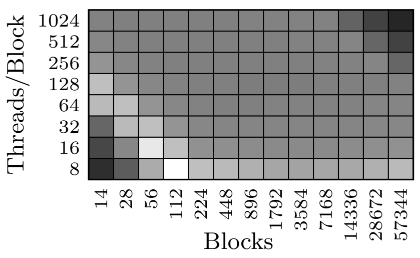
Figure 8(a) shows that the ideal grid configuration is greater than or equal to 14,336 total threads, but less than or equal to 229,376 total threads, with a minimum of 64 threads per block. If the total number of threads is lower than the lower threshold, there are not enough threads to saturate the PCIe bus with memory requests and to hide memory latency. If there are more threads than the upper threshold, the scheduling overhead seems more significant to the execution time.



(a) Hash table for 1,000 groups.



(b) Hash table for 1 million groups.



(c) Hash table for 200 million groups.

Figure 8: Evaluation of grid parameters for three different number of groups (GTX Titan, Murmur3, fill factor 50 %). The performance is encoded from black (worse) to white (best).

Figure 8(b) shows a different behavior. Here, it seems beneficial to have 7,168 or more total threads if we only use 32 threads per block. The number of threads can be explained with the caching behavior. For 1 million groups, the hash table does not fit in the L2 cache, so each thread has to access global memory, unless the data is in the cache by chance. Usually, a thread has to access only one cache line during a lookup or insert, since the cache-line size is 128 bytes, accommodating 16 hash table buckets in our test scenario. Assuming that the hash bucket for a given key is on average in the middle of a cache line, a second cache line needs to be loaded only if the linear probing goes beyond 8 steps. With 14,336 threads running simultaneously (original configuration), 1.75 MB will be loaded for the first cache line. Since this does not fit into the L2 cache (1.5 MB), the threads are evicting each others' cache lines, so that linear probing, even within the same cache line, could result in multiple loads from global memory. However, with 7,168 threads, only 0.875 MB are loaded, fitting perfectly in the L2 cache, which results in "undisturbed" linear probing for each thread. Having even fewer threads would benefit from the same effect, but it also under-utilizes the system, causing worse performance. More threads are possible if there are only 32 threads per block, because our test GPU can only schedule and execute 16 blocks simultaneously on one multiprocessor [21], generating 32 threads $*$ 16 blocks $*$ 14 multiprocessors = 7,168 threads running simultaneously.

Figure 8(c) shows the grid configurations for the 2 GB problem. We assumed in the previous section that there is high pressure on the TLB cache, which results in bad performance. Surprisingly, the performance improves when we reduce the number of threads and therefore take away some of the pressure on the TLB. This works even below the 32 threads per block that are needed to utilize the multiprocessors.

From the results shown in Figure 8, we chose the best-performing configurations for the different behaviors, and evaluated the configurations in our initial test scenario with growing numbers of groups. Specifically, we chose the following blocks $\times$ threads configurations: 14×1024, 112×64, and 112×8. The results are shown in Figure 9. As expected, the single configurations are optimal in the regions for which they were selected. Please note the bad performance of 112×64 and 112×8 when the execution is not bound by the L2 cache or the 2 GB problem. Also interesting is the speedup of these two compared to the initial setting (14×1024) in their specific regions. In Region III,

