Performance Analysis and Automatic Tuning of Hash Aggregation on GPUs

Viktor Rosenfeld[1] Sebastian Breß[1,2] Steffen Zeuch[1,2] Tilmann Rabl[3,*] Volker Markl[1,2]
[1]German Research Center for Artificial Intelligence [2]Technische Universität Berlin [3]Hasso Plattner Institute, University of Potsdam

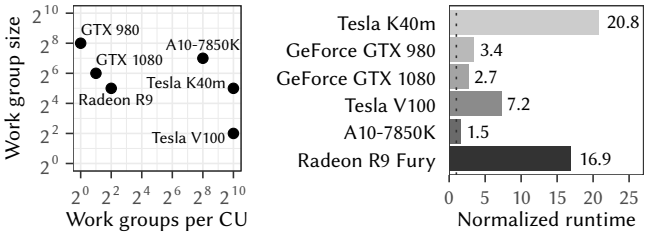
ABSTRACT

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 average, our algorithm converges on a result in less than 1% of the time required for a full evaluation of the search space. In this time, it finds execution parameters that are at most 1% slower than the optimum in 90% of our experiments. In the worst case, our algorithm finds execution parameters that are at most 1.29× slower than the optimum.

ACM Reference Format:
Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Mark. 2019. Performance Analysis and Automatic Tuning of Hash Aggregation on GPUs. In *International Workshop on Data Management on New Hardware (DaMoN'19), July 1, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3329785.3329922

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 [10] 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, 8, 10, 19]. 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.



(a) Thread configurations **(b) Performance penalty**

Figure 1: The best thread configuration is GPU-dependent.

Parallelized hash aggregation has been extensively evaluated on multi-core CPUs [3, 10, 19]. However, to date there exists only a single in-depth study of hash aggregation on GPUs: Karnagel et al. [8] 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:

(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

*Work conducted while the author was employed at Technische Universität Berlin.

DaMoN'19, July 1, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6801-8/19/07…$15.00
https://doi.org/10.1145/3329785.3329922

GPU-specific and that implementations optimized for a specific GPU are up to 21× slower on other GPUs.
(2) We devise an algorithm to choose fast GPU-specific implementations during the execution of a hash aggregation operator. Our algorithm reliably finds fast execution parameters in a fraction of the time required for a full evaluation of the search space.

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. [8] 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:

```
SELECT g, sum((a − b)²) / count(*) FROM R GROUP BY g;
```

We chose this query because it contains arithmetic operations both inside an aggregation function, i.e., $\texttt{sum}((a-b)^2)$, as well as outside of the aggregation, i.e., it divides the result of `sum` by `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 [11] 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 Initialize kernel. This kernel marks every hash bucket as empty and stores an initial value for each aggregation function, e.g., zero for `sum` and `count`. (2) The operator then processes the input in a block-wise fashion and calls the 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 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 Aggregate kernel. It then calls the Finalize kernel which iterates over the hash table, performs the computations outside of the aggregation functions, e.g., `sum / count`, and materializes the result. In order to run the same code on AMD and NVIDIA GPUs, our kernels are implemented in OpenCL [18].

Hash table parameters. We use multiply/shift [9] 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% [16].

2.2 Parallelization strategies

The Aggregate kernel implements one of three parallelization strategies which have been shown to yield high throughput on GPUs [8]. The first two strategies are also commonly used on multi-core CPUs [3, 19]. The third is specifically optimized to use fast local memory found on GPUs [8].

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 five 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 Aggregate kernel. (3) We evaluate the performance penalty when executing an 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. (5) We analyze the degree of runtime variation and the influence of outliers on different GPUs.

