# Query Processing On Heterogeneous Systems

vorgelegt von
Dipl.-Inf. Viktor Rosenfeld
ORCID: 0000-0002-6001-4442

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 8. Dezember 2023

Berlin 2023

# Abstract

Today's computing systems are highly heterogeneous, both in terms of the hardware that they are built from and the software that they run. This heterogeneity provides key benefits, since specialized processors provide the performance necessary to process increasing amounts of data, and specialized software system enable programmers with different needs and expertise to extract knowledge from data. However, the integration of heterogeneous hardware and software into a cohesive system poses many challenges, e.g., how to distribute work among heterogeneous processors, how to reduce the complexity of programming heterogeneous processors, and how to reduce overheads when moving data and computation between different software systems.

In this thesis, we investigate how heterogeneous hardware and software impacts query processing, and develop tools to manage this heterogeneity.

In our first contribution, we survey query processing systems that target both CPUs and GPUs. We develop a classification scheme to categorize how to distribute query processing tasks on these processors, and review techniques to program CPUs and GPUs in a query processing system, and to overcome the data transfer bottleneck. Our analysis shows that systems with dedicated GPUs require different strategies to distribute the workload than systems with integrated GPUs.

In our second contribution, we investigate how a query processing systems can automatically adapt its low-level operator implementation to the processor it runs on. First, we perform an extensive experimental analysis of different implementations of two data processing operators on various CPUs, GPUs, and an Intel Xeon Phi coprocessor. We find that every processor requires specific implementations to achieve high performance, that heuristics derived from the analysis of one processor are not necessarily transferable to others, and that some processors are more difficult to optimize for than others. Then, we develop two algorithms to find a fast operator implementation at runtime, which differ in the amount of information they incorporate during their search. Our evaluation of these algorithms shows that it is necessary to exploit information about processor characteristics to reduce the search time for a fast implementation.

In our third contribution, we focus on integrating query processing systems that run on the Java Virtual Machine (JVM) and those that execute native machine code. Concretely, we develop two approaches how to execute Java-based UDFs in a query processing engine written in C++. First, we transparently transform a tuple-based UDF into a UDF that processes an entire batch, which effectively eliminates the overhead of moving execution from the C++ engine into an embedded JVM. This approach works well if the UDF creates many objects, including Java strings. Second, we translate the UDF bytecode to LLVM IR and link it directly with the C++ engine. This approach works well if the UDF is compute-intensive.

Our investigation into a variety of heterogeneous hardware and software scenarios shows that we often face similar challenges when integrating them. Thus, the techniques investigated and developed in this thesis constitute effective building blocks to extract performance from today's heterogeneous computing systems.

# Zusammenfassung

Heutige Computersysteme sind außerordentlich heterogen. Sie bestehen aus vielseitig spezialisierter Hardware und führen unterschiedlichste Softwaresysteme aus. Einerseits bietet diese Heterogenität wesentliche Vorteile, da z.B. spezialisierte Prozessoren die notwendige Leistung für die Verarbeitung wachsender Datenmengen bereitstellen und spezialisierte Softwaresysteme Programmierern mit unterschiedlichen Anforderungen und Fachkenntnissen ermöglichen, Wissen aus Daten zu erschließen. Andererseits birgt die Integration heterogener Hard- und Software aber auch viele Herausforderungen. So stellt sich beispielweise die Frage, wie die Arbeitsteilung zwischen heterogenen Prozessoren erfolgen soll, wie sich die Komplexität der Programmierung heterogener Prozessoren reduzieren lässt und wie der Overhead bei der Übertragung von Daten und der Verteilung von Berechnungen zwischen verschiedenen Softwaresystemen minimiert werden kann.

In dieser Arbeit untersuchen wir, wie heterogene Hard- und Software die Anfrageverarbeitung beeinflusst, und entwickeln Werkzeuge, um diese Heterogenität zu bewältigen.

Im ersten Beitrag untersuchen wir Systeme, die die Anfrageverarbeitung auf CPUs und GPUs aufteilen. Hierfür entwickeln wir ein Klassifikationsschema, um Strategien zur Verteilung von Teilaufgaben der Anfrageverarbeitung auf diesen Prozessoren zu kategorisieren. Darüber hinaus untersuchen wir Methoden um die Architektur eines Anfragesystems an heterogene Prozessoren anzupassen und um langsame Datentransfers zwischen CPU und GPU zu vermeiden. Unsere Analyse zeigt, dass Systeme mit dedizierten GPUs andere Strategien zur Arbeitsaufteilung erfordern als Systeme mit integrierten GPUs.

Im zweiten Beitrag untersuchen wir, wie ein Anfrageverarbeitungssystem die Implementierung von datenverarbeitenden Operatoren automatisch an den Prozessor adaptieren kann, auf dem es ausgeführt wird. Hierfür führen wir zunächst eine umfangreiche experimentelle Analyse verschiedener Implementierungsvarianten für zwei Operatoren auf verschiedenen CPUs, GPUs und einem Intel Xeon Phi Coprozessor durch. Unsere Analyse zeigt, dass jeder Prozessor eine spezifische Implementierung benötigt, um Daten schnell zu verarbeiten. Darüber hinaus stellen wir fest, dass Heuristiken, die aus der Analyse eines bestimmten Prozessors abgeleitet wurden, nicht unbedingt auf andere Prozessoren übertragbar sind und dass einige Prozessoren schwieriger zu optimieren sind als andere. Anschließend entwickeln wir zwei Algorithmen, die es einem Anfragesystem ermöglichen, automatisch eine schnelle Operatorimplementierung zu finden. Die Algorithmen unterscheiden sich in der Art der Informationen, die sie bei der Suche berücksichtigen. Unsere Auswertung dieser Algorithmen zeigt, dass es notwendig ist, Informationen über die Eigenschaften des Prozessors zu berücksichtigen, um die Suchzeit für eine schnelle Implementierung zu reduzieren.

Im dritten Beitrag befassen wir uns mit der Integration von Anfrageverarbeitungssystemen, die auf der Java Virtual Machine laufen, und solchen, die nativen Maschinencode ausführen. Konkret entwickeln wir zwei Ansätze, um Java-basierte UDF auf einem in C++ geschriebenen Anfrageverarbeitungssystem auszuführen. Im ersten Ansatz wandeln wir eine UDF, die für jedes Tupel ausgeführt wird, in eine UDF um, die einen Block von Tupel verarbeitet, wodurch der Kontextwechsel-Overhead zwischen der Ausführung von C++-Code und der UDF eliminiert wird. Dieser Ansatz funktioniert gut für UDFs, die viele Objekte erzeugen, einschließlich Java-Strings. Im zweiten Ansatz übersetzen wir die Java UDF in Maschinencode und führen sie direkt im C++ Anfragebearbeitungssystem aus. Dieser Ansatz funktioniert gut für rechenintensive UDFs.

Unsere Untersuchung einer Reihe von Szenarien heterogener Hard- und Software zeigt, dass wir bei der Integration dieser Systeme oft vor ähnlichen Herausforderungen stehen. Somit stellen die in dieser Arbeit erarbeiteten Methoden wirksame Bausteine dar, um die Leistungsfähigkeit heutiger heterogener Rechensysteme auszunutzen.

# Contents

# 1 Introduction

## 1.1. Heterogeneous hardware and software in the era of big data

Big data increasingly shapes decision making in the sciences, industry, and society as a whole [2, 280, 316]. The modern scientific toolbox includes data-driven exploration as an important tool in addition to experimentation, modeling, and simulation [126]. For businesses, data constitute an essential factor of production, similarly to raw materials and human labor [193]. Governments rely on data to increase their operational efficiency and to set public policy [157]. Heterogeneous hardware and software systems underpin these societal trends.

The term *big data* puts the focus primarily on the amount of data. Indeed, the amount of digital data has grown exponentially since the early 2000s [127]. The world wide web, particularly e-commerce and social media web sites, sensors in mobile devices and the emerging Internet of Things, large scientific experiments, and traditional retail businesses generate datasets that are measured in terabytes and petabytes. However, the era of big data would not be possible without two concurrent trends in computing [2]. First, the exponential growth in computing performance [68], as well as open source software systems, have provided the resources and tools to store and process this abundance of data at greatly reduced costs. And second, data has become democratized. More and more people work with data, not just database administrators and IT specialists, but also business managers, scientists, journalists, and consumers.

When we look at the computing infrastructure to process big data, we find that hardware and software systems are highly heterogeneous [347]. On the hardware side, computing systems contain a large variety of microprocessors with different capabilities, e.g., multi-core processors, graphic processing units (GPUs), digital signal processors (DSPs), tensor processing units (TPUs), or field-programmable gate arrays (FPGAs) [347]. On the software side, data scientist work with a variety of software environments and programming languages [140]. In part, this heterogeneity is the result of market forces, as hardware manufacturers and software developers continuously refine their products. These market forces in turn are shaped by fundamental trade-offs, which constrain the performance of the computing infrastructure, as products are designed for different market segments. In a nutshell, manufacturers of microprocessors have to trade off microprocessor performance against energy efficiency, whereas software environments trade off application performance against programmer productivity and supported functionality.

This heterogeneity is both a benefit and a challenge. While it enables the tools that allow a diverse set of users to process large data sets, it also increases the complexity of the computing infrastructure. Therefore, in this thesis, we investigate how heterogeneous hardware and software impacts query processing, and develop tools to manage this heterogeneity. Before we describe this thesis' goals and contributions, we motivate the forces that drive hardware and software heterogeneity in more detail.

### 1.1.1. Drivers of hardware heterogeneity

One of the main constraints on the performance of computing hardware, and specifically of microprocessors, is their power consumption. Since the early 2000s, it has not been economical to increase microprocessor performance by simply increasing their operating speed. To overcome this so-called power wall, the microprocessor industry has pursued three strategies to continue the exponential growth of microprocessor performance. Specifically, modern microprocessors (a) parallelize execution on multiple independent processing cores, and they are either (b) specialized to meet specific application demands, or (c) integrate multiple specialized processing cores.

As a consequence of these three strategies, the processors deployed in today's computing systems are highly heterogeneous. They contain multiple processing cores with diverse characteristics and integrate them in different ways. We can find such heterogeneous systems across the entire range of computing systems, from super computers [321] and cloud infrastructure [15, 92, 202], to consumer laptops [21], mobile devices [255], and embedded industry applications [290]. To achieve high performance, applications running on modern computing system have to take their heterogeneous nature into account and use the available compute resources effectively. They have to match diverse application demands to the most suitable processing core and adapt their algorithms and implementations to the characteristics of the processor that they run on.

**The power wall.** In the 1980s and 1990s, two mechanisms drove the exponential growth in microprocessor performance. First, processor manufacturers relied on Dennard scaling [67] to reduce the size of transistors and increase their operating frequency. A higher operating frequency means that a microprocessor can execute the same instructions in less time. Second, processors manufacturers exploited the increasing transistor budget to implement large caches as well as microarchitectural advances, such as deep pipelines, branch prediction and speculative execution [125]. These techniques further reduce the latency of individual instructions. Thus, even when programs were left unchanged, their performance increased over time just by running them on a faster processor. Specifically, sequential processor performance doubled every 2.5 years between 1978 and 1986 and every 1.5 years between 1986 and 2003 [124]. Unfortunately, this is no longer the case.

An undesirable side effect of increasing the sequential performance of microprocessors is a corresponding increase in their power consumption, since power consumption scales linearly with operating frequency [125]. By 2003, high-performance commodity CPUs operated at 3 GHz and drew more than 100 Watts [273]. Since then, processor frequencies have stagnated [273]. Increasing them further would produce excessive heat and require cooling solutions that are impractical and uneconomical. Furthermore, the microarchitectural techniques that reduce the latency of individual instructions are not energy-efficient. Over multiple processor generations, their impact on power consumption is quadratic compared to their impact on sequential performance [100].

**Parallelization.** Even though processor frequencies have stagnated, the size of transistors is still shrinking and the number of transistors of modern microprocessors is still increasing [273]. Processor manufacturers utilize this transistor budget to integrate multiple independent processing cores, which run in parallel, in a single microprocessor. Thus, modern processors rely on exploiting data-level parallelism and task-level parallelism to achieve high performance.

**Specialization.** Modern multi-core CPUs integrate a moderate number of parallel processing cores, but these cores are still optimized for sequential performance in order to speed up the inherently sequential parts of an application [39]. Consequently, multicore CPUs are a good fit for applications which are latency-critical or dominated by sequential execution paths. However, not all applications can be characterized this way.

To meet specific application demands, processor manufacturers have created specialized microprocessor designs. For example, GPUs are optimized for throughput performance to speed up applications that are highly data-parallel [182]; TPUs are optimized for low-precision matrix operations which are commonly found in deep learning applications [138]; FPGAs allow developers to create processors that are customized to a specific application [79]; digital signal processors (DSPs) process (digitized) analog signals, such as sounds, speech, images, videos, or radar pulses [73]; neuromorphic processors [66, 197] emulate the behaviors and properties biological neurons; and molecular dynamics processors [232, 285] simulate biological molecules at the atomic level. Specialized processors avoid allocating transistors on hardware features that do not fit application demands, which enables them to achieve higher performance than general-purpose processors while staying in the same power budget.

**Integration.** GPUs, TPUs, and FPGAs are originally designed as dedicated coprocessors, which are connected to a traditional multi-core CPU over a system bus. Alternatively, multiple specialized processing cores can also be integrated into a single asymmetric and/or heterogeneous multi-core processor. Asymmetric multi-core processors (AMPs) combine a small number of complex high-performance cores with a large number of simple low-performance cores to speed up both the latency-critical sequential parts as well as the throughput-critical parallel parts of an application [128]. Compared to traditional multi-core processors, in which all cores are of the same design, AMPs can achieve better overall performance within the same transistor budget [128].

Heterogeneous multi-core processors, which integrate specialized processing cores to support specific applications in a highly power-efficient way, achieve even better performance than AMPs for parallel workloads [57]. Such processors can reduce the operating frequency of different processing cores to balance the power budget according to application demands [41], or turn off individual cores completely when they are not in use [168]. Moving data between specialized cores is also faster and requires less energy than moving it between the CPU and a dedicated coprocessor, because the cores are physically closer together and are connected by an on-chip interconnect [21].

### 1.1.2. Drivers of software heterogeneity

We can identify at least three factors that lead to heterogeneous software systems. First, specialized computing hardware, such as the specialized microprocessors we discussed in the previous section, has to be programmed with dedicated programming tools. Second, programmers themselves are not a homogeneous group and require different software environments to be productive, depending on their skill set and the priorities imposed by the problem they want to solve. And third, the complexity and variety of data analysis and other programming tasks often require specialized tools at different stages.

As a consequence, we have to integrate separate software components into a heterogeneous software ecosystem, in order to enable different users perform complex data analysis tasks in a way that meets their skills and their performance requirements. Consider, for example, the ecosystem created around Apache Hadoop [309]. Since its initial release, a large number of software systems have been released, which either build upon Hadoop, or provided specialized services, e.g., Apache Hive [310], a distributed data warehouse; Apache Mahout [311], a distributed machine learning library; and Zookeeper [314], a coordination service for distributed applications. Later data analytics systems, e.g., Apache Spark [345] and Apache Flink [51], improve on Hadoop's capabilities, but maintain a high degree of compatibility. For example, they can interoperate with services such as the Hadoop Distributed File System (HDFS) [287], use common data formats such as Apache Avro [307] and Apache Parquet [312], and are even able to reuse user-defined functions (UDFs) written for Hadoop [59].

**Heterogeneous hardware.** The specialization of computing hardware directly leads to heterogeneous software environments. For each processor type, there are low-level programming languages or language constructs, which provide the necessary programming abstractions to match the processor design, and offer a high degree of control over the hardware. For example, on multi-core CPUs, pthreads [210] and OpenMP [65] are C language constructs to implement task parallelism and data parallelism, respectively; NVIDIA provides CUDA [211] as a dedicated programming environment for NVIDIA GPUs; similarly, Google provides TensorFlow [5] to program TPUs; and circuits for FPGAs are often implemented in the low-level hardware description languages Verilog [320] and VHDL [22]. Consequently, programmers have to employ multiple programming frameworks to develop applications which utilize specialized processors in heterogeneous computing systems.

**Diverse user requirements.** Programmers are a very diverse group with different skill sets, different motivations to write software, and different performance needs. We can distinguish between novice and expert programmers; amateurs and professionals; as well as regular programmers, for whom software development is the main professional activity, and casual programmers, such as data scientists or researchers, who write programs as a means to some other professional end [158]. These groups require different programming environments to work productively and will trade off application performance to a different degree. Expert programmers prefer programming languages that let them concisely express an abstract solution using high-level constructs, but also provide low-level control over the hardware, which is the key to predictable performance [246].

In contrast, casual programmers often choose high-level programming languages with rich libraries that allow them to quickly develop solutions to their problems [131]. For example, Python [254] is a popular programming language among data scientists because it includes powerful libraries for statistical analysis, machine learning, and data mining applications [140, 251]. These libraries provide so much value that data scientists choose Python even though it is about 50× slower than an equivalent, unoptimized C program, and 60000× slower than an optimized version [124].