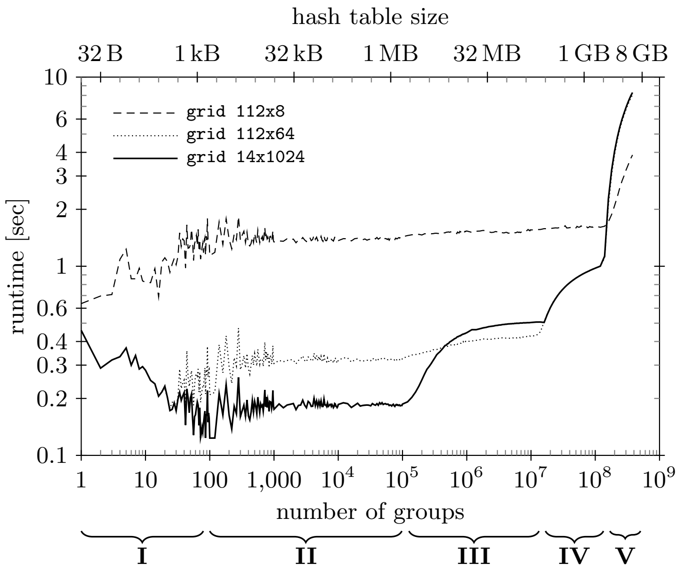


Figure 9: Best performing grid configurations. (GTX Titan, Murmur3, fill factor 50 %)

112×64 achieves a 1.2× speedup, and in Region V, 112×8 achieves a 2.2× speedup compared to the initial setting. We did not find better suited grid configurations for Region IV.

Simple Rule-based Optimizer. Finally, we can take the parameter insights for the hash table fill factors and the CUDA grid configurations and build a simple model that switches between the settings, depending on the estimated number of groups. The model's decisions can be done in two steps by (1) setting the optimal hash table and (2) setting the optimal grid configuration. Both decisions can be described by a set of rules, where the first applicable rule is chosen. Rules for the hash table size:

1. Use L2 as the hash table size if the fill factor is below 50 %.
2. Use 50 % fill factor if the hash table is smaller than 2 GB.
3. Use 83 % fill factor for the rest.

When the hash table size has been decided, the grid can be easily determined by these rules:

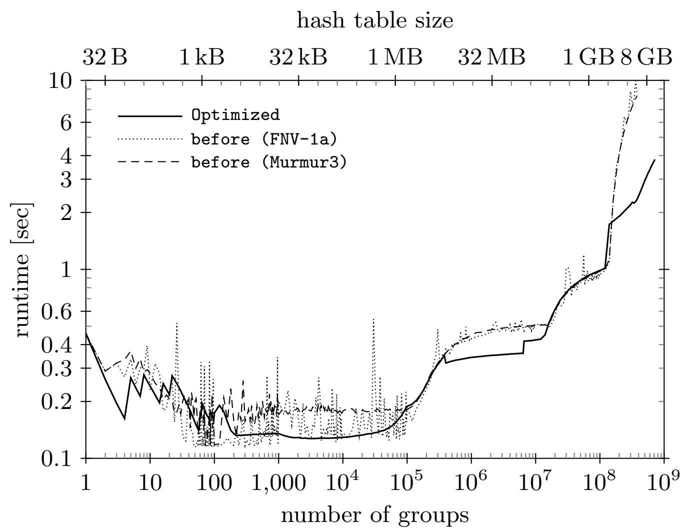


Figure 10: Optimized execution applying the proposed rules. (GTX Titan, Murmur3)

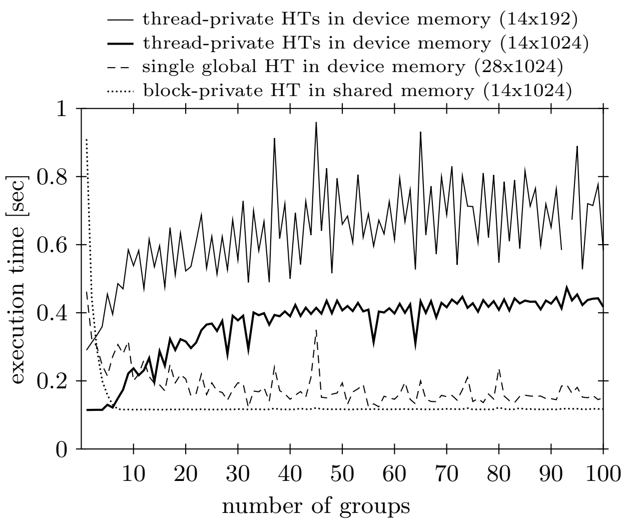


Figure 11: Performance of alternative implementations for group-by operators for low cardinality. (GTX Titan, Murmur3, 50 % fill factor)

1. Use 14×1024 if the hash table is smaller than 4 times the L2.
2. Use 112×64 if the hash table is smaller than 2 GB.
3. Use 112×8 for the rest.