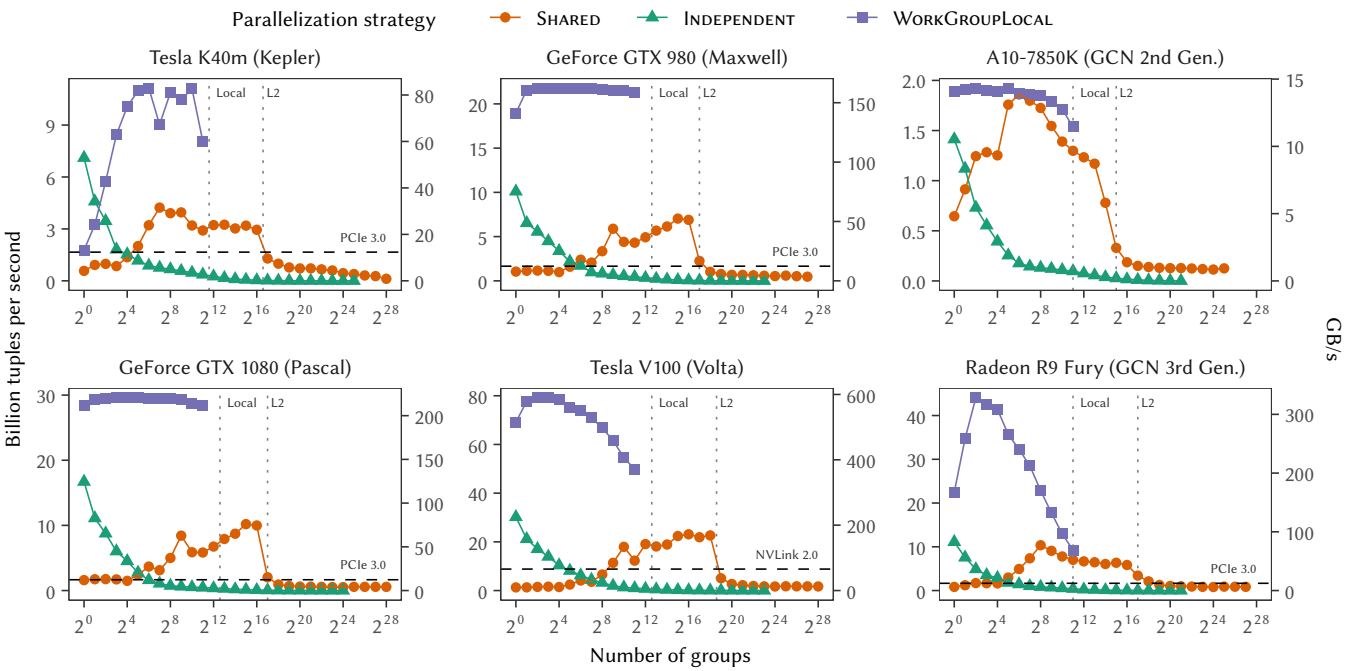


Figure 2: Throughput of parallelization strategies depending on group cardinality (different scales on y axis).

3.1 Experimental setup

In our evaluation, we focus on 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 groups per compute unit 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. We list the memory configuration and additional properties of these GPUs in Table 2 in Appendix A.

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. The group values are randomly generated from a uniform distribution.

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 exhibit a high degree of 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 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. [8] who also evaluated a Kepler GPU. On this microarchitecture, atomic operations on local memory are implemented using a lock/update/unlock pattern that

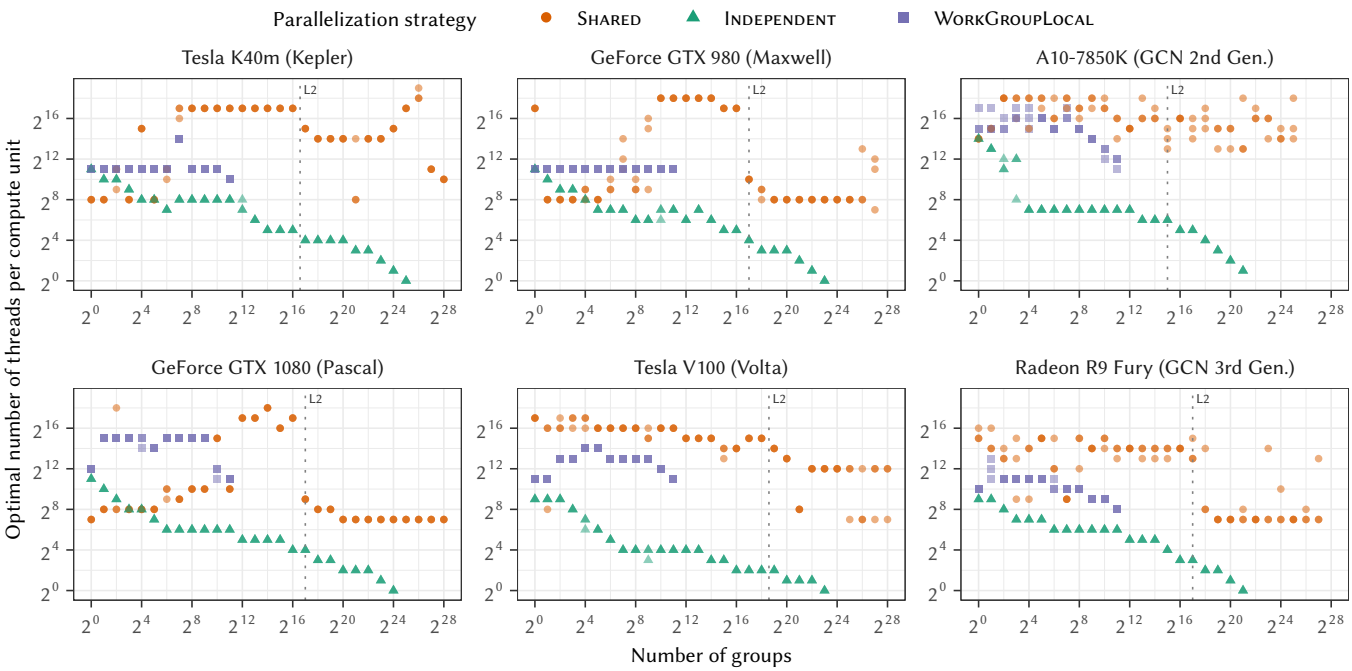


Figure 3: Optimal number of threads depending on group cardinality.

is slow when contention is high [12]. Starting with the Maxwell microarchitecture, atomics on local memory are implemented with native instructions. Consequently, WorkGroupLocal is at least 1.3× 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 [10].

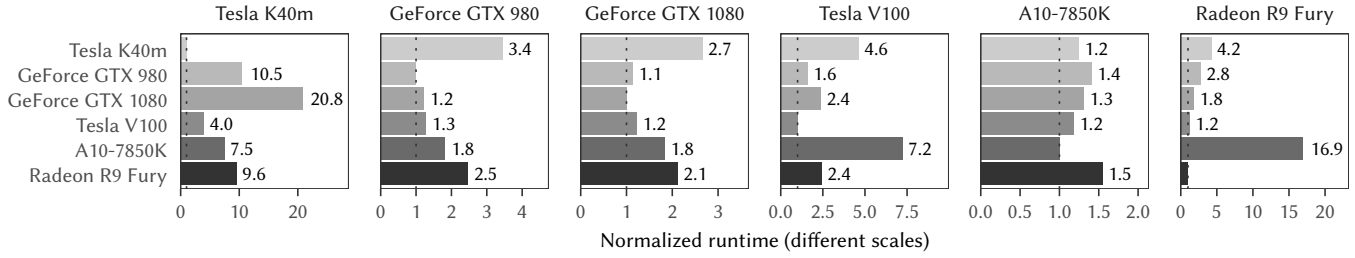
The plots in Figure 2 show the raw performance of the Aggregate kernel without data transfers. The A10-7850K can access main memory directly, i.e., the plot shows the actual throughput of the hash aggregation operator. On dedicated GPUs, performance is limited by the data transfer bandwidth, indicated 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.

