Robust Query Processing in Co-Processor-accelerated Databases

Sebastian Breß[*]
German Research Center for Artificial Intelligence
Germany
sebastian.bress@dfki.de

Henning Funke
TU Dortmund University
Germany
henning.funke@tu-dortmund.de

Jens Teubner
TU Dortmund University
Germany
jens.teubner@cs.tu-dortmund.de

ABSTRACT

Technology limitations are making the use of *heterogeneous computing devices* much more than an academic curiosity. In fact, the use of such devices is widely acknowledged to be the only promising way to achieve application-speedups that users urgently need and expect. However, building a robust and efficient query engine for heterogeneous co-processor environments is still a significant challenge.

In this paper, we identify two effects that limit performance in case co-processor resources become scarce. *Cache thrashing* occurs when the working set of queries does not fit into the co-processor's data cache, resulting in performance degradations up to a factor of 24. *Heap contention* occurs when multiple operators run in parallel on a co-processor and when their accumulated memory footprint exceeds the main memory capacity of the co-processor, slowing down query execution by up to a factor of six.

We propose solutions for both effects. *Data-driven operator placement* avoids data movements when they might be harmful; *query chopping* limits co-processor memory usage and thus avoids contention. The combined approach—*data-driven query chopping*—achieves robust and scalable performance on co-processors. We validate our proposal with our open-source GPU-accelerated database engine CoGaDB and the popular star schema and TPC-H benchmarks.

1. INTRODUCTION

Modern processors hit the *power wall*, which forces vendors to optimize a processor's performance within a certain energy budget [6]. As a consequence, processor manufactures started to specialize processors. Thus, experts predict that future machines will consist of a set of heterogeneous processors, where each processor is optimized for a certain application scenario [6]. This trend has already become commodity, e.g., in the form of graphics processors (GPUs), many integrated cores architectures (MICs), or field-programmable gate arrays (FPGAs). Not taking advantage of them for query processing means to leave available resources unused.

[*]Work done when author was working at TU Dortmund.

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 ACM 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.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882936

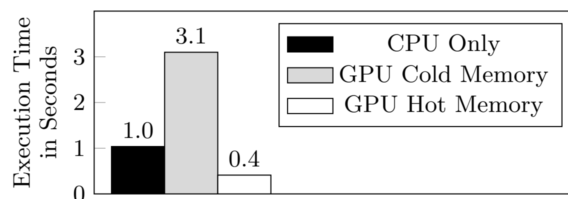


Figure 1: Impact of different query execution strategies on performance of a star schema benchmark query (Q3.3) on a database of scale factor 20. Using a GPU slows the system down in case input data is not cached on the GPU.

The potential of heterogeneous systems is often limited by the capacity of the communication channel between the co-processor and its host system [11]. We illustrate this problem in Figure 1 for a GPU-based co-processor, which we use as a poster child in the context of this work.

We obtained this figure by executing Query 3.3 from the Star Schema Benchmark *(a)* on a commodity CPU; *(b)* using a GPU accelerator, assuming a *cold-cache* scenario (i.e., all data has to be transferred to the GPU before an operator starts, which is very likely for ad-hoc queries); and *(c)* using the GPU accelerator in a *hot-cache* setting (more details on our experimentation platform follow later in the text). Clearly, while a GPU co-processor has the potential to speed up query execution by a factor of 2.5 (consistent with earlier results on GPU-accelerated query processing [13]), data transfer costs turn the situation into a performance degradation by more than a factor of three.

Techniques for co-processor acceleration typically make two assumptions:

1. The input data is cached on the co-processor and the working set fits into the co-processor memory.
2. Concurrent queries do not access the co-processor simultaneously.

Co-processors can slow down query processing significantly if these assumptions are violated, which is especially likely for ad-hoc queries. The goal of this paper is to understand *why* performance degrades under realistic conditions and *how* we can automatically detect when co-processors will slow down the DBMS (e.g., for many ad-hoc queries) and when co-processors will improve performance (e.g., for re-occuring or ad-hoc queries accessing the data cached on the co-processor).

In this work, we identify two problems: *cache thrashing* and *heap contention*. *Cache thrashing* occurs if the working set of the DBMS does not fit into the memory of a co-processor (assumption 1 violated), causing expensive evictions and re-caching. *Heap contention* is the situation where too many

operators use the co-processor in parallel (assumption 2 violated), so their combined heap memory demand exceeds the device's memory capacity.

We show how existing techniques from other domains can mitigate the problem and thus achieve robust query processing in heterogeneous systems. First, we *place data before query execution*, according to the workload pattern; we assign operators only to (co-)processors where necessary data is already cached, thus avoiding the cache thrashing problem. Second, we defer *operator placement* to the query execution time, so we can dynamically react to faults. This way, we can react to out-of-memory situations and limit heap space usage to avoid heap contention. Third, we limit the number of parallel running operators on a co-processor to reduce the likelihood that *heap contention* occurs. The main benefit of our approaches lies in optimizing the worst-case execution time and thus, make it feasible to use co-processors in practice.

To validate the effect of the strategies, we provide a detailed experimental evaluation based on our open-source database engine *CoGaDB*[1]. CoGaDB is a from-scratch implementation of a column-oriented in-memory database engine, with GPU support from the ground.

The remainder of the paper is structured as follows. In Section 2, we present the state of the art on co-processing in databases. Then, we discuss our techniques: data-driven operator placement in Section 3, run-time placement in Section 4, and the query chopping technique in Section 5. Then, we present our extensive experimental evaluation in Section 6. Finally, we present related work in Section 7 and conclude in Section 8.

2. STATE OF THE ART

In this section, we provide a brief overview of the basics of co-processors, their resource limitations, the operator placement problem, and the evaluation system CoGaDB.

2.1 Co-Processors

Compared to CPUs, today's co-processors are specialized processors spending more chip space to light-weighted cores than on control logic such as branch prediction and pipelining [34]. In order to feed these cores, co-processors need high-bandwidth memory, which in turn means that the total memory capacity is relatively small (up to 16 GBs for high end GPUs or Xeon Phis) to achieve reasonable monetary costs. Since co-processors are separate processors, they are usually connected to the CPU by the PCIe bus, which often ends up being the bottleneck in CPU/co-processor systems [11].

A common optimization is to use part of the co-processor memory as cache, which can reduce the data volume that needs to be transferred to the co-processor. The remaining part of the co-processor's memory is used as heap for intermediate data structures and results. Naturally, the caching strategy cannot avoid the cost of moving results back from the co-processor to the host system.

2.2 Resource Limitations

The limited resource capacity of co-processors poses a major problem for database operators. For instance, the probability that an operator runs out of memory on a co-processor is significantly higher compared to a CPU operator. In case memory becomes scarce, operators will fail resource allocations and have to abort and clean up.

[1]cogadb.cs.tu-dortmund.de/wordpress/download

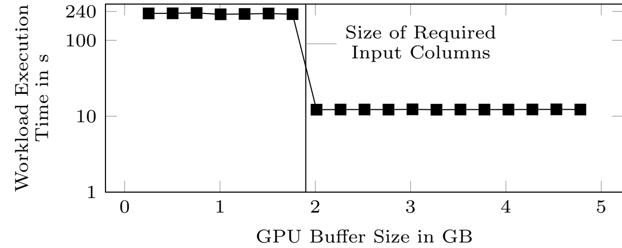


Figure 2: Execution time of selection workload. The performance degrades by a factor of 24 if not all input columns fit into the cache.

To be practically useful, database engines must have a mechanism to deal with such fault situations. Previous work [35] did so by aborting and re-starting the entire query. However, such simplistic solutions will hardly meet user expectations in realistic settings. As the use of GPU acceleration proliferates (and more users share the co-processor), the probability for resource contention sharply increases, resulting in starvation and other problems.

CoGaDB offers a more sophisticated mechanism for fault handling. In case of resource exhaustion, the system will have to re-start only the single failed operator. This way, we avoid losing already computed results and continue query processing immediately.

2.3 Query Processing

Most approaches for co-processor acceleration assume that first, the input data fits into the co-processor cache and second, no concurrent queries access the co-processor simultaneously. Next, we show the performance penalties that occur if these assumptions are not met, which is more the norm than the exception.

Cache Thrashing Figure 2 shows the query execution time of a workload of selection operators on the GPU with varying GPU buffer size. The workload consists of eight selections, which filter on eight different columns. Furthermore, the selections are executed interleaved, a common scenario in databases where different queries access different data. As input data, we chose the fact table of the star schema benchmark (scale factor 10). We provide the detailed workload in Section B.1. All required input columns have a total size of 1.9 GB. In the case that not all input columns fit into the co-processor's data cache, we observe a performance degradation of a factor of 24 due to *cache thrashing*. Caches commonly use a least-recently-used strategy, a least-frequently-used strategy or variants of the two strategies. For both strategies, at least one column needs to be evicted from the cache if the cache size is smaller than the accumulated memory footprint of required input columns (1.9 GB). Evicting the least recently used column practically makes the cache useless in case the memory footprint exceeds the cache size. This is because the evicted columns will be accessed by the next query of the workload (cf. Appendix B.1), which results in a copy operation.

Heap Contention Even if we solve the cache thrashing problem, we run into a similar effect in case we execute operators in parallel. We illustrate the problem with a selection workload on a GPU with increasing number of parallel users. The selection queries require four different operators to be executed consecutively to compute the result. All selections filter the same input columns to avoid the cache-trashing effect. The workload is fixed and consists of 100 queries,

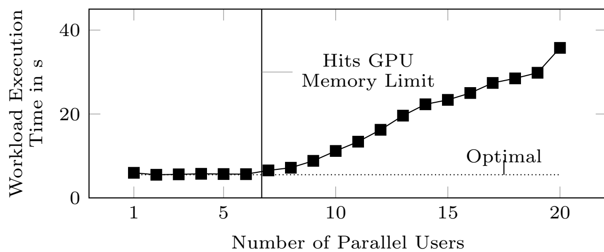


Figure 3: Execution time of a selection workload. With increasing parallelism, more operators allocate memory on the GPU, which leads to performance degradation when memory capacity is exceeded.

but we increase the number of parallel user sessions that execute the workload. We discuss the detailed workload in Appendix B.2.

Since all workloads contain the same amount of work, an ideal system could execute all workloads in the same time. Figure 3 shows the actual effect on our execution platform. Clearly visible is a performance degradation of up to factor six compared to a single user execution.

What we observe here is that the intermediate data structures of the parallel running operators exceed the co-processor's memory when seven or more users use the graphics processor concurrently (assumption 2 violated), which causes operators to run out of memory.

Once this happens, operator execution will be aborted on the GPU and moved to another processor instead, which increases the IO on the bus and degrades performance. We call this effect *heap contention*.