High-level languages, such as Python, target traditional CPUs and do not support specialized processors out of the box. Libraries in these languages have to achieve two objectives. On the one hand, they have to provide the domain-specific functionality required by casual programmers, such as data scientists. On the other hand, they have to transparently support multiple processing cores and specialized processors, so that data scientists can benefit from improved application performance. Consequently, modern heterogeneous software environments bridge high-level and low-level programming languages, in order to resolve the tension between programmer productivity and application performance.

**Complexity of data analysis tasks.** The complexity and variety of today's data analysis tasks are additional factors which results in heterogeneous software environments. Commonly expressed by the phrase "one size does not fit all", the demands of a particular application scenario, such as analytical processing or stream processing, prompt a specific software implementation to include functionality that is of no use to other applications, or even degrades their performance [301]. Moreover, a data processing task may benefit from, or even require, the combination of multiple software systems in a complex data processing pipeline [10, 143]. For example, if the data is stored in multiple locations and/or formats, or if the system in which the data is stored does not support a specific analysis method, a complex data processing pipeline has to involve multiple systems in order to complete the task. In other cases, splitting a complex task into smaller steps and executing each step on a specialized system improves overall performance.

## 1.2. Thesis goals and contributions

As we have seen, heterogeneous hardware and software are an integral part of today's computing systems. This heterogeneity is both a challenge and an opportunity for query processing. On the one hand, system developers have to adapt query processing code to exploit the computing power of specialized processors, in order to deal with the increasing processing demands of big data. On the other hand, integrating multiple software systems to improve user productivity while retaining high performance, is the key to solve complex data processing tasks and provide access to many users, in order to further democratize data analysis. Therefore, this thesis has the following two goals.

*Thesis goals*

(1) Investigate how heterogeneous hardware and software impacts query processing.

(2) Develop techniques to achieve high query processing performance on heterogeneous hardware and software systems.

Since hardware and software heterogeneity is a very broad topic, we focus on three specific scenarios to reduce the scope of this thesis. In the following, we discuss the heterogeneous aspect of each scenario, develop the specific research goals and questions, and summarize our results and contributions.

### 1.2.1. Query processing on heterogeneous CPU/GPU systems

In the first scenario, we investigate the impact of heterogeneous processing architectures on query processing.

**Heterogeneous aspect.** A *heterogeneous CPU/GPU system* is a computing system which contains both CPUs and GPUs as processors. We call a query processing system which exploits both CPUs and GPUs for query processing a *heterogeneous query processing system.* A heterogeneous query processing system has to address three challenges in order to effectively exploit the processing power of heterogeneous CPU/GPU systems.

(1) CPUs and GPUs are optimized for different applications and therefore have different capabilities. It follows that some tasks of a query processing system may run faster on the CPU and other tasks may run faster on the GPU. A heterogeneous query processing system has to distribute the query processing workload on processors with different capabilities in a way that exploits the strengths of each processor and avoids their weaknesses.

(2) CPUs and GPUs have to be programmed differently. A major difference between both processors is how they employ multiple threads to solve a complex problem. On CPUs, threads typically work on independent subproblems, whereas on GPUs, groups of threads have to cooperate to achieve high performance. A heterogeneous query pro-

cessing system has to support implementations for multiple processors efficiently, i.e., without increasing development and maintenance costs.

(3) CPUs and GPUs can be integrated in different ways which affects the speed at which the GPU can access data in memory. Dedicated GPUs must transfer data from system main memory over a slow interconnect before they can process it. A heterogeneous query processing system has to overcome this data transfer bottleneck.

**Research goals.** In this part of the thesis, we study how heterogeneous query processing systems address these three challenges. Specifically, we focus on the following research goals.

*Research goals (first scenario)*

(1.1) Review and classify techniques to distribute a query processing workload on CPUs and GPUs, to program heterogeneous CPU/GPU systems efficiently, and to overcome the data transfer bottleneck.

(1.2) Classify heterogeneous query processing systems according to the techniques which they employ, and identify best practices and open research problems.

**Contributions.** In this part of the thesis, we make the following contributions.

(1) For our review, we conduct a survey of the academic literature on query processing on heterogeneous CPU/GPU systems. Our survey covers a diverse set of complete query processing systems, e.g., relational query processors, stream processing systems, hybrid analytical/transactional systems, and key value stores; as well as implementations which perform a specific query processing tasks, including joins, sorting, indexing, and spatio-temporal query processing.

(2) We develop a classification scheme to categorize techniques to distribute a query processing workload on CPUs and GPUs.

(3) Based on this classification scheme, and a review of the techniques to program heterogeneous CPU/GPU systems efficiently and to overcome the data transfer bottleneck, we categorize the surveyed query processing systems. This classification allows us to identify common strategies to overcome performance bottlenecks, as well as to identify query processing tasks which have not yet been investigated on heterogeneous CPU/GPU systems.

(4) We summarize the history of employing GPU hardware for query processing and identify GPU hardware which has not yet been broadly studied in the context of query processing.

**Key insights.** The key insight of our review is that heterogeneous CPU/GPU systems with dedicated and integrated GPUs require different strategies to distribute the workload on CPU and GPU in order to achieve high performance.

The major limiting factor of query processing performance of heterogeneous CPU/GPU systems with dedicated GPUs is the data transfer bottleneck between the CPU and the GPU. To mitigate the effect of this bottleneck, query processing systems targeting dedicated GPUs should offload specialized coarse-grained tasks to the GPU. This strategy largely precludes fine-grained cooperation between the CPU and the GPU because such cooperation relies on the frequent and low-latency exchange of data.

In contrast, heterogeneous CPU/GPU system with integrated GPUs are not constrained by the data transfer bottleneck. Instead, the CPU and the GPU in such systems are connected by a fast and cache-coherent on-chip fabric which enables both processors to modify shared data structures simultaneously. Query processing systems targeting integrated GPUs can divide the workload into fine-grained cooperative tasks and assign each task to a processor that best matches its processing characteristics.

**Basis of work.** The content described in this part of the thesis was published in ACM Computing Surveys Volume 55, Issue 1, in 2022 [266].

### 1.2.2. Operator variant tuning on heterogeneous processors

In the previous scenario, we examined the challenges of query processing on multiple heterogeneous processor architectures in a comprehensive fashion. In the second scenario, we drill down on a specific aspect of such a query processing system: its low-level operator implementation.

**Heterogeneous aspect.** To achieve high performance, the operator implementation of a query processing system has to be tailored to the specific processor it runs on. The developers of a query processing system have many degrees of freedom when designing its operator implementation. For example, they have to decide how to parallelize the workload and distribute it on multiple threads, which data structures to use, and whether to apply low-level optimizations, such as branch-free execution.

One approach to develop a heterogeneous query processing system is to include handwritten operator implementations for each targeted processor architecture. For example, GDB [116] contains two completely separate operator implementations, one written in CUDA [211] for NVIDIA GPUs, and one written in OpenMP [65] for multi-core CPUs. This approach incurs substantial development and maintenance costs as the number of targeted processor architectures, and programming frameworks we need to use, increases.

The developers have to manually tune a specialized operator implementation for each target architecture, and then they have to maintain this implementation as the query processing system evolves.

**Research goals.** In this part of the thesis, we aim to mitigate the costs of supporting multiple operator implementations in a heterogeneous query processing system, by letting the system automatically adapt its operator implementation to the processor it runs on. Specifically, we focus on the following research questions.

*Research questions (second scenario)*

(2.1) How sensitive are processors to changes in the operator implementations?

(2.2) How can a query processing systems learn fast operator implementations automatically, without manual tuning?

**Contributions and results.** Our first two contributions are an extensive experimental performance analysis of two common database operators, i.e., selection and hash aggregation, on multiple processor architectures, including multi-core CPUs, GPUs, and a Xeon Phi coprocessor. Even with these simple operations, a small number of implementation parameters yield a large search space, consisting of hundreds, or even thousands, of possible operator implementations. We find that the best operator implementation depends on the specific processor. Implementation heuristics derived from analyzing one processor are not necessarily transferable to other processors, even if they are based on similar microarchitectures by the same vendor. Furthermore, our analysis reveals that some processors are more difficult to optimize for, because any deviation from a specific set of implementation parameters incurs a significant performance penalty. In contrast, on other processors, many different implementations are competitively fast.

Our third and fourth contributions are two algorithms which enable a query processing system find fast operator implementations at runtime during query execution. The first algorithm extends micro adaptivity [258] with a search strategy to handle a large number of possible operator implementation candidates. The second algorithm extends a local search to handle performance plateaus and variations in operator runtime to overcome local optima in the search space. The key insight of our analysis of these two algorithms is that it is necessary to exploit information about the processor and the search space to reduce the search time for a fast variant.

**Basis of work.** The content described in this part of the thesis is based on two publications presented at ADMS 2015 [268] and at DaMoN 2019 [267].

### 1.2.3. Processing Java UDFs in a C++ environment

Whereas the previous two scenarios focused on the effect of hardware heterogeneity on query processing, in this scenario, we focus on software heterogeneity. Specifically, this scenario is motivated by an effort to extend Apache Spark [345] with the ability to process transactions, while retaining the familiar SparkSQL [20] user interface for analytical queries. To this end, we replace the exiting Spark execution engine with Wildfire [26], a hybrid transactional/analytical engine.

**Heterogeneous aspect.** Whereas Apache Spark is written in Scala [74], a programming language that is compiled to bytecode which runs on the Java Virtual Machine (JVM) [184], the Wildfire engine is written in C++, which is compiled to machine code that executes natively on the processor. This difference in programming language presents a challenge when executing SparkSQL queries with user-defined functions (UDFs) in Wildfire: How can we execute arbitrary code that is written in a JVM-based language inside a query engine that does not run on the JVM.

The JVM specification provides a standardized mechanism, the Java Native Interface (JNI) [234], to integrate with programs which are written in other programming languages. Through the JNI, a program can instantiate an embedded JVM and execute Java bytecode inside it. However, a JNI call is orders of magnitude slower than a simple function call in Java or in a compiled language such as C++. This overhead is significant if we have to use the JNI to invoke a Java-based UDF from C++ for every data tuple processed by a SparkSQL query.

**Research goals.** In this part of the thesis, we aim to improve the interoperability of UDF-based query processing systems which run on the JVM and those implemented in other programming languages. Specifically, we focus on the following research questions.

*Research questions (third scenario)*

(3.1) How can we mitigate the overhead of calling Java UDFs over JNI from C++ code for every data tuple?

(3.2) Can we bypass the JNI entirely and call Java UDFs directly from C++ code?

(3.3) How can a query processing system, which is written in a language that compiles to machine code such as C++, execute tuple-based Java UDFs efficiently and transparently to the user?

**Contributions and results.** We develop two approaches to reduce the overhead of calling tuple-based Java UDFs from C++. In the first approach, we transparently generate a strided execution wrapper to transform a UDF which processes a single tuple into a UDF which processes a batch of tuples. By amortizing the cost of a single JNI call over multiple input tuples, this approach matches the performance of executing the Java UDF inside the original JVM-based query processing system. In the second approach, we compile the bytecode of the Java UDF into machine code and link it directly with a C++ query processing engine. This approach bypasses the JNI entirely.

Comparing these two approaches shows that compiling a Java UDF into machine code is faster if the UDF is compute-intensive and does not create many Java objects. However, if the UDF does create many objects, including Java strings, it is faster to transform a tuple-based UDF into a batch UDF and execute it inside an embedded JVM.

**Basis of work.** The content described in this part of the thesis was published at ACM SoCC 2017 [269].

## 1.3. Impact of thesis contributions

**Publications of thesis contributions.** The primary results of this thesis are based on four publications, which have been presented at international conferences and workshops, or published in a journal.

- V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl. 2015. The operator variant selection problem on heterogeneous hardware. In *Proc. of ADMS@VLDB'15*, 1–12
- V. Rosenfeld, R. Mueller, P. Tözün, and F. Özcan. 2017. Processing Java UDFs in a C++ environment. In *Proc. of ACM SoCC'17*, 419–431. DOI: 10.1145/3127479.3132022
- V. Rosenfeld, S. Breß, S. Zeuch, T. Rabl, and V. Markl. 2019. Performance analysis and automatic tuning of hash aggregation on GPUs. In *Proc. of DaMoN'19*. DOI: 10.1145/3329785.3329922
- V. Rosenfeld, S. Breß, and V. Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Comput. Surv.*, 55, 1. DOI: 10.1145/3485126

**Open source contributions.** We have released the source code and experiments which make up our second research scenario, i.e., operator variant tuning on heterogeneous processors, under the Apache License 2.0.

- https://git.tu-berlin.de/viktor-rosenfeld/perseus. This repository contains our OpenCL implementations of the selection operator, as well a C++ implementation of the variant tuning framework to learn fast operators at runtime.

- https://git.tu-berlin.de/viktor-rosenfeld/gpu-hash-aggregation-analysis. This repository contains our OpenCL implementations of the hash aggregation operator, as well as a Python implementation of our local search algorithm to handle performance plateaus and runtime variation.

**Additional publications.** In addition, the author collaborated on the following two publications while working on this thesis.

- T. Behrens, V. Rosenfeld, J. Traub, S. Breß, and V. Markl. 2018. Efficient SIMD vectorization for hashing in OpenCL. in *Proc. of EDBT'18*, 489–492. DOI: 10.5441/002/edbt.2018.54

- C. Kotselidis, I. Komnios, O. Akrivopoulos, S. Bress, K. Doka, H. Mohammed, G. Mylonas, V. Spitadakis, D. Strimpel, J. Fumero, F. S. Zakkak, M. Papadimitriou, M. Xekalaki, N. Foutris, A. Stratikopoulos, N. Koziris, I. Konstantinou, I. Mytilinis, C. Bitsakos, C. Tsalidis, C. Tselios, N. Kanakis, C. Lutz, V. Rosenfeld, and V. Markl. 2020. Efficient compilation and execution of JVM-based data processing frameworks on heterogeneous co-processors. In *Proc. of DATE'20*, 175–179. DOI: 10.23919/DATE48585.2020.9116246

## 1.4. Structure of the thesis

The remainder of this thesis is structured as follows.

**Chapter 2.** In this chapter, we focus on query processing on heterogeneous hardware, specifically on systems that contain both CPUs and GPUs. We identify three key challenges that such heterogeneous systems present, i.e., that a query processing system has to distribute the workload on processors with different capabilities; that the implementation of the query processing system has to be adapted to both processors; and that the query processing system has to mitigate the slow connection between the CPU and GPU. We then conduct a survey of the academic literature to review, describe, and classify techniques which address these challenges. Based on our survey, we formulate performance guidelines for query processing on heterogeneous CPU/GPU systems.

**Chapter 3.** Next, we focus on a specific challenge of query processing on heterogeneous hardware, i.e., the need to adapt the query processing code to different processors. We show that even a few implementation parameters create a large optimization space containing thousands of different implementation variants and that GPUs are especially sensitive to the selection of optimal implementation parameters. To reduce the cost of manual tuning and maintaining multiple specific implementations in a code base, we develop two algorithms to find fast implementations at runtime.

**Chapter 4.** In this chapter, we focus on query processing in a heterogeneous software environment, specifically the integration of query processing systems written in a JVM-based language and systems written in a compiled language. We develop two approaches, *strided execution inside an embedded JVM* and *JIT-compilation of the Java bytecode*, to execute Java-based UDFs containing arbitrary code inside a query engine written in C++. Our approaches mitigate or eliminate the large runtime overheads typically associated with calling Java code from C++ for each processed tuple.

**Chapter 5.** We conclude this thesis by discussing open research questions for future work.

**Appendices.** Additionally, in Appendix A, we discuss the scheduling decisions of selected heterogeneous query processing systems, which we surveyed in Chapter 2. In Appendix B, we provide comprehensive data from experiments performed in Chapter 3.

# 2 Query processing on heterogeneous CPU/GPU systems

## 2.1. Problem statement

To process the vast amount of data generated by businesses and scientific communities, data processing engines need to take advantage of heterogeneous processor architectures integrated in modern computing systems [3]. Indeed, the research community has embraced data processing on heterogeneous processors such as graphics processing units (GPUs) [45, 116, 286], field-programmable gate arrays (FPGAs) [79, 134, 305], and asymmetric multi-core processors (AMPs) [204].

In this chapter, we conduct a comprehensive review of the impact of heterogeneous CPU/GPU systems on query processing, i.e., computing systems which contain both multi-core CPUs and GPUs as processors. We focus on GPUs because they have been one of the most successful specialized processors in the last two decades. Not only are GPUs ubiquitous in consumer electronics, e.g., mobile phones and laptops [21, 255], but they are also commonly used in cloud environments [15, 92, 202], super computer installations [321], and embedded industry applications [290]. Compared to AMPs, which are geared towards the mobile market, GPUs provide significant processing power. Compared to FPGAs, they are much simpler to program [79].

Just as hardware is becoming increasingly diverse, so are the applications that process data and the demands these applications place on query processing systems. Traditional database systems process queries that are based on relational algebra [23]. Relational queries consist of *selections* and *projections* to filter and transform data; *joins* to combine multiple data sources; *groupings* and *aggregations* to categorize and summarize data; as well as set operations. Online analytical processing (OLAP) systems perform interactive, analytical queries over large, multi-dimensional data sets to support business decisions [135]. OLAP queries are dominated by joins and grouped aggregations. Hybrid transactional/analytical (HTAP) systems integrate transactional and analytical workloads in one system to support analytical queries over up-to-date data. Spatial query systems operate on data that represents geometrical primitives, i.e., points, lines, and polygons, as well as non-spatial, relational data [75]. Whereas the previously described systems operate on stored data, stream processing systems [1, 51] operate on continuously updating streams of data and order them according to a time domain [63]. Stream queries process data in terms of *windows*, which define the portions of the input that are considered during query execution.