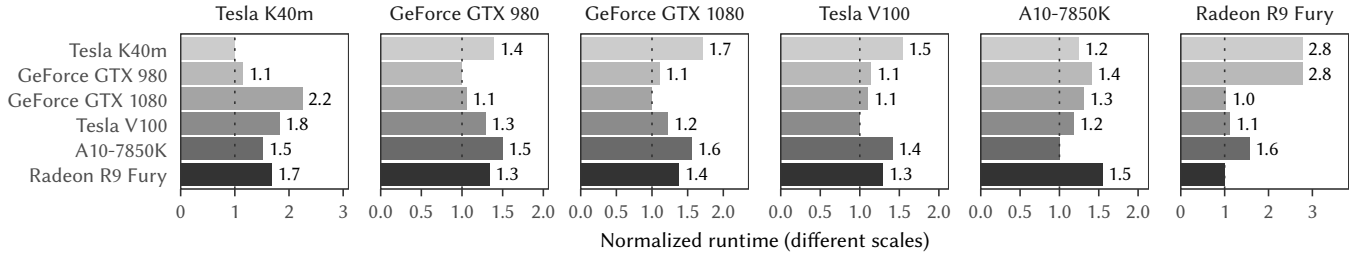
3.3 Thread configuration

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 effects of this variation in Section 3.5 and analyze it in detail in Section 3.6.

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 32768 and 8192 threads, respectively. Finally, the two AMD GPUs show a downward trend clustered around 65536 and 2048 threads. (3) Shared aggregation exhibits the most variation. A common pattern is a change at the boundary of the L2 cache. This pattern is most pronounced on the GeForce GTX 980, the GTX 1080, and on the Radeon R9 Fury. Note that different thread configurations can yield the same number of threads. For example, on the GeForce GTX 980, the fastest thread configurations consist of 2048 threads but



(a) Input data is already placed on the GPU.



(b) Input data has to be transferred to the GPU.

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

the actual configurations vary between 2×1024, 4×512, and 32×64 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 placed in GPU memory. The subplots in each column represent a GPU on which we execute 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 2.8×, 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 performance of different thread configurations for Shared aggregation with 2^{27} groups on the Tesla V100. The heatmap appears to be convex at first glance, but there are multiple local minima, as indicated by the bold values. Two of these local minima, at 1×128 and 1024×4 threads, are selected as the global minimum in different samples, as indicated by Figure 3 in Section 3.3. The local minima are surrounded by *performance plateaus*, i.e., regions where we cannot reliably determine which thread configuration is the fastest. We define that two thread configurations are part of a performance plateau when one of their runtimes is contained within an interval around the other. In Figure 5, we show in italics that the two local minima at 1×128 and 1024×4 threads are part of a single performance plateau. The extent of performance plateaus depend on the size of the interval we allow around each runtime. In our analysis,

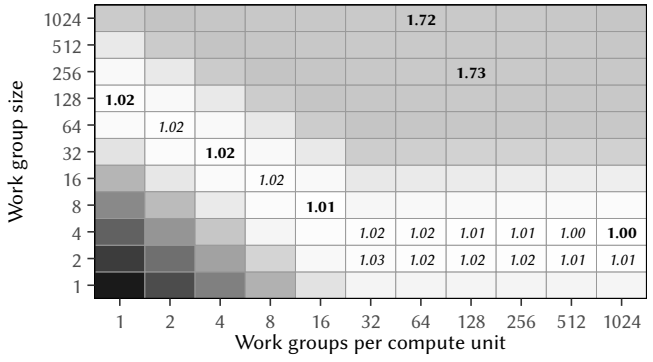


Figure 5: Normalized runtime depending on thread configuration. Darker colors are slower. Bold values indicate local minima. Italics indicate a performance plateau.

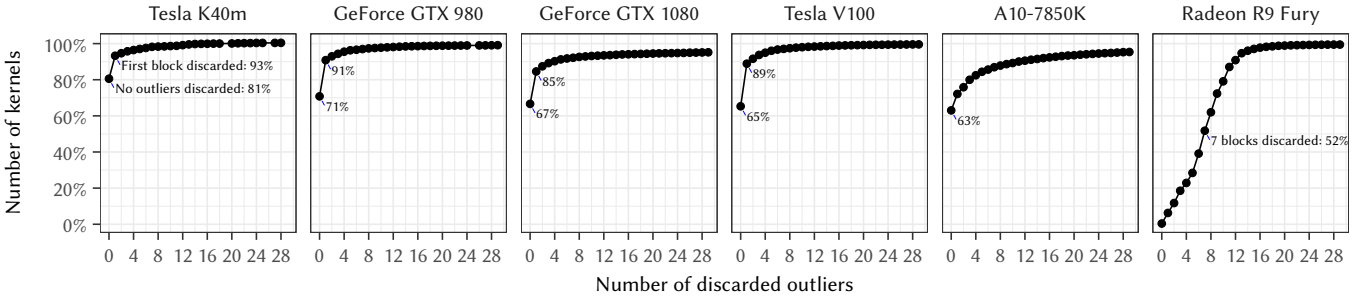


Figure 6: Influence of outliers at the beginning of the measurement on the degree of variation of Aggregate kernels.

we set this interval to either a single standard deviation or 10% of the absolute value, whichever is greater, in each direction. Given this definition, only 2% out of 1143 tested search spaces have more than one local minimum.