It is clearly visible that co-processors can slow down database query processing significantly, if we have no additional mechanism. To achieve robust query processing, we need to ensure that co-processors will never slow down the DBMS and that co-processors improve performance with increasing fraction of the working set that fits into their memory.

2.4 Operator Placement

All co-processor-accelerated database systems either process all queries on the co-processor (e.g., GPUDB [37], Multi-Qx-GPU [35], and Red Fox [36]), or try to balance the processing tasks between processors (e.g., CoGaDB, GPUQP [13], MapD [24]) by performing *operator placement* (i.e., a complete query plan is analyzed and each operator is assigned to a processor). Both strategies use the same principles: First, they create a query plan that is fixed during query execution. Second, the execution engine is responsible to transfer data to the processors where the operators are executed. Since data movement can be very expensive, this strategy is combined with data caching on co-processors.

2.5 Evaluation System: CoGaDB

CoGaDB is a main memory database system using a column-oriented data layout, has an SQL interface, and a Selinger-style optimizer. Similar to other OLAP DBMSs such as MonetDB [17], CoGaDB is an interpreter-based engine and processes queries by calling an operator-specific function for each operator in the query execution plan. The operator functions consume the complete input and materialize their output. Thus, CoGaDB does not employ pipelining. In case an operator contains two or more child operators, the child operators are evaluated in parallel (inter-operator parallelism). CoGaDB makes use of all CPU cores (or GPU cores, respectively) to speed up query processing (intra-operator parallelism). Furthermore, CoGaDB uses the hardware-oblivious query optimizer HyPE to solve the operator placement problem [7, 9]. We compare the performance of CoGaDB to the GPU-accelerated database engine MonetDB/Ocelot in Section A.

2.5.1 Fault Tolerance

In error situations (e.g., an operator runs out of memory), CoGaDB restarts the operator on a CPU. This is in contrast to the earlier system of Wang and others, which handled out-of-memory situations by aborting entire queries [35]. Thus, to cope with the resource restriction of GPUs, we provide a CPU-based fallback handler for every operator. This way, query processing can always continue though at a two-fold cost: First, the aborted operator will run slower than anticipated; second, the query processor performs more data transfer operations than expected.

Note that it is not practical to use a wait-and-admit approach for memory allocation. Since we cannot provide a concise upper bound for many database operators (e.g., joins), we are forced to allocate memory in several steps and hold onto already allocated memory. If this is done in parallel for multiple operators, the DBMS will run into deadlocks. Thus, CoGaDB aborts operators immediately if an allocation fails to avoid the overhead of deadlock detection.

2.5.2 Operator Placement

He and others early recognized the importance of operator placement and data movement [13]. To address the problem, they performed backtracking for sub-query plans and combined optimal sub plans to the final query plan [13]. From our experience in CoGaDB, this backtracking approach can be very time consuming. Thus, CoGaDB uses an iterative refinement optimizer that only considers query plans where multiple successive operators are executed on the same processor, a technique that already proved very useful in distributed databases because it limits communication between processors/nodes. We use this established approach for query optimization as a baseline for the experiments in the remainder of this paper.

2.5.3 Performance Optimizations for Data Transfer

CoGaDB implements the recommended optimizations to reduce data transfer bottlenecks [1]. It uses asynchronous data transfers via CUDA streams, which is required to get the full PCIe bus bandwidth and allows for parallel data transfer and computation on GPUs. For this, it is required to copy data into page-locked host memory as staging area, which cannot be swapped to disk by the OS. *Unified Virtual Addressing* (UVA) performs data transfers implicitly and transparently to the application but pays the same data transfer cost as manual data placement.

3. DATA-DRIVEN OPERATOR PLACEMENT

All of the heuristics in Section 2.5.2 assume that data placement is *operator-driven*. That is, for each operator the optimizer first decides on a processor, then—if necessary—moves required data to that processor. In case the hot data of the workload does not fit in the co-processor's data cache, the system runs into the cache thrashing effect.

In this section, we discuss how we can completely avoid the cache thrashing effect by first, deciding on a *data placement* and second, perform *operator placement* according to the data placement. For this, we analyze the workloads access pattern and place the most frequently accessed data in the co-

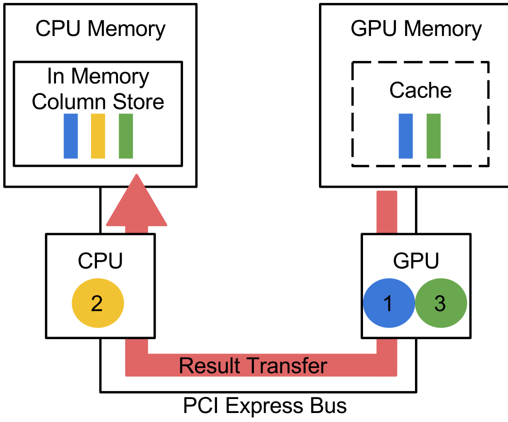


Figure 4: Principle of Data-Driven operator placement. Operators 1 and 3 are pushed to the cached data on the co-processor, whereas the data of operator 2 is not cached and must run on a CPU.

processor cache. Then, we place operators on the co-processor if and only if their input data is cached. Consequently, excessive evictions and re-caching cannot happen because the data placement is decided by one central component: the data placement manager.

3.1 Data-Driven: Push Operators to Data

As an alternative strategy to operator-driven data placement, we propose a *Data-Driven Operator Placement* (in short *Data-Driven*). The idea is that a storage adviser pins frequently used access structures to the co-processor's data cache and the query processor automatically places operators on the co-processor, if and only if the input data is available on the co-processor. Otherwise, the query processor executes the operator on a CPU. This strategy assumes that operators run faster on the co-processor if the data is cached on the co-processor. However, this is consistent with many studies (e.g., He and others [13]) and with our experiments in Section 6 and Section A in the appendix. In case the input data is *not* cached, join operators running on a co-processor can outperform their CPU counterparts, but most operators running on co-processors will not [13]. We illustrate *Data-Driven* in Figure 4. Here, we have three operators, where the input data of operators 1 and 3 are cached on the co-processor, whereas the input data of operator 2 is not. Therefore, operators 1 and 3 are pushed to the co-processor, whereas operator 2 must be executed on a CPU. This is similar to data-oriented transaction execution [29], where each processor core is responsible for processing transactions on a certain database partition. Similarly, we pin certain database access structures such as columns to the co-processor to profit from a perfect cache hit rate.

3.2 Automatic Data Placement

When we only execute operators on a co-processor in case their input is cached on that co-processor, then we need a background job that analyzes the access patterns of the query workload and automatically places frequently required access structures on a co-processor. We keep the data-placement strategy simple and use a least-frequently-used strategy. For this, the storage manager keeps statistics about how frequently and how recently access structures were used by the query processor. Each column in the database has an access counter, which is incremented each time an operator accesses a column. We show the cache management in Algorithm 1. Periodically, we fill the cache with the most frequently queried columns and access structures. This strategy commonly caches join columns and frequently used filter columns of dimension tables. Note that LRU and LFU strategies perform similarly well, as we show in Section E. Note that running queries can continue execution when the background job adjusts the data placement. We use reference counters for access structures to determine when they are no longer in use and can clean up evicted data when it is no longer used during query processing. Furthermore, we use fine-grained latching to avoid that running queries block when accessing the cache.

Algorithm 1 Data placement updates the coprocessor's cache content using the access counts from query processing.

Input: Columns C, Cache State CS
1: $K = \text{sort}_{\text{access_count,descending}}(C)$
2: used_buffer=0
3: $C_{cacheold} = CS$.cached_columns
4: $C_{cachenew} = \emptyset$
5: **for** $i = 1$; used_buffer + K_i.size $\leq CS$.buffer_size and $i < |K|$; $i++$ **do**
6: $\quad C_{cachenew} = C_{cachenew} \cup K_i$
7: $\quad$ used_buffer+=K_i.size
8: **end for**
9: CS.evict($C_{cacheold} \setminus C_{cachenew}$)
10: CS.cache($C_{cachenew} \setminus C_{cacheold}$)

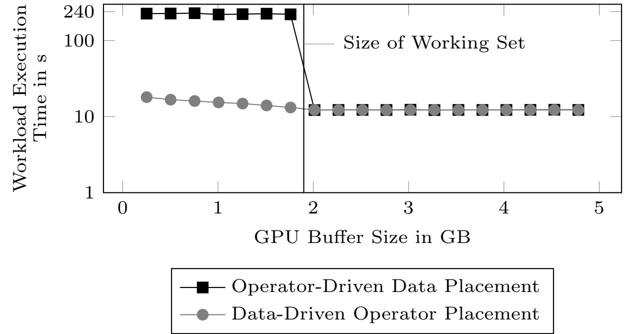


Figure 5: Execution time of selection workload. The performance degradation can be avoided by data-driven operator placement.

3.3 Query Processing

A consequence of *Data-Driven* is that operators are automatically chained (e.g., consecutive operators are executed on the same processor) from the leaf operators, until an n-ary operator ($n \geq 1$) is found where at least one input column is not available in the co-processor memory. Then, the operator chain is not continued and the remaining part of the query is processed on a CPU. This is because *Data-Driven* requires that *all* input columns are resident in the co-processor memory. *Data-Driven* can process the complete query on a co-processor in case the memory capacity is sufficient. If not all input data fits in the co-processor's memory, the co-processor is only used to the degree where the input data fits in the co-processor's memory, avoiding delays by transfer operations and ensuring graceful degradation.

We illustrate the behavior of *Data-Driven* on our two problem cases from the introduction in Figure 5 and 7, respectively. *Data-Driven* eliminates the performance degradation,

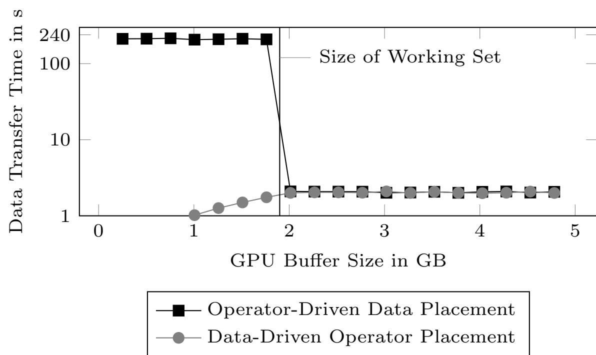


Figure 6: Time spent on data transfers in the selection workload.