All of these query processing operations are data-intensive but they are constrained by different bottlenecks. Stateless operations, e.g., selections, projections and ungrouped aggregations, are bandwidth-bound. In contrast, stateful operations, e.g., joins and grouped aggregations, are typically latency-bound. Many operations are also compute-intensive, e.g., sorting. Spatial predicates, e.g., *contains* or *overlaps*, are especially compute-intensive because they operate on arbitrary geometric shapes and perform many comparisons.

GPUs provide many opportunities to speed up data-intensive, latency-critical, or compute-intensive query processing tasks. However, the differences between CPU and GPU architectures, and the way GPUs are integrated into a computing system, also present significant challenges.

First, CPUs and GPUs make different performance tradeoffs. Whereas CPUs focus on single-thread latency, GPUs are optimized for data-parallel and throughput-oriented applications. To use each processor effectively, a heterogeneous query processing system has to distribute the workload in a way that exploits their different capabilities and keeps both CPUs and GPUs occupied.

Second, since CPUs and GPUs are based on different hardware architectures, we have to adapt the implementation of query processing algorithms and operators to each processor. However, supporting CPUs and GPUs in separate code bases increases both development and maintenance costs. To reduce its implementation complexity, a heterogeneous query processing system should integrate both code bases and abstract hardware-specific differences.

And third, high-performance GPUs are dedicated devices, which are connected to the CPU over a comparatively slow system bus. To realize any potential speedup from processing data on such a GPU, a heterogeneous query processing system must overcome this bottleneck.

## 2.2. Contributions

In this chapter, we review and describe techniques that address the challenges of query processing on heterogeneous CPU/GPU systems, and explore how these challenges and techniques interact with each other. To this end, we conduct a survey of the relevant academic literature. Based on our analysis, we identify the main trends in heterogeneous query processing and open research problems. Specifically, we make the following contributions.

(1) We summarize advances in GPU hardware and their use in query processing. We also identify GPU hardware that are not yet broadly studied in data processing research (Section 2.4).

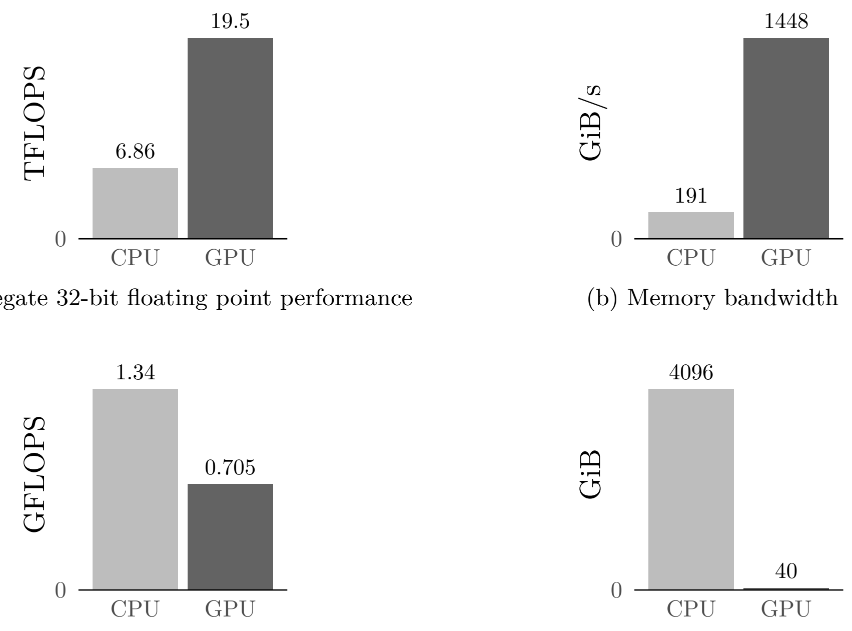
(2) We develop a classification scheme to categorize the distribution of query processing tasks on heterogeneous CPU/GPU systems (Section 2.5).

(3) We review techniques for reducing the implementation complexity of heterogeneous query processing (Section 2.6) and for mitigating the data transfer bottleneck (Section 2.7).

(4) Based on our classification and review of these techniques, we categorize query processing systems on heterogeneous CPU/GPU systems. We cover a diverse set of query processing systems, including relational query processing, stream processing, and key-value stores; as well as specific query processing tasks, e.g., join processing, sorting, index operations, and spatio-temporal queries (Section 2.8).

The key insight of our analysis is that *integrated and dedicated GPUs have to be treated as different classes of heterogeneous query processing systems.* Dedicated GPUs remain constrained by slow data transfers and benefit from scheduling coarse-grained tasks. In contrast, integrated GPUs benefit from scheduling fine-grained tasks on the most suitable processor since CPUs and GPUs can cooperate efficiently.

Before we continue with the results of our survey, we describe the key engineering tradeoffs of CPUs and GPUs and highlight important differences between these processor architectures.



(a) Aggregate 32-bit floating point performance

(b) Memory bandwidth

(c) Serial 32-bit floating point performance

(d) Memory size

Figure 2.1.: Performance comparison between a CPU (AMD EPYC 7702P) and a GPU (NVIDIA Ampere A100). The GPU has 2.8× more aggregate compute performance than the CPU and 7.6× faster memory bandwidth. However, the CPU has 1.9× more serial performance and can access two orders of magnitude more memory.

## 2.3. Processor architectures

In this section, we describe the architectures of CPUs and GPUs as well as different strategies for integrating GPUs in a heterogeneous system. We also briefly introduce the traditional GPU programming model and describe differences to CPU programming.

GPUs are typically characterized by high computational power and memory bandwidth, especially compared to CPUs. For example, in Figure 2.1, we compare two recent high performance processors, the AMD EPYC 7702P CPU [302] and the NVIDIA Ampere A100 GPU [217]. The Ampere A100 has 2.8× more 32-bit floating point performance and 7.6× higher memory bandwidth than the EPYC 7702P. These performance advantages of GPUs over CPUs are often cited as a major motivation to use GPUs for query processing in database research [46, 80, 94, 116, 119, 120, 149, 298].

Yet it is too simplistic to reduce GPUs to these performance advantages. In fact, when we focus on other metrics, CPUs outperform GPUs. For example, as Figure 2.1 also shows, the EPYC 7702P is 1.9× faster than the Ampere A100 when executing *serial*

Table 2.1.: Comparison of processor properties of an AMD EPYC 7702P CPU and an NVIDIA Ampere A100 GPU.

| | AMD EPYC 7702P | NVIDIA Ampere A100 |
|---|---:|---:|
| Release year | 2019 | 2020 |
| Transistors | 38.7 billion | 54.2 billion |
| Thermal design power | 200 W | 400 W |
| Independent cores / SMs | 64 cores | 108 SMs |
| Concurrent threads | 2 / core | 128 / SM |
| Maximum frequency | 3.35 GHz | 1.41 GHz |
| Register file size | 6.4 KiB / core | 256 KiB / SM |
| L1 data cache | 32 KiB / core | 192 KiB / SM |
| L2 cache | 512 KiB / core | — |
| Last-level cache | 16x 16 MiB | 40 MiB |
| Memory interface | 8x 64-bit DDR4-3200 | 10x 512 bit HBM2 |
| Memory clock | 1.6 GHz | 1.215 GHz |

32-bit floating point code. It can also directly access two orders of magnitude more memory.

Instead, the different performance characteristics of CPUs and GPUs indicate that they are optimized for different usage scenarios. Both processor types are constrained by the power wall, i.e., the requirement to keep their power consumption, and the resulting heat dissipation, inside a manageable level [39]. To achieve high performance under these constraints, the architectures of CPUs and GPUs are based on different design trade-offs, which are driven by concrete application requirements. Concretely, CPUs and GPUs differ in how they allocate transistors to implement logic, cache, and control functionality. This specialization implies that the choice of the best processor depends on the type of the problem. We will show in later sections how this specialization is relevant to data processing.

In the following, we describe the design considerations that motivate the architecture of CPUs and GPUs in more detail. In Table 2.1, we contrast a number of processor properties of the EPYC 7702P and the Ampere A100 for reference.

### 2.3.1. Conventional CPUs

The primary optimization goal of conventional CPUs is their *serial performance* [39]. Historically, manufacturers relied on *Dennard scaling* [67] to increase processor frequency and thus processing speed. Dennard scaling relates the size of transistors with their operating frequency and voltage. Specifically, the power consumption of a processor is proportional to $CV^2f$, where the capacitive load $C$ is a function of the number of transistors, $V$ is the operating voltage, and $f$ the processor frequency [125]. As transistors shrink, more of them can be integrated on a die and their operating frequency increases. To keep power consumption constant, the operating voltage has to be reduced. This effect alone has led to a $100\times$ performance increase of recent CPUs compared to early CPUs [39].

Processor vendors have also implemented microarchitecture advances that extract *implicit instruction level parallelism* (ILP) from the instruction stream, to improve serial performance. The core technique of these advances is the processor pipeline which overlaps the execution stages of different instructions. Ideally, the pipeline is always full and the processor can issue and complete one instruction per cycle per functional unit. However, the pipeline stalls when instructions are dependent on each other, or when the processor has to wait on memory access.

Modern CPUs implement a number of techniques to keep the pipeline from stalling and increase ILP [125]. For example, *branch prediction* continues to fetch and decode instructions of the predicted branch in the instruction stream, which keeps the early stages of the pipeline full. *Speculative execution* also executes the instructions of predicted branches and only discards their results, if the prediction later proves to be incorrect. *Out-of-order execution* reorders the instruction stream to reduce the impact of dependent instructions and memory stalls. Together with Dennard scaling, these microarchitecture advances have increased scalar performance by 52% per year between 1986 and 2003 [124].

The exploitation of ILP is limited by the performance of the memory system [125]. Data references stall the processor pipeline if the processor cannot find independent instructions to execute. The length of the stall depends on the memory latency and the number of concurrent memory accesses that can be satisfied by the available memory bandwidth. Unfortunately, the rate of improvement of memory performance has lagged processor performance over time, both for latency and bandwidth [242]. Whereas early CPUs could access memory in a single clock cycle, they now have to wait hundreds of cycles [39]. In typical programs, especially in those that depend on integer performance [125], there is not enough instruction level parallelism available to overcome this access latency [332]. To reduce memory access latency, modern CPUs use a substantial amount of their transistor budget, up to 50%, to implement large cache hierarchies [39]. These caches allow the CPU to exploit temporal and spatial data access locality. However, even with perfect caches, the performance of data-intensive applications is limited by memory access due to compulsory cache misses when loading previously unseen data [338].

The processes that drove performance increases in the past no longer work. On the one hand, the increase in processor frequencies has flattened since 2003 [304]. Due to physical limitations, manufacturers cannot further reduce operating voltages without compromising reliability. Thus, they cannot increase the operating frequency without excessive power consumption and heat dissipation [35]. On the other hand, the microarchitecture advances to increase ILP are not energy-efficient because their implementation requires an increasing amount of the processor's transistor budget [100]. The scalar performance increase of subsequent microarchitectures is proportional to the square root of the number of transistors [250], a phenomenon which has been called *Pollack's rule* [39]. Consequently, scalar performance has slowed to 23% per year between 2003 and 2011, to 12% per year between 2011 and 2015, and to just 3.5% per year after 2015 [124].

Since scalar performance is no longer increasing, manufacturers have turned to increase throughput, by exploiting explicit data parallelism. For example, chip multiprocessors, or *multi-core CPUs*, use their transistor budget to integrate multiple processor cores on a single die. *Simultaneous multi-threading* (SMT) enables multiple independent threads to utilize different execution units of a core which explicitly increases ILP. *Vectorized SIMD instructions* work on multiple data items in a single cycle. These developments mean that multi-core CPUs are becoming more similar to GPUs, which we discuss next.

### 2.3.2. Dedicated GPUs

GPUs were originally developed as special-purpose processors to accelerate graphics rendering in 3D games. The generic computing capabilities of GPUs are an artifact of making the graphics rendering pipeline more flexible to better support a greater variety of 3D games [182] (see Section 2.4.3 for details). Consequently, GPUs are optimized for *throughput application* of which graphics rendering is a prime example. Throughput applications are characterized by three interrelated characteristics, which influence the design trade-offs of GPU hardware: a high degree of data parallelism, latency tolerance, and high demands on memory bandwidth [182].

Instead of extracting the implicit instruction level parallelism from an instruction stream, GPUs rely on *explicit data parallelism* to keep processing cores busy. Consequently, GPUs use their transistor budget to implement many simple processing cores instead of implementing fewer complex processing cores, as CPUs do. For example, while GPUs are pipelined and support super-scalar execution [225], they do not utilize branch prediction or speculative execution. As a result, the processing performance of

GPUs scales (almost) linearly with the transistor budget, whereas the microarchitectural enhancements of CPUs scale only proportionally to the square root of the number of transistors [250].

Since GPUs are optimized for aggregate throughput instead of serial performance, the latency of processing an individual data item is less important. This latency tolerance has two important effects on the hardware design. First, it allows us to reduce the processing frequency and use more transistors to implement processing cores within a given power budget. Second, instead of reducing the latency of an individual data item through caches and microarchitectural advances, the latency is hidden by processing other data items.

When we reduce the processing frequency, we can keep the aggregate throughput constant by increasing the number of processing cores accordingly. Since processing frequency and capacitive load of the processor have a linear relationship with power, the overall power consumption is constant in this case. However, when we lower the processing frequency, we can also lower the operating voltage of the processor which has a quadratic relationship with power consumption. Therefore, we can increase the number of processing cores even more while staying within the power budget of the processor.

To support latency hiding, the GPU hardware allows for a massive oversubscription of threads. For example, each streaming multiprocessor (SM) of an Ampere A100 GPU can execute four independent warps at a time [217]. (A warp is a group of threads that execute the same instruction. See Section 2.3.4 for details.) At the same time, each SM can manage 64 different warps that await execution. At each cycle, the SM can switch between active and inactive warps without overhead. To support these many threads, GPUs contain very large register files which are orders of magnitude larger than the register files of CPUs. Compared to CPUs, the GPU cache hierarchy also places more emphasis on large L1 caches, which are close to the processing cores. In contrast, the shared last-level cache is smaller on GPUs than on CPUs. As Table 2.1 shows, the Ampere A100 has overall 8× fewer cache resources available per SM than the EPYC 7702P has per core. The streaming data access pattern of GPU graphics workloads exhibits relatively little data reuse and therefore caches are less useful [182].

The memory subsystem of GPUs is optimized for high memory bandwidth, in order to feed input data to the large number of processing cores. GPUs typically use more independent memory controllers than CPUs and therefore have a wider memory data bus. The memory bus is also clocked faster, up to 7 GHz for GDDR6 memory [226]. High-performance GPUs use three-dimensional stacked memory, which is packaged together

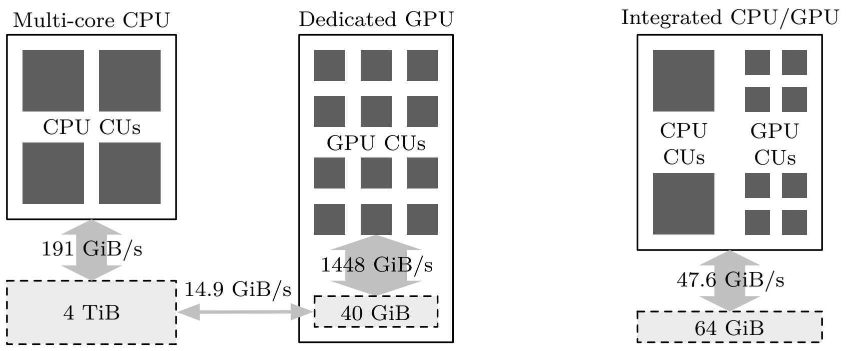


Figure 2.2.: Memory size and bandwidth for a multi-core CPU and a dedicated GPU connected over PCIe 3.0 (left) and an integrated CPU/GPU processor (right).

with the GPU processor die in a single package [224]. Stacked memory is addressed through an ultra-wide data bus. For example, the Ampere A100 uses ten 512-bit memory controllers which results in a overall bus width of 5120 bits [217]. This is an order of magnitude wider than the 8x 64-bit bus width of the EPYC 7702P.

### 2.3.3. GPU integration

Traditionally, GPUs are dedicated processors which are accessed over a system bus. We show such an arrangement Figure 2.2 (left side). In a typical system, the CPU and the GPU are connected by a PCI Express (PCIe) 3.0 bus [13] which offers up to 14.9 GiB/s in theoretical bandwidth. This connection is an order of magnitude slower than the main memory bandwidth of the CPU and two orders of magnitude slower than GPU memory. It therefore represents a significant performance bottleneck [99, 189, 343]. Furthermore, the separate CPU and GPU memory spaces are not coherent. Consequently, shared data structures have to be synchronized manually which increases implementation complexity [189]. Recent GPU architectures reduce this problem somewhat. For example, AMD GPUs use PCIe atomics [13] to synchronize execution between CPUs and GPUs [7]. NVIDIA GPU support software-assisted memory coherence and system-wide atomics through page faults and automatic page migrations but this mechanism causes runtime overheads [275]. On IBM Power9 systems, NVIDIA GPUs can be connected to the CPU over NVLink 2.0 [225], which supports cache coherence and atomic operations between CPU and GPU memory in hardware and eliminates these overheads [189]. NVLink 2.0 is also 5× faster than PCIe 3.0 which reduces the effects of data transfer bottleneck [189].

