Efficient Query Processing in Co-Processor-accelerated Databases

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Sebastian Breß

geb. am 25.11.1988 in Quedlinburg

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Jens Teubner
Prof. Dr. Kai-Uwe Sattler

Magdeburg, den 30.10.2015

Breß, Sebastian:
Efficient Query Processing in Co–Processor-accelerated Databases
Dissertation, University of Magdeburg, 2015.

Abstract

Advancements in hardware changed the bottleneck of modern database systems from disk IO to main memory access and processing power. Since the performance of modern processors is primarily limited by a fixed energy budget, hardware vendors are forced to specialize processors. Consequently, processors become increasingly heterogeneous, which already became commodity in the form of accelerated processing units or dedicated co-processors such as graphics processing units.

However, building a robust and efficient query engine for such heterogeneous co-processor environments is still a significant challenge. Although the database community developed fast parallel algorithms for a large number of heterogeneous processors, we still require methods to use these processors *efficiently* during query processing.

This thesis shows how we can build database management systems that efficiently use heterogeneous processors to reliably accelerate database query processing. Thus, we explore the design space of such co-processor-accelerated DBMSs to derive a generic architecture of such systems. Our investigations reveal that one of the most crucial problems in database engines running on heterogeneous processors is to decide which operator of a query should be executed on which processor. We refer to this as the *operator placement problem*. Common analytical modeling would greatly increase the complexity of a DBMS, because this complexity directly relates to the degree of heterogeneity of processors. Thus, we present a framework for hardware-oblivious operator placement called HyPE, which hides the processor heterogeneity from the DBMS by using statistical learning methods and efficiently balances the load between processors.

Furthermore, we examine performance and scalability of query processing for co-processor-accelerated DBMSs in case common assumptions of co-processing techniques are not met. Our investigations show that co-processors can significantly slow down a DBMS when not used appropriately and develop approaches that avoid using co-processors when we expect a performance degradation.

Throughout this thesis, we show the efficiency of our approaches by integrating them into our open source database engine CoGaDB.

Zusammenfassung

Hardwareentwicklungen haben den klassischen Flaschenhals von Datenbanksystemen von Platten IO zu Hauptspeicherzugriffen und Rechenleistung verschoben. Da die Leistung von modernen Prozessoren in erster Linie durch ein festes Energiebudget begrenzt ist, sind Hardwarehersteller gezwungen Prozessoren zu spezialisieren. Daher werden Prozessoren immer heterogener, was bereits durch die Etablierung von Accelerated Processing Units oder dedizierten Coprozessoren wie beispielsweise Grafikprozessoren beobachtet werden kann. Allerdings ist noch unklar, wie ein robustes und effizientes Datenbankmanagementsystem für solche heterogenen Coprozessorumgebungen aufgebaut sein muss. Obwohl die Datenbank Community effiziente parallele Algorithmen für viele heterogene Prozessoren entwickelt hat, fehlt es weiterhin an Methoden, um diese Prozessoren während der Anfrageverarbeitung effizient zu nutzen.

Diese Dissertation zeigt, wie wir Datenbankmanagementsysteme bauen können, die heterogene Prozessoren effizient und zuverlässig zur Beschleunigung der Anfrageverarbeitung nutzen können. Daher untersuchen wir typische Entwurfsentscheidungen von coprozessorbeschleunigten Datenbankmanagementsystemen und leiten darauf aufbauend eine generische Architektur für solche Systeme ab. Unsere Untersuchungen zeigen, dass eines der wichtigsten Probleme für solche Datenbankmanagementsysteme die Entscheidung ist, welche Operatoren einer Anfrage auf welchem Prozessor ausgeführt werden sollen. Wir bezeichnen dies als das *Operatorplatzierungsproblem*. Klassische analytischen Modellierung würde die Komplexität des Datenbankmanagementsystems erheblich erhöhen, weil diese Komplexität direkt mit den Grad der Heterogenität der Prozessoren zunimmt. Wir entwickeln ein Framework für Operatorplatzierung namens HyPE, dass die Prozessorheterogenität vor dem DBMS mit Hilfe von statistischen Lernmethoden versteckt und effizient die Last zwischen den Prozessoren verteilt.

Darüber hinaus untersuchen wir die Leistung und Skalierbarkeit der Anfrageverarbeitung in coprozessorbeschleunigten Datenbanken in Situationen, in denen typische Annahmen von Coprozessortechniken verletzt werden. Unsere Untersuchungen zeigen, dass Coprozessoren ein Datenbankmanagementsystem deutlich verlangsamen können, wenn sie in ungeeigneter Weise verwendet werden und entwickelt Ansätze, die die Nutzung von Coprozessoren vermeiden, wenn ein Leistungsabfall zu erwarten ist.

Im Laufe dieser Arbeit zeigen wir die Effizienz unserer Ansätze durch ihre Integration in unser Open-Source Datenbankmanagementsystem CoGaDB.

Acknowledgements

First of all, I wish to thank my advisors Gunter Saake and Jens Teubner for their guidance and patience. Without them, this thesis would not have been possible as it is now. Additionally, I would like to thank Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn, which supported me especially when I started this thesis and helped me to write my first papers.

I would like to give my special thanks to Max Heimel. Since we met at BTW 2013, we frequently worked together, discussed new ideas, wrote papers, and even organized a workshop. It is no exaggeration to say that his advice greatly impacted and improved my research, always having an open ear for my concerns and also by mercilessly reviewing paper drafts. Thanks Max, you are great!

Now I wish to thank David Broneske, Sebastian Dorok, and Andreas Meister. Since they joined the database group in Magdeburg, we had many fruitful discussions, which led to fresh ideas and changed my point of view at problems. Thanks guys!

I would not have gotten into this "mess" of writing a dissertation, were it not for Alexander Grebhahn and Reimar Schröter. They had to cope with me since we were university students. Despite that, they encouraged me to apply for a position as PhD student in the database group in Magdeburg.

I also wish to thank Norbert and Janet Siegmund, who also supported me in my PhD studies. Especially Norbert taught me how to focus on the task at hand, to be efficient and also gave me important advice for conducting sound experiments. Janet taught me the importance of statistical tests and gave important advice for many aspects of life.

Many other people supported me with advice, such as Mario Pukall, Sandro Schulze and Martin Schäler. For example, I learned from Martin to be a nifty badass, when required. I would also like to thank the many authors and co-authors I worked with: Ladjel Bellatrache, Tobias Lauer, Michael Saecker, Felix Beier, Hannes Rauhe to just name some of them. Surely I have forgotten some people, but people who know me know that I am forgetful. Nevertheless, I do appreciate the support.

Finally, I would like to thank my family, especially my grandmother for always being strict with me, expecting the best performance possible, and teaching me the self-discipline I required to complete this thesis.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Traditionally, database systems managed databases that were primarily stored on secondary storage and only a small part of the data could fit in main memory. Therefore, disk IO was the dominating cost factor. Nowadays, it is possible to equip servers with several terabytes of main memory, which allows us to keep databases in main memory to avoid the IO bottleneck. Therefore, the performance of databases became limited by memory access and processing power [129]. Thus, the database community developed cache-efficient database algorithms, which would only need a small number of CPU cycles to process a tuple. For further performance improvements, database systems rely on increasing memory bandwidth and processing power. However, due to developments in processors (e.g., multi-core CPUs), database systems no longer get faster automatically by buying new hardware. We need to adapt database systems to these multi-core processors to achieve peak performance [186].

1.1 Performance Limitations of Modern Processors

Up until 2005, the performance of single-threaded micro processors increased tremendously. This was due to two effects. First, the number of transistors on a chip doubled every 18 months, which is known as Moore's Law [137]. Second, the power density of transistors was constant and hence, smaller transistors required less voltage and current, which is known as Dennard scaling [55]. While Moore's Law continues until today, Dennard scaling hit a wall, because the leakage current increases with smaller transistors [98]. Leakage current increases power consumption of transistors, even if the transistors are turned off. Thus, the static power dissipation increases as well. Therefore, with constant chip size and an increasing number of transistors, the static power dissipation increased. This, in turn, increases overall power consumption and the produced heat [98]. As processors have to work with a certain power budget and need to be cooled down to remain operational, the performance of modern processors is primarily bound by a fixed energy budget [33].

As a consequence, vendors had to concentrate on processors with multiple cores to make use of the increasing number of transistors to improve performance. However, the benefit of increasing parallelism will not justify the costs of process scaling (i.e., creating even smaller transistors) [62]. Experts predict that this trend will not scale beyond a few hundred cores [80]. Even with multi-core designs at 22 nm, 21 % of a chip's transistors need to be powered off. With 8 nm transistors, it is expected that over 50 % of transistors need to be powered off [62].

Modern processors can either operate with lower clock rate or turn off parts of the chip, commonly referred to as *dark silicon* to remain in the power constraints [33, 62, 80].

The key to exploit the increasing number of transistors is to provide a large number of cores that are specialized for certain tasks.

Then, the cores that are most suitable for the current workload are powered on until the energy budget is reached [62, 80]. Inevitably, this will cause processors to become increasingly *heterogeneous*.

In summary, modern processors hit the *power wall*, which forces vendors to optimize a processor's performance within a certain energy budget [33]. As a consequence, processor manufacturers started to specialize processors to certain application domains. For example, graphics processing units were optimized for compute intensive rendering applications [174].

Experts predict that future machines will consist of a set of heterogeneous processors, where each processor is optimized for a certain application scenario [33, 80]. 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). Processor vendors also combine heterogeneous processors on a single die, for example a CPU and a GPU. Co-processors with dedicated memory are especially interesting, as they increase the total memory bandwidth of a machine and thus, can reduce the negative impact of the memory wall [129]. Not taking advantage of (co-)processors for query processing means to leave available resources unused.

1.2 Databases in the Heterogeneous Many Core Age

Unsurprisingly, the database community investigates how we can exploit the parallelism and processor heterogeneity to improve performance of query processing, which can be seen in the many specialized workshops that have emerged (e.g., the DaMoN [8, 115] and ADMS [32] workshop series). Most publications investigate single database operators on certain (co-)processors in isolation (e.g., GPU [85], APU [88, 89], MIC [106, 127], FPGA [50, 140]).

In this thesis, we investigate the scalability limits of database engines that support additional (co-)processors and show how we can overcome these limitations.

While it is crucial to have efficient database algorithms on modern (co-)processors, the DBMS needs to decide which operators of a query should run on which (co-)processor to efficiently use these additional processing resources, a problem which we refer to as the *operator placement problem*. This coupled query processing over multiple heterogeneous (co-)processors received little attention so far but is essential for any DBMS facing a set of heterogeneous processors. The challenge is to find an approach that works for a wide range of heterogeneous processors without any customization. Otherwise, the complexity of the DBMS software increases with the number of supported processors.

An inherent property of heterogeneous (co-)processors is that their performance may differ significantly depending on the operation. When the DBMS can choose the optimal processor for each operator, some processors are typically overloaded while other processors are idle, which wastes processing resources we could have used to improve performance. Therefore, we need to balance the load on each processor to achieve peak performance.

The next major challenge a database architect runs into is because of common assumptions of co-processing approaches: First, the working set of the database fits in the co-processor's memory, which avoids communication costs. Second, there is no concurrent use of a co-processor by two or more operators or users. We investigate the scalability of our GPU-accelerated database engine in terms of database size and and the number of concurrent users and find that these usual assumptions severely limit the scalability of the DBMS. This is because we always run into the same limitations: First, the interconnection bus between the CPUs and the co-processors and second, the small memory capacity of co-processors. We now elaborate more details on the aforementioned challenges.

1.2.1 Operator Placement

While it is crucial to have efficient database algorithms on modern (co-)processors, we need heuristics that allow the DBMS to use (co-)processors efficiently during query processing. This coupled query processing over multiple heterogeneous (co-)processors received little attention so far.

With multiple heterogeneous (co-)processors in the same machine, the DBMS faces the challenge of picking the optimal (fastest) processor for each database operator in a query plan to minimize the query's response time. In this thesis, we refer to this challenge as the *operator placement problem*. The DBMS requires an additional optimizer step during physical optimization, where it computes a suitable operator placement. Operator placement introduces two challenges:

1. Similar to other optimization problems, the operator placement problem has a search space of exponential size: For n (co)-processors and m operators in the query plan, the query optimizer needs to explore n^m query plans to find the optimal solution. Consequently, we require efficient heuristics to find good query plans while maintaining an acceptable optimization time (in the order of tens of milliseconds).

2. Regular DBMSs typically use cost functions to estimate result cardinalities of each query operator and then, choose a query plan with minimal expected intermediate result sizes. However, the estimated cardinalities of intermediate results alone are not sufficient for operator placement (e.g., the input table has 1000 rows, should we use the CPU or the GPU to evaluate the selection?). Therefore, we need to estimate the actual execution time of query operators on each processor to perform operator placement.

For both problems, we can adapt existing solutions for query optimization and operator run-time estimation. We can solve the first problem by standard algorithms such as iterative refinement, genetic algorithm, simulated annealing, or a heuristic specifically tailored to the operator placement problem. The second problem, however, is not straightforward. Here, the state of the art to estimate the run-time of an operator is to analyze the processor architecture and the algorithm in detail and, based on that, create an analytical cost model (e.g., for CPUs [130] and GPUs [85]).

The Need for Hardware-Obliviousness

Analytical cost models have the advantage that their estimations are predictable and fast (only a couple of CPU cycles). Their major drawback lies in the large effort of creating and maintaining them. To emphasize the complexity of such cost models, we present the parameters of a cost model for a join algorithm as a running example.

First, we require the features of the input tables (e.g., number of rows in each table) and of the operator (e.g., join selectivity). Second, we need to consider the data transfer cost between processors: If the input data is not accessible to a processor, then we have to copy it to the correct processor memory before starting processing. Third, we need to know hardware-dependent parameters such as cache size and memory latency.[1] Fourth, we require the algorithm's implementation details (e.g., do we use a partitioned or an unpartitioned hash table?)

If we take all these parameters together, it becomes apparent how complex and maintenance intensive these models become in practice. Note that we need one analytical cost model for each processor architecture and each database algorithm. Considering the advancement in processor architectures with each new processor generation, we need to include the processor generation as well (e.g., Ivy Bridge and Haswell for Intel CPUs or Fermi and Kepler for Nvidia GPUs). Thus, most of the complexity of analytical cost models comes from the hardware and algorithm dependent parameters.

Scalability Limits of Analytical Cost Models

We provide an example to give an intuition over the number of models. We assume ten database algorithms that can run on four different processors (CPU, Integrated GPU, External GPU, MIC). Furthermore, we want to support the three most recent processor

[1] The performance of hash join algorithms is often limited by memory latency.

generations. In the worst case, this makes a total of $10 \cdot 4 \cdot 3 = 120$ analytical cost models that need to be manually maintained and updated whenever an algorithm is changed or a new processor type needs to be supported. Fortunately, the architectures are typically not completely different, so it is possible to use certain models for multiple processors and calibrate model parameters automatically by running micro benchmarks. However, we still need to verify the correctness of such models for each processor. Consequently, the database query optimizer's complexity heavily depends on the supported processor architectures and database algorithm implementations, which will change over time.

With the increasing heterogeneity of modern processors—accompanied with increasing diversity of database algorithms, each optimized for a certain processor—it becomes apparent that the cost of creating and maintaining analytical cost models for each database algorithm on each processor type gets prohibitively large. Fortunately, the complexity of the algorithm-dependent parameters can be reduced when using a set of hardware-oblivious database operators as suggested by Heimel and others [95]. However, as the actual operator run-time is still heavily dependent on the processor, we still have a large number of models.

In this thesis, we treat processors and database algorithms as black boxes and learn cost models for run-time estimation automatically during query processing.

1.2.2 Efficient Query (Co-)Processing

Due to the heterogeneous nature of (co-)processors, they are suited differently for each database operator. While joins are typically faster on GPUs, selections can be faster processed on CPUs, in case the input data is not cached on the GPU [85]. A naive algorithm for operator placement could just assign each operator to the fastest processor for this kind of operation. However, this can easily lead to load imbalance: certain processors become overloaded whereas other processors may become underutilized or idle. The key to maximize performance is to exploit the parallelism between processors.

Aside from load imbalance, there is a second major factor we need to consider to achieve efficient query processing in co-processor-accelerated databases: oversimplification. Most papers proposing techniques for co-processors make at least one of the two following assumptions. First, the input data fits in the co-processors memory (and is cached) and second, there are no concurrent queries in the system. Considering the—compared to CPU main memory size—small co-processor memory capacities, the first assumption is frequently violated, whereas the second assumption is violated by virtually all systems. However, breaking these assumptions can slow down the performance of query processing by a factor of up to 24, as we will see in Chapter 7. Consequently, we can hardly bound the worst-case execution time of a query and hence, cannot ensure scalability and robustness of our query processor.

1.2.3 Summary

From our discussions up to now, we derive three requirements a database engine for heterogeneous processor environments must meet:

1. **Hardware Obliviousness:** The DBMS needs to work in environments where no detailed knowledge about processors is available.
2. **Algorithm-Speed and Processor-Load Awareness:** The DBMS needs to perform operator placement and, therefore, needs to consider the processing speed of algorithms on all (co-)processors and the load on each (co-)processor.
3. **Robustness:** DBMSs inevitably run into workloads that violate typical assumptions of co-processor-accelerated database techniques. The DBMS still needs to achieve scalable performance in single-user and multi-user workloads.

Based on these requirements, we identify three research challenges:

1. We need to solve the operator placement problem in a hardware-oblivious way.
2. We need to efficiently process multiple *operators* concurrently. Here, we need to exploit the inter-processor parallelism to maximize performance.
3. We need to efficiently process multiple *queries* concurrently. In such complex scenarios, we need measures to ensure robustness and efficiency of query processing even under resource constraints of modern co-processors.

1.3 Contributions

We now describe the contributions of the thesis. First, we discuss the typical design of co-processor-accelerated databases to provide a general understanding of the design choices and system aspects (Chapter 2). We will focus on GPU-accelerated database engines, because GPUs are the most common representative of co-processors. Thus, we use GPUs as a poster child for co-processors in the remainder of this thesis.

From these theoretical results, we discuss the architecture of CoGaDB, a GPU-accelerated database engine. We built CoGaDB to investigate how state-of-the-art approaches interact in a single system and how such a system scales. We discuss how existing techniques and the contributions of this thesis interact with each other in a single system (Chapter 3).

Parallel to the development of CoGaDB, many other GPU-accelerated database engines were built. Thus, we refine our understanding of the architecture of co-processor accelerated database engines by comparing existing prototypes, identifying and classifying commonly used optimizations, and proposing a generic architecture for co-processor-accelerated DBMSs (Chapter 4).

Then, having a general understanding of the system aspects, we present a generic way to solve the operator placement problem (Chapter 5). Furthermore, we discuss how we can accelerate the performance by inter-processor parallelism between heterogeneous processors (Chapter 6). Finally, we investigate the scalability of CoGaDB, by stressing it in terms of database size and number of parallel users. We propose simple but efficient techniques that significantly increase the scalability of co-processor-accelerated databases by avoiding common resource contention (Chapter 7).

1.3.1 The Design Space of GPU-Accelerated Database Systems

In Chapter 2, we summarize the major findings on GPU-accelerated data processing. Based on this survey, we present key properties, important trade-offs and typical challenges of GPU-aware database architectures.

The material was published in the following paper:

[43] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Exploring the design space of a GPU-aware database architecture. In *Proc. Int'l Workshop on GPUs In Databases (GID)*, pages 225–234. Springer, 2013

The thesis author is the first author and wrote the paper. The other co-authors improved the material and it's presentation.

1.3.2 CoGaDB as Evaluation System

When this thesis started, there was no GPU-accelerated DBMS available that would allow us to conduct our research. To investigate the scalability of co-processing approaches in a single system and to show the viability of our techniques, we develop CoGaDB, a main-memory DBMS with built-in GPU acceleration, which is optimized for OLAP workloads. CoGaDB uses the self-tuning optimizer framework HyPE to build a hardware-oblivious optimizer, which learns cost models for database operators and efficiently distributes a workload on available processors. Furthermore, CoGaDB implements efficient algorithms on CPU and GPU and efficiently supports star joins. We show how these novel techniques interact with each other in a single system. Our evaluation shows that CoGaDB quickly adapts to the underlying hardware by increasing the accuracy of its cost models at runtime. CoGaDB will serve us as running example and as evaluation platform in the remainder of this thesis. Therefore, we discuss how the developed approaches of this thesis can work together in a single system. We discuss CoGaDB in Chapter 3 and discuss our contributions in detail in later chapters. The material was published in the following papers:

[35] S. Breß. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *The VLDB PhD workshop, PVLDB*, 6(12):1398–1403, 2013

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

The author is the only author of both publications. Norbert Siegmund, David Broneske, Tobias Lauer, Ladjel Bellatreche, and Gunter Saake provided feedback for drafts of [35]. Jens Teubner and Theo Härder commented on drafts of [36]. The author is the initiator, system architect, and primary developer of CoGaDB. Additionally, the following people contributed to CoGaDB: David Broneske, Sebastian Dorok, Andreas Meister, Henning Funke, Stefan Noll, Florian Treinat, Jens Teubner, René Hoyer, Patrick Sulkowski, Steven Ladewig, Robin Haberkorn, Jan Wedding, Darius Brückers, Sebastian Krieter, Steffen Schulze, John Sarrazin, Daniel Smit, Christian Lausberger, and Julian Blank.

1.3.3 Survey of GPU-accelerated Database Systems

In Chapter 4, we conduct an in-depth analysis of how state-of-the-art database prototypes manage heterogeneous processor environments, demonstrated by systems that support *Graphics Processing Units* (GPUs). Based on this survey, and our findings from Chapter 2, we classify architectural properties and common optimizations of GPU-accelerated DBMSs. Finally, we propose a reference architecture, indicating how GPU acceleration can be integrated in existing DBMSs. This material was published in the following paper:

[44] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. GPU-accelerated database systems: Survey and open challenges. *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, 15:1–35, 2014

The author is the first author and wrote the paper. Max Heimel contributed many ideas, helped with the classifications, and detailed many sections. The co-authors improved the material and it's presentation.

1.3.4 Hardware-Oblivious Operator Placement

In Chapter 5, we present HyPE, a hardware-oblivious framework for operator placement. HyPE automatically learns and adapts cost models to predict the execution time of arbitrary database algorithms (i.e., implementations of relational operators) on any (co-)processor. HyPE uses the cost models to predict execution times and places database operators on available (co-)processors. We demonstrate its applicability for two common use cases in modern database systems. Additionally, we contribute an in-depth discussion of HyPE's operator model, the required steps for deploying HyPE in practice and the support of complex operators requiring multi-dimensional learning strategies:

[38] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake. Automatic selection of processing units for coprocessing in databases. In *Proc. Int'l Conf. on Advances in Databases and Information Systems (ADBIS)*, pages 57–70. Springer, 2012

[37] 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

[42] S. Breß, M. Heimel, M. Saecker, B. Köcher, V. Markl, and G. Saake. Ocelot/HyPE: Optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment*, 7(13):1609–1612, 2014

Publication [37] is an extended version of paper [38]. Both papers were a collaborative effort of the database groups of university of Magdeburg and Ilmenau university of technology. The author is the first author of both papers and contributed the hardware-oblivious operator placement framework (also named as decision model). The index scan use case was contributed by Felix Beier [23] and the update merging use case was contributed by Hannes Rauhe [168]. The paper was collaboratively written and the experiments were conducted by these three authors. Eike Schallehn, Kai-Uwe Sattler and Gunter Saake improved the material and it's presentation.

Publication [42] was a collaborative effort between the university of Magdeburg and the Technische Universität Berlin. The paper discusses the combination of the Ocelot engine and the HyPE optimizer. The author wrote the paper and is the inventor of the HyPE optimizer, whereas Max Heimel improved the material and it's presentation and is the inventor and developer of Ocelot [95]. The integration work was done in equal parts by the author and Max Heimel. Michael Saecker and Bastian Köcher were co-developers of Ocelot. The remaining authors improved the material and it's presentation.

1.3.5 Load-aware Inter-Processor Parallelism

Building on top of the learning-based estimator for operator run-times, we also need to steer the individual load on each processor. Therefore, we need to efficiently distribute a workload on available (co-)processors while providing accurate performance estimates for the query optimizer. We contribute heuristics that optimize query processing for response time and throughput simultaneously via inter-processor parallelism. Our empirical evaluation reveals that the new approach achieves speedups up to 1.85 compared to state-of-the-art approaches while preserving accurate performance estimations. Furthermore, we use a simulation to assess the performance of our best approach for systems with multiple co-processors and derive some general rules that impact performance in those systems. We discuss this material in Chapter 6, which was published in the following papers:

[45] S. Breß, N. Siegmund, L. Bellatreche, and G. Saake. An operator-stream-based scheduling engine for effective GPU coprocessing. In *Proc. Int'l Conf. on Advances in Databases and Information Systems (ADBIS)*, pages 288–301. Springer, 2013

[46] 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

Publication [46] is an extended version of paper [45]. The author is the primary author of both papers, wrote the papers, implemented the discussed approaches and conducted the experiments. Norbert Siegmund, Max Heimel, Michael Saecker, Tobias Lauer, Ladjel Bellatreche, and Gunter Saake improved the material and it's presentation.

1.3.6 Robust Query Processing

In Chapter 7, we stress the scalability of CoGaDB and identify two effects that cause poor performance during query processing in case co-processor resources become scarce: cache thrashing and heap contention. *Cache thrashing* occurs when the working set of queries does not fit into the co-processors data cache and degrades performance by a factor of up to 24. *Heap contention* occurs when multiple operators run in parallel on a co-processor and their accumulated memory footprint exceeds the co-processors memory capacity, which slows down performance up to a factor of six.

We propose a solution for each effect, *data-driven operator placement* for cache thrashing and *query chopping* for heap contention and find that the combined approach—data-driven query chopping—achieves robust and scalable performance on co-processors. We validate our proposal on a popular OLAP benchmark: the star schema benchmark.

[39] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, 2016. to appear

The author wrote the paper, implemented the approaches and conducted the experiments. Henning Funke tuned the GPU backend of CoGaDB and extended it with efficient GPU algorithms for the join and semi-join operations, which were required for the experiments. Henning Funke and Jens Teubner helped developing the material and it's presentation. In Chapter 8, we summarize our results and provide an overview of possible future work.

2. The Design Space of GPU-Accelerated Database Systems

In this chapter, we take a closer look at how today's database engine prototypes manage heterogeneous environments, demonstrated by systems that support *Graphics Processing Units* (GPUs). The GPU is the pioneer of modern co-processors, and – in the last decade – it matured from a highly specialized processing device to a fully programmable, powerful co-processor. This development inspired the database research community to investigate methods for accelerating database systems via GPU co-processing. Several research papers and performance studies demonstrate the potential of this approach [18, 57, 85, 88, 156, 157] – and the technology has also found its way into commercial products (e.g., Jedox [1] or ParStream [2]).

Using graphics cards to accelerate data processing is tricky and has several pitfalls: First, for effective GPU co-processing, the transfer bottleneck between CPU and GPU has to either be reduced or concealed via clever data placement or caching strategies. Second, when integrating GPU co-processing into a real-world *Database Management System* (DBMS), the challenge arises that DBMS internals – such as data structures, query processing and optimization – are traditionally optimized for CPUs. While there is ongoing research on building GPU-aware database systems [63], no unified GPU-aware DBMS architecture has emerged so far.

In this chapter, we point out the lack of a unified GPU-aware architecture and derive – based on a literature survey – a reduced design space for such an architecture. Thus, we traverse the design space for a GPU-aware database architecture based on results of prior work.

The chapter is structured as follows: In Section 2.1, we provide background information about GPUs and discuss related work. We explore the design space for GPU-accelerated DBMSs with respect to functional and non-functional properties in Section 2.2.

2.1 Preliminary Considerations

In this section, we provide a brief overview over the basics of main memory databases, the architecture of graphics cards, the applied programming model and related work.

2.1.1 Main-Memory DBMSs

With increasing capacity of main memory, it is possible to keep a large fraction of a database in memory. Thus, the performance bottleneck shifts from disk access to main-memory access, the *memory wall* [129]. For main-memory DBMSs, the architecture was heavily revised from a tuple-at-a-time volcano-style query processor to operator-at-a-time bulk processing and from a row-oriented data layout to columnar storage.[1] This increases the useful work per CPU cycle [31] and makes more efficient use of caches [129]. Furthermore, most systems compress data using light-weight compression techniques (e.g., dictionary encoding) to reduce the data volume and the required memory bandwidth for an operator [4]. For more details about main memory DBMSs, we defer the interested reader to the book of Plattner and Zeier [159].

2.1.2 Graphics Card Architecture

Figure 2.1 shows the architecture of a modern computer system with a graphics card. The figure shows the architecture of a graphics card from the *Tesla* architecture of NVIDIA. While specific details might be different for other vendors, the general concepts are found in all modern graphic cards. The graphics card – henceforth also called the *device* – is connected to the *host system* via the *PCIExpress bus*. All data transfer between host and device has to pass through this comparably low-bandwidth bus.

The graphics card itself contains one or more GPUs and a few gigabytes of *device memory*.[2] Typically, host and device do not share the same address space, meaning that neither the GPU can directly access the main memory nor the CPU can directly access the device memory.

The GPU itself consists of a few *multiprocessors*, which can be seen as very wide SIMD processing elements. Each multiprocessor packages several *scalar processors* with a few kilobytes of high-bandwidth, on-chip *shared memory*, cache, and an interface to the device memory.

[1] Many main-memory OLTP systems use a row-oriented data layout.
[2] Typically around 2-4 GB on mainstream cards and up to 16GB on high-end devices.

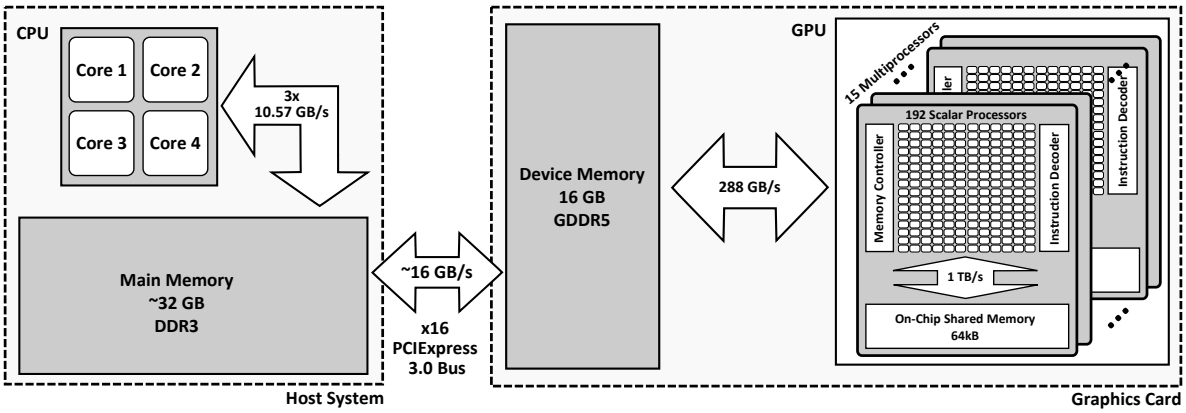


Figure 2.1: Overview: Exemplary architecture of a system with a graphics card.

2.1.3 Programming a GPU

Programs that run on a graphics card are written in the so-called *kernel programming model*. Programs in this model consist of *host code* and *kernels*. The host code manages the graphics card, initializing data transfer and scheduling program execution on the device. A kernel is a simplistic program that forms the basic unit of parallelism in the kernel programming model. Kernels are scheduled concurrently on several scalar processors in a SIMD fashion: Each kernel invocation - henceforth called *thread* - executes the same code on its own share of the input. All threads that run on the same multiprocessor are logically grouped into a *workgroup* [98].

Currently, two major frameworks are used for programming GPUs to accelerate database systems, namely the *Compute Unified Device Architecture* (CUDA) and the *Open Compute Language* (OpenCL). Both frameworks implement the kernel programming model and provide APIs that allow the host CPU to manage computations on the GPU and data transfers between CPU and GPU. In contrast to CUDA, which supports NVIDIA GPUs only, OpenCL can run on a wide variety of devices from multiple vendors [69]. However, CUDA offers advanced features such as allocation of device memory inside a running kernel or *Uniform Virtual Addressing* (UVA), a technique where CPUs and GPUs share the same virtual address space and the CUDA driver transfers data between CPU and GPU transparently to the application [146].[3]

2.1.4 Performance Factors of GPUs

One of the most important performance factors in GPU programming is to avoid data transfers between host and device: All data has to pass across the PCIexpress bus, which is the bottleneck of the architecture. Data transfer to the device might therefore consume all time savings from running a problem on the GPU. This becomes especially

[3]We are aware that these features are included in OpenCL 2.0 but no OpenCL framework supports these features yet.

evident for I/O-bound algorithms: Since accessing the main memory is roughly two to three times faster than sending data across the PCIexpress bus, the CPU will usually have finished execution before the data has even arrived on the device.

Graphics cards achieve high performance through massive parallelism. This means that a problem should be easy to parallelize to gain most from running on the GPU. Otherwise, the problem cannot be efficiently processed by GPUs. The major difference to CPUs is that GPUs are optimized for *throughput* (e.g., finishing as many threads per time unit as possible), whereas CPUs are optimized for *response time* (e.g., finishing a single thread as fast as possible). CPUs rely on caching to hide memory access latencies and pipelining to improve the number of instructions per second. By contrast, GPUs use the massive parallelism to hide memory access latencies of threads by executing instructions of other workgroups. With a sufficient number of workgroups, the GPU has not to wait until a memory access finished. Instead, other workgroups can be executed. Since the GPU multi processors are free to switch the workgroup after issuing an instruction, no branch prediction is required, because the GPU can execute other workgroups until an instruction finished. This saves chip space and allows to spend more transistors on light-weight cores [98].

Another performance pitfall in GPU programming is caused by divergent code paths. Since each multiprocessor only has a single instruction decoder, all scalar processors execute the same instruction at a time. If some threads in a workgroup diverge, for example due to data-dependent conditionals, the multiprocessor has to serialize the code paths, leading to performance losses. While this problem has been somewhat alleviated in the latest generation of graphics cards, it is still recommended to avoid complex control structures in kernels where possible [98].

2.1.5 Related Work

There are two general surveys on GPU co-processing. The first survey is from Owens and others, which discusses the state-of-the-art in GPGPU computing [149]. They cover a wide area of research, mainly GPGPU techniques (e.g., stream operations, data structures, and data queries) and GPGPU applications (e.g., databases and data mining, physically-based simulation, and signal and image processing).

The second survey is from Mittal and Vetter [136], which provide an up-to-date survey on heterogeneous computing techniques. They focus on approaches that use CPU and GPU *together* to increase performance.

In contrast to Owens and others [149] and Mittal and Vetter [136], we focus on recent trends in GPU-accelerated data management to derive a GPU-aware database architecture and open research questions. Meister and others surveyed approaches for database query optimization and other optimization tasks in the context of potential co-processor acceleration [135].

2.2 Exploring the Design Space of a GPU-aware DBMS Architecture

In this section, we explore the design space of a GPU-accelerated database management system from two points of views: Non-functional properties (e.g., performance and portability) and functional properties (e.g., transaction management and processing model). Note that while we focus on relational systems, most of our discussions apply to other data models as well.

2.2.1 Non-Functional Properties

In the following, we discuss non-functional properties which DBMSs are typically optimized for, namely performance and portability, and the introduced problems when supporting GPUs. Tsirogiannis and others found that in most cases, the configuration performing best is also the most energy efficient configuration due to the large up-front power consumption in modern servers [190]. Therefore, we will not discuss energy efficiency separately, as energy efficiency is already covered by the performance property.

Performance.

Since the GPU is a specialized processor, it is faster on certain tasks (e.g., numerical computations) than CPUs, whereas CPUs outperform GPUs for tasks that are hard to parallelize or that involve complex control flow instructions. He and others observed that joins are 2–7 times faster on the GPU, whereas selections are 2–4 times slower, due to the required data transfers [86]. The same conclusion was made by Gregg and others, who showed that a GPU algorithm is not necessarily faster than its CPU counterpart, due to the expensive data transfers [77]. One major point for achieving good performance in a GPU-accelerated DBMS is therefore to avoid data transfers where possible.

Another problem is the selection of the optimal processing device for a given operation. For instance: While the GPU is well suited for easily parallelizable operations (e.g., predicate evaluation, arithmetic operations), the CPU is the vastly better fit when it comes to operations that require complex control structures or significant inter-thread communications (e.g., hash table creation or complex user-defined functions). Selecting the optimal device for a given operation is a non-trivial operation, and – due to the large parameter space (e.g., He and others [85]) – applying simple heuristics is typically insufficient. We argue that there are four major factors that need to be considered for such a decision (1) the operation to execute, (2) the features of the input data (e.g., data size, data type, operation selectivity, data skew), (3) the computational power and capabilities of the processing devices (e.g., number of cores, memory bandwidth, clock rate), and (4) the load on the processing device (e.g., even if an operation is typically faster on the GPU, one should use the CPU when the GPU is overloaded) [37]. Therefore, we argue that a complex decision model that incorporates these four factors, is needed to decide on an optimal operator placement.

Portability.

Modern DBMSs are tailored towards CPUs and apply traditional compiler techniques to achieve portability across the different CPU architectures (e.g., x86, ARM, Power). By using GPUs – or generally, heterogeneous co-processors – this picture changes, as CPU code cannot be automatically ported to run efficiently on a GPU. Also, certain GPU toolkits – such as CUDA – bind the DBMS vendor to a certain GPU manufacturer.

Furthermore, processing devices themselves are becoming more and more heterogeneous [173]. In order to achieve optimal performance, each device typically needs its own optimized version of the database operators [48, 171]. However, this means that supporting all combinations of potential devices yields an exponential increase in required code paths, leading to a significant increase in development and maintenance costs.

There are two possibilities to achieve portability also for GPUs: First, we can implement all operators for all vendor-specific toolkits. While this has the advantage that special features of a vendor's product can be used to achieve high performance, it leads to high implementation effort and development costs. Examples for such systems are GPUQP [85] or CoGaDB [35], a column-oriented and GPU-accelerated DBMS. Second, we can implement the operators in a generic framework, such as OpenCL, and let the hardware vendor provide the optimal mapping to the given GPU. While this approach saves implementation effort and simplifies maintenance, it also suffers from performance degradation compared to hand-tuned implementations. To the best of our knowledge, the only system belonging to the second class is Ocelot [95], which extends MonetDB with OpenCL-based operators.

Summary.

From the discussion, it is clearly visible that GPU acceleration complicates the process of optimizing GPU-accelerated DBMSs for non-functional properties such as performance and portability. Thus, we need to take special care to achieve comparable applicability with respect to traditional DBMSs.

2.2.2 Functional Properties

We now discuss the design space for a relational GPU-accelerated DBMS with respect to functional properties. We consider the following design decisions: (1) main-memory vs. disk-based system, (2) row-oriented vs. column-oriented storage, (3) processing models (tuple-at-a-time model vs. operator-at-a-time), (4) GPU-only vs. hybrid device database, (5) GPU buffer management (column-wise or page-wise buffer), (6) query optimization for hybrid systems, and (7) consistency and transaction processing (lock-based vs. lock free protocols).

Main-Memory vs. Hard-Disk-Based System.

He and others demonstrated that GPU-acceleration cannot achieve significant speedups if the data has to be fetched from disk, because of the IO bottleneck, which dominates

execution costs [85]. Since the GPU improves performance only once the data has arrived in main memory, time savings will be small compared to the total query runtime. Hence, a GPU-aware database architecture should make heavy use of in-memory technology.

Row-Stores vs. Column Stores.

Ghodsnia compares row and column stores with respect to their suitability for GPU-accelerated query processing [71]. Ghodsnia concluded that a column store is more suitable than a row store, because a column store (1) allows for coalesced memory access on the GPU, (2) achieves higher compression rates (an important property considering the current memory limitations of GPUs), and (3) reduces the volume of data that needs to be transfered. For example, in case of a column store, only those columns needed for data processing have to be transferred between processing devices. In contrast, in a row-store, either the full relation has to be transferred or a projection has to reduce the relation to the data needed to process a query. Both approaches are more expensive than storing the data column wise. Bakkum and others came to the same conclusion [17]. Furthermore, given that we already concluded that a GPU-aware DBMS should be an in-memory database system, and that current research provides an overwhelming evidence in favor of columnar storage for in-memory systems [30]. We therefore conclude that a GPU-aware DBMS should use columnar storage.

Processing Model.

There are basically two alternative processing models that are used in modern DBMS: the tuple-at-a-time model [74] and operator-at-a-time bulk processing [132]. Tuple-at-a-time processing has the advantage that intermediate results are very small, but has the disadvantage that it introduces a higher per tuple processing overhead as well as a high cache miss rate. In contrast, operator-at-a-time processing leads to cache friendly memory access patterns, making effective usage of the memory hierarchy. However, the major drawback is the increased memory requirement, since intermediate results are materialized [132].

Tuple-at-a-time approaches usually apply the so-called *iterator model*, which applies virtual function calls to pass tuples through the required operators [74]. Since graphics cards lack support for virtual function calls – and are notoriously bad at runing the complex control logic that would be neccesary to emulate them – this model is unsuited for a GPU-accelerated DBMS. Furthermore, tuple-wise processing is not possible on the GPU, due to lacking support for inter-kernel communication [40]. We therefore argue that a GPU-accelerated DBMS should use an operator-at-a-time model.

In order to avoid the IO overhead of this model, multiple authors have suggested a hybrid strategy that uses dynamic code compilation to merge multiple logical operators, or even express the whole query in a single, runtime-generated operator [54, 143, 192]. Using this strategy, it is not necessary to materialize intermediate results in the GPU's device memory: Tuples are passed between operators in registers, or via shared memory. This approach is therefore an additional potential execution model for a GPU-accelerated DBMS.

Database in GPU RAM vs. Hybrid Device Database.

Ghodsnia proposed to keep the complete database resident in GPU RAM [71]. This approach has the advantage of vastly reducing data transfers between host and device. Also, since the GPU RAM has a bandwidth that is roughly 16 times higher than the PCIe Bus (3.0), this approach is very likely to significantly increase performance. It also simplifies transaction management, since data does not need to be kept consistent between CPU and GPU.

However, the approach has some obvious shortcomings: First, the GPU RAM (up to ≈16 GB) is rather limited compared to CPU RAM (up to ≈2 TB), meaning that either only small data sets can be processed, or that data must be partitioned across multiple GPUs. Second, a pure GPU database cannot exploit full inter-device parallelism, because the CPU does not perform any data processing. Since CPU and GPU both have their corresponding sweet-spots for different applications (cf. 2.2.1), this is a major shortcoming that significantly degrades performance in several scenarios.

Since these problems outweigh the benefits, we conclude that a GPU-accelerated DBMS should make use of all available storage and not constrain itself to GPU RAM. While this complicates data processing, and requires a data-placement strategy[4], we still expect the hybrid to be faster than a pure CPU- or GPU-resident system. The performance benefit of using both CPU and GPU for processing was already observed for hybrid query processing approaches (e.g., He and others [85]).

Effective GPU Buffer Management.

The buffer-management problem in a CPU/GPU system is similar to the one encountered in traditional disk-based or in-memory systems. That is, we want to process data in a faster, and smaller memory space (GPU RAM), whereas the data is stored in a larger and slower memory space (CPU RAM). The novelty in this problem is that – in contrast to traditional systems – data can be processed in both memory spaces. In other words: We can transfer data, but we do not have to! This *optionality* opens up some interesting research questions that have not been covered in traditional database research.

Data structures and data encoding are often highly optimized for the special properties of a processing device to maximize performance. Hence, different kinds of processing devices use an encoding optimized for the respective device. For example, a CPU encoding has to support effective caching to reduce the memory access cost [129], whereas a GPU encoding has to ensure coalesced memory access of threads to achieve maximal performance [146]. This usually requires trans-coding data before or after the data transfer, which is an additional overhead that can break performance.

Another interesting design decision is the granularity that should be used for managing the GPU RAM: pages, whole columns, or whole tables? Since we already concluded that

[4]Some potential strategies include keeping the hot set of the data resident on the graphics card, or using the limited graphics card memory as a low-resolution data storage to quickly filter out non-matching data items [154].

a GPU-accelerated database should be columnar, this basically boils down to comparing page-wise vs. column-based caching. Page-wise caching has the advantage that it is an established approach, and is used by almost every DBMS, which eases integration into existing systems. However, a possible disadvantage is that – depending on the page size –, the PCIe bus may be underutilized during data transfers. Since it is more efficient to transfer few large data sets than many little datasets (with the same total data volume) [146], it could be more beneficial to cache and manage whole columns.

Query Placement and Optimization.

Given that a GPU-aware DBMS has to manage multiple processing devices, a major problem is to automatically decide which parts of the query should be executed on which device. This decision depends on multiple factors, including the operation, the size & shape of the input data, processing power and computational characteristics of CPU and GPU as well as the optimization criterion. For instance: Optimizing for response time requires to split a query in parts, so that CPU and GPU can process parts of the query in parallel. However, workloads that require a high throughput need different heuristics. Furthermore, given that we can freely choose between multiple different processing devices with different energy characteristics, non-traditional optimization criteria such as energy-consumption, or cost-per-tuple become interesting in the scope of GPU-aware DBMSs.

He and others were the first to address hybrid CPU/GPU query optimization [85]. They used a Selinger-style optimizer to create initial query plans and then used heuristics and an analytical cost-model to split a workload between CPU and GPU. Przymus and others developed a query planner that is capable of optimizing for two goals simultaneously (e.g., query response time and energy consumption) [162]. Heimel and others suggest using GPUs to accelerate query optimization instead of query processing. This approach could help to tackle the additional computational complexity of query optimization in a hybrid system [94]. It should be noted that there is some similarity to the problem of query optimization in the scope of distributed and federated DBMSs [119]. However, there are several characteristics that differentiate distributed from hybrid CPU/GPU query processing:

1. In a distributed system, nodes are autonomous. This is in contrast to hybrid CPU/GPU systems, because the CPU has to explicitly send commands to the co-processors.
2. In a distributed system, there is no global state. By contrast, in hybrid CPU/GPU systems the CPU knows which co-processor performs a certain operation on a specific dataset.
3. The nodes in a distributed system are loosely coupled, meaning that a node may loose network connectivity to the other nodes or might crash. In a hybrid CPU/GPU system, nodes are tightly bound. That is, no network outages are possible due to a high bandwidth bus connection, and a GPU does not go down due to a local software error.