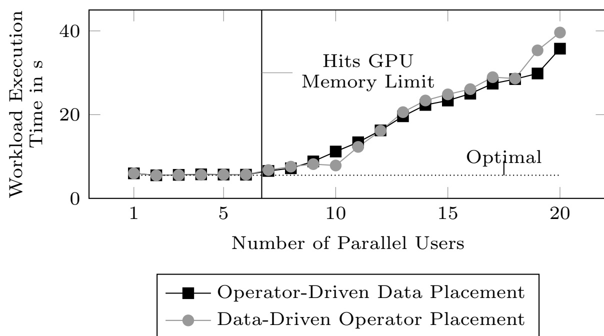


Figure 7: Execution time of a selection workload. *Data-Driven* has the same performance degradation as operator-driven data placement.

which we observed for the classic approach: operator-driven data placement. We show in Figure 6 that the performance degradation is caused by the enormous data transfer times caused by *cache thrashing*. However, with *Data-Driven* this cannot happen because the co-processor is not used for an operator, when its input data is not cached. We also observe that with increasing buffer size, *Data-Driven* gets faster with the number of input columns that fit into the cache, until it reaches the optimum.

3.4 Problems in Concurrent Workloads

The second problem case, where we observe performance degradation with an increasing number of concurrent operators on a co-processor, is not solved with *Data-Driven* as illustrated by Figure 7. This effect is caused by an increased data transfer overhead due to operator aborts on co-processors. In case too many operators run in parallel, their collective memory demand exceeds the co-processors memory capacity (*heap contention*).

In case of the parallel selection workload, we require a column C as input data. CoGaDB implements the GPU selection algorithm of He and others [13], which requires a memory footprint of 3.25 times the size of the input column. For n parallel queries, a column size C of 218 MB (fact table columns of star schema benchmark for scale factor 10), and a GPU memory capacity M of $5GB$, we can execute $n = \frac{|M|}{3.25 \cdot |C|} \approx 7$ parallel users without running in the memory limit. This is exactly the point where the performance starts to degrade in Figure 7.

In case of more than 7 parallel queries, some operators run out of memory and are restarted on a CPU. Since the operator placement decisions were all done during *query*

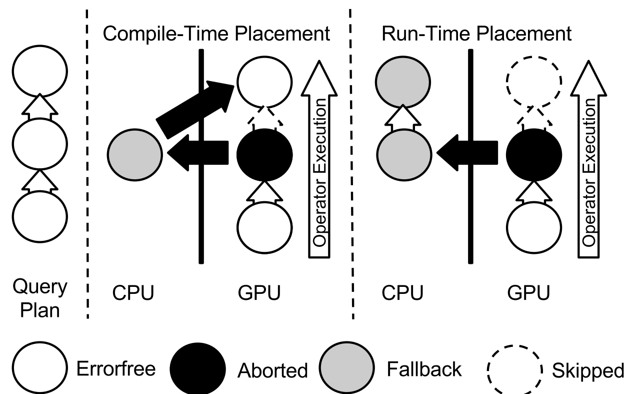


Figure 8: Flexibility of run-time placement. Compile-time heuristics force the query processor to switch back to the GPU after an operator aborts, whereas run-time heuristics avoid this overhead.

compile-time, the successor operator is still executed on the co-processor, which causes additional data transfers that were not anticipated by the optimizer. We discuss how we can solve this issue by performing operator placement at query run-time in the next section.

4. RUN-TIME OPERATOR PLACEMENT

Until now, we discussed heuristics for the operator placement problem applied at query *compile-time*, i.e., they decide on a fixed operator placement *before* a query runs. This strategy has three drawbacks:

1. It cannot predict error situations such as out of memory scenarios, where a co-processor operator needs to abort.
2. Compile-time heuristics need to rely on sufficiently accurate cardinality estimates to estimate the data transfer volume. However, it is still very difficult to provide cardinality estimations.
3. GPU code is particularly sensitive to environment parameters such as current load or usage of heap memory. Inherently, those parameters cannot be known before the actual execution time—the classical dilemma of multi-query optimization.

A way to escape this dilemma has been proposed by Boncz and others [4]. By separating query optimization into strategical and tactical decisions, many important runtime parameters can be considered for the optimization process. Whereas the database optimizer performs the strategic optimization (e.g., the structure of the query plan), a run-time optimizer conducts the tactical optimization (e.g., operator placement and algorithm selection).[2] Therefore, run-time placement can dynamically react to unforeseen events (e.g., out of memory conditions) and does not need any cardinality estimates, because it performs operator placement after all input relations are available. In this section, we discuss how run-time operator placement helps a query processor to react to faults during operator execution on co-processors.

4.1 Run-Time Flexibility

If the optimizer performs operator placement decisions at run-time, it can dynamically react to unforeseen events, such as aborting co-processor operators. Since memory is scarce in co-processors, there is always the possibility that an operator cannot allocate enough memory. In this case, CoGaDB

[2]This separation requires the DBMS to use operator- or vector-at-a-time processing, which is the case for CoGaDB.

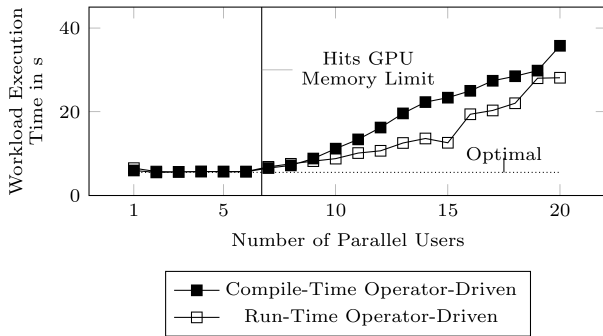


Figure 9: Run-time operator placement improves performance, but does not achieve the optimum.

discards the work of the operator and restarts the operator on the CPU. This mechanism is very efficient, because operators typically start with the allocation of memory for their input data and data structures. So most of the time, co-processor operators abort without wasting any resources.

Run-time resource contention interacts poorly with compile-time operator placement. We illustrate the problem in Figure 8. We assume a simple query execution plan that is placed completely on the GPU. The second operator runs out of memory and aborts, so CoGaDB creates a fall back operator and executes it on the CPU. The problem is that the third operator is still placed on the GPU, and requires to copy the result data to the GPU. In contrast, a run-time placement heuristic schedules the third operator, after the second operator aborted, and hence, places the third operator on the CPU to avoid the copy cost. Note that this procedure can repeat itself multiple times in the same query, and may cause large performance degradations.

4.2 Query Processing

We illustrate in Figure 9 that run-time operator placement reduces the performance penalty of concurrent running queries of up to a factor of two. However, run-time placement is still more than two times slower than the optimal case. The reason for this is that run-time placement avoids data transfer overhead in case of operator aborts. Instead, the aborted operators are restarted and lose their co-processor acceleration, so we still pay a performance penalty.

5. MINIMIZING OPERATOR ABORTS

In this section, we discuss how we can reduce the overhead of running multiple queries in parallel by reducing the probability that operators abort. Based on this, we present our technique *query chopping*, which reduces the maximum number of operators that can run in parallel by using the thread pool pattern.

5.1 Probability of Operator Aborts

To avoid operator aborts, we reduce the probability that an operator runs out of memory. One way to achieve this is to prohibit the DBMS to execute multiple GPU operators concurrently (e.g., He and others [13]). However, Wang and others showed that we can improve the performance of query processing by allowing moderate parallel execution [35]. Wang and others proposed to use an admission control mechanism for queries, limiting the total number of queries in the DBMS and hence, the number of parallel queries that concurrently access the co-processor. While we cannot use this approach on an operator granularity, we can put an upper bound on the number of operators concurrently executing on a co-processor by using the thread pool pattern. Here, processing tasks are not *pushed* towards the processor, but *pulled* by the processor. If no worker thread is available for an operator, it is kept in a queue, until a prior operator finishes execution. This way, we do not artificially limit the number of concurrent queries in the DBMS, but still avoid that queries run into resource contention problems due to the use of co-processors.

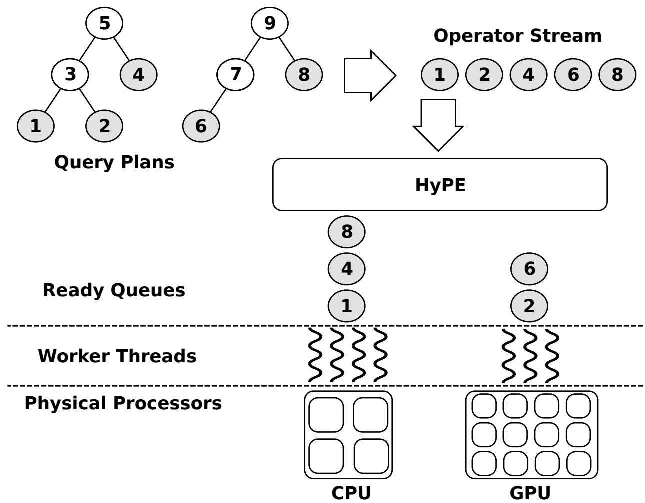


Figure 10: Query Chopping.

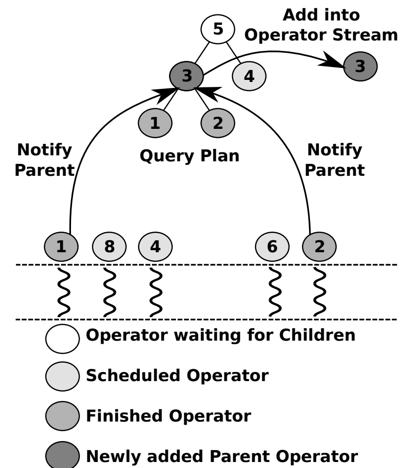


Figure 11: Query Chopping: Finished operators pull their parents into the global operator stream.

5.2 Query Chopping

Based on our discussions, we now present our novel technique *query chopping* (*Chopping*), which is essentially a progressive query optimizer that performs operator placement at query run-time and limits the number of operators executing in parallel via a thread pool.

Our goal is to accelerate a workload of queries by using the available (co-)processors, but avoid that these accelerators can slow down query processing. We implemented *Chopping* as an additional layer between the strategic optimizer of CoGaDB and the hardware oblivious query optimizer HyPE. *Chopping* takes n queries, chops off their leaf operators, and inserts them into a global operator stream. Since the

leaf operators have no dependencies, they can immediately start their execution independent of each other. HyPE then schedules the operators on the available processors, and selects for each operator a suitable algorithm. We summarize our discussion in Figure 10.

When an operator finishes execution, it notifies its parent. After all children of an operator completed, the operator inserts itself into the global operator stream. Once the root operator of a query plan has finished execution, the query result is returned. We illustrate this procedure in Figure 11.