Alternatively, CPUs and GPUs can also be integrated on the same die to form an integrated CPU/GPU processor. We show such an arrangement in Figure 2.2 (right

Table 2.2.: Processor properties of the AMD Ryzen 7 Pro 4750G integrated CPU/GPU.

| | CPU | GPU |
|---|---:|---:|
| Independent cores / CUs | 8 cores | 8 CUs |
| Maximum frequency | 4.4 GHz | 2.1 GHz |
| Aggregate FP32 performance | 1.13 TFLOPS | 2.15 TFLOPS |
| Serial FP32 performance | 1.76 GFLOPS | 1.05 GFLOPS |
| Last-level cache | 8 MiB | 1 MiB |
| Release year | | 2020 |
| Transistors | | 9.8 billion |
| Thermal design power | | 65W |
| Memory interface | | 2x 64-bit DDR4-3200 |
| Memory size & speed | | 64 GiB @ 47.6 GiB/s |

side). In Table 2.2, we summarize the properties of the AMD Ryzen 7 Pro 4750G [21] as an example (cf. the properties of a conventional CPU and dedicated GPU in Table 2.1).

Compared to accessing a dedicated GPU over a system bus, the integration on a single processor die has a number of advantages. (1) CPU and GPU access main memory directly. Consequently, the memory space for the GPU is larger than on dedicated GPUs and there is no need to copy data [263]. (2) Memory accesses by the CPU and GPU are coherent. Both can simultaneously access and modify shared data structures, e.g., through system-wide atomics, which allows for more fine-grained cooperation and simpler implementations [40]. (3) Because the CPU and GPU are more closely connected, integrated CPU/GPU processors have a better power/performance ratio than dedicated GPUs [295].

However, integrated CPU/GPU processors also have a number of disadvantages compared to dedicated GPUs. The lower memory bandwidth leads to more memory stalls [121] and reduces the performance of memory-bound kernels [295]. Most importantly, the CPU and GPU have to share common resources and compete for main memory access. The high bandwidth required by the GPU can easily saturate the shared memory subsystem and increase the memory access latency of the CPU [150]. Furthermore, the entire processor is constrained by a common power envelope, and, therefore, integrated CPU/GPU processors have less raw performance than combining a dedicated GPU with a conventional CPU in a system.

In Table 2.3, we summarize advantages and disadvantages of integrating GPUs over a system bus or on the same processor die.

Table 2.3.: Comparison of advantages (in *italics*) and disadvantages of accessing GPUs over a system bus or integrating them on the same processor die.

| | System bus | Processor die |
|---|---|---|
| Data transfer bandwidth | Slow | *No transfer* |
| GPU memory capacity | Small | *Medium* |
| GPU memory bandwidth | *High* | Medium |
| Raw performance | *High* | Medium |
| Power/performance ratio | Worse | *Better* |
| Data synchronization | Manual | *Automatic* |

### 2.3.4. GPU programming model

GPUs are programmed in a specialized programming model that allows programmers to formulate a parallel program in a scalable way [211]. The programming model represents GPU hardware as an abstract parallel processor. It defines how a parallel program is executed on the processor and how the workload is partitioned to achieve scalable parallelism. As a result, GPU programming differs from CPU programming in a number of important ways. Two popular implementations of this programming model are CUDA [211, 216] and OpenCL [300, 318]. In the following, we primarily use OpenCL terminology to describe the programming model but also state equivalent CUDA terms.

#### 2.3.4.1. Abstract parallel programming model

OpenCL represents parallel processors, e.g., multi-core CPUs or GPUs, as *computational devices* consisting of *compute units* (CUs). Often, but not always, these compute units map to actual hardware entities. For example, on NVIDIA GPUs, each compute unit maps to a streaming multiprocessor. On simultaneously multithreaded CPUs, a compute unit represents a logical CPU core.

An OpenCL program is divided into *host code* and *device code*. The host code executes in a single thread on the host CPU. It is responsible for coordinating operations on the device, e.g., initiating the execution of device code and transferring data between separate memory spaces. The device code executes in parallel on the OpenCL device. It consists of *kernels* which are scalar functions, expressing the operations on a single datum of a data-parallel task. Conceptually, a kernel contains the operations of a single iteration of a parallelized loop.

When launching a kernel, the programmer specifies a hierarchy of independent kernel instances that execute on the device. Each kernel instance is called a *work item* (or a thread in CUDA). Individual work items are arranged into *work groups* (called thread block in CUDA). All work items of a kernel invocation make up the *nd-range* of the kernel (called grid in CUDA). The work items of a single work group can cooperate with each other through special instructions, fast barrier synchronizations, and a very fast shared memory space called *local memory*. The last two hardware features enable the work items of a work group to process a datum, store the result in a shared cache, and wait until the other work items have finished their computations before accessing their results. In contrast, work items from different work groups execute completely independently. To force a synchronization between all work items, the kernel must terminate and the programmer must launch a new kernel with a new nd-range of work items.

#### 2.3.4.2. Scalable parallelism

It is through this two-tiered hierarchy of work items and work groups that the OpenCL and CUDA programming models support scalable parallelism [211]. Using the programming model, a programmer must partition a problem into two levels. The first level, i.e., the individual work groups, work on coarse-grained subproblems which can be solved independently in parallel. Each work group executes on a dedicated compute unit. Multiple work groups can execute on different compute units in parallel or on the same compute unit sequentially. The second level, i.e., the work items within a single work group, work on fine-grained subproblems which can be solved cooperatively in parallel.

The GPU hardware supports this fine-grained thread and data parallelism through fast barrier synchronization, access to shared local memory, lightweight thread creation, and zero-overhead scheduling. Additionally, independent nd-ranges can execute concurrently given sufficient hardware resources. This concurrent execution allows for coarse-grained task parallelism.

#### 2.3.4.3. Differences to CPU programming

A major difference between programming on GPUs and CPUs are the number of running threads, and how these threads work together. In general, on CPUs, comparatively few threads operate independently on coarse-grained subproblems. Specifically, on multi-core CPUs, each CPU core typically executes a single hardware thread that consumes an independent partition of the data [175]. Although threads running on different CPU cores can communicate with each other, they must avoid performance pitfalls caused by accessing shared or nearby data, e.g., false sharing. In contrast, GPUs execute many thousands of hardware threads to hide the latency of individual operations. Moreover, the threads that correspond to work items of the same work group have to cooperate with each other to achieve peak performance. To this end, these work items can access a fast local memory to exchange data and synchronize through fast barrier synchronizations. A classic example of a data processing task that showcases this cooperation to achieve high throughput is parallel reduction on GPUs [187].

A second important difference between GPU and CPU programming is the *Single Instruction, Multiple Thread (SIMT)* [182] execution model. In the SIMT execution model, a number of work items share the instruction pointer and execute a common instruction. For example, on NVIDIA GPUs, 32 threads make up a *warp* and execute the same instruction at the same time. Similarly, on AMD GPUs, 64 work items make up a *wavefront*. The SIMT execution model is similar to the *Single Instruction, Multiple Data (SIMD)* execution model supported by CPU vector instructions. However, a crucial difference is that GPU kernels are written as scalar functions, independent of the SIMD instruction width of the processor. The GPU hardware also takes care of masking results when different work items follow separate branches in the kernel code.

Nevertheless, to maximize performance, programmers still have to take hardware details, such as the warp size, into account. For example, programmers should avoid diverging code paths for the threads inside a warp [89, 119]; utilize warp-level primitives [181], e.g., warp-level reductions [217] or ballot and shuffle instructions [89, 177, 289]; and let the threads of a warp access adjacent global memory locations, so that the GPU can coalesce these accesses into as few memory transactions as possible [19, 213].

## 2.4. Evolution of query processing on GPUs

In this section, we describe how query processing on GPUs developed over time. Figure 2.3 shows a timeline of important technological advances and how they influenced data processing research. In the following, we discuss these relationships in more detail.

### 2.4.1. Fixed-function graphics pipelines

Modern programmable GPUs have evolved incrementally from fixed-function graphics pipelines [182]. These fixed-function pipelines split the image rendering process into different hardware-accelerated stages. For example, a vertex processor projects the vertices of a three-dimensional scene onto the two-dimensional screen space. The transformed vertices are assembled into primitives such as triangles and rasterized into pixel fragments. A fragment processor then determines the color of each pixel fragment based on textures and screen content samples.

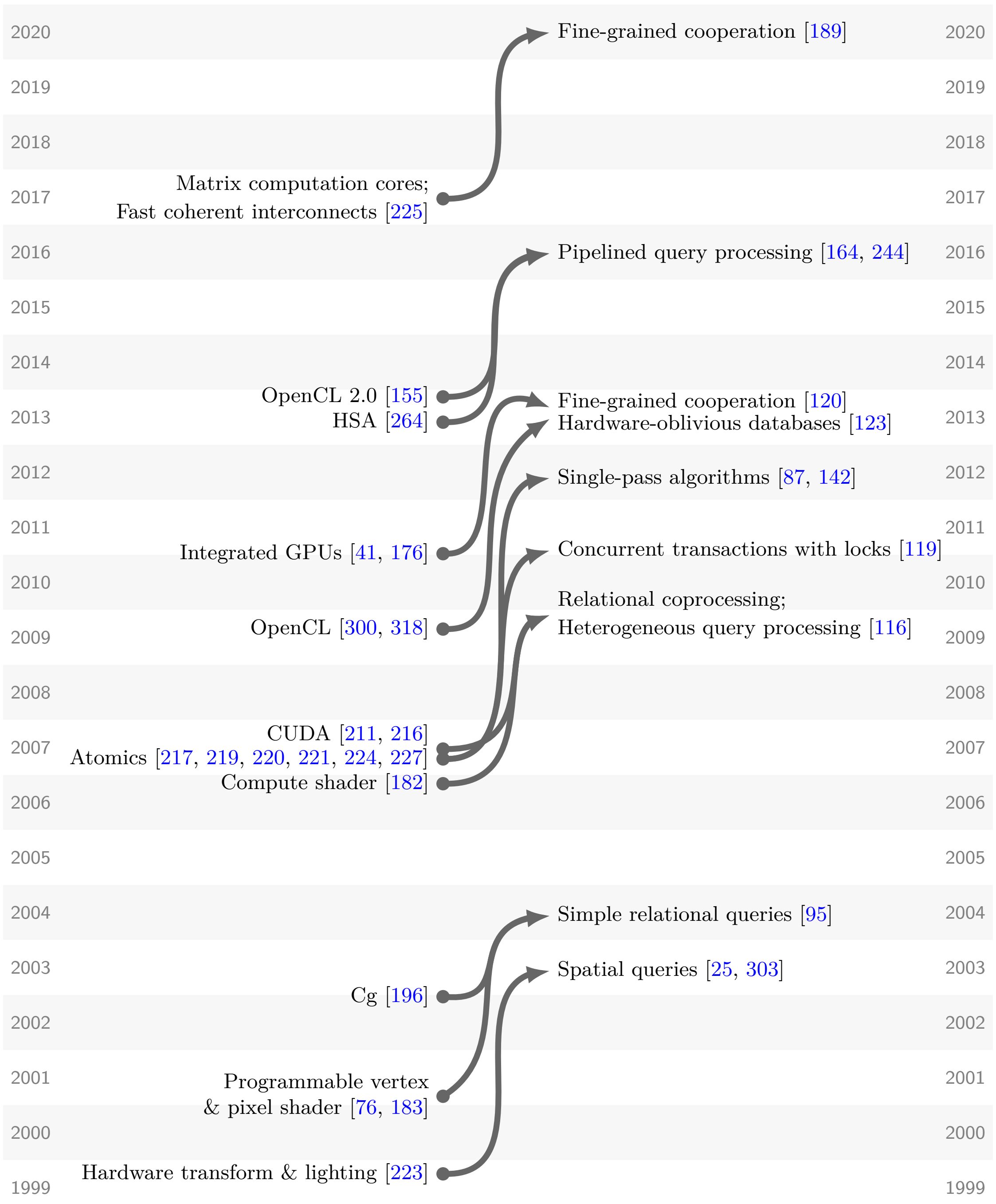

2020
2019
2018
2017
2016
2015
2014
2013
2012
2011
2010
2009
2008
2007
2006
2005
2004
2003
2002
2001
2000
1999
Fine-grained cooperation [189]
Matrix computation cores; Fast coherent interconnects [225]
Pipelined query processing [164, 244]
OpenCL 2.0 [155]
HSA [264]
Fine-grained cooperation [120]
Hardware-oblivious databases [123]
Single-pass algorithms [87, 142]
Integrated GPUs [41, 176]
Concurrent transactions with locks [119]
Relational coprocessing; Heterogeneous query processing [116]
OpenCL [300, 318]
CUDA [211, 216]
Atomics [217, 219, 220, 221, 224, 227]
Compute shader [182]
Simple relational queries [95]
Spatial queries [25, 303]
Cg [196]
Programmable vertex & pixel shader [76, 183]
Hardware transform & lighting [223]

Figure 2.3.: Timeline of GPU technology advances (left) and their influence on query processing research (right).

Even though early GPUs were not programmable, they offered enough functionality and computational power to accelerate spatial selections and joins [303]. Notably, a proof-of-concept implementation was integrated with Oracle 9, as an early example of heterogeneous execution on CPUs and GPUs [25].

### 2.4.2. Programmable graphics pipelines

To improve the image rendering capabilities of their GPUs, ATI and NVIDIA introduced programmable vertex and fragment processors [76, 183]. While limited at first, these processors soon supported large programs and dynamic control flow [348]. At the same time, high-level programming languages, such as *C for graphics* (Cg) [196], were developed with the explicit purpose to support general purpose computing on GPUs. However, algorithms still had to be expressed in terms of the image rendering process [112]. For example, data was stored in textures, i.e., two-dimensional arrays containing four-component pixel color vectors. To trigger any computation, a quadrilateral had to be rendered. Furthermore, programmable fragment processors lacked integer arithmetic and could not perform scatters, i.e., writes to random memory locations [95].

Govindaraju et al. repurposed specific hardware features of the fragment processor to perform general data processing tasks. For example, multi-attribute predicate evaluations were implemented with the depth and stencil tests, which are used to quickly discard pixel fragments [95]; comparator mapping and value comparisons in sorting networks were built on texture mapping and color blending hardware [94, 96]; and ungrouped aggregations were implemented with occlusion queries [95]. With these primitives, Govindaraju et al. implemented single table relational query processing [95], quantile and frequency estimation on data streams [96], and external sorting [94] on GPUs.

### 2.4.3. General purpose parallel processing

The utilization of the vertex and fragment processor depends on the specific graphics processing task [182]. Tasks which do not strike the correct balance leave processing resources idle. To counter this inefficiency, ATI developed a unified vertex and fragment processor for the XBox 360 [17]. Subsequently, NVIDIA introduced the Tesla architecture as a unified general purpose parallel processor [182]. The Tesla architecture was the first to support CUDA (Compute Unified Device Architecture) [211, 216], which is a parallel programming model to formulate computation tasks without expressing them in terms of the image rendering process. CUDA exposes hardware features that are not accessible in graphics-based programming APIs and provides a significant performance boost [113]. For example, individual threads can cooperate and have access to fast, on-chip memory.

With the availability of CUDA, a number of GPU-optimized, parallel data processing primitives were devised, e.g., scan [113, 281]; gather and scatter [115]; map, split, and sort [118]; and reduce and filter [116]. Based on these primitives, He et al. implemented access methods and relational operators in GDB, a fully-functional relational query processor on the GPU [116]. GDB also included a CPU implementation and supported relational query processing on heterogeneous CPU/GPU systems.

### 2.4.4. Fast atomic operations

The very first general purpose GPUs did not support atomic read/modify/write operations [219]. This limitation precluded data-parallel operator implementations that rely on locks, e.g., single-pass selections or joins. Consequently, these operations were implemented using lock-free algorithms that relied on multiple passes to decouple the computation of write positions and writing results [113, 116, 118, 343] These algorithms read their inputs and evaluate the selection predicate or join condition multiple times, i.e., they are neither memory nor work-efficient.

On the first GPUs that supported them, atomic operations on the same address were slow and presented a significant bottleneck [227]. However, hardware support for atomic operations has been regularly improved in subsequent generations [217, 221, 224, 227, 228]. Nowadays, they are orders of magnitude faster. Atomics enable memory- and work-efficient single-pass algorithms [87, 120, 142] as well as lock-based concurrency control [119]. System-wide atomics [40, 225] also allow CPUs and GPUs to access shared data structures simultaneously, which enables fine-grained cooperation between both processors [120, 189].

### 2.4.5. Parallel processing for heterogeneous systems

CUDA [211, 216] is a proprietary framework that only supports NVIDIA GPUs. In contrast, OpenCL [300, 318] is a parallel programming standard that supports GPUs from multiple vendors and also other parallel processors, e.g., multi-core CPUs or FPGAs. OpenCL offers *functional portability*, i.e., the same code can run on any OpenCL-capable processor.