The final evaluation of our test scenario is shown in Figure 10. Murmur3 has a more reliable performance than FNV-1a, but is mostly slower than FNV-1a. Our changes to the hash table size made the Murmur3 performance comparable to the best-case FNV-1a performance, while also being reliable. We achieve a speedup of up to 4.4× compared to the original version using Murmur3 or FNV-1a, and we are able to work with up to 1.8× more groups because of the smaller fill rate for large numbers of groups (> 2 GB).

Besides the promising speedups, we could not fix the problems caused by the atomic contentions or the significant 2 GB problem by only changing parameters. To make further improvements, we would need to change our grouping operator significantly at the algorithmic and implementation levels, which we consider next.

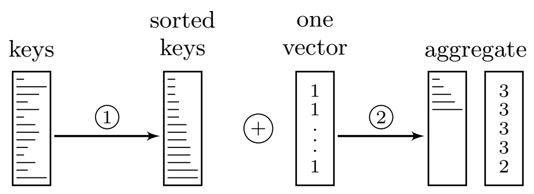
5. ALGORITHMIC APPROACHES

In the last section, we discussed the parameter selection for an implementation having a single hash table. Now, we describe two different algorithmic changes. First, we study how to reduce contention in case the group-by has few groups. Then, we briefly discuss sort-based aggregation and compare it with the hash-based approach described earlier.

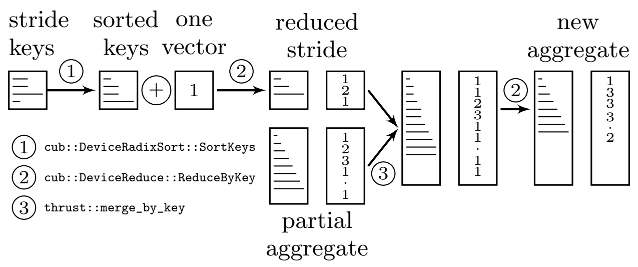
5.1 Hash table placement

In order to reduce contention on the atomic when the group-by contains few groups, we introduce multiple hash tables into which the groups are inserted during the scan. Before calling the Finalizer kernel, the partial aggregates from the different hash tables are aggregated and inserted into a final global hash table. We show two alternative placement strategies. In the first strategy, we place the hash tables into processor-specific shared memories. Each of these hash tables is only shared by the threads running in a particular thread block on a processor. The advantage is twofold: By distributing the inserts/updates over multiple hash tables, we reduce the contention on the atomic updates. Additionally, shared memories are co-located with the processors and are faster than the shared L2 cache. In the second placement strategy, we provide a private hash table to each worker thread, such that no sharing occurs. Since there are many threads on each processor, the hash table cannot be placed into shared memory for capacity reasons. Instead, we place the hash tables in device memory.

Figure 11 compares the performance on a GTX Titan of these new distributed hash table strategies against the single globally shared hash table from the previous sections. We can see that for block-private hash tables and after 6 groups, the execution time quickly drops to a constant 0.11 sec, i.e., the limit imposed by the available PCIe bandwidth. We test two configurations for thread-private hash tables. First, we use a minimal configuration with 192 threads/block. This choice makes sure that we occupy all compute cores of a processor on the GTX Titan. In this setting, tables with up to 73 buckets fit into the L2 cache of the GPU. The second configuration uses 1024 threads/block, which corresponds to the maximum number of threads per block that is supported by the GPU. Now, only hash tables with up to 13 buckets fit into L2. Note, however, that the configuration with 1024 threads/block is faster than 192 threads/block over the entire measured range. Thus, cache misses do not play a role here. The larger thread count is able to hide the miss latencies and is faster due to the higher degree of parallelism. We can now extend our rule for region I: if we anticipate a cardinality of ≤ 6, we switch to thread-private hash tables with a 14x1024 thread configuration. Otherwise, we use the approach in which the hash tables are placed in shared memory and shared by the threads of a block. Beyond region I, we switch to the global hash table approach as described in Section 2, because the hash table does not fit into shared memory anymore.