This technique works for single-query and multi-query workloads. The virtue of the strategy is that we always know the exact input cardinalities during the tactical optimization, which increases the accuracy of HyPE's cost models and, hence, its operator placement and algorithm selection. Furthermore, we can fully benefit from HyPE's load balancing capabilities.

Additionally, *Chopping* manages the concurrency on a processor at the operator level. HyPE's execution engine virtualizes the physical processors, e.g., we can create multiple worker threads per processor to achieve inter-operator parallelism. For each physical processor, we maintain a *ready queue*, where all worker threads of that processor pull new operators from. This mechanism can also be used to decide on the number of threads for a single operator. In a workload with low concurrency, an algorithm could decide to use more threads to execute faster, whereas in highly parallel workloads, each operator would use only a single thread. To balance the load between CPU and GPU, we keep track of the load on each processor by estimating the completion time of each processor's ready queue.

A thread pool avoids that operators run out of memory by accumulated memory usage, but does not guarantee that single operators get enough heap memory. However, the heap memory usage of operators directly corresponds to the database size. When the input columns exceed a certain memory capacity, the operators cannot allocate enough memory for intermediate data structures and results, which forces the system to process the workload on CPU only. Therefore, a thread pool is sufficient to solve the heap contention problem.

Chopping puts only an upper bound to the concurrency of operators: It is up to the operating system or the co-processor's driver when and how many operators are executed in parallel. Thus, *Chopping* steers the parallelism on all processors and leaves it to the scheduling mechanisms of the OS to place threads on a certain CPU socket to a specific core. Thus, *Chopping* seamlessly integrates with existing approaches.

5.3 Query Processing

We illustrate in Figure 12 that *Chopping* achieves near optimal performance. This is because *Chopping* also limits the number of operators that can run concurrently on a co-processor. This significantly reduces the probability that operators run into the heap contention effect and need to abort. We illustrate this in Figure 13. It is clearly visible that operator-driven data placement at compile-time leads to the most operator aborts. Simple run-time placement reduces this overhead, as it continues query processing on the CPU if an operator aborts, thus relieving the GPU heap. If we additionally limit the inter-operator parallelism we achieve near optimal results.

5.4 Data-Driven Query Chopping

We now discuss how *Data-Driven* and *Chopping* work together. The combined strategy places frequently used

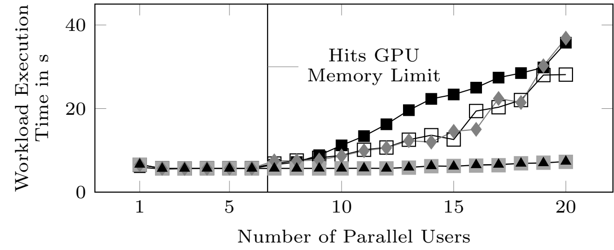


Figure 12: Dynamic reaction to faults and limiting the number of parallel running GPU operators achieves near optimal performance.

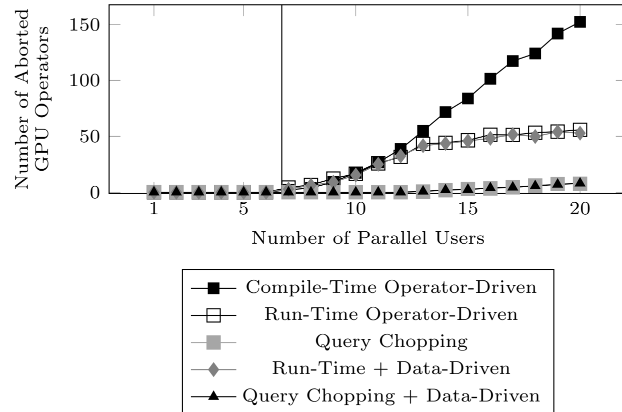


Figure 13: Run-time placement reduces heap contention by continuing execution in the CPU. *Chopping* limits the number of parallel running GPU operators and further decreases the abort probability.

access structures in the co-processor's data cache using a periodic background job to avoid cache thrashing. Then, starting from the leaf operators of the query, all operators are pushed on the co-processor if the input data of an operator is cached and on the CPU otherwise. The operator is then put into the ready queue of the selected processor. After a worker thread executed an operator, it notifies its completion to the parent operator and fetches the next operator from the ready queue. The parent operator is processed in the same way, when all child operators completed execution. This procedure is continued until all operators of the query finished. The trick is that when operator aborts are detected, query processing continues on the CPU, because the output data no longer resides on the co-processor. Thus, the combined strategy avoids memory thrashing and heap contention.

5.5 Applicability To Other Processing Models

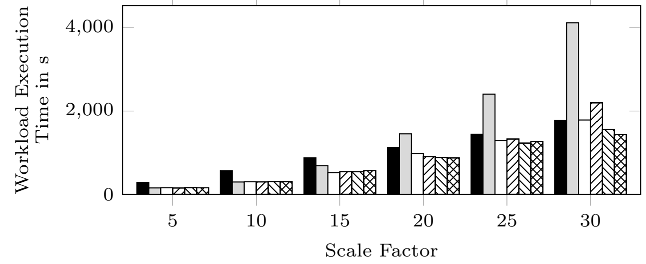
Aside from operator-at-a-time processing, two alternative processing models have been proposed in the past: vector-a-time processing and query compilation. An inherent property of many real world queries is that they contain multiple *pipeline breakers*, which force the DBMS to materialize intermediate results, regardless of the processing model. In this cases, cache thrashing and heap contention can lead to the same performance penalties observed in this paper.

Vectorized Execution. Vector-at-a-time processing works on cache-resident chunks of columns (vectors), avoids materialization cost of operator-at-a-time processing, and supports pipelining [5]. Support of *Data-Driven* is straightforward: Cached vectors would be processed on the co-processors and the remaining vectors on the CPU. In case the co-processor

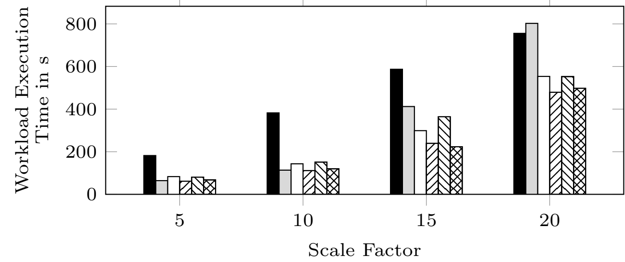
finished its cached vectors, we can exploit an additional optimization: The vector-at-a-time scheme can overlap data transfer and computation on the co-processor, which allows to process vectors on all processors (e.g., Chen and others [10]). We expect that such a cross-processor database engine would significantly boost performance, but the observations in this paper are still valid in such a system. Heap contention is reduced to pipeline-breaking operators, but for a reasonable complex query workloads the DBMS is still required to deal with this problem.

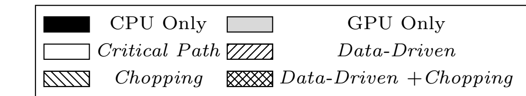
Compiled Execution. Query compilation generates C or assembly code for a specific query and executes the compiled program [21, 27]. Operators that can be pipelined are compiled to a single function (called pipeline), which typically processes data in a single pass. Thus, intermediate results are only materialized for pipeline breakers [27]. These pipelines then need to be placed on a (co-)processor. In this scenario, *Data-Driven Chopping* is still required to avoid cache trashing and heap contention. An extension of the morsel framework [22] would lead to the option of streaming data to co-processors as it is a vector-at-a-time processing using compiled pipelines. Therefore, the reasoning for vector-at-a-time execution applies for query compilation as well.

Summary. In summary, the cache thrashing and heap contention effects are inherent to all database engines regardless of their processing model. Therefore, our discussed approaches are useful in all co-processor-accelerated database engines.



(a) Workload of SSBM queries.





(b) Workload of selected TPC-H queries.

Figure 14: Average workload execution time of SSBM and selected TPC-H Queries. *Data-Driven* combined with *Chopping* can improve performance significantly and is never slower than any other heuristic.

6. EFFECTS ON FULL WORKLOAD

In this section, we quantitatively assess the performance of our proposed approaches on two more complex workloads: the star schema benchmark and the TPC-H benchmark.

6.1 Experimental Setup

As evaluation platform, we use a machine with an Intel Xeon CPU E5-1607 v2 with four cores @3.0 GHz (Ivy Bridge), 32 GB main memory, and an NVIDIA GTX 770 GPU with 4 GB of device memory. On the software side, we use Ubuntu 14.04.2 (64 Bit) as operating system and the NVIDIA CUDA driver 331.113 (CUDA 5.5). Before starting the benchmark, we pre-load the database into main memory and access structures in the GPU memory, until the GPU buffer size is reached.

For each SSBM workload, we run all SSBM queries (Q1.1-Q4.3). In case of the TPC-H workload, we run a subset of the queries (Q2-Q7). The remaining TPC-H queries are not fully supported in the current version of CoGaDB. We elaborate details about the benchmarks and the selection of queries in Appendix C. For each experiment, we run a workload two times to warm up the system. Then, we run the workload 100 times, and show the execution time of the workload and the time to transfer data over the PCIe bus. We focus our discussions on *Data-Driven*, *Chopping*, and *Data-Driven Chopping*.

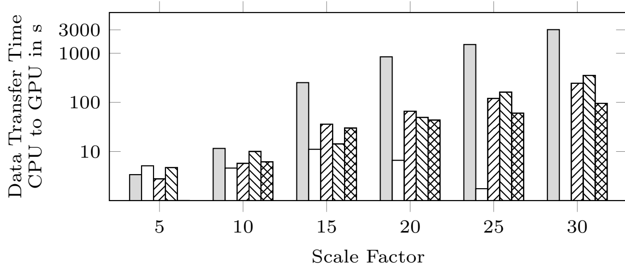
6.2 Detailed Experiments

We validate our proposed heuristics on the SSBM and TPC-H benchmark in terms of performance and caused IO on the PCIe bus. We investigate both effects—cache thrashing when a working set exceeds the co-processor cache and heap contention by excessive inter-operator parallelism—on the SSBM and TPC-H benchmark. For this, we perform one experiment per effect. We conduct an experiment where we increase the scale factor of both benchmarks (single user) and measure performance and transfer times for inter-processor communication to investigate the cache thrashing effect. To understand how different execution and placement strategies react to heap contention situations, we conduct an experiment where we increase the number of concurrently running queries of both benchmarks and measure workload execution time and transfer times for inter-processor communication for a scale factor of 10.