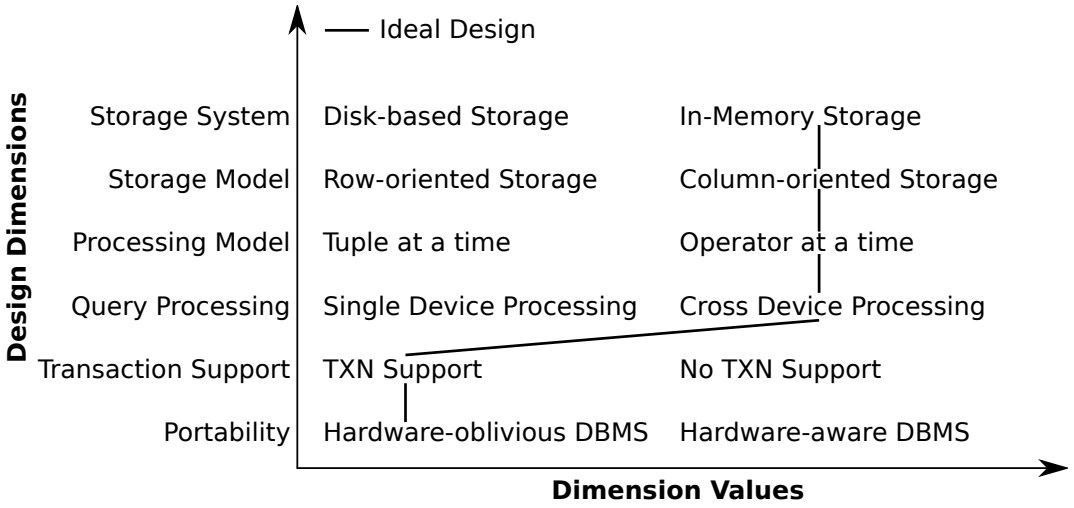


Figure 2.2: Design space of GPU-aware DBMSs

We conclude that traditional approaches for a distributed system do not take into account specifics of hybrid CPU/GPU systems. Therefore, tailor-made co-processing approaches are likely to outperform approaches from distributed or federated query-processing.

Consistency and Transaction Processing.

Keeping data consistent in a distributed database is a widely studied problem. But, research on transaction management on the GPU is almost non-existent. The only work we are aware of is by He and others [87] and indicates that a locking-based strategy significantly breaks the performance of GPUs [87]. They developed a lock-free protocol to ensure conflict serializability of parallel transactions on GPUs. However, to the best of our knowledge, there is no work that explicitly addresses transaction management in a GPU-accelerated DBMS. It is therefore to be investigated how the performance characteristics of established protocols of distributed systems compare to tailor-made transaction protocols.

Essentially, there are three ways of maintaining consistency between CPU and GPU: (1) Each data item could be kept strictly in one place (e.g., using horizontal or vertical partitioning). In this case, we would not require any replication management and would have to solve a modified allocation problem. (2) We can use established replication mechanisms, such as read one write all or primary copy. (3) The system can perform updates always on one processing device (e.g., the CPU) and periodically synchronize these changes to the other devices.

2.3 Summary

We summarize the results of our theoretical discussion in Figure 2.2. A GPU-aware database system should reside in-memory and use columnar storage. As processing model, it should implement the operator-at-a-time bulk processing model, potentially enhanced by dynamic code compilation. The system should make use of all available (co-)processors in the system (including the CPU!) by having a locality-aware query optimizer, which distributes the workload across all available processing resources. We further investigate this issue in Chapter 5, Chapter 6, and Chapter 7. In case the GPU-aware DBMS needs transaction support, it should use an optimistic transaction protocol, such as the timestamp protocol. Finally, in order to reduce implementation overhead, the ideal GPU-accelerated DBMS would be hardware-oblivious, meaning all hardware-specific adaption is handled transparently by the system itself.

While this theoretical discussion already gave us a good idea of how the reference architecture for a GPU-accelerated DBMS should look like, we will take a closer look at existing GPU-accelerated DBMSs to refine our results. In Chapter 3, we present CoGaDB, which is one example of a GPU-accelerated DBMS. In Chapter 4, we will survey other GPU-accelerated DBMSs.

3. System Overview: CoGaDB

In this chapter, we present our system CoGaDB[1], a column-oriented, GPU-accelerated DBMS, which puts together existing work and our thesis contributions in a high-performance OLAP engine that makes efficient use of GPUs to accelerate analytical query processing. We contribute a discussion of our design decisions, provide insights in CoGaDB's parallel query processor and discuss how these techniques interact with our *Hybrid Query Processing Engine* (HyPE) in a single system. HyPE learns cost models for database operators running in heterogeneous processor environments and performs operator placement for query plans. Thus, CoGaDB is the first DBMS that can estimate the run-time of database operators on processors without detailed knowledge of the underlying hardware and efficiently balances workloads between all processors.

This chapter will provide an overview of how the developed approaches of this thesis can work together in a single system. Detailed discussions and evaluations about hardware-oblivious operator placement, load balancing, and robust query processing will follow in Chapter 5, Chapter 6, and Chapter 7, respectively.

The remainder of the chapter is structured as follows. In Section 3.1, we will provide an overview of CoGaDB and discuss implementation details of the query processor in Section 3.2. Then, in Section 3.3, we present our optimizer HyPE, followed by a performance evaluation in Section 3.4. Afterwards, we outline our future development in Section 3.5, present related work in Section 3.6, and conclude in Section 3.7.

3.1 CoGaDB: A GPU-accelerated DBMS

In this section, we provide an overview of CoGaDB's architecture, including details on storage manager, processing and operator model, and GPU memory management. We illustrate CoGaDB's architecture in Figure 3.1.

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

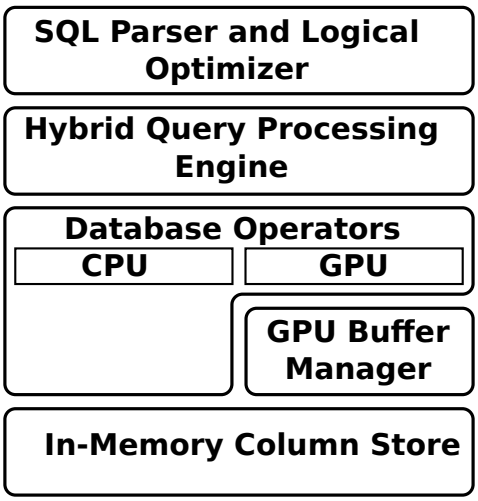


Figure 3.1: CoGaDB's Architecture.

3.1.1 System Overview

CoGaDB's primary goal is to show that we can optimize queries for heterogeneous processor machines without knowing the details of database algorithms and processors. Thus, we can build a query optimizer that scales with the increasing number of heterogeneous processors in today's and future machines without increasing maintenance effort. Since GPU acceleration is most beneficial for data warehousing workloads, we designed CoGaDB as a relational GPU-accelerated OLAP engine. In order to quickly build a working prototype, we support currently only three data types: 32-Bit integers, 32-Bit floats, and variable-length strings. However, these data types are already sufficient to support the *Star Schema Benchmark* (SSBM), a popular OLAP benchmark [204].

In order to provide efficient data processing capabilities, CoGaDB makes use of parallel libraries, such as Intel's *Threading Building Blocks* (TBB) library[2] on the CPU and the Thrust library[3] on the GPU [24]. As GPGPU framework, we decided to use NVIDIA's CUDA framework, because it has the most mature development toolkit and we expect a vendor-specific framework to deliver the best performance.

3.1.2 Storage Manager

As in every DBMS, the backbone of CoGaDB is its storage manager. Since we primarily target OLAP workloads, we choose a columnar data layout, because storing data column-wise allows for more efficient use of the memory hierarchy and higher compression rates. The work of He and others showed that GPU acceleration is not beneficial in case we have to fetch the data from disk [85]. Thus, another important consideration is that GPU acceleration is only beneficial in case we have all required data to answer a

[2]https://www.threadingbuildingblocks.org
[3]http://thrust.github.io

query cached in main memory. Therefore, CoGaDB's storage manager is an in-memory column store. In case the database fits not entirely into main memory, CoGaDB relies on the operating systems virtual memory management to swap cold data to disk.

3.1.3 Processing and Operator Model

As already mentioned, CoGaDB should be able to use all available processors to improve the performance of query processing. Therefore, we need a work unit on which we can build our operator placement. At the query level, a single processor is generally not suited for all operations contained in a query. Thus, we choose the operator level as granularity, because we can place operators to a processor suitable for that operator type.

There are basically two ways how to process queries: Pipelining, where each operator requests the next block of rows and bulk processing, where each operator consumes its input and materializes its output. In a heterogeneous processor machine, where we want to avoid data transfers between processors, we need to ensure that we assign each processor enough work in order to use the system efficiently (intra-operator parallelism). Furthermore, we can achieve parallelism between operators (inter-operator parallelism) if we construct bushy query plans and process independent sub-plans in parallel, which is common in parallel database systems [183]. In summary, CoGaDB uses the operator-at-a-time processing model and combines it with operator-based scheduling to distribute a set of queries on all available processing resources.

3.1.4 GPU Memory Management

A processor with dedicated memory requires a data placement strategy that moves data to the memory where it is needed. In CoGaDB, this is handled by the central GPU buffer manager. All GPU operators request their input columns from the buffer manager. If a column is not dormant in the GPUs memory, it is transferred to the GPU. The same principle is used for additional access structures, such as join indexes.

Memory Allocation Policy

Each GPU operator needs additional memory for data processing, such as memory for the result buffer and temporary data structures. There are two basic memory allocation strategies GPU operators can use. First, allocate the complete memory it needs to complete the computation (pre-allocation). Second, allocate memory as late as possible (allocate as needed). The first strategy avoids GPU operator aborts due to out-of-memory conditions during processing, whereas the allocate-as-needed strategy uses the GPU memory more economical, allowing for concurrently executed GPU operators or a larger GPU buffer for recently used columns. Additionally, the pre-allocation strategy is hard to implement because it is difficult to accurately estimate the result size of an operator. Thus, CoGaDB uses an allocate-as-needed strategy for memory allocation.

Memory Deallocation Policy

Since the memory capacity of a GPU is limited compared to the CPUs memory, it is likely that a GPU operator using the allocate-as-needed strategy will run out of memory while processing an operator, especially in workloads with parallel queries. In this case, cached data has to be removed from the GPU memory. CoGaDB first removes cached columns, because it is relatively cheap to copy them again to the GPU compared to join indexes. In case the removal of columns did not free enough memory, the cached join indexes are removed from the GPU memory.

Fault Tolerance

In case the GPU operator still has insufficient GPU memory after the memory cleanup, it has two options. First, it can wait until enough memory is available for allocation. Second, it can abort, discarding the work it has done so far and starting a fall-back handler that processes the operator on the CPU. With the first option, however, we can run into a similar situation as two transactions waiting for two locks: A cyclic dependency may occur, which causes a deadlock. Consider two GPU operators O_1 and O_2. Operator O_1 and O_2 allocate memory for their first processing step. Then, for a second processing step, both operators try to allocate more memory, but both fail, because O_1 and O_2 have already allocated GPU memory in the first processing step.

Thus, CoGaDB uses the second strategy, aborting a GPU operator and restarting the operator on the CPU. This strategy is similar to timestamp ordering of transactions, because the operator that comes too late (runs out of memory) is aborted. However, depending on the workload, the costs due to operator aborts can be significant.

3.2 Query Processor

In this section, we present details on CoGaDB's query processor, including operators, materialization strategy, operator parallelism, and parallel star joins.

3.2.1 Operators

We now elaborate details on the relational operators implemented in CoGaDB.

Selection

On the CPU, we use a parallel version of the predicated SIMD scan from Zhou and Ross [209]. Each thread scans its own partition of the input column and writes its matching *Tuple Identifiers* (TIDs) to a local output buffer.[4] However, CoGaDB requires that selection operators return a sorted continuous array of TIDs, so we need to combine the local results. Since each thread counts its number of matches, we can use this

[4] In column stores it is common to use the position of an element in an array as TID.

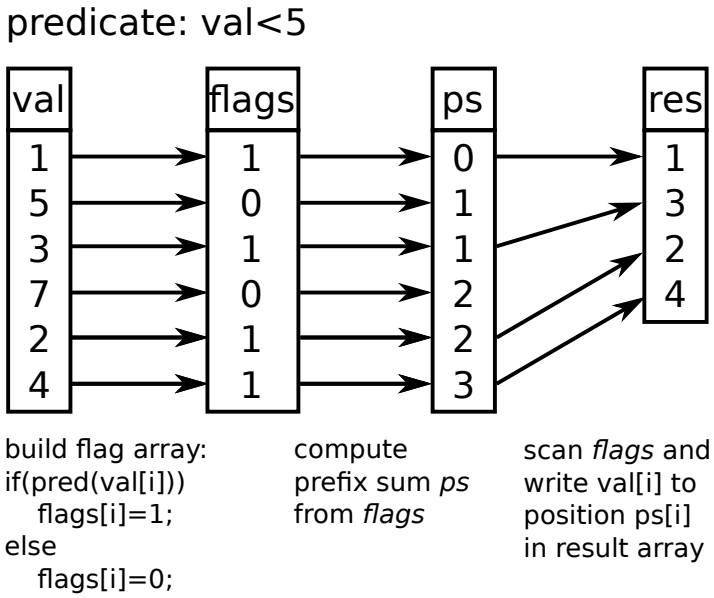


Figure 3.2: Parallel selection on GPUs for predicate $val < 5$.

information to compute the size of the complete result and the memory region, where each thread has to copy their results into. The last step is also done in parallel.

On the GPU, CoGaDB uses the parallel selection of He and others [85]. The key idea of the algorithm is to avoid lock contention when writing the result to the output buffer in parallel. Thus, most GPU algorithms for data processing perform an operation twice. During the first execution, the result size and the write positions for each thread are computed. During the second execution, the result is written to the output buffer in parallel using the write positions from the first phase. Since these two step algorithms require no locks, they scale to a large amount of threads, which makes them very efficient on GPUs. We illustrate these steps for a GPU selection in Figure 3.2. First, the operator performs a first scan of the column to compute a flag array, where a flag is one, if and only if the predicate matched the row. Second, a prefix sum is computed from the flag array to obtain the result size and the write positions of every thread in the output buffer. Finally, the input column is read again but this time each thread knows the position where it has to write its result. Note that the GPU selection needs to do more work compared to the CPU algorithm. However, due to the fine grained parallelism on GPUs, it is possible to perform this extra work very efficiently.

CoGaDB also supports string columns on the GPU by applying dictionary compression. Since the dictionary is not sorted, we can currently evaluate only is-equal and is-unequal predicates.

Complex Selection

Many real world queries do not filter one column only but define complex predicates.

Heimel and others identify three strategies for evaluating complex predicates [91]: chaining operators, complex predicate interpretation, and dynamic code generation and compilation. The chaining operators strategy builds a query plan where each predicate is evaluated independently, followed by operators that merge the result (e.g., combining bitmaps or lists of TIDs). Complex predicate interpretation stores the predicates in an abstract syntax tree and evaluates all predicates directly with a single pass over a table. Dynamic code generation and compilation creates a kernel that directly evaluates all predicates. Similar to complex predicate interpretation, the query compilation technique needs only a single pass over the data. However, query compilation suffers from high upfront costs of compiling a kernel, whereas complex predicate interpretation requires a large kernel with several conditional statements, which leads to branch divergence and, hence, inefficiency on GPUs.

In contrast, the chaining operators strategy allows CoGaDB to use very efficient GPU kernels to evaluate single predicates and to perform the result combination. Scans on different columns can be evaluated on separate processors in parallel to hide the costs of multiple passes over the data. Since CoGaDB's selection operators return sorted lists of TIDs, we use union and intersection operators to compute disjunctions and conjunctions, respectively. We use as set operators the parallel GPU algorithms of the Thrust library.

Join

The most time-intensive operator during relational OLAP is the computation of joins between the fact table and a dimension table. Thus, it is crucial to support efficient join implementations. CoGaDB offers three different join types: a generic join, a primary-key/foreign-key join (PK-FK join), and a fetch join. The generic join makes no assumptions about the input tables and is always applicable. However, generic joins can degenerate to cross joins, which produce very large output results that do not fit in GPU memory. Since this join type is typically not used in CoGaDB, we implemented only a generic CPU hash join.

The most common join type in OLAP workloads is the PK-FK join. Since we know that for PK-FK joins the number of result rows is the number of foreign keys in the foreign key table, we use an optimized version of the indexed-nested-loop join of He and others [85] for the GPU. The algorithm first sorts the primary-key column and, second, assigns all threads a number of foreign keys, which are looked up in the sorted primary key column using binary search. Therefore, we skip the phase where each thread first counts their matching result tuples. On the CPU, we use a hash join, where we build the hash table serially, and perform the pruning phase in parallel.

For OLAP queries, it is often more efficient to pre-filter a dimension table, before performing a join with the fact table. However, in this case, CoGaDB cannot use a

PK-FK join, because the PK-FK relationship may be broken. However, we can precompute a join index, and use the matching TIDs of the dimension table to extract the matching TIDs from the fact table. This optimization proved to be very efficient, on the CPU and the GPU. Since a fetch join is basically a modified version of the merge step from a sort-merge join, we adapted the merging algorithm of He and others [85].

In case more than one join is involved in a query, CoGaDB checks whether the join can be combined in a star join [147], where n dimension tables are joined with a fact table. We provide more details on CoGaDB's star join in Section 3.2.4.

Sorting

Sorting is an important building block of many operators, such as joins. CoGaDB uses the parallel sort primitive of Intel's TBB library on the CPU and the Thrust library on the GPU. For order by statements, CoGaDB needs to sort a table by multiple columns. For this, we start sorting a group of columns $A_1, ..., A_n$ first by A_n, then we retrieve the resulting TID list and fetch the values A_{n-1} to obtain the reordered version of A'_{n-1}. We continue this until we sorted A_1, which results in the final TID list that is the correct sorting order of all groups. Note that this requires a stable sorting algorithm. This approach may seem inefficient, but note that this primitive is used for the final table sort specified in SQL's order-by clause, which typically does not exceed several hundred tuples. Furthermore, we can perform the sorting on the CPU and the GPU.

Group By

CoGaDB uses sorting based grouping algorithms, one generic and one specialized algorithm. The generic algorithm uses the multi-column sort algorithm. A typical optimization is to pack a group of columns in a single 32-Bit integer (or another *bank size*) [108, 166]. Since we sort by the group keys and need the corresponding TID list, we need another 32 Bit as payload. Then, a 32-Bit group key is stored in the upper 32 Bit of a 64-Bit integer and the payload is stored in the lower 32 Bit. We illustrate this principle in Figure 3.3. Then, we sort the array of 64-Bit values, extract the lower 32 Bit as result TID list, and obtain a correct grouping with a single sort operation, either on the CPU or the GPU.

Aggregation and Arithmetic Operations

Based on an input grouping, an aggregation combines data from a column using an aggregation function. CoGaDB supports as aggregation function SUM, COUNT, MIN, and MAX. For the CPU, we use a serial reduction function, whereas we use a parallel reduction function from the Thrust library on the GPU. Since many queries do not just apply an aggregation function, but also perform column algebra operations, we need to efficiently express complex aggregation functions (e.g., select sum(A+B/C) ...). Similar to other column stores (e.g., MonetDB [100]), CoGaDB has separate operators for column arithmetic and constructs for an algebra expression an operator tree that computes the result.

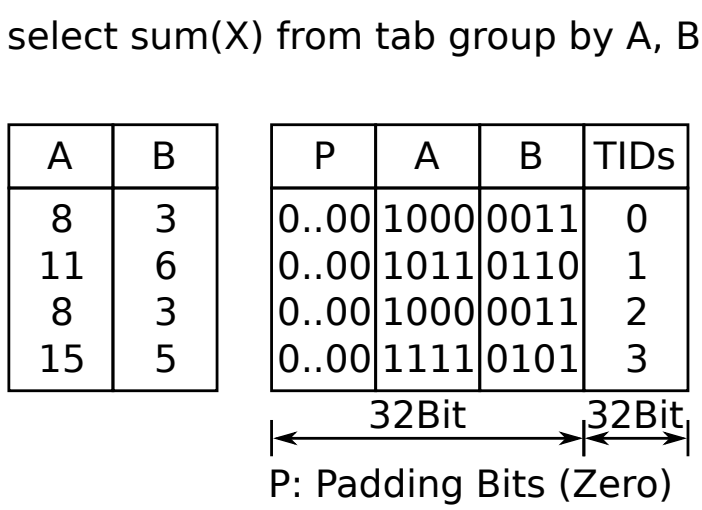


Figure 3.3: Packing values of multiple columns to group a table by columns A and B with a single sorting step.

3.2.2 Materialization Strategy

In every column store, there is a time where the internal columnar representation has to be transformed to a row-wise representation. There are two basic options: Reconstructing tuples as early as possible (early materialization) or as late as possible (late materialization). Manegold and others [131] and Abadi and others [6] found that late materialization is more efficient than early materialization in case queries contain highly selective predicates, and the data is aggregated by a query [6], which is typically the case for OLAP queries. Thus, CoGaDB uses late materialization by transforming the result table in a row-oriented table.

3.2.3 Operator Parallelism

In a bulk processor, each operator can use intra-operator parallelism to increase its efficiency. Another form of parallelism is building bushy query execution plans, where independent sub-plans can be processed in parallel. CoGaDB exploits both types of parallelism, but, in our implementation, we gave priority to inter-operator parallelism, because high parallelism inside operators can lead to over-utilization of processors in case we have large bushy query plans. We expect that advanced approaches such as morsel-driven parallelism [126] or the admission control mechanism of DB2 BLU [167] can avoid over-utilization more efficiently.

While parallel execution of threads is a zero-effort solution on CPUs, we have to add another mechanism on GPUs. CUDA uses the concept of streams to structure parallelism between kernels. Two kernels can be executed in parallel, if and only if they are queued in two different CUDA streams and no kernel is assigned to the default stream 0 [146]. The same principle goes for interleaving copy operations with kernel executions, another important optimization for GPU-accelerated DBMSs.

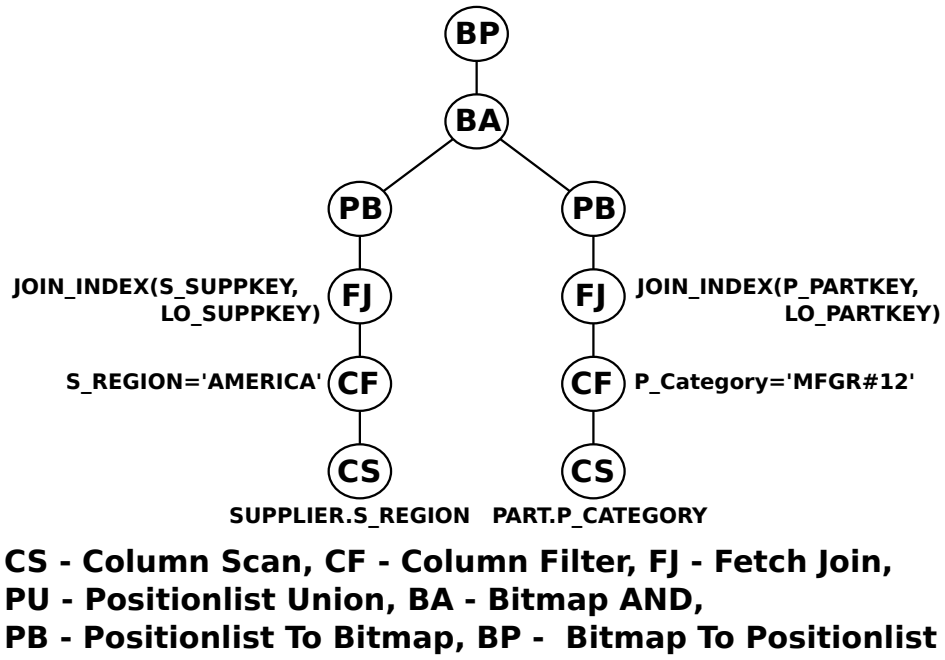


Figure 3.4: Query plan for star join for SSBM query 2.1.

We achieve inter-operator parallelism on GPUs by creating n CUDA streams managed by a *stream manager*. Each GPU operator requests a stream, and the stream manager assigns a stream from a fixed set of streams using the round-robin strategy.

3.2.4 Parallel Star Joins

In case a query contains multiple joins between the fact table and the dimension tables, the joins can be rewritten into a star join, which can process each join between the fact table and a dimension table in parallel [147]. Abadi and others proposed the invisible join [5], an extension of O'Neil's approach [147]. The invisible join is a late materialized join, which reduces the number of expensive random accesses on dimension tables (e.g., with an unsorted position list) [6].

The invisible join works in three phases. First, the predicates of the query are applied to the dimension tables. For each dimension table, we get a list of dimension table keys and insert them into a hash table. In the second step, the corresponding foreign key in the fact table are pruned in the hash table of the corresponding dimension and produces a bitmap indicating the matching rows of the fact table for one dimension. Then, the bitmaps of all dimensions are combined using a bitwise AND operation. In the third phase, the matching rows in the fact tables are looked up in the dimension tables to construct the result of the star join.

CoGaDB implements a variant of the invisible join, where the building of the hash table and pruning of the fact table is replaced by a fetch join from a join index. This

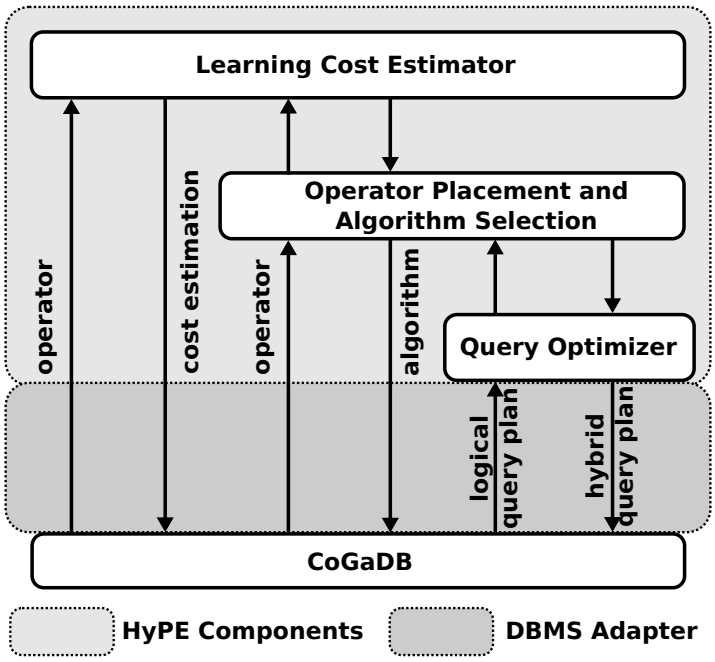


Figure 3.5: Architecture of HyPE.

significantly improved the performance, especially for large dimension tables. The extracted keys from the join index are converted in a bitmap and combined by a bitwise AND operation. The resulting bitmap is converted back to a position list, which is the output format expected by CoGaDB's query processor. In a final step, the result table of the star join is computed by joining the filtered fact table with the filtered dimension tables. CoGaDB's query processor can perform any step from phase one and two of the invisible join on the CPU or the GPU. We illustrate the first two steps in Figure 3.4.

Since query plans for the star join operators are very large, bushy trees, CoGaDB's query optimizer executes different parts of the invisible join on the CPU and the GPU, which leads to inter-device parallelism.

3.3 Hybrid Query Optimizer

Up to now, we discussed how CoGaDB's query processor executes queries. In a heterogeneous processor system, it is crucial that the processed plan makes efficient use of the computational resources and available memory bandwidth on all processors. In order to achieve this goal, CoGaDB uses HyPE for the physical optimization and operator placement. HyPE consists of three components: the estimation component, the operator placement and algorithm selector, and the hybrid query optimizer (cf. Figure 3.5). This section provides an overview of our research results in Chapter 5 and Chapter 6 and puts them into the context of a database engine. Detailed discussions follow in the respective chapters.

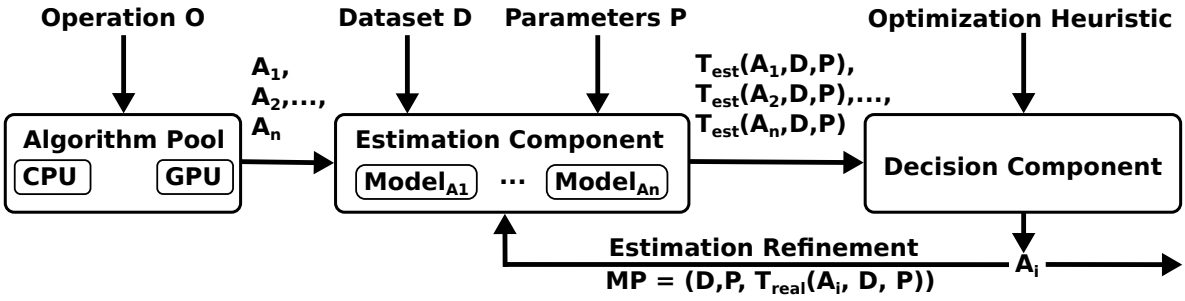


Figure 3.6: HyPE's underlying decision model

3.3.1 Estimation Component

HyPE aims to be hardware- and algorithm-oblivious, which means that it requires minimal knowledge of the underlying processors or the implementation details of database operators. To achieve this goal, the estimation component uses simple regression models (e.g., least squares) to approximate the performance behavior of database algorithms on different hardware w.r.t. properties of the input data (e.g., data size) and properties of the operation (e.g., selectivity). The algorithm runtimes of the first queries executed by CoGaDB serve as training data, and HyPE continuously monitors algorithm runtimes of queries and refines cost models at runtime to improve the model's accuracy. Since CoGaDB uses a bulk processor, the overhead of this continuous monitoring and adaption is minimal, because it is done once for each operator invocation.

3.3.2 Operator Placement and Algorithm Selection

Based on the cost estimator, the optimizer needs to decide for each operator in a query plan, on which processor it should execute the operator, and which algorithm should be used. For each operation, the available algorithms are fetched from a pool of algorithms. Then, an estimation component computes for each algorithm an estimated execution time. Finally, a decision component selects an algorithm according to a user-specified optimization heuristic. After the algorithm finished execution, it returns its execution time to the estimation component in order to refine future estimations. We summarize HyPE's decision model in Figure 3.6. Thus, HyPE solves the operator placement and the algorithm-selection problem in a single step, because it decides for a certain algorithm, which is specific to a certain processor type (e.g., CPU or GPU). In case of multiple devices, HyPE assigns unique identifiers to each algorithm that allows us to pin-point which processor belongs to which algorithm. The optimal algorithm is selected according to an optimization strategy. By default, HyPE uses *Waiting Time Aware Response Time* (WTAR), a scheduling strategy that considers the load on each processor and the estimated execution times of all algorithms for a certain operator. We discuss WTAR in detail in Chapter 6. HyPE keeps track, on which operator was assigned to which processor and uses the accumulated estimated execution times of all operators inside a ready queue as measure for the load condition on a processor, as

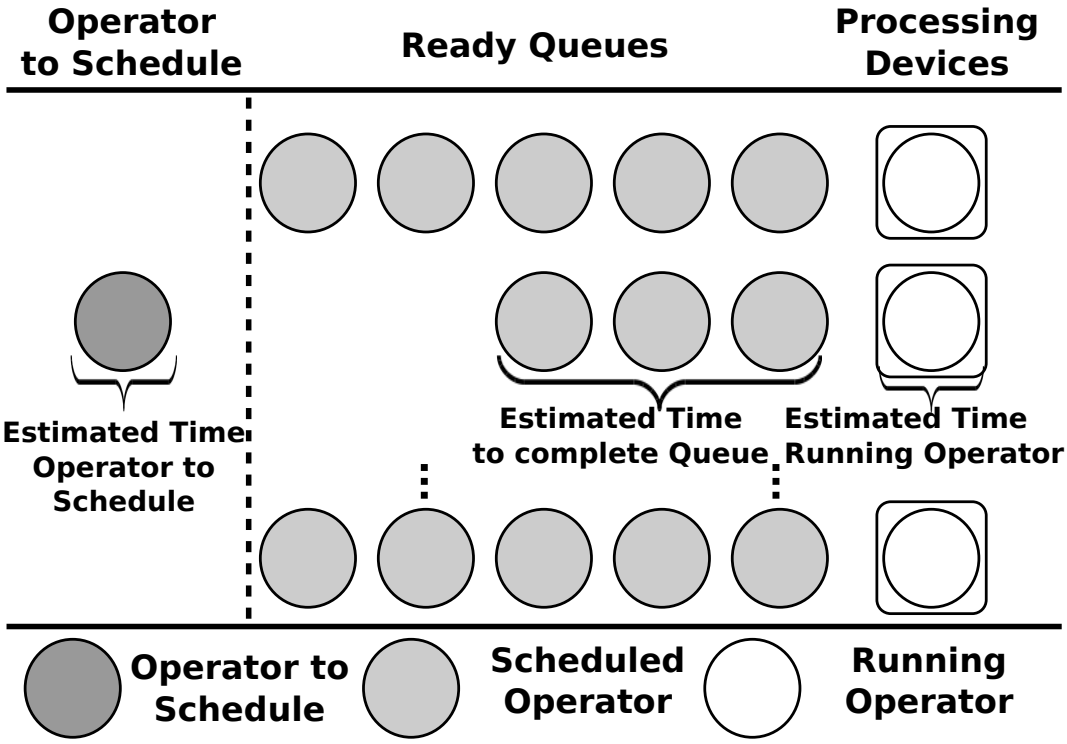


Figure 3.7: Load Tracking with Ready Queues

illustrated by Figure 3.7. The processor that is expected to have the minimal response time is used to execute the operator. This allows us to automatically balance the operators in a query plan on available processors.

3.3.3 Hybrid Query Optimization

The query optimizer assigns for each operator in a query plan a suitable target processor and algorithm. HyPE supports two query optimization modes. In the first mode, HyPE traverses the query plan and requests operator placements from the algorithm selector [40]. Although this is a greedy strategy, it does consider the load on the processors, in case the WTAR optimization strategy is used. Interestingly, the greedy strategy coupled with WTAR schedules queries in a way that independent sub-plans are evaluated in parallel on different processors, which can lead to significant performance gains. In the second mode, HyPE creates a set of candidate plans and performs a classical cost-based optimization, where the costs are not cardinalities, but (estimated) execution times. We explain the detailed algorithm in Appendix A.6. This approach often suffers from poor cardinality estimates, but this problem is not specific to CoGaDB, it is inherent in all DBMSs.

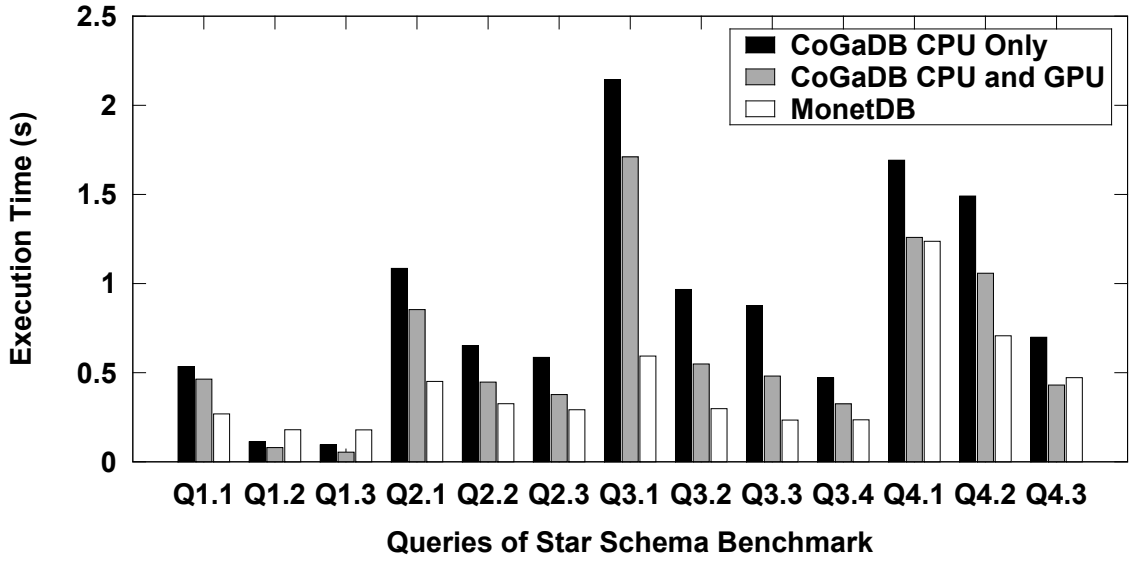


Figure 3.8: Response times of CoGaDB for the SSBM with scale factor 15.

3.4 Performance Evaluation

After describing the architectural design considerations and implementation details, we now conduct a performance evaluation in order to show the efficiency of CoGaDB.

3.4.1 Evaluation Setup

We conduct our experiments on a machine with an Intel® Core™i7-4770 CPU having 4 cores (@3.40GHz) with 24-GB of main memory and a GeForce® GTX 660 GPU (@980MHz) with 1.5-GB device memory.[5] Since CoGaDB is optimized for OLAP benchmarks, we use the Star Schema Benchmark [148], which is a popular OLAP benchmark frequently used for performance evaluations [126, 204]. For all experiments, we use a star schema benchmark database generated with scale factor 15. Therefore, the complete database does not fit into the GPU's memory.

3.4.2 Experiments

In our experiments, we want to answer two questions:

1. Can GPU acceleration significantly improve the performance of query processing, even if the database does not fit in the GPU memory?
2. Can we achieve stable performance in a GPU-accelerated DBMS, in case we use a learning-based optimizer with no detailed information about the processors of a machine?

[5]Hyper-threading was enabled during our experiments.

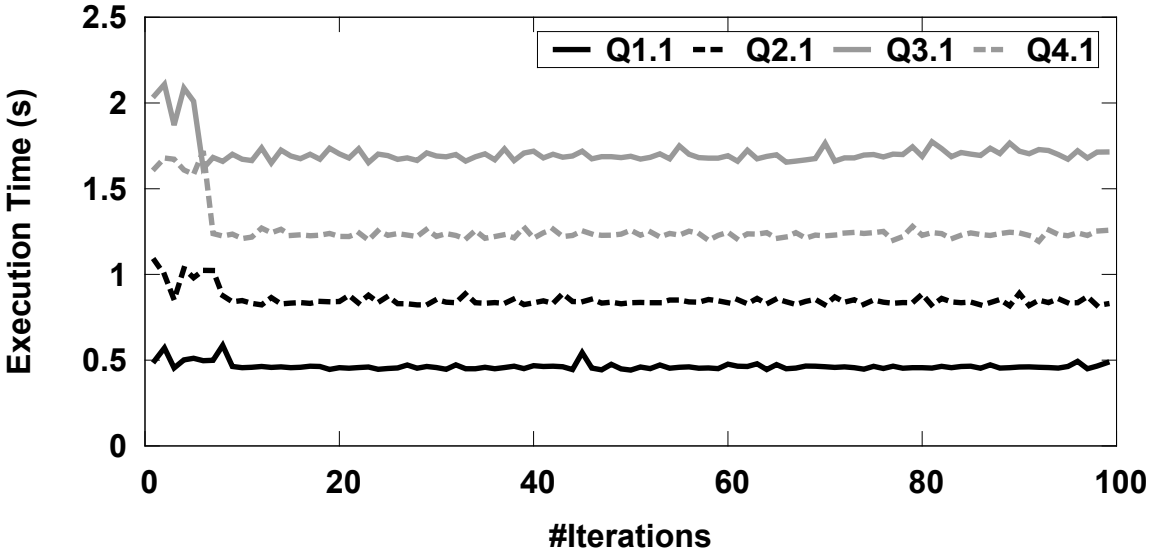


Figure 3.9: Response times of selected SSBM queries in CoGaDB over 100 executions.

Performance Gain of GPU Acceleration

We execute a workload containing all SSBM queries 100 times and executed first all queries of the benchmark. Then, we repeated this process until we executed all queries 100 times. Note that the first run of the SSBM workload was used as warm-up queries, where performance is not considered. We computed the average of the remaining 99 measurements and show the performance of CoGaDB for each query of the SSBM in Figure 3.8 in CPU-only mode and with GPU acceleration. As baseline, we included the results of MonetDB, a highly optimized main-memory DBMS [100].

We observe that CoGaDB's performance is significantly improved by GPU acceleration, where query 3.3 benefits the most from the GPU (by factor 1.8), whereas query 1.1 has the least benefit from GPU acceleration (by factor 1.15). Furthermore, we compare the performance of CoGaDB with MonetDB. For the experiments with MonetDB, we used MonetDB 11.17.13 and optimized it for performance as follows. First, we configured MonetDB to be compiled with optimization and without debugging. Second, we set the database to read-only mode. This allows MonetDB to use more efficient MAL plans. Finally, we set the OID size to 32 Bit to be comparable with the 32-Bit TID size of CoGaDB.

We can see that CoGaDB's performance is in the same order of magnitude as that of MonetDB.[6] Hence, CoGaDB is a suitable evaluation platform for query optimization and load balancing in heterogeneous processor systems. However, MonetDB is still significantly faster for some queries (e.g., Q1.1, Q2.1, Q3.1-Q3.3, and Q4.2) even if we

[6] A more recent performance comparison between CoGaDB and MonetDB/Ocelot can be found in Appendix A.3.

enable GPU acceleration in CoGaDB. This is because MonetDB is a highly optimized system that contains many heavily tuned algorithms. If these optimizations would be included in CoGaDB, we could expect CoGaDB to be as fast as (or even outperform) MonetDB. There are two more major reasons for the difference in performance. The first is in the way the two engines parallelize queries, and the second is the data transfer bottleneck.

Parallelization Strategies

MonetDB uses a technique called mitosis, where the query plan is replicated and each thread processes its own plan, which is pinned to a fraction (horizontal partition) of the input BATs. CoGaDB evaluates each child in parallel and, therefore, the performance is bound by the longest path in the query plan. Hence, during query execution, CoGaDB does typically not (yet) utilize all cores to 100%, because not all CPU operators have a parallel version (yet). We discuss possible extensions to improve CoGaDB's performance in Section 3.5.

Data Transfer Bottleneck

CoGaDB is not generally faster than MonetDB, even though it uses an additional processor with significantly higher raw processing power than a CPU. Compared to MonetDB, which uses the CPU only, this seems like a poor result. However, the performance of query processing on GPUs does not simply scale with the number of cores of a GPU. The main problem is the data transfer between the CPU and the GPU [77].

Since CoGaDB makes heavy use of join indexes (similar to MonetDB or MonetDB's OpenCL Extension Ocelot [95]), it also suffers a performance penalty in case we need to copy the index first (but if it is cached in the GPUs memory, the speedup is significant).

On our test machine for the used scale factor of 15, a join index needs a memory capacity of 686 MB. Since CoGaDB uses half of the GPU's memory for buffering ($\approx$750 MB) to leave enough free memory for temporary data structures and results of GPU operators, only one join index fits in the GPU buffer at a time, which limits the benefit of the GPU: With more device memory, the performance of CoGaDB would increase as well. In Chapter 7, we discuss approaches that reduce this negative impact of the data transfer bottleneck.

The data transfer bottleneck is bidirectional, so CoGaDB's performance suffers also from large intermediate results, which need to be transferred back to the CPU. For all queries were CoGaDB performs poorly compared to MonetDB, the query selectivity is relatively small. For the invisible join, this means that larger position lists have to be intersected, which leads to higher intersection costs and depending on the query plan, higher data transfer costs.

Benefit of Adaptive Physical Query Optimizer

We now conduct experiments to answer research question 2. We executed the same workload from the previous experiment, but, this time, we visualize the query execution times of queries Q1.1, Q2.1, Q3.1, and Q4.1 over time in Figure 3.9. For the first ten executions, the execution times of queries have a high variance. After this initial phase, the execution times remain stable. During the unstable phase, CoGaDB has no cost models for all processors, and assigns processors to operators using a round-robin strategy. After performance models become available, CoGaDB chooses the processor (and algorithm) that are optimal according to the learned cost models. Therefore, the performance gradually improves over time and remains stable.

3.5 Future Development

In this section, we describe future developments on CoGaDB: efficient algorithms, support of the CUBE operator and other co-processors, and alternative query processing and cardinality estimation approaches.

Efficient Algorithms Although we invested much time in tuning CoGaDB's database algorithms on the CPU and the GPU, the primary focus was still in exploiting the heterogeneous nature of the modern hardware landscape and, thus, on cost estimation and load-aware operator placement. However, for future work, we will adapt approaches for efficient joins [19, 20] and aggregations [203]. Here, we have two implementation choices: Using hardware-oblivious operators written in a processor-independent language (e.g., OpenCL [95]) or tailoring algorithms for every processor type [48, 49].

CUBE Operator The CUBE operator [75] is a compute-intensive operator, which is frequently used in OLAP scenarios. Hence, it would be beneficial to investigate the potential performance gains by offloading parts of the computation to GPUs.

Support for other Co-Processors Aside GPUs, other architectures have merged for co-processors such as *Multiple Integrated Cores* MICs (e.g., Intel Xeon Phi). It would be interesting to investigate the performance properties of MICs for DBMSs to identify the optimal (co-)processor for a certain task or workload.

Query Processing Strategies Aside from tuple-at-a-time volcano-style and operator-at-a-time bulk processing, there are alternative query processing strategies such as query compilation [143] or vectorized execution [31]. It is not yet clear which strategy is optimal for heterogeneous processor environments. For a fair comparison, all strategies should be implemented in a single system.

Cardinality Estimation Our query optimizer relies on accurate cardinality estimates, which still poses major problems. Markl and others developed progressive optimization, a technique where checkpoints are inserted in the query plan [133]. In

case the cardinality estimates are too inaccurate at a checkpoint, a re-optimization is triggered. Stillger and others proposed LEO, DB2's learning optimizer, which continuously monitors cardinality estimations and iteratively corrects statistics and cardinality estimations [185]. Heimel and others offloaded selectivity estimation to the GPU, which allows them to use more compute-intensive approaches such as kernel density estimators to increase estimation accuracy [92–94]. Heimel and others [93] and Andrzejewski and others [13] investigated efficient computation of the optimal bandwidth parameter for kernel density estimators.

3.6 Related Work and Systems

In this section, we will discuss related systems. Yuan and others study the performance behavior of OLAP queries on GPUs with their system GPUDB [204] that compiles queries to driver programs, which call pre-implemented GPU operators. Thus, GPUDB performs only dispatcher and post-processing tasks on the CPU. Wang and others developed MultiQx-GPU [193], an extension of GPUDB that can process several queries in parallel on GPUs.

He and others develop GPUQP, the first DBMS accelerated by GPUs [85]. Similar to CoGaDB, GPUQP can execute each operator on the CPU or the GPU. However, GPUQP uses analytical cost models to decide on an operator placement, whereas CoGaDB relies on learned cost models and runtime refinement. Based on GPUQP, Zhang and others develop OmniDB [206], where the main focus is to exploit all heterogeneous processors while keeping a maintainable code base. They use an architecture based on adapters to decouple the database kernel from the operators.

The same goal is addressed by Ocelot [95], a hardware-oblivious database engine that extends MonetDB by a set of OpenCL operators, which can be dynamically compiled to any OpenCL-compliant device, such as CPUs, GPUs, or Xeon Phis. Thus, Ocelot leaves the handling of heterogeneity to the hardware vendors, while maintaining high performance.

Bakkum and Chakradhar developed Virginian, whose main focus is high efficiency of database queries on GPUs [17]. Therefore, it implements the opcode model, where each operator is associated with a unique id and called from a management GPU kernel. This allows Virginian to execute any query with a single GPU kernel. Mühlbauer and others developed a heterogeneity-conscious job-to-core mapping approach [141], similar to the scheduler in HyPE.

3.7 Conclusion

In this chapter, we presented CoGaDB, a system designed to target the hardware heterogeneity on the query optimizer level. We outlined design decisions and implementation details of CoGaDB and discussed how we combine a modern, GPU-accelerated DBMS with a hardware-oblivious query optimizer, which we discuss in detail in Chapter 5 and Chapter 6.