To summarize, individual thread configuration search spaces are *nearly convex*, i.e., they typically have a single local minimum if we account for runtime variation.

3.6 Aggregate kernel runtime variation

As we mentioned in the previous two sections, the runtime of the Aggregate kernel exhibits a high degree of variation on some GPUs. It is necessary to take this variation into account when comparing the performance of different thread configurations, as we did in Section 3.5. Thus, in this experiment, we analyze the degree of variation of the Aggregate kernel runtimes on different GPUs. In the following, we first quantify the degree of variation for each GPU and then analyze the influence of outliers.

Degree of variation. To quantify the degree of variation, we use the *coefficient of variation*, i.e., the ratio of the standard deviation and the mean, of each sample. This relative metric captures the fact that the degree of variation must be understood in the context of the measured data. The same standard deviation may indicate a low degree of variation for slow kernels and a high degree of variation for fast kernels. In Figure 7, we summarize the coefficient

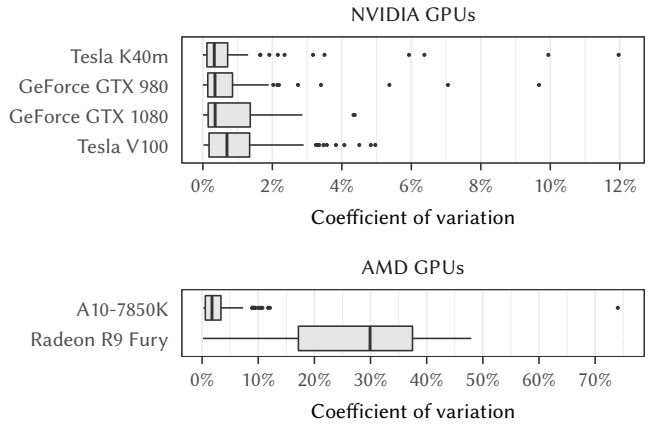


Figure 7: Degree of variation of Aggregate kernel runtimes (different scales on x axis for NVIDIA and AMD GPUs).

of variation for every Aggregate kernel. Note that we use a different scale on the x axis for NVIDIA and AMD GPUs. For clarity, we only show the variation of the fastest thread configuration for each parallelization strategy and group cardinality. This filtering biases the plot towards fast Aggregate kernels but it resembles the plot which includes all thread configurations, except for outliers. However, it is more important to consider the degree of variation when comparing fast Aggregate kernels, as we are interested in finding these. Our main observation is that NVIDIA GPUs exhibit a low degree of runtime variation, with a median coefficient of variation below 0.7%. In contrast, the Radeon R9 Fury exhibits a substantial degree of runtime variation. On AMD GPUs, the outliers of the measured kernel runtimes often exhibit a higher magnitude than the outliers we encounter on NVIDIA GPUs. In addition, the samples collected on the Radeon have significantly more outliers than those collected on other GPUs.

Influence of outliers. The outliers of the measured execution runtimes are not uniformly distributed. Instead, they are typically clustered at the beginning of each sample. This behavior is demonstrated in Figure 6. The figure shows the number of Aggregate kernels for which the coefficient of variation, when it is computed over a rolling window of three blocks, drops below a threshold value when we discard outliers in the beginning of the measurement. For this analysis, we choose 1.05% as the threshold. This value corresponds to the 99th percentile of the minimal coefficient of variation computed over any consecutive window of three blocks for every tested Aggregate kernel. We observe three key points. (1) For a majority of kernels on NVIDIA GPUs and on the AMD A10-7850K, there are no significant outliers in the beginning of our measurements. Concretely, after processing the first window of three blocks, the coefficient of variation is smaller than the threshold for at least 63% of the Aggregate kernels. (2) If necessary, discarding just a single outlier substantially reduces the degree of variation on NVIDIA GPUs, as indicated by the steeply rising curves. (3) On AMD GPUs, the influence of outliers is more pronounced, as indicated by the slowly rising curves. Especially on the Radeon R9 Fury, even after we discard the first seven blocks as outliers, the coefficient of variation is smaller than the threshold for only 52% of the Aggregate kernels.

To summarize, NVIDIA GPUs exhibit a low degree of variation, which can be further reduced by discarding a single outlier in the beginning of the measurement. In contrast, AMD GPUs, especially the Radeon R9 Fury, exhibit a high degree of variation.

3.7 Key insights

From our experiments we derive five 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 21× slower on other GPUs when input data is already placed in GPU memory, and up to 2.8× 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 [8], 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 Aggregate kernel is bounded by global GPU memory latency and not by the data transfer. (4) 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. (5) NVIDIA GPUs generally have a lower degree of variation than AMD GPUs.

4 DYNAMIC SELECTION OF EXECUTION PARAMETERS

In this section, we describe our optimization algorithm to find fast Aggregate kernels 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. First, we provide an overview of our algorithm and describe in detail how it handles performance plateaus and measurement outliers. Afterwards, we evaluate our algorithm on our six test GPUs. We conclude with a discussion on how to integrate it with query execution in database management systems.