Functional portability and wide-ranging hardware support reduce the implementation complexity of heterogeneous query processing systems. For example, *hardware-oblivious databases*, e.g., Ocelot [123], contain a single operator implementation in OpenCL that runs on CPUs and GPUs. OpenCL is also a popular compilation target for heterogeneous query processing systems that employ query compilation, e.g., Voodoo [248] and Hawk [46].

As we describe in Section 2.3.4, the programming models of CUDA and OpenCL treat GPUs as coprocessors, which are controlled by a host CPU. In contrast, the Heterogeneous System Architecture (HSA) [264] treats CPUs and GPUs types as first class processors. HSA is a programming platform for heterogeneous systems with a shared coherent memory space, e.g., integrated GPUs. It is particularly suited to implement fine-grained cooperation on these processors [205].

CUDA, OpenCL, and HSA are low-level programming environments for heterogeneous CPU/GPU systems. Alternatively, developers can program heterogeneous systems in a high-level language, which uses LLVM [174] during compilation. LLVM is a compiler framework built around a machine-independent intermediate representation which can be lowered to machine code for various CPU and GPU architectures [185]. Indeed, AMD and NVIDIA use LLVM internally in their GPU compiler infrastructure [8, 215].

### 2.4.6. Integrated GPUs

Traditionally, GPUs are dedicated devices, which are connected to the system over a system bus. Over time, transistor size scaling has enabled vendors to integrate CPUs, GPUs, and other components on a single processor die [41, 176]. We discuss the architectural differences between dedicated and integrated GPUs in detail in Section 2.3.3.

### 2.4.7. Inter-kernel communication

To improve utilization, GPUs can execute multiple kernels at the same time [227]. However, the traditional GPU programming model described in Section 2.3.4 assumes that GPU kernels run in isolation and do not communicate with each other. Recent programming frameworks lift this limitation and allow for inter-kernel communication. For example, OpenCL 2.0 [155] supports *pipes* to exchange data between kernels running on the same processor. HSA supports *signals* to synchronize the execution of concurrent kernels on different processors [205]. Both OpenCL pipes and HSA signals have been used to implement in-cache pipelined query processing on GPUs [164, 244], similarly to vectorization on CPUs [38].

### 2.4.8. Fast coherent interconnects

Dedicated GPUs are typically connected over a comparatively slow system bus, e.g., PCIe 3.0. This connection represents a major bottleneck in heterogeneous query processing systems. Although faster interconnects between CPUs and GPUs exist, e.g., NVLink 2.0 [225], they are only supported by a few vendors or not yet commercially available. We discuss the data transfer bottleneck and the effect of faster interconnects on it in detail in Section 2.7.3.

### 2.4.9. Hardware accelerated matrix computations

Recent GPUs [30, 217, 225] contain specialized processing cores to accelerate matrix computations which form the basic building blocks of machine learning workloads. These specialized cores provide similar capabilities as dedicated processors specialized for deep learning applications, e.g., Google's Tensor Processing Unit (TPU) [138].

Early research shows that TPUs can accelerate relational operators, however, their implementation on programming frameworks for TPUs is cumbersome and inefficient [129]. In this way, relational query processing on these cores is similar to GPU programming prior to general purpose programming frameworks such as CUDA and OpenCL.

However, these specialized processing cores make GPUs an interesting hardware platform to investigate how to integrate machine learning tasks with traditional query processing pipelines and how to leverage machine learning for query optimization [3].

### 2.4.10. Combination of general purpose processing and graphics rendering capabilities

After the introduction of CUDA, most query processing research focused on the general purpose parallel processing capabilities of GPUs. Nevertheless, the GPU graphics rendering pipeline can still be used to speed up query processing tasks, e.g., spatial join queries [344]. These queries rely on computationally expensive point-in-polygon tests for which the GPU rasterization hardware is optimized. In modern graphics rendering languages, e.g., OpenGL [153], users can also combine graphics rendering functionality with general purpose computation for query processing tasks [344].

### 2.4.11. Summary

As this overview shows, the research community has embraced GPUs for query processing. New hardware features are typically exploited within two to four years in research prototypes and heterogeneous execution has been a focus from the start. However, so far we are not aware of research that utilizes specialized processing cores for matrix computations to combine query processing with machine learning.

## 2.5. Scheduling

A core task of query processing on a heterogeneous CPU/GPU system is to decide how to schedule a workload on multiple compute units with different capabilities. In this section, we present a scheme to classify these scheduling decisions. We start by formulating a number of requirements that the scheduling process should take into account. Afterwards, we give a brief overview of the dimensions of our classification scheme and then present each dimension in detail.

### 2.5.1. Scheduling policy requirements

Based on our review of different query processing systems, we have synthesized the following four requirements when scheduling a workload on a heterogeneous CPU/GPU system.

(1) *The scheduling policy should account for the architectural differences between CPUs and GPUs.* Since CPUs and GPUs are designed for different application requirements, it follows that one or the other may be more suitable to run a specific data processing task. A scheduling policy should exploit the strength of each processor and avoid its weaknesses and limitations.

(2) *The scheduling policy should account for workload characteristics.* The suitability of a processor may depend on data characteristics, e.g., the input size [201], or query characteristics, e.g., the mixture between search and update queries [355].

(3) *The scheduling policy should account for the integration of the CPU and the GPU.* On heterogeneous systems with a dedicated GPU, data transfers should be minimized because of their large overhead. In contrast, on systems with a cache-coherent integrated CPU/GPU, tasks on CPU and GPU compute units can cooperate closely without synchronizing via system main memory [40].

(4) *CPU and GPU utilization should be balanced.* When utilization is unbalanced, the resources of the idle processor are wasted [355]. However, a query processing system can maximize its overall throughput if idle processors can work on other independent tasks [201, 247].

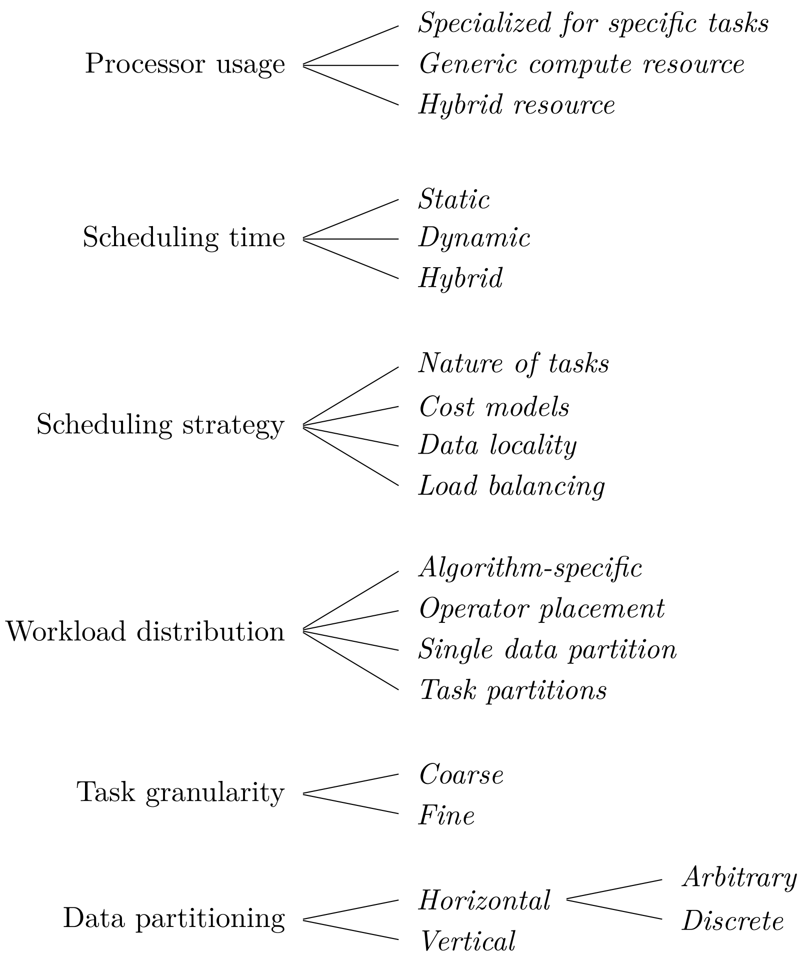


Figure 2.4.: Scheduling dimensions.

### 2.5.2. Classification overview

In their survey on heterogeneous computing techniques, Mittal and Vetter propose two dimensions to categorize these scheduling decisions [203]. The first dimension categorizes the *scheduling time*, i.e., when a scheduling decision is made. A program may decide the schedule *statically* (before program execution), *dynamically* (during execution), or employ a *hybrid* schedule (interleaving static and dynamic schedules for different tasks). The second dimension categorizes the *scheduling strategy*, i.e., the underlying determinants for the scheduling decision. In our review of different query processing systems, we have found the *nature of the task*, *cost models*, *data locality*, and *load balancing* as determinants for scheduling decisions.

Table 2.4.: Publications describing or implementing different processor usage strategies.

| Usage | Publications |
|---|---|
| Specialized | Approximate & Refine [247], Statistical coprocessor [122], Mega-KV [356], Caldera [19], Raza et al. [261], GSS [33], HELLS join [148], STIG [70], HB$^+$-tree [283], Stehle et al. [298], G-Grid [177], GAT [351], Sioulas et al. [289] |
| Generic | GDB [116], CoGaDB [42], SABER [163], DB2 BLU [201], HERO [145], HetExchange [54], FineStream [352], Beier et al. [28], He et al. [120], Bøgh et al. [34] |
| Hybrid | He et al. [121], DIDO [355], SCCG [333], Gubner et al. [106], Lutz et al. [189] |

In the context of query processing, we propose four additional dimensions. The third dimension categorizes the *processor usage*, i.e., whether they are used as *specialized resources for specific tasks*, as *generic compute resources*, or as *hybrid resources*, i.e., as a mixture of the two approaches. The remaining dimensions are the *workload distribution*, the *task granularity*, and the *data partitioning scheme*. The first three dimensions classify how a scheduling decision is made. The last three dimensions further classify the types of tasks that are scheduled. We give an overview of these six dimensions in Figure 2.4.

#### 2.5.2.1. Task types

For the purpose of our classification, we define a task as a *unit of work for which a query processing system decides whether to execute it on a CPU or on a GPU*. Such a task can be very simple, e.g., the computation of a hash value as part of a hash join, or very complex, e.g., the execution of a complete OLTP query inside an HTAP query engine. A task can also work on just a subset of the data. In this case, multiple similar tasks are parallelized across the CPU and the GPU.

We intentionally adopt this broad definition to capture the very different scheduling choices of the query processing systems which we evaluated in this survey.

### 2.5.3. Processor usage

We discuss this dimension first because it greatly influences the scheduling time and strategy.

Since CPUs and GPUs have different capabilities, a query processing system may use them as *specialized processors for specific tasks*. We call these *specialized systems*. Alternatively, a query processing system may use CPUs and GPUs as *generic compute resources*. We call these *generic systems*.

In a specialized system, a developer analyzes the query processing system, identifies the processing requirements of individual tasks, and determines a fixed assignment of

Table 2.5.: Publications describing or implementing different scheduling time strategies.

| Time | Publications |
|---|---|
| Static | GDB [116], Approximate & Refine [247], CoGaDB [42], He et al. [121], Statistical coprocessor [122], Mega-KV [356], Caldera [19], Raza et al. [261], GSS [33], He et al. [120], HELLS join [148], STIG [70], HB$^+$-tree [283], Stehle et al. [298], G-Grid [177], GAT [351], Sioulas et al. [289] |
| Dynamic | Breß et al. [44], SABER [163], DB2 BLU [201], HERO [145], FineStream [352], SCCG [333], Beier et al. [28], Bøgh et al. [34] |
| Hybrid | DIDO [355], HetExchange [54], Gubner et al. [106], Lutz et al. [189] |

each task to the most suitable processor. In other words, specialized systems use a static schedule based on the nature of the task. A major advantage of this approach is that the implementation can fully exploit the specific capabilities of each processor, as well as the specific integration of the CPU and the GPU. However, a static schedule can leave a processor underutilized or fail to adapt to specific workloads [355].

In a generic system, the processors are not distinguished by their capabilities but by their relative throughput. Typically, throughput is explicitly quantified using cost models. However, a system may also implicitly balance the workload on each processor through work stealing or morsel-driven parallelism [175]. An advantage of this approach is that systems can balance processor utilization and adapt to workload changes in a flexible manner [163, 333, 355]. Note that such a system may still exploit the specific capabilities of CPUs and GPUs by providing specialized implementations for each processor type [116, 145].

A query processing systems may also use processors as *hybrid resources*. Such a system assigns some tasks to a specific processor based on their nature, while other tasks can execute on any processor.

We classify existing work according the processor usage in Table 2.4.

### 2.5.4. Scheduling time

The scheduling time determines when a system assigns a task to a specific processor. A schedule may be *static*, i.e., fixed before program execution, or *dynamic*, i.e., adaptively set while the program executes. The processor usage partially determines the scheduling time. In specialized systems, the schedule is always statically determined by the design of the implementation. In contrast, in generic systems, the schedule may be static or dynamic. A system may also implement a *hybrid schedule*, i.e., statically assign CPU and GPU resources to execute parts of a query during query optimization, but then

Table 2.6.: Publications describing or implementing different scheduling strategies.

| Strategy | Publications |
|---|---|
| Nature of task | Approximate & Refine [247], Statistical coprocessor [122], He et al. [121], Mega-KV [356], DB2 BLU [201], Caldera [19], DIDO [355], Raza et al. [261], SCCG [333], GSS [33], HELLS join [148], STIG [70], HB$^+$-tree [283], Stehle et al. [298], G-Grid [177], GAT [351], Sioulas et al. [289], Gubner et al. [106], Lutz et al. [189] |
| Cost models | GDB [116], Breß et al. [43], Karnagel et al. [147], CoGaDB [42], He et al. [121], SABER [163], DIDO [355], HERO [145], FineStream [352], Beier et al. [28], He et al. [120], HB$^+$-tree [283] |
| Data locality | Breß et al. [44], HERO [145], HetExchange [54] |
| Load balancing | SABER [163], DB2 BLU [201], DIDO [355], HetExchange [54], SCCG [333], Bøgh et al. [34], Gubner et al. [106], Lutz et al. [189] |

modify these decisions dynamically during query execution, e.g., based on data locality and processor load. The reasons for these static and dynamic scheduling decisions are based on the scheduling strategy which we discuss next.

We classify existing work according the scheduling time in Table 2.5.

### 2.5.5. Scheduling strategy

The scheduling strategy determines which reasons a query processing system considers when it assigns a task to a specific processor. Again, the processor usage partially determines the scheduling strategy. Specialized systems decide the schedule based on the *nature of tasks*. Generic systems also consider the task nature in their scheduling decisions. In addition, generic systems take other metrics into account, i.e., the expected costs of a task based on a *cost model*, the *data locality*, and the *processor load*.

We classify existing work according the scheduling strategy in Table 2.6.

#### 2.5.5.1. Nature of tasks

This scheduling strategy aims to account for architectural differences between CPUs and GPUs, as well as for workload characteristics, by applying task or data-based heuristics to assign tasks to the most suitable processor. In the following, we summarize guidelines for tasks that should run on the CPU and GPU. Note that these guidelines are not absolute. Other considerations, e.g., the data locality and current processor load, which we discuss in sections 2.5.5.3 and 2.5.5.4, may cause a query processing system to schedule a task on a less suitable processor.

**Guidelines for CPU tasks.** The CPU is especially suited to process:

*(1) Branch-heavy tasks*, e.g., tree traversals [70, 351] or finding shortest paths in graphs [177]. The hardware support for branch prediction and speculative execution on CPUs can extract implicit instruction parallelism from such an execution stream [125]. In contrast, on GPUs these tasks may cause control flow divergence and increase execution time [213]. However, the effect of control flow divergence due to variable-sized data or skewed data can be mitigated by rebalancing the workload across the threads of a warp [89].

*(2) Small data batches*, e.g., sorting or aggregating a small number of tuples [201] or low-frequency updates in key-value store workloads [355]. These either do not provide enough data parallelism to fully utilize GPU resources or the speedup that can be attained by processing these on the GPU does not overcome the overhead of the data transfer.

*(3) Operations with small state*, e.g., aggregating over a small number of groups [201]. If the state fits into CPU caches, the CPU does not have to wait for slow random main memory accesses. It can therefore process data at a much faster rate than sending it over the system bus.

*(4) Preprocessing or postprocessing very large data sets*, e.g., an initial partitioning step of a hash join [289] or a final merge of sorted runs [298]. These steps reduce the size of the working sets that have to be processed on the GPU at a time, and overcome the constraints of limited GPU memory.

*(5) Transactional queries in HTAP systems.* The latency-critical nature of these queries matches the task-parallel nature of CPUs [19, 261]. Furthermore, accessing small values of random individual tuples over PCIe reduces the effective PCIe bandwidth and exacerbates the data transfer bottleneck [207]. GPUTx [119] proposes to process transactions on GPUs in bulk to achieve high throughput at the cost of increased latency. However, it is limited to GPU-resident data, so it does not suffer from slow random data access over PCIe.

*(6) System calls*, e.g., for network processing and memory allocation [355, 356] or for reading data from storage [149]. The traditional GPU programming model (see Section 2.3.4) is designed to offload computations and does not provide access to system calls. NVIDIA GPUs can read directly from other PCIe devices, including from network and storage, using GPUDirect [222]. However, this is not a generic interface and the host CPU must still initiate communication. GENESYS [330] is a proof-of-concept of a generic GPU system call implementation for Linux kernels.