(a) Unstrided Group-By



(b) Strided Group-By

Figure 12: Unstrided and strided sort-based grouping aggregation

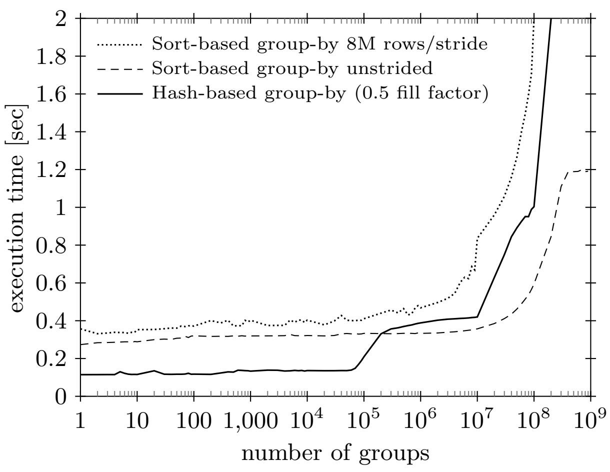


Figure 13: Hash- vs. sort-based group-by for Query B: `SELECT MOD(col1,?),COUNT(*) FROM atable GROUP BY MOD(col1,?)` **(GTX Titan, 50 % fill factor for hash based group-by)**

5.2 Sort vs. Hash-based Group-By

Instead of hashing, grouping can also be implemented using sorting [18]. In this section, we compare two sort-based grouping implementations with the hashed grouping described earlier. We use the CUB [20] (version 1.8.2) and Thrust [16] (version 1.4.1) libraries that provide efficient implementations of the level sorting and reduction operations we need for the grouping operator.

The first implementation is non-strided (see Figure 12(a)), that is, the entire key column from the input table is copied into the GPU. The key column is then sorted using the device-level radix sort from CUB, denoted as ①. The sorted keys, and thus the runs of the same key, are reduced with a sum operator on a vector consisting of only 1 value. This segmented reduction [5] produces the count aggregate and is computed using `DeviceReduce::ReduceByKey` from CUB, ②. Due to the out-of-place implementation of the sort and the `ReduceByKey` operation, the available GPU memory is essentially cut in half.

The implementation that works on strides is shown in Figure 12(b). As in the hash-based approach described earlier, the table is processed in strides of rows. For every stride of rows, the partial aggregate is computed via radix sort (① in Figure 12(b)) and then by sum reduction with a vector of 1s whose length is equal to the size of the stride ②. Next, the aggregate of this stride of rows needs to be merged into the partial aggregate computed so far. This is accomplished by merging the two pre-sorted sequences in parallel (③ in Figure 12(b)), using `thrust::merge_by_key`. We use the Thrust library for this operation, as there is no such function available in CUB at the time of writing. Finally, the merged sequences are reduced ③ to obtain the new aggregate.

Figure 13 compares the two sort-based grouping algorithms with the previous hash-based approaches. The hashed grouping contains the optimization for small group cardinality described in Section 5.1. It can be seen that hash-based grouping outperforms the two sort-based alternatives for up to 200,000 groups. After that, the unstrided sort approach outperforms hashing. For less than 200,000 groups, the hashing algorithm is faster, with its single pass over the input. Sort-based grouping has higher initial costs and requires multiple radix passes. For more than 100,000 groups, the hash table grows beyond the L2 size, the scan speed drops, and gets overtaken by sort.