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 and follows the gradient to a local minimum. During the descent, the algorithm treats performance plateaus as a special case. If the runtimes of two configurations are similar, it explores the search space from both thread configurations, effectively forking the path taken through the search space. Algorithm 1 shows the pseudocode of our algorithm. In the following, we describe its key aspects in detail.

Notation and definitions. We use c_i to represent a thread configuration and the notation $t(c_i)$ to express its runtime. We define the runtimes of c_i and c_j as similar if one of them is within the interval determined a similarity coefficient s around the other: $t(c_i) \sim_s t(c_j) \iff (1-s)t(c_i) \leq t(c_j) \leq (1+s)t(c_i)$.

Inputs, initial steps, and main optimization loop. The inputs of Algorithm 1 are an *initial thread configuration* c_0, a *similarity coefficient* s, and a *pruning factor* p. The similarity coefficient is used by our algorithm to identify two runtimes as part of a performance plateau. In contrast, the pruning factor is used to exclude parts of the search space when exploring multiple branches from performance plateaus. During execution, our algorithm maintains a

Input: An initial thread configuration c_0; a similarity coefficient s; a prune factor $p > 1$.

```
1  t(c_0) ← ProcessBlocksWith(c_0)
2  c_f ← c_0
3  Q ← {c_0}
4  while Q ≠ ∅ do
5      c_i ← Peek(Q)
6      if t(c_i) > p × t(c_f) then
7          Pop(Q)                          ▷ Prune slow reference configurations.
8      else
9          N ← UntestedNeigborhoodOf(c_i)
10         if N ≠ ∅ then
11             c_j = Pop(N)                ▷ Evaluate neighbor of current ...
12             t(c_i) ← ProcessBlocksWith(c_j)   ... configuration.
13             if t(c_i) ~_s t(c_j) then   ▷ Keep record of ...
14                 Push(Q, c_j)            ... performance plateaus.
15             else if t(c_j) < t(c_i) then   ▷ Follow gradient ...
16                 ReplaceFirst(Q, c_j)    ... in search space.
17             end
18             if t(c_j) < t(c_f) then
19                 c_f ← c_j               ▷ Update fastest configuration.
20             end
21         else
22             Pop(Q)                      ▷ Backtrack from local minimum.
23         end
24     end
25 end
```

Algorithm 1: Dynamic selection of thread configurations.

FIFO queue Q containing reference positions c_i from which it explores parts of the search space. It also tracks the fastest thread configuration c_f it has encountered so far. The algorithm starts by executing the initial thread configuration c_0 on a number of blocks to determine its runtime (line 1). It then sets c_0 as the fastest thread configuration encountered so far and initializes the reference queue Q with c_0 (lines 2–3). The algorithm then enters the main optimization loop which continues as long as there are reference positions in the queue (line 4). In each loop iteration, the algorithm first compares the execution time of the reference configuration c_i at the top of the queue to the fastest known thread configuration c_f. If c_i is slower than indicated by the pruning factor p, it is removed from the queue and pruned (lines 5–7). Otherwise, the algorithm selects a neighbor c_j of the current thread configuration c_i and evaluates its runtime on a number of blocks (lines 9–12).

Handling performance plateaus. When comparing the runtimes of two thread configurations c_i and c_j, the algorithm distinguishes three results of the comparison to handle performance plateaus. (1) If c_i and c_j have similar runtimes with regard to the similarity range s, i.e., if they are part of a performance plateau, both are added to the top of the queue (lines 13–14). In subsequent loop iterations, the algorithm follows the gradient in the search space from c_i and c_j independently, until one or both of the branches are pruned. (2) Otherwise, if c_j is strictly faster than c_i, then c_i is replaced with c_j in the queue (lines 15–16). (3) Otherwise, if c_j is strictly slower than c_i, the algorithm tries out a different untested neighbor c_j' of c_i in the next loop iteration. If there are no more neighbors of c_i, the algorithm has reached a local minimum. It

removes c_i from the queue and backtracks to a previously encountered c_i' inside a performance plateau (line 22). Note that by setting the similarity coefficient s to 0, the algorithm ignores performance plateaus and strictly follows the gradient in the search space.

Handling measurement outliers. When measuring the runtime of a thread configuration, the algorithm has to satisfy two conflicting requirements. On the one hand, we want to reduce the influence of slow thread configurations. On the other hand, we want to reduce the influence of any outliers in the beginning of the measurement. To this end, the algorithm executes a thread configuration on three blocks. It then determines the degree of variation and compares it to a threshold value v_{max}. If the variation is below the threshold, the algorithm returns the mean runtime as the measurement. Otherwise, the algorithm discards the first measured value and executes the thread configuration on another block. The algorithm continues until the variation drops below the threshold or it has processed b_{max} blocks. This process is encapsulated by the function `ProcessBlocksWith(`c_i`)` in Algorithm 1. Instead of computing the coefficient of variation, as we do in Section 3.6, we compute the range between the minimal and maximal measured values and divide it by the mean. This approach significantly reduces overhead by eliminating a costly square root operation that is part of computing the standard deviation. We set $v_{max} = 0.019$, which corresponds to the 99th percentile of the normalized range coefficient computed over windows of three blocks.