Our evaluation shows that the design, where an optimizer has no detailed knowledge of the hardware, is feasible. Furthermore, we showed that such a system can be competitive to highly optimized main-memory databases such as MonetDB.

4. A Survey of GPU-accelerated DBMSs

In Chapter 2, we investigated the general design space of co-processor-accelerated DBMSs. In Chapter 3, we discussed one example system in detail. In this chapter, we survey other GPU-accelerated database prototypes to understand the design decisions and performance optimizations of other systems and widen our perspective on co-processor-accelerated data management. Therefore, we conduct an in-depth literature survey of eight GPU-accelerated database management systems to validate and refine our theoretical discussions in Chapter 2. This complements our findings in proposing a reference architecture. In detail, we make the following contributions:

1. We discuss eight *GPU-accelerated DBMSs* to review the state-of-the-art, collect prominent findings, and complement our discussion on a GPU-aware DBMS architecture.
2. We create a classification of required architectural properties of GPU-accelerated DBMSs.
3. We summarize optimizations implemented by the surveyed systems and derive a general set of optimizations that a GPU-accelerated DBMS should implement.
4. We propose a reference architecture for GPU-accelerated DBMSs, which provides insights on how we can integrate GPU-acceleration in main-memory DBMSs.

We find that GPU-accelerated DBMSs should be in-memory column stores, should use the block-at-a-time processing model and exploit all available processing devices for query processing by using a GPU-aware query optimizer. Thus, main memory DBMSs are similar to GPU-accelerated DBMSs, and most in-memory, column-oriented DBMSs can be extended to efficiently support co-processing on GPUs.

First, we describe our research methodology. Second, we discuss the architectural properties of all systems that meet our survey selection criteria. Third, we classify the systems according to our design criteria (cf. Section 2.2). Based on our classification, we then discuss further optimization techniques used in the surveyed systems. Then, we derive a reference architecture for GPU-accelerated DBMSs based on our results. Finally, we will use this reference architecture for GPU-accelerated DBMSs to identify a set of extensions that is required to extend existing main-memory DBMSs to support efficient GPU co-processing.

4.1 Research Methodology

In this section, we state the research questions that drive our survey. Then, we describe the selection criteria to find suitable DBMS architectures in the field of GPU-acceleration. Afterwards, we discuss the properties we focus on in our survey. This properties will be used as base for our classification.

4.1.1 Research Questions

RQ1: Are there recurring architectural properties among the surveyed systems?

RQ2: Are there application-specific classes of architectural properties?

RQ3: Can we infer a reference architecture for GPU-accelerated DBMSs based on existing GPU-accelerated DBMSs?

RQ4: How can we extend existing main-memory DBMSs to efficiently support data processing on GPUs?

4.1.2 Selection Criteria.

Since this survey should cover relational GPU-accelerated DBMS, we only consider systems that are capable of using the GPU for most relational operations. That is, we disregard stand-alone approaches for accelerating a certain relational operator (e.g., He and others [86, 88]), special co-processing techniques (e.g., Pirk and others [157]), or other – non data-processing related – applications for GPUs in database systems [94]. Furthermore, we will not discuss systems using other data models than the relational model, such as graph databases (e.g., Medusa from Zhong and He [207, 208]) or MapReduce (e.g., Mars from He and others [84]). Also, given that publications, such as research papers or whitepapers, often lack important architectural informations, we strongly preferred systems that made their source code publicly available. This allowed us to analyze the source code in order to correctly classify the system.

4.1.3 Comparison Properties.

According to the design decisions discussed in Section 2.2, we present for each GPU-accelerated DBMS the storage system, the storage and processing model, query placement and query optimization, and support for transaction processing. The information for this comparison is taken either directly from analyzing the source code – if available –, or from reading through published articles about the system. If a properties is not applicable for a system, we mark it as not applicable and focus on unique features instead.

4.2 GPU-accelerated DBMS

Based on the discussed selection criteria, we identified the following eight academic[1] systems that are relevant for our survey:

System	Institute	Year	Open Source	Ref.
CoGaDB	Universität Magdeburg	2013	yes	[35, 45]
GPUDB	Ohio State University	2013	yes	[204]
GPUQP	Hong Kong University of Science and Technology	2007	yes	[85]
GPUTx	Nanyang Technological University	2011	**no**	[87]
MapD	Massachusetts Institute of Technology	2013	**no**	[138]
Ocelot	Technische Universität Berlin	2013	yes	[95]
OmniDB	Nanyang Technological University	2013	yes	[206]
Virginian	NEC Laboratories America	2012	yes	[17]

In Figure 4.1, we illustrate the chronological order in which the first publications for each system were published.It is clearly visible that most systems were developed very recently and only few systems are based on older systems. Hence, we expect little influence on the concrete DBMS architecture between each other and hence, a strong external validity of our results.

CoGaDB

CoGaDB focuses on GPU-aware query optimization to achieve efficient co-processor utilization during query processing. We discuss CoGaDB in detail in Chapter 3, but we briefly summarize the design decisions for completeness.

[1]Note that we deliberately excluded commercial systems such as Jedox [1] or Parstream [2], because they are neither available as open source nor have publications available that provide full architectural details.

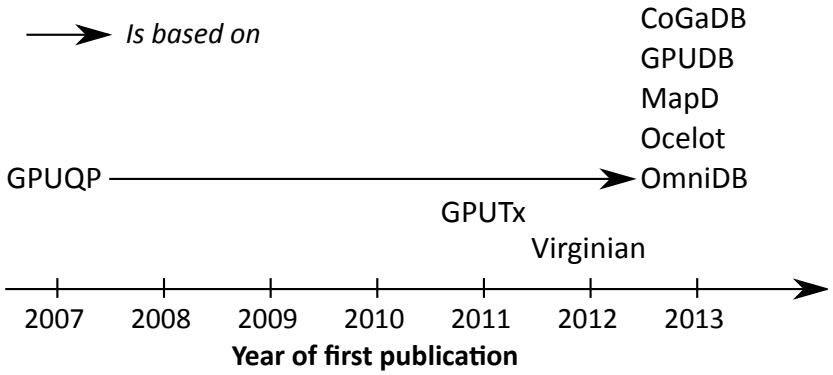


Figure 4.1: Time line of surveyed systems.

Storage System:

CoGaDB persists data on disk, but loads the complete database into main memory on startup. If the database is larger than the main memory, CoGaDB relies on the operating system's virtual memory management to swap the least recently used memory pages on disk.

Storage Model:

CoGaDB stores data in data structures optimized for in-memory databases. Hence, it stores the data column-wise and compresses VARCHAR columns using dictionary encoding [26]. Furthermore, the data has the same format when stored in the CPU's or the GPU's memory.

Processing Model:

CoGaDB uses the operator-at-a-time bulk processing model to make efficient use of the memory hierarchy. This is the basis for efficient query processing using all processing resources.

Query Placement & Optimization:

CoGaDB uses the *Hybrid Query Processing Engine* (HyPE) as physical optimizer [35]. HyPE optimizes physical query plans to increase inter-device parallelism by keeping track of the load condition on all (co-)processors (e.g., the CPU or the GPU).

Transactions:

Not supported.

GPUDB

In order to study the performance behaviour of OLAP queries on GPUs, Yuan and others developed GPUDB[2] [204].

[2]Source code available at: https://code.google.com/p/gpudb/.

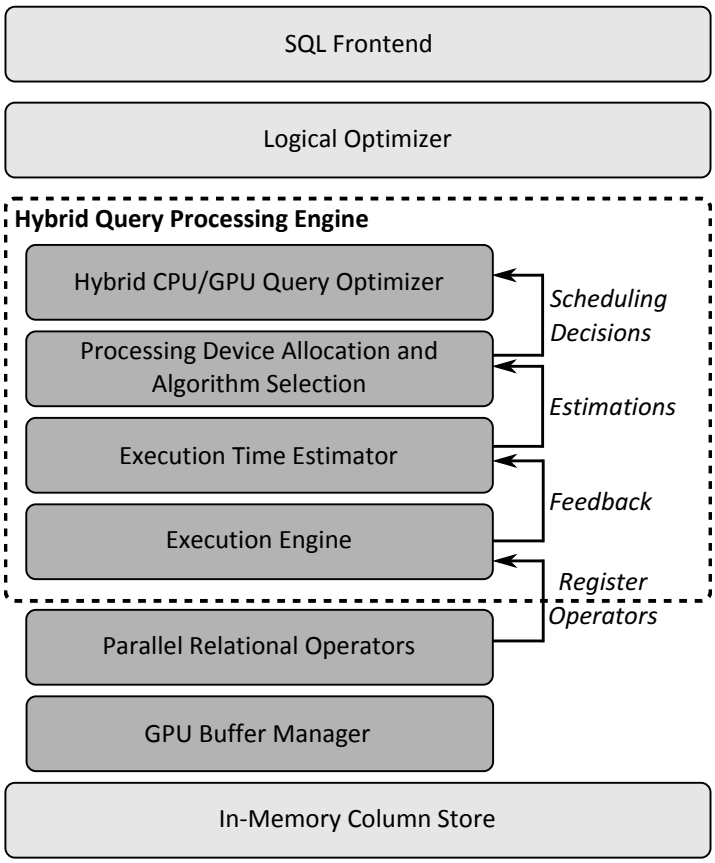


Figure 4.2: The architecture of CoGaDB, taken from [41]

.

Storage System:

GPUDB keeps the database in the CPU's main memory to avoid the hard-disk bottleneck. Yuan and others identified a crucial optimization for main-memory DBMS with respect to GPU accelerated execution: In case data is stored in pinned host memory, query execution times can significantly improve (i.e., Yuan and others observed speedups up to 6.5x for certain queries of the *Star Schema Benchmark* (SSB) [165]).

Storage Model:

GPUDB stores the data column-wise because GPUDB is optimized for warehousing workloads. Additionally, GPUDB supports common compression techniques (run length encoding, bit encoding, and dictionary encoding) to decrease the impact of the PCIe bottleneck and to accelerate data processing.

Processing Model:

GPUDB uses a block-oriented processing model: Blocks are kept in GPU RAM until they are completely processed. This processing model is also known as vectorized

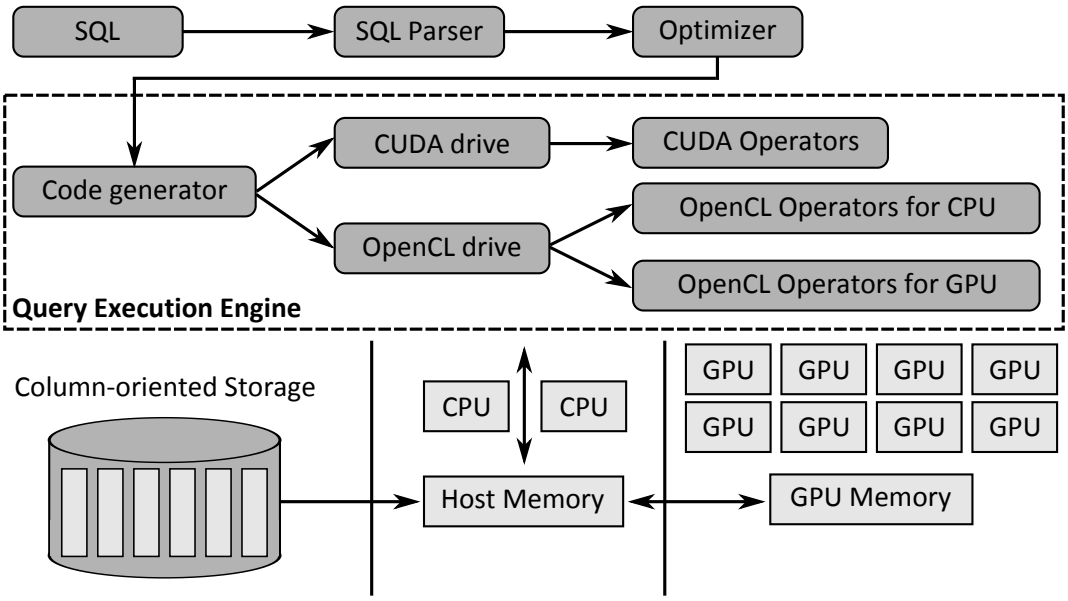


Figure 4.3: GPUDB: Query engine architecture, taken from [204]

processing [172]. Thus, the PCIe bottleneck can be further reduced by overlapping data transfers with computation. For certain queries, Yuan and others observed speedups up to 2.5x compared to no overlapping of processing and data transfers.

GPUDB compiles queries to *driver programs*. A driver program executes a query by calling pre-implemented GPU operators. Hence, GPUDB executes all queries on the GPU and the CPU performs only dispatcher and post processing tasks (i.e., the CPU is used less than 10 % of the time during processing SSB queries [204]).

Query Placement & Optimization:

GPUDB has no support for executing queries on the CPU and GPU in parallel.

Transactions:

Not supported.

GPUQP

He and others developed GPUQP[3], a relational query processing system, which stores data in-memory and uses the GPU to accelerate query processing [85]. In GPUQP, each relational operator can be executed on the CPU or the GPU.

Storage System:

GPUQP supports in-memory and disk-based processing. Apparently, GPUQP also attempts to keep data cached in GPU memory. Unfortunately, the authors do not provide any details about the used data placement strategy.

[3]Source code available at: http://www.cse.ust.hk/gpuqp/.

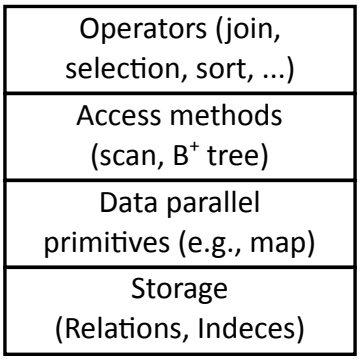


Figure 4.4: Execution engine of GPUQP, taken from [85]

Storage Model:

Furthermore, GPUQP makes use of columnar storage and query processing, which fits the hardware capabilities of modern CPUs and GPUs.

Processing Model:

GPUQP's basic processing strategy is operator-at-a-time bulk processing. However, GPUQP is also capable of partitioning data for one operator and execute the operator on the CPU and the GPU concurrently. Nevertheless, the impact on the overall performance is small [85].

Query Placement & Optimization:

GPUQP combines a Selinger-style optimizer [179] with an analytical cost model to select the cheapest query plan. For each operator, GPUQP allocates either the CPU, the GPU, or both processors (partitioned execution). The query optimizer splits a query plan to multiple sub-plans containing at most ten operators. For each sub-query, all possible plans are created and the cheapest sub-plan is selected. Finally, GPUQP combines the sub-plans to a final physical query plan.

He and others focus on optimizing single queries and do not discuss multi-query optimization. Furthermore, load-aware query scheduling is not considered and there is no discussion of scenarios with multiple GPUs.

Transactions:

Not supported.

GPUTx

In order to investigate relational transaction processing on graphics cards, He and others developed GPUTx, a transaction processing engine that runs on the GPU [87].

Storage System & Model:

GPUTx keeps all OLTP data inside the GPU's memory to minimize the impact of the PCIe bottleneck. It also applies a columnar data layout to fit the characteristics of modern GPUs.

Processing Model:

The processing model is not built on relational operators as in GPUQP. Instead, GPUTx executes pre-compiled stored procedures, which are grouped into one GPU kernel. Incoming transactions are grouped in *bulks*, which are sets of transactions that are executed in parallel on the GPU.

Query Placement & Optimization:

Since GPUTx performs the complete data processing on the GPU, query placement approaches are not needed.

Transactions:

GPUTx is the only system in our survey – and that we are aware of – that supports running transactions on a GPU. It implements three basic transaction protocols: Two-phase locking, partition-based execution and k-set-based execution. The major finding of GPUTx is that locking-based protocols do not work well on GPUs. Instead, lock-free protocols such as partition-based execution or k-set should be used.

MapD

Mostak developed MapD, which is a data processing and visualization engine, combining traditional query processing capabilities of DBMSs with advanced analytic and visualization functionality [138]. One application scenario is the visualization of twitter messages on a road map[4], in which the geographical position of tweets is shown and visualized as heat map.

Storage System:

The data processing component of MapD is a relational DBMS, which can handle data volumes that do not fit the main memory. MapD also tries to keep as much data in-memory as possible to avoid disk accesses.

Storage Model:

MapD stores data in a columnar layout, and further partitions columns into *chunks*. A chunk is the basic unit of MapD's memory manager. The basic processing model of MapD is processing one operator-at-a-time. Due to the partitioning of data into chunks, it is also possible to process on a per-chunk basis. Hence, MapD is capable of applying block-oriented processing.

[4] http://mapd.csail.mit.edu/tweetmap/

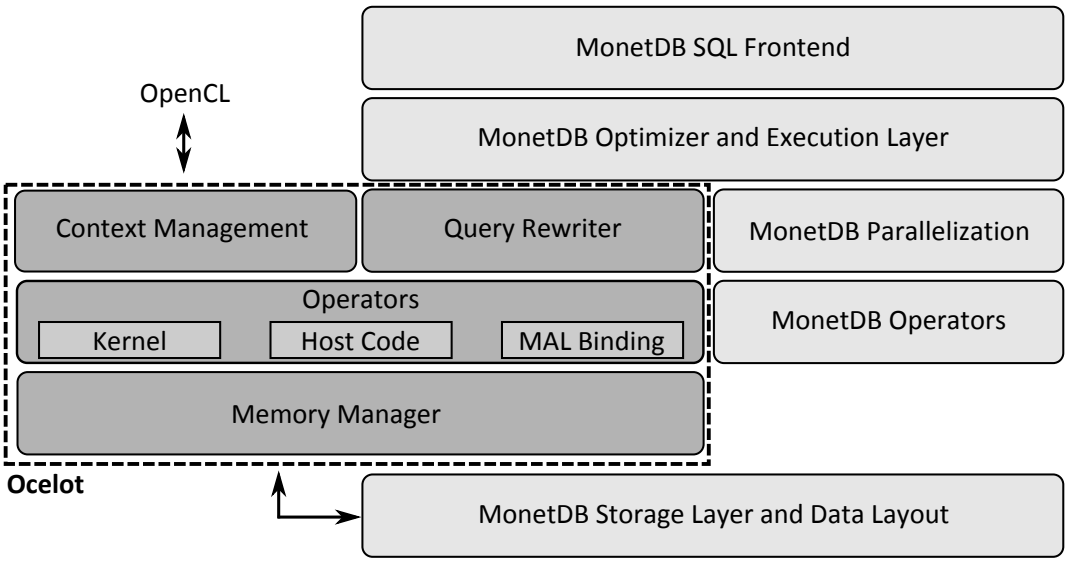


Figure 4.5: The architecture of Ocelot, taken from [95]

Processing Model:

MapD processes queries by compiling a query to executable code for the CPU and GPU.

Query Placement & Optimization:

The optimizer tries to split a query plan in parts, and processes each part on the most suitable processing device (e.g., text search using an index on the CPU and table scans on the GPU). MapD does not assume that an input data set fits in GPU RAM, and it applies a streaming mechanism for data processing.

Transactions:

Not supported.

Ocelot

Heimel and others develop Ocelot[5], which is an OpenCL extension of MonetDB, enabling operator execution on any OpenCL capable device, including CPUs and GPUs [90, 95].

Storage System:

Ocelot's storage system is built on top of the in-memory model of MonetDB. Input data is automatically transferred from MonetDB to the GPU when needed by an operator. In order to avoid expensive transfers, operator results are typically kept on the GPU. They are only returned at the end of a query, or if the device memory is too filled to fulfill requests. Additionally, Ocelot implements a device cache to keep relevant input data available on the GPU.

[5]Source code available at: https://bitbucket.org/msaecker/monetdb-opencl.

Storage Model:

Ocelot/MonetDB stores data column-wise in *Binary Association Tables* (BATs). Each BAT consists of two columns: One (optional) head storing object identifiers, and one (mandatory) tail storing the actual values.

Processing Model:

Ocelot inherits the operator-at-a-time bulk processing model of MonetDB, but extends it by introducing lazy evaluation and making heavy use of the OpenCL event model to forward operator dependency information to the GPU. This allows the OpenCL driver to automatically interleave and reorder operations, e.g., to hide transfer latencies by overlapping the transfer with the execution of a previous operator.

Query Placement & Optimization:

In MonetDB, each query plan is represented in the *MonetDB Assembly Language* (MAL) [100]. Ocelot reuses this infrastructures and adds a new query optimizer, which rewrites MAL plans by replacing data processing MAL instructions of vanilla MonetDB with the highly parallel OpenCL MAL instructions of Ocelot.

Query Placement & Optimization:

Ocelot does not support cross-device processing, meaning it executes the complete workload either on the CPU or on the GPU.

Transactions:

Not supported.

OmniDB

Zhang and others developed OmniDB[6], a GPU-accelerated DBMS aiming for good code maintainability while exploiting all hardware resources for query processing [206]. The basic idea is to create a hardware oblivious database kernel (qkernel), which accesses the hardware via *adaptors*. Each adapter implements a common set of operators decoupling the hardware from the database kernel.

Storage System & Model:

OmniDB is based on GPUQP, and hence, has similar architectural properties to GPUQP. OmniDB keeps data in-memory in a column-oriented data layout.

[6]Source code available at: https://code.google.com/p/omnidb-paralleldbonapu/.

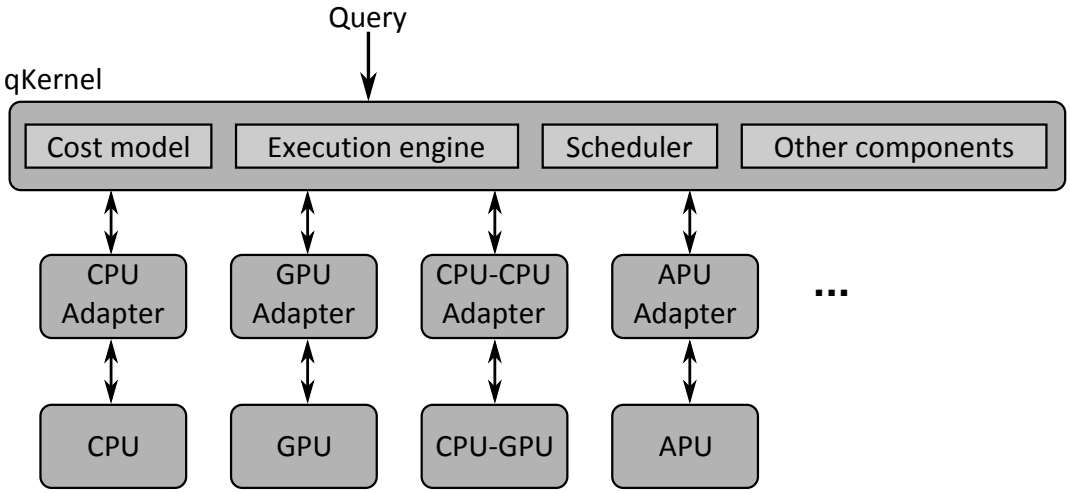


Figure 4.6: OmniDB: Kernel adapter design, taken from [206]

Processing Model:

OmniDB schedules and processes *work units*, which can vary in granularity (e.g., a work unit can be a query, an operator, or a chunk of tuples). Although it is not explicitly mentioned in the paper [206], the fact that OmniDB can process also chunks of tuples is a strong indicator that it supports block-oriented processing.

Query Placement & Optimization:

Regarding query placement and optimization, OmniDB chooses the processing device with highest throughput for a *work unit*. To avoid overloading a single device, OmniDB's scheduler ensures that the workload on one processing device may not exceed a certain percentage of the average workload on all processing devices. The cost model relies on the adapters to provide cost functions for the underlying processing devices.

Transactions:

Not supported.

Virginian

Bakkum and others develop Virginian[7], which is a GPU-accelerated DBMS keeping data in main memory and supporting filter and aggregation operations on all processing devices [17].

Storage System:

Virginian uses no traditional caching of operators, but *uniform virtual addressing* (UVA). This technique allows a GPU kernel to directly access data stored in pinned host memory. The accessed data is transferred over the bus transparently to the device and efficiently overlaps computation and data transfers.

[7]Source code available at: https://github.com/bakks/virginian.

Storage Model:

Virgnian implements a data structure called *tablet*, which stores fixed size values column oriented. Additionally, tables can handle variable sized data types such as strings, which are stored in a dedicated section inside the tablet. Thus, Virginian supports strings on the GPU. This is a major difference to other GPU-accelerated DBMSs, which apply dictionary compression on strings first and work only on compressed values in the GPU RAM.

Processing Model:

Virginian uses operator-at-a-time processing as basic query-processing model. It implements an alternative processing scheme. While most systems call a sequence of highly parallel primitives requiring one new kernel invocation per primitive, Virginian uses the opcode model, which combines all primitives in a single kernel. This avoids writing data back to global memory and reading it again in the next kernel ultimately resulting in block-wise processing on the GPU.

Query Placement & Optimization:

Virginian can either process queries on the CPU or on the GPU. Thus, there is no mechanism splitting up the workload between CPU and GPU processing devices and hence, no hybrid query optimizer is available.

Transactions:

Not supported.

4.3 Classification

We now classify the surveyed systems according to the architectural properties discussed in Section 2.2.

4.3.1 Storage System

For all eight systems, it is clearly visible that they are designed with main-memory databases in mind, keeping a large fraction of the database in the CPU's main memory. GPUQP and MapD also support disk-based data. However, since fetching data from disk is very expensive compared to transferring data over the PCIe bus [85], MapD and GPUQP also keep as much data as possible in main memory. Therefore, we mark all systems as main-memory storage and GPUQP and MapD additionally as disk-based storage.

DBMS	Storage System		Storage Model	
	Main-Memory Storage	Disk-based Storage	Column Store	Row Store
CoGaDB	✓	×	✓	×
GPUDB	✓	×	✓	×
GPUQP	✓	✓	✓	×
GPUTx	✓	×	✓	×
MapD	✓	✓	✓	×
Ocelot	✓	×	✓	×
OmniDB	✓	×	✓	×
Virginian	✓	×	○	○

Table 4.1: Classification of Storage System and Storage Model – *Legend:* ✓– Supported, × – Not Supported, ○ – Not Applicable

4.3.2 Storage Model

All systems store their data in a columnar layout, there is no system using row-oriented storage. One exception is Virginian, which stores data mainly column-oriented, but also keeps complete rows inside a tablet data structure. This representation is similar to PAX, which stores rows on one page, but stores all records column-wise inside a page [9]. As the tablet data structure does neither fit well for column store or row store, we classify it as not applicable in Table 4.1.

4.3.3 Processing Model

The processing model varies between the surveyed systems. The first observation is that no system uses a traditional tuple-at-a-time volcano model [74], as was hypothesized in Section 2.2. Most systems support operator-at-a-time bulk processing [132]. The only exception is GPUTx, which does not support OLAP workloads, because it is an optimized OLTP engine. Since we cannot assign GPUTx to any processing model in our classification, we mark it as not applicable in Table 4.2. GPUDB, MapD, OmniDB, and Virginian have basic capabilities for block-oriented processing. Additionally, GPUDB and MapD apply a compilation-based query processing strategy.[8] Virginian does not support query compilation. Instead, it uses a single GPU kernel that implements a virtual machine, which calls other GPU kernels (the primitives) in the context of the same kernel, efficiently saving the overhead of reading and writing the result from the GPU's main memory.

4.3.4 Query Placement and Optimization

We identify two major groups of systems: The first group performs nearly all data processing on one processing device (GPUDB, GPUTx, Ocelot, Virginian), whereas

[8]Note that both systems still apply a block-oriented processing model. This is due to the nature of compilation-based strategies, as discussed in Section 2.2.

DBMS	Processing Model		
	Operator-at-a-Time	Block-at-a-Time	Just-in-Time Compilation
CoGaDB	✓	×	×
GPUDB	✓	✓	✓
GPUQP	✓	×	×
GPUTx	○	○	○
MapD	✓	✓	✓
Ocelot	✓	×	×
OmniDB	✓	✓	×
Virginian	✓	✓	×

Table 4.2: Classification of Processing Model – *Legend:* ✓– Supported, × – Not Supported, ○ – Not Applicable

DBMS	Query Processing	
	Single-Device Processing	Cross-Device Processing
CoGaDB	✓	✓
GPUDB	✓	×
GPUQP	✓	✓
GPUTx	✓	×
MapD	✓	✓
Ocelot	✓	×
OmniDB	✓	✓
Virginian	✓	×

Table 4.3: Classification of Query Processing – *Legend:* ✓– Supported, × – Not Supported, ○ – Not Applicable

the second group is capable of splitting the workload in parts, which are then processed in parallel on the CPU and the GPU (CoGaDB, GPUQP, MapD, OmniDB). We mark systems in the first group as systems that support only single-device processing (SDP), whereas systems of the second group are capable of using multiple devices and thereby allowing cross-device processing (CDP). Note that a system supporting CDP is also capable of executing the complete workload on one processing device (SDP). The hybrid query optimization approaches of CoGaDB, GPUQP, MapD, and OmniDB are mostly greedy strategies or other simple heuristics. It is still an open question how to efficiently trade off between inter-processor parallelization and costly data transfers to achieve optimal performance. For instance: So far, there are no query optimization approaches for machines having multiple GPUs.

4.3.5 Transaction Processing

Apart from GPUTx, none of the surveyed GPU-accelerated DBMSs support transactions. GPUTx keeps data strictly in the GPU's RAM, and needs to transfer only incoming transactions to the GPU and the result back to the CPU. Since GPUTx achieved a 4-10 times higher throughput than a comparable CPU-based OLTP engine, there is a need for further research in the area of transaction processing in GPU-accelerated DBMSs so that OLTP systems can also benefit from GPU acceleration. Apparently, online analytical processing and online transactional processing can be significantly accelerated by using GPU acceleration. However, it is not yet clear which workload type is more suitable for which processing device type. Furthermore, the efficient combination of OLTP/OLAP workloads is still an active research field (e.g., Kemper and Neumann [114]). Thus, it is an open question whether and under which circumstances GPU-acceleration is beneficial for such hybrid OLTP/OLAP workloads.

4.3.6 Portability

The only GPU-accelerated DBMSs having a portable, hardware-oblivious database architecture are Ocelot and OmniDB. All other systems are either tailored to a vendor specific programming framework or have no technique to hide the details of the device-specific operators in the architecture. Ocelot's approach has the advantage that only a single set of parallel database operators has to be implemented, which can then be mapped to all processing devices supporting OpenCL (e.g., CPUs, GPUs, or Xeon Phis). By contrast, OmniDB uses an adapter interface, in which each adapter provides a set of operators and cost functions for a certain processing-device type. It is unclear, which approach will lead to the best performance/maintainability ratio, and how large the performance loss is compared to a hardware-aware system. However, if portability can be achieved with only a small performance degradation, it would substantially benefit the maintainability and applicability of GPU-accelerated DBMSs [206]. Hence, the trend towards hardware-oblivious DBMSs is likely to continue.

4.4 Optimizations for GPU-accelerated DBMSs

We will now discuss and summarize potential optimizations, which a GPU-accelerated DBMS may implement to make full use of the underlying hardware in a hybrid CPU/GPU system. Additionally, we briefly discuss existing approaches for each optimization. As already discussed, data transfers have the highest impact on GPU-accelerated DBMS performance. Hence, every optimization avoiding or minimizing the impact of data transfers are mandatory. We refer to these optimizations as cross-device optimizations. Based on our surveyed systems, we could identify the following *cross-device optimizations*:

Efficient Data Placement Strategy: There are two possibilities to manage the GPU RAM. The first possibility is an explicit management of data on GPUs using a

DBMS	Transaction Support	Portability	
		Hardware Aware	Hardware Oblivious
CoGaDB	×	✓	×
GPUDB	×	✓	×
GPUQP	×	✓	×
GPUTx	✓	✓	×
MapD	×	✓	×
Ocelot	×	×	✓
OmniDB	×	×	✓
Virginian	×	✓	×

Table 4.4: Classification of Transaction Support and Portability – *Legend:* ✓– Supported, × – Not Supported, ∘ – Not Applicable

buffer-management algorithm. The second possibility is using mechanisms such as *Unified Virtual Addressing* (UVA), which enables a GPU kernel to directly access the main memory. Kaldewey and others observed a significant performance gain (3-8x) using UVA for Hash Joins on the GPU compared to the CPU [109]. Furthermore, data has not to be kept consistent between CPU and GPU, because there is no "real" copy in the GPU RAM. However, this advantage can also be a disadvantage, because caching data in the GPU RAM can avoid the data transfer from the CPU to the GPU.

GPU-aware Query Optimizer: A GPU-accelerated DBMS should make use of all processing devices to maximize performance. Therefore, it should offload operations to the GPU. However, offloading single operations of a query plan does not necessarily accelerate performance. Hence, a GPU-aware optimizer has to identify sub plans of a query plan, which it can process on the CPU or the GPU [85]. Furthermore, the resulting plan should minimize the number of copy operations [40]. Since optimizers are typically cost based, a GPU-accelerated DBMS needs for each GPU operator a cost model. The most common approach is to use analytical models (e.g., Manegold and others for the CPU [130] and He and others for the GPU [85]). However, with the increasing hardware complexity, machine-learning-based models become increasingly popular [37].

Data Compression: The data placement and query optimization techniques attempt to avoid data transfers as much as possible. To reduce overhead in case a GPU-accelerated DBMS has to perform data transfers, the data volume can be reduced by compression techniques. Thus, compression can significantly decrease processing costs [204]. Fang and others discussed an approach, which combines different lightweight compression techniques to compress data at the GPU [64]. They developed a planner for cascading compression techniques, which decides on a suitable subset and order of available compression techniques. Przymus and Kaczmarski

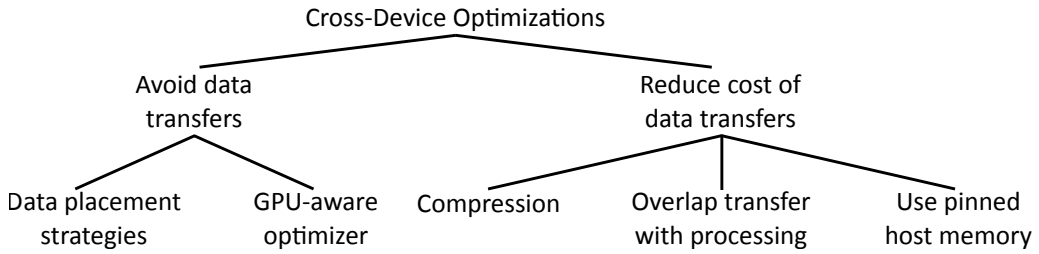


Figure 4.7: Cross-device optimizations

focused on compression for time-series databases on the GPU [161]. Andrzejewski and Wrembel discussed compression of bitmap indexes on the GPU [14].

Overlap of Data Transfer and Processing: The second way to accelerate processing, in case a data transfer needs to performed, is overlapping the execution of a GPU operator with a data transfer operation [17, 204]. This optimization keeps all hardware components busy, and basically narrows down the performance of the system to the PCIe bus bandwidth.

Pinned Host Memory: The third way to accelerate query processing in case we have to perform a copy operation is keeping data in pinned host memory. This optimization saves one indirection, because the DMA controller can transmit data directly to the device [204]. Otherwise, data has to be copied in pinned memory first, introducing additional latency in data transmission. However, using pinned host memory has the drawback that the amount of available pinned host memory is much smaller than the amount of unpinned memory (i.e., memory that can be paged to disk by the virtual memory manager) [174]. Therefore, a GPU-accelerated DBMS has to decide which data it should keep in pinned host memory. It is still an open issue how much memory should be spent on a pinned host memory buffer for faster data transfers to the GPU.

Figure 4.7 classifies the identified cross-device optimizations.

The second class of optimizations we identified, targets the efficiency of operator execution on a single processing device. We refer to this class of optimizations as *device-dependent optimizations*. Since we focus on GPU-aware systems, we only discuss optimizations for GPUs. Based on the surveyed systems, we summarize the following GPU-dependent optimizations:

Block-oriented Query Processing: A GPU-accelerated DBMS can avoid the overhead of writing results of an operator back to a processing device's main memory by processing data on a per block basis rather than on a per operator basis. The idea is to process data already stored in the cache (CPU) or shared memory (GPU), which saves memory bandwidth and significantly increases performance

of query processing [31, 204]. Additionally, block-oriented processing is a necessary prerequisite for overlapping processing and data transfer for single operations and allows for a more fine grained workload distribution on available processing devices [206]. Note that traditional pipelining of blocks between GPU operators is not possible, because inter-kernel communication is undefined [40]. While launching a new kernel for each block is likely to be expensive, query compilation and kernel fusion are promising ways to allow block-oriented processing on the GPU as well.

Compilation-based Query Processing: Compiling queries to executable code is a common optimization in main-memory DBMSs [54, 143, 192]. As already discussed, query compilation allows for block-oriented processing on GPUs as well and achieves a significant speedup compared to primitive-based query processing (e.g., operator-at-a-time processing [85]). However, query compilation introduces additional overhead, because compiling a query to executable code typically is more expensive than building a physical query execution plan. Yuan and others overcome this shortcoming by pre-compiling operators. Thus, they only need to compile the query plan itself to a driver program [204]. A similar approach called *kernel weaver* is used by Wu and others [198]. They combine CUDA kernels for relational primitives into one kernel. This has the advantage that the optimization scope is larger and the compiler can perform more optimizations. However, the disadvantage is the increased compilation time. Rauhe and others introduce in their approach two processing phases: compute and accumulate. In the compute phase, a number of threads are assigned to a partition of the input data and each thread performs all operations of a query on one tuple and then, continues with the next tuple, until the thread processed its partition. In the accumulate phase, the intermediate results are combined to the final result [169].

All-in-one Kernel: A promising alternative to compilation-based approaches is to combine all relational primitives into one kernel [17]. Thus, a relational query has to be translated to a sequence of op codes. An op code identifies the next primitive to be executed. Therefore, it is basically an on-GPU virtual machine, which saves the initial overhead of query compilation. However, the drawback is a limited optimization scope compared to kernel weaver [198].

Portability: Until now, we mainly discussed performance optimizations. However, each of the discussed optimizations are mainly implemented device dependent. This increases the overall complexity of a GPU-accelerated DBMS. The problem gets even more complex with new processing device types such as accelerated processing units or the Intel Xeon Phi. Heimel and others implemented a hardware oblivious DBMS kernel in OpenCL and still achieved a significant acceleration of query processing [95]. Zhang and others implemented *q-kernel*, a hardware-oblivious database kernel using device adapters to the underlying processing devices [206]. It is still not clear which part of a kernel should be hardware oblivious and which part should be hardware aware. For the parts that have to be hardware

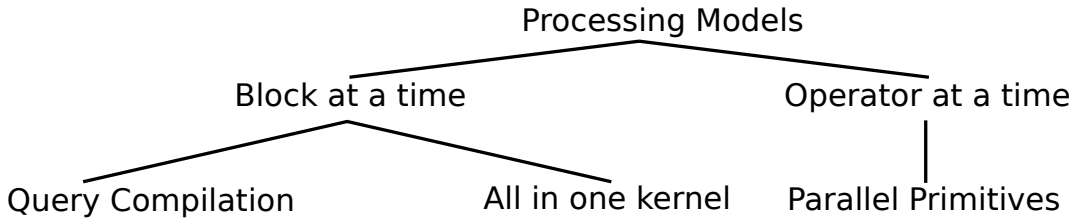


Figure 4.8: Device-dependent optimizations: Efficient processing models

aware, modern software engineering methods such as software product lines can be used to manage the GPU-accelerated DBMS's complexity [48].

Figure 4.8 illustrates the identified device-dependent optimizations and the relationships between them.

4.5 A Reference Architecture for GPU-accelerated DBMSs

Based on our in-depth survey of existing GPU-accelerated DBMSs, we now derive a reference architecture for GPU-accelerated DBMSs. After careful consideration of all surveyed systems, we decided to use the GPUQP [85]/OmniDB [206] architecture as basis for our reference architecture, because they already include a major part of the common properties of the surveyed systems. We illustrate the reference architecture in Figure 4.9.

We will describe the query-evaluation process in a top-down view. On the upper levels of the query stack, a GPU-accelerated DMBS is virtually identical to a "traditional" DBMS. It includes functionality for integrity control, parsing SQL queries, and performing logical optimizations on queries. Major differences between main-memory DBMSs and GPU-accelerated DBMSs emerge in the physical optimizer. While classical systems choose the most suitable access structure and algorithm to operate on the access structure, a GPU-accelerated DBMS has to additionally decide for each operator on a processing device. For this task, a GPU-accelerated DBMS needs refined[9] cost models that also predict the cost for GPU and CPU operations. Based on these estimates, a scheduler can allocate the cheapest processing device. Furthermore, a query should make use of multiple processing devices to speed up execution. Hence, the physical optimizer has to optimize hybrid CPU/GPU query plans, which significantly increases the optimization space.

Relational operations are implemented in the next layer. These operators typically use access structures to process data. In GPU-accelerated DBMSs, access structures have

[9]Since these models need to be able to estimate comparable operator runtimes across different devices, we and others [35] argue that dynamic cost models, which apply techniques from Machine Learning to adapt to the current hardware, are likely required here.

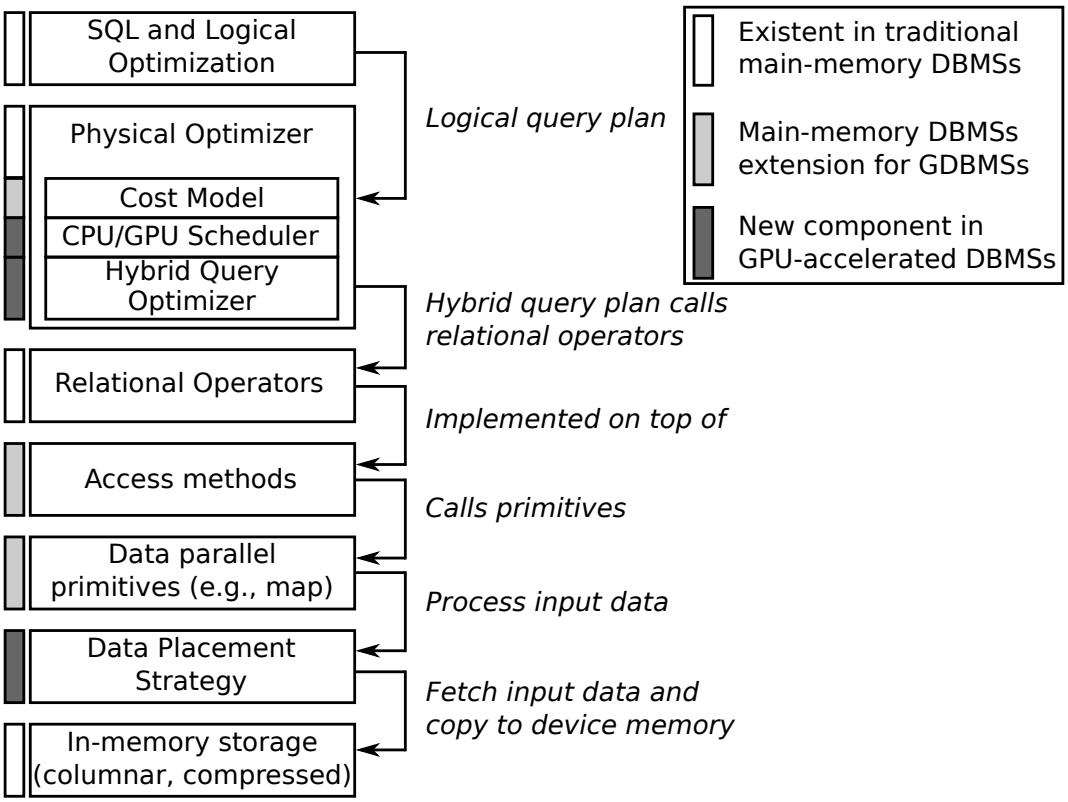


Figure 4.9: Layered architecture of GPU-accelerated DBMSs

to be reimplemented on GPUs to achieve a high efficiency. However, depending on the processing device chosen by the CPU/GPU scheduler, different access structures are available. This is an additional dependency the query optimizer needs to take into account.

Then, a set of parallel primitives can be applied to an access structure to process a query. In this component, the massive parallelism of CPUs and GPUs is fully used to speed up query processing. However, a GPU operator can only work on data stored in GPU memory. Hence, all access structures are built on top of a data-placement component, that caches data on a certain processing device, depending on the access patterns of the workload (e.g., certain columns for column scans or certain nodes of tree indexes [23, 176]). Note that the data-placement strategy is the most performance critical component in a GPU-accelerated DBMS due to the major performance impact of data transfers.

The backbone of a GPU-accelerated DBMS is a typical in-memory storage, which frequently stores data in a column-oriented format.[10] Compression techniques are not only

[10]We are aware that some in-memory DBMSs can also store data row-oriented, such as HyPer [114]. However, in GPU-accelerated DBMSs, row-oriented storage either increases the data volume to be

beneficial in keeping the major part of a database in-memory, compression also reduces the impact of the PCIe bottleneck.

4.6 Summary: Extension points for Main-Memory DBMSs

In summary, we can extend most main-memory DBMSs supporting column-oriented data layout and bulk processing to be GPU-accelerated DBMSs. We identify the following extension points: Cost models, CPU/GPU scheduler, hybrid query optimizer, access structures and algorithms for the GPU, and a data placement strategy.

Cost Models: For each processor, we need to estimate the execution time of an operator. This can be either done by analytical cost models (e.g., Manegold and others for CPUs [130] and He and others for GPUs [85]) or learning-based approaches (e.g., Ilić and Sousa [102]).

CPU/GPU Scheduler: Based on the cost models, a scheduler needs to allocate processing devices for a set of operators (e.g., CHPS from Ilić and Sousa, or StarPU from Augonnet and others [15]).

Hybrid Query Optimizer: The query optimizer needs to consider the data transfer bottleneck and memory requirements of operators to create a suitable physical execution plan. Thus, the optimizer should make use of cost models, a CPU/GPU scheduler, and heuristics minimizing the time penalty of data transfers.

Access structures and algorithms for the GPU: In order to support GPU-acceleration, a DBMS needs to implement access structures on the GPU (e.g., columns or B^+-trees) and operators that work on them. Here, the most approaches were developed [18, 57, 85, 88, 156, 157].

Data Placement Strategy: A DBMS needs to keep track of which data is stored on the GPU, and which access structure needs to be transferred to GPU memory [85]. Aside from a manual memory management, it is also possible to use techniques such as UVA and let the GPU driver handle the data transfers transparently to the DBMS [204]. However, this may result in less efficiency because a manual memory management can exploit knowledge about the DBMS and the workload.

Implementing these extensions is a necessary precondition for a DBMS to support GPU co-processing efficiently.

transfered or requires a projection operation before the transfer. A row-oriented layout also makes it difficult to achieve optimal memory access patterns on a GPU.

4.7 Open Challenges and Research Questions

In this section, we identify *open challenges* for GPU-accelerated DBMSs. We differentiate between two major classes of challenges, namely the IO bottleneck, which includes disk IO as well as data transfers between CPU and GPU, and query optimization.

4.7.1 IO Bottleneck

In a GPU-accelerated DBMS, there are two major IO bottlenecks. The first is the classical disk IO, and the second bottleneck is the PCIe bus. As for the latter bottleneck, we can differ between avoiding and reducing the impact of the bottleneck.