An interesting observation is that strided sort is always worse than hashing. Strided sort is necessary to overlap the I/O from nonvolatile storage with the GPU processing in a real system. The CUB and Thrust code assume that the inputs are located in device memory. However, a complete column of a table may not fit in memory. Hence, the table is scanned in strides of rows, as previously described for the hashed grouping. The aggregate for each stride must be merged with the partial aggregate accumulated thus far. This merging happens out-of-place, so for the merging aggregate of each stride, the old partial aggregate needs to be moved in memory. This is true even when replacing the merging strategy depicted in Figure 12(b) with a merge tree consisting of pair-wise merges. In summary, we found that sort-based hashing only provides an advantage if it can be done in an unstrided fashion.

6. RELATED WORK

Hash tables are widely used for grouping, aggregation, and joining in GPU-accelerated database systems, as well as in other fields like computer graphics. Alcantara [1, 2] analyzes different GPU hashing approaches, including cuckoo hashing, chaining, and open addressing. In Alcatara's work, several probing functions and fill rates were evaluated. However, the problems arising with different hash table sizes and grid configurations were not discussed.

Yuan et al. [28] use cuckoo hashing for joins and aggregation. For the latter, they build the hash table on the group-by keys first and aggregate columns in a second step by scanning the hash values. Cuckoo hashing uses multiple hash functions and actively reorganizes entries in the hash table to reduce the number of lookups. Similar to cuckoo hashing is Robin Hood hashing by Celis [8], in which hash table entries are reorganized depending on the number of lookups needed to find an entry. García et al. [11] further refine this approach for the GPU. The result is a small number of lookups for all entries and an only slightly larger number of lookups in the worst case. This is especially useful for GPUs, since no thread will stall others by having to do many more lookups. However, we chose to keep entries that were inserted in place without reorganizing. Since we are PCIe-bandwidth bound in most cases, we can afford to do more lookups while keeping the algorithm and data structure simple.

He et. al [13, 14] use hashing in hash indexes optimized with a multi-pass scatter and gather operation. They use chaining for the hash table entries. However, the problem with chaining is the unknown number of entries per hash table bucket, which cannot be handled efficiently by the GPU, where no dynamic memory allocation is supported between multiple kernel executions.

In this paper, we use open addressing for the hash table performing our group-by aggregation. Kaldewey et al. [17] use open addressing for the hash table implementing their hash join on a GPU, with a constant hash table fill rate of 50 %. Bordawekar [6] proposes an open addressing approach with multi-level bounded linear probing, where the hash table has multiple levels to reduce the number of lookups during linear probing. Again, we are mainly PCIe-bandwidth bound, so we can afford these lookups in preference to a simple implementation.

The GPU-based database systems from Breß et al. [7] and Heimel et al. [15] both implement sort-based grouping aggregation.

7. CONCLUSIONS

In this paper, we have evaluated different implementations of the grouping operator for GPUs, comparing hash-based grouping with sort-based grouping, and assessing the performance impact of several key parameters of those algorithms as the number of groups varies over a wide range. All implementations maintain the temporary data structures for the aggregation in the GPU's device memory, while we stream the input columns from the host memory into the GPU. Unlike previous work, we include these data transfers in our end-to-end analysis. Although our implementation of the grouping algorithms was relatively straightforward, the performance effects seen for different numbers of groups were rather unexpected and surprising. We divided these effects into five regions, and explained the anomalous behavior for each region in detail. Through this analysis, we found better configurations and designed simple rules to adapt to the workload. For example, we showed experimentally that hash-based grouping dominated sort-based grouping, except in the rare case when the input data was small enough that it could be processed in one stride. We determined that a private hash table per thread worked better for a small number (< 6) of groups, a local hash table per thread block worked well when it fits into shared memory, and a global hash table was better for hash table sizes beyond the GPU's shared memory. For the latter, we determined experimentally how to adjust the hash table size and fill factor dynamically to trade off bucket contention with cache utilization. Lastly, we found the best combinations of CUDA grid parameters to optimize throughput.

