Performance Analysis and Automatic Tuning of Hash Aggregation on GPUs

Viktor Rosenfeld  
DFKI, Berlin  
viktor.rosenfeld@dfki.de

Sebastian Breß  
DFKI, Berlin  
sebastian.bress@dfki.de

Tilmann Rabl  
Technische Universität Berlin  
rabl@tu-berlin.de

Steffen Zeuch  
DFKI, Berlin  
steffen.zeuch@dfki.de

Volker Markl  
Technische Universität Berlin  
volker.markl@tu-berlin.de

ABSTRACT

Hash aggregation is an important data processing primitive which can be significantly accelerated by modern graphics processors (GPUs). Previous work derived heuristics for GPU-accelerated hash aggregation from the study of a particular GPU. In this paper, we examine the influence of different execution parameters on GPU-accelerated hash aggregation on four NVIDIA and two AMD GPUs based on six different microarchitectures. While we are able to replicate some of the previous results, our main finding is that optimal execution parameters are highly GPU-dependent. Most importantly, execution parameters optimized for a specific GPU are up to 21× slower on other GPUs. Given this hardware dependency, we present an algorithm to optimize execution parameters at runtime. On GPUs with low runtime variation, our algorithm finds execution parameters that are less than 4% slower than the optimum on average and less than 18% slower in the worst case.

ACM Reference Format:

1 INTRODUCTION

Hash aggregation is an important data processing primitive. It is commonly used to implement the final aggregation in OLAP queries, to group the results of subqueries, or to eliminate duplicates. The performance of parallelized hash aggregation is mainly determined by the efficient use of processor caches [9] and by the amount of contention caused when multiple threads access a single hash table [3]. Both factors are directly related to the number of groups. Consequently, multiple parallelization strategies have been proposed that maximize cache efficiency and minimize the effects of contention depending on the group cardinality [3, 18, 9, 7]. Furthermore, the performance of GPUs kernels is strongly influenced by the thread configuration, i.e., the number of work groups per compute unit (CU) and the number of work items per work group (also called the work group size). We refer to the parallelization strategy and the thread configuration as the execution parameters of the hash aggregation operator.

Parallelized hash aggregation has been extensively evaluated on multi-core CPUs [5, 18, 9]. However, to date there exists only a single in-depth study of hash aggregation on GPUs: Karnagel et al. [7] derived rule-based heuristics to choose optimal execution parameters based on an analysis of a single NVIDIA Kepler GPU.

In this paper, we investigate the impact of the GPU hardware on hash aggregation across different GPU vendors and models. To this end, we evaluate the performance of three parallelization strategies and the influence of different thread configurations on six GPUs based on different microarchitectures. Specifically, we look at four NVIDIA GPUs based on the Kepler, Maxwell, Pascal, and Volta microarchitectures, as well as two AMD GPUs based on the 2nd and 3rd generation Graphics Core Next (GCN) microarchitectures.

Our main finding is that the optimal execution parameters strongly depend on the executing GPU. For example, in Figure 1a we show the thread configuration of each GPU that yields the fastest runtime when executing a simple sum aggregation over $2^{24}$ groups. On every GPU tested, a different thread configuration is the fastest. In Figure 1b, we show the performance penalty when we run a thread configuration that is optimized for a specific GPU on another GPU. Executing a configuration optimized for another GPU is up to 21× slower, even when both GPUs are produced by the same vendor.

In other words, our analysis shows that heuristics derived from the study of a single GPU cannot be generalized to other GPUs.

Additionally, our analysis shows that thread configuration search spaces are nearly convex, i.e., they have a single local minimum if we account for runtime variation. We exploit this property to devise an algorithm to find fast thread configurations during the execution of the hash aggregation operator. To summarize, we make the following contributions: 

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DaMoN ’19, July 01, 2019, Amsterdam, The Netherlands
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN X...X
https://doi.org/X

---

Figure 1: The best thread configuration is GPU-dependent.
(1) We perform an extensive experimental evaluation of hash aggregation on six different NVIDIA and AMD GPUs. Our analysis shows that optimal execution factors are highly GPU-specific and that implementations optimized for a specific GPU are up to 21x slower on other GPUs. Furthermore, we find that aggregation throughput is not bounded by data transfers to the GPU if the hash table exceeds the L2 cache.

(2) We devise an algorithm to choose fast GPU-specific implementations during the execution of a hash aggregation operator. On GPUs with low runtime variation, it finds execution parameters that are less than 4% slower than the optimum on average, and at most 18% slower in the worst case.