**Guidelines for GPU tasks.** The GPU is especially suited to process:

*(1) Compute-intensive operations*, e.g., relational queries with many aggregation functions [201], spatial queries containing point-in-polygon tests [70, 333, 351], or skyline queries containing dominance tests [33]. These queries can be effectively parallelized and benefit from the high aggregate computing power of the GPU.

*(2) Random access to large working sets*, e.g., batched lookups of individual keys in hash tables [355, 356] or B+-tree indexes [283]. These tasks benefit from the latency hiding capabilities and the high memory bandwidth of GPUs. In contrast, CPUs incur costly memory stalls if the data does not fit into CPU caches.

*(3) Operations with large state*, e.g., hash joins [142, 189, 289], hash aggregations [149, 201], or Bloom filters [106] with many keys. Because of its superior random access performance, the GPU can efficiently process much larger state than the CPU. The GPU can improve throughput even further by partitioning the input internally so that the state of each partition can be processed in fast local memory [289].

*(4) Analytical queries in HTAP systems.* The data-intensive nature of these queries matches the data-parallel nature of GPUs [19, 261]. Furthermore, these queries often process large contiguous blocks data of data, e.g., entire columns, which utilizes PCIe bandwidth efficiently [207], and multiple queries can share the transferred data [261].

#### 2.5.5.2. Cost models

This scheduling strategy aims to account for architectural differences between CPUs and GPUs, as well as for workload characteristics, in a more quantitative way than the previously described heuristics. Given a cost model that estimates the throughput of individual tasks on each processor, as well as any data transfer costs, we can schedule tasks and partition data in a way that maximizes the overall throughput of a query processing system. Some costs models also factor in additional overheads, e.g., a processor-specific execution latency [147] or transfer initialization costs [116].

**Estimating throughput.** The throughput of a task can be estimated in two ways. *Black-box* cost models treat tasks as opaque operations and learn their processor-specific throughput based on historical data. They can be applied without detailed knowledge of the hardware or the implementation of the task.

In contrast, *white-box* cost models estimate throughput analytically. They break down a task into individual components and analyze its memory access pattern. From this analysis they estimate the execution time and the memory access time. The individual components can be data processing primitives, e.g., *map* or *reduce*, which are evaluated

Table 2.7.: Publications describing or implementing cost models.

| Cost model | Publications |
|---|---|
| Black box | Breß et al. [43], CoGaDB [42], SABER [163], FineStream [352], Karnagel et al. [147], HERO [145], Beier et al. [28], HB$^+$-tree [283] |
| White box | GDB [116], DIDO [355], He et al. [120], He et al. [121] |

with micro benchmarks [116]. Alternatively, a cost model may count individual instructions and combine them with the theoretical peak instructions per cycle throughput of a processor [120, 121, 355]. The memory access time of tasks that run on the CPU can be estimated using well-known cost models [191]. However, due to architectural differences, we cannot apply these directly to tasks executed on the GPU. Instead, GPU cost models have to account for the difference between coalesced and non-coalesced device memory access [116]. For integrated CPU/GPU systems, there are a number of cost models that treat both processor types in a unified way [120, 121, 355] and also account for the interference when both processors execute tasks [355].

In Table 2.7, we list publications for different cost model types.

#### 2.5.5.3. Data locality

This scheduling strategy aims to minimize the impact of the data transfer bottleneck on systems with dedicated GPUs. Query processing systems which employ cost models as a scheduling strategy implicitly take data locality into account as they will not assign a task to a processor if the data transfer results in an overall slower execution [116, 147]. However, cost models are not necessary to exploit data locality, as it suffices to track the location of data during the execution of a query. For example, HetExchange [54] assigns a specific GPU or a specific CPU core to each instance of its query pipelines. By tracking this information, HetExchange can process data blocks based on their locality even if the pipeline crosses device boundaries multiple times.

Early decisions during the execution of a query can limit the choices in subsequent stages because they determine the location of intermediate results [145]. To increase its choices, a query processing system can cache intermediate results even if it moves them to another processor [145]. Breß et al. [44] implement a scheduling policy in CoGaDB [42] that relies solely on data locality for its scheduling decisions. They cache frequently-accessed columns on the GPU and schedule tasks to run on the GPU only if all their inputs are present in GPU memory.

#### 2.5.5.4. Load balancing

This scheduling strategy aims to maximize the utilization of all computing resources in a heterogeneous CPU/GPU system. Specialized query processing systems implement a fixed task assignment to each processor. In this case, the processor load can vary a lot because it depends on the relative computing power of the CPU and GPU. In contrast, generic query processing systems can balance the load of these processors by freely assigning or migrating tasks to them.

The systems in this survey employ the following four load balancing approaches.

*(1) Use a cost model to estimate and track the execution time of individual tasks.* Normally, such a system would assign a task to the processor which can execute it in the shortest amount of time. By keeping track of the estimated execution times of the tasks that are currently scheduled in the system, it can determine if a slower processor would be able to finish this task earlier. This approach is taken, e.g., by CoGaDB [42] and SABER [163].

*(2) Use a cost model to partition the data according to the relative throughput of each processor.* Such a system does not have to track the execution time of the tasks running in the system because it aims to balance each operation on all processors. This approach is taken, e.g., by GDB [116], the HB$^+$-tree [283], and He et al. [120, 121],

*(3) Partition the data into discrete batches and opportunistically assign a task to an idle processor.* This approach is similar to morsel-driven parallelism [175]. For example, HetExchange [54] partitions the input data of individual relational queries and Lutz et al. [189] split the input data of individual hash joins. Note that for this use case, dedicated GPUs require larger batches for efficient execution than CPUs to overcome data transfer overheads [106, 189].

*(4) Partition the data into discrete batches and perform work stealing.* Such a system assigns a task to a particular processor according to its nature, but as long as the task is not started, another processor can work on it. This approach is taken, e.g., by DIDO [355] and Wang et al. [333].

**Combining load balancing with other scheduling strategies.** Load balancing is often employed together with other strategies to combine static and dynamic scheduling. For example, HetExchange [54] statically assigns CPU and GPU resources to the execution of a query during query optimization but then routes data blocks dynamically to different processors during query execution. DIDO [355] employs a cost model to assign operators of its fixed processing pipeline to CPU or GPU compute units. However, since each query is independent of the others, an idle compute unit can steal the execution of a query batch.

Load balancing can also be employed to reduce load imbalances that arise from using CPUs and GPUs as specialized processors based on the nature of a task. For example, Wang et al. [333] normally schedule a compute-intensive spatial query on the GPU and a parsing task on the CPU. However, either processor can pick up a task from the other processor if it has free resources. Finally, Appuswamy et al. [19] propose to process analytical queries on the GPU by default, but also involve the CPU if it has free resources.

**Avoiding task interference.** Related to the goal of maximizing processor utilization is the need to avoid task interference. Workloads on the CPU and GPU can interfere with each other because they share common resources. On integrated GPUs, the bandwidth requirements of the large number of GPU threads tend to monopolize the on-chip network, the last-level cache, and the memory controllers [150]. This monopolization increases the latency of CPU threads and therefore decreases their performance. DIDO [355] attempts to model the effect of this interference explicitly in its cost model to determine the expected runtime of a task.

Dedicated GPUs can avoid this interference if they work on GPU-resident data because of their separate memory space. However, workloads on dedicated GPUs can also interfere with CPU workloads when both processors access data in main memory [19, 106, 261, 289]. Sioulas et al. [289] observe this effect in a multi-socket system during an initial partitioning of the data on the CPU, to reduce the working set size of a hash join on the GPU. The cache coherency traffic between two CPU sockets caused by this partitioning interferes with the concurrent data transfer to the GPU. Since the overall performance is bounded by the data transfer over PCIe, they reduce the number of partitioning threads on the CPU to limit this interference.

Caldera [19] and Raza et al. [261] propose a design of an HTAP system which executes OLTP queries on the CPU and OLAP queries on the GPU, in order to reduce the house pattern [252], i.e., the reduction of OLTP throughput as the number of OLAP clients increases. However, since both workloads still require access to main memory, this separation reduces interference only to a certain extent. Caldera organizes data in a unified storage system and employs copy-on-write snapshots when OLTP queries update data. These updates compete with data transfers to the GPU, which increases the execution time of OLAP queries and can cause the throughput of OLTP queries to collapse. This is effect is especially strong if the hot set of the OLTP queries contains most of the data and if there are few OLAP queries per snapshot to maximize data freshness. To overcome this interference, Raza et al. [261] separate the storage of both workloads and periodically merge OLTP updates into the data used for OLAP queries.

### 2.5.6. Workload distribution

The strategy to divide a workload into smaller units depends on the processor usage of the query processing system.

#### 2.5.6.1. Using CPUs and GPUs as specialized processors

As we described in Section 2.5.3, in specialized query processing systems, the assignment of individual tasks to a processor depends on the specific algorithm design chosen by a developer. This approach allows developers to tailor the implementation to each processor and to work around weaknesses, e.g., excessive data transfer to dedicated GPUs. For example, Stehle et al. [298] employ the GPU to sort large runs, which fit into GPU memory, but then merge the sorted runs on the CPU, which can access them sequentially. Using this division of labor, data is only transferred a single time to and from the GPU.

A common design pattern that we encountered in our survey is to generate result candidates on one processor and verify them on the other processor to produce final results. This approach is taken, e.g., by Li et al. [177] and by Zhang et al. [351], to process compute-intensive spatial queries. Note that Li et al. [177] perform the candidate generation on the GPU, whereas Zhang et al. [351] perform the result verification on GPU. In other words, these systems make different decisions which of the two tasks, candidate generation or result verification, is executed on the GPU, depending on which task is more compute-intensive and easier to parallelize.

A related approach is to reformulate a data processing problem so that it conforms to the candidate generation and results verification pattern. This approach is taken, e.g., by the *Approximate and Refine* processing model for relational queries [247] and by the key-value store Mega-KV [356]. These systems first compute an approximate answer based on lossily compressed data on the GPU and then refine the approximate answer on the CPU. In both cases, their goal is to maximize the utility of GPU-resident data, in order to process data that is much larger than GPU memory while avoiding costly data transfers between the CPU and the GPU.

Table 2.8.: Publications describing or implementing different workload distributions in generic systems.

| Workload distribution | Publications |
|---|---|
| Operator placement | CoGaDB [42], Karnagel et al. [146], DIDO [355], HERO [145], FineStream [352] |
| Single data partition | SABER [163], Beier et al. [28], Bøgh et al. [34] |
| Task-specific partitions | GDB [116], He et al. [120, 121], DB2 BLU [201], HetExchange [54], SCCG [333], Gubner et al. [106], Lutz et al. [189] |

#### 2.5.6.2. Using CPUs and GPUs as generic compute resources

A generic query processing system can employ one of the following three approaches to distribute its workload.

*(1) Operator placement.* In this approach, tasks correspond to specific operators in the query plan. Each task processes all of its input data either on the CPU or GPU. On the one hand, each task can be scheduled on the most suitable processor. Thus, even when tasks do not run concurrently, this approach may improve performance compared to running all tasks on the fastest processor [147]. On the other hand, this approach may lead to excessive data transfers between CPU and GPU. It may also lead to an unbalanced utilization of the CPU and GPU [120]. In the extreme case, all tasks are executed on a single processor and the other is idle.

*(2) Single partition of the data.* In this approach, the data is partitioned once and the CPU and GPU execute the entire query on their partition. On the one hand, both the CPU and the GPU contribute to the progress of the query. With an accurate cost model, or dynamic work stealing of discrete data batches, both processors can finish at the same time. On the other hand, both processors also execute tasks for which they may not be suitable.

*(3) Task-specific data partitions.* In this approach, the system partitions the data individually for each task, to take the task-specific throughput of each processor into account. This approach combines the advantages of the previous two approaches [120]. The schedule takes the strengths of each processor into account but no processor is idle. When the partition ratios of different tasks vary greatly, because the characteristics of the tasks match different processors, this approach leads to increased communication and data transfer between the CPU and the GPU. Therefore, it is especially suited for integrated CPU/GPU systems, where CPU and GPU compute units can closely cooperate [120].

In Table 2.8, we classify existing work according these approaches.

Table 2.9.: Publications describing or implementing different task granularities in generic systems, from coarse to fine-grained.

| Granularity | Publications |
|---|---|
| Query | SABER [163] |
| Pipeline | HetExchange [54] |
| Operator | GDB [116], CoGaDB [42], DB2 BLU [201], DIDO [355], FineStream [352], SCCG [333], Lutz et al. [189] |
| Primitive | HERO [145], He et al. [120, 121] |

Table 2.10.: Publications describing or implementing data partitioning strategies.

| Partitioning | Size | Publications |
|---|---|---|
| Horizontal | Arbitrary | GDB [116], He et al. [120, 121], DB2 BLU [201] |
| | Discrete | SABER [163], HetExchange [54], DIDO [355], SCCG [333], Beier et al. [28], Bøgh et al. [34], Gubner et al. [106], Lutz et al. [189] |
| Vertical | | Approximate & Refine [247], CoGaDB [42] |

### 2.5.7. Task granularity

As we mentioned in the previous section, in specialized query processing systems, the assignment of individual tasks to a processor depends on the specific algorithm design chosen by a developer. Generic systems divide the workload along a continuum of *fine-grained* or *coarse-grained* tasks. In Table 2.9, we give specific examples of different granularities and classify existing work.

Scheduling more fine-grained tasks has a number of advantages because they are less complex and have limited functionality compared to more coarse-grained tasks. First, they can be more easily mapped to a suitable processor [120]. Second, black-box cost models that rely on historical data can make more precise predictions of their runtime [145]. For example, the specific execution semantics of a complex operator may depend on its position in the query plan. Therefore, more coarse-grained database operators exhibit a larger runtime variation than more fine-grained primitives [145]. However, a large number of fine-grained tasks can lead to more data transfers when each task is scheduled independently, without taking the resulting data transfers in subsequent tasks of the query plan into account [145]. Therefore, more fine-grained tasks work especially well on integrated CPU/GPU systems, where CPU and GPU compute units can closely cooperate [120].

### 2.5.8. Data partitioning

Finally, we can categorize query processing systems by their strategy to partition the input data. A system can partition data *horizontally* in arbitrarily-sized partitions or discretely-sized batches. As we describe in Section 2.5.5.4, partitioning the data horizontally forms the basis for balancing the load on CPUs and GPUs in a heterogeneous system, either through cost models, work stealing, or morsel-driven parallelism. A system can also partition data *vertically*. CoGaDB [44] partitions data by columns to cache columns on the GPU and increase data locality. Approximate & Refine [247] partitions data by bits to compute an approximate query answer on the GPU and increase the utility of GPU-resident data. We classify existing work according the partitioning strategy in Table 2.10 and provide a more granular classification in Table 2.15 on page 60.

### 2.5.9. End-to-end integration

A complete heterogeneous query processing system must take the availability of computing resources into account before scheduling tasks on the CPU or the GPU. DB2 BLU demonstrates that this can be achieved in an ad-hoc manner in an industrial database. DB2 BLU tracks the usage of GPU memory and only assign tasks to a GPU if it has enough resources [201]. This approach assumes that the system will only assign tasks to the GPU that benefit from GPU acceleration in the first place, according to the heuristics described in Section 2.5.5.1. However, if the GPU is busy, the CPU can still make progress on the query.

A more involved scheduler employs cost models to track the estimated execution time of the current CPU and GPU workloads. This approach is taken by CoGaDB [42] and SABER [163] to balance the load of both processors. The same information can be used to reduce the query admission rate under general high system load.

### 2.5.10. Summary

A heterogeneous query processor has many degrees of freedom to decide the general strategy on how to employ each processor; when to fix the schedule and based on what strategy; and how to distribute the workload and at what granularity. In this section, we reviewed these scheduling dimensions in general. In Section 2.8, we categorize different heterogeneous query processing systems according to this classification scheme and discuss main trends. In Appendix A we describe the scheduling decisions of selected heterogeneous query processing systems in more detail.

## 2.6. Architecture-specific query processing

As we discussed in Section 2.3, CPUs and GPUs have to be programmed differently due to their architectural differences. Consequently, we also have to adapt query processing code to the specific processor it runs on to achieve peak performance. Unfortunately, this adaptation increases the development cost of heterogeneous query processing systems. Developers of such systems must not only have expertise in query processing but also detailed knowledge about different processor architectures. In this section, we review techniques to reduce this implementation complexity. We approach the performance implications of a heterogeneous system on two levels, the high-level query execution plan and low-level operator implementations.

### 2.6.1. High-level query execution plan

Query processing systems that use CPUs and GPUs as dedicated processors for specific tasks typically implement a fixed query processing pipeline that already takes the different architectures of CPUs and GPUs into account. In these cases, the system has little or no degree of freedom to choose between different high-level plans.

However, in other cases, specifically for relational query processing, the query processing system has a wide latitude to choose a physical query execution plan. With the exception of Agbaria et al. [9], all of the surveyed relational query processors on GPUs execute query plans that are produced by a traditional query optimizer that targets CPUs. On the one hand, this approach greatly simplifies the implementation complexity of a heterogeneous query processor. It reduces the scheduling problem to a decision where to execute specific operators and/or primitives and how to distribute the input data. On the other hand, a query plan optimized for CPUs is not necessarily optimal for GPUs [9].