Disk-IO Bottleneck: GPU-accelerated operators are of little use for disk-based database systems, where most time is spent on disk I/O. Since the GPU improves performance only once the data is in main memory, time savings will be small compared to the total query runtime. Furthermore, disk-resident databases are typically very large, making it harder to find an optimal data placement strategy. However, database systems can benefit from GPUs even in scenarios where not the complete database fits into main memory. As long as the *hot data* fits into main memory, GPUs can accelerate data processing. It is still an open problem to which degree a database has to fit into the CPU's main memory, so GPU acceleration pays off.

Data Placement Strategy: GPU-accelerated databases try to keep relational data cached on the device to avoid data transfer. Since device memory is limited, this is often only possible for a subset of the data. Deciding which part of the data should be offloaded to the GPU – finding a *data placement strategy* – is a difficult problem that currently remains unsolved.

Reducing PCIe Bus Bottleneck: Data transfers can be significantly accelerated by keeping data in pinned host memory. However, the amount of available pinned memory is much more limited compared to the amount of available virtual memory. Therefore, a GPU-accelerated DBMS has to decide which data to keep in pinned memory. Since data is typically cached in GPU memory, a GPU-accelerated DBMS needs a multi-level caching technique, which is yet to be found.

4.7.2 Query Optimization

In GPU-accelerated DBMSs, query processing and optimization have to cope with new challenges. We identify as major open challenges a generic cost model, an increased complexity of query optimization due to the larger optimization space, insufficient support for using multi-processing devices for query-compilation approaches, and accelerating different workload types.

Generic Cost Model: From the query-optimization perspective, a GPU-accelerated DBMS needs a cost model to perform cost-based optimization. In this area, two basic cost-model classes have emerged. The first class consists of analytical cost models and the second class makes use of machine-learning approaches to learn cost models for some training data. While analytical cost models excel in computational efficiency, learning-based strategies need no knowledge about the underlying hardware and can adapt to changing data. It is still open which kind of cost model is optimal for GPU-accelerated DBMSs.

Increased Complexity of Query Optimization: Having the option of running operations on a GPU increases the complexity of query optimization: The plan search space is significantly larger and a cost function that compares run-times across architectures is required. While there has been prior work in this direction [37, 40, 85], GPU-aware query optimization remains an open challenge.

Query Compilation for Multiple Devices: With the upcoming trend of query compilation, the basic problem of processing-device allocation remains the same as in traditional query optimization. However, as of now, the available compilation approaches only compile complete queries for either the CPU or the GPU. It is still an open challenge how to compile queries to code that uses more than one processing device concurrently.

Considering different Workload Types: OLTP as well as OLAP workloads can be significantly accelerated using GPUs. Furthermore, it became common to have a mix of both workload types in a single system. It remains open, which workload types are more suited for which processing-device type and how to efficiently schedule OLTP and OLAP queries on the CPU and the GPU.

4.8 Conclusion

Over the last years, many prototypes of GPU-accelerated DBMSs have emerged implementing new co-processing approaches and proposing new system architectures. We argue that we need to take into account tomorrows hardware in today's design decisions. In Chapter 2, we theoretically explored the design space of GPU-aware database systems. In summary, we argue that a GPU-accelerated DBMS should be an in-memory, column-oriented DBMS using the block-at-a-time (or, more specifically vector-at-a-time [31]) processing model, possibly extended by a just-in-time-compilation component. The system should have a query optimizer that is aware of co-processors and data-locality, and is able to distribute a workload across all available (co-)processors.

In this chapter, we validated these findings by surveying the implementation details of eight existing GPU-accelerated DBMSs and classifying them along the mentioned dimensions. Additionally, we summarized common optimizations implemented in GPU-accelerated DBMSs and inferred a reference architecture for GPU-accelerated DBMSs,

which may act as a starting point in integrating GPU-acceleration in popular main-memory DBMSs. Finally, we identified potential *open challenges* for further development of GPU-accelerated DBMSs.

Our results are not limited to GPUs, but should also be applicable to other co-processors. The existing techniques can be applied to virtually all massively parallel processors having dedicated high-bandwidth memory with limited storage capacity.

The remainder of the thesis investigates different scalability aspects of co-processor accelerated databases. Chapter 5 shows how we can abstract from processor heterogeneity and algorithm implementation details for the operator placement problem, a critical optimization that ensures that processing resources are used efficiently. In Chapter 6, we discuss how we can exploit parallelism between processors to improve performance. Furthermore, we investigate the scalability of database query processing for an increasing number of co-processors. In Chapter 7, we investigate the scalability of co-processor-accelerated DBMSs in terms of increasing database size and number of parallel users.

5. Hardware-Oblivious Operator Placement

In the previous chapters, we learned that modern processors become increasingly heterogeneous. As different processors perform differently for the same operation, it is important to select a suitable processor for each operator. This *operator placement* requires to estimate execution times of all database operator implementations (i.e., database algorithms) and select the cheapest processor. However, this is not easily possible with the state-of-the-art approach to create analytical cost models. We discussed in Chapter 1 that we need to abstract from this processor heterogeneity to keep the system complexity manageable.

In this chapter, we will take an alternative path to analytical cost models: We treat database operators as black boxes and learn their performance behavior via statistical regression. We then use the learned cost functions to predict execution times and to select the cheapest processor for an operator.

> *At this point, we explicitly point out that we are users of concepts of statistical learning; we do not contribute anything new to the field of statistical learning.*

A framework for solving the operator placement problem must be able to generate precise estimations of total processing costs, depending on available hardware, data volumes and distributions, and the system load when the system is actually deployed. This is further complicated by the rather complex algorithms which are required to exploit the processing capabilities of GPUs and for which precise cost estimations are difficult.

In this chapter, we present HyPE, a hardware-oblivious framework for operator placement. HyPE automatically learns and adapts cost models to predict the execution time of arbitrary database algorithms (i.e., implementations of relational operators) on any (co-)processor. This is possible because of three aspects of database systems:

1. Database operators can all be computed in $O(n)$ or $O(nlog(n))$ time complexity, which greatly simplifies regression.[1]

2. The parameters that significantly impact performance of database operations are well understood. Therefore, the challenge of learning the performance behavior of database operators is greatly reduced, as these parameters do not need to be identified first.

3. We do not need to learn a general model that can be applied to arbitrary databases and workloads. Instead, we learn a specialized model only, which supports the current databases and workloads run by the DBMS.

HyPE uses the cost models to predict execution times and place database operators on available (co-)processors. Furthermore, HyPE continuously monitors execution times of algorithms and refines estimations as it obtains more training data. We demonstrate HyPE's applicability for two common use cases in modern database systems. Additionally, we contribute an overview of GPU-co-processing approaches, an in-depth discussion of HyPE's operator model, the required steps for deploying HyPE in practice and the support of complex operators requiring multi-dimensional learning strategies.

5.1 Co-Processing Approaches for Databases Systems

In this section, we contribute a short survey and classification of DBMS operations that can be offloaded to co-processors, especially the GPU. We introduce two important use cases in detail and show why we need automatic scheduling for these operations.

5.1.1 Co-processors in a DBMS

Figure 5.1 puts the research in the field of database co-processing in three different classes, namely query processing, query optimization, and database tasks.

Query Processing

We can find a plenitude of research that focuses on using GPUs and other co-processors to accelerate relational operators. Especially for joins there is a large variety of approaches for executing them on the GPU [86, 109, 156], on FPGAs [188], and even on Network Processing Units [72]. Other work also addresses the co-processing of all relational operators [18, 57, 85]. Index scan acceleration was investigated by Beier and others [23] and Kim and others [117]. Knn-search was studied by Wang and others [195], Garcia and others [67] and Barrientos and others [21]. Spatial range queries

[1] The non-indexed nested loop join is an exception as it has quadratic time complexity ($O(n^2)$). However, every query optimizer tries it's hardest to avoid nested loop joins as they quickly degrade in performance.

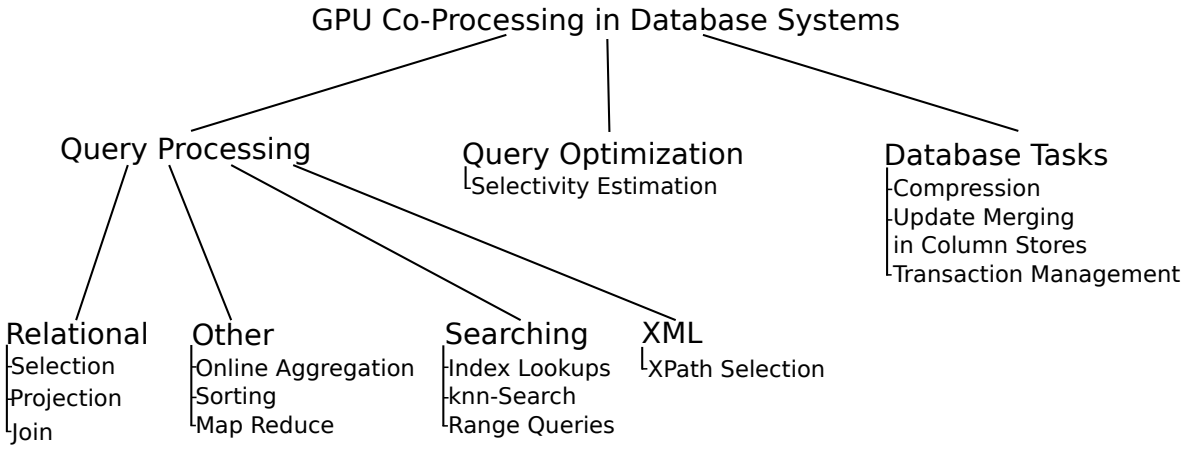


Figure 5.1: Classification of Co-Processing Approaches

were investigated by Pirk and others [158]. There is also work addressing sorting [73], online aggregation [124], and XML path filtering [139]. There are several approaches addressing MapReduce, e.g., He and others developed Mars [84] and stream processing on GPUs, e.g., Karnagel and others [111] and Pinnecke and others [153].

Query Optimization

Augustyn and Zederowski describe how to calculate the query selectivity estimation with a DCT algorithm on the GPU [16]. Heimel and Markl use kernel density estimation to estimate the query selectivity [94] and see this as a first step to optimize queries with the help of a co-processor.

Database Tasks

There are more calculation intensive operations that are executed by the DBMS to maintain the stored data. Krüger and others studied the process of merging the update buffer into the main storage of an In-Memory Column Store with the help of a GPU [122]. Data compression on GPUs were investigated in [14, 64]. He and others focused on transactional processing with graphic cards [87].

They all have in common that the operation or an essential part of it can be offloaded to the co-processor. Because of the offloading itself this involves some overhead, i.e., for small problem sizes the overhead often dominates the actual execution time. Also, the parameters of the operation and the data distribution change the calculation in a way that it does not fit to the co-processor's architecture anymore. Therefore, we cannot say that the operation is always faster on the co-processor than on the CPU counterpart and vice versa. Furthermore, without knowledge of the hardware, an a priori configuration is likely to be infeasible and the user of the DBMS is not able to decide which version is best. Therefore, a hybrid scheduling framework is needed which chooses automatically the fastest algorithms depending on the operation to perform, its parameters, and the properties of data. Furthermore, it may be beneficial to utilize both, CPU and GPU, to

increase throughput. Every use case can be seen as one operator, where we can decide to offload the calculation to a potential co-processor or execute it on the CPU. We choose two use cases and executed them as operator with the help of our framework.

5.1.2 Use Cases for Co-Processing in Database Systems

We now discuss our use cases, namely sorting of data and index scans.

Data Sorting

The first use case we considered is the classical computational problem of sorting elements in an array which has been widely studied in the literature. Especially for database operations such as sort-merge joins or grouping it is an important primitive that impacts query performance. Therefore, many approaches exist to improve runtimes with (co-)processing sort kernels on modern hardware (e.g., Govindaraju and others [73]).

As parallel sort implementations, we used the primitives provided by the CUDA framework for GPUs and the Intel TBB implementation as its CPU counterpart. We included this primitive in our experiments since the CPU and GPU algorithms show multiple break-even points (cf. Section 5.4.2), challenging our learning framework to make the right scheduling decisions. Further, it is a multi-dimensional problem when using the number of elements to be sorted and the number of CPU cores as parameters during scheduling.

Index Scan

The second important primitive for query processing is an efficient search algorithm. Indexes like height-balanced search trees as commonly used structures can be beneficial to speed up lookup operations on large datasets. Several variants exist for various use cases, e.g., B-trees for searching in one-dimensional datasets where an order is defined, or R-trees to index multi-dimensional data such as geometric models. To ease the development of such indexes, frameworks such as GiST [96] encapsulate complex operations such as the insertion and removal of key values to/from tree nodes and height-balancing. To implement a new index type, only the actual key values and key operations, such as query predicates, have to be defined by the developer. E.g., minimal bounding rectangles with n coordinates and an intersection predicate are required to define an n-dimensional R-tree [96]. To speed up GiST lookup operations with co-processors such as GPUs, Beier and others implemented a framework that abstracts from the hardware where index scans are actually executed and therefore, hide the complexity for adapting and tuning the algorithms for each specific platform [23].

To maximize scan performance on massively parallel hardware such as GPUs, a fine-granular parallelization has to be found which is depicted in Figure 5.2. All lookup operations are grouped in batches for each index node. The batches start at the root node and are streamed through the tree until they are filtered or reach the leaf layer,

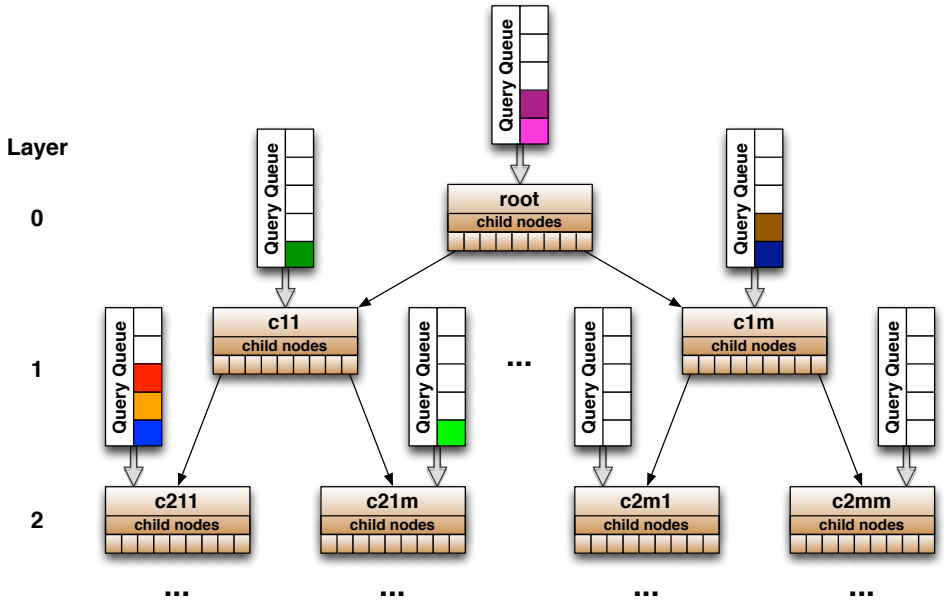


Figure 5.2: Index Tree Scan

returning final results. All required index nodes can be scanned in parallel on a GPU or CPU processor core. For a scan, all query predicates are tested against the key values representing a child node. All of these tests are independent and can be processed in parallel by a GPU core's thread processors.

To achieve optimal scan performance, it is required to determine which node has to be scanned by which (co-)processor. This decision has to be made in each iteration and depends on the number of a tree node's children (*slots*). Large nodes result in many tests per node but less index levels to be scanned while small nodes reduce the required scan time per node but result in deeper trees. Even more important is the number of *queries per scan task* since the node size is determined once when the index is created and, in the current implementation, will not change at runtime. The batch size depends on the application's workload and the layer where a node resides. Large batches are expected at levels near the root since all queries have to pass them first. Smaller ones are expected near the leaf layer because queries are filtered out and distributed over the entire tree. The parameters' impact on scan performance is illustrated in Figure 5.3 where the GPU speedup $s = \frac{CPU\ time}{GPU\ time}$ is plotted for different parameter combinations. For small node and batch sizes, the GPU scan is up to 2.5 times slower ($= \frac{1}{s}$) than its CPU counterpart. For large batches and/or nodes, the transfer overhead to the GPU can be amortized and a scan can be nearly twice as fast on the GPU. The break even points where both algorithms have the same runtime ($s = 1$) are depicted with the dotted line. These points depend on hardware characteristics such as cache sizes etc. To ease the deployment of the hardware-accelerated index framework on various platforms, the scheduler greatly benefits from a learning decision component that automatically adapts these thresholds. Our presented scheduler framework treats

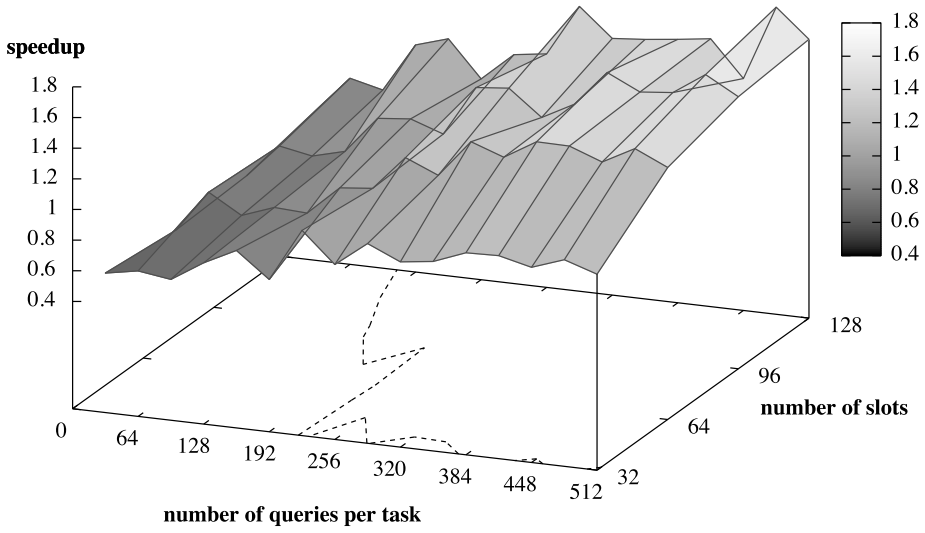


Figure 5.3: Index Scan - GPU Speedup

the underlying hardware as black-box, effectively reducing the number of knobs to achieve near-optimal performance as we show in Section 5.3.

5.2 Operator Model

In this section, we present the operator model of our framework. Furthermore, we discuss assumptions and restrictions of the operator model with respect to system load and data characteristics.

5.2.1 Base Model

As shown in the previous sections, it is required to find parallelizable subtasks that can either be executed by the CPU or a co-processor such as a GPU. These subtasks, which we refer to as operators in the following, are the base granularity for a scheduling decision that is driven by our operator placement framework, which we introduce in Section 5.3.

We model an operator as an algorithm that solves a specific (sub-)task on some kind of input data and produces a certain amount of output data, which can be used as input for the next operator in a query plan. Figure 5.4 illustrates this issue. For accessing the input data and storing results, a certain amount of time is required, e.g., I/O time for fetching data from disk, or transferring data from/to an attached co-processor device. These times can easily be measured in current systems or calculated, e.g., when the bus bandwidth and transfer rates of the underlying hardware are known. For actually solving the task, computation time is consumed by the involved processor. This time depends on the algorithm that is used - e.g., different sorting or join algorithms - as well as characteristics of the data to be processed, for example the selectivity of a

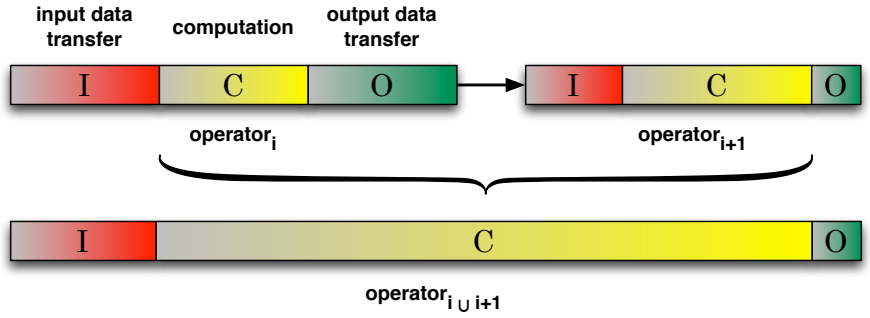


Figure 5.4: Operator Model

complex filter, or the number of groups during a grouping/aggregation operator. Due to this dynamic runtime dependency, the computation time is harder to estimate than transfer times. Further, the computation time includes hidden data access times such as main memory or cache accesses. Those internal times are hard to measure when the algorithm is executed and usually require sophisticated profiling tools that utilize dedicated system-specific hardware counters.

We regard operators as primitives where the computation phase is an atomic part that starts right after the input data is available. This perfectly matches the computing model of modern GPUs where input data needs to be transferred to the device before a parallel kernel starts processing, and its output has to be transferred back to the host if it cannot be reused by another kernel. This model is common in literature (e.g., He and others [85]). Of course, especially on GPUs there are complex access patterns which overlap data accesses with computation and, therefore, multiple transfer/computation phases interleave. With *Unified Virtual Addressing* (UVA) the CUDA runtime itself transfers the data between different devices and the CPU's RAM when it is needed. However, the transfer is hidden from the developer with this technique but it is still the bottleneck in many cases. Further, several algorithms cannot be expressed with a single atomic computation phase. Within a hash join operator, the probe phase needs to wait until all keys have been inserted into a hashtable. In such cases, the operator could be split further into dependent primitives. In general, this could be done and a scheduling decision could be made for each resulting primitive. But in most cases this approach is counterproductive. First, it complicates the decision because many possible execution plans are generated. Second, splitting such operators that are tightly coupled will hardly result in any improvement with separate processing decisions. Most likely, data transfer costs will negate any performance improvements or even lead to slowdowns. With a simple approximation, data locality can be increased: operators are merged to single operator as illustrated in the bottom of Figure 5.4, for which a single scheduling decision is made and all internal data transfers are hidden in the computation phase.

5.2.2 Assumptions & Restrictions

The execution times that need to be estimated for all phases for a scheduling decision are impacted by two important parameters that are dynamically changing during system

runtime. Since they are hard to model, we have to make some assumptions that are explained in the following.

System Load

Concurrently executed tasks occupy shared resources which directly impacts the runtimes that shall be estimated. Taking the load into account is currently infeasible for several reasons:

1. **It is hard to model.** Depending on the type of concurrently running tasks, different resources might be affected. Data-intensive tasks will impact the available memory bandwidth. Further, caches might be affected. Depending on the implementation, transfer operations can therefore be blocked or delayed, rendering transfer times nearly unpredictable. Similar for compute-intensive tasks which, e.g., impact processor as well as cache utilizations. Depending on the operating system's or hardware scheduler's behavior, expected computation times will become inaccurate.

2. **It is hard to measure.** For some parameters such as disk I/O or main memory usage, it is simple to determine the current utilization as long as a central management instance such as the bufferpool is available. But for most parameters as caches or the processor utilization, such an instance does not exist. Therefore, it is hard to obtain such measures without approaches implemented in profilers that sample instruction caches and query dedicated hardware counters. Moreover, the system load can quickly change, depending on the arrival rate of concurrent tasks.

3. **It is hard to influence.** As mentioned above, a central workload management instance does not exist on most systems. Therefore, it is impossible to control available resources such as memory bandwidth and assign them to dedicated tasks. However, for processing power some frameworks such as Intel TBB allows the assignment of CPU resources. By the time this thesis was written, GPUs (Nvidia *Fermi* architecture [144]) were designed such that as many cores as possible are assigned to one task. So the GPU is either busy or not. Concurrent kernels could not be executed in parallel if they were not managed by the same host context. In the meantime, this changed with the *Hyper-Q* feature of Nvidia's newest *Kepler* architecture [145]. But to the best of our knowledge, it is not possible to explicitly control how many cores are assigned to one kernel.

For these reasons, we currently make some assumptions about the load parameter and do not take it into account when deriving operator execution models. Since for a scheduling decision only the delta between a CPU and GPU execution is relevant, we assume that the system load has approximately the same impact on both execution units. For data transfers this assumption is valid since both share the same system bus to access the memory. For processing resources, we assume that there exists some kind of management layer as TBB or a GPU device manager [194] which knows if processors

are available or not and can assign them accordingly. This assumption is valid because for using our framework, there has to be a scheduler that uses the decision and which has to be able to assign the task to a dedicated execution unit.

Data Characteristics

Computation times are directly, and output transfer times indirectly through result set sizes, impacted by runtime-specific data-related parameters such as cardinality, selectivity, and skew. Many approaches exist to estimate these parameters such as Getoor and others [70], or Chen and others [52]. Therefore, we assume that these parameters can be obtained before the scheduling decision is made and can be used as input parameters for the operator execution model which is learned by our framework.

5.3 Decision Model

As shown in Section 5.2, reasonably guiding a scheduling decision between different processor types is a complex task with many impacts. Our contribution to tackle this problem statement is a framework which uses machine learning techniques in order to find suitable execution models for arbitrary algorithms that can be mapped to the operator model introduced in 5.2.1. The framework treats the underlying system hardware as black box and adapts a cost model according to the conditions during runtime.

5.3.1 Problem Definition

To make the scheduling decision, the total execution time T^p_{total} of a an algorithm instance a which solves a problem of class A (e.g., hash join and merge join both solve the join problem) needs to be estimated for each available (co-)processor p. It comprises the in/output transfer time (T^p_i/T^p_o) and the actual computation time (T^p_c). Since all of these times depend on several parameters, we assume that there exist (unknown) functions t^p_i, t^p_c, and t^p_o which describe the system behavior for that specific (co-)processor:

$$T^p_{total} = T^p_i + T^p_c + T^p_o \text{ where} \tag{5.1}$$

$$T^p_i = t^p_i(\underline{inputsize}, \underline{HW}, \underline{load}) \tag{5.2}$$

$$T^p_c = t^p_c(\underline{inputsize}, \underline{a}, \underline{HW}, \underline{P}, \underline{load}) \tag{5.3}$$

$$T^p_o = t^p_o(\underline{outputsize}, \underline{A}, \underline{HW}, \underline{P}, \underline{load}) \tag{5.4}$$

The influences of the data size parameters, as well as the system hardware HW are considered to be static when we ignore the dynamic system *load* as argued in Section 5.2.2 and assume that the hardware is not changed during runtime. The abstract (multi-dimensional) parameter P has to be assessable by external methods (cf. Section 5.2.2)

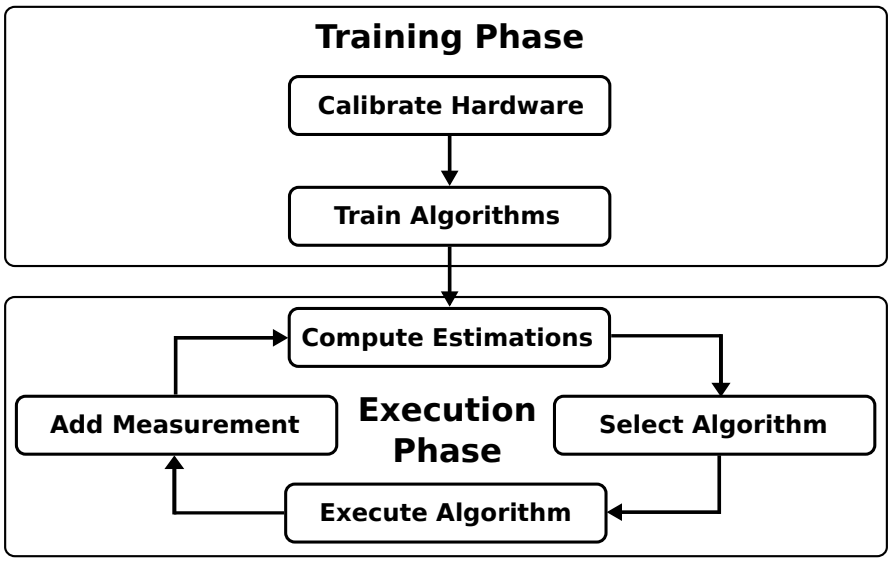


Figure 5.5: Modeling Workflow

and can comprise any number of other dynamic parameters such as selectivity or cardinality. The *outputsize* depends on these characteristics as well as on the algorithm class A. The actual algorithm instance a does not have an impact since we consider the output of all $a \in A$ to be the same (all algorithms produce the same solution). Only the computation time depends on it. Therefore, the *outputsize* can be estimated in advance.

The actual task now is to model these black-box functions somehow. This can be achieved with various techniques, e.g., using analytical models which are rather static, or learning-based approaches that require an expensive learning phase (cf. Section 5.5). To be flexible and reduce the model overhead, we decided to use statistical methods that create an initial model with a short training phase and, during runtime, improves it with observing execution times for a specific input parameter set when algorithms are processed by a certain processor.

5.3.2 Modeling Workflow

The workflow for creating the operators' execution models is illustrated in Figure 5.5. It consists of two phases. During the training phase, the initial model estimation is created. First, the static hardware-specific behavior for data transfers needs to be approximated. Therefore, in a short calibration operation, copy operations are scheduled without applying computations on the data. As shown in [85, 130] the memory access behavior for the host system can be determined. For GPU data transfers, we copied data blocks with varying sizes from the host's RAM to an Nvidia Tesla C1060 GPU's VRAM. Since concurrent asynchronous copy operations are supported by modern GPUs, we executed 1000 copy transactions for fully utilizing the available bandwidth. Additionally, synchronous copies, which are serialized by the GPU driver, were scheduled to obtain

transfer times that are required for small, single blocks. The required transfer times as well as the bandwidth utilization is shown in Figure 5.6.

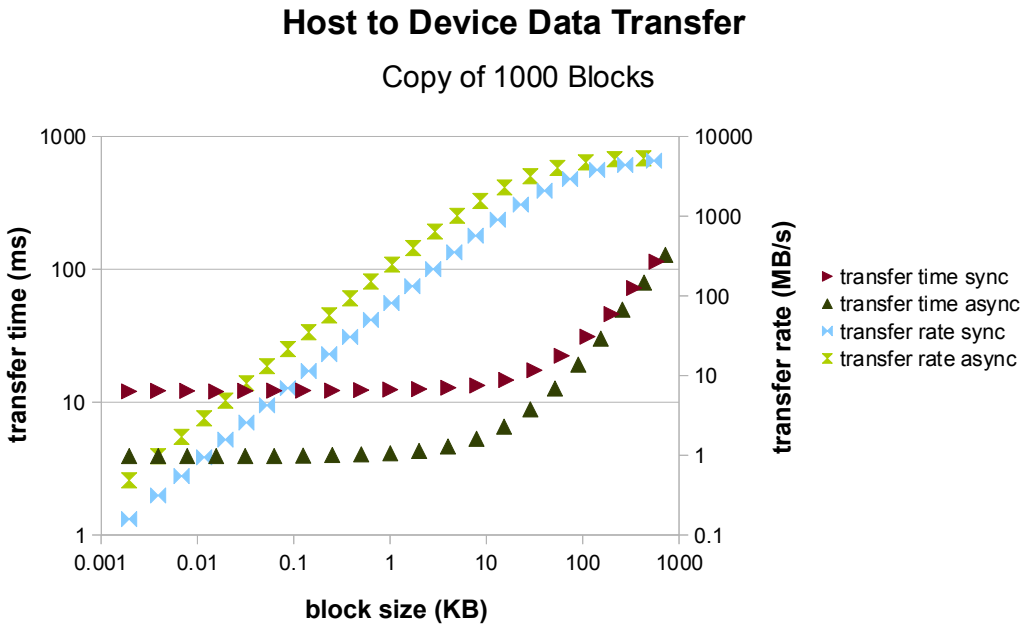


Figure 5.6: GPU Data Transfers

Although the theoretical bandwidth in our system (PCIe 2.0x16 bus) is approximately 8 GB/s, the real achieved bandwidth depends on the size of the copied data. Figure 5.6 illustrates this issue. The maximal rate we achieved was ≈5.7 GB/s. A block size of 0.7 MB is sufficient for fully utilizing the PCIe bus (synchronous copy). This size can be decreased by an order of magnitude by using asynchronous copy instructions because they are executed in parallel and hide the overhead (≈1μs at sync copy of block size 1 byte) for calling the GPU driver's copy interface.

This calibration needs to be done once when the system is set up or the hardware configuration changed. Afterwards, the initial computation time models for the available algorithms need to be created. This can either be done once during system setup, or during runtime when operators need to be scheduled. For some iterations, all operator versions are executed on all available processors - or in a round robin fashion choosing one of them. No reasonable decision can be made at this point in time. The runtimes are measured and the computation time is calculated using the transfer time models obtained during the calibration. Together with the used parameters, these measures form interpolation points that are used to approximate functions describing the computation behavior. We use the ALGLIB [28] package to calculate the approximations (cf. Section 5.4.1).

After the initial training cycles finished, the execution phase starts. The models learned so far are used to guide scheduling decisions (Figure 5.7). The processor with minimal expected costs ($T_{est}(a_i, D)$) is chosen and real execution times ($T_{real}(a_i, D)$) are

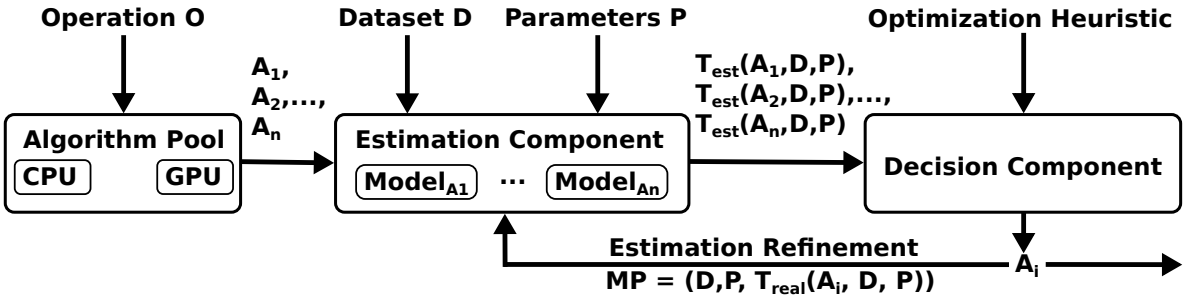


Figure 5.7: Decision Model Overview

measured. As in COMET [205], the model is updated in case $T_{est}(a_i, D)$ for the given data set parameters D (comprising the size and any available characteristics) differs too much from the actual measure $T_{real}(a_i, D)$. The (D, T_{real})-measurement pair (MP) is used to update the model accordingly.

Adding *Measurement Pairs* to the model increases the amount of memory which is required for storing it, as well as the computation time for calculating the approximation functions. Therefore, we batch recomputations and further limit the maximum amount of interpolation points using ring buffers that automatically overwrite old measurements when new pairs are added and the buffers are full. Using ring buffers causes very low overheads but may lead to worse approximations compared to other aging mechanisms which evaluate the significance for each MP and evict the least important one.

5.3.3 Model Deployment

The learning process of the decision model is generic, i.e., no prior knowledge about hardware parameters (e.g., clock rate, memory bandwidth) or details about used algorithms (e.g., radix join or sort-merge join) is required at development time. However, the model needs to know for which measure it has to optimize (e.g., response time) as well as the relevant parameters to estimate the metric. Furthermore, depending on the relationship between the parameters and the measure, different learning methods need to be used.

So far, we used response time as the measure we want to optimize. However, other measures could be used, for example the expected throughput of an operation or the energy consumption. It just requires that the problem definition in Section 5.3.1 has to be adapted. The workflow for calibrating and adjusting the model during runtime does not need to be changed. In general, the following steps are required to deploy the framework:

1. **Identify operators**: the task has to be divided into parallelizable subtasks that can be considered for co-processing. Usually some kind of profiling is required to find bottlenecks where utilizing a specialized co-processor is promising. Then, the task has to be mapped to the operator model introduced in Section 5.2.

2. **Operator implementation**: the algorithm(s) solving the task have to be implemented and tuned for each kind of processing unit that shall be supported. Frameworks such as OpenCL, CUDA or TBB help to abstract from the actual hardware layer that will later be used.

3. **Identify scheduling measure**: depending on the goal why the co-processor shall be used, a measure such as response time has to be defined which can be used for optimizing the scheduling decision.

4. **Identify parameters**: the most important parameters that impact the optimization measure have to be found. Note, that there needs to be a way for estimating them before the scheduling decision is made, because they will be used as input for the model.

In database systems, database operators and the parameters that impact the performance of each database operator are well understood. Therefore, step 1 and 4 can be omitted except for user-defined functions.

5.3.4 Implementation Details: Customizing the Framework

We implemented the decision model in HyPE, which is an open-source, hardware-oblivious framework for operator placement in hybrid CPU/co-processor environments.[2]

HyPE contains an extensible plug-in architecture allowing the user to implement custom optimizations. All plug-ins can be implemented by (1) creating a simple class that inherits from an abstract base class and (2) registering the derived class in the plug-in manager. Currently, HyPE supports three plug-in types: *statistical methods*, *recomputation heuristics*, and *optimization criteria*, which can be exchanged dynamically.

Statistical Methods

The user can deploy a machine learning algorithm currently not implemented in HyPE by creating a derived class of *StatisticalMethod* and implement two functions: The first is *recomputeApproximationFunction*, which applies the statistical method to the current set of observations of an algorithm and returns a new approximation function. The second function is *computeEstimation*, which gets the feature vector of the data set as parameter and returns an estimated execution time. In HyPE, a feature vector consists of the data set parameters D and operator parameters P, which consists of the size and other relevant characteristics (cf. Section 5.3.1).

Recomputation Heuristics

Recomputation heuristics steer the process of runtime refinement for the cost estimations as discussed in Section 5.2.2. Currently, the user can choose between no runtime

[2] http://cogadb.cs.tu-dortmund.de/wordpress/hype-library

adaption (e.g., after the initial training phase, there will be no additional overhead for training, but the initial training has to become longer) and periodic adaption, which recomputes the approximation functions of each algorithm periodically. The user can add a tailor-made recomputation heuristic by creating a class that inherits from the base class *RecomputationHeuristic* and implement the pure virtual method *internal_recompute*, which gets an algorithm as input and returns true, in case the approximation function of the algorithm should be recomputed and false otherwise. Note that an algorithm stores all of it's statistical information (e.g., the set of all MPs, number of executions, and the average estimation error).

Optimization Criteria

An optimization criterion sets the optimization goal of the scheduler (e.g., response time or throughput) by using a scheduling strategy. HyPE will schedule the operators according to the scheduling strategy. The user can add a scheduling strategy by creating a class that inherits from the abstract base class *OptimizationCriterion* and implement its pure virtual method *getOptimalAlgorithm_internal*, which gets the feature vector of the input data set and an operation as parameter and returns a scheduling decision. The interested reader can find detailed information in the official documentation.[3]

5.4 Evaluation

To evaluate the applicability of our approach, we have to clarify:

1. How well do the automatically learned models represent the real execution on the (co-)processors, i.e., can HyPE produce reasonable scheduling decisions?
2. Do the applications benefit from the framework, i.e., does the hybrid processing model outweigh its learning overhead and improves the algorithms' performance regarding the metric used in the decision component?

5.4.1 Test Setup

We trained our framework for execution models of different CPU/GPU algorithms for the use cases described in Section 5.1.2. For choosing appropriate statistical methods in the estimation component, we performed several experiments with the ALGLIB [28] package and found the *least squares method* and *spline interpolation* to obtain good estimation errors at reasonable runtime overheads (cf. Section 5.4.2) for one-dimensional parameters and multi-parameter fitting for multi-dimensional ones.

The runtimes of the sort workload are impacted by the size of the array to be sorted, as well as the number of CPU cores that are used for it. The number of used GPU cores cannot be controlled. However, for the index scan, only one parameter was chosen,

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/hype/current/doc

namely the number of queries in a batch for a node to be scanned (which corresponds to a slice in the multi-dimensional parameter space illustrated in Figure 5.3. The number of slots per inner node is assumed to be constant because in the current implementation it is determined once when the index is created and cannot be modified afterwards. Further, the index assigns each node to a single core only. Hence, the number of CPU cores is, in contrast to the sort workload, not a parameter for the scan.

As sort workload, an array of up to 32M 32-bit integers were randomly generated and processed by a varying number of CPU threads, or by a single GPU. For the index framework, we used a 3-D R-tree implementation as discussed in Section 5.1.2 and by Beier and others [23] to execute node scans on CPU cores or offload them to the GPU. As input data for the R-tree, we artificially generated nodes with 96 disjoint child keys. Due to that we were able to generate queries with certain *selectivities*. To fully utilize all GPU cores, 128 scan tasks were scheduled at the same time. Further details can be found in [23]. If not otherwise stated, data transfer times were included in all of the measurements since they can dominate the overall performance [77].

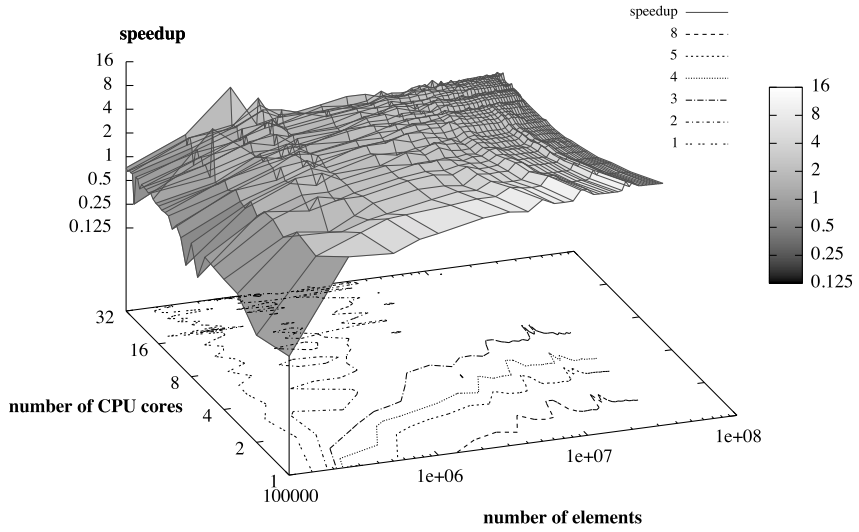
The sort experiments were conducted on a 2.30 GHz 24-core Intel Xeon CPU E5-2630 linux server and an Nvidia Tesla C2050. For our the index experiments, we used a linux server, having a 2.27 GHz Intel Xeon CPU and an Nvidia Tesla C1060 device attached via PCIe 2.0x16 bus. The CUDA driver version 4.0 was used.

5.4.2 Model Validation

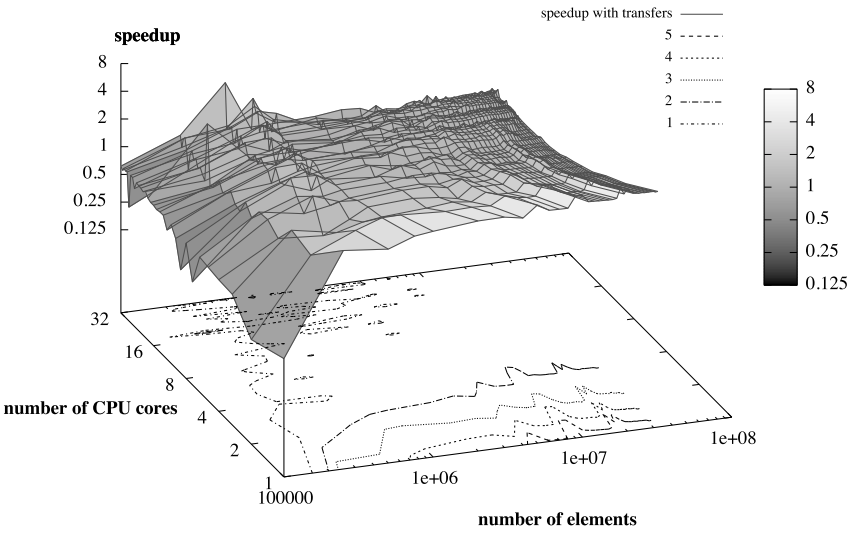
In order to evaluate the scheduling decisions performed by HyPE, we executed a training phase with input data generated from the entire parameter space to obtain global execution models for these problem classes.

The speedups ($CPU\ time/GPU\ time$) for the sort experiments are illustrated in Figures 5.8(b) and 5.8(a). It clearly shows that hybrid processing can be beneficial as there are regions with $speedup < 1$ where the CPU is faster and areas where the GPU performs better ($speedup > 1$), $speedup = 1$ denote break even points. To visualize these regions, we plotted contour lines in the xy-plane that show when a certain speedup value is exceeded. Note, that including data transfers (Figure 5.8(b)) does not change the shape of the surface but only shifts it in z-direction, i.e., transfers are not impacted by the runtime-specific parameters as discussed in Section 5.3.1.

This surface has to be learned incrementally by the framework. In practice, it is not feasible to learn algorithm behavior statically for all parameter combinations since they grow exponentially with an increasing number of parameter dimensions. We trained our model with 50 randomly chosen parameter combinations from the generated workload and plotted the relative estimation errors for the whole parameter space (Figure 5.9(a)). The relative estimation error denotes the average absolute difference between each estimation value and its corresponding measure in the real workload as in [10]. After this initial training phase, further measures were added to the model. The errors after 250 additional samples are plotted in Figure 5.9(b).