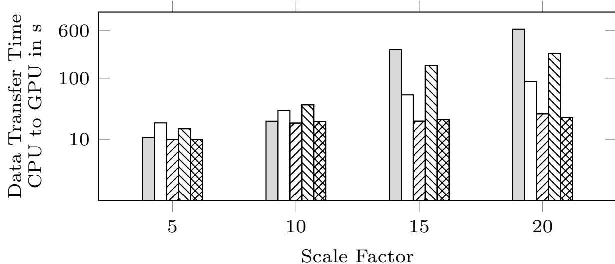
As reference points, we included two heuristics for operator-driven data placement at compile-time to reflect the state of the art. The *GPU Preferred* heuristic executes all operators on the GPU, and only switches back to the CPU in case an operator runs out of memory. The *Critical Path* is the default iterative refinement optimizer of CoGaDB, which creates a hybrid CPU/GPU plan with the lowest response time. We describe *Critical Path* in detail in Appendix D. These heuristics all use operator-driven data placement at compile time. We compare these heuristics to the three basic variants of our proposal: *Data-Driven* at compile-time, *Chopping* with operator-driven data placement, and *Data-Driven Chopping*.

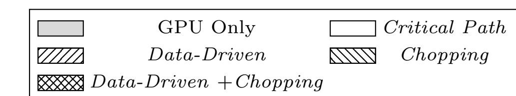
6.2.1 Scaling Database Size

To understand how different execution and placement strategies react to cache thrashing situations, we scale up the SSBM and TPC-H databases to increase the memory requirements of the working set. We show the results in Figure 14 for the SSBM and TPC-H benchmark. It is clearly visible that a GPU-only execution is not suitable for growing database sizes, as it is inferior to the remaining strategies. The reason is that a large portion of the execution time is spend on data



(a) Workload of SSBM queries.





(b) Workload of selected TPC-H queries.

Figure 15: Average data transfer time CPU to GPU of SSBM and selected TPC-H Queries. *Data-Driven* combined with *Chopping* saves the most IO.

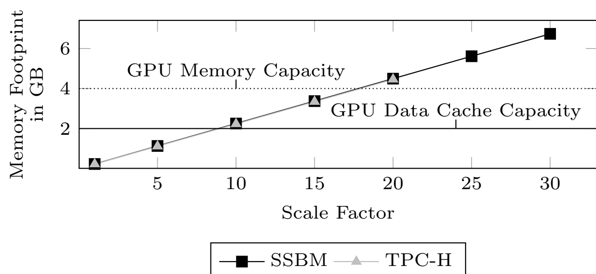


Figure 16: Memory footprint of workloads

transfers from CPU to GPU as illustrated by Figure 15. The performance of the GPU-only approach falls behind starting at scale factor 15 (SSBM and TPC-H). Figure 16 shows the memory footprint of the SSBM and TPC-H Workload. Starting from scale factor 15, it significantly exceeds the data cache, and thus, clearly shows the cache-thrashing effect.

As expected from our observations in the selection workload, *Data-Driven* and *Chopping* reduce the workload execution time compared to the GPU-only approach, however, *Data-Driven* can still slow down performance compared to a CPU-only approach. The combined *Data-Driven Chopping* approach can improve performance even when resources become scarce, and never performs worse than a CPU-only approach. Thus, *Data-Driven Chopping* fulfills our requirements for robust query processing.

Data-Driven saves data transfers from the CPU to the GPU, because *Chopping* runs into the cache thrashing effect while *Data-Driven* avoids this overhead. We can observe this effect in the increased copy times of *Chopping* compared to *Data-Driven* in the TPC-H workload (cf. Figure 15(b)). At the same time, the data transfer time from GPU to CPU is larger for *Data-Driven* compared with *Chopping*, because *Data-Driven* alone cannot react to aborted operators.

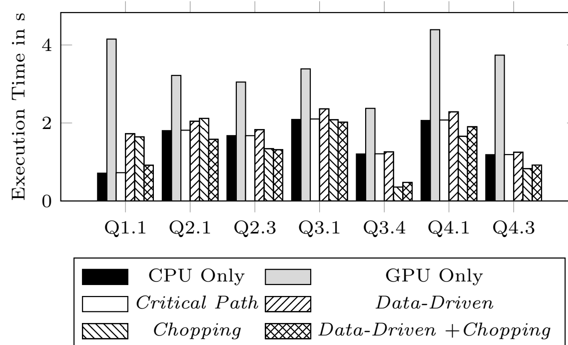


Figure 17: Query execution times for selected SSBM queries for a single user and a database of scale factor 30.

According to the savings of IO time of *Chopping* compared with *Data-Driven*, we conclude that the increasing database size can also lead to the heap contention effect, where operators need to abort because they run out of heap memory. This situation is difficult to predict and *Chopping* provides a simple and cheap error handling.

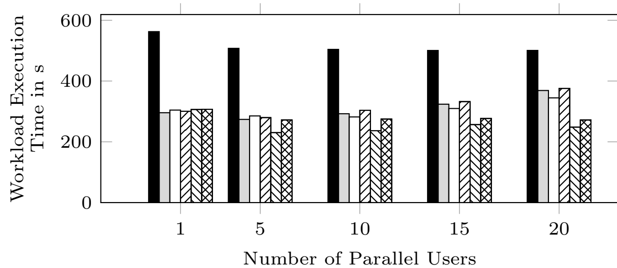
To get a more detailed understanding of what happens when memory resources become scarce, we investigate the query run-time of selected SSBM queries at scale factor 30 and illustrate them in Figure 17. The GPU-Only approach slows down each query. *Critical Path* is always as fast as the CPU-Only approach, because it detected the performance degradation due to the co-processor and only uses the CPU. For low selectivity queries (Q1.1, Q2.1, Q3.1, Q4.1), *Data-Driven Chopping* has little impact on performance. However, for high selectivity queries (Q2.3, Q3.4, Q4.3) we observe a performance improvement of up to factor 2.5 (Q3.4). A detailed examination of the query plans revealed that the (pushed-down) selections are put on the GPU, and frequently the first join, which accelerates the query. Since the intermediate results are small, it is cheap to switch back to the CPU in case a required input column is not cached (e.g., a join column). The reason low selectivity queries cannot be accelerated as well is twofold. First, it is more costly to switch back to the CPU and second, larger intermediate results increase the probability of memory scarcity and hence, operator aborts.

Overall, we conclude that of the compared strategies, the combination *Data-Driven Chopping* achieves the best performance (Figure 14) and minimal IO (Figure 15).

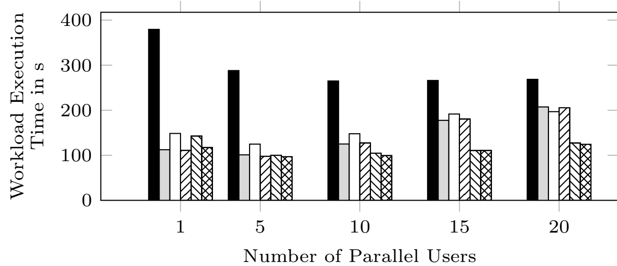
6.2.2 Scaling User Parallelism

To show the heap contention effect, we use a SSBM and a TPC-H database with fixed size (scale factor 10) and increase the number of parallel running queries (users). For each workload, we execute all queries 100 times. We repeat the workload multiple times and present the average execution time. Note that the total number of queries in the workload is fixed, only the number of parallel running queries changes.

Parallel query execution slows down query processing by a factor of 1.24 for the SSBM workload and by a factor of 1.85 for the TPC-H workload compared to a naive use of the GPU, as we show in Figure 18. Compared to a GPU Only execution, *Data-Driven Chopping* achieves a speedup by a factor of 1.36 for the SSBM and 1.66 for the TPC-H workload and uses significantly less resources.



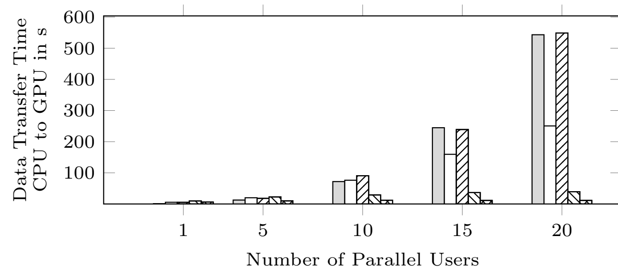
(a) Workload of SSBM queries.



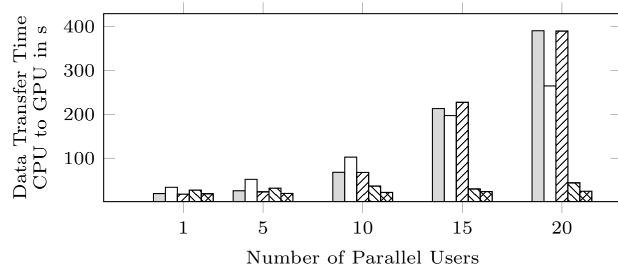
CPU Only | GPU Only | *Critical Path* | *Data-Driven* | *Chopping* | *Data-Driven +Chopping*

(b) Workload of selected TPC-H queries.

Figure 18: Average workload execution time of SSBM and TPC-H queries for varying parallel users. The dynamic reaction to faults of *Chopping* results in improved performance.



(a) Workload of SSBM queries



GPU Only | *Critical Path* | *Data-Driven* | *Chopping* | *Data-Driven +Chopping*

(b) Workload of selected TPC-H queries

Figure 19: Data transfer times CPU to GPU of SSBM and TPC-H workload for varying parallel users. *Chopping* reduces IO significantly especially with increasing number of parallel queries.

Chopping and *Data-Driven Chopping* reduce the required IO significantly—especially for workloads with many parallel users—as we show in Figure 19. *Data-Driven Chopping* reduces the time required for data transfers from CPU to GPU by a factor of 48 for the SSBM and 16 for the TPC-H workload. The main reason for the improved performance is the fine grained concurrency limitation of *Chopping*. However, run-time placement without parallelism control significantly reduces resource consumption as well.

To quantify the cost of aborted GPU operators, we measure the time from begin to abort of GPU operators and add them to a counter. We call this metric the *wasted time* and illustrate it for the SSBM experiments in Figure 20. *Chopping* and *Data-Driven Chopping* reduce the wasted time by up to a factor of 74. Note that both the copy time and wasted time can be larger than the overall workload execution time, because they reflect the total time, whereas the workload execution time represents the response time.