With the exception of GPUTx [119], early relational query processors target analytical workloads. Consequently, they are bulk processors and implement an operator-at-a-time processing model [45]. However, this processing model suffers from high materialization costs [38]. It generates a large amount of GPU device memory traffic [87] and large data transfers over the system bus [44].

Modern query processors on CPUs implement a processing model that is based on in-cache vectorization [38] or on query compilation [208]. A detailed analysis of these two processing models shows that both achieve similar performance for analytical queries but neither is clearly dominated by the other and the fastest processing model depends on the specific query [151]. Vectorization is better at hiding cache miss latencies whereas query

Table 2.11.: Publications describing or implementing different query processing models.

| Processing model | Publications |
|---|---|
| Bulk processing | GDB [116], Ocelot [123], CoGaDB [42], SABER [163], DB2 BLU [201], HERO [145] |
| Vectorization | He et al. [120], He et al. [121], GPL [244], Körber et al. [164] |
| Query compilation | Hawk [46], HorseQC [87], HetExchange [54], DogQC [89], |

compilation has a higher computational efficiency [151]. Consequently, recent state-of-the-art relational query processors implement a combination of both processing models and employ vectorization and code generation for different parts of a query plan [151, 172, 199].

As we show in Table 2.11, vectorization and query compilation have been implemented in relational query processors on the GPU. Furthermore, He et al. [120, 121] and HetExchange [54] implement pipelined query processing across CPUs and GPUs. However, so far there is no apples-to-apples comparison that investigates the relative advantages of these processing models on GPUs.

For example, query compilation can exacerbate the effect of control flow divergence when different threads of a warp become inactive because of filter predicates or skewed data distributions [89]. In this case, the warp workload inside a pipeline must be rebalanced to reduce the effect of control flow divergence and achieve robust query execution times [89]. Furthermore, by its nature, query compilation generates coarse-grained tasks. It is therefore particularly well suited for heterogeneous query processing on systems with dedicated GPUs. In contrast, on integrated GPUs, coarse-grained compiled query pipelines limit the ability of a system to schedule fine-grained tasks on the most suitable processor. However, query compilers can adapt the task granularity by introducing artificial pipeline breakers into the query plan and splitting a query pipeline into separate tasks to better take advantage of the different capabilities of CPUs and GPUs.

### 2.6.2. Low-level operator implementations

In contrast to high-level query plans, low-level operator implementations are regularly optimized for a specific processor type. For example, multiple GPU threads often need to effectively cooperate with each other to achieve peak performance [89, 187] which is not the case on CPUs. Consequently, a heterogeneous query processor that uses CPUs and GPUs as generic compute resources requires optimized operator implementations for both processors. In the extreme case, GDB [116] contains completely separate

Table 2.12.: Publications describing or implementing different operator implementation strategies.

| Strategy | Publications |
|---|---|
| Separate operator implementations | GDB [116], CoGaDB [42], SABER [163], DB2 BLU [201], HERO [145], SCCG [333], Beier et al. [28], Lutz et al. [189] |
| Hardware-oblivious operators | Ocelot [123], DIDO [355], He et al. [120, 121], FineStream [352] |
| Template-based code generation | Bøgh et al. [34], HetExchange [54] |
| Intermediate representation | Voodoo [248], Hawk [46] |
| Learned operator implementations | Rosenfeld et al. [268], Hawk [46], Rosenfeld et al. [267] |

implementations that utilize different programming frameworks, i.e., CUDA [211] for GPU operators and OpenMP [65] for CPU operators. Having separate operator implementations for different processors greatly increases implementation complexity of a heterogeneous query processor. In the following, we discuss four strategies to reduce this implementation complexity: *hardware-oblivious databases*, *template-based code generation*, *intermediate languages*, and *learned operator implementations*. In Table 2.12, we list publications that describe or implement these strategies.

#### 2.6.2.1. Hardware-oblivious operators

Hardware-oblivious databases run the same operator implementations on any processor. DIDO [355], He et al. [120, 121], and FineStream [352] rely on OpenCL [300] to support operator implementations for different processors from a common code base. OpenCL code is formulated in an abstract programming model (see Section 2.3.4) and hardware-specific functionality is offloaded to a vendor-provided OpenCL driver. However, although OpenCL provides *functional portability*, it does not ensure *performance portability* [272]. An OpenCL-based operator implementation still has to be optimized to a specific processor in order to achieve peak performance. Thus, no system is entirely hardware-oblivious. Even Ocelot [123], which introduced the term, uses processor-optimized memory access patterns, hash table implementations, and reduction schemes. Nevertheless, Ocelot uses an integrated code base for CPU and GPU operators which reduces code volume and complexity [123].

#### 2.6.2.2. Template-based code generation

Query processors that rely on query compilation can adapt the generated code to a specific processor type. A straight-forward approach is to first generate a high-level execution plan template that contains stubs for individual data processing operators [54]. During a second compilation pass, these stubs are specialized by processor-specific code generators.

#### 2.6.2.3. Intermediate representation

Improving on template-based code generation, a query compiler can first generate a query plan in a hardware-oblivious, declarative intermediate representation (IR). Such a formalized IR allows the query processing system to reason about hardware-specific optimizations in a structured way. For example, Voodoo [248] uses *control vectors* to map data items to virtual partitions. These virtual partitions express information about parallel execution, e.g., using SIMD registers on CPUs or interleaved access on GPUs, in a declarative, hardware-independent way. Hawk [46] uses *pipeline programs*, which are parameterized representations of operator pipelines [208], to describe the dataflow of an operator pipeline and to encapsulate implementation variants of individual operators. These pipeline programs can be adapted to a specific processor using well-defined transformation rules. This approach is not limited to query processing systems. For example, TVM [52] applies structured transformations over an IR for deep learning programs.

#### 2.6.2.4. Learned operator implementations

Even a small number of implementation parameters create a large space of possible operator implementations [268]. GPUs are especially sensitive to implementation details and suffer large performance penalties from implementations that are not optimized for the specific GPU [267]. Thus, devising fast operator implementations requires careful tuning of implementation parameters which is time-consuming and relies on expert knowledge. However, given a declarative specification of the implementation search space, a query compiler can adapt its output to a specific processor automatically [46]. A number of search strategies, based on genetic algorithms [268], local search [46, 267], or machine learning [52], have been proposed to automate the task of finding fast operator implementations.

### 2.6.3. Reverse engineering of GPU hardware details

Hardware vendors are reluctant to disclose low-level hardware details which are useful to optimize operator implementations on GPUs [137], e.g., the TLB cache size [144]. Consequently, such information has to be reverse engineered through microbenchmarking. Jia et al. [136, 137] provide an overview of previous microbenchmarking efforts on NVIDIA GPUs.

### 2.6.4. Summary

Heterogeneous relational query processors typically execute query plans generated by a traditional query optimizer that targets CPUs. We are only aware of one publication investigating whether query plans should be adapted to run efficiently on GPUs [9]. In addition, there is no apples-to-apples comparison between vectorization and query compilation on GPUs. In contrast, low-level operator implementations are highly optimized for different processor types. A number of strategies, based on OpenCL and code generation, are used to produce processor-specific optimized operator implementations from a shared code base, in order to reduce implementation complexity.

## 2.7. The data transfer bottleneck

Dedicated GPU have more compute power and much higher memory bandwidth than CPUs. However, to process data on a dedicated GPU, it must first be transferred over a system bus, e.g., PCIe 3.0. Unfortunately, the transfer bandwidth of PCIe 3.0 is an order of magnitude slower than that of CPU main memory (see Figure 2.2 on page 23). This imbalance leads to a severe bottleneck when using dedicated GPUs for data processing [99]. In fact, reducing transfers, or increasing transfer bandwidth, has a bigger impact on performance than increasing the computational power of GPUs [261, 343]. In this section, we discuss the impact of this data transfer bottleneck as well as common mitigation techniques.

### 2.7.1. Impact of the data transfer bottleneck

The data transfer bottleneck impacts operator performance and data management complexity.

#### 2.7.1.1. Impact on operator performance

The impact of the data transfer bottleneck on operator performance depends on their data access characteristics and computational complexity.

**Bandwidth-bound operations.** Since CPU main memory is faster than transferring data over PCIe 3.0, operations that process data sequentially, e.g., selections or ungrouped aggregations, are always faster on the CPU [116, 189].

**Latency-bound and compute-bound operations.** Latency-bound operations typically perform random access to (large) internal state, e.g., joins [55, 142, 189, 289], grouped aggregations [149, 201, 267], or index lookups [28, 283]. Since GPU device memory is faster than CPU main memory and larger than CPU caches, these operations can overcome the data transfer bottleneck. Operations that perform multiple passes over their inputs, e.g., sorting [94, 201, 298], also benefit from fast GPU memory. Compute-bound operations, e.g., temporal-spatial queries [177, 351], benefit from the computational power of the GPU.

**Complex queries.** Early research on relational query processing on GPUs sought to quantify data transfer overheads [116, 118]. However, these implementations performed multiple passes over the same data, which exacerbates the bottleneck, and did not overlap data transfers with computation, which is a common mitigation technique (see Section 2.7.2.1). Recent work on single-pass compiled query pipelines shows that all SSBM [230] and most TPC-H [323] queries are bottlenecked by the PCIe 3.0 transfer bandwidth [87].

#### 2.7.1.2. Impact on data management complexity

The need to transfer data over PCIe 3.0 complicates data management in three important ways.

**Pinned memory.** To achieve high transfer speeds, data must be "pinned" in main memory, i.e., memory pages have to be locked at their physical location [213]. In this case, the GPU can transfer data via direct memory access (DMA), which does not require CPU or operating system (OS) intervention. However, pinned memory cannot be paged out by the OS and therefore is a scarce resource. Copying data on-demand into pinned buffers can match the speed of transferring from pinned memory directly, but exhibits high CPU utilization [189].

**Transfer of large sequential blocks.** PCIe 3.0 transfers data in packets which consist of a header of 20-28 bytes and a payload of up to 512 bytes [207]. Irregular

Table 2.13.: Publications describing or implementing techniques to mitigate the data transfer bottleneck.

| Technique | Publications |
|---|---|
| Overlapping | Kaldewey et al. [142], GPUDB [343], Karnagel et al. [149], HB$^+$-tree [283], SABER [163], Caldera [19], Stehle et al. [298], HetExchange [54], Sioulas et al. [289], Rosenfeld et al. [267], Gubner et al. [106], Raza et al. [261] |
| Compression | Fang et al. [80], Pirk et al. [249], GPUDB [343], Rozenberg et al. [271], |
| Approximation | Pirk et al. [247], Zhang et al. [356] |
| Caching | Ocelot [123], CoGaDB [44], HERO [145], GAT [351], Raza et al. [261] |
| Data locality | CogaDB [44], HERO [145], HetExchange [54, 55] |
| Single-pass algorithms | Kaldewey et al. [142], Karnagel et al. [149], HorseQC [87], Lutz et al. [188] |
| Heterogeneous execution | Statistical coprocessor [122], Stehle et al. [298], STIG [70], GAT [351] |
| Faster system bus | Lutz et al. [189], Raza et al. [261] |

transfers of small data significantly reduce the effective transfer bandwidth [207] and compete with random access to main memory by the CPU [261]. A query processing system can overcome these overheads only when the amount of transferred data is significantly reduced, e.g., through previously applied, highly selective filter predicates [261, 343].

**Manual synchronization.** The separate GPU device memory is not coherent with CPU main memory. Therefore, a data processing system must manually synchronize data structures that are shared across both processors.

### 2.7.2. Mitigation techniques

In the following, we review seven techniques to mitigate the data transfer bottleneck: (1) overlapping transfer with execution; working on (2) compressed or (3) approximate data; (4) caching, (5) exploiting data locality; (6) single-pass algorithms; and (7) heterogeneous execution. These techniques can and should be combined to improve performance. We list publications describing or implementing these techniques in Table 2.13.

#### 2.7.2.1. Overlapping transfer with execution

A naive implementation would first transfer data to a dedicated GPU, then execute a kernel, and finally transfer the results back. In such a serialized processing scheme, the GPU and the system bus are often idle and the effective transfer bandwidth and computational throughput are greatly reduced.

To reduce idle time, GPUs can overlap kernel execution and data transfers in both directions, either through a software-managed *staged pipeline* or hardware-managed *zero-copy* [213]. In a software-managed staged pipeline, we partition the data and then transfer and process each partition independently. In this strategy, the data is pushed to the GPU. Using hardware-managed zero-copy, CPU memory is mapped into the GPU address space. When memory-mapped data is accessed by a kernel, the GPU transparently transfers it over the PCIe bus. In this strategy, the data is pulled by the GPU. Both strategies achieve the same sequential transfer bandwidth [189] but differ in their ability to cache data or exploit selective predicates.

Zero-copy reduces software complexity but transfers data on every access. Therefore, it should not be used for data that is accessed multiple times. Since zero-copy only transfers data that is actually accessed by the kernel, previously applied selective filter predicates reduce the amount of transferred data. However, in this case, the GPU transfers small data packets over the PCIe 3.0 bus which reduces the effective transfer bandwidth [207] and competes with random main memory accesses of the CPU [261]. In contrast, pipelining allows us to cache data in GPU memory and access it multiple times. However, pipelining always transfers all of the data and does not benefit from previously applied selective filter predicates.

The best transfer strategy depends on the data access pattern. Selecting an appropriate transfer strategy for different columns of relational data yields faster performance than using either zero-copy or explicit pipelining alone [261].

Note that to overlap kernel execution and transfer over PCIe, both approaches require data to be pinned in CPU memory, which is a scarce resource (see Section 2.7.1.2). In contrast, dedicated GPUs that are connected by a coherent system bus can also access unpinned memory directly, including memory that is paged out by the OS [189].

#### 2.7.2.2. Compression

In disk and memory-based database systems, compression is a proven technique to improve query performance [4, 98, 335, 360]. Compression provides two benefits [98]. (1) It effectively trades computation time to decompress data against the time and space required to transfer and store uncompressed data. (2) Some operations can work directly on compressed data, which also reduces computation time.

A number of lightweight compression schemes have been implemented on the GPU [80, 271]. Given the computational power of GPUs and their high memory bandwidth, compression is particularly effective: multiple lightweight compression schemes can be cascaded to achieve a high compression ratio [80]. Thus compression alleviates both the limited transfer bandwidth of PCIe as well as the small capacity of GPU memory. On integrated GPUs, compression can improve performance by increasing the effective main memory bandwidth [121]. To improve access locality, CPU-optimized databases typically store compressed data in chunks, where each chunk contains all the information required to decompress a single [360] or multiple [172] attribute(s) of a number of elements. However, such a storage layout is not suitable for GPUs; instead, the information required to decompress a single attribute is stored in separate arrays [271].

#### 2.7.2.3. Approximation

The lightweight compression schemes discussed in the previous section are lossless, i.e., we can reconstruct the exact data. Alternatively, the GPU can also operate on lossily compressed data. Lossy compression achieves a very high compression ratio at the cost of a post-processing step on the CPU, since the GPU only produces an approximate result.

For example, in the Approximate & Refine query processing model [247], individual attributes are bitwise partitioned. The most significant bits of an attribute are stored on the GPU and the least significant bits on the CPU. Each relational operator first computes an approximate result on the GPU which is then refined on the CPU to produce an exact result. This scheme not only enables the GPU to process data that is much larger than GPU memory but also eliminates the transfer of input data entirely.

#### 2.7.2.4. Caching

We can use GPU memory to cache input data and intermediate results. Caching is especially useful for operations that access data multiple times, e.g., sorting. However, the relatively small capacity of GPU memory limits the effectiveness of caching to reduce transfers of large data sets. Moreover, caching is not effective when data changes often, e.g., for stream processing or key-value store workloads.

*Data-driven operator placement* [44] combines caching with a scheduling strategy. The query processing system monitors the query workload and periodically moves frequently accessed data to the GPU. Subsequent operations are scheduled on the GPU only if all their inputs reside in GPU memory. *Transfer sharing* [261] applies the concept of scan sharing [110] to data transfers to the GPU. The system monitors data transfer requests and consolidates transfers of the same data by different queries. Both approaches amortize the data transfer costs over multiple queries.

#### 2.7.2.5. Data locality

Data transfers can be avoided if a query processing task is scheduled on the processor in whose memory the input data is already located. This heuristic is especially useful for query processing systems which treat CPUs and GPUs as generic compute resources, as well as for heterogeneous computing systems which contain multiple GPUs. We discuss data locality as a scheduling strategy in detail in Section 2.5.5.3.

Data locality is also an important consideration on integrated GPUs, since CPU and GPU compute units incorporate separate cache hierarchies [355].

#### 2.7.2.6. Single-pass algorithms

Since early GPUs did not support atomic operations well, data processing on GPUs relied on lock-free algorithms that perform multiple passes over the input [116, 118]. These algorithms materialize intermediate results and are often bound by GPU memory bandwidth [87]. More importantly, if the data does not fit into GPU memory, it must be transferred over the PCIe bus multiple times. Single-pass algorithms eliminate these multiple transfers and also use GPU memory more efficiently [87].