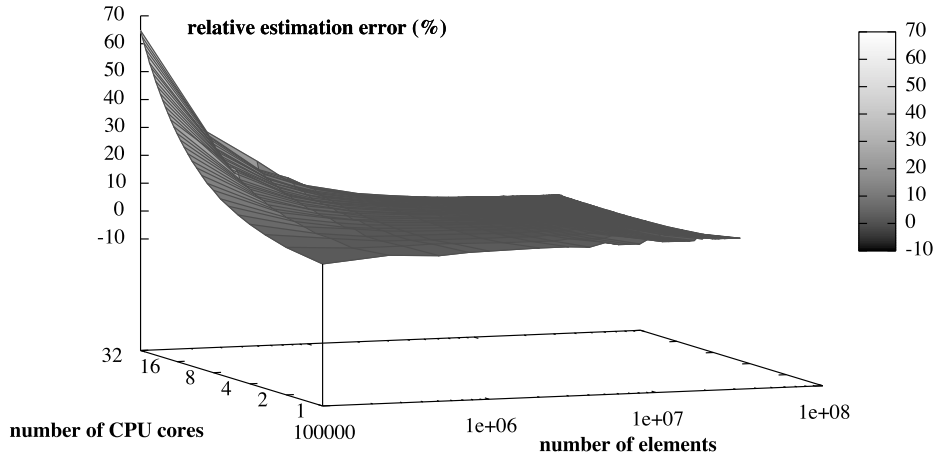
(a) Without Data Transfers



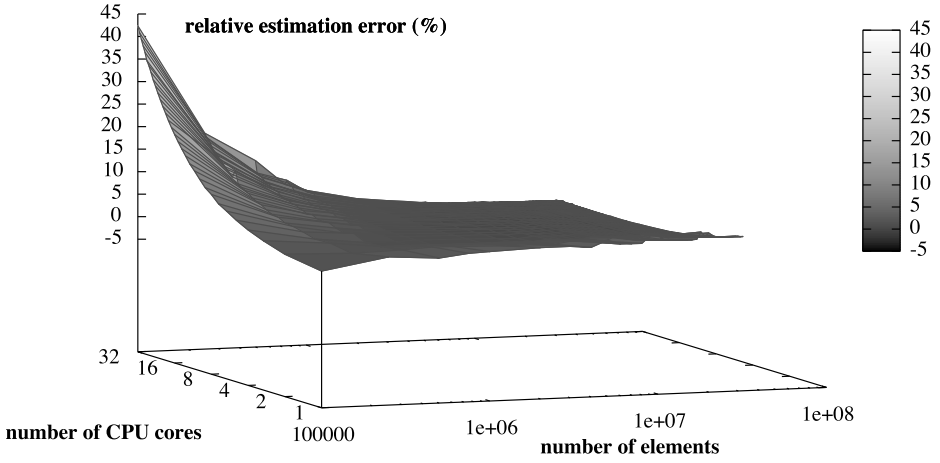
(b) With Data Transfers

Figure 5.8: Sorting Workload

The largest errors occur in the area of many CPU cores and small data sizes. This is caused by the execution time jitter of the parallel sorting routine which is high for small data sets when too many cores are utilized. The results show that for most parameter combinations the estimation errors are ≈ 0 which confirms that HyPE can effectively handle multi-dimensional learning problems. Further, it shows that the model can be



(a) Training Length: 50 operations

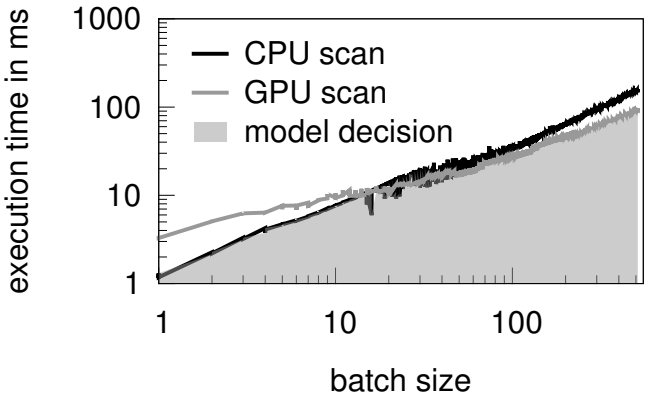


(b) Training Length: 300 operations

Figure 5.9: Relative Sorting Estimation Errors

trained incrementally since the estimation errors decreased after additional samples have been added.

The results for the index scan are illustrated in Figure 5.10. We illustrated the runtimes of each algorithm as lines and shaded the area of the respective model decision after the training. For smaller input sizes, the overhead for invoking GPU kernels and data transfers dominate the execution and the CPU algorithm is faster. On larger inputs, the GPU can fully utilize its advantage through parallel processing when computation becomes dominating. Only one break even point exists in this slice of the multi-dimensional parameter space which could be calibrated statically. Utilizing our self-adapting decision model for this is beneficial since it does not make any assumptions regarding underlying hardware and, therefore, offers the flexibility to use the index framework on various platforms - for which is actually designed for.



algorithm	relative error
CPU scan	8.99 %
GPU scan	3.99 %

Figure 5.10: Index Workload

5.4.3 Model Improvement

The sole number of wrong decisions is not sufficient to evaluate the approximations as a whole. Although wrong scheduling may lead to severe performance degradations, it may have negligible consequences too. Latter happens when a wrong decision is made for parameters near the break-even-point where measures for multiple algorithms are nearly the same. Therefore, we have to define a measure that evaluates how much the application would benefit from hybrid scheduling for a specific workload. To quantify the performance gain, we define the **model improvement** as:

$$model\ improvement(DM_i \rightarrow DM_j, W) = \frac{T_{DM_i}(W) - T_{DM_j}(W)}{T_{DM_i}(W)} \tag{5.5}$$

This ratio indicates how the measures used as optimization goal - T for runtime in our case - will change when instead of a decision model DM_i another DM_j would be used on a specific workload W. A workload comprises a set of tasks that shall be scheduled depending on the learned task parameter(s), e.g., input size or selectivity. In the following, DM_{real} indicates the model learned during the training phase. DM_{ideal} is the hypothetical model that always choses the fastest algorithm, for the best processing device. DM_{ideal} indicates the upper bound for the hybrid scheduling approach and can never be achieved when the model learning and adaption overhead is considered. But it indicates the capabilities of improvements that can be achieved for the respective problem class.

A hybrid approach is beneficial when the model improvement measure compared to the trivial models that always chooses the same algorithm for the same processing device is positive. Otherwise, the overhead for learning and adapting parameters cancels out any performance gain. Since actual data distributions may deviate from the parameters provided as trainings samples, a suitable workload has to be defined for each use case. For lack of an actual sorting workload, we focus on the index use case in the following.

A workload for the index use case is multi-dimensional. Several parameters impact the performance of the CPU and GPU scan algorithms. We already mentioned the

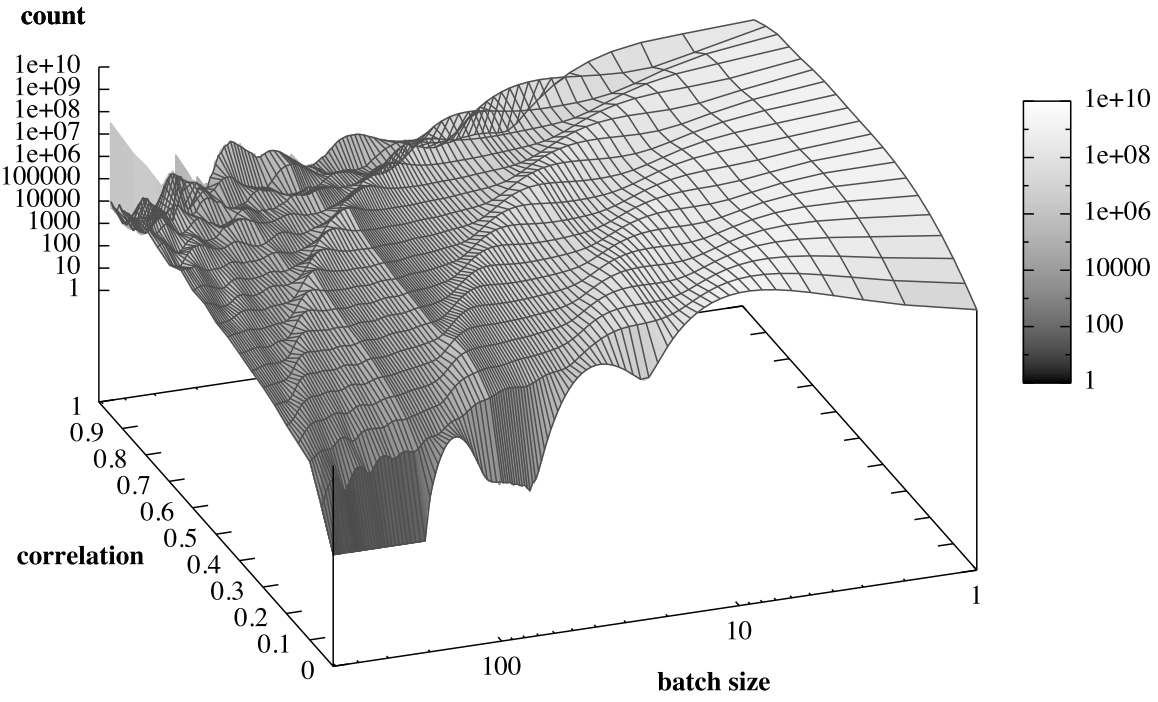
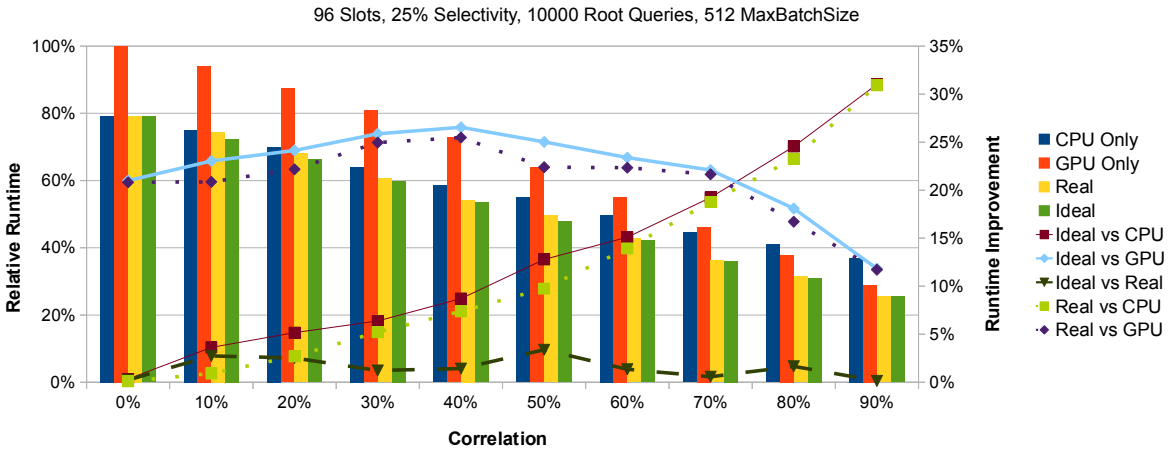


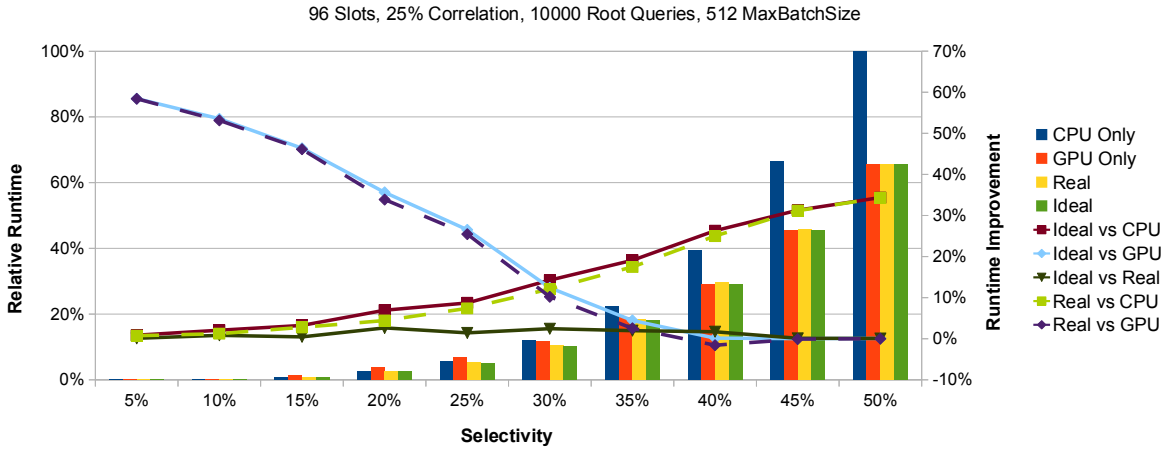
Figure 5.11: Index Batch Sizes

number of slots as well as the *query batch size* in Section 5.1.2. The number of slots currently does not change after the index was created. Therefore, we focus on the batch size as it is workload dependent. After an initial query batch has been scheduled to the root node, parameters influencing the size of subsequent child node batches are *selectivity* and *correlation* of the query predicates. Selectivity denotes how many child nodes are selected by a predicate. To specify it, we generated 3-D R-tree nodes with equal structure and non-overlapping keys. Correlation influences which of the slots are selected by a predicate compared to others in a batch. We modeled it as probability that the next slot to be selected is the one with the lowest unused id. In the other case, any of the remaining slots is chosen with equal probability. Since all nodes have the same structure, the actual selected slot is irrelevant.

To analyze the correlation impact, we generated a full R-tree with 5 levels and 96 slots, leading to a total number of 8 billion indexed entries which is realistic, e.g., for modern CAD applications. 10,000 queries with 25% selectivity were generated for the root node to fully utilize all GPU cores for all tree level iterations. Due to hardware restrictions (shared memory cache size) the maximum size of a batch is 512 for our environment. Larger batches were split into max-size ones and a smaller one. We counted the number of batches for each size with varying correlation probabilities (Figure 5.11). The "waves" correspond to tree layers in the tree. Their heights differ in at least one order of magnitude since queries spread in the entire tree. Most batches are small and occur at the leaf layer. The high number of max-sized batches results from the previously



(a) Correlation



(b) Selectivity

Figure 5.12: Index Improvement

described cutoff. An increasing correlation flattens the surface because queries select the same region and therefore form larger batches for subsequent layers.

Based on this workload, we measured improvements achievable with the hybrid processing model (Figure 5.12(a)). The normalized total runtimes for each decision model are illustrated as bars and are decreasing with higher correlations where the total number of batches decreases significantly, because queries select the same region. Model improvements are depicted as lines. Although trivial models achieved high qualities during the training phase, the hybrid approach shows significant improvements on this workload. Selecting the CPU for the large number of small batches and the GPU for large ones improves the overall performance up to 30%. Note that our learned model is closed to the ideal one. Their runtimes differ in $< 5\%$, including the overhead. The benefit of utilizing the GPU as co-processor increases with higher correlation since batches become larger. Correlations are typical, e.g., for computer graphics applications.

We repeated the experiment with varying selectivities. Figure 5.12(b) shows that increasing selectivities lead to higher advantages for using the GPU since queries selecting more child slots lead to larger batch sizes. For very high selectivities choosing the GPU only approach would not cause notable performance degradations because each batch produces child batches near or above the break-even point. When this point at ≈40% selectivity is reached, the learned decision model sometimes incorrectly suggests the CPU due to approximation errors, leading to a slight negative improvement. But it is <3% which is acceptable for gaining the flexibility of being able to adapting to changing parameters. Using our framework for this use case is therefore recommendable.

5.5 Related Work

In this section, we discuss related work on query optimization, analytical cost models, learning-based prediction of execution times, decision models and hybrid scheduling frameworks.

5.5.1 Query Optimization

Optimization in parallel database systems has similar tasks as optimization of GPU co-processing: optimizing the response time and scheduling operations to resources [51]. Most approaches follow the two-phase optimization approach [99]. First, the database optimizer creates a best sequential query plan. Second, an additional optimizer allocates the operators to the parallel resources to minimize the response time [83]. Thereby, communication costs [82] and different kinds of shared resources [68] have to be taken into account. Lanzelotte and others noticed the enlarged search space and the problem of not optimal sub-plans during dynamic programming style enumeration [123]. The authors showed that randomized search approaches during optimization have a good performance for parallel database systems. Our approach is also based on the two-phase model. We schedule a serial plan between GPU and CPU.

The parallelization of queries using threads of multi-core systems is also related [120]. Krikellas and others used several greedy and dynamic programming approaches to schedule an operator tree on different threads to minimize the response time. Their approach is based on a symmetric environment and does not have to consider communication costs.

Karnagel and others evaluated the impact of compile-time and run-time optimization with the Ocelot Engine [110]. 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.

5.5.2 Analytical Cost Models

Manegold and others propose a framework that is able to create cost functions of database operations by combining their memory access patterns. The cost functions

can be used to estimate the execution time of operators [130]. He and others contribute an analytical cost model for their relational GPU operators [85]. Kothapalli and others utilize CUDA kernel pseudocode analysis to predict the performance of GPU algorithms. Schaa and others developed an analytical cost model that executes estimated execution times for GPU algorithms [175]. Their approach works for varying input sizes and numbers of GPUs.

Wu and others developed analytical cost models to estimate query execution times in dynamic (a priori unknown) concurrent workloads [200]. Predictions are computed by estimating I/O and CPU costs using the optimizer's cost model. These costs are combined with a model for the buffer pool and a model for the query queuing mechanism to merge I/O and CPU costs of concurrent queries to obtain an estimation [200].

Wu and others show that a well calibrated optimizer cost model combined with a refinement of cardinality estimates is always competitive, and often better, than machine learning approaches [201]. They present an approach that can predict the distribution of the prediction error, which provides a measure for how accurate an estimation is.

5.5.3 Learning-based Execution Time Estimation

Learning-based estimation of execution times typically requires that some training workload is available, so that a model can be learned by a statistical regression method to later predict a certain metric (e.g., query response time or disk I/O). In this section, we focus on database specific approaches that use machine learning for execution time prediction. We provide a general overview of statistical learning in Appendix A.1.

Gupta and others develop *Predicting Query Run-Time* (PQR) Trees [79], which estimate the time range when a data warehouse query will finish. PQR trees are essentially decision trees [105]. Predictions are obtained by extracting a feature vector from the query plan and computing a load vector and traversing the PQR tree with these information, until a leaf node is reached, which contains the time range.

Matsunaga and others assess how well different machine learning approaches predict resource utilization in applications. Furthermore, they present PQR2, an improved version of PQR.

Ganapathi and others [65] predict different metrics, such as execution time and disk I/O, using a machine learning approach called *Kernel Canonical Correlation Analysis* (KCCA), which is a generalization of principal component analysis [105]. KCCA allows to estimate query run-time and other performance relevant parameters such as disk I/O and network traffic.

Iverson and others developed an approach that estimates execution times of tasks in the context of distributed systems [103]. The approach estimates execution times using k-nearest-neighbor regression, a non parametric regression technique [105]. Then, tasks are placed on the most suitable machine. To this end, the solved problem is very similar to the operator placement problem. Furthermore, Iverson and others use

analytic benchmarking and code profiling to detect performance differences between machines, in order to store only a unified set of observations. By contrast, HyPE stores observations for each database algorithm for each processor separately.

Choi and others build simple cost models that estimate the run-time of applications on the CPU and the GPU [53]. They collect past execution times and use the average execution time of each processor for an application as an estimate to schedule the application on the fastest processor.

Akdere and others develop an approach for the estimation of execution times on query as well as operation level [10]. The basic idea of their approach is to perform a feature extraction on queries and compute execution time estimations based on them.

Zhang and others present COMET, which is a method for predicting the costs of complex XML queries [205]. To compute cost estimations, COMET uses the transform regression technique [151], which combines linear regression trees [142] and linear regression/generalized additive models [105]. COMET is similar to HyPE, as it also uses learning-based approaches to abstract from complexity of analytical cost models. The continuous collection of statistics allows both frameworks to adapt to changing workloads. The difference is in the application domain: COMET predicts costs of XML queries whereas HyPE estimates execution times of operators on heterogeneous processors. Therefore, COMET has to use more sophisticated regression techniques so it can cope with arbitrarily complex XML queries. By contrast, HyPE can exploit common knowledge of database operators to restrict the learning problem and hence, can use simpler and faster approaches such as linear or k-nearest-neighbor regression.

5.5.4 Decision Models

Kerr and others present a model that allows to choose between a CPU and a GPU implementation [116]. This choice is made statically in contrast to our work and introduces no runtime overhead but cannot adapt to new load conditions.

Zhang and others introduced an alternative optimization heuristic in their system OmniDB, which schedules work units on available (co-)processors. For each work unit, the scheduler chooses the processing device with the highest throughput. To avoid overloading a single processing device, the scheduler ensures that the workload on each processing device may not exceed a predefined fraction of the complete workload in the system [206].

Karnagel and others present *Heterogeneity-aware physical Operator Placement* (HOP) [112], which is similar to our decision model HyPE from this chapter. HOP combines analytical modeling for data transfers and other overheads while using a learning-based estimation of operator execution times. In case no information are available, HOP uses rule-based heuristics for operator placement. Both decision models can be directly applied to the query optimization process by traversing a query plan and place each operator to a suitable processor.

However, there are also some differences. First, HyPE abstracts away more hardware details and thus, is hardware oblivious, whereas HOP requires more information about the heterogeneous processors. This reflects the fundamental trade-off in heterogeneous systems: Either we try to abstract from the details of individual processors (potentially loosing accuracy), or we explicitly model the heterogeneity, which allows us to be more accurate but also increases the complexity of the DBMS with the number of processors it supports.

5.5.5 Hybrid Scheduling Frameworks

Ilić and others showed that large benefits for database performance can be gained if the CPU and the GPU collaborate [101]. They developed a generic scheduling framework [102], which is similar to HyPE, but does not consider specifics of database query processing. They applied their scheduling framework to databases and tested it with two queries of the TPC-H benchmark. However, they do not explicitly discuss hybrid query processing.

Augonnet and others develop StarPU, which can distribute parallel tasks on heterogeneous processors [15]. Both frameworks are extensible and have to be investigated to which degree they can be customized, so they can be used in a database optimizer. The biggest difference to our decision model is that it is tailor made for use in a database optimizer, so it does not enforce to use tasks abstractions.

5.6 Conclusions

We have presented an adaptive framework for hardware-oblivious operator placement to support cost-based operator placement decisions for heterogeneous processor environments, where detailed information on involved processing units is not available. In the considered use cases, we investigated the performance of operations either on CPUs or on GPUs. Our approach refines cost functions by using linear regression after comparing actual measurements with estimates based on previous ones. The resulting functions were used as input for cost models to improve the scheduling of standard database operations such as sorting and scans. The evaluation results show that our approach achieves near optimal decisions and quickly adapts to workloads. While our work is tailor-made for GPU support, the addressed problems and requirements of learning cost models are also relevant in a number of other scenarios.

6. Load-Aware Inter-Co-Processor Parallelism

In the previous chapter, we discussed how we can perform operator placement in a hardware-oblivious way for single operators. In most database workloads, multiple queries run concurrently, and therefore, the DBMS has to place multiple operators on different heterogeneous processors.

Only few solutions address the challenge of utilizing multiple processing devices efficiently (i.e., using the processing device that promises the highest gain w.r.t. an optimization criterion while keeping all processing devices busy to increase performance). There are two major classes of solutions in this field: (1) heterogeneous task-scheduling approaches and (2) tailor-made co-processing approaches. With (1), we do not know the specifics of database systems (e.g., the set of operations and data representations, access structures, optimizer specifics, concurrency control). Additionally, task scheduling approaches typically require a system to use task abstractions of a certain framework (e.g., Augonnet and others [15] or Ilić and others [102]). Since DBMSs have their own task abstractions, a large part of code would have to be rewritten. With (2), we are bound to operations in a specific DBMS (e.g., He and others [85] or Malik and others [128]).

Problem Statement

The problem is that in real life systems, a machine may contain several heterogeneous co-processors besides a few CPUs. Each processing device has its own load and in case they are not homogeneous, different processing speed for each database algorithm. However, there is no state-of-the-art approach that is capable to distribute a workload of database operators on such a system while taking into account processor load and database algorithm speed on each processing device.

Research Question

In a more general context, we have to answer the following research question: How can we distribute a workload of database operators on processing devices with different *operator speeds* and *processor load* efficiently?

Contributions

In this chapter, we make the following contributions:

1. We introduce heuristics that allow us to handle operator streams and efficiently utilize inter-device parallelism by adding new optimization heuristics for response time and throughput.
2. We provide an extension to HyPE, which implements the heuristics.
3. We present an exhaustive evaluation of our optimization heuristics w.r.t. varying parameters of the workload using micro benchmarks (e.g., to identify the most suitable heuristic).
4. Finally, we investigate how the best optimization heuristic scales with an increasing number of co-processors and an increasing speed difference between processing devices, thus proving the overall applicability of our load-aware scheduling in databases.

Major Findings

We find that our approaches can reliably balance a workload not a priori known on all available (co-)processors. The (co-)processors may have significantly different processing speeds for certain operations, which are automatically learned by our system. In a further series of experiments in a simulator, we find that the dominating performance bottleneck in a multi co-processor system is to transfer result data back to the CPU.

Outline

The chapter is structured as follows: In Section 6.1, we present our preliminary considerations. We discuss operator-stream-based query processing as well as HyPE's extensions in Section 6.2. We introduce our optimization heuristics in Section 6.3 and provide an exhaustive evaluation using micro benchmarks in Section 6.4. We conduct additional experiments with our simulator in Section 6.5 and discuss related work in Section 6.6.

6.1 Preliminary Considerations

In this section, we briefly recapitulate the essence of our decision model and its optimization heuristics from the previous chapter, because we extend it by operator-stream-based scheduling in this chapter.

6.1.1 Decision Model

Since we will build on the results from Chapter 5, we now recapitulate the essence of our decision model.

The main idea is to assign each operation O a set of algorithms (e.g., the algorithm pool AP_O), where each algorithm utilizes exactly one processing device, such as a CPU or a *co-processor* (CP). Hence, a decision for an algorithm A using processing device X leads to an execution of operation O on X. This way, the model does not just decide on a processing device, but on a concrete algorithm on a processing device, thereby removing the need for a separate physical optimization stage. For an incoming data set D, the execution times of all algorithms of operation O are estimated using an *estimation component*, which passes the estimated execution times to a *decision component*. The decision component receives an optimization heuristic as an additional input, which allows the user to tune for response time or throughput. The decision component returns the algorithm A_i that has to be executed. We explain our heuristics in detail in Section 6.3. Furthermore, our approach is able to notice changes in the environment (e.g., changing data and workloads) and can adjust its scheduling decisions accordingly. This is a crucial property for use in a query optimizer, because the optimizer relies on cost estimations and decisions of HyPE. The model is a stable basis for an optimizer, because it:

1. Delivers reliable and accurate estimated execution times, which are used to compute the quality of a plan and enables the optimizer for a cost-based optimization and an accurate query-response-time estimation.
2. Refines its estimations at run-time, making them more robust in case of changes in data or workloads.
3. Decides on the fastest algorithm and therefore, processing device.
4. Requires no a priori knowledge about the deployment environment, for example the hardware in a system, because it learns the hardware characteristics by observing the execution-time behavior of algorithms.

Before being used, the decision model needs to be configured for an application by defining a fixed set of operations, where each operation has at least one algorithm. Afterwards, the user needs to define a feature vector for each operation, which describes the relevant factors for cost estimation (e.g., data size and selectivity for selections). Then, the user has to specify a learning method and a load adaption heuristic for each algorithm, and an optimization heuristic for each operator. In this chapter, we propose and evaluate new optimization heuristics. We provide more details on the other parts of HyPE in Chapter 5.

6.1.2 Optimization Heuristics

Until now, we only considered response-time optimization [37], which works fairly well for scenarios where the CPU and CP outperform each other depending on input data size and selectivity. However, for scenarios where the execution times of CPU and CP

algorithms do not have a break even point, meaning that they are not equally fast for a given data set, simple response-time optimization as in Chapter 5 is insufficient, because the faster processing device is over utilized while the slower processing device is idle.

This was the main criticism of Pirk and others for operation-based scheduling [154]. They introduced a co-processing technique *bitwise distribution*, which utilizes the CPU and the CP to process one operation, which achieves good device utilization by using the CP to pre-process a low-resolution index of the data and refining this intermediate result on the CPU. A different way to approach the problem of *efficient* utilization of processing resources in a CPU/CP system is the purposeful use of slower processing devices to achieve inter-device parallelism. The main challenge is to optimize the response time of single operations, while optimizing throughput for an overall workload.

Although HyPE was primarily designed to optimize performance, it could also optimize for other metrics, such as minimal energy consumption or memory usage.

6.2 Operator-Stream-based Query Processing

Next, we briefly describe the application scenario. Then, we compare single and multi-query optimization to motivate query processing based on operator streams. Such a query processor serializes a set of queries to an operator stream, which is then scheduled to available processing devices. Therefore, we extend HyPE to support operator-stream-based scheduling. Finally, we discuss the requirements of an efficient query-serialization algorithm.

6.2.1 Single versus Multi-Query Optimization

Single-query optimization is not suitable in case a set of complete queries is processed in parallel and operators from different queries can be executed interleaved. This is because of the high sensitivity of co-processors to *cache thrashing*. Let us consider an example with two queries: For each query, one operator is executed consecutively, but operators from different queries are executed interleaved. Each operator detects that the cached data of the previous operator has to be thrown away so that the new operator can do its job, which is similar to trashing during buffer management in traditional databases. We investigate the cache-thrashing effect in detail in Chapter 7. By contrast, a global optimization strategy can consider data locality for a query workload to build a global query graph, but at the cost of a significantly increased response time for single queries. Thus, it is necessary to combine on-the-fly operator scheduling with query optimization. We propose a two phase solution: First, serialize a set of queries to an operator stream and second, schedule the operator stream on all available co-processors. We discuss how we perform the query serialization in Section 7.4.2. In this chapter, we focus on the second part: distributing an already serialized workload on all available processing resources, which we discuss next.

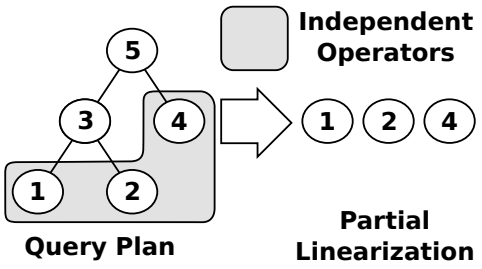


Figure 6.1: Example for Partial Query Linearization

6.2.2 Operator-Stream Extension of HyPE

We implemented our decision model from prior work [37] and our heuristics, which we discuss in Section 6.3, in HyPE. To support operation-stream-based scheduling, we refine our decision model as follows. Let $Op(O, D)$ be the application of the operation O to the data set D. A workload W is a sequence of operators:

$$W = Op_1(D_1, O_1)Op_2(D_2, O_2)\ldots Op_n(D_n, O_n) \tag{6.1}$$

Note that any query plan can be linearized into an operator stream by using materialization and chaining of outputs into inputs. Hence, a data set D_j can be the result of an operator Op_i $(i < j)$, which allows for data dependencies between operators. However, we assume that the stream contains only operators without any dependencies. This in turn means that queries may only be linearized in part (e.g., only leaf operators of a query plan are inserted in the stream; an operator inserts its parent in the stream on termination). We exemplify this in Figure 6.1, where the leaf nodes of the query plan (1,2,4) do not have any dependency and hence can be added to the stream. In contrast, operators 3 and 5 depend on the results of their child operators and have to wait for their completion. HyPE works in two phases: In the *training phase*, a part of the workload is used to train the model's approximation functions. In the *operational phase*, HyPE provides estimated execution times. On system startup, there are no approximation functions available such that HyPE cannot provide meaningful execution-time estimates, which may lead to poor results. Furthermore, processing-device utilization may change due to changing data and workloads. Hence, it is important not to schedule all operations at once, even if meaningful approximation functions exist. Based on these insights, we developed the following scheduling mechanism for HyPE.

We add a *ready queue* to each processing device. In case a ready queue is full, no more operators may be scheduled to the corresponding processing device. That way, HyPE keeps the processing devices busy, while maintaining the flexibility to react to changing processing device utilization (e.g., by keeping track of each ready queues estimated finishing time). Therefore, we have to select a suitable operator-queue length. A longer queue will reduce the likelihood that a processor is idle. However, at the same time, it will also reduce estimation quality, as the cost estimator has to predict execution times that will occur further away in the future. Based on our experiments, we found that a maximal queue length of 100 operators achieves good and stable performance.

6.2.3 Mapping Query Plans to Operator-Streams

From our workload definition, we accept operator streams only as input. However, a database query typically consists of multiple data-intensive operators, which depend on each other (e.g., a join gets a filtered column from a selection as input). The operator-stream processor needs to consider these inter-operator dependencies between operators of queries when generating a stream of *independent* operators.

Precondition

The precondition to map query plans to operator streams is that the operators are "self-contained" schedulable units. That is, operators can be processed independently from each other on different processing devices. Therefore, we need operator-at-a-time bulk processing [132] (or at least block-wise processing like in Vectorwise [31] or MapD [138]), in which one operator consumes its input and materializes its output. Bulk processing allows for operator-based scheduling, the precondition for our operator-stream scheduling, but does not support inter-operator parallelism by pipelining, which may have a negative impact on performance on CPU side. However, pipelining can also be achieved by compiling sub-queries between pipeline breakers (e.g., sort operations) [143]. Then, a compiled sub-query can be executed as one bulk operator. However, it is unclear how to estimate the cost of the compiled (sub-)queries. For ad-hoc queries, it is not feasible to learn the performance behavior of compiled queries. Furthermore, we cannot combine cost models that were learned for bulk operators. Therefore, our approach does not work currently for compiled queries.

Challenges of a Query-Serialization Algorithm

A simple way to create an operator stream from a query plan is to perform a bottom-up traversal and add the operators of the same level to the operator stream. This ensures that dependent operators are executed after their predecessors.[1] However, an efficient query-serialization algorithm needs to optimize for three goals simultaneously:

Data-Locality-Aware Operator Placement

An efficient strategy executes operators on the same processing device, where their predecessors were executed. This way, the overhead due to data transfers between CPUs and CPs is reduced.

Inter-Device Parallelism

It is important to use all available processing resources to decrease a queries response time (i.e., using an already busy processing device slows down query processing). Therefore, a query should be executed on not only one, but multiple processing devices to efficiently exploit inter-device parallelism. However, if we use more (co)-processors, additional data transfers may become necessary in case data is not cached in a CP.

[1] Note that some systems (e.g., MonetDB) already possess plans in form of an operator stream (e.g., MonetDB's MAL plans).

Heterogeneous Properties

Database operations have very different properties. A selection is well suited to run on CPUs, because of their highly optimized branch prediction mechanisms whereas aggregations are typically faster on GPUs because of their outstanding numeric processing power. By assigning an operation to the most suitable processing device, we can fully exploit the heterogeneous nature of hybrid CPU/CP systems. However, this may conflict with data locality and inter-device parallelism.

Toward a Mapping Strategy

Overall, we need a strategy that serializes a query plan to an operator stream such that (1) the dependencies between the operators are not violated, (2) available processing devices are fully utilized, (3) the overhead of data transfers is kept low due to clever data placement, and (4) each processing device processes the operations that it can handle most efficiently.

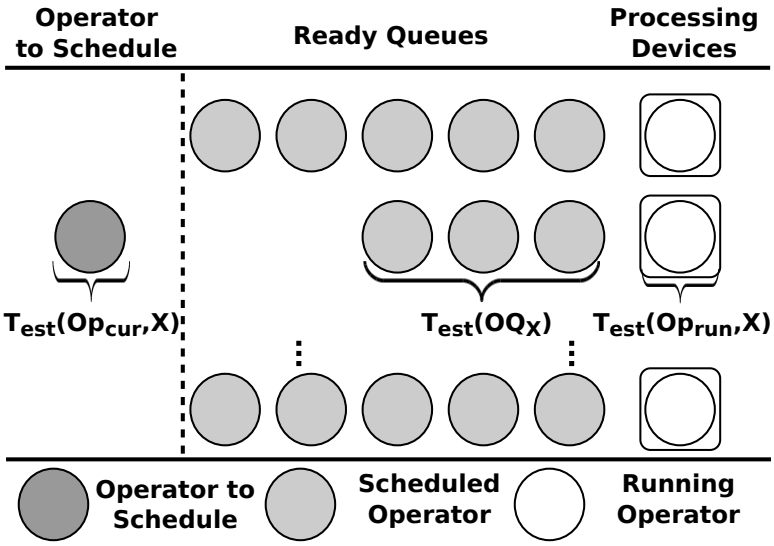
One suitable strategy would be to assign subtrees of the query plan to processing devices, so that the data-intensive processing is distributed on different processing devices (CPU and co-processors) and the assembly of the results is done by one processing device (typically the CPU). Since we focus in this chapter on scheduling already serialized queries to available processing devices, we address the serialization of queries to operators streams in Chapter 7.

6.3 Optimization Heuristics

In this section, we discuss our main contribution, the heuristics for response time and throughput optimization for efficient processing device utilization.

6.3.1 Assumptions for Load Adaption

Our decision model continuously refines performance estimations by collecting new observations (data properties and execution time). However, this mechanism requires a continuous supply of new observations, which means that every processing device has to be used regularly [37]. This in turn means that each processing device is used for data processing, resulting in *inter-device parallelism*. In other words, all techniques enforcing inter-device parallelism ensure the continuous supply of new observations and that the performance models also reflect the current data properties (e.g., skew). The downside of continued refinement of cost models is a steady overhead during run-time. However, this overhead is negligible in HyPE, because it assumes bulk processing, a coarse-grained granularity for monitoring. By default, HyPE updates the cost model of an algorithm in mini batches. Typically, it collects 100 new observations and then, recomputes the cost model, which reduces computational overhead and the impact of outliers.



Op_{run} – Operator currently Executed, Op_{cur} – Operator to be Scheduled,
OQ_X – Ready Queue of Processing Device X,
$T_{est}(Op, X)$ – Estimated Execution Time of Operator Op on Processing Device X

Figure 6.2: WTAR: Load Tracking with Ready Queues

6.3.2 Response Time

Next, we discuss two heuristics for response-time optimization.

Simple Response Time (SRT)

The decision component gets a set of operators with their estimated execution times as input. The SRT heuristic chooses the algorithm that is likely to have the smallest execution time. The problem with SRT is that using always the fastest algorithm does not consider when the corresponding algorithm is actually executed. If the model shifts the whole workload to the GPU, operators have to wait, until their predecessors are executed. Therefore, over utilization of a single device slows the processing of a workload down in two ways: (1) individual execution times are likely to increase, and (2) the waiting time until an operator can start its execution increases.

Waiting-Time-Aware Response Time (WTAR)

We propose an optimization approach WTAR that is aware of the waiting time of operations on all processing devices and schedules an operation to the processing device with the minimal time the operation needs to terminate. WTAR is a modified version of the *heterogeneous earliest finishing time* (HEFT) algorithm [189]. In contrast to HEFT, WTAR is designed to schedule an operator stream, and therefore, does not assume an a priori known workload. Furthermore, WTAR uses per operation cost estimations instead of the average algorithm (task) execution cost. This is because HyPE provides accurate performance estimations for algorithms. Let $T_{est}(Op_i, X)$ be the estimated execution time of operator Op_i on processing device X, OQ_X be the operator queue of

all operators waiting for execution on processing device X and $T_{est}(OQ_X)$ the estimated completion time of all operators in OQ_X:

$$T_{est}(OQ_X) = \sum_{Op_i \in OQ_X} T_{est}(Op_i, X) \tag{6.2}$$

Let Op_{run} be the operator that is currently executed, $T_{fin}(Op_{run}, X)$ the estimated time until Op_{run} terminates, and Op_{cur} the operator that shall be executed on the processing device that will likely yield the smallest response time. Then, the model selects the processing device X, where $\min(T_{est}(OQ_X) + T_{fin}(Op_{run}, X) + T_{est}(Op_{cur}, X))$. Note that this approach avoids the overloading of one processing device, because it considers the time an operator has to wait until it is executed. If this waiting time gets too large on processing device X w.r.t. processing device Y, the model will choose Y.

6.3.3 Throughput

Next, we discuss heuristics that optimize a workload's throughput.

Round Robin (RR)

Round robin is a simple and widely used algorithm [187], which assigns tasks alternating to available processing devices without considering task properties. We use it as a reference measure to compare our approaches with throughput-oriented algorithms. RRs simplicity is its major disadvantage: it only achieves good results in case processing devices execute tasks equally fast or else RR over/under utilizes processing devices, which may lead to significant performance penalties. An over utilization of a slow processing device is worse than over utilizing the fastest processing device, as in case of SRT. Hence, we propose a more advanced heuristic for throughput optimization in the following.

Threshold-based Outsourcing (TBO)

Recall that a decision for an algorithm executes an operation on exactly one processing device (e.g., CPU merge sort on the CPU and GPU merge sort on the GPU). The problem with SRT is that it over utilizes a processing device in case one algorithm always outperforms the others. This violates the basic assumption of our decision model that all algorithms are executed regularly. Therefore, we modify SRT to choose a sub-optimal algorithm (and therefore, a sub-optimal processing device) under the condition that the operation is not significantly slower.[2] Therefore, we need to keep track of (1) the passed time to decide, *when* a different algorithm (processing device) should be used and (2) the estimated slowdown, a sub-optimal algorithm execution may introduce to prevent an outsourcing of operations to unsuited processing devices (e.g., let the GPU process a very small data set).

[2]Hence, TBO is limited to one optimized algorithm per device.

With (1), we add a timestamp to all operations as well as their respective algorithms. Each operation O is assigned to one logical clock $C_{log}(O)$, which basically provides the number of operation executions. Each time an algorithm A is executed, its timestamp $ts(A)$ is updated to the current time value ($ts(A) = C_{log}(O)$). In case the difference of the timestamps of A and $C_{log}(O)$ exceeds a threshold $\mathcal{T}_{log}$ (a logical time), the respective processing device X of A is considered idle. Therefore, an operation should be outsourced to X in order to balance the system load on the processing devices. With (2), we introduce a new threshold, the maximal relative slowdown $MRS(A_{opt}, A)$ of algorithm A compared to the fastest (optimal) algorithm A_{opt}.

The operation O may be executed by a sub-optimal algorithm A (using a different processing device) if and only if A was not executed for at least $\mathcal{T}_{log}$ times and the relative slowdown does not exceed the threshold $MRS(A_{opt}, A) > \frac{T_{est}(A) - T_{est}(A_{opt})}{T_{est}(A_{opt})}$. For our experiments, we performed a pre-pruning phase to identify a suitable configuration of the parameters. We found that $MRS(A_{opt}, A) = 50\%$ and $\mathcal{T}_{log} = 2$ leads to good and stable performance. This configuration means that TBO attempts to outsource an operation to a processing device X if it was not used for the last two operations. For n co-processors, $\mathcal{T}_{log}$ should be $n + 1$.

Probability-based Outsourcing (PBO)

The problem to select the fastest processing device can be transformed into a *multi-armed bandit* (MAB) problem [191]. A multi-armed bandit problem is to select the bandit (processing device) with the highest benefit (lowest processing time). An efficient algorithm has to balance between using the bandit with the highest known reward (called *exploitation*) and searching for a bandit with higher reward (called *exploration*). An efficient approach for the MAB problem is the *softmax learning* strategy [178], which assigns each arm a probability to be optimal and randomly chooses an arm w.r.t. to the computed probability distribution.

In contrast to the traditional multi-armed bandit problem, we can use all arms (processing devices) in parallel. Hence, we want to favor the processing device with the highest benefit (smallest execution time) while using the other processing devices in parallel to reduce the overall workload execution time. Therefore, we adapt the idea of the softmax learning strategy and assign each algorithm of each processing device a probability for executing an operation.[3] Hence, we use the continuous *exploration* of softmax learning with constant tuning parameter ($\tau = 1$). The probability depends on the estimated execution time of $A_i \in AP_O$ for the data set D:

$$P(A_i) = 1 - \frac{T_{est}(A_i)}{\sum_{A_j \in AP_O} T_{est}(A_j)} \tag{6.3}$$

[3]We assume one optimized algorithm per processing device, because we focus on inter-processor parallelism. Hence, the algorithm pool AP_O contains one algorithm per PD.

Consequently, HyPE favors algorithms on faster processing devices over algorithms on slower processing devices. For long term scheduling, this strategy leads to a statistically uniform distribution of load on all processing devices. Furthermore, it utilizes all processing devices, even if one device (e.g., a GPU) always outperforms another processing device (e.g., a CPU). Therefore, all processing devices are kept busy with a reduced probability to under/over utilize a processing device compared to SRT or RR.

6.3.4 Other Optimization Goals

HyPE can be used for other optimization goals as well. For example, in environments where energy consumption is the most critical factor, HyPE can learn the correlation between an operators feature vector and the operators energy consumption. Similar, the memory consumption can be an issue (e.g., if the co-processor has very small device memory), and the algorithm with the smallest memory footprint should be used. To use HyPE with a different optimization goal, the user has to supply measurements from an appropriate measure for the optimization goal (e.g., joule for energy consumption). Since the focus of this thesis is to optimize the performance of database systems by either optimizing response time or throughput, we will not further investigate other optimization goals.

6.4 Evaluation

To judge feasibility of our heuristics, we conducted several experiments that evaluate response time and throughput for four use cases: aggregations, column scans, sorts, and joins. We selected these use cases, because they are essential stand alone operations during database query processing, but some are also sub-operations of complex operations such as the CUBE operator [75] (e.g., aggregations and selections). Although some optimization heuristics have already proven applicable (e.g., response time), we are interested in specific aspects for all optimization heuristics relevant for a database system. The goal of the evaluation is to answer the following research questions:

RQ1: Which of our optimization heuristics perform best under varying workload parameters?
RQ2: How does the optimization heuristic impact the quality of estimated execution times?
RQ3: Which optimization heuristic leads to best CPU/GPU utilization ratio and overall performance?
RQ4: How much overhead does the training phase introduce w.r.t. the workload execution time?
RQ5: Which optimization heuristics are suitable for which use cases?

Providing answers for the aforementioned questions is crucial to meet the requirements for an optimizer and to judge feasibility of our overall approach.

6.4.1 Experiment Overview

In the following, we describe the experiments we conducted to answer the research questions. First, we present implementation details on our use cases as well as the experimental design (i.e., which benchmarks we used). Second, we discuss the experiment variables. Third, we present the analysis procedure. Since our evaluation system CoGaDB is a column-store, we only need to model the part of the database that is accessed by the generated queries. Therefore, it would not reduce the performance to have many columns in a table (e.g., in fact tables).

Aggregation

A data set for an aggregation operation is a table with two integer columns in a key-value form whereas the key refers to a group for which their values (second column) needs to be aggregated. We use as aggregation function SUM, because it is a very common aggregation function in database systems.

Column Scan

A data set for a column scan operation is a table with one column. The values in the column are integer values ranging from 0 to 1000. The benchmark generates an operation by computing a random filter value $val \in \{0, \ldots, 1000\}$ and a filter condition $filt_{cond} \in \{=, <, >\}$.

Sort

A data set for a sort operation is a table with one column. The values in the column are integer values ranging from 0 to 1.000.000.000 to create data sets with varying number of duplicates. We used the highly optimized and parallel sort algorithms of the Threading Building Blocks Library for the CPU and the Thrust Library for the GPU.[4]

Join

A join operation gets two data sets as input, in which the first data set contains a table with one column having the primary keys (T_{PK}) and the second data set contains a table with the foreign-key column (T_{FK}). T_{PK} always contains as many disjoint keys as specified in the data-set size. T_{FK} corresponds to exactly one T_{PK}. To generate an input data set of size X, we generate 10% of X as primary keys and 90% of X as foreign keys, because foreign key tables are typically much larger than primary key tables, especially in a data warehouse environment. In the experiments, the benchmark randomly selects a combined (T_{PK}, T_{FK}) data set and computes the join between the two tables. We adapted the sort-merge join of He and others for the GPU [85] and a hash join on the CPU.

[4] www.threadingbuildingblocks.org, thrust.github.com

Experimental Design

The used benchmark is a crucial point to conduct a sound experiment. We use a micro benchmark for single operations in CoGaDB. The user has to specify three parameters: a maximal data set size, the number of data sets in the workload and a data-set generation function, for which we input the first two parameters, and get in return a data set for the respective operation. The specified number of data sets is generated using a use-case-specific data generator function to allow for an evaluation of HyPE without restricting generality. We measure the overall runtime (including data transfers), estimation error, device utilization, and training length for a workload. The test machine has an Intel® Core™i5-2500 CPU @3.30 GHz with 4 cores and 8 GB DDR3 main memory @1333 MHz, and a NVIDIA® GeForce® GT 640 GPU (compute capability 2.1) with 2 GB device memory. The operating system is Ubuntu 12.04 (64 bit) with CUDA 5.0 (driver 304.54). For all experiments, all data sets fit into main memory. The source code of CoGaDB and our benchmark is available online to enable reproducibility.[5]

Variables

We conduct experiments to identify which of our heuristics perform best under certain conditions. We evaluate our approach for the following variables: (1) number of operations in the workload ($\#op$), (2) number of different input data sets in a workload ($\#datasets$), and (3) maximal size of data sets ($size_{max}$).