The remainder of this paper is structured as follows. In Section 2, we describe the implementation of a GPU-accelerated aggregation operator. We evaluate the influence of execution parameters on different GPUs in Section 3. In Section 4, we present an algorithm to choose fast thread configurations dynamically at runtime. We discuss related work in Section 5 and conclude in Section 6.

2 GPU-ACCELERATED HASH AGGREGATION

In this section, we describe the general implementation of a GPU-accelerated hash aggregation operator and three parallelization strategies that we examine in this paper.

2.1 Operator implementation

Our operator implementation is based on the scheme described by Karnagel et al. [7] with a few modifications to adapt it to GPUs by different manufacturers. We use the following SQL query as an example to describe the implementation in detail:

\[
\text{SELECT } g, \frac{\text{sum}(a - b)}{\text{count(*)}} \text{ FROM } R \text{ GROUP BY } g;
\]

We chose this query because it contains arithmetic operations both inside an aggregation function, i.e., \(\text{sum}(a - b)\), as well as outside of the aggregation, i.e., it divides the result of \(\text{sum}\) by \(\text{count}\).

**Assumptions.** We make the following assumptions. (1) The group cardinality \(|g|\) is known so we can size a hash table in advance and do not have to resize it during aggregation. Note that group cardinalities for arbitrary column combinations can be estimated with high accuracy and low overhead [4]. (2) The hash table fits into GPU device memory. Current GPUs support up to 48 GB of device memory [10] which allows for very large group cardinalities. (3) The input table \(R\) is stored in main memory and does not necessarily fit into GPU device memory. Integrated GPUs can access main memory directly but dedicated GPUs require a data transfer of the input over a system bus, such as PCI Express or NVLink.

**Execution stages.** We implement different stages of the hash aggregation operator in three kernels. (1) The operator first allocates sufficient memory on the GPU for the hash table and calls the \text{Initialize} kernel. This kernel marks every hash bucket as empty and stores an initial value for each aggregation function, e.g., zero for \(\text{sum}\) and \(\text{count}\). (2) The operator then processes the input in a block-wise fashion and calls the \text{Aggregate} kernel for each block. This kernel determines the hash bucket, performs computations inside aggregation functions, e.g., \((a - b)^2\) in our example, and updates all aggregates. It also tracks the number of non-empty hash buckets. In our implementation, we orchestrate the transfer of input data explicitly, instead of relying on unified memory. This allows us to measure the raw execution speed of the \text{Aggregate} kernel which is useful if the input table already resides on the GPU. We use a block size of 16 MB per column and overlap execution and data transfer. (3) Once the input has been processed, the operator allocates sufficient memory to store the final result based on the count of non-empty hash buckets determined by the \text{Aggregate} kernel. It then calls the \text{Finalize} kernel which iterates over the hash table, performs the computations outside of the aggregation functions, e.g., \(\text{sum} / \text{count}\), and materializes the result. In order to run the same code on AMD and NVIDIA GPUs, our kernels are implemented in OpenCL [17].

**Hash table parameters.** We use multiply/shift [8] as the hash function and linear probing as the hashing scheme. These hash table parameters achieve the highest throughput in an aggregation scenario, which consists only of insertions and successful lookups, if the load factor is below 90% [15].

2.2 Parallelization strategies

The \text{Aggregate} kernel implements one of three parallelization strategies which have been shown to yield high throughput on GPUs [7]. The first two strategies are also commonly used on multicore CPUs [3, 18]. The third is specifically optimized to use fast local memory found on GPUs [7].

**Shared.** In this strategy, every thread aggregates into a single, shared hash table which is placed in global GPU memory. Concurrent updates to the same hash bucket are resolved with atomic access primitives. For large group cardinalities and a uniform distribution of group values, contention is negligible because the chance of two threads accessing the same hash table bucket is small.

**Independent.** In this strategy, each thread aggregates into a thread-private hash table, thereby eliminating contention entirely. The private hash tables are placed in global GPU memory. Once a block has been processed, the private tables are merged into a global table. This strategy is feasible for very small group cardinalities. In general, a GPU has to execute many threads, thereby creating many hash table duplicates. However, all hash tables have to fit into the L2 cache to minimize memory latency.

**WorkGroupLocal.** In this strategy, the threads of a work group cooperatively aggregate into a hash table that is placed in fast local memory. Concurrent accesses are resolved using atomic access primitives. Once a block has been fully processed, the intermediate result is merged into a table stored in global GPU memory. Note that the local memory region is relatively small, typically between 32 and 96 kB. Therefore, we can use this strategy only for small to medium group cardinalities.