This in-depth analysis provided insights that enabled us to improve both the performance and robustness of our initial implementation significantly by: (1) identifying flaws in and changing the hash function we used, and (2) optimizing the performance for a given number of expected groups via simple rules that adaptively adjust combinations of: (a) the grouping algorithm, (b) whether to localize hash tables or not, (c) the hash table's fill factor, and (d) the CUDA grid parameters. Our analysis demonstrates empirically how GPU performance can benefit from analyzing the performance in detail for different algorithms, data structures, and parameter settings.

Finally, we think that the observations we made and the optimizations we performed are also applicable to other database operators and algorithms that we may run advantageously on a GPU. For example, hash joins also employ large hash tables to store join payloads, so should benefit from our analysis and optimization rules. Other operators with random memory access patterns, such as index accesses or binary search, should also benefit from the observations described in this paper. Therefore, our optimizations are likely to improve an entire family of algorithms being executed on GPUs.

8. REFERENCES

[1] D. Alcantara. *Efficient Hash Tables on the GPU.* PhD thesis, University of California, Davis, 2011.

[2] D. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Building an efficient hash table on the GPU. *GPU Computing Gems*, 2, Aug. 2011.

[3] AMD. *White Paper: AMD Graphics Cores Next (GCN) Architecture*, June 2012.

[4] A. Appleby. Murmurhash project, 2008. http://code.google.com/p/smhasher/.

[5] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11):1526–1538, Nov. 1989.

[6] R. Bordawekar. Evaluation of parallel hashing techniques. GPU Technology Conference, 2014.

[7] S. Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[8] P. Celis. *Robin Hood Hashing.* PhD thesis, University of Waterloo, Waterloo, Canada, Canada, 1986.

[9] J. P. Costa, J. Cecílio, M. Pedro, and P. Furtado. Overcoming the scalability limitations of parallel star schema data warehouses. In *Algorithms and Architectures for Parallel Processing*, volume 7439 of *Lecture Notes in Computer Science*, pages 473–486. Springer, 2012.

[10] G. Fowler, L. C. Noll, and P. Vo. FNV hash. http://www.isthe.com/chongo/tech/comp/fnv/.

[11] I. García, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel hashing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 161:1–161:8, New York, NY, USA, 2011. ACM.

[12] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proceedings of the IEEE*

International Symposium on Performance Analysis of Systems and Software, ISPASS '11, pages 134–144, Washington, DC, USA, 2011. IEEE Computer Society.

[13] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA, 2007. ACM.

[14] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[15] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, July 2013.

[16] J. Hoberock and N. Bell. Thrust library, 2008. http://https://thrust.github.io.

[17] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *DaMoN'12*, DaMoN'12, pages 55–62. ACM, 2012.

[18] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389, Aug. 2009.

[19] Kronos Group. OpenCL 1.0. http://www.khronos.org/opencl/.

[20] D. Merrill. NVIDIA CUB Library, 2011. http://nvlabs.github.io/cub/.

[21] NVIDIA. *White Paper: NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.

[22] NVIDIA. *CUDA C Programming Guide*, 7.0 edition, March 2015.

[23] NVIDIA. *NVRTC–CUDA Runtime Compilation*, 7.0 edition, March 2015.

[24] NVIDIA. *Parallel Thread Execution ISA*, 4.2 edition, March 2015.

[25] L. Nyland and S. Jones. Understanding and using atomic memory operations. In GTC 2012, Session S3101.

[26] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, Aug. 2013.

[27] G. L. Sanders and S. Shin. Denormalization effects on performance of RDBMS. In *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 3 - Volume 3*, HICSS'01, pages 3013–. IEEE Computer Society, 2001.

[28] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.*, 6(10):817–828, Aug. 2013.