Initial thread configurations. The number of loop iteration of our algorithm, and therefore its runtime, depends on the initial thread configuration c_0. Based on our analysis in Section 3, we choose an initial configuration that minimizes algorithm runtime. To this end, we determine the thread configurations with the lowest normalized runtimes averaged over all GPUs and group cardinalities. Specifically, we use 1×512 threads for Shared, 1×256 threads for Independent, and 4×512 threads for WorkGroupLocal aggregation. Note that AMD GPUs only support 256 work items per work group, so we adjust the initial configurations accordingly.

Optimizing multiple parallelization strategies. To support multiple initial thread configurations c_0', the algorithm evaluates and adds them to the queue of reference positions before entering the optimization loop. This approach allows us to probe the thread configuration search space from multiple positions and to optimize multiple parallelization strategies simultaneously.

Support for other operators. Our algorithm can optimize any operator that satisfies the following two requirements in addition to a block-wise processing model. (1) The thread configuration search space has to be (nearly) convex, since the algorithm exploits this property to efficiently search it. (2) The algorithm must be able to change the implementation of the operator for each processed block without loosing the progress made by processing previous blocks. For example, hash aggregation satisfies the second requirement because every parallelization strategy merges the results of processing a block into a single, globally shared aggregation table. This shared hash table encapsulates the global state of the aggregation. Many important database operations satisfy the second requirement, e.g., any operator that materializes its output, such as selections and partitioned joins [2, 5, 15, 17]. Note that some operator implementations may use local state that is incompatible with other implementations, as long as it can be discarded once the block is processed. For example, in our analysis, the number of private hash table copies used by Independent and WorkGroupLocal aggregation differs. We can also process blocks with different hash functions, hashing schemes, and load factors.

Table 1: Tested hyper parameters.

Parameters	Description
$s = 0$	Ignore performance plateaus
$s = 0.06,\ \ p = 1.07$	Detect performance plateaus
$v_{max} = 0.019$	Variation threshold to discard outliers
$b_{max} = 10$	Max. number of blocks for measurements

4.2 Algorithm evaluation

To evaluate our algorithm, we examine the influence of the two hyper parameters we introduced to manage performance plateaus, i.e., the similarity range s and the pruning factor p.

Experimental setup. We evaluate our algorithm with a variety of different hyper parameters. For brevity, we report the results of two scenarios. (1) In the *strict runtime comparison* scenario, the algorithm ignores performance plateaus. This scenario serves as our baseline. (2) In the *detection of performance plateaus* scenario, we use a similarity coefficient $s = 0.06$ to detect performance plateaus and a pruning factor $p = 1.07$ to remove slow thread configurations during the search. Increasing these values further does not result in an improvement of the runtime of the found thread configuration. The other hyper parameters are listed in Table 1. To speed up the evaluation of multiple hyper parameters, we inject the execution runtimes measured in Section 3 into our algorithm.

Metrics. We evaluate four metrics. (1) The *quality of the found thread configuration* c_f is its normalized runtime relative to the fastest thread configuration in the search space for a particular group cardinality. (2) The *cost expended for optimization* is the difference between the cumulative normalized runtime until the algorithm converges to a thread configuration and the number of processed blocks. This metrics indicates how many additional blocks the system could have processed if it had known the fastest thread configuration a priori. (3) The *algorithm overhead* is the time spent by the optimization algorithm to make its decision as a fraction of the total processing time of a block. (4) The *optimization effort* is the runtime of the algorithm until it converges to c_f as a fraction of the time required for a full evaluation of the search space. In the following, we present the results of our evaluation according to these metrics.

(1) Quality of found configuration. Figure 8 shows the runtime of the found Aggregate kernel relative to the fastest configuration per group cardinality. By simply following the gradient of the thread configuration search space, the algorithm finds the fastest configuration in 36% of our experiments, even if it ignores performance plateaus. When treating thread configurations with similar runtimes as performance plateaus, the algorithm finds the fastest configuration in 62% of our experiments. In fact, in 90% of the time, the found configuration is at most 1% slower than the fastest. The worst-case performance of the found configuration improves from a factor of 1.39× to 1.29×.

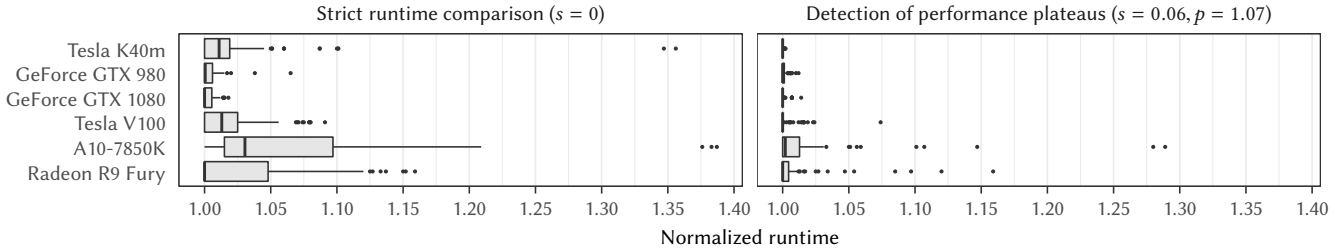


Figure 8: Quality of found configuration.