3 EXPERIMENTAL EVALUATION

In this section, we examine how hardware differences influence the performance of hash aggregation on GPUs. To this end, we perform four experiments on six AMD and NVIDIA GPUs. (1) We evaluate the influence of the parallelization strategy and (2) the thread configuration on the performance of the \text{Aggregate} kernel. (3) We evaluate the performance penalty when executing an \text{Aggregate} kernel optimized for a specific GPU on other GPUs. (4) We analyze the shape of the thread configuration search spaces, i.e., we test if they have a single local minimum.
3.1 Experimental setup

In our evaluation, we want to analyse the effect of contention and cache efficiency on hash aggregation performance. Therefore, we use the following query with a single aggregate and no additional computation: SELECT g, \sum(v) FROM R GROUP BY g; We vary the group cardinality |g| by powers of two between 1 and 2^28. The other evaluation parameters are as follows.

**Execution parameters.** For each group cardinality, we execute the three parallelization strategies described in Section 2.2. We vary the number of work items per work group in powers of two, from 1 to 1024. Similarly, we vary the number of work items per work group in powers of two, from 1 to the maximum work group size, i.e., 256 on AMD GPUs and 1024 on NVIDIA GPUs. In total, we evaluate up to 363 different combinations for each group cardinality. Depending on the group cardinality, some combinations are not possible because they exceed resource limitations.

**GPUs.** We run our experiments on the AMD A10-7850K (based on the 2nd generation GCN microarchitecture), the Radeon R9 Fury (GCN 3rd Gen.), the NVIDIA Tesla K40m (Kepler), the GeForce GTX 980 (Maxwell), the GeForce GTX 1080 (Pascal), and the Tesla V100 (Volta). The A10-7850K is integrated with the host CPU. The Tesla V100 is connected over NVLink 2.0 and the others over PCIe 3.0.

**Input data.** The input consists of two 32-bit integer values in columnar format. Each column is split into blocks of 16 MB. We process 32 blocks, so that the total input size is 1 GB. However, our analysis is fundamentally independent of the input size because we execute the `AGGREGATE` kernel on individual blocks and overlap kernel execution with data transfer.

**Measurement.** We measure the time to process a block with the `AGGREGATE` kernel using OpenCL profiling. We treat the input of 1 GB as a single sample consisting of 32 observations and compute the mean runtime per block. Some GPUs have a very large runtime variation. Therefore, to verify our measurements, we collect three samples consisting of 32 observations each. Unless otherwise stated, we report the results of the first sample, which indicates that there are no differences between the samples. We only measure the `AGGREGATE` kernel because the `INITIALIZE` and `FINALIZE` kernels are fixed costs regardless of the input size.

3.2 Parallelization strategy

In a first experiment, we evaluate how the group cardinality influences the performance of the parallelization strategies on different GPUs. Figure 2 shows the throughput of the fastest thread configuration for each of the three parallelization strategies. The subplots have different scales on the y axis because we want to emphasize the relative differences for each individual GPU (absolute differences between GPUs are more than an order of magnitude). We report the number of processed input tuples per second on the left y axis of each subplot and the derived throughput in GB/s on the right.

As long as the hash table fits into local GPU memory, `WORKGROUPLOCAL` is the fastest parallelization strategy. The only exception is the Tesla K40m, where `INDEPENDENT` is faster than `WORKGROUPLOCAL` for small group cardinalities. This behavior is consistent with results reported by Karnagel et al. [7] who also evaluated a Kepler GPU. On this microarchitecture, atomic operations on local memory are implemented using a lock/update/unlock pattern that is slow when contention is high [11]. Starting with the Maxwell microarchitecture, atomic operations on local memory are implemented with native instructions. Consequently, `WORKGROUPLOCAL` is at least 1.3x faster than `INDEPENDENT` on other GPUs. When the hash table does not fit into local GPU memory, `SHARED` is the fastest parallelization strategy. There is a steep drop in performance once the size of the hash table exceeds the L2 cache of the GPU. This behavior is consistent with reported results on CPUs [9].

The plots in Figure 2 show the raw performance of the `AGGREGATE` kernel without data transfers. The A10-7850K can access main memory directly, i.e., the plot shows the actual throughput of the hash aggregation operator. On dedicated GPUs, performance is limited by the data transfer bandwidth, indicated by the dashed lines in Figure 2, as long as the hash table fits into the L2 cache. However, for larger hash tables, the raw performance of the `AGGREGATE` kernel drops below the data transfer rate. For these hash tables, performance is limited by the global GPU memory latency.

To summarize, the fastest parallelization strategies are `WORKGROUPLOCAL` when the hash table fits into local memory and `SHARED` otherwise. The only exception are GPUs which do not support fast
atomic operations on local memory, e.g., Kepler GPUs. On these, Independent aggregation is faster than WorkGroupLocal for small hash tables. Moreover, the hash aggregation operator is limited by the data transfer rate when the hash table fits into the L2 cache and by the raw performance of the Aggregate kernel otherwise.