Analysis Procedure

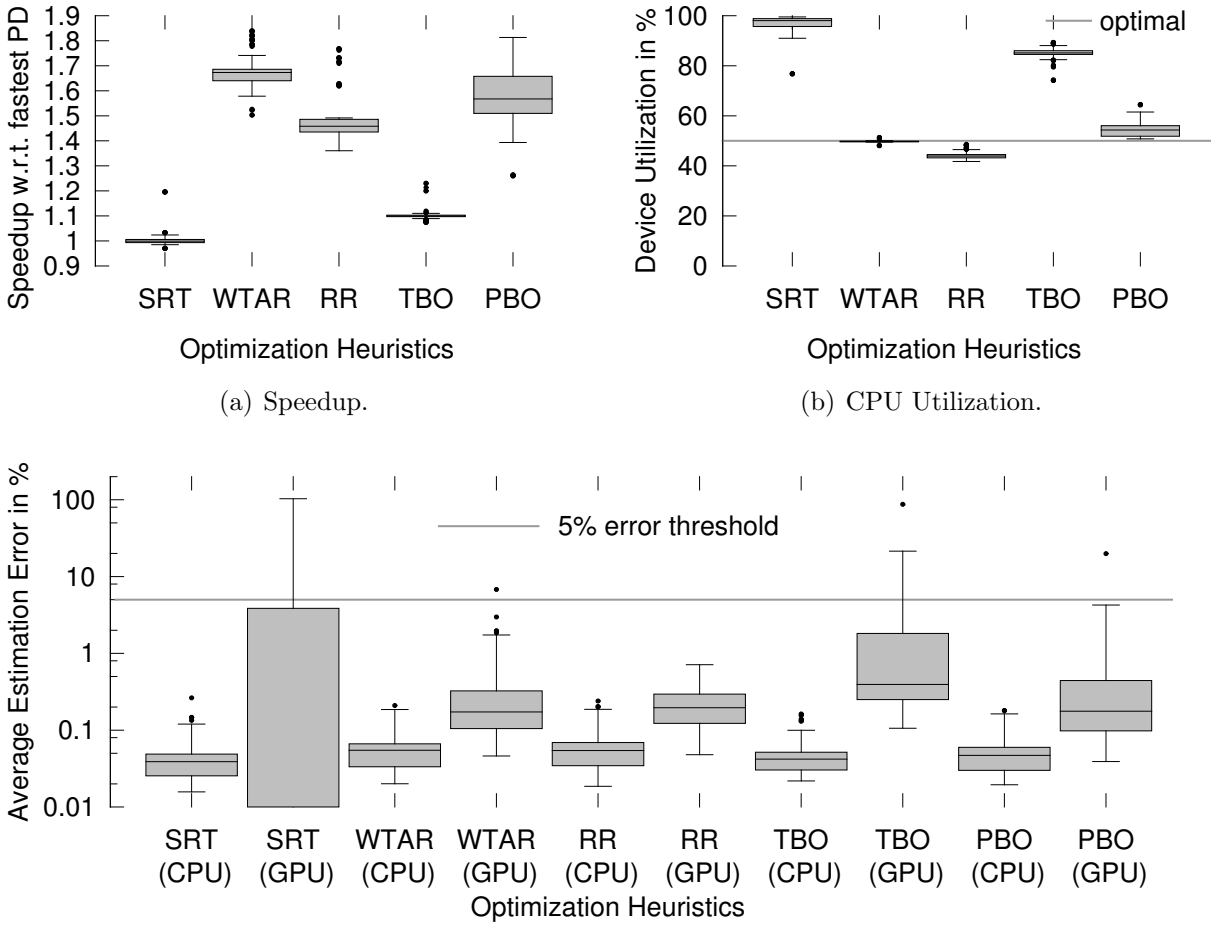
We evaluate our results separately for each use case using boxplots over all related experiments to prove that our optimization heuristics are stable for the whole parameter space ($\#op, \#datasets, size_{max}$). We vary the three variables in a ceteris paribus analysis [177] with $(500, 50, 10MB)$ as base configuration and only vary one parameter at a time, leaving the other parameters constant (e.g., $(1000, 50, 10MB)$,$(500, 100, 10MB)$):

1. $\#op \in \{500, \ldots, 8000\}$
2. $\#datasets \in \{50, \ldots, 500\}$
3. $size_{max} \in \{10MB, \ldots, 100MB\}$

Note that higher values for $\#datasets$ or $size_{max}$ would result in a database exceeding our main memory and hence, violating our in-memory assumption.[6] As quality measures, we consider (1) the speedup w.r.t. the execution of a workload on the fastest processing device, which can be obtained using static scheduling approaches (e.g., Kerr and others [116]), (2) average estimation errors, which is ideally zero, and (3) device utilization. In case the workload is unevenly distributed, one processing device is over utilized, whereas others are under utilized, increasing execution skew. An ideal device utilization in a scenario of n processing devices is that each processing device processes $1/n$ of the workload. For our test environment, a perfect utilization would be to use 50% of workload execution time on CPU and 50% on GPU. (4) Finally, we investigate the relative training times depending on the optimization heuristics.

[5] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/supplemental.php

[6] Note that we need additional memory for intermediate results or the operating system.



(a) Speedup.

(b) CPU Utilization.

(c) Average Estimation Errors of Cost Models Depending on Optimization Heuristics.

PD – Processing Device, SRT – Simple Response Time, WTAR – Waiting Time Aware Response Time, RR – Round Robin, TBO – Threshold-based Outsourcing, PBO – Probability-based Outsourcing

Figure 6.3: Aggregation Use Case.

6.4.2 Results

Now, we present only the results of the experiments. In Section 6.4.3, we answer the research questions and discuss the achieved speedups, estimation accuracy, device utilization, and relative training times of the heuristics. The results are accumulated over all experiments and displayed as box plots to illustrate the typical characteristics (e.g., mean and variance) w.r.t. a quality measure. A box plot visualizes a data distribution by drawing the median, the interquartile ranges as box, and extremes as whiskers [12]. Note that 50% of the points are in the box, and 95% are between the whiskers. Outliers are drawn as individual points.

Result of Aggregation

Figure 6.3(a) illustrates the achieved speedups for the fastest processing device (CPU) of our optimization heuristics over all experiments. To answer RQ1, we observe that (1)

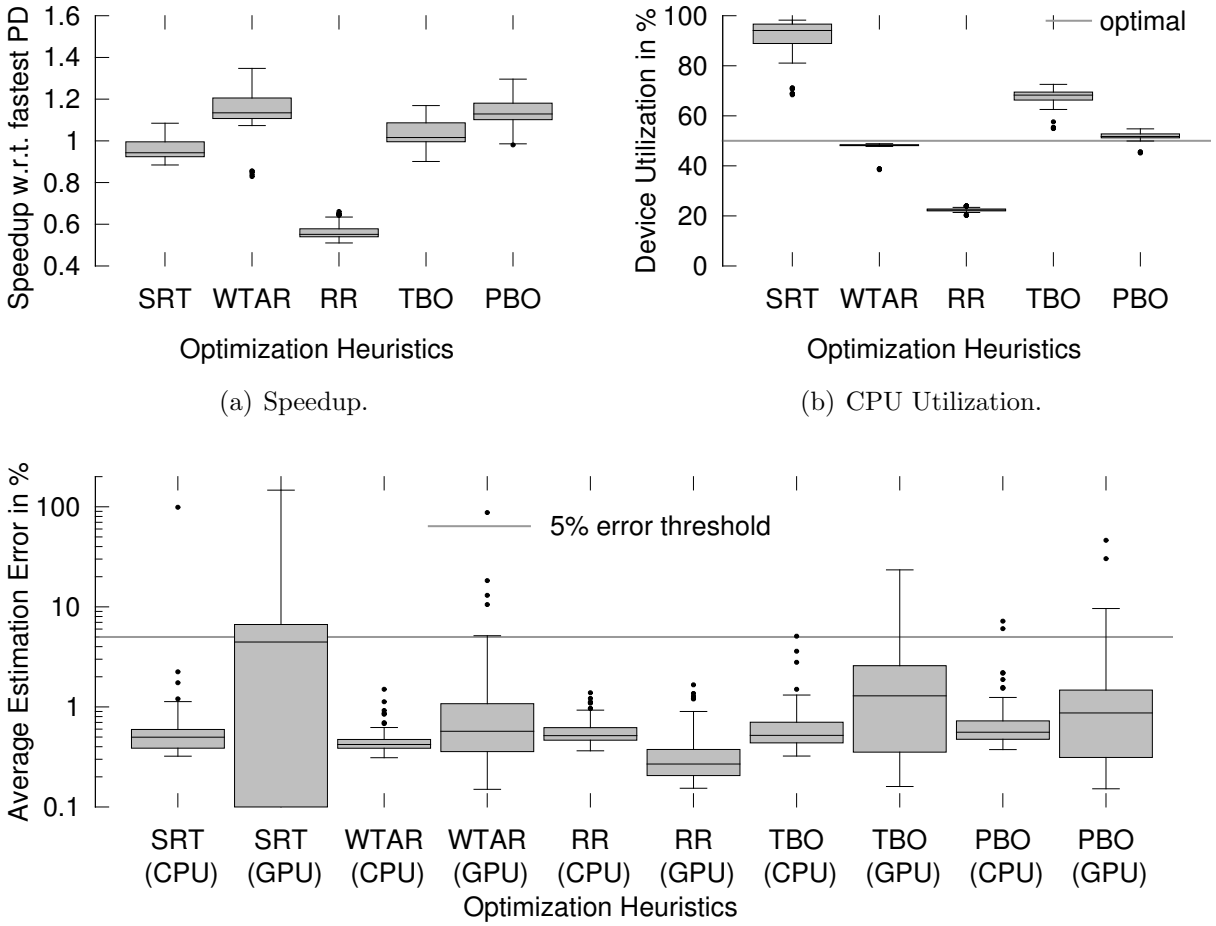
SRT has no significant speedup compared to the fastest processing device, because the box plot is located at 100%, which means that only a single device (CPU) is utilized. (2) WTAR is significantly faster than PBO (by 3%) and RR (by 8%), and (3) TBO is inferior to RR (by 31%), WTAR (by 37%), and PBO (by 34%), but achieves higher performance than SRT (by 8%). Figure 6.3(b) illustrates the device utilization over all experiments. The gray horizontal line exemplifies the ideal device utilization. To answer RQ3, we observe that (1) SRT and TBO tend to over utilize the CPU. (2) We see that the box plot of WTAR lies on the horizontal line, which represents the best utilization. WTAR, PBO, and RR are performing best, whereas RR has slightly worse device utilization than WTAR and is slightly better than PBO. Figure 6.3(c) shows the estimation accuracy of the optimization heuristics. To answer RQ2, we observe that (1) the accuracy is typically higher for CPU algorithms compared to GPU algorithms, (2) SRT and TBO exceed our error threshold for GPU algorithms, whereas the other optimization heuristics are acceptable, because the estimation error is smaller than our defined 5% threshold.

Results of Column Scan

Figure 6.4(a) illustrates the achieved speedups for the fastest processing device (CPU) of our optimization heuristics over all experiments. For this use case, the CPU consistently outperforms the GPU by an average speedup of 2.7. We see that WTAR and PBO have the lowest response time, which answers RQ1. In contrast to aggregations, the t-tests indicate that there is no systematic difference between WTAR and PBO for column scans (e.g., they are equally fast). Furthermore, we observe that RR performs poorly (e.g., $\approx$2 times worse than WTAR). Figure 6.4(b) illustrates the device utilization over all experiments. To answer RQ3, we observe that (1) SRT does not lead to a speedup for column scans, because SRT over utilizes one processing device on a regular basis indicating that it is not suitable for efficient task distribution and (2) WTAR and PBO achieve nearly ideal device utilization, whereas TBO tends to over utilize the CPU. RR consistently over utilizes the GPU, which explains the poor performance of RR. We show the estimation accuracy of the optimization heuristics for column scans in Figure 6.4(c). We see that SRT, TBO, and PBO exceed the error threshold on the GPU.

Results of Sort

Figure 6.5(a) illustrates the achieved speedups for the fastest processing device (GPU) of our optimization heuristics over all experiments. For the sort use case, the GPU consistently outperforms the CPU by an average factor of 2.42. We see that WTAR has the lowest response time and leads to a speedup of $\approx$ 1.4, which answers RQ1. Furthermore, we observe that (1) SRT achieves no speedup compared with a GPU only scenario. (2) Since the GPU is faster than the CPU, the RR heuristic leads to poor performance, because it heavily over utilizes the CPU leading to a higher workload execution time than executing all operations on the GPU. (3) TBO does not outsource the operations to the GPU aggressively enough, leading to a performance



(a) Speedup.

(b) CPU Utilization.

(c) Average Estimation Errors of Cost Models Depending on Optimization Heuristics.

PD – Processing Device, SRT – Simple Response Time, WTAR – Waiting Time Aware Response Time, RR – Round Robin, TBO – Threshold-based Outsourcing, PBO – Probability-based Outsourcing

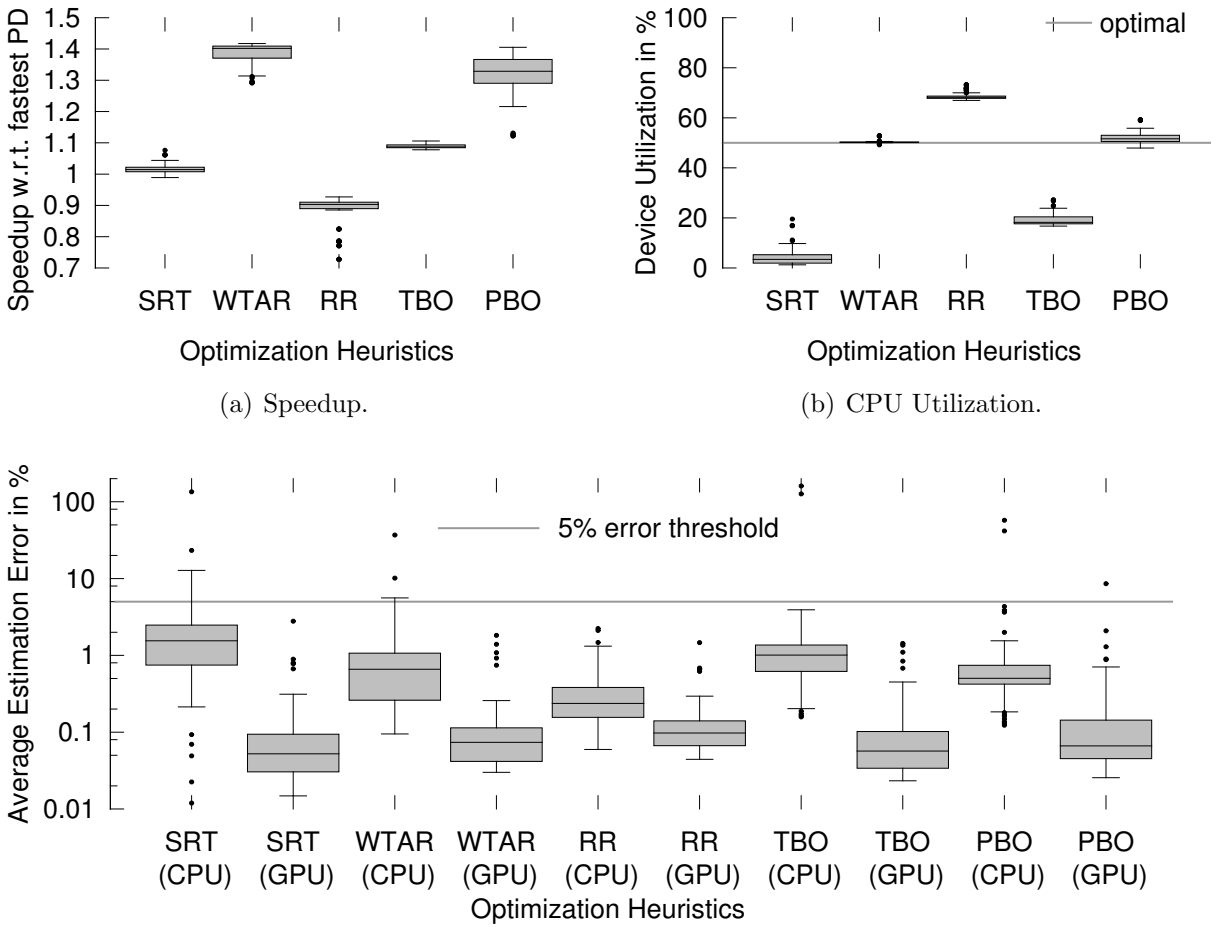
Figure 6.4: Column Scan Use Case.

penalty compared with WTAR (by 20%). However, TBO is still faster compared to a CPU only approach (by 9%).

Figure 6.5(b) illustrates the device utilization over all experiments. To answer RQ3, we observe that (1) SRT consistently over-utilizes the GPU. This limited inter-device parallelism causes a significant slowdown compared to WTAR and (2) WTAR and PBO achieve nearly ideal device utilization, whereas TBO tends to over utilize the GPU. We show the estimation accuracy of the optimization heuristics for sorts in Figure 6.5(c). We make the same observations as for aggregations and selections. That is, heuristic WTAR outperforms all other heuristics (RQ2).

Results of Join

Figure 6.6(a) illustrates the achieved speedups of our optimization heuristics compared to the fastest processing device (GPU). For the join use case, the GPU consistently



(a) Speedup.

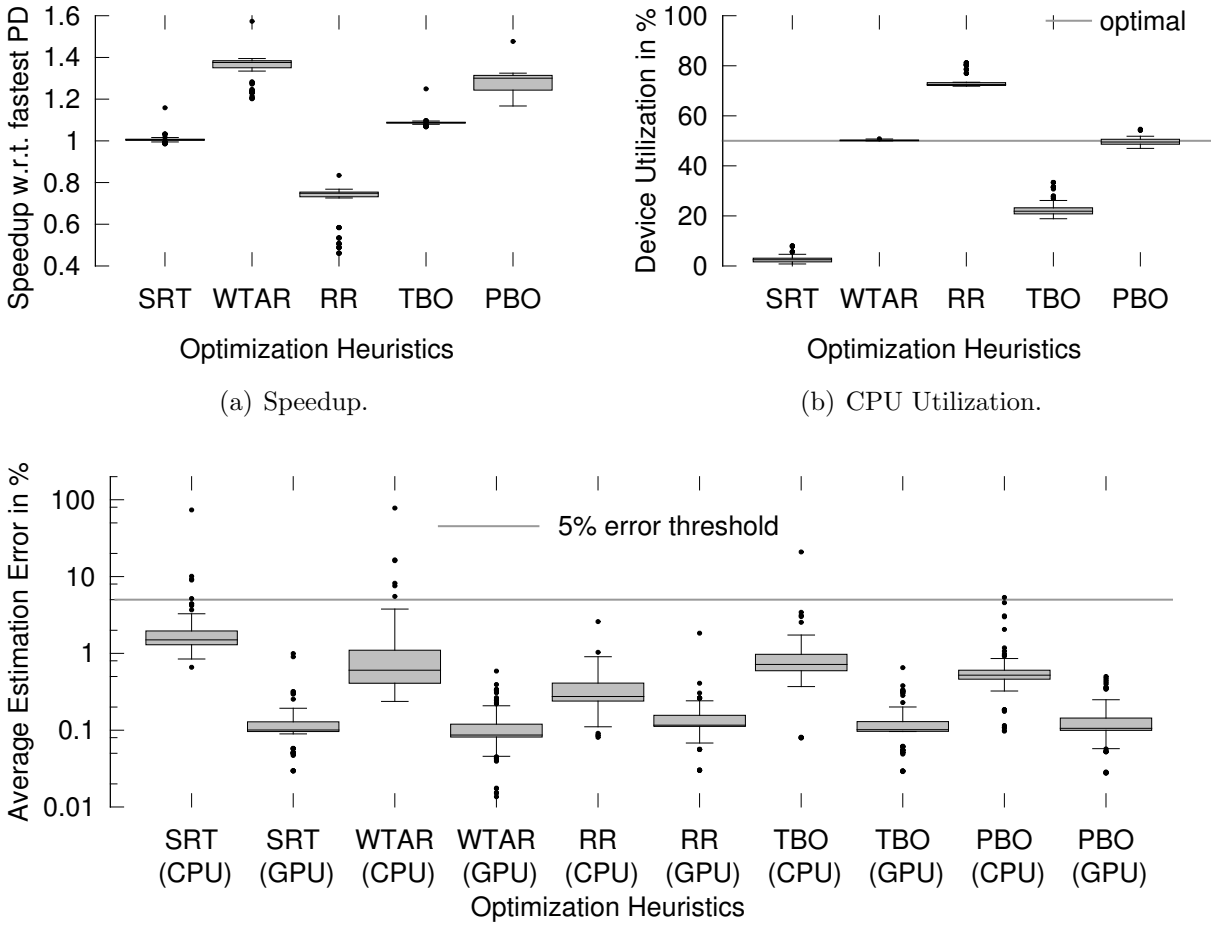
(b) CPU Utilization.

(c) Average Estimation Errors of Cost Models Depending on Optimization Heuristics.

PD – Processing Device, SRT – Simple Response Time, WTAR – Waiting Time Aware Response Time, RR – Round Robin, TBO – Threshold-based Outsourcing, PBO – Probability-based Outsourcing

Figure 6.5: Sort Use Case.

outperforms the CPU by an average factor of 3.56. We see that WTAR has the lowest response time, which answers RQ1. Furthermore, we observe that (1) SRT achieves no speedup with respect to the GPU due to missing inter-device parallelism. (2) The RR heuristic leads to poor performance, because it heavily over utilizes the CPU. (3) TBO does not outsource the operations to the GPU aggressively enough leading to a performance penalty similar to the other use cases. Figure 6.6(b) illustrates the device utilization over all experiments. To answer RQ3, we observe that (1) SRT consistently over-utilizes the GPU (which is the fastest processing device for this use case) and (2) WTAR achieves nearly ideal device utilization, whereas RR tends to over utilize the CPU, which leads to a performance decrease of 20% compared with a GPU only execution. SRT and TBO over utilize the GPU on a regular basis, causing a significant slowdown compared to WTAR. We show the estimation accuracy of the optimization



(a) Speedup.

(b) CPU Utilization.

(c) Average Estimation Errors of Cost Models Depending on Optimization Heuristics.

PD – Processing Device, SRT – Simple Response Time, WTAR – Waiting Time Aware Response Time, RR – Round Robin, TBO – Threshold-based Outsourcing, PBO – Probability-based Outsourcing

Figure 6.6: Join Use Case.

heuristics for joins in Figure 6.6(c). We make the same observations as for aggregations, selections, and sorts. That is, heuristic WTAR outperforms all others (RQ2).

6.4.3 Discussion

Overall, WTAR outperforms the other heuristics, especially when relative speed of processing devices differs. To ensure that our results are not coincidence, we performed t-tests with $\alpha = 0.001$ [12]. The result is that WTAR is significantly faster than the other heuristics. However, we could not verify for the column scan use case that WTAR is significantly faster than PBO. However, this also means that *no* heuristic was significantly faster than WTAR for all use cases. Therefore, we conclude that WTAR achieves the highest performance for all uses cases. Furthermore, it has a very low variance in workload execution time and therefore, achieves stable performance. Hence, the answer for RQ5 is that there is one heuristic performing best for all use cases:

WTAR. The speedup experiments allow for a direct comparison with static scheduling approaches, which select one processing device before runtime such as Kerr and others [116]. We measured average performance improvements of ≈69% (aggregations), ≈14% (column scans), ≈38% (sorting) and ≈35% (joining) for WTAR w.r.t. to the fastest processing device.

Regarding RQ2, the estimation quality of WTAR, PBO, and RR is stable across different use cases over the investigated parameter space, whereas SRT and TBO frequently exceeds the error threshold (5%).

To answer RQ3, we discuss device utilization. WTAR proved superior to PBO, RR, TBO and SRT. RR gets worse with increasing speed difference of processing devices. In contrast, WTAR delivers nearly ideal device utilization with marginal variance over a large parameter space. SRT mostly uses one processing device, indicating that it is not suitable for efficient task distribution. Overall, SRT performs poorly in case there is no break-even point between CPU and GPU algorithms execution-time curves, because SRT over utilizes one processing device, resulting in execution skew and increasing overall workload execution time. However, our evaluation in Chapter 5 clearly shows the benefit of SRT in case a break-even point exists.

To answer RQ4, we investigate HyPE's overhead by measuring the training time and compute the relative training time w.r.t. the workload execution time for aggregations (Figure 6.7(a)), columns scans (Figure 6.7(b)), sorting (Figure 6.7(c)), and joining (Figure 6.7(d)). It is clearly visible that the performance impact of the training is marginal.

Summary

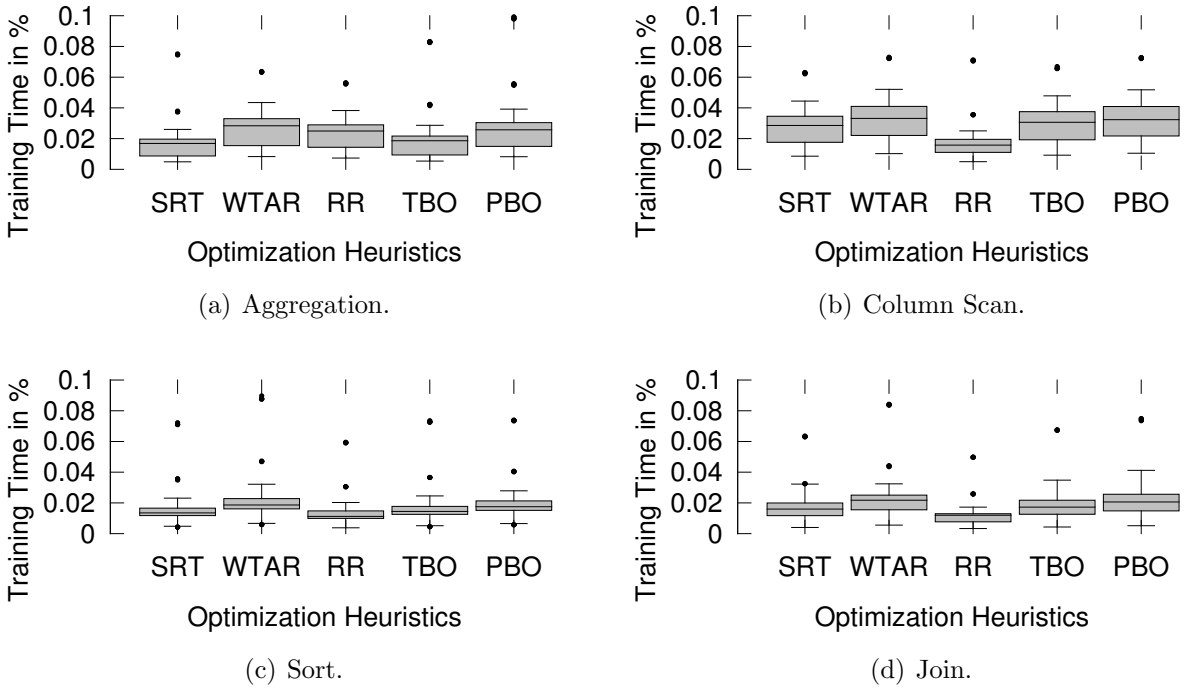
For all use cases, WTAR outperformed all other optimization heuristics in terms of performance, estimation accuracy, and equal processing device utilization. In some experiments, RR caused slightly better estimated execution times. Overall, we observe that estimation accuracy strongly depends on the optimization heuristics (RQ2), because the heuristics directly influence the operations executed on the processing devices, which in turn trains the approximation functions more or less. RR is likely to perform worse than WTAR in case a processing device is significantly faster than the others, because in this case RR leads to an uneven device utilization, which we observed for column scans, sorts and joins. We conclude that out of the considered optimization heuristics, WTAR is the most suitable for use in a database optimizer (RQ1–5).

6.4.4 Threats to Validity

We now discuss threats to internal and external validity.

Threats to Internal Validity

We performed a t-test to ensure that our results are statistically sound and did not occur by chance. Furthermore, we have to consider measurement bias when measuring



(a) Aggregation.

(b) Column Scan.

(c) Sort.

(d) Join.

Figure 6.7: Relative Training Time of Use Cases.

execution times and device utilization. Therefore, we repeated each experiment during our ceteris paribus analysis five times. We depict all measurements as box plots including the outliers and use the arithmetic mean during heuristic comparison. This allows for a precise evaluation of the reliability of our approach.

Threats to External Validity

We are aware that using micro benchmarks does not automatically reflect the performance behavior of real-world DBMS. However, we argue that (1) they are a necessity for an in-depth analysis of our optimization heuristics and (2) we selected a representative set of operator types that are very common. We investigated workloads of the same operator types and thus, experimented with homogeneous workloads only. However, the operator type is irrelevant for the load balancing heuristics, because the heuristics consider the estimated execution times only. Each workload consists of operators that have different run times, which is the major reason for load imbalance and our experiments provide strong evidence that this is handled well by WTAR. We evaluate the WTAR heuristic as part of the query chopping technique in Section 7.5, where the workloads consist of different operator types.

The implementation details of database operators may differ in industrial DBMS. We counter that by using a learning-based approach to allow us to accurately predict performance without knowing the algorithms in detail. We address hardware heterogeneity in the same way.

We modeled only the relevant parts of a database for our benchmarks. However, this is sufficient in a column store, in which no additional processing costs arise for one operation when additional columns are added to the table (except the tuple materialization operator, but we assume the use of late materialization [6]). We cannot automatically generalize our results to all DBMS workloads, but we carefully performed many experiments for four different, but common use cases.

Finally, we only did experiments for a setup of one CPU and one GPU. To judge feasibility of our approach in a general scenario of n co-processors, additional evaluation is necessary, which we present next.

6.5 Simulation

In this section, we discuss the scalability of query processing for an increasing number of co-processors and performance diversity. We investigate the following research questions:

RQ6: How does the best optimization heuristics scale for n processing devices ($n > 2$)?
RQ7: What are the most important impact factors on the performance of our best optimization heuristic on a hybrid CPU/CP system?

Providing answers for the aforementioned questions is crucial to predict the scaling behavior of our overall approach.

6.5.1 Experiment Overview

In the following, we describe our experiment that we conducted to answer the research questions. First, we present implementation details of our simulator and the experimental design. Second, we discuss the experiment variables. Third, we present the analysis procedure.

Architecture of Simulator

Our previous use cases considered only two processing devices: one CPU and one GPU. Furthermore, we cannot vary the processing devices relative speed to each other: On varying hardware platforms, processing devices will have a different relative speed to each other (e.g., a CPU of one system is 1.5 times faster as the GPU and in the other system 2 times slower). Since an exhaustive hardware analysis is infeasible, we run a simulation that abstracts from real hardware and allows us to tune the important processor parameters.

We model the simulation environment as follows: We assume that we have one optimized algorithm per operation per processing device. Hence, we have n algorithms $A_1, ..., A_n$ and n processing devices $PD_1, ..., PD_n$, where algorithm A_i is executed on processing device PD_i. To model different performance of each processing device, we introduce the relative speed $rsp(PD_i)$, which contains the average speedup of a processing device

PD_i to processing device PD_1. That is, PD_1 acts as the base line and all other $n-1$ processing devices behave relative to PD_1. For example, a relative speed greater one means that PD_i is a faster processing device than PD_1 and vice versa. Note that the relative speed depends also on the operation. The goal is to schedule the operator workload on all available processing devices in consideration of their relative speed and their current load condition. An algorithm's execution time $T(A_i)$ is a function of the input data size $size$ and a jitter function jit:

$$T(A_i) = size \cdot rsp(PD_i) + |jit| \tag{6.4}$$

The jitter function models the variance in execution times for all algorithms. We use a normal distribution with $\mu = 0$ and $\sigma = 100\mu s$ to generate the jitter times. σ was selected according to the jitter we observed for the other use cases. Since jitter adds a random time to an algorithm's execution time, we use the absolute value of jit.

Furthermore, we have to consider data transfers, the major bottleneck of a hybrid CPU/co-processor system. The typical workflow for a co-processor is to transfer the input data from the CPU to the co-processor, process the input data, and transfer the results back to the CPU.

Transferring the input data from the CPU to the co-processor should be avoided with a suitable *data placement strategy*. However, this strategy will never be able to completely avoid data transfers. We come back to this issue in Section 7.2. Therefore, we assign each processing device a *cache hit rate* (CHR). The CHR of a processing device is the probability that the input data is cached in the processors local memory.

In general, results have to be transfered back from a co-processor to the CPU. However, result sizes are often smaller than the input data sizes (e.g., for selections and aggregations). Therefore, we introduce the *average selectivity factor* (ASF) of the simulated operation, which is the ratio of the number of result tuples and the number of input tuples.[7]

The overhead introduced by the data transfers over the bus is also dependent on the operation: compute intensive operations such as rendering tasks can neglect data transfer cost, but data intensive tasks have to carefully consider the data transfer overhead. Therefore, we introduce the *Relative Bus Speed* (RBS), which specifies the ratio of the computation time for a data set D on processing device zero (CPU) and the transfer time for D.

A further limitation of the bus is that data can only be transfered simultaneously in two different directions (e.g., one transfer from CPU to GPU and vice versa). All other transfer requests are serialized by the hardware. We simulate this behavior by two locks, one lock for each transfer direction.

[7] We currently consider only single operations in the simulator.

Experimental Design

Similar to the other use cases, we generate a workload consisting of 10,000 operations and 100 different data sets. A data set consists of a single value indicating the size. An operation gets a data set as input and waits a certain time depending on the data size. Therefore, we can simulate a multiple number of processing devices as the CPU has physical cores. The source code of our simulator is available online as part of HyPE.[8]

Variables

We conduct experiments to identify which parameters have the highest influence on the performance of a hybrid CPU/CP system. We evaluate our best heuristic WTAR for the following variables: (1) number of available processing devices (#PD), (2) the relative speed (RS) of the co-processors compared to the CPU, (3) the average cache hitrate (CHR) of the co-processors, and (4) the average operator selectivity factor (ASF).

Analysis Procedure

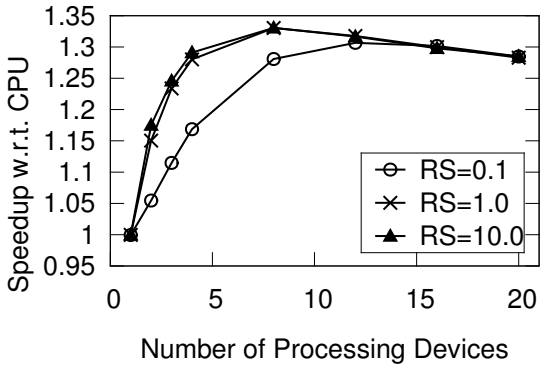
We evaluate our results separately for each experiment. In each experiment, we vary two of our independent variables, whereas the dependent variable is always the speedup of executing the workload on the whole system with respect to executing the workload on a single CPU. We vary the four variables in the following intervals:

1. $\#PD \in \{1, \ldots, 20\}$
2. $RS \in \{\frac{1}{10}\ \frac{1}{9}\ \ldots, 1, 2, \ldots, 10\}$
3. $CHR \in \{0.0, 0.1, \ldots, 1.0\}$
4. $ASF \in \{0.0, 0.1, \ldots, 1.0\}$

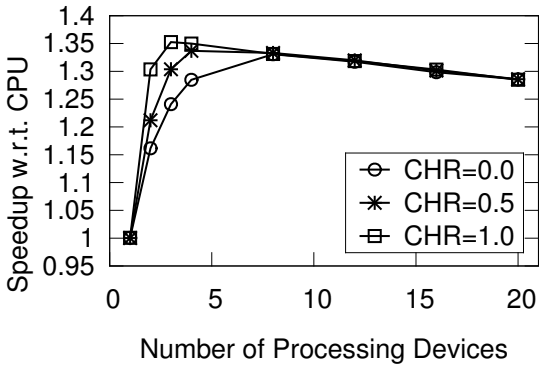
For our base configuration, we assume that a co-processor is roughly two times faster in processing a data set than the CPU (RS=2), the average cache hitrate is about 50% (CHR=0.5), the average selectivity factor is 1.0 (e.g., as for sorts or primary key/foreign key joins) and the number of processing devices is 20 (#PD=20). We conduct four experiments, in which we vary two variables in their specified intervals:

1. We investigate the influence of different relative speeds between the CPU and the CPs. Therefore, we vary relative speed (RS) and number of processing devices (#PD) while keeping the other parameters constant (CHR=0.5 and ASF=1.0).
2. Then, we vary the cache hitrate and number of processing devices and keep operator selectivity and relative speed constant (ASF=1.0 and RS=2).
3. Since the average operator selectivity has a high impact on the performance, we vary the average operator selectivity and number of processing devices while keeping cache hitrate and relative speed constant (CHR=1.0 and RS=2). Note that the cache hitrate is set to 1.0 so it cannot act as confounding variable in this experiment.

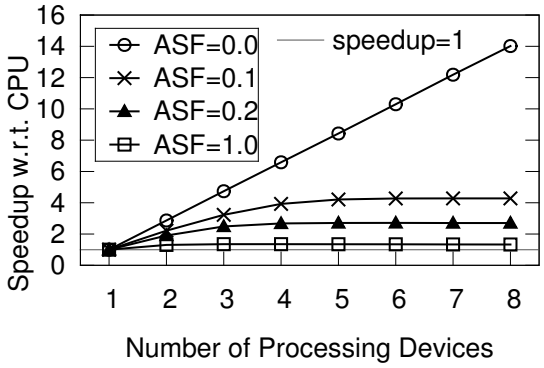
[8] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/supplemental.php



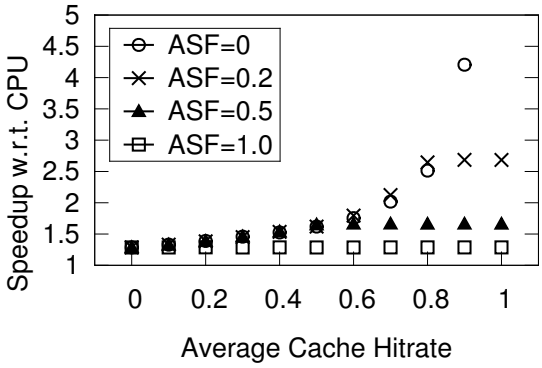
(a) Varying relative speed and #PD.
Constant: CHR=0.5, ASF=1.0



(b) Varying CHR and #PD.
Constant: RS=2, ASF=1.0



(c) Varying ASF and #PD.
Constant: RS=2, CHR=1.0



(d) Varying CHR and ASF.
Constant: RS=2, #PD=20

CHR – Cache Hitrate, ASF – Average Operator Selectivity Factor, RS – Relative Speed

Figure 6.8: Simulator Results for Waiting-Time-Aware Response Time.

4. Finally, to decide which factor has the greatest impact on performance, we vary cache hitrate and average operator selectivity (#PD=20 and RS=2).

We restricted the analysis to WTAR, as it proved to be the best heuristic in the two-device scenario.

6.5.2 Results

Now, we present only the results of the experiments. In Section 6.5.3, we answer the research questions and discuss the achieved speedups.

Varying relative speed and number of processing devices

We illustrate the results in Figure 6.8(a). The relative speed of processing devices has a high impact on the speedup if and only if the co-processors are slower than the Bus and the CPU (i.e., RS=0.1 means that the co-processors are ten times slower than the CPU). In this case, we observe an almost linear grow of the speedup with increasing number

of processing devices until #PD=8, from which the speedup remains constant. Further investigation revealed that the bus was fully utilized starting from eight processing devices. By contrast, when the co-processors are equally fast (RS=1) or faster (RS=10) than the CPU, the bus becomes fully utilized starting by four processing devices. The performance degrades starting by more than eight processing devices.

Varying cache hitrate and number of processing devices

We visualize the results of this experiment in Figure 6.8(b). We observe that the cache hitrate has only a small impact on the speedup when the average operator selectivity is very small. Hence, a large portion of the input data has to be transfered back from the co-processors to the CPU, which slows down the performance. Therefore, we will now investigate how the speedup develops for varying operator selectivity.

Varying average operator selectivity and number of processing devices

We illustrate the results in Figure 6.8(c). We observe that the speedup significantly increases with a higher operator selectivity (and hence, a lower selectivity factor). Note that we set the cache hitrate in this experiment to 100%. We can clearly identify the operator selectivity as one dominating factor for the performance of a hybrid CPU/CP system. However, we now have to identify whether the cache hitrate or the operator selectivity has the highest impact.

Varying cache hitrate and average operator selectivity

We visualize the results of this experiment in Figure 6.8(d). We observe a significant speedup only when the operator selectivity is less than or equal to 0.5 and the cache hitrate is at least 50%. Furthermore, we can observe that the cache hitrate is the dominating performance factor until it exceeds 50%. Then, the operator selectivity becomes the dominating factor. We encounter significantly higher speedups only for a cache hitrate of at least 70% and a operator selectivity factor of less than or equal to 0.2.

6.5.3 Discussion

We now discuss the results of our experiments. For the first experiment, we observed that the higher the relative speed of a co-processor is, the more likely is that the bus becomes the bottleneck of the system. Therefore, the relative speed is only a minor performance factor, because of the data transfer overhead over the PCIe bus. A slow co-processor just ”hides” latency due to data transfers over the bus, but as soon as the bus is fully utilized, no further performance improvement can be achieved.

For the second experiment, we observe that a higher cache hitrate increases the performance of the system. However, the speedup does not grow linearly with the number of processing devices. The reason for this is that the results still have to be copied back from the co-processor to the main memory. In case the operator selectivity factor

is too high, the co-processors will compete for the bus most of the time, which can significantly degrade performance.

For the third and fourth experiment, we observe that the cache hitrate and the operator selectivity are the dominant factors for the performance of a hybrid CPU/CP system. The cache hitrate is the dominating performance factor until it exceeds 50%. Then, the operator selectivity becomes the dominating factor (RQ7). We observe significantly higher speedups only for a cache hitrate of at least 70% and an operator selectivity factor of less than or equal to 0.2 (RQ6).

Consequences for query processing

As hypothesized in Section 3, data transfers of multiple co-processors may render some co-processors irrelevant for performance, because once the bus bandwidth is fully used, no further acceleration is possible. When co-processors spent more time on waiting for bus access than on data processing, we call this *bus trashing*. Based on this discussion, we derive the following possible solutions for bus trashing:

1. Add multiple independent PCIe Express Bus systems in one machine following the tuning principle partitioning breaks bottlenecks [180].
2. Use compression techniques to reduce the data volume (Fang and others [64] and Przymus and others [161]).
3. Execute only operations with high selectivity on the co-processors.

It is unrealistic that point 3 is applicable most of the time. However, queries often consist of sequences of operations that process data of their predecessor and pass their output to their successor. Such *operator chaining* minimizes data transfers. One way to implement operator chaining is to construct bushy query trees and execute each separate path (or sub-tree) in the query tree on another co-processor. An operator chain may be executed on the co-processor if and only if the selectivity of the data passes a certain threshold (e.g., 10%). He and others discussed this for a single CPU/GPU system [85], but the idea is applicable to a more general scope.

6.5.4 Threats to Validity

We now discuss threats to internal and external validity.

Threats to Internal Validity

We carefully calibrated our simulator according to the bandwidth of our test machine's main memory and the PCIe Bus. We measured the main-memory bandwidth of our machine with the linux tool *mbw* and observed an average bandwidth of 23.43 GiB/s. The PCIe Bus has a maximum speed of 8 GiB/s, leading to a relative bus speed of ≈ 0.3413.

Threats to External Validity

We are aware that our simulator cannot capture all architectural details of a hybrid CPU/CP system. However, we have modeled the most important impact factors of such systems for database query processing: relative processing device speed, cache hitrate, and operator selectivity, which we derived from existing work [77, 85] and our experiences with CoGaDB. In our experiments, we always assumed that each simulated co-processor has the same speed. This is a common scenario in practice, where a machine contains GPUs (or parallel accelerator cards such as the Intel Xeon Phi) from the same vendor and product.

6.6 Related Work

We now present related work in the fields of hybrid CPU/GPU query processing, self-tuning databases and heterogeneous task scheduling.

6.6.1 Hybrid CPU/GPU Query Processing

He and others developed GDB, a GPU-accelerated DBMS [85]. In contrast to HyPE, they use an analytical cost model, which needs to be updated for each new generation of GPUs. Furthermore, their model cannot adapt to changing data and workloads.

Malik and others proposed a tailor-made scheduling approach for OLAP in hybrid CPU/GPU environments [128]. They introduced an analytical calibration-based cost model to estimate runtimes on CPUs and GPUs. Since the approach is specific to their implementation, it cannot be easily applied to other DBMSs.

Rauhe and others used just-in-time query compilation for complete OLAP queries to reduce the overhead due to data transfers and synchronization [169]. They achieve speedups up to five by combining multi-threaded execution with SIMD capabilities of GPUs. However, they execute one query either on the CPU or the GPU, while HyPE allows for concurrent processing on all (co-)processors.

Similarly, Yuan and others investigated the performance of OLAP queries on GPUs. They compile SQL queries to a 'driver program', which then executes the query using pre-implemented relational operators [204]. Wu and others proposed *Kernel Weaver*, a compiler framework, which combines GPU kernels of relational operators. Their goal is to reduce the data volume that needs to be transfered over the bus and to exploit code optimizations enabled by the combined kernels [198]. Both approaches perform all processing on the GPU and hence, omit possible performance gains due to inter-device parallelism.

Przymus and others proposed a bi-objective query planner based on marked models [160, 162]. Their framework enables the DBMS to optimize query execution time and one additional goal such as energy consumption.

Zhang and others introduced an alternative optimization heuristic in their system OmniDB, which schedules *work units* on available (co-)processors. For each work unit,

the scheduler chooses the processing device with the highest throughput. To avoid overloading a single processing device, the scheduler ensures that the workload on each processing device may not exceed a predefined fraction of the complete workload in the system [206].

6.6.2 Self-Tuning

Zhang and others developed COMET, an approach for estimating the cost of XML operators using the statistical learning technique *transform regression* [205]. Our approaches have in common that we do not need detailed cost models of the operators but learn them on the fly by observing the correlation between an operator's characteristic features and execution time. The difference is that we focus on allocating co-processors for relational operators whereas Zhang and others focus on cost prediction for XML queries.

Răducanu and others introduced the concept of micro adaptivity [172]. Their approach chooses from a set of algorithm implementations the one with lowest execution cost. In contrast, our approaches distribute operators on a set of processing devices according to the processor's speed. Hence, the approaches are complementary: While we choose a suitable processing device, Răducanu and others select a suitable algorithm implementation.

6.6.3 Heterogeneous Task Scheduling

Kerr and others developed a model, which selects CPU and GPU algorithms statically before runtime [116]. Hence, their approach does not introduce any runtime overhead and can utilize CPU and GPU at runtime for different database operations. The major drawback is that no inter-device parallelism can be achieved for a single operation class, because either every operation in the workload is executed on the CPU or the GPU.