We show the execution time of selected SSBM queries for a workload serving 20 parallel users (cf. Figure 21). *Chopping* and *Data-Driven Chopping* are faster than the other heuristics for queries Q3.1, Q3.4, Q4.1 (up to a factor of 3.5 for Q3.4), competitive for queries 2.1, 2.3, 4.3, and slower for query Q1.1 (by a factor of 1.78 compared to GPU Only). As reference point, we use a GPU Only execution combined with an admission control mechanism that admits only a single query at a time. This way, intermediate results are consumed as fast as possible and heap contention is avoided. However, such admission control at query level increases query latencies (cf. Figure 21 and Figure 25 in the appendix). *Chopping* is either as fast as admission control (Q4.1 and Q4.2) or faster and improves query response times of up to a factor of 1.4. Compared to admission control, some queries are significantly faster using *Data-Driven Chopping* (Q3.4 by a factor of 3.6 and Q4.1 by a factor of 1.5), while other queries are slowed down (e.g., Q1.3 by a factor of 1.25). This is due to different operator placement enforced by the *Data-Driven* heuristic. With respect to the average query execution time, the average query response time of *Data-Driven Chopping* is improved by a factor of 1.1 and with *Chopping* it is improved by a factor of 1.2. By examining execution times of all queries, we observe that short running queries become slower to some degree, whereas long running queries are accelerated. This is not surprising as long running queries either include more operators or process more data, which both increases the probability of operator aborts and thus, benefit more from *Chopping*. Short running queries are not always executed at full speed but can be decelerated by the concurrency limitation of *Chopping*.

We conclude that heap contention occurs in complex workloads and can significantly decrease performance and increase resource usage. Furthermore, we have seen that *Chopping* and *Data-Driven Chopping* both significantly accelerate query processing and reduce resource consumption by avoiding heap contention.

During our experiments we also discovered that the heap contention effect can be much stronger in case the join order is sub-optimal, because the greater intermediate results increase processing time and the probability of operator aborts.

We confirm the observations of Wang that executing too many queries in parallel on GPUs degrades performance [35]. Our solution *Data-Driven Chopping* limits the use of the

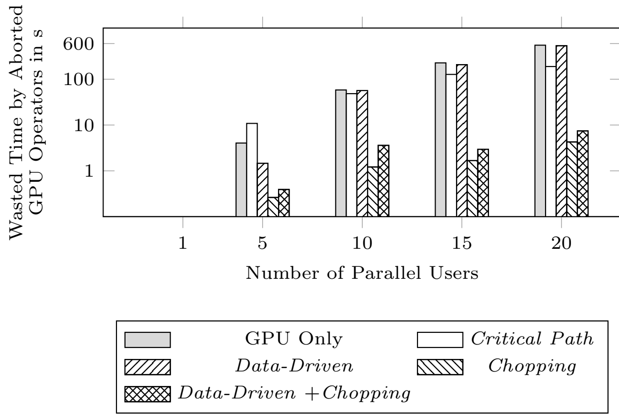


Figure 20: Wasted time by aborted GPU operators depending on the number of parallel users for the SSBM. With an increasing number of users, the wasted time increases significantly because of heap contention.

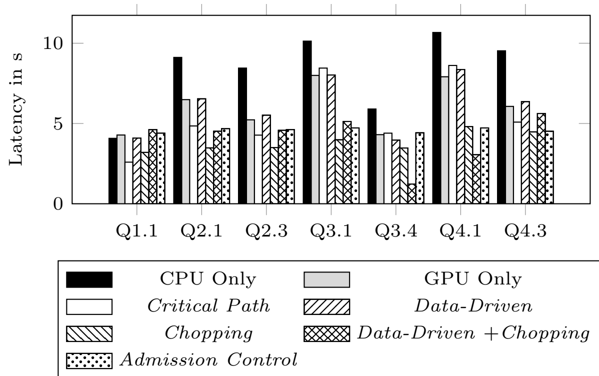


Figure 21: Latencies for selected SSBM queries for 20 users and a database of scale factor 10.

GPU to the degree where it is beneficial and thus, avoids heap contention.

6.3 Discussion

In all of our experiments, the strategy which combines *Data-Driven* with *Chopping* achieved the best overall result. Either it was the fastest strategy, or it was as fast as the other strategies while minimizing IO. This is not surprising as this is the only strategy that avoids the cache thrashing and the heap contention effect, and thus, achieves the most stable performance, especially when compared to the state-of-the-art heuristics, which use a operator-driven data placement at query compile-time. We also learned that cache thrashing has a much stronger effect on a complex query workload than heap contention.

Data-Driven Chopping improves several metrics at once:

1. **Worst-Case Execution Time:** *Data-Driven Chopping* improves performance when resources become scarce of up to factor 24 in our micro benchmarks and up to factor 2 for the SSBM and TPC-H workload.
2. **PCIe Traffic:** *Data-Driven Chopping* requires less IO (up to a factor of 48) on the PCIe bus. In database engines using block-oriented query execution, we can spend these resources to stream data blocks from CPU to the co-processor when the co-processor finished its local data.

Our results have several implications. We can use co-processors only for a part of the workload. We can improve the scalability by compressing the database, which shifts the point where performance breaks down to a larger scale factor or number of users. Thus, compression neither solves the cache thrashing nor the heap contention problem. However, it is common to use multiple GPUs in a single machine, which can handle larger databases and more parallel users. This scale up by multiple co-processors can help us to process workloads that have resource demands exceeding the resources of a single co-processor. Our *Data-Driven* strategy can support multiple co-processors by performing horizontal partitioning. However, the basic problems and their solutions stay the same. Additionally, our results show that GPUs—and other co-processors—alone are not a viable solution. CPUs and co-processors need to work together to perform query processing efficiently.

7. RELATED WORK

In this section, we discuss related work on co-processor-accelerated DBMSs and concurrent query processing.

7.1 Co-Processor-accelerated DBMSs

He and others developed the first GPU-accelerated database engine, namely GPUQP [13]. GPUQP can use CPU and GPU in the same query and uses a modified backtracking optimizer: Each query plan is decomposed into sub-plans with at most 10 operators. Then, the optimizer performs a backtracking search to find the optimal plan for each sub-plan. Finally, the physical query plan is created by combining the optimal sub-plans. GPUQP could afford to create many plan candidates, because it used analytical cost models, where each estimation can be computed in a couple of CPU cycles. However, for learning-based approaches, computing an estimation can take a non-negligible amount of time (in the order of several micro seconds). Since CoGaDB uses the learned cost models of HyPE during optimization, using backtracking or dynamic programming approaches is very expensive. In this work, we counter this drawback of learning-based cost models by introducing simple but efficient heuristics.

He and others investigated the performance of hash joins on a coupled CPU/GPU architecture [14]. For each step in the hash join, a certain part of the input data is placed on the CPU and on the GPU to fully occupy both processors and minimize execution skew.

Zhang and others developed OmniDB [38], a database engine that targets heterogeneous processor environments and focuses on hardware obliviousness, similar to Ocelot [15]. OmniDB schedules so called *work units* on the available processors. Each work unit is placed on the processor with the highest throughput, but only a certain fraction of the workload may be executed on each processor to avoid overloading. We were not able to include this heuristic in CoGaDB, because CoGaDB uses a bulk processor, whereas the heuristic of Zhang assumes a vector-at-a-time processing model.

Pirk and others propose the approximate and refine technique [30], where data is lossily compressed using the bitwise decomposition technique [31]. The idea is to compute an approximate result on lossily compressed data, which is cached on a co-processor. Then, the result is refined on the CPU, which has the missing information lost by the compression and filters out false positives. The technique completely avoids data transfer from the CPU to a co-processor, similar to our *Data-Driven* technique. However, approximate and refine requires a refinement step after each GPU operator. On the one hand, this distributes the load on CPU and GPU

and allows for inter-device parallelism. On the other hand, the data transfer from the co-processor to the CPU is likely to become the bottleneck. *Data-Driven* avoids to copy back intermediate results if possible, but is also likely to require a larger device memory footprint. It would be worthwhile to compare these techniques in future work.

Karnagel and others extended the hardware-oblivious database engine Ocelot by their heterogeneity-aware operator placement [20], which uses a combination of analytical and learning-based cost models to predict the performance of operators, similar to He and others [13] and Yuan and others [37]. Karnagel and others place operators at run-time and use operator-driven data placement. However, they do not support inter-operator parallelism on a single processor.

Karnagel and others evaluated the impact of compile-time and run-time optimization with the Ocelot Engine [19]. They conclude that both approaches are similarly efficient, where run-time placement is easier to implement and global optimization achieves an overall more robust performance. We make similar observations in CoGaDB in case no memory thrashing or heap contention occurs. However, when hitting the resource limits of co-processors, the operator placement should be done at run-time.

Except some management tasks, GPUDB [37], MultiQx-GPU [35] and Red Fox [36] process queries only on the GPU, and hence, use no query optimization heuristic for operator placement. Ocelot is capable of running on all OpenCL capable processors, but Ocelot cannot make automatic placement decisions by itself [15]. For operator placement, Ocelot makes use of the HyPE optimizer [8].

Aside from GPUs, there are other co-processors to accelerate database query processing, such as MICs [18, 23], Cell Processors [16], and FPGAs [25].

7.2 Concurrent Query Processing

Parallel query processing on co-processors and its problems with resource contention is strongly related to concurrent query processing in general.

Harizopoulos and others contribute QPipe, a relational engine that uses *simultaneous pipelining* and focuses on parallel OLAP workloads [12]. Since concurrent queries are likely to access the same data and perform similar operators, it is possible to perform common disk accesses or to reuse common intermediate results. Since QPipe detects commonalities between queries at run-time, it does not need a multi-query optimizer. Our strategy *Data-Driven* shares the basic idea, because co-processor operators share the cached access structures of CoGaDB to avoid the data transfer overhead.

Arumugam and others developed the DataPath system, which uses a purely data-driven approach to push data to processors, where it is consumed by any interested computation [3]. This allows for heavy sharing of computational and memory resources and may be a viable way to use co-processors in case the database is orders of magnitudes larger than the co-processors memory.

Psaroudakis and others propose to decompose database operators into tasks, which can be efficiently executed on multi-socket, multi-core machines [33]. However, a fixed concurrency level is not optimal, and thus, needs to be adapted at run-time. Furthermore, they decompose complex operators into several tasks according to a certain task granularity to efficiently parallelize OLAP queries. These results can also benefit *Chopping*, and other run-time placement strategies, to adjust the concurrency in a workload, especially on a co-processor to find the optimal concurrency level, where co-processor operators seldom abort and we can sufficiently use the co-processor to accelerate query processing. The operator decomposition can also help us to process operators on CPUs and co-processors concurrently.

In a further study, Psaroudakis and others investigate under which conditions a data warehouse should use simultaneous pipelining and global query plans [32].

Leis and others present the Morsel framework, which multiplexes a query workload to a fixed set of worker threads at the granularity of blocks of tuples (morsels) [22]. Parallelism is achieved by processing different morsels in parallel by the same operator pipeline. These pipelines are created by just-in-time query compilation [27]. Furthermore, the Morsel framework is NUMA aware, because it prefers to work on morsels in local NUMA regions. Thus, it shares the idea of processing data locally similar to *Data-Driven* and also uses a thread pool pattern to avoid over-commitment similar to *Chopping*. Mühlbauer and others build a heterogeneity-aware operator placement on top of the Morsel framework to optimize databases for performance and energy efficiency on the ARM big.LITTLE processor [26].