### 3.3 Thread configuration

Having determined the fastest parallelization strategy for each group cardinality, we now evaluate which thread configurations yield the best performance on different GPUs. For our analysis, we multiply the number of work groups per compute unit and the number of work items per work group of the fastest thread configuration to determine the optimal number of threads per compute unit. The scatter plots in Figure 3 show the optimal number of threads of each parallelization strategy depending on the group cardinality, i.e., the number of threads that yields the fastest performance of the Aggregate kernel. We plot all three measured samples which is why in some plots there are multiple values per group cardinality and parallelization strategy. These multiple optimal thread configurations are an indication that the runtime of the Aggregate kernel has a high variation on some GPUs. We discuss the effect of this variation in Section 3.5.

Every GPU exhibits a distinct profile in Figure 3 but we can identify three common patterns. (1) Independent aggregation shows a downward trend on every GPU. For this strategy, each thread requires a private copy of the hash table, straining GPU memory resources as the group cardinality increases. (2) For WorkGroupLocal aggregation, the optimal number of threads are clustered around GPU-specific values. The GeForce GTX 980 exhibits the least variation with 2048 threads over the entire range of groups. On the Tesla K40m, the fastest configurations also consist of 2048 threads but there are two outliers. The GeForce GTX 1080 and the Tesla V100 exhibit an inverted bowl-shaped pattern clustered around 65536 and 2048 threads, i.e., 32 work groups per compute unit and 64 work items per work group.

To summarize, the fastest thread configuration for each parallelization strategy is dependent on the group cardinality and the executing GPU. As we show in the next section, these hardware differences have a significant influence on performance.

### 3.4 Aggregate kernel performance

In this experiment, we demonstrate the importance of optimizing the thread configuration for every individual GPU. For each GPU and group cardinality, we determine the performance penalty when executing the thread configurations that are optimized for one of the other five GPUs. To compare the runtimes across group cardinalities, we normalize them relative to the fastest thread configuration for each GPU. In Figure 4a, we show the maximum performance penalty, over all group cardinalities, when the input data is already cached on the GPU. We normalize the runtimes across group cardinalities by the raw performance of the Aggregate kernel. The shaded bars in each row represent thread configurations that are optimized for a specific GPU. On the Tesla K40m or the Radeon R9 Fury, the performance penalty is up to an order of magnitude. Even when we account for the data transfer, it is up to 2.8x, as show in Figure 4b.

To summarize, even when input data is not cached on the GPU, there is a large performance penalty when we execute a thread configuration that is optimized for another GPU.

### 3.5 Thread configuration search spaces

In this experiment, we evaluate the properties of thread configuration search spaces when we fix the group cardinality and the parallelization strategy. As an example, we show in Figure 5 the...
A10-7850K

Figure 4: Maximum runtime penalty of Aggregate kernels optimized for specific GPUs (bars) executed on other GPUs (boxes).

3.6 Key insights

From our experiments we derive four key insights. (1) Independent aggregation is not competitive on newer GPUs that implement fast atomics on local memory. Instead, WorkGroupLocal should be used whenever the hash table fits into local memory. (2) The fastest thread configuration is highly GPU-specific. A thread configuration optimized for a specific GPU is up to 21x slower on other GPUs when input data is already placed in GPU memory, and up to 2.8x slower when the input has to be transferred to the GPU. Taken together, these two findings show that previously formulated heuristics, which are derived from the study of a specific NVIDIA Kepler GPU [7], are not generalizable to other GPUs. (3) We also show that when the hash table does not fit into the L2 cache of the GPU, the performance of the hash aggregation operator is bounded by the raw performance of the Aggregate kernel and not by the data transfer. (4) Finally, our analysis shows that thread configuration search spaces restricted to a specific parallelization strategy and group cardinality are nearly convex if we account for runtime variation, i.e., they typically have a single local minimum.

4 Dynamic Selection of Execution Parameters

In this section, we describe an algorithm to find fast thread configurations at runtime. It is based on the performance analysis of the previous section and exploits the nearly convex nature of the thread configuration search space. Due to limited space, we only provide a sketch of the algorithm and a limited evaluation.