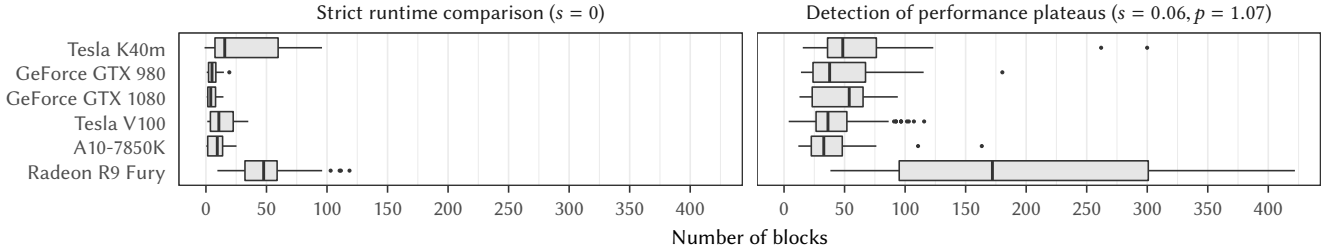


Figure 9: Cost expended for optimization.

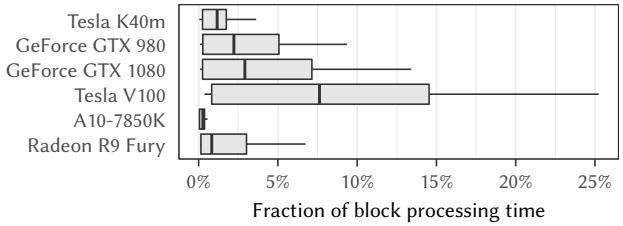


Figure 10: Algorithm overhead during query execution.

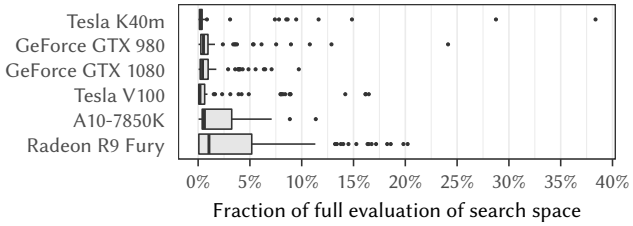


Figure 11: Optimization effort compared to full evaluation.

(2) Cost expended for optimization. Figure 9 shows how many additional blocks the database could have processed in the time our algorithm converges to a thread configuration, if it had known the fastest configuration from the beginning. This cost increases as the algorithm treats configurations with similar runtimes as performance plateaus and processes additional blocks to reduce the influence of outliers. However, this work is not wasted as the algorithm still makes progress towards the result. When the algorithm detects performance plateaus, the median cost is between 33 and 54 blocks, which in our setup corresponds to 1.06 GB and 1.73 GB of data, respectively. The median cost on the Radeon R9 Fury, which exhibits a high degree of runtime variation, is 172 blocks.

(3) Algorithm overhead during query execution. Figure 10 shows the amount of overhead created by the handling of performance plateaus and the detection of outliers. The median overhead per GPU is between 0.3% and 7.6%. The overhead is generally larger for faster GPUs, e.g., the Tesla V100, and for faster kernels, e.g., WorkGroupLocal aggregation.

(4) Comparison to full evaluation. Figure 11 shows the time our algorithm requires to converge to a fast variant as a fraction of the time required for a full evaluation of the search space. On every GPU, the median runtime is below 1.05% of a full evaluation. There are a few outliers when the algorithm requires more than 20%. In these cases, the thread configuration search space contains many configurations with similar runtimes which form large performance plateaus. However, since these configurations are only marginally slower than the fastest, they do not add significant overhead to query execution.

4.3 Summary of key results and discussion

From our evaluation, we derive three key results. (1) By following the gradient in the thread configuration search space, our algorithm finds fast execution parameters for the Aggregate kernel in 36% of our experiments. (2) Treating thread configurations with similar runtimes as performance plateaus improves the success rate to 62%. The worst-case runtime of the found thread configuration improves from a factor of 1.39× to 1.29× compared to the fastest. (3) The runtime of the algorithm is highest on GPUs that exhibit a large degree of runtime variation but is still less than 1% of a full evaluation of the search space on average.

Our algorithm is designed to optimize execution parameters during query execution. However, fast execution parameters depend not only on the GPU, as we have shown in Section 3, but also on query and data characteristics [3, 8, 19]. Performing the dynamic selection for every query adds significant overhead, as Figure 9 indicates. Alternatively, if the query workload is known a priori, our algorithm can determine the fastest execution workload per query and store these values. As Figure 11 shows, our algorithm

greatly reduces the evaluation time compared to a full evaluation of the search space. To support arbitrary queries, our algorithm can determine fast execution parameters for a representable set of benchmark queries. These queries would contain various combinations of aggregation functions and execute over data sets with different data characteristics. Based on the execution parameters determined for these queries, we can build a model to predict the fastest set of execution parameters for any query on a specific GPU. Our algorithm can then fine-tune these predicted thread configurations during query execution.

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 [6], Ocelot [7], GPUDB [20], CoGaDB [1], GPL [13], and Voodoo [14].

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 [17]. They make no assumptions about the search space, whereas we exploit its convex shape.

Micro adaptivity [15] 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.

Zeuch et al. employ a cost model based on performance counters to optimize the predicate order during query execution [21]. Conversely, our algorithm makes decisions based on runtime.

Hawk [2] is a hardware-adaptive query compiler for heterogeneous processors. It performs a separate tuning step to optimize operator implementations based on a representative query workload. In contrast to our work, Hawk performs a simple structured experiment and makes no assumptions about the search space.

6 CONCLUSION

Hash aggregation is an important data processing primitive which can be significantly accelerated by modern GPUs. In this paper, we demonstrate that the optimal implementation of a GPU-accelerated hash aggregation operator strongly depends on the executing GPU and that findings presented in previous work are not generalizable. To address this hardware dependency, we propose an optimization algorithm to find GPU-adapted operator implementations.