Wang and others investigated concurrent query processing on GPUs in their system MultiQx-GPU [35]. They argue that we cannot utilize the PCIe bus bandwidth, device memory bandwidth, and compute utilization with a single query, and propose to execute queries concurrently on the GPU. However, due to the limited device memory, the DBMS needs to be careful to not overload the GPU, because otherwise the performance decreases. Wang and others use an admission control mechanism to steer the concurrency.

However, our observations differ for concurrent workloads for the *GPU Preferred* strategy, which achieved 62% better performance for a workload with ten parallel queries. We explain this difference by the differences in the database engines and their fault-tolerance mechanisms. Wang and others execute complete queries on the GPU, and use their cost-driven replacement technique to swap data to the CPU's main memory in case memory becomes scarce, which causes high PCIe bus traffic. CoGaDB also evicts cached data to successfully complete a query, but uses CPU and GPU to process queries. Otherwise, CoGaDB aborts an operator and restarts it on a CPU. With our *Data-Driven* heuristic, we avoid this data transfer overhead and can cheaply outsource load from the GPU in case it becomes overloaded. We expect similar results if our approaches are applied to other systems, such as Ocelot [15] or MultiQx-GPU [35].

8. SUMMARY

In this paper, we investigated robust query processing in heterogeneous co-processor environments. Since co-processors typically have a small dedicated memory, it is crucial to cache frequently-accessed data in the co-processor's memory. We identify two effects during query processing on co-processors that can lead to poor performance: Cache thrashing and heap contention.

We showed that placing operators on co-processors, where their input is cached and the remaining operators are processed on the CPU, is the key to overcome cache thrashing. The heap contention problem appears in parallel user workloads, where multiple operators use a co-processor. We can solve this issue by using a pool of worker threads that pulls operators to the co-processor, and at the same time dynamically reacts to operator faults (e.g., out of memory). We showed that our technique *Data-Driven Chopping* combines these approaches and achieves robust and stable query processing compared to the state of the art of query processing on co-processors.

9. REFERENCES

[1] CUDA C programming guide, CUDA version 6.5, 77–78. NVIDIA, 2014. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980. ACM, 2008.

[3] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath system: A data-centric analytic processing engine for large data warehouses. In *SIGMOD*, pages 519–530. ACM, 2010.

[4] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.

[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[6] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[7] S. Breß, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084–1096, 2013.

[8] S. Breß, M. Heimel, M. Saecker, B. Kocher, V. Markl, and G. Saake. Ocelot/HyPE: Optimized data processing on heterogeneous hardware. *PVLDB*, 7(13):1609–1612, 2014.

[9] S. Breß, N. Siegmund, M. Heimel, M. Saecker, T. Lauer, L. Bellatreche, and G. Saake. Load-aware inter-co-processor parallelism in database query processing. *Data & Knowledge Engineering*, 93(0):60–79, 2014.

[10] L. Chen, X. Huo, and G. Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *SC*, pages 25:1–25:11. IEEE, 2012.

[11] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*, pages 134–144. IEEE, 2011.

[12] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394. ACM, 2005.

[13] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query co-processing on graphics processors. In *ACM Trans. Database Syst.*, volume 34. ACM, 2009.

[14] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *Proc. VLDB Endow.*, 6(10):889–900, 2013.

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

[16] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the Cell broadband engine. In *DaMoN*, pages 4:1–4:6. ACM, 2007.

[17] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[18] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *PVLDB*, 8(6):642–653, 2015.

[19] T. Karnagel, D. Habich, and W. Lehner. Local vs. global optimization: Operator placement strategies in heterogeneous environments. In *DAPHNE, EDBT/ICDT Workshops*, pages 48–55, 2015.

[20] T. Karnagel, M. Heimel, M. Hille, M. Ludwig, D. Habich, W. Lehner, and V. Markl. Demonstrating efficient query processing in heterogeneous environments. In *SIGMOD*, pages 693–696. ACM, 2014.

[21] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624. IEEE, 2010.

[22] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754. ACM, 2014.

[23] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. Goh, and R. Huynh. Optimizing the mapreduce framework on Intel Xeon Phi coprocessor. In *Big Data*, pages 125–130. IEEE, 2013.

[24] T. Mostak. An overview of MapD (massively parallel database). White Paper, MIT, April 2013. http://geops.csail.mit.edu/docs/mapd_overview.pdf.

[25] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.

[26] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Kemper, and T. Neumann. Heterogeneity-conscious parallel query execution: Getting a better mileage while driving faster! In *DaMoN*, pages 2:1–2:10. ACM, 2014.

[27] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[28] P. O'Neil, E. J. O'Neil, and X. Chen. The star schema benchmark (SSB), 2009. Revision 3, http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[29] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2):928–939, 2010.

[30] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *ICDE*. IEEE, 2014.

[31] H. Pirk, T. Sellam, S. Manegold, and M. Kersten. X-Device Query Processing by Bitwise Distribution. In *DaMoN*, pages 48–54. ACM, 2012.

[32] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing data and work across concurrent analytical queries. *PVLDB*, 6(9):637–648, 2013.

[33] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In *ADMS*, pages 36–45. VLDB Endowment, 2013.

[34] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[35] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with GPUs. *PVLDB*, 7(11):1011–1022, 2014.

[36] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red Fox: An execution environment for relational query processing on GPUs. In *CGO*, pages 44:44–44:54. ACM, 2014.

[37] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB*, 6(10):817–828, 2013.

[38] S. Zhang et al. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *PVLDB*, 6(12):1374–1377, 2013.

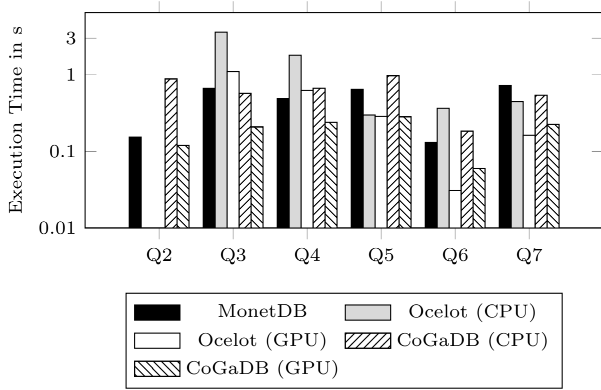


Figure 22: Query execution times for selected TPC-H queries for a single user and a database of scale factor 10.

APPENDIX

A. PERFORMANCE COMPARISON: OCELOT VERSUS CoGaDB

In this section, we conduct a performance comparison of CoGaDB and a state-of-the-art database engine with GPU support: Ocelot [15] (at revision 3e75851).[3] As Ocelot is an extension of MonetDB, we include measurements of MonetDB as well. Note that MonetDB, Ocelot, and CoGaDB exploit all cores of the CPU to maximize performance. For a fair comparison with CoGaDB, we optimized MonetDB/Ocelot as follows.

We set the databases to read-only mode and set the size of OIDs to 32 bit. Furthermore, we measured a warm system after the queries where run before to ensure that the database resides in memory. For all measurements with MonetDB/Ocelot, we used the benchmark scripts that are provided by the author of Ocelot as part of the source code.

We show the average query execution times for the TPC-H benchmark in Figure 22 and the SSBM in Figure 23. Both engines show that GPUs can significantly accelerate query processing. We will now discuss the performance for the GPU and CPU backends. Since we are interested in the raw query processing power, we choose a configuration with a single user and use databases with scale factor 10, where neither memory thrashing nor heap contention occurs. We omit SSBM Query 2.2 and TPC-H query 2 for Ocelot, as it does not support them.

For the SSBM workload, the GPU backends of Ocelot and CoGaDB perform equally well for queries Q1.1-Q1.3 and Q3.1-Q4.3. Ocelots GPU backend is faster for the queries Q2.1 and 2.3. As for the TPC-H workload, CoGaDB is faster for queries Q3 and Q4, equally fast for queries Q5 and Q7 and slower for query Q7. Thus, the GPU backends of Ocelot and CoGaDB are both highly optimized and competitive in performance.

As for the CPU backend, we see that Ocelot performs better than CoGaDB for all SSBM queries except Q1.1-1.3 and Q3.1. For queries Q4.1-Q4.3, the performance improvement is comparatively small. For the TPC-H queries, CoGaDB is faster or competitive to Ocelot for queries Q3,Q4,Q6, Q7 and slower for query Q5. Besides TPC-H query Q2, CoGaDB is never significantly slower than MonetDB, and for some queries even faster (e.g., SSBM Q4.1).

[3]https://bitbucket.org/msaecker/monetdb-opencl

We conclude that CoGaDB is competitive in performance to the MonetDB/Ocelot system and thus, is a suitable basis for our performance studies.

B. MICRO BENCHMARKS

In this section, we describe our micro benchmarks which show the cache thrashing and heap contention effects.

B.1 Serial Selection Workload

Our first benchmark shows the cache thrashing effect. For this, we execute queries serially and use multiple selection queries that access different input columns of the fact table from the star schema benchmark. We show the queries in Figure 1. A workload consists of 100 repetitions of these queries. We execute the workload multiple times and show the average execution time.

```
1 select * from order where quantity<1
2 select * from order where discount>10
3 select * from order where shippriority>0
4 select * from order where extendedprice<100
5 select * from order where ordtotalprice<100
6 select * from order where revenue<1000
7 select * from order where supplycost<1000
8 select * from order where tax>10
```

Listing 1: Serial Selection Queries. Note the interleaved execution. The order table is an alias for the lineorder table.

B.2 Parallel Selection Workload

Our second benchmark shows the heap contention effect. To show the problems of aborted co-processor operators, we use a more complex selection query, but filter only two columns which fit into the co-processors data cache (cf. Listing 2). The query is derived from query Q1.1 of the star schema benchmark. For each experiment, we execute 100 queries, but increase the number of parallel user sessions that execute the workload. Since all workloads contain the same amount work, an ideal system could execute all workloads in the same time. The only difference is that with increasing number of parallel users, the parallelism in the DBMS changes from intra-operator parallelism to inter-operator parallelism.

```
select * from order where discount
between 4 and 6 and quantity between 26 and 35
```

Listing 2: Parallel Selection Query

C. WORKLOADS

In this section, we briefly present the star schema benchmark and our modifications to the TPC-H benchmark.