Iverson and others proposed a learning-based approach which requires no hardware specific information similar to our model [104]. However, our used statistical methods and architectures differ.

Augonnet and others introduced StarPU, a heterogeneous scheduling framework that provides a unified execution environment and runtime system [15]. StarPU can distribute parallel tasks in environments with heterogeneous processors such as hybrid CPU/GPU systems and can construct performance models automatically, similar to HyPE.

Ilić and others developed CHPS, an execution environment similar to HyPE and StarPU [102]. CHPS main features are (1) support of a flexible task description mechanism, (2) overlapping of processor computation and data transfers and (3) automatic construction of performance models for tasks. Ilić and others applied CHPS on TPC-H queries Q3 and Q6. They observed significant performance gains, but used tailor-made optimizations for the implementation of the queries [101].

Pienaar and others identify four critical factors a heterogeneous scheduling framework needs to fulfill: Suitability, locality, availability, and criticality [152]. Based on these criteria, they propose *model-driven runtime* (MDR), a run-time system for heterogeneous platforms that runs operator acyclic graphs. Communication overhead is estimated using linear regression and runtime of tasks is estimated via k-nearest neighbor regression. Based on these performance models, MDR assigns tasks to processors.

Acosta and others present a library that performs dynamic load balancing for iterative algorithms [7]. During the first iterations, information about execution times is collected and used in later iterations to balance the load between processors.

Yang and others optimized the Linpack benchmark on a heterogeneous CPU/GPU super computer [202]. They present an adaptive framework that balances the workload over all CPUs and GPUs.

Belviranli and others present an approach that partitions input data on the fly and processes these partitions on different processors [25]. These partitions (or blocks of loop iterations) need to be assigned to different processors, which works in two phases. In the adaption phase, a performance model for each processor is computed. In the completion phase, the remaining workload is assigned to the processors according two the performance model that was determined in the adaption phase. Belviranli and others especially focus on the optimal size of the partitions to minimize load imbalance while maximizing the utilization of each processor.

Boyer and others discuss how we can achieve reliable performance when partitioning work between heterogeneous processors [34]. The idea is to dynamically react to changing performance of processors to avoid execution skew. Therefore, small portions of the input data is send to each processor. Then, the work for each processor is partitioned according to the observed execution times.

Ravi and Agrawal also investigate how the optimal chunk size of on-the-fly data partitioning can be determined [170]. They present a cost model to derive the optimal chunk size for a given heterogeneous system.

Shukla and Bhuyan present an approach that blends GPU memory in the virtual memory of the CPU and automatically transfers data between the memories [182]. This allows for data partitioned allocation in CPU and GPU memory and allows both processors to work on the input data simultaneously.

Kofler and others compile OpenCL programs written for a single processor to an OpenCL program that can run on multiple processors concurrently using the Insieme compiler [118]. Then, the tasks are partitioned such that CPUs and GPUs work together, which accelerates task processing. The task partitioning is done based on a learned prediction model using artificial neural networks and considers static program features as well as properties of the input data.

Choi and others present *Estimated Execution Time* (EET) scheduling, which estimates how long an application will run on a certain processor and puts the task to the processor

with the smallest expected execution time [53]. EET scheduling also considers how long the currently running application will need to complete. Thus, EET scheduling is similar to our heuristic WTAR. The main difference is that EET scheduling uses rough statistics such as the average execution time of previous application runs to predict the execution time, whereas WTAR relies on HyPE's execution time prediction, which uses a history of observations and statistical regression to obtain performance models depending on the input data and operator properties.

Binotto and others contribute a dynamic run-time scheduler similar to HyPE, which collects performance information at run-time and assigns tasks to processors based on execution times that are stored in a history [27].

Song and Dongarra present a framework for solving linear algebra plans that scales for a large number of nodes, where each node contains a CPU and multiple GPUs [184]. The idea is to partition input matrices into tiles and process each tile on a CPU or a GPU on a node in the cluster.

Jiménez and others propose several scheduling algorithms for heterogeneous processor systems [107]. Their performance-history-based scheduling keeps track of execution times of applications on each processor and also considers the completion time of running applications.

Grewe and O'Boyle present an approach for static task partitioning for OpenCL applications [78]. Before the application is run, code features are extracted from OpenCL kernels, which are then used to compute a suitable partitioning.

Garba and González-Vélez propose to organize parallel programs in pipelines, where each pipeline stage is dynamically assigned to processors depending on the processor utilization [66]. The goal is to use processors that are currently idle to improve the performance.

Wang and others present co-scheduling based on asymptotic profiling, which is a run-time scheduling strategy that minimizes the synchronization overhead between CPUs and GPUs compared to periodical load balancing operations [196].

Lee and others propose *Cooperative Heterogeneous Computing* (CHC) [125], which allows CUDA kernels to also run on CPUs by compiling PTX code to LLVM IR [56]. Based on this capability, CHC executes CUDA applications on CPU and GPU concurrently by assigning different thread blocks to CPU and GPU.

Gregg and others present a dynamic scheduling approach for applications that can either use the CPU or the GPU [76]. By performing application scheduling on a global scope, the overall throughput and application response times can be improved. This is achieved by using a history of performance measurements for individual applications to estimate how long an application will run using linear regression.

Albayrak and others present an profiling-based approach for kernel mapping to CPUs and GPUs [11]. Kernel characteristics are identified using offline profiling and a greedy

strategy selects a processor depending on the expected execution time and data dependencies between kernels.

Shirahata and others discuss how map tasks can be scheduled on CPUs and GPUs in mapreduce frameworks to reduce the execution time of mapreduce jobs [181]. The scheduling is performed based on observed execution times of map tasks on CPU and GPU.

A major problem of existing approaches is the high integration effort for DBMS and the fact that the optimizer needs to use the task abstractions of the scheduling frameworks (e.g., CHPS and StarPU). Since optimizers of existing DBMS are extremely complex, an approach is needed that allows for minimal invasive integration in the optimizer, while enabling the optimizer for efficient co-processing. We developed HyPE to close this gap.

6.7 Conclusion

Efficient co-processing is an open challenge yet to overcome in database systems. In this chapter, we extended our hybrid query processing engine by the capability to handle operator streams and to exploit inter-device parallelism. Furthermore, we discussed optimization heuristics for response time and throughput.

We validated our extensions on five use cases, namely aggregations, column scans, sorts, joins and simulations. Hence, we showed that our approach works with the most important primitives in column-oriented DBMS. We achieved speedups up to 1.85 compared to our previous solution and static scheduling approaches while delivering accurate performance estimations for CPU and GPU operators without any a priori information on the deployment environment.

Furthermore, we investigated the scaling behavior of database query processing for an increasing number of co-processors and found that we can achieve significant speedups with multiple co-processors in case the data is cached in at least 50% of the cases and the operator selectivity is equal or below 20%. In Chapter 7, we develop a query processing approach called query chopping that serializes a set of queries to an operator stream and compare it's performance with traditional single-query optimization.

7. Robust Query Processing

In the previous chapters, we presented a framework for hardware-oblivious operator placement and showed how we can balance workloads between heterogeneous processors. The potential of heterogeneous systems is often limited by the capacity of the communication channel between the co-processor and its host system [77]. Although we always included data transfer times in our measurements, we find in this chapter that we require a deeper understanding of the effects caused by the communication channel on analytical database workloads. We illustrate this problem in Figure 7.1 for a GPU-based co-processor, which we use as a poster child in this chapter.

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 [85]), 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 chapter 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-occurring or ad-hoc queries accessing the data cached on the co-processor).

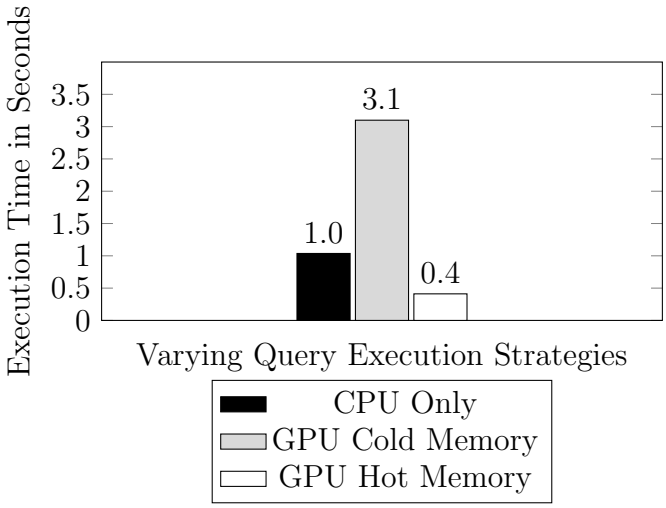


Figure 7.1: Impact of different query execution strategies on performance of a star schema benchmark query (Q3.3) on a database of scale factor 20. Using a GPU slows the system down in case input data is not cached on the GPU.

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.

To validate the effect of the strategies, we provide a detailed experimental evaluation based on our open-source database engine CoGaDB.

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

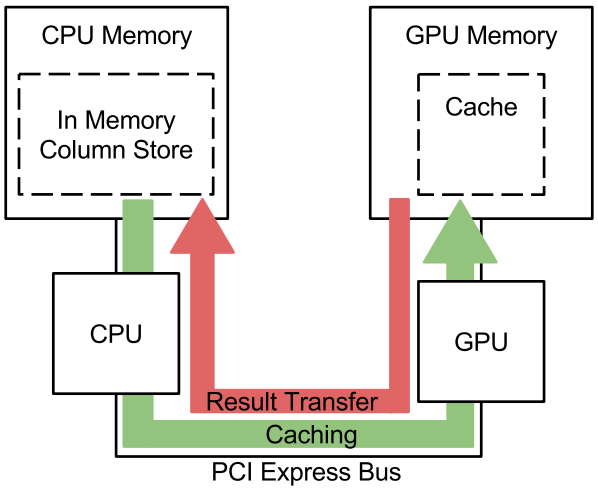


Figure 7.2: Caching mechanism for co-processors. Note the reserved memory that is used as cache.

7.1 State of the Art

In this section, we recapitulate the basics of co-processors and provide a brief overview of their resource limitations. Then, we discuss how other systems handle the operator placement problem. Finally, we repeat important details on CoGaDB.

7.1.1 Data Placement on 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 [174]. 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 [77].

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. We illustrate this in Figure 7.2.

7.1.2 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 cleanup.

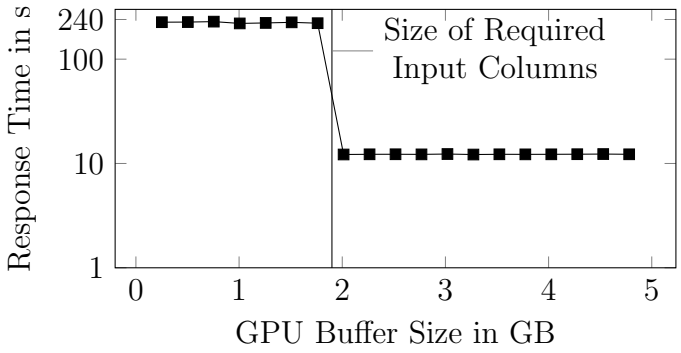


Figure 7.3: Execution time of selection workload. The performance degrades by a factor of 24 if not all input columns fit into the cache.

To be practically useful, database engines must have a mechanism to deal with such fault situations. Previous work [193] 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.

7.1.3 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 rather than the exception.

Cache Thrashing Figure 7.3 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 A.4.1. All required input columns have a total size of 1.9 GB. In the case that not all input columns fit into the buffer, 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 A.4.1), 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, but we increase the number of parallel user sessions that execute the workload. We discuss the detailed workload in Appendix A.4.2.

Since all workloads contain the same amount of 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. Thus, we expect no change in workload execution time. Figure 7.4 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.

Aborted operators need to be processed on another processor, 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.

7.1.4 Operator Placement

All co-processor-accelerated database systems either process all queries on the co-processor (e.g., GPUDB [204], MultiQx-GPU [193], Red Fox [199] and Virginian [17]), or try to balance the processing tasks between processors (e.g., CoGaDB, GPUQP [85], MapD [138]) 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.

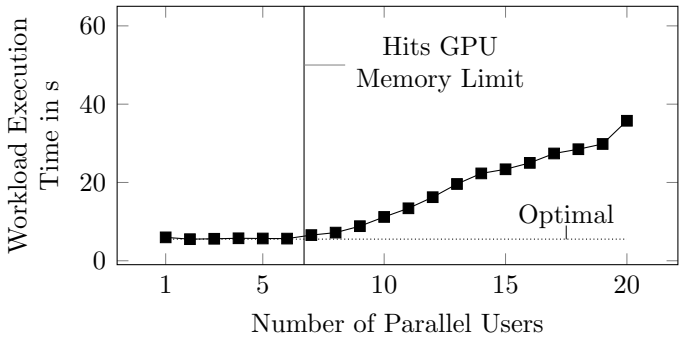


Figure 7.4: Execution time of a selection workload. With increasing parallelism, more operators allocate memory on the GPU, which leads to performance degradation when memory capacity is exceeded.

7.1.5 Evaluation System: CoGaDB

As we discussed in Chapter 3, CoGaDB is a main memory database system using a column-oriented data layout and parallel operators on CPU and GPU to efficiently process analytical workloads. Similar to other OLAP DBMSs such as MonetDB [100], CoGaDB keeps the database in-memory and uses operator-at-a-time bulk processing. Thus, it employs data-parallelism within operators and inter-operator parallelism by evaluating child operators in parallel. However, CoGaDB does not employ pipelining. CoGaDB can cope well with allocation failures caused by the memory scarcity of co-processors. We compare the performance of CoGaDB to the GPU-accelerated database engine MonetDB/Ocelot in Section A.3.

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 an earlier system, which handled out-of-memory situations by aborting entire queries [193]. 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.

Operator Placement

He and others early recognized the importance of operator placement and data movement [85]. To address the problem, they performed backtracking for sub-query plans and combined optimal sub plans to the final query plan [85]. From our experience in CoGaDB, this backtracking approach can be very time consuming. Thus, based on our findings in Chapter 5, CoGaDB uses an iterative refinement optimizer that only considers query plans where multiple successive operators are executed on the same processor,

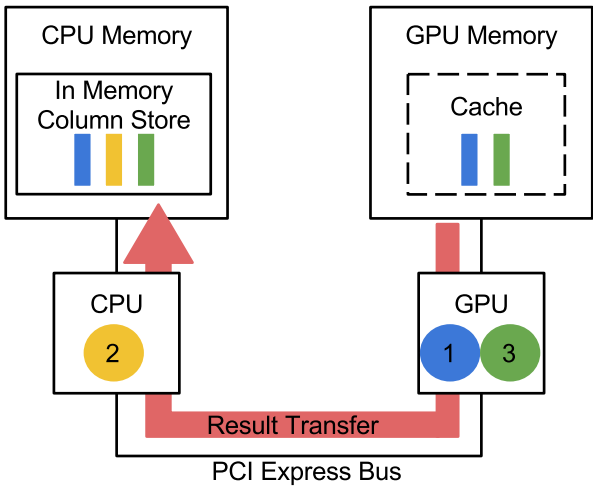


Figure 7.5: Principle of Data-Driven operator placement. Operators 1 and 3 are pushed to the cached data on the co-processor, whereas the data of operator 2 is not cached and must run on a CPU.

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 chapter.

Performance Optimizations for Data Transfer

CoGaDB implements the recommended optimizations to reduce data transfer bottlenecks [3]. 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.

7.2 Data-Driven Operator Placement

All of the heuristics in Section 7.1.5 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-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. Additionally, data placement can be optimized at a workload level which in turn helps to minimize I/O.

7.2.1 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. We illustrate *Data-Driven* in Figure 7.5. 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 [150], 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.

7.2.2 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. For this, the storage manager keeps statistics about how frequently and how recently access structures were used by the query processor. Our implementation places the most frequently used access structures on the co-processor until the buffer space is exceeded. This data placement process is the only component that may change the data placement to avoid cache thrashing and runs periodically (e.g., every ten seconds) in the background to support dynamically changing workloads. 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.

7.2.3 Query Processing

A consequence of *Data-Driven* is that operators are automatically chained from the leaf operators, until an n-ary operator ($n > 1$) 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 7.6 and 7.8, respectively. *Data-Driven* eliminates the performance degradation, which we observed for the classic approach: operator-driven data placement.

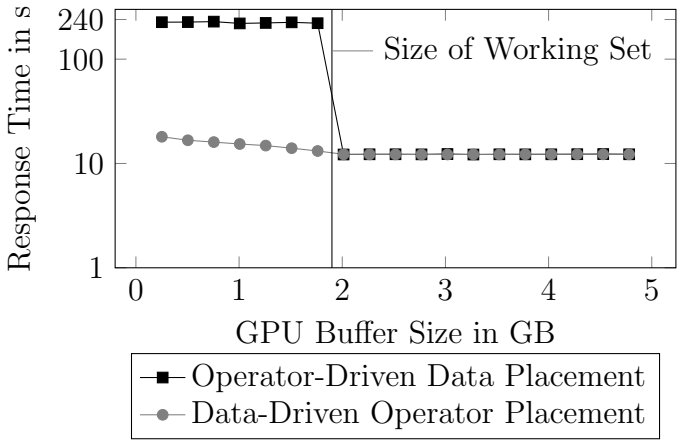


Figure 7.6: Execution time of selection workload. The performance degradation can be avoided by data-driven operator placement.

We show in Figure 7.7 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.

7.2.4 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 7.8. Deeper investigation revealed that 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 [85], 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 7.8.

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 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.

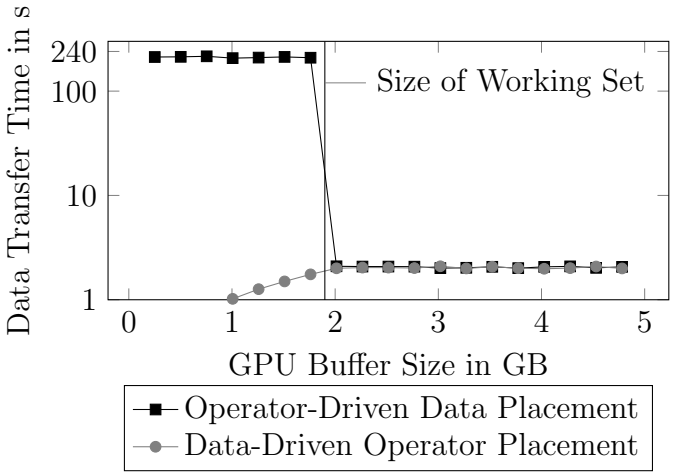


Figure 7.7: Time spent on data transfers in the selection workload.

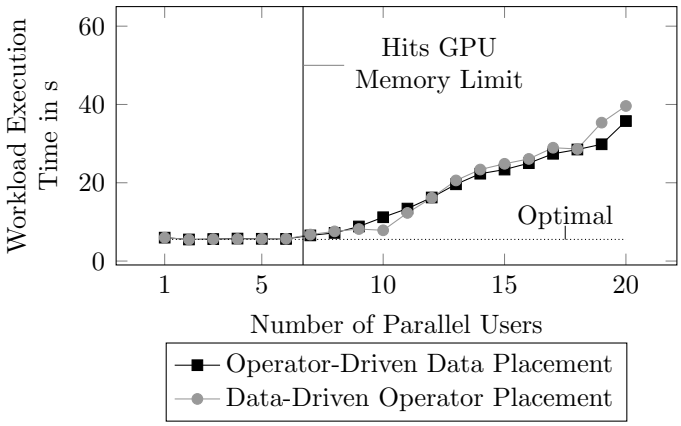


Figure 7.8: Execution time of a selection workload. *Data-Driven* has the same performance degradation as operator-driven data placement.

7.3 Run-time operator placement

Until now, we discussed heuristics for the operator placement problem applied at query *compile-time*. They all have in common that 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. If we want to react to these kinds of events, the optimizer needs to place operators at run-time.
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.

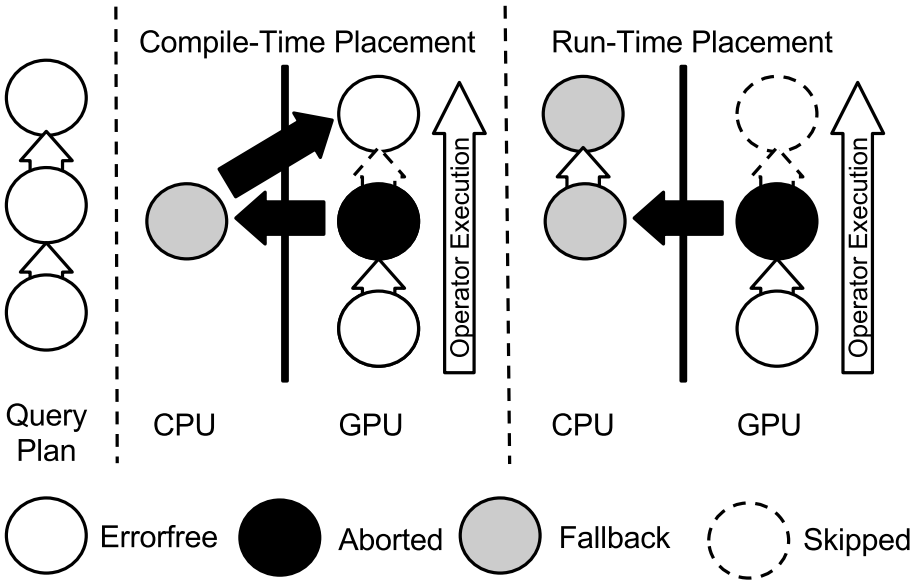


Figure 7.9: Flexibility of run-time placement. Compile-time heuristics force the query processor to switch back to the GPU after an operator aborts, whereas run-time heuristics avoid this overhead.

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 [29]. By separating query optimization into strategical and tactical decisions, many important runtime parameters can be considered for the optimization process. While the database optimizer still 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).[1] 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.

7.3.1 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 discards the work of the operator and restarts the operator on the CPU. This mechanism is very efficient, because operators typically

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

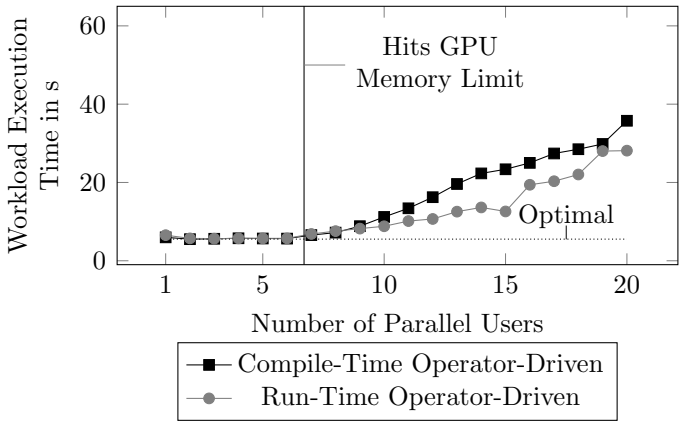


Figure 7.10: Run-time operator placement improves performance, but does not achieve the optimum.

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 7.9. 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.

7.3.2 Query Processing

We illustrate in Figure 7.10 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.

7.4 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 maximal number of operators that can run in parallel by using the thread pool pattern.

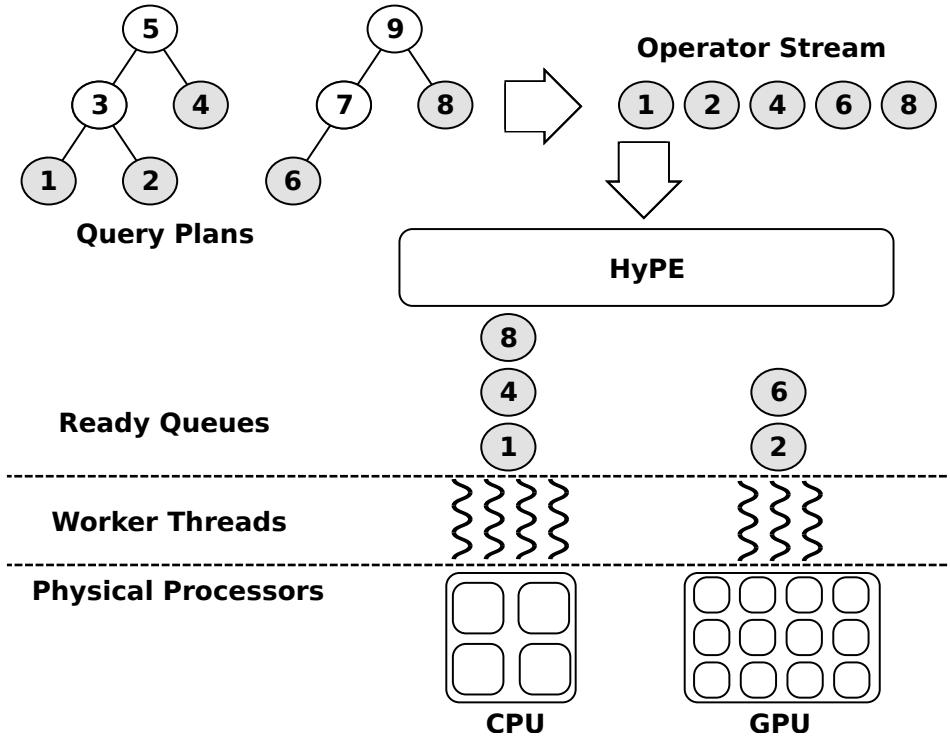


Figure 7.11: Query Chopping.

7.4.1 Probability of Operator Aborts

In a robust query processor, it is essential to dynamically react to faults. However, it is even better if the DBMS could avoid faults in the first place. In our case, we want to reduce the probability that an operator runs out of memory. We can achieve this by prohibiting the DBMS to execute multiple GPU operators concurrently (e.g., He and others [85]). However, Wang and others showed that we can improve the performance of query processing by allowing moderate parallel execution [193]. 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.

7.4.2 Query Chopping

Based on our discussions, we now present our novel technique *query chopping* (*Chopping*), which builts on our results from Chaper 6. In essence, *Chopping* is a progressive

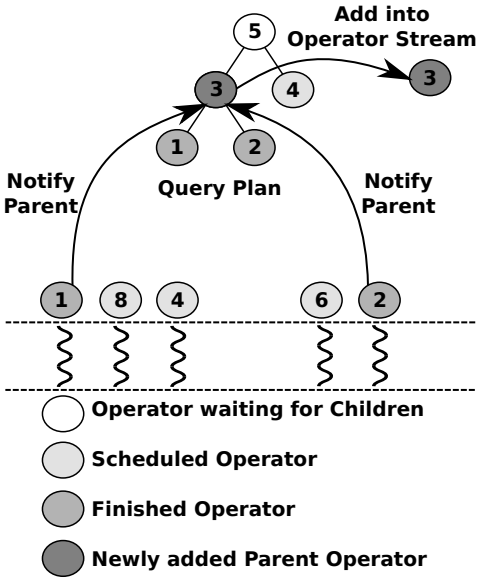


Figure 7.12: Query Chopping: Finished operators pull their parents into the global operator stream.

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 a DBMS 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 using our heuristic waiting-time-aware response time as discussed in Section 6.3, and selects for each operator a suitable algorithm. We summarize our discussion in Figure 7.11.

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. After the root operator of a query plan finished execution, the query result is returned. We illustrate this procedure in Figure 7.12.

This technique works for single-query and multi-query workloads. The beauty 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

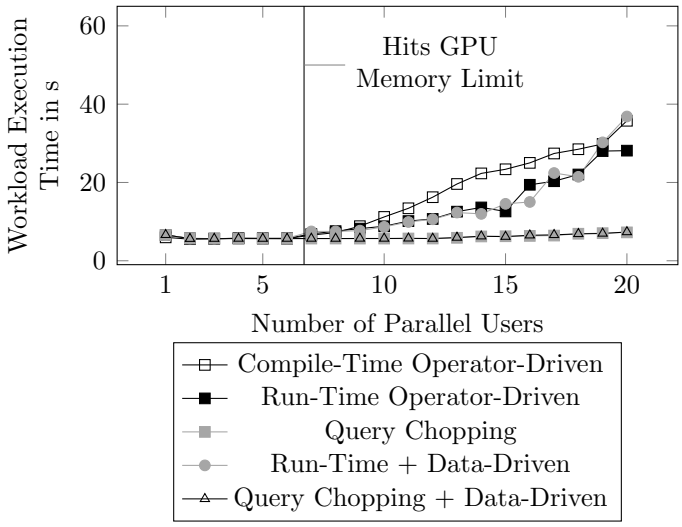


Figure 7.13: Dynamic reaction to faults and limiting the number of parallel running GPU operators achieves near optimal performance.

processor, we maintain a *ready queue*, where all worker sets 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.

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.

7.4.3 Query Processing

We illustrate in Figure 7.13 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 7.14. 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.

7.4.4 Data-Driven Query Chopping

We now discuss how *Data-Driven* and *Chopping* work together. The combined strategy places frequently used access structures in the co-processor's data cache using a periodic

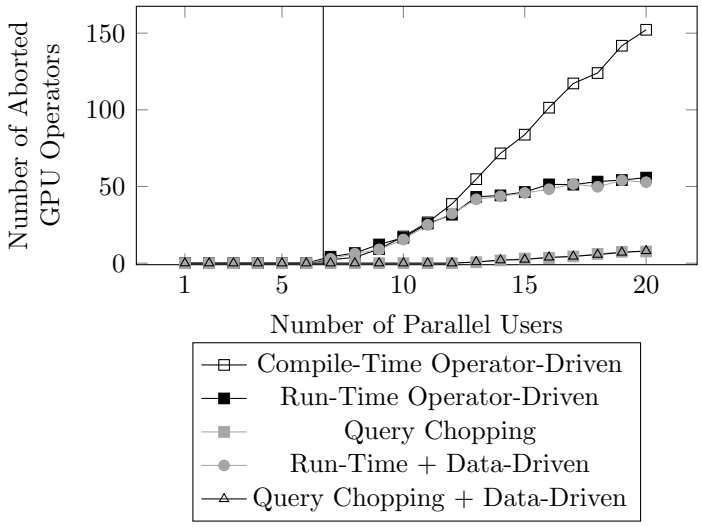


Figure 7.14: Run-time placement reduces heap contention by continuing execution in the CPU. *Chopping* limits the number of parallel running GPU operators and further decreases the abort probability.

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 it's 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 is no longer dormant on the co-processor. Thus, the combined strategy avoids memory thrashing and heap contention.

7.5 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.

7.5.1 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 A.5. 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 the comparison of two state-of-the-art heuristics, *Data-Driven*, *Chopping*, and *Data-Driven Chopping*.

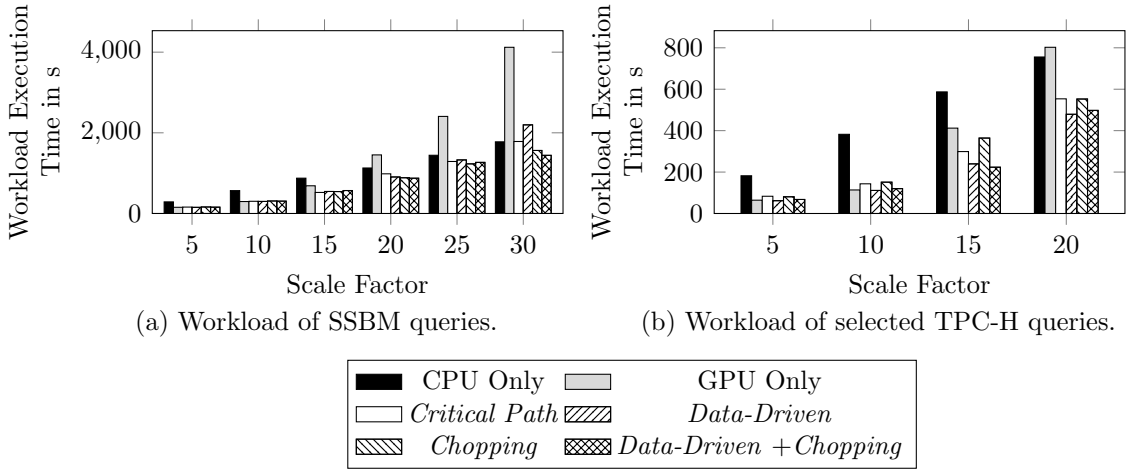
7.5.2 Detailed Experiments

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 A.6. 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*.

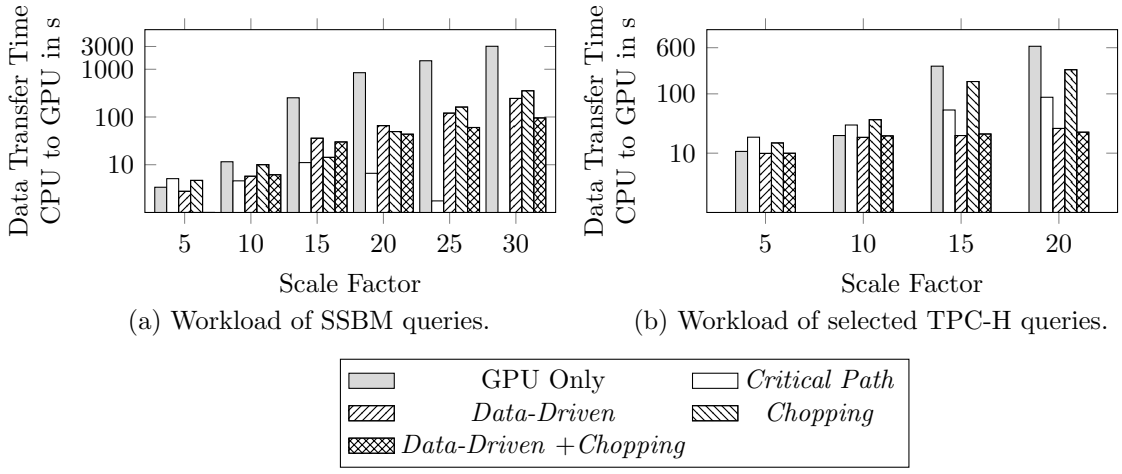
Scaling Database Size

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 7.15 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 transfers from CPU to GPU as illustrated by Figure 7.16. The performance of the GPU-only approach falls behind starting at scale factor 15 (SSBM and TPC-H). Figure 7.17 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.



(a) Workload of SSBM queries. (b) Workload of selected TPC-H queries.

Figure 7.15: 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.



(a) Workload of SSBM queries. (b) Workload of selected TPC-H queries.

Figure 7.16: Average data transfer time CPU to GPU of SSBM and selected TPC-H Queries. *Data-Driven* combined with *Chopping* saves the most IO.

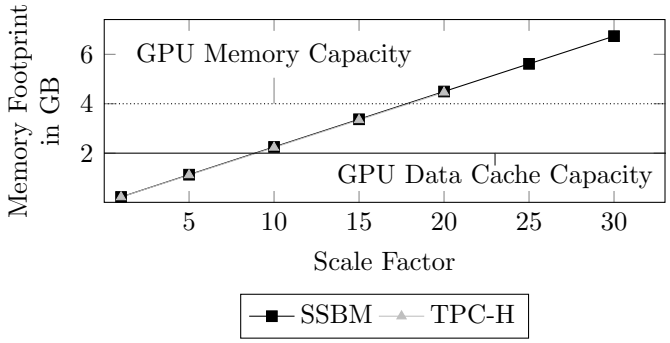


Figure 7.17: Memory footprint of workloads

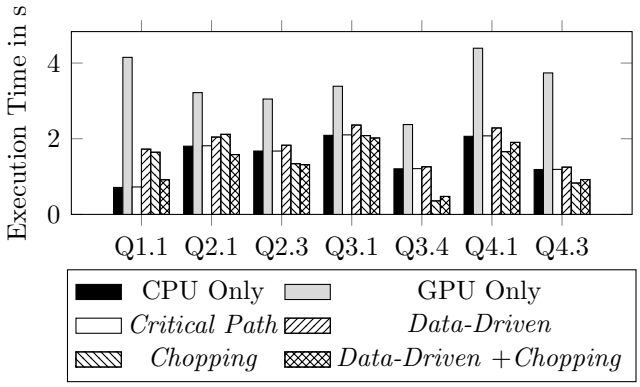


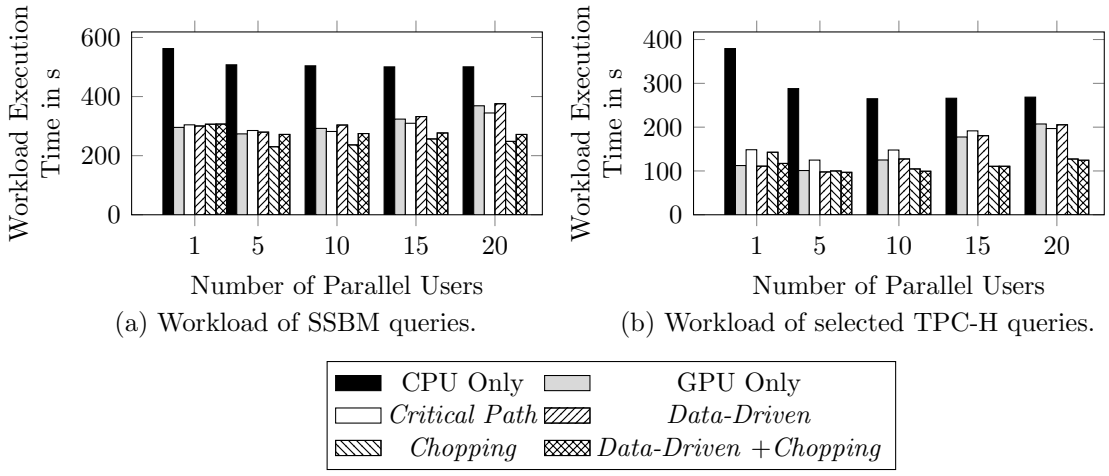
Figure 7.18: Query execution times for selected SSBM queries for a single user and a database of scale factor 30.

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 7.16 (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.

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 7.18. 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.



(a) Workload of SSBM queries.

(b) Workload of selected TPC-H queries.

Figure 7.19: 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.

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 7.15) and minimal IO (Figure 7.16).

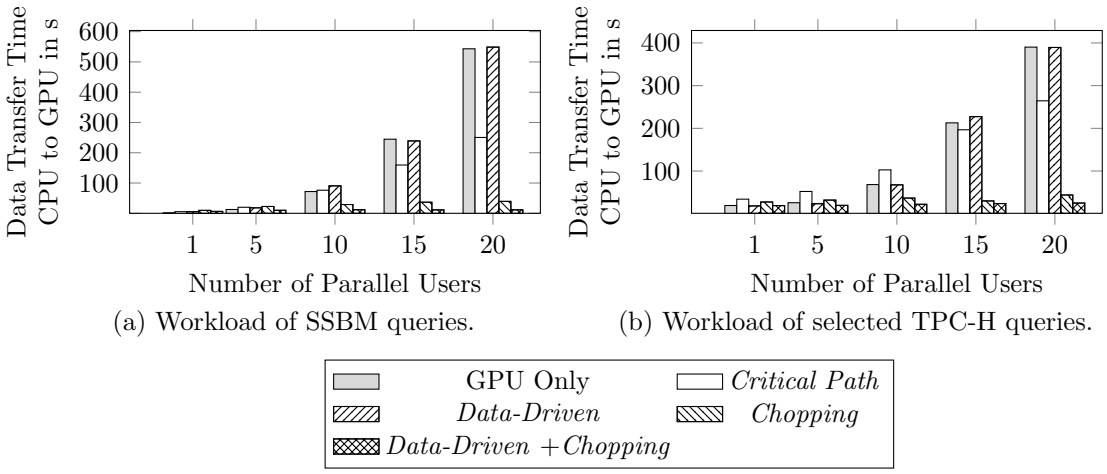
Scaling User Parallelism

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 7.19. 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.

Chopping and *Data-Driven Chopping* reduce the required IO significantly—especially for workloads with many parallel users—as we show in Figure 7.20. *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*



(a) Workload of SSBM queries.

(b) Workload of selected TPC-H queries.

Figure 7.20: 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.

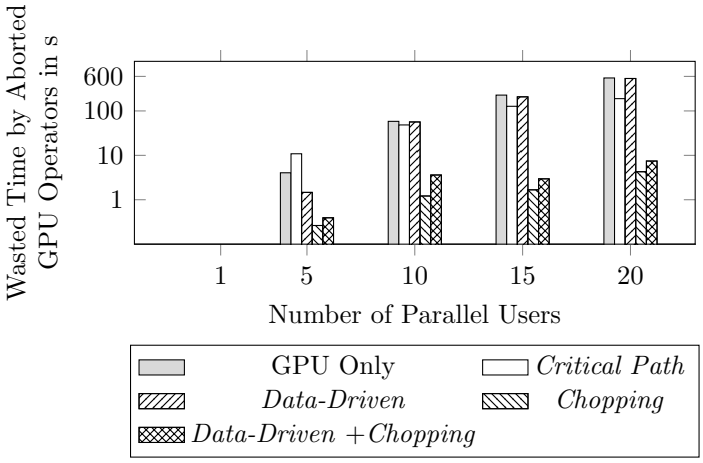


Figure 7.21: Wasted time by aborted GPU operators depending on the number of parallel users for the SSBM. With an increasing number of users, the wasted time increases significantly because of heap contention.

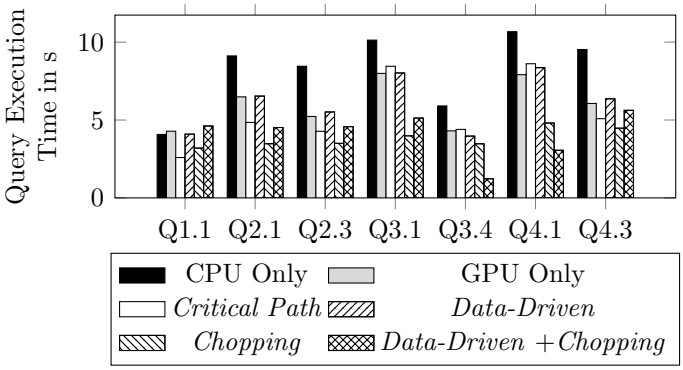


Figure 7.22: Query execution times for selected SSBM queries for 20 users and a database of scale factor 10.

time and illustrate it for the SSBM experiments in Figure 7.21. *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 7.22). *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). 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 [193]. Our solution *Data-Driven Chopping* limits the use of the GPU to the degree where it is beneficial and thus, avoids heap contention.

7.5.3 Results

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.

Our results have several implications. We can use co-processors only for a part of the workload. However, it is common to use multiple GPUs in a single machine, which can handle larger databases and more parallel users. Thus, 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.

7.6 Related Work

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

7.6.1 Co-Processor-accelerated DBMSs

He and others developed the first GPU-accelerated database engine, namely GPUQP [85]. 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 [88]. 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 [206], a database engine that targets heterogeneous processor environments and focuses on hardware obliviousness, similar to Ocelot [95]. 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 [157], where data is lossily compressed using the bitwise decomposition technique [158]. 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 [113], which uses a combination of analytical and learning-based cost models to predict the performance of operators, similar to He and others [85] and Yuan and others [204]. 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 [110]. 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 [204], MultiQx-GPU [193], Red Fox [199] and Virginian [17] 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 [95]. For operator placement, Ocelot makes use of the HyPE optimizer [42].

Aside from GPUs, there are other co-processors to accelerate database query processing, such as MICs [106, 127, 155], Cell Processors [97], and FPGAs [140].

7.6.2 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 [81]. 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.

Psaroudakis and others propose to decompose database operators into tasks, which can be efficiently executed on multi-socket, multi-core machines [164]. 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 [163].

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) [126]. Parallelism is achieved by processing different morsels in parallel by the same operator pipeline. These pipelines are created by just-in-time query compilation [143]. 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 [141].

Wang and others investigated concurrent query processing on GPUs in their system MultiQx-GPU [193]. 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 [95] or MultiQx-GPU [193].

7.7 Summary

In this chapter, 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.

The cache thrashing problem occurs in case the working set of queries does not fit in the co-processor's memory. 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. The performance degrades if the accumulated memory footprint of all operators on the co-processor exceeds the memory. 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.

Although we conducted our evaluation on GPUs only, we expect the same observations for other co-processors, such as the Intel Xeon Phi, or any accelerator card with dedicated memory that is connected to the CPU via a similar interconnect as the PCIe bus. Since we integrated our heuristics in the open source database engine CoGaDB, which has an extensible backend for co-processors, it is straightforward to apply our techniques to other co-processors.

8. Wrap-Up

In this chapter, we summarize the contributions of our thesis, discuss the results and provide an overview of possible future work.

8.1 Summary

The performance of modern processors is no longer bound primarily by transistor density but by a fixed energy budget, the *power wall* [33]. Whereas CPUs often spend additional chip space on more cache capacity, other processors spend most of their chip space on light-weight cores, which omit heavy control logic and are thus, more energy efficient. Therefore, future machines will likely consist of a set of heterogeneous processors, having CPUs and specialized co-processors such as GPUs, Multiple Integrated Cores (MICs), or FPGAs. Hence, the question of using co-processors in databases is not *why* but *how* we can do this most efficiently.

This thesis developed techniques that help to fulfill the three requirements of a database engine for heterogeneous processor environments:

1. **Hardware Obliviousness:** The DBMS needs to work in environments where no detailed knowledge about processors is available.

2. **Algorithm-Speed and Processor-Load Awareness:** The DBMS needs to perform operator placement and therefore needs to consider the processing speed of algorithms on all (co-)processors and the load on each (co-)processor.

3. **Robustness:** DBMSs inevitably run into workloads that violate typical assumptions of co-processor-accelerated database techniques. The DBMS still needs to achieve robust and stable performance in single-user and multi-user workloads.

8.1.1 Survey Of GPU-accelerated Database Systems

The pioneer of modern co-processors is the GPU, and many prototypes of GPU-accelerated DBMSs have emerged over the past years implementing new co-processing approaches and proposing new system architectures. We argue that we need to take into account tomorrows hardware in today's design decisions. Therefore, we theoretically explored the design space of GPU-aware database systems. In summary, we argue that a co-processor-accelerated DBMS should be an in-memory, column-oriented DBMS using the block-at-a-time processing model, possibly extended by a just-in-time-compilation component. The system should have a query optimizer that is aware of co-processors and data-locality, and is able to distribute a workload across all available (co-)processors.

We validated these findings by surveying the implementation details of eight existing prototypes of co-processor-accelerated DBMSs and by classifying them along the mentioned dimensions. Additionally, we summarized common optimizations implemented in co-processor-accelerated DBMSs and inferred a reference architecture for co-processor-accelerated DBMSs, which may act as a starting point in integrating GPU-acceleration in popular main-memory DBMSs. Finally, we identified potential *open challenges* for further development of co-processor-accelerated DBMSs.

Our results are not limited to GPUs, but should also be applicable to other co-processors. The existing techniques can be applied to virtually all massively parallel processors having dedicated high-bandwidth memory with limited storage capacity.

8.1.2 CoGaDB

We presented CoGaDB, a system designed to target the hardware heterogeneity on the query optimizer level. We outlined design decisions and implementation details of CoGaDB and discussed how we combine a modern, GPU-accelerated DBMS with a hardware-oblivious query optimizer.

Our evaluation shows that the design, where an optimizer has no detailed knowledge of the hardware, is feasible. Furthermore, we showed that such a system can be competitive to highly optimized main-memory databases such as MonetDB.