We extensively 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 five major findings.

(1) Heuristics derived in previous work [8] are not applicable to newer GPUs which implement fast atomics on local memory.
(2) The optimal thread configuration is highly dependent on the executing GPU. A thread configuration optimized for a specific GPU is up to 21× slower than the optimum when it is executed on another GPU if the input is already placed in GPU memory. It is up to 2.8× slower if data has to be transferred to the GPU.
(3) 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.
(4) NVIDIA GPUs exhibit a low degree of runtime variation whereas AMD GPUs exhibit a higher degree of variation.
(5) 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.

Based on these findings, we propose an algorithm to find fast operator implementations dynamically at runtime. It exploits the convexity of the search space to reduce the search time to a fraction of the time required for a full evaluation. By treating thread configurations with similar runtimes as performance plateaus, if finds fast implementations in 90% of our experiments. Depending on the GPU, the worst case performance of the found implementation is up to 1.29× slower than the optimum. 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 GPU PROPERTIES

In Table 2, we list the memory configuration and additional properties of the six GPUs evaluated in this paper.

ACKNOWLEDGMENTS

This work was funded by the EU project E2Data (780245), the DFG Priority Program Scalable Data Management for Future Hardware (MA4662-5), and the German Ministry for Education and Research as BBDC II (01IS18025A).

Table 2: GPU properties.

	AMD A10-7850K	AMD R9 Fury	NVIDIA Tesla K40m	NVIDIA GeForce GTX 980	NVIDIA GeForce GTX 1080	NVIDIA Tesla V100
Microarchitecture	GCN 2nd Gen.	GCN 3rd Gen.	Kepler	Maxwell	Pascal	Volta
Integration	on die	PCIe 3.0	PCIe 3.0	PCIe 3.0	PCIe 3.0	NVLink 2.0
Compute units	8	56	15	16	20	80
Global memory	1.5 GB	4 GB	11.2 GB	3.9 GB	7.9 GB	15.8 GB
Local memory	32 kB	32 kB	48 kB	96 kB	96 kB	96 kB
L2 cache	512 kB	2 MB	1.5 MB	2 MB	2 MB	6 MB

REFERENCES

[1] S. Breß, H. Funke, and J. Teubner. "Robust Query Processing in Co-Processor-accelerated Databases". In: *SIGMOD*. 2016, pp. 1891–1906.

[2] S. Breß et al. "Generating custom code for efficient query execution on heterogeneous processors". In: *VLDBJ* 27.6 (Dec. 2018), pp. 797–822.

[3] J. Cieslewicz and K. A. Ross. "Adaptive Aggregation on Chip Multiprocessors". In: *VLDB*. 2007, pp. 339–350.

[4] M. J. Freitag and T. Neumann. "Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates". In: *CIDR*. 2019.

[5] H. Funke et al. "Pipelined Query Processing in Coprocessor Environments". In: *SIGMOD*. 2018, pp. 1603–1618.

[6] B. He et al. "Relational Query Coprocessing on Graphics Processors". In: *TODS* 34.4 (2009), 21:1–21:39.

[7] M. Heimel et al. "Hardware-Oblivious Parallelism for In-Memory Column-Stores". In: *PVLDB* 6.9 (2013), pp. 709–720.

[8] T. Karnagel, R. Mueller, and G. M. Lohman. "Optimizing GPU-accelerated Group-By and Aggregation". In: *ADMS@VLDB*. 2015, pp. 13–24.

[9] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*. 2nd ed. Vol. 3. 1998.

[10] I. Müller et al. "Cache-Efficient Aggregation: Hashing Is Sorting". In: *SIGMOD*. 2015, pp. 1123–1136.

[11] NVIDIA. *Quadro RTX 8000 Data Sheet*. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-rtx-8000-us-nvidia-946977-r1-web.pdf (visited on: 2019/03/26). 2019.

[12] NVIDIA. *Tuning CUDA Applications for Maxwell*. https://docs.nvidia.com/cuda/maxwell-tuning-guide/ (visited on: 2019/03/26). 2017.

[13] J. Paul, J. He, and B. He. "GPL: A GPU-based Pipelined Query Processing Engine". In: *SIGMOD*. 2016, pp. 1935–1950.

[14] H. Pirk et al. "Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware". In: *PVLDB* 9.14 (Oct. 2016), pp. 1707–1718.

[15] B. Răducanu, P. Boncz, and M. Zukowski. "Micro Adaptivity in Vectorwise". In: *SIGMOD*. 2013, pp. 1231–1242.

[16] S. Richter, V. Alvarez, and J. Dittrich. "A Seven-dimensional Analysis of Hashing Methods and Its Implications on Query Processing". In: *PVLDB* 9.3 (Nov. 2015), pp. 96–107.

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

[18] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73.

[19] Y. Ye, K. A. Ross, and N. Vesdapunt. "Scalable Aggregation on Multicore Processors". In: *DaMoN*. 2011, pp. 1–9.

[20] Y. Yuan, R. Lee, and X. Zhang. "The Yin and Yang of Processing Data Warehousing Queries on GPU Devices". In: *PVLDB* 6.10 (Aug. 2013), pp. 817–828.

[21] S. Zeuch, H. Pirk, and J.-C. Freytag. "Non-invasive Progressive Optimization for In-memory Databases". In: *PVLDB* 9.14 (Oct. 2016), pp. 1659–1670.