C.1 Star Schema Benchmark

The *Star Schema Benchmark* (SSBM) is a popular OLAP benchmark, derived from the TPC-H benchmark by applying de-normalization. The SSBM is frequently used for performance evaluation, such as in C-Store [2] or GPUDB [37].

C.1.1 Schema and Data

The SSBM uses a classical star schema with one fact table *lineorder* and four dimension tables *supplier*, *part*, *date*, and *customer*. Similar to TPC-H, we can adjust the size of the database by a scale factor. If not defined differently, we use a scale factor of 10 (LINEORDER contains 60,000,000 tuples).

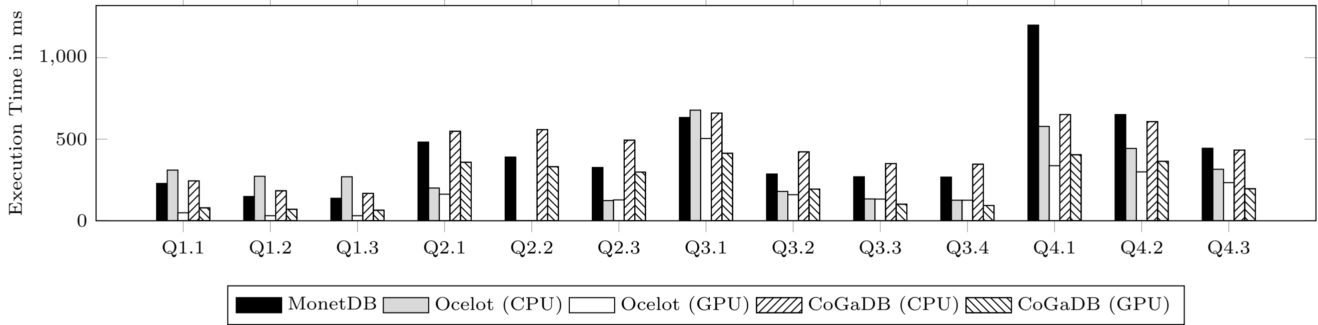


Figure 23: Query execution times for SSBM queries for a single user and a database of scale factor 10.

C.1.2 Queries

The SSBM defines 13 queries, which are grouped into four categories (flights). In each category, the basic query is the same, but different queries have different selectivities. One category basically models a drill-down operation in a data warehouse. Furthermore, the number of required join operations vary from 1 (category 1) to 4 (category 4) join operations. Therefore, with increasing category number, query complexity increases. For further details on the SSBM, we refer the reader to the work of O'Neil and others [28].

C.2 Modifications to TPC-H Benchmark

As the TPC-H benchmark is widely known, we will focus on our modifications to the workload. Our goal is to run a representative set of TPC-H queries that benchmark relational operators. Advanced capabilities such as case statements, joins with arbitrary join conditions, and substring functions are not in our scope and queries that use these features are omitted. Thus, we focused on the efficient support of six TPC-H queries (Q2-Q7) on CPU and GPU to evaluate the memory thrashing and heap contention effects.

D. CRITICAL PATH HEURISTIC

We now discuss the *Critical Path* heuristic for operator placement used by default in CoGaDB. *Critical Path* optimizes for response time and operates as classic cost-based optimizer, where multiple plan candidates are enumerated and the cheapest plan is selected for execution.

To achieve a low response time for queries, we need to optimize the *critical path*, which is the path from a leaf operator to the plan's root that takes the longest time to execute. As data transfers are expensive, we only consider plans where such a path is either completely executed on the CPU or the co-processor. For each binary operator, the processing on the co-processor is continued only if both child operators were executed on the co-processor.

This heuristic also significantly reduces the search space, as we have an exponential complexity in the number of leaf operators instead of an exponential complexity in the number of all operators. However, for large queries this can also become expensive, so we use an iterative refinement algorithm that prunes the search space further and runs for a fixed amount of iterations.

Based on these concepts, the *Critical Path* heuristic works as follows. Each leaf operator is assigned to the CPU and a pure CPU plan is created. From this initial plan, we first investigate all plans where a single leaf operator (and it's path to the first binary parent operator) is executed on the co-processor. For the fastest plan, one additional leaf operator is placed on the co-processor and the next iteration takes place.

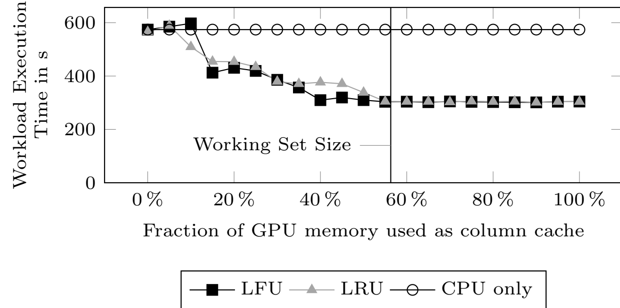


Figure 24: Execution time of a workload of interleaved SSBM queries with different placement strategies on a database of scale factor 10.

The additional reduction of the optimization space makes *Critical Path* quadratic in the number of leaf operators. The algorithm terminates if all plans of the reduced optimization space were examined or a fixed number of iterations has passed in case the plan contains too many leaf operators.

The *Critical Path* heuristic produces faster query plans than simulated annealing or genetic algorithms in most cases. Occasionally, the produced plans of simulated annealing or genetic algorithms are more efficient, but the plans produced by *Critical Path* are still competitive. Thus, the problem specific information of *Critical Path* provide an advantage over established optimization strategies that explore a larger part of the optimization space.

E. DATA PLACEMENT STRATEGIES

We compare a LFU and a LRU strategy in our data-driven strategy for a complete SSBM workload in Figure 24. The amount of GPU memory that is used as column cache is varied between 0% and 100%. If input data does not fit in the GPU memory, all processing on that data will be performed by the CPU. As expected, the execution times are improving until the working set fits entirely in the GPU memory, with no slowdown for cases where no column fits into the cache. The LFU strategy performs slightly better than the LRU strategy for corner cases, where different columns are cached first by LRU and LFU. This leads to a slight slowdown of LRU. Overall, we conclude that the background policy is effective for complex query workloads such as the SSBM and that the data placement strategy itself has only a minor impact on performance. Thus, the observed performance gain comes clearly from our data driven strategy.

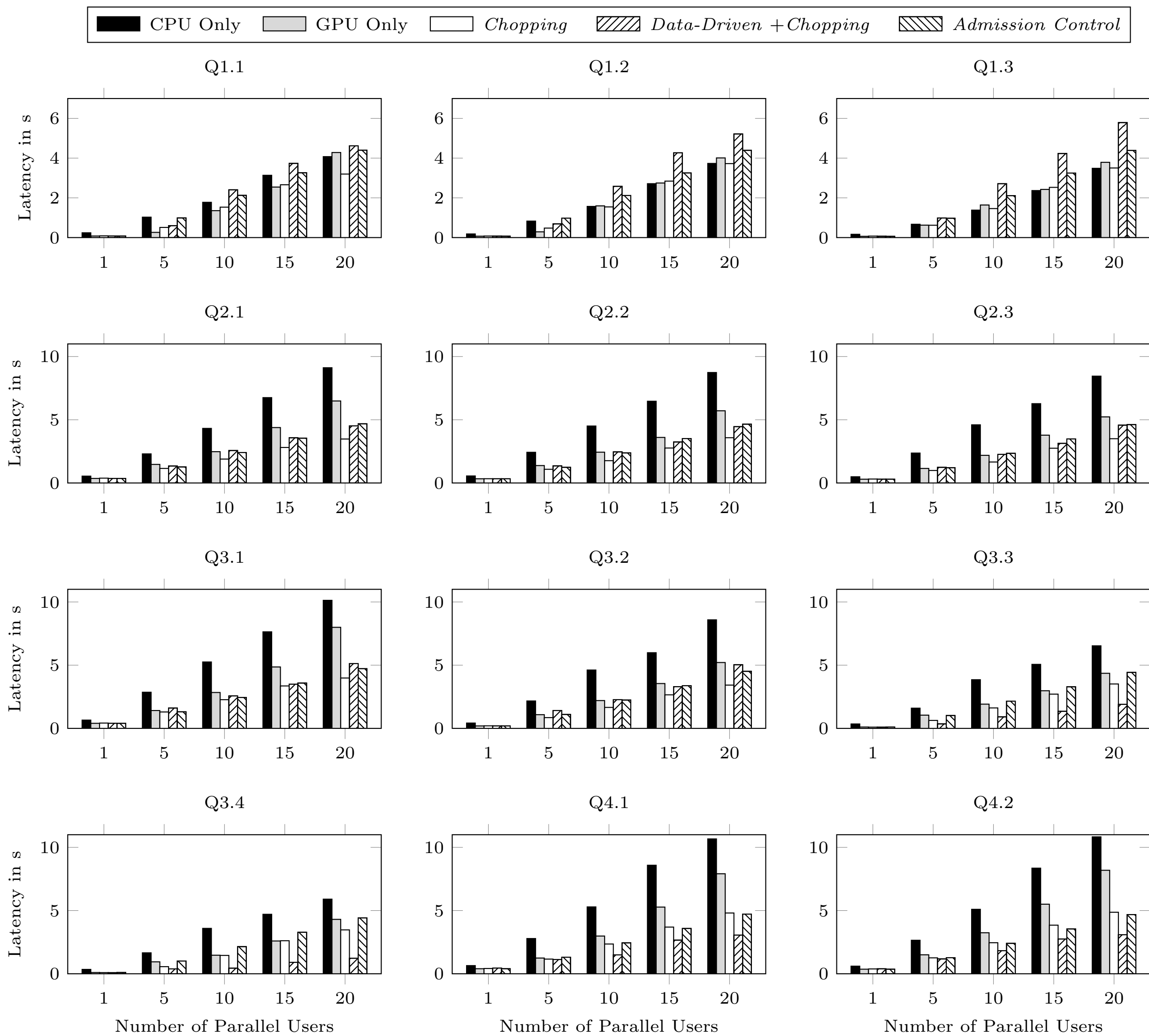


Figure 25: Latencies of all SSBM queries with a varying number of parallel users and a database of scale factor 10.

Acknowledgements

We thank the anonymous reviewers of SIGMOD for their helpful comments. We also thank Max Heimel from TU Berlin, David Broneske, Sebastian Dorok, and Andreas Meister from University of Magdeburg, and Thomas Lindemann and Michael Kußmann from TU Dortmund University for helpful feedback and discussions. The work has received funding from the Deutsche Forschungsgemeinschaft (DFG), Collaborative Research Center SFB 876, project C5 (http://sfb876.tu-dortmund.de/) and from the European Union's Horizon2020 Research & Innovation Program under grant agreement 671500 (project "SAGE").