8.1.3 Hardware Oblivious Operator Placement

We presented an adaptive framework for hardware-oblivious operator placement to support cost-based operator placement decisions for heterogeneous processor environments, where detailed information on involved processing units is not available. In the considered use cases, we investigated the performance of operations either on CPUs or on GPUs. Our approach refines cost functions by using linear regression after comparing actual measurements with estimates based on previous ones. The resulting functions were used as input for cost models to improve the scheduling of standard database operations such as sorting and scans. The evaluation results show that our approach achieves near optimal decisions and quickly adapts to workloads.

8.1.4 Load-aware Inter-Processor Parallelism

Efficient co-processing is an open challenge yet to overcome in database systems. We extended our hybrid query processing engine by the capability to handle operator streams and optimization heuristics for response time and throughput. We validated our extensions on five use cases, namely aggregations, column scans, sorts, joins and simulations. Hence, we showed that our approach works with the most important primitives in column-oriented DBMS. We achieved speedups up to 1.85 compared to our previous solution and static scheduling approaches while delivering accurate performance estimations for CPU and GPU operators without any a priori information on the deployment environment.

Furthermore, we investigated the scaling behavior of our approach and found that we can achieve significant speedups with multiple co-processors in case the data is cached in at least 50% of the cases and the operator selectivity is equal or below 20%.

8.1.5 Robust Query Processing

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 GPUs that can lead to poor performance: Cache thrashing and heap contention.

The cache thrashing problem occurs in case the working set of queries does not fit in the co-processors memory. We showed that placing operators on co-processors where their input is cached, and process the remaining processors 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. The performance degrades if the accumulated memory footprint of all operators on the co-processor exceed the memory. 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.

Although we conducted our evaluation on GPUs only, we expect the same observations for other co-processors, such as the Intel Xeon Phi, or any accelerator card with dedicated memory that is connected to the CPU via a similar interconnect as the PCIe bus. Since we integrated our heuristics in the open source DBMS CoGaDB, which has an extensible backend for co-processors, it is straightforward to apply our techniques to other co-processors.

8.2 Discussion

Heterogeneous processor systems come in two basic configurations: Either we have processors with dedicated memories, which need to communicate with each other over an interconnection bus (configuration 1), or all processors share the same memory (configuration 2).

Database systems that run on machines of configuration 1 are always limited by the scarce dedicated co-processor memory (e.g., CPUs combined with dedicated accelerators such as GPUs or MICs). Here, it is crucial to perform a data-driven operator placement combined with an optimizer that performs such decisions at run-time, so it can dynamically react to faults. In configuration 1, operator placement decisions are mostly performed according to the data, the only situations where run-time-based operator placement is needed, is when multiple co-processors have the required input data of an operator cached.

Since processors share their main memory in machines of configuration 2 (e.g., APUs), the interconnect bottleneck disappears. Here, the DBMS is free to place each operator to the most suited processor. In configuration 2, our hardware oblivious query optimizer and our load balancing strategies can be applied to optimize performance. The run-time adaption is especially useful for FPGAs, where the hardware itself can change even at run-time by partial reconfiguration [22].

In summary, we investigated the scalability of database engines on configuration 1 machines regarding the number of heterogeneous processors in the machine, the database size, and the number of parallel users. We found that data processing runs always in the same two limitations: First, the scarce memory capacity of co-processor and second, the interconnection bus. Based on these insights, we developed approaches for efficient and scalable query processing in database systems on heterogeneous processor systems in common hardware setups. In practice, we expect a combination of configuration 1 and 2 machines, where multiple APUs have access to the same main memory with multiple, possible different dedicated co-processors connected to them. However, we have no access to this kind of hardware and hence, could not conduct experiments with our approaches.

We evaluated HyPE in the context of relational database systems only but there is no principal limitation that prevents its use in other domains that face heterogeneous processing hardware such as distributed systems or high-performance computing.

8.3 Future Work

In this section, we identify *open challenges* for GPU-accelerated DBMSs. We differentiate between two major classes of challenges, namely the IO bottleneck, which includes disk IO as well as data transfers between CPU and GPU, and query processing.

8.3.1 IO Bottleneck

In a co-processor-accelerated DBMS, there are two major IO bottlenecks. The first is the classical disk IO, and the second bottleneck is the PCIe bus. We can reduce the latter bottleneck by stream-based GPU algorithms or by keeping data in pinned host memory to reduce copy latencies.

Disk-IO Bottleneck: GPU-accelerated operators are of little use for disk-based database systems, where most time is spent on disk I/O. Since the GPU improves performance only once the data is in main memory, time savings will be small compared to the total query runtime. Furthermore, disk-resident databases are typically very large, making it harder to find an optimal data placement strategy. However, database systems can benefit from GPUs even in scenarios where not the complete database fits into main memory. As long as the *hot data* fits into main memory, GPUs can accelerate data processing.

Stream-based GPU Algorithms: Databases are typically orders of magnitude larger than the dedicated memory capacity of GPUs. In such cases, caching the input data is nearly impossible, and speed ups in computation are quickly consumed by the overhead of data transfers. However, if the DBMS does not use the GPU, it leaves processing resources unused. We expect that this contradiction can be resolved by performing the same operation together by the CPU and other heterogeneous processors in the machine. This would allow to accumulate the compute power and could lead to additional speedups. We can achieve this by using GPUs and other co-processors as a stream processor where data is pulled towards each processor in the machine. Faster processors could take over input data of slower processors, which would ensure that query processing is always accelerated, or at least cannot be slowed down. One promising way to implement such a streamed execution is the vectorized query processing model of Boncz and others [31]. Alternatively, the DBMS could use query compilation.

Reducing the PCIe Bus Bottleneck: Data transfers can be significantly accelerated by keeping data in pinned host memory. However, the amount of available pinned memory is much more limited compared to the amount of available virtual memory. Therefore, a co-processor-accelerated DBMS has to decide which data to keep in pinned memory. Since data is typically cached in GPU memory, a co-processor-accelerated DBMS needs a multi-level caching technique, which is yet to be found.

8.3.2 Query Processing

In co-processor-accelerated DBMSs, query processing and optimization have to cope with new challenges. We identify as major open challenges insufficient support for using multi-processing devices for query-compilation approaches and accelerating different workload types.

Query Compilation for Multiple Devices: With the upcoming trend of query compilation, the basic problem of processing-device allocation remains the same as in traditional query optimization. However, as of now, the available compilation approaches only compile complete queries for either the CPU [121, 143] or the GPU [198, 204]. It is still an open challenge how to compile queries to code that uses more than one processing device concurrently.

Considering different Workload Types: OLTP as well as OLAP workloads can be significantly accelerated using GPUs. Furthermore, it became common to have a mix of both workload types in a single system. It remains open, which workload types are more suited for which processing-device type and how to efficiently schedule OLTP and OLAP queries on the CPU and the GPU.

8.3.3 Extension Points in CoGaDB

In this section, we describe future developments on CoGaDB: efficient algorithms, support of the CUBE operator and other co-processors, and alternative query processing and cardinality estimation approaches.

Efficient Algorithms Although we invested much time in tuning CoGaDB's database algorithms on the CPU and the GPU, the primary focus was still in exploiting the heterogeneous nature of the modern hardware landscape. However, for future work, we will adapt approaches for efficient joins [19, 20] and aggregations [203]. Here, we have two implementation choices: Using hardware-oblivious operators written in a processor-independent language (e.g., OpenCL [95]) or tailoring algorithms for every processor type [47–49].

CUBE Operator The CUBE operator [75] is a compute-intensive operator, which is frequently used in OLAP scenarios. Hence, it would be beneficial to investigate the potential performance gains by offloading parts of the computation to GPUs.

Support for other Co-Processors Aside GPUs, other architectures have emerged for co-processors such as *Multiple Integrated Cores* MICs (e.g., Intel Xeon Phi). It would be interesting to investigate the performance properties of MICs for DBMSs to identify the optimal (co-)processor for a certain task or workload.

Query Processing Strategies Aside from tuple-at-a-time volcano-style and operator-at-a-time bulk processing, there are alternative query processing strategies such as query compilation [143] or vectorized execution [31]. It is not yet clear which strategy is optimal for heterogeneous processor environments. For a fair comparison, all strategies should be implemented in a single system.

Cardinality Estimation Our query optimizer relies on accurate cardinality estimates, which still poses major problems. Markl and others developed progressive optimization, a technique where checkpoints are inserted in the query plan [133]. In

case the cardinality estimates are too inaccurate at a checkpoint, a re-optimization is triggered. Stillger and others proposed LEO, DB2's learning optimizer, which continuously monitors cardinality estimations and iteratively corrects statistics and cardinality estimations [185]. Heimel and Markl offloaded selectivity estimation to the GPU, which allows them to use more compute-intensive approaches to increase estimation accuracy [94].

Query Optimization Even with accurate cardinality estimations, the optimizer is faced with a large space of potential query plans. For these scenarios, it would be worthwhile to investigate whether we can accelerate the query optimization itself and use the additional time to produce better query plans, as proposed by Meister [134, 135].

Big Data Use Cases With additional use cases than the standard OLAP benchmarks it is possible to evaluate the discussed techniques in real applications. For example, Dorok developed a database schema and user-defined functions to perform genome analysis tasks (i.e., variant calling) in a main-memory DBMS [58–61].

A. Appendix

A.1 Overview of Statistical Learning Methods

In this section, we provide a brief overview of statistical learning. If not specified differently, all material is based on the book of James and others [105].

A.1.1 The General Learning Problem

The general learning problem states that we have a set of independent variables $X = (X_1, \cdots, X_n)$ (also called features or predictors) and want to predict a dependent variable Y (also called response). Formally, there is a function f that describes the relationship between independent variables and the dependent variable:

$$Y = f(X_1, \cdots, X_n) + \epsilon_i \tag{A.1}$$

The function f is unknown and needs to be estimated. ϵ_i is a random error term, which has a mean of zero and describes the part of Y that cannot be predicted using X. Therefore, ϵ_i is also called the *irreducible error*.

By estimating f, we can have two goals. The first goal is *prediction*, which computes predictions (or responses) for Y for a known X. The second goal is *inference*, which models the relationship between the independent variables and the dependent variable (i.e., how Y changes when the independent variables $X_i \in X$ change). In this thesis, we are only interested in predictions, and therefore, omit details about inference.

To predict the dependent variable Y given X, we need to estimate the unknown function f. We denote the estimated function as $\hat{f}$ and the estimated response as $\hat{Y}$. The real function $f(X)$ is then:

$$f(X) = \hat{f}(X) + \epsilon_r \tag{A.2}$$

Since $\hat{f}$ is an estimate of f, there will be an error in the prediction, which is qualified by the random error term ϵ_r. By selecting better regression methods, we can estimate f more accurately and reduce ϵ_r. Therefore, ϵ_r is also called the *reducible error*.

A.1.2 Parametric and Non-Parametric Regression

The goal of regression is to learn a function $\hat{f}(X)$ that accurately predicts a response Y for each observation (X, Y) of a training data set so that $Y \approx \hat{f}(X)$. There are two classes of regression methods: parametric and non-parametric methods. The major difference between them is that parametric methods make assumptions on the form of f (e.g., a linear function), and hence solve a restricted learning problem, whereas non-parametric methods make no assumptions on the form of f.

Parametric Methods

Parametric regression methods follow a two step scheme. In the first step, we have to choose a functional form, which we want to learn. For example, we could assume a linear relationship and use a linear model:

$$\hat{f}(x) = \beta_1 x + \beta_0 \tag{A.3}$$

In the second step, we need to fit the function to the training data by estimating the coefficients of the function. In our example, these are β_1 and β_0:

$$Y \approx \beta_1 x + \beta_0 \tag{A.4}$$

Essentially, parametric models reduce the problem of estimating f to estimating the coefficients of a known function form.

Non-Parametric Methods

By contrast, non-parametric regression methods make no assumptions over the function form and avoid the danger of choosing an unsuitable functional form. However, they are required to be more flexible to accurately follow the data points, and this often means that they require much more training data compared to parametric models to make accurate predictions. Furthermore, the high flexibility can lead to a phenomenon called *overfitting*, where the estimated function follows the noise of the training data too closely. Note that this can also happen for parametric models.

A.1.3 Parametric and Non-Parametric Regression Methods

We now describe for each class of regression methods one representative: We discuss linear regression as example for parametric regression and k-nearest neighbor regression for non-parametric regression.

Linear Regression

Linear regression fits a linear model to a set of observations. Hence, linear regression assumes that all n independent variables $X_1, \cdots, X_n$ have a linear relationship with the dependent variable Y. Each independent variable X_i gets a coefficient β_i that quantifies this relationship. Thus, the functional form for n independent variables is:

$$Y = X_n\beta_n + \cdots + \beta_1 X_1 + \beta_0 + \epsilon \tag{A.5}$$

ϵ represents the random error term and includes the reducible and the irreducible error. To fit the model to the training data, we need to estimate each coefficient β_i. Thus, the estimated function $\hat{f}$ is:

$$\hat{f}(X) = \hat{\beta}_n X_n + \cdots + \hat{\beta}_1 X_1 + \hat{\beta}_0 \tag{A.6}$$

A very common approach to estimate the coefficients $\beta_0, \cdots, \beta_n$ is the least squares method, were we choose the coefficients that have the minimal *residual sum of squares* (RSS):

$$\text{RSS} = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \tag{A.7}$$

$$= \sum_{i=1}^{m} (y_i - \hat{\beta}_n x_n - \cdots - \hat{\beta}_1 x_1 - \hat{\beta}_0)^2 \tag{A.8}$$

Here, for each observation (x_i, y_i) in the training data set, the difference between prediction $\hat{y}_i$ and observation y_i is squared. If the difference is very small (< 1), the square will be even smaller. Otherwise, the square will especially penalize large differences between prediction and observation.

To compute the coefficients where the RSS is minimal, we need to compute the derivate function. As we have n independent variables, we have to compute the partial derivate for each independent variable X_i, which results in a system of linear equations. Solving this system of equations results in the desired estimates of the coefficients. Since we want to provide and overview, we will not discuss further details and refer the interested reader to the book of James and others for more details [105].

K-Nearest Neighbor Regression

The *K-Nearest Neighbor* (KNN) regression is a simple non-parametric regression method. For a given K and a set of independent variables X, the K closest points to X are selected and stored in the set N. Then, we can estimate $\hat{Y}$ as follows:

$$\hat{f}(X) = \frac{1}{K} \sum_{(x_i, y_i) \in N} y_i \tag{A.9}$$

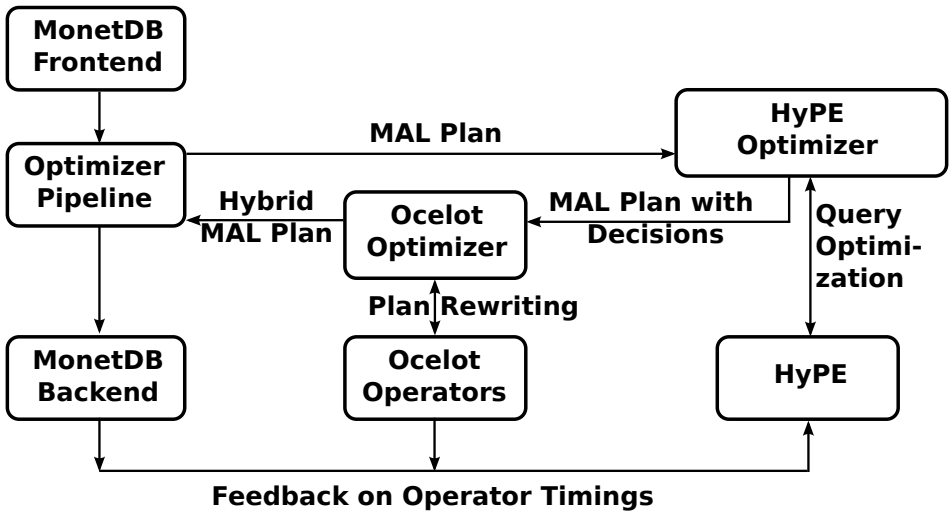


Figure A.1: Overview over the Ocelot/HyPE system.

It is common to use the euclidean distance of two vectors in the feature space (all possible vectors of independent variables) as closeness metric.

A.2 Application of the HyPE Optimizer to MonetDB/Ocelot

In this thesis, we focused on a single database engine. To show that the HyPE optimizer is general enough to be used by other database engines, we integrated HyPE in Ocelot. In this section, we provide details on the integration aspects of the combined Ocelot/HyPE system. The material was published in the following paper:

[42] S. Breß, M. Heimel, M. Saecker, B. Köcher, V. Markl, and G. Saake. Ocelot/HyPE: Optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment*, 7(13):1609–1612, 2014

A.2.1 Motivation

Due to its flexible engine, Ocelot already offers the possibility to place operators across all available devices. However, this placement has to be specified manually, meaning the user has to annotate a specific device for each operator in the query plan. While this approach might be feasible for recurring queries that are specified ahead of time, it is impractical for ad-hoc scenarios. In these cases, it becomes mandatory to decide the placement automatically and without user interaction. Otherwise, we will not be able to fully exploit the heterogeneity of the machine. Accordingly, integrating a specialized optimizer such as HyPE to automate the placement decisions is the logical next step towards our goal of building a DBMS for heterogeneous systems.

A.2.2 Integration

HyPE is implemented in a very straightforward modular fashion, which allowed us to integrate it fairly seamlessly. Basically, there were only two required steps: First, we needed to register Ocelot's operators to HyPE, which was easily accomplished using HyPE's APIs. Second, we needed a mechanism to convey the placement decisions from HyPE to Ocelot, and subsequently feed query runtimes from Ocelot back to HyPE. This step was implemented by directly integrating HyPE into the query optimization pipeline of MonetDB.

The query optimizer of MonetDB is structured as a list of sequential optimizer stages, where each stage transforms a plan into a more efficient, but equivalent one. The final optimized plan is then executed by MonetDB's plan interpreter [100]. This modular design allowed us to easily integrate HyPE's decision logic into the query optimizer. In particular, we only had to add a new optimizer stage that runs before Ocelot injects its operations into the MonetDB plan. This newly added optimizer stage works as follows:

1. Before executing the query, we transform the given MonetDB query plan into the internal representation used by HyPE and hand it over.
2. HyPE chooses a physical plan according to a query optimization heuristic which is specified by the user.
3. Then, the HyPE optimizer step retrieves the scheduling decisions from the resulting HyPE plan and assigns them to their corresponding MAL operators.
4. Afterwards, the Ocelot optimizer step replaces a MAL operator with the respective Ocelot operator and sets the processing device decided by HyPE.
5. Finally, MonetDB executes the query plan. After the query has finished, Ocelot retrieves the measured execution times of all operators and sends them as feedback to HyPE. This information is then used to adjust the learned cost models.

In general, HyPE optimizes the plan ahead of query execution. However, depending on the chosen heuristic, HyPE can also rewrite the query plan during execution, and hence, re-optimize the plan at runtime (e.g., in case cardinality estimates exceed an error threshold).

A.3 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 [95] (at revision 3e75851).[1] As Ocelot is an extension of MonetDB, we include measurements of MonetDB as well. For a fair comparison with CoGaDB, we optimized MonetDB/Ocelot as follows.

We set the database 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

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

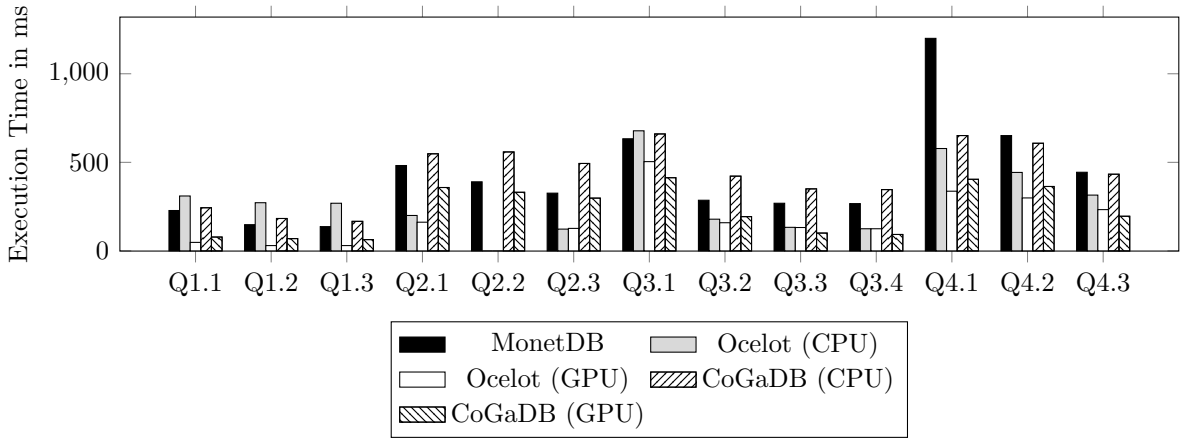


Figure A.2: Query execution times for SSBM queries for a single user and a database of scale factor 10.

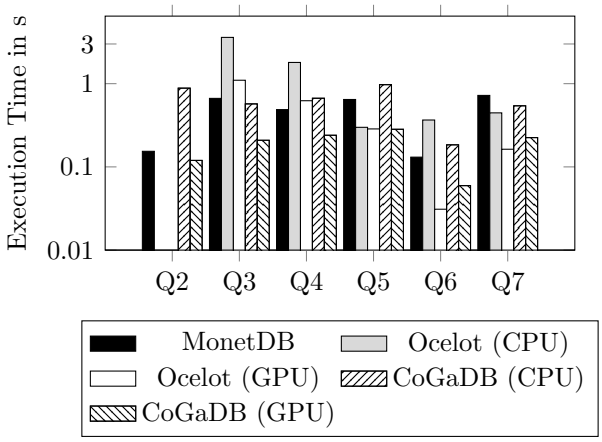


Figure A.3: Query execution times for selected TPC-H queries for a single user and a database of scale factor 10.

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 SSBM in Figure A.2 and the TPC-H benchmark in Figure A.3. 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).

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

A.4 Micro Benchmarks

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

A.4.1 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 A.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 A.1: Serial Selection Queries. Note the interleaved execution. The order table is an alias for the lineorder table.

A.4.2 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 A.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 A.2: Parallel Selection Query

A.5 Workloads

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

A.5.1 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 [5] or GPUDB [204].

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).

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 [148].

A.5.2 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.

A.6 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. 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 [197]. Thus, the problem specific information of *Critical Path* provide an advantage over established optimization strategies that explore a larger part of the optimization space.

Bibliography

[1] Palo GPU accelerator. White Paper, 2010.

[2] Parstream – turning data into knowledge. White Paper, November 2010.

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

[4] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[5] D. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 967–980. ACM, 2008.

[6] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 466–475. IEEE, 2007.

[7] A. Acosta, V. Blanco, and F. Almeida. Dynamic load balancing on heterogeneous multi-GPU systems. *Computers & Electrical Engineering*, 39(8):2591–2602, 2013.

[8] A. Ailamaki, P. Boncz, and S. Manegold, editors. *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, 2005.

[9] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 169–180. Morgan Kaufmann Publishers Inc., 2001.

[10] M. Akdere, U. Cetintemel, E. Upfal, and S. Zdonik. Learning-based query performance modeling and prediction. Technical report, Department of Computer Science, Brown University, 2011.

[11] O. Albayrak, I. Akturk, and O. Ozturk. Effective kernel mapping for opencl applications in heterogeneous platforms. In *Proc. Int'l Conf. on Parallel Processing Workshops (ICPPW)*, pages 81–88. IEEE, 2012.

[12] T. Anderson and J. Finn. *The New Statistical Analysis of Data*. Springer, 1st edition, 1996.

[13] W. Andrzejewski, A. Gramacki, and J. Gramacki. Graphics processing units in acceleration of bandwidth selection for kernel density estimation. *Applied Mathematics and Computer Science*, 23(4):869–885, 2013.

[14] W. Andrzejewski and R. Wrembel. GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. In *Proc. Int'l Conf. on Database and Expert Systems Applications (DEXA)*, pages 315–329. Springer, 2010.

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice & Experience*, 23(2):187–198, 2011.

[16] D. Augustyn and S. Zederowski. Applying CUDA Technology in DCT-Based Method of Query Selectivity Estimation. In *Proc. Int'l Workshop on GPUs In Databases (GID)*, pages 3–12. Springer, 2012.

[17] P. Bakkum and S. Chakradhar. Efficient data management for GPU databases. 2012. http://pbbakkum.com/virginian/paper.pdf.

[18] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proc. Int'l Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, pages 94–103. ACM, 2010.

[19] C. Balkesen, G. Alonso, J. Teubner, and T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[20] C. Balkesen, J. Teubner, G. Alonso, and T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

[21] R. Barrientos, J. Gomez, C. Tenllado, and M. Prieto. Heap based k-nearest neighbor search on GPUs. In *XXI Jornadas de Paralelismo*, pages 559–566, 2010.

[22] A. Becher, F. Bauer, D. Ziener, and J. Teich. Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration. In *Proc. Int'l Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.

[23] F. Beier, T. Kilias, and K.-U. Sattler. GiST scan acceleration using coprocessors. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 63–69. ACM, 2012.

[24] N. Bell and J. Hoberock. *GPU Computing Gems Jade Edition*, chapter Thrust: A Productivity-Oriented Library for CUDA, pages 359–371. Morgan Kaufmann Publishers Inc., 2011.

[25] M. Belviranli, L. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):57:1–57:20, 2013.

[26] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 283–296. ACM, 2009.

[27] A. Binotto, C. Pereira, A. Kuijper, A. Stork, and D. Fellner. An effective dynamic scheduling runtime and tuning system for heterogeneous multi and many-core desktop platforms. In *Proc. Int'l Conf. on High Performance Computing & Communication (HPCC)*, pages 78–85. IEEE, 2011.

[28] S. Bochkanov and V. Bystritsky. ALGLIB. http://www.alglib.net, 2013. [Online; accessed 26-April-2013].

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

[30] P. Boncz, M. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[31] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proc. Int'l Conf. on Innovative Data Systems Research (CIDR)*, pages 225–237, 2005.

[32] R. Bordawekar, C. Lang, and B. Gedik, editors. *Proc. Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2013.

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

[34] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proc. Int'l Conf. on Computing Frontiers (CF)*, pages 21:1–21:10. ACM, 2013.

[35] S. Breß. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *The VLDB PhD workshop, PVLDB*, 6(12):1398–1403, 2013.

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

[37] 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.

[38] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake. Automatic selection of processing units for coprocessing in databases. In *Proc. Int'l Conf. on Advances in Databases and Information Systems (ADBIS)*, pages 57–70. Springer, 2012.

[39] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, 2016. to appear.

[40] S. Breß, I. Geist, E. Schallehn, M. Mory, and G. Saake. A framework for cost based optimization of hybrid CPU/GPU query plans in database systems. *Control and Cybernetics*, 41(4):715–742, 2012.

[41] S. Breß, R. Haberkorn, and S. Ladewig. CoGaDB reference manual, 2014. http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/current/doc/refman.pdf.

[42] S. Breß, M. Heimel, M. Saecker, B. Köcher, V. Markl, and G. Saake. Ocelot/HyPE: Optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment*, 7(13):1609–1612, 2014.

[43] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Exploring the design space of a GPU-aware database architecture. In *Proc. Int'l Workshop on GPUs In Databases (GID)*, pages 225–234. Springer, 2013.

[44] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. GPU-accelerated database systems: Survey and open challenges. *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, 15:1–35, 2014.

[45] S. Breß, N. Siegmund, L. Bellatreche, and G. Saake. An operator-stream-based scheduling engine for effective GPU coprocessing. In *Proc. Int'l Conf. on Advances in Databases and Information Systems (ADBIS)*, pages 288–301. Springer, 2013.

[46] 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.

[47] D. Broneske. Adaptive reprogramming for databases on heterogeneous processors. In *SIGMOD/PODS Ph.D. Symposium*, pages 51–55. ACM, 2015.

[48] D. Broneske, S. Breß, M. Heimel, and G. Saake. Toward hardware-sensitive database operations. In *Proc. Int'l Conf. on Extending Database Technology (EDBT)*, pages 229–234. OpenProceedings.org, 2014.

[49] D. Broneske, S. Breß, and G. Saake. Database scan variants on modern CPUs: A performance study. In *Proc. Int'l Workshop on In Memory Data Management and Analytics (IMDM)*, pages 97–111. Springer, 2014.

[50] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *Proc. Int'l Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 151–160. ACM, 2014.

[51] S. Chaudhuri. An overview of query optimization in relational systems. In *Symposium on Principles of Database Systems (PODS)*, pages 34–43. ACM, 1998.

[52] C. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 161–172. ACM, 1994.

[53] H. Choi, D. Son, S. Kang, J. Kim, H. Lee, and C. Kim. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *The Journal of Supercomputing*, 65(2):886–902, 2013.

[54] J. Dees and P. Sanders. Efficient many-core query execution in main memory column-stores. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 350–361. IEEE, 2013.

[55] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[56] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364. ACM, 2010.

[57] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili. Efficient Relational Algebra Algorithms and Data Structures for GPU. Technical report, Center for Experimental Research in Computer Systems (CERS), 2012.

[58] S. Dorok. The relational way to dam the flood of genome data. In *SIGMOD-/PODS Ph.D. Symposium*, pages 9–13. ACM, 2015.

[59] S. Dorok, S. Breß, H. Läpple, and G. Saake. Toward efficient and reliable genome analysis using main memory database systems. In *Proc. Int'l Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 34:1–34:4. ACM, 2014.

[60] S. Dorok, S. Breß, and G. Saake. Toward efficient variant calling inside main-memory database systems. In *Proc. Int'l Workshop on Biological Knowledge Discovery and Data Mining (BIOKDD-DEXA)*, pages 41–45. IEEE, 2014.

[61] S. Dorok, S. Breß, J. Teubner, and G. Saake. Flexible analysis of plant genomes in a database management system. In *Proc. Int'l Conf. on Extending Database Technology (EDBT)*, pages 509–512. OpenProceedings.org, 2015.

[62] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. Int'l Symposium on Computer Architecture (ISCA)*, pages 365–376. ACM, 2011.

[63] R. Fang, B. He, M. Lu, K. Yang, N. Govindaraju, Q. Luo, and P. Sander. GPUQP: Query co-processing using graphics processors. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 1061–1063. ACM, 2007.

[64] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.

[65] A. Ganapathi, H. Kuno, U. Dayal, J. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 592–603. IEEE, 2009.

[66] M. Garba and H. González-Vélez. Asymptotic peak utilisation in heterogeneous parallel CPU/GPU pipelines: a decentralised queue monitoring strategy. *Parallel Processing Letters*, 22(2), 2012.

[67] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *Proc. Int'l Conf. on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1–6. IEEE, 2008.

[68] M. Garofalakis and Y. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 296–305. Morgan Kaufmann Publishers Inc., 1997.

[69] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing With Opencl*. Elsevier Science & Technology, 2012.

[70] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 461–472. ACM, 2001.

[71] P. Ghodsnia. An in-GPU-memory column-oriented database for processing analytical workloads. In *The VLDB PhD Workshop*. VLDB Endowment, 2012.

[72] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*. ACM, 2005.

[73] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 325–336. ACM, 2006.

[74] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 102–111. ACM, 1990.

[75] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[76] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. In *Proc. Int'l Workshop on Applications for Multi- and Many-Core Processors (A4MMC)*, 2011.

[77] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *Proc. Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 134–144. IEEE, 2011.

[78] D. Grewe and M. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proc. Int'l Conf. on Compiler Construction (CC)*, pages 286–305. Springer, 2011.

[79] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting query execution times for autonomous workload management. In *Proc. Int'l Conf. on Autonomic Computing (ICAC)*, pages 13–22. IEEE, 2008.

[80] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[81] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 383–394. ACM, 2005.

[82] W. Hasan. *Optimization of SQL Queries for Parallel Machines*, volume 1182 of *Lecture Notes in Computer Science*. Springer, 1996.

[83] W. Hasan, D. Florescu, and P. Valduriez. Open issues in parallel query optimization. *SIGMOD Record*, 25(3):28–33, 1996.

[84] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 260–269. ACM, 2008.

[85] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Relational query co-processing on graphics processors. In *ACM Transactions on Database Systems (TODS)*, volume 34, pages 21:1–21:39. ACM, 2009.

[86] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 511–524. ACM, 2008.

[87] B. He and J. Yu. High-Throughput Transaction Executions on Graphics Processors. *Proceedings of the VLDB Endowment*, 4(5):314–325, 2011.

[88] 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.

[89] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled CPU-GPU architectures. *Proc. VLDB Endow.*, 8(4):329–340, 2014.

[90] M. Heimel. Designing a database system for modern processing architectures. In *SIGMOD/PODS Ph.D. Symposium*, pages 13–18. ACM, 2013.

[91] M. Heimel, F. Haase, M. Meinke, S. Breß, M. Saecker, and V. Markl. Demonstrating self-learning algorithm adaptivity in a hardware-oblivious database engine. In *Proc. Int'l Conf. on Extending Database Technology (EDBT)*, pages 616–619. OpenProceedings.org, 2014.

[92] M. Heimel, M. Kiefer, and V. Markl. Demonstrating transfer-efficient sample maintenance on graphics cards. In *Proc. Int'l Conf. on Extending Database Technology (EDBT)*, pages 513–516. OpenProceedings.org, 2015.

[93] M. Heimel, M. Kiefer, and V. Markl. Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 1477–1492. ACM, 2015.

[94] M. Heimel and V. Markl. A first step towards GPU-assisted query optimization. In *Proc. Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 33–44, 2012.

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

[96] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 562–573. Morgan Kaufmann Publishers Inc., 1995.

[97] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the cell broadband engine. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 4:1–4:6. ACM, 2007.

[98] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[99] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

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

[101] A. Ilić, F. Pratas, P. Trancoso, and L. Sousa. *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*, chapter High-Performance Computing on Heterogeneous Systems: Database Queries on CPU and GPU, pages 202–222. IOS Press, 2011.

[102] A. Ilić and L. Sousa. CHPS: An Environment for Collaborative Execution on Heterogeneous Desktop Systems. *International Journal of Networking and Computing*, 1(1):96–113, 2011.

[103] M. Iverson, F. Ozguner, and G. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In *Proc. Int'l Symposium on High Performance Distributed Computing (HPDC)*, pages 263–270. IEEE, 1996.

[104] M. Iverson, F. Özgüner, and L. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Transactions on Computers*, 48(12):1374–1379, 1999.

[105] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer, 2014.

[106] S. Jha, B. He, M. Lu, X. Cheng, and H. Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *Proceedings of the VLDB Endowment*, 8(6):642–653, 2015.

[107] V. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proc. Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 19–33, 2009.

[108] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *Proceedings of the VLDB Endowment*, 1(1):622–634, 2008.

[109] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 55–62. ACM, 2012.

[110] T. Karnagel, D. Habich, and W. Lehner. Local vs. global optimization: Operator placement strategies in heterogeneous environments. In *Proc. Int'l Workshop on Data (Co-)Processing on Heterogeneous Hardware (DAPHNE)*, pages 48–55. OpenProceedings.org, 2015.

[111] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The hells-join: A heterogeneous stream join for extremely large windows. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 2:1–2:7. ACM, 2013.

[112] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. Heterogeneity-aware operator placement in column-store DBMS. *Datenbank-Spektrum*, 14(3):211–221, 2014.

[113] T. Karnagel, M. Hille, M. Ludwig, D. Habich, W. Lehner, M. Heimel, and V. Markl. Demonstrating efficient query processing in heterogeneous environments. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 693–696. ACM, 2014.

[114] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 195–206. IEEE, 2011.

[115] A. Kemper and I. Pandis, editors. *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*. ACM, 2014.

[116] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU workloads and systems. In *Proc. Int'l Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, pages 31–42. ACM, 2010.

[117] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 339–350. ACM, 2010.

[118] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proc. Int'l Conf. on Supercomputing (ICS)*, pages 149–160. ACM, 2013.

[119] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[120] K. Krikellas, M. Cintra, and S. Viglas. Scheduling threads for intra-query parallelism on multicore processors. Technical Report EDI-INF-RR-1345, University of Edinburgh, School of Informatics, 2010. http://www.inf.ed.ac.uk/publications/report/1345.html.

[121] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 613–624. IEEE, 2010.

[122] J. Krueger, M. Grund, I. Jaeckel, A. Zeier, and H. Plattner. Applicability of GPU computing for efficient merge in in-memory databases. In *Proc. Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. VLDB Endowment, 2011.

[123] R. Lanzelotte, P. Valduriez, M. Zaït, and M. Ziane. Invited project review: Industrial-strength parallel query optimization: issues and lessons. *Information Systems*, 19(4):311–330, 1994.

[124] T. Lauer, A. Datta, Z. Khadikov, and C. Anselm. Exploring graphics processing units as parallel coprocessors for online aggregation. In *Proc. Int'l Workshop on Data Warehousing and OLAP (DOLAP)*, pages 77–84. ACM, 2010.

[125] C. Lee, W. Ro, and J. Gaudiot. Cooperative heterogeneous computing for parallel processing on CPU/GPU hybrids. In *Proc. Int'l Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, pages 33–40. IEEE, 2012.

[126] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 743–754. ACM, 2014.

[127] M. Lu, L. Zhang, H. Huynh, Z. Ong, Y. Liang, B. He, R. Goh, and R. Huynh. Optimizing the mapreduce framework on Intel Xeon Phi coprocessor. In *Proc. Int'l Conf. on Big Data (Big Data)*, pages 125–130. IEEE, 2013.

[128] M. Malik, L. Riha, C. Shea, and T. El-Ghazawi. Task scheduling for GPU accelerated hybrid OLAP systems with multi-core support and text-to-integer translation. In *Proc. Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1987–1996. IEEE, 2012.

[129] S. Manegold, P. Boncz, and M. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[130] S. Manegold, P. Boncz, and M. Kersten. Generic database cost models for hierarchical memory systems. In *Proceedings of the VLDB Endowment*, pages 191–202. VLDB Endowment, 2002.

[131] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-conscious radix-decluster projections. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 684–695. VLDB Endowment, 2004.

[132] S. Manegold, M. Kersten, and P. Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment*, 2(2):1648–1653, 2009.

[133] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 659–670. ACM, 2004.

[134] A. Meister. GPU-accelerated join-order optimization. *The VLDB PhD workshop, PVLDB*, 2015.

[135] A. Meister, S. Breß, and G. Saake. Toward GPU-accelerated database optimization. *Datenbank-Spektrum*, 15(2):131–140, 2015.

[136] S. Mittal and J. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, pages 1–35, 2015. to appear.

[137] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

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

[139] R. Moussalli, R. Halstead, M. Salloum, W. Najjar, and V. Tsotras. Efficient XML path filtering using GPUs. In *Proc. Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2011.

[140] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[141] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Kemper, and T. Neumann. Heterogeneity-conscious parallel query execution: Getting a better mileage while driving faster! In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 2:1–2:10. ACM, 2014.

[142] R. Natarajan and E. Pednault. Segmented regression estimators for massive data sets. In *Proc. SIAM Int'l Conf. on Data Mining (SDM)*, pages 566–582. SIAM, 2002.

[143] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[144] NVIDIA. NVIDIAs next generation CUDA compute architecture: Fermi. Whitepaper, NVIDIA Corp., 2009.

[145] NVIDIA. NVIDIAs next generation CUDA compute architecture: Kepler TM GK110. Whitepaper, NVIDIA Corp., 2012.

[146] NVIDIA. NVIDIA CUDA C programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2014. pp. 31–36, Version 6.0, [Online; accessed 18-May-2014].

[147] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

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

[149] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[150] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment*, 3(1-2):928–939, 2010.

[151] E. Pednault. Transform regression and the kolmogorov superposition theorem. In *Proc. SIAM Int'l Conf. on Data Mining (SDM)*, pages 35–46. SIAM, 2006.

[152] J. A. Pienaar, A. Raghunathan, and S. Chakradhar. MDR: Performance model driven runtime for heterogeneous parallel platforms. In *Proc. Int'l Conf. on Supercomputing (ICS)*, pages 225–234. ACM, 2011.

[153] M. Pinnecke, D. Broneske, and G. Saake. Toward GPU accelerated data stream processing. In *Proc. German Nat'l Workshop on Foundations of Databases (GvD)*, pages 78–83. CEUR-WS, 2015.

[154] H. Pirk. Efficient cross-device query processing. In *The VLDB PhD Workshop*. VLDB Endowment, 2012.

[155] H. Pirk, S. Madden, and M. Stonebraker. By their fruits shall ye know them: A data analyst's perspective on massively parallel system design. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 5:1–5:6. ACM, 2015.

[156] H. Pirk, S. Manegold, and M. Kersten. Accelerating foreign-key joins using asymmetric memory channels. In *Proc. Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 585–597. VLDB Endowment, 2011.

[157] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 508–519. IEEE, 2014.

[158] H. Pirk, T. Sellam, S. Manegold, and M. Kersten. X-device query processing by bitwise distribution. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 48–54. ACM, 2012.

[159] H. Plattner and A. Zeier. *In-Memory Data Management*. Springer, 2nd edition, 2012.

[160] P. Przymus. *Query Optimization in heterogeneous CPU/GPU environment for time series databases*. PhD thesis, University of Warsaw, Warsaw, Poland, March 2014.

[161] P. Przymus and K. Kaczmarski. Dynamic compression strategy for time series database using GPU. In *Proc. Int'l Conf. on Advances in Databases and Information Systems (ADBIS)*, pages 235–244. Springer, 2013.

[162] P. Przymus, K. Kaczmarski, and K. Stencel. A bi-objective optimization framework for heterogeneous CPU/GPU query plans. In *Proc. Int'l Workshop on Concurrency, Specification and Programming (CS&P)*, pages 342–354. CEUR-WS, 2013.

[163] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing data and work across concurrent analytical queries. *Proceedings of the VLDB Endowment*, 6(9):637–648, 2013.

[164] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In *Proc. Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 36–45. VLDB Endowment, 2013.

[165] T. Rabl, M. Poess, H.-A. Jacobsen, P. O'Neil, and E. O'Neil. Variations of the star schema benchmark to test the effects of data skew on query performance. In *Proc. Int'l Conf. on Performance Engineering (ICPE)*, pages 361–372. ACM, 2013.

[166] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 60–69. IEEE, 2008.

[167] V. Raman and others. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091.

[168] H. Rauhe. *Finding the Right Processor for the Job – Co-Processors in a DBMS*. PhD thesis, Ilmenau University of Technology, Ilmenau, Germany, October 2014.

[169] H. Rauhe, J. Dees, K.-U. Sattler, and F. Faerber. Multi-level parallel query execution framework for CPU and GPU. In *Proc. Int'l Conf. on Advances in Databases and Information Systems (ADBIS)*, pages 330–343. Springer, 2013.

[170] V. Ravi and G. Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. In *Proc. Int'l Conf. on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2011.

[171] V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl. The operator variant selection problem on heterogeneous hardware. In *Proc. Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. VLDB Endowment, 2015.

[172] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 1231–1242. ACM, 2013.

[173] M. Saecker and V. Markl. Big data analytics on modern hardware architectures: A technology survey. In *Business Intelligence – Second European Summer School (eBISS)*, pages 125–149. Springer, 2012.

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

[175] D. Schaa and D. Kaeli. Exploring the multiple-GPU design space. In *Proc. Int'l Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12. IEEE, 2009.

[176] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. QuEval: Beyond high-dimensional indexing à la carte. *Proceedings of the VLDB Endowment*, 6(14):1654–1665, 2013.

[177] E. Schlicht. *Isolation and Aggregation in Economics*. Springer, 1985.

[178] S. Scott. A modern bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658, 2010.

[179] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 23–34. ACM, 1979.

[180] D. Shasha and P. Bonnet. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann, 2002.

[181] K. Shirahata, H. Sato, and S. Matsuoka. Hybrid map task scheduling for GPU-based heterogeneous clusters. In *Proc. Int'l Conf. on Cloud Computing Technology and Science (CLOUDCOM)*, pages 733–740. IEEE, 2010.

[182] S. Shukla and L. Bhuyan. A hybrid shared memory heterogeneous execution platform for PCIe-based GPGPUs. In *Proc. Int'l Conf. on High Performance Computing (HiPC)*, pages 343–352. IEEE, 2013.

[183] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., 6th edition, 2010. pages 802–804.

[184] F. Song and J. Dongarra. A scalable framework for heterogeneous GPU-based clusters. In *Proc. Int'l Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 91–100. ACM, 2012.

[185] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 19–28. Morgan Kaufmann Publishers Inc., 2001.

[186] H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005. http://www.gotw.ca/publications/concurrencyddj.htm.

[187] X. Tang and S. Chanson. Optimizing static job scheduling in a network of heterogeneous computers. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, pages 373–382. IEEE, 2000.

[188] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 625–636. ACM, 2011.

[189] H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[190] D. Tsirogiannis, S. Harizopoulos, and M. Shah. Analyzing the energy efficiency of a database server. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 231–242. ACM, 2010.

[191] J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Proc. European Conf. on Machine Learning (ECML)*, pages 437–448. Springer, 2005.

[192] S. Viglas. Just-in-time compilation for SQL query processing. *Proceedings of the VLDB Endowment*, 6(11):1190–1191, 2013.

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

[194] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *Proc. Int'l Conf. on High Performance Computing & Simulation (HPCS)*, pages 24–32. IEEE, 2011.

[195] W. Wang and L. Cao. Parallel k-nearest neighbor search on graphics hardware. In *Proc. Int'l Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 291–294. IEEE, 2010.

[196] Z. Wang, L. Zheng, Q. Chen, and M. Guo. CAP: Co-scheduling based on asymptotic profiling in CPU+GPU hybrid systems. In *Proc. Int'l Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, pages 107–114. ACM, 2013.

[197] J. Wedding. Operator placement heuristics in heterogeneous processor environments. Master thesis, Otto-von-Guericke Universität Magdeburg, Germany, 2015.

[198] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Proc. Int'l Symposium on Microarchitecture (MICRO)*, pages 107–118. IEEE, 2012.

[199] 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 *Proc. Int'l Symposium on Code Generation and Optimization (CGO)*, pages 44:44–44:54. ACM, 2014.

[200] W. Wu, Y. Chi, H. Hacígümüş, and J. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment*, 6(10):925–936, 2013.

[201] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacıgümüş, and J. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 1081–1092. IEEE, 2013.

[202] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive optimization for petascale heterogeneous CPU/GPU computing. In *Proc. Int'l Conf. on Cluster Computing (CLUSTER)*, pages 19–28. IEEE, 2010.

[203] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *Proc. Int'l Workshop on Data Management on New Hardware (DaMoN)*, pages 1–9. ACM, 2011.

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

[205] N. Zhang, P. Haas, V. Josifovski, G. Lohman, and C. Zhang. Statistical learning techniques for costing XML queries. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 289–300. VLDB Endowment, 2005.

[206] S. Zhang, J. He, B. He, and M. Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *Proceedings of the VLDB Endowment*, 6(12):1374–1377, 2013.

[207] J. Zhong and B. He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 99:1–14, 2013.

[208] J. Zhong and B. He. Parallel graph processing on graphics processors made easy. *Proceedings of the VLDB Endowment*, 6(12):1270–1273, 2013.

[209] J. Zhou and K. Ross. Implementing database operations using SIMD instructions. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 145–156. ACM, 2002.

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht."

Magdeburg, den 30.10.2015

Sebastian Breß