4.1 Algorithm overview

Given a group cardinality and a parallelization strategy, our algorithm explores the thread configuration search space to find a fast configuration. Instead of evaluating all of the up to 121 thread configurations, it starts from an initial thread configuration \( t_0 \) and follows the gradient to a local minimum. During the descent, we treat performance plateaus as a special case. We define two thread configurations as similar when one of their runtimes is within an
AMD GPUs only support 256 work items per work group, so we focus on the Tesla K40m and the GeForce GTX 980, there are still a few groups that implement fast atomics on local GPU memory. (2) The operator variants to adapt to changes in data characteristics. In contrast to our work, the search space contains only a small number of implementations that are known to perform well.

Micro adaptivity [14] uses a multi-armed bandit strategy to select operator variants to adapt to changes in data characteristics. In contrast to our work, the search space contains only a small number of implementations that are known to perform well. Micro adaptivity [14] uses a multi-armed bandit strategy to select operator variants to adapt to changes in data characteristics. In contrast to our work, the search space contains only a small number of implementations that are known to perform well.

4.2 Evaluation and discussion

Due to limited space, we only provide an aggregate evaluation of the dynamic selection algorithm. We evaluate two metrics. (1) The quality of the found thread configuration $v_f$ is its normalized runtime relative to the fastest thread configuration in the search space for a particular group cardinality. (2) The effort is the runtime of the algorithm until it converges to $v_f$ as a function of the time required for a full evaluation of the thread configuration space. Note that the size $s$ of the similarity interval determines a tradeoff between the two metrics. As we increase the similarity interval to identify performance plateaus, we test more and slower thread configurations which increases the effort. For this evaluation, we treat thread configurations as similar when their runtimes are within 3% of each other, as it provides a reasonable tradeoff. For each group cardinality and GPU, we perform 30 evaluations of the algorithm. The boxplots in Figure 6 show the median quality and effort, over all group cardinalities, respectively. On average, the algorithm converges in just 1% of the time required for a full evaluation of the search space. On NVIDIA GPUs, the median runtime of the found thread configuration is within 4% of the fastest. However, on the Tesla K40m and the GeForce GTX 980, there are still a few groups for which the found thread configuration is up to 1.18× slower than the fastest. This indicates that the similarity range of 3% is too narrow to identify performance plateaus in all cases. On the Radeon R9 Fury, the median runtime is 1.22× slower than the fastest. However, the converged thread configurations are within a standard deviation of the fastest in 99% of the evaluations. The slowdown for this GPU is caused by its very high runtime variation.

5 RELATED WORK

In this section, we discuss related work that we have not yet described. We group related work by topics.

Data processing on GPUs. As GPUs have become more prevalent, they have been used as query processors in dedicated database research prototypes, e.g., GDB [5], OcElo [6], GAPDB [19], CoGaDB [1], GPL [12], Voodoo [13], and Hawk [2].

Operator tuning during query execution. Rosenfeld et al. propose a genetic algorithm to find optimal execution parameters, including the thread configuration, for different operators on heterogeneous processors [16]. They make no assumptions about the search space, whereas we exploit its convex shape.

Zeuch et al. employ a cost model based on performance counters to optimize the predicate order during query execution [20]. Conversely, our algorithm makes decisions based on runtime.

6 CONCLUSION

In this paper, we evaluate the influence of two execution parameters, the parallelization strategy and the thread configuration, on GPU-accelerated hash aggregation. Based on a study of six AMD and NVIDIA GPUs, our analysis yields four major findings. (1) Heuristics derived in previous work [7] are not applicable to newer GPUs which implement fast atomics on local GPU memory. (2) The optimal thread configuration is highly dependent on the executing GPU. (3) A thread configuration optimized for a specific GPU is up to 21× slower than the fastest and 54× slower when data has to be transferred to the GPU. (4) The runtime of hash aggregation is limited by raw aggregation kernel performance, and not the data transfer rate, when the hash table exceeds the L2 cache of the GPU. We also show that the thread configuration search space for a specific parallelization strategy and group cardinality is nearly convex, i.e., it has a single local minimum when we account for runtime variation. We exploit this property in an algorithm that integrates into a block-based query execution strategy to find fast thread configurations dynamically at runtime. On NVIDIA GPUs, we are able to find thread configurations that are less than 4× slower than the fastest on average and less than 18% slower in the worst case. However, on GPUs with high runtime variation we have to trade-off the quality of the found variant against the overhead of finding it.

Our algorithm is not limited to hash aggregation. It can be used for any operator which allows us to change the actual implementation during query execution while still making progress, as long as the thread configuration search space is convex.

(a) Quality of found variant
(b) Evaluation time

Figure 6: Quality and effort of dynamic selection.
REFERENCES


[16] V. Rosenfeld et al. "The Operator Variant Selection Problem on Heterogeneous Hardware". In: ADMS@VLDB. 2015, pp. 1–12.