#### 2.7.2.7. Heterogeneous execution

A query processing system, which treats CPUs and GPUs as specialized processors, can implement an algorithm to solve a complex query processing task that reduces the impact of the data transfer bottleneck. In the following, we give three examples. (1) We already discussed approximate processing on the GPU which requires a post-processing step on the CPU to produce exact results. (2) The CPU can use an index to filter data that is transferred to the GPU [70, 351]. (3) To sort very large data sets, we can partition the data and sort each partition on the GPU using a software-managed data transfer pipeline. The sorted partitions are then merged on the CPU at full main memory speed [298].

### 2.7.3. Faster system bus

The techniques described in the previous section are software approaches to mitigate the effect of the data transfer bottleneck. By connecting GPUs over a faster system bus than PCIe 3.0, we can also directly increase the transfer bandwidth. PCIe 4.0, which doubles the transfer bandwidth, is already supported in recent CPUs [240, 302, 329] and GPUs [192, 217]. IBM has announced support for PCIe 5.0 on POWER10 CPUs [297], which again doubles the transfer bandwidth.

There are also dedicated buses that provide fast transfers and coherent access between CPUs and GPUs. However, they are not yet widely deployed. At the moment, IBM Power9 CPUs [274] can access NVIDIA GPUs over NVLink 2.0 [225] which provides 5× the transfer bandwidth of PCIe 3.0 [189]. AMD and Intel also integrate fast coherent CPU/GPU interconnects in current supercomputer architectures [161, 231]. These are similar to, or based on, Infinity Architecture 3, which is 4.5× faster than PCIe 3.0 [241], and Compute Express Link over PCIe 5.0 [284], which is 4× faster than PCIe 3.0.

These interconnects reduce, but do not eliminate, the impact of the data transfer bottleneck, especially for scan-dominated queries [189]. More importantly, since these interconnects support coherent access between the CPU and GPU, both processors can modify shared data structures simultaneously. This capability allows for fine-grained cooperation and makes dedicated GPUs more similar to integrated GPUs.

However, none of these developments fundamentally change the status quo that accessing CPU main memory is significantly faster than transferring data to a dedicated GPU. We list publications that evaluate query processing over fast interconnects in Table 2.13.

### 2.7.4. Summary

The slow system bus represents a significant bottleneck for query processing on dedicated GPUs that constraints the implementation of heterogeneous query processing systems. Nevertheless, a number of techniques are able to mitigate the data transfer bottleneck effectively.

## 2.8. System survey

In this section, we survey the literature on query processing on heterogeneous CPU/GPU systems. We classify query processing systems according to their scheduling decisions, their approach to handle architecture-specific implementations, and the techniques employed to mitigate the data transfer bottleneck. In this section, we discuss the major trends, whereas we describe selected systems in detail in Appendix A.

Table 2.14.: Reviewed publications.

(a) Full query processing systems

| Publication | Year | Domain |
|---|---|---|
| GDB [116] | 2009 | Relational queries |
| Approximate & Refine [247] | 2014 | Relational queries |
| CoGaDB [42] | 2014 | Relational queries |
| He et al. [121] | 2014 | Relational queries |
| Stat. coproc. [122] | 2015 | Query optimization |
| Mega-KV [356] | 2015 | Key-value store |
| SABER [163] | 2016 | Stream processing |
| DB2 BLU [201] | 2016 | Relational queries |
| Caldera [19] | 2017 | Hybrid transactional/analytical |
| DIDO [355] | 2017 | Key-value store |
| HERO [145] | 2017 | Relational queries |
| HetExchange [54] | 2019 | Relational queries |
| Raza et al. [261] | 2020 | Hybrid transactional/analytical |
| FineStream [352] | 2020 | Stream processing |

(b) Individual query processing tasks

| Publication | Year | Domain |
|---|---|---|
| SCCG [333] | 2012 | Image processing |
| Beier et al. [28] | 2012 | Generalized index |
| GSS [33] | 2013 | Skyline operator |
| He et al. [120] | 2013 | Hash join |
| HELLS join [148] | 2013 | Stream join |
| STIG [70] | 2016 | Spatio-temporal index |
| HB$^+$-tree [283] | 2016 | B$^+$ tree |
| Stehle et al. [298] | 2017 | Radix sort |
| Bøgh et al. [34] | 2017 | Skycube operator |
| G-Grid [177] | 2018 | Road networks |
| GAT [351] | 2018 | Trajectory queries |
| Gubner et al. [106] | 2019 | Bloom filter for joins |
| Sioulas et al. [289] | 2019 | Hash join |
| Lutz et al. [189] | 2020 | Hash join |

### 2.8.1. Selection criteria

Our survey covers the literature published in relevant peer-reviewed data processing conferences in the last decade. We broadly categorize the literature into two groups: those that cover *full query processing systems* and those that focus on an *individual query processing tasks*. We show the covered publications in Table 2.14 along with the publication year and the domain. In total, we cover seven relational query processors, two key-value stores, two stream processing systems, two HTAP systems, one query optimization system, as well as fourteen papers on individual query processing tasks.

The classical GPU programming model described in Section 2.3.4 treats the GPU as a coprocessor controlled by a host CPU. In this programming model, the CPU is always involved in GPU processing since it orchestrates computation on the GPU and data transfers. We exclude from our survey query processing systems that limit the CPU to these tasks. Instead, we only include systems in which both the CPU and the GPU either process data directly, or perform an incidental task such as query optimization.

### 2.8.2. Classification criteria

In Table 2.15, we categorize query processing systems according the classification scheme we developed in Section 2.5 to describe their scheduling decisions. An overview of the classification scheme is shown in Figure 2.4 on page 34. Additionally, we include the type of GPU integration as a category. As we will discuss shortly, the type of GPU integration strongly influences the scheduling decisions. Note that if the workload is distributed in an algorithm-specific way, we do not provide the task granularity or data partitioning. If the workload is distributed by a single data partition, there is typically no task granularity because the CPU and the GPU perform the same tasks on different data.

In Table 2.16, we categorize the processing model and the operator implementation strategy of query processing systems that use both processors as generic compute resources. A summary of the two categories is shown in tables 2.11 and 2.12 on page 48. Note that the processing model is only a useful category if the system processes generic queries, i.e., for relational queries or stream queries, as well as for HTAP systems. Key-value stores or individual query processing tasks typically either implement a fixed pipeline or a single operator.

In Table 2.17, we list which techniques to mitigate the data transfer bottleneck are employed by query processing systems that are implemented on a dedicated GPU. A

Table 2.15.: Overview of scheduling decisions of reviewed heterogeneous query processing systems.

| Publication | GPU integration | Processor usage | Scheduling time | Scheduling strategy | Workload distribution | Task granularity | Data partitioning |
|---|---|---|---|---|---|---|---|
| *Full query processing systems* | | | | | | | |
| GDB [116] | Dedicated | Generic | Static | Cost model[1,2] | Task partitions | Operator | Tuples |
| Approx. & Refine [247] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | Bits |
| CoGaDB [42] | Dedicated | Generic | Both[3] | Data locality, cost model[1,2] | Operator placement | Operator | Columns |
| He et al. [121] | Integrated | Hybrid[8] | Static | Task nature, cost model[1] | Task partitions | Primitive | Tuples |
| Stat. coproc. [122] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| Mega-KV [356] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| SABER [163] | Dedicated | Generic | Dynamic | Load balancing, cost model | Single partition | Query | Data batch |
| DB2 BLU [201] | Dedicated | Generic | Dynamic | Task nature, load balancing | Task partitions | Operator | Tuples |
| Caldera [19] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | Query type | — |
| DIDO [355] | Integrated | Hybrid[4] | Hybrid[5] | Task nature, load balancing, cost model | Operator placement | Operator | Query batch |
| HERO [145] | Both | Generic | Dynamic | Data locality, cost model[2] | Operator placement | Primitive | — |
| HetExchange [54] | Dedicated | Generic | Hybrid[6] | Load balancing, data locality | Task partitions | Pipeline | Data batch |
| Raza et al. [261] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | Query type | — |
| FineStream [352] | Integrated | Generic | Dynamic | Cost model | Operator placement | Operator | — |
| *Individual query processing tasks* | | | | | | | |
| SCCG [333] | Dedicated | Hybrid[7] | Dynamic | Task nature, load balancing | Task partitions | Operator | Polygon pairs |
| Beier et al. [28] | Dedicated | Generic | Dynamic | Cost model | Single partition | — | Query batch |
| GSS [33] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| He et al. [120] | Integrated | Generic | Static | Cost model[1] | Task partitions | Primitive | Tuples |
| HELLS join [148] | Integrated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| STIG [70] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| HB$^+$-tree [283] | Dedicated | Specialized | Static | Task nature, cost model[1] | *Algorithm-specific* | — | — |
| Stehle et al. [298] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| Bøgh et al. [34] | Dedicated | Generic | Dynamic | Load balancing | Single partition | — | Cuboids, points |
| G-Grid [177] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| GAT [351] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| Sioulas et al. [289] | Dedicated | Specialized | Static | Task nature | *Algorithm-specific* | — | — |
| Gubner et al. [106] | Dedicated | Hybrid[9] | Hybrid | Task nature, load balancing | Task partitions | Operator | Key batch |
| Lutz et al. [189] | Dedicated[10] | Hybrid[11] | Hybrid | Task nature, load balancing | Task partitions | Operator | Tuple batch |

[1] Including implicit load balancing through data partitioning.
[2] Including implicit data locality by modelling transfer costs.
[3] CoGaDB first implemented static scheduling [42]. Follow-up work switched to dynamic scheduling [44].
[4] DIDO implements a fixed-length pipeline. The first and last stages are always scheduled on CPU compute units. The middle stages can be scheduled on CPU or GPU compute units.
[5] Query pipelines are compiled statically for each query batch. Within a query batch, different query types are routed to different query pipelines.
[6] HetExchange schedules operator pipelines on CPU cores or GPUs at query compile time. However, data batches are routed to different processors at runtime.
[7] SCCG implements a fixed-length pipeline. Some stages are always scheduled on the CPU. Some stages can be scheduled on the CPU or the GPU.
[8] The system can schedule every primitive on any compute unit, however a prefetching primitive is always scheduled on the CPU.
[9] Both CPU and GPU perform bloom filter lookups. The final join is performed only on the CPU.
[10] The GPU is connected using NVLink 2.0 which is fast interconnect that supports cache-coherent access.
[11] Any processor can normally build and probe the hash table, however small hash tables are typically build on the GPU.

summary of these techniques is shown in Table 2.13 on page 53. We also specify whether a system can scale to arbitrarily sized inputs using out-of-core processing.

### 2.8.3. Discussion

Two categories have an outsized influence on the scheduling decisions of heterogeneous query processing systems: the processor usage strategy and the type of GPU integration. In the following we discuss these in more detail. We also discuss general trends how these systems implement operators for multiple processors and mitigate the data transfer bottleneck.

#### 2.8.3.1. Influence of processor usage

Seven of the fourteen full query processing systems use CPUs and GPUs as generic compute resources (*generic systems*). Specifically, these are systems that process arbitrary relational or stream queries, e.g., GDB [116], CoGaDB [42], He et al. [121], DB2 BLU [201], HERO [145], HetExchange [54], and SABER [163]. Additionally, two systems, He et al. [121] and DIDO [355] use CPUs and GPUs as hybrid resources (*hybrid systems*). In contrast, eight of the fourteen systems implementing an individual query processing task use CPUs and GPUs as specialized processors (*specialized systems*). Since they focus on a specific operation, these systems implement a heterogeneous processing strategy that assigns specific tasks to each processor.

The processor usage in turn strongly influences the remaining scheduling decisions. Specialized systems map specific tasks to the most suitable processor, i.e., they execute a static schedule based on the nature of the task. In our survey, the only exception is the HB$^+$-tree [283] which uses a cost model to determine how many nodes below the root are processed on the CPU instead of the GPU.

In contrast, twelve of the fifteen generic or hybrid systems execute a dynamic or hybrid schedule. Since these systems can execute tasks on any processor, they often defer this decision to runtime. However, GDB [116], CoGaDB [42], and He et al. [121] execute a static physical query plan determined by a cost model during query compilation. Generic or hybrid systems utilize all scheduling strategies and nine of them use a combination of different strategies. Nine systems use cost models, eight use explicit load balancing, six use task nature, and three use explicit data locality as their scheduling strategy. Additionally, another four and three systems use their cost models to implicitly balance processor load and exploit data locality, respectively.

Eight of the fifteen generic or hybrid systems distribute the workload using task-specific partitions. CoGaDB [42], and HERO [145], DIDO [355], and FineStream [352] implement an operator placement strategy. Beier et al. [28] and Bøgh et al. [34] implement a single operation and distribute the data to perform heterogeneous processing. SABER [163] distributes complete queries to the CPU or the GPU.

Six of the eight generic or hybrid systems that employ cost models implement fine-grained tasks. There are two explanations. (1) CoGaDB [42] and HERO [145] implement operator placement strategies for relational queries based on learned cost models. For this use case, Karnagel et al. [145] argue that fine-grained tasks exhibit less runtime variation and are therefore more suitable for the cost model than coarse tasks. (2) The remaining systems, DIDO [355], He et al. [120, 121], and FineStream [352], are implemented on integrated GPUs. These GPUs favor fine-grained tasks as we discuss next.

#### 2.8.3.2. Influence of GPU integration

23 of the 28 surveyed query processing systems target dedicated GPUs. Since dedicated GPUs offer high compute performance and GPU memory bandwidth, they are interesting research targets. Conversely, integrated GPUs, which are traditionally targeted towards the mobile market, are less powerful and possibly attract fewer researchers.

However, research results based on dedicated GPUs are not necessarily transferable to integrated GPUs. The GPU integration strongly influences how to implement a heterogeneous query processing system, specifically the task granularity. Every system in our survey targeting integrated GPUs employs fine-grained tasks. Indeed, the close cooperation between CPU and GPU compute units is one of the main advantages of these systems. In contrast, systems targeting dedicated GPUs generally use coarse tasks. In our survey, the only exceptions are CoGaDB [42] and HERO [145]. As we already discussed, fine-grained tasks are suitable for their learned cost models because they exhibit less runtime variation.

The dependence of the task granularity on the type of GPU integration strongly indicates that the data transfer over the slow system bus is the main bottleneck for heterogeneous query processing on dedicated GPU. Coarse tasks are more efficient for dedicated GPUs because there is less communication over the system bus. In contrast, on integrated GPUs, a heterogeneous query processing system has more freedom to schedule fine-grained tasks on the most suitable processor.

Table 2.16.: Processing model and operator implementations for heterogeneous query processing systems that use CPUs and GPUs as generic compute resources.

(a) Full query processing systems

| Publication | Processing model | Operator implementation |
|---|---|---|
| GDB [116] | Bulk | Separate |
| CoGaDB [42] | Bulk | Separate |
| He et al. [121] | Vectorization | Oblivious |
| SABER [163] | Bulk | Separate |
| DB2 BLU [201] | Bulk | Separate |
| DIDO [355] | *Fixed pipeline* | Oblivious |
| HERO [145] | Bulk | Separate |
| HetExchange [54] | Compilation | Template |
| FineStream [352] | Bulk | Oblivious |

(b) Individual query processing tasks

| Publication | Processing model | Operator implementation |
|---|---|---|
| SCCG [333] | *Fixed pipeline* | Separate |
| Beier et al. [28] | *Fixed pipeline* | Separate |
| He et al. [120] | Vectorization | Oblivious |
| Bøgh et al. [34] | *Single operator* | Template |
| Gubner et al. [106] | Vectorization | Separate |
| Lutz et al. [189] | *Single operator* | Separate |

#### 2.8.3.3. Processing model and operator implementations

Six of the nine generic or hybrid full query processing systems use a bulk processing model. Only HetExchange [54] employs query compilation and He et al. [121] employ in-cache vectorization on integrated GPUs. Additionally, Caldera [19] and Raza et al. [261] employ query compilation in the implementation of their HTAP systems.

Nine of the fifteen generic or hybrid systems execute separate operator implementations on CPUs and GPUs. One exception is HetExchange [54] which combines query compilation with template-based operator implementations. He et al. [121], DIDO [355], FineStream [352], and He et al. [120], use a hardware-oblivious operator implementation on both processors. However, we suspect that this is due to a selection bias. Each of these systems targets an integrated GPU, specifically an AMD Llano or Kaveri processor [40, 41]. These processors are not supported by CUDA. Instead, the authors program them

Table 2.17.: Support for out-of-core processing (OOC) and data transfer optimizations for heterogeneous query processing systems on dedicated GPUs.

(a) Full query processing systems

| Publication | OOC | Optimizations |
|---|---|---|
| GDB [116] | Yes | Compression |
| Approximate & Refine [247] | No | Compression, Approximation |
| CoGaDB [42] | No | Overlap, Caching, Locality |
| Stat. coproc. [122] | — | Heterogeneous execution |
| Mega-KV [356] | Yes | Approximation |
| SABER [163] | Yes | Overlap |
| DB2 BLU [201] | Yes | Overlap, Single-pass algorithm |
| Caldera [19] | Yes | Overlap |
| HERO [145] | Yes | Caching, Locality |
| HetExchange [54] | Yes | Overlap, Single-pass algorithm, Locality |
| Raza et al. [261] | Yes | Overlap, Caching, Fast interconnect |

(b) Individual query processing tasks

| Publication | OOC | Optimizations |
|---|---|---|
| SCCG [333] | Yes | — |
| Beier et al. [28] | Yes | Caching |
| GSS [33] | No | — |
| STIG [70] | Yes | Heterogeneous execution |
| HB$^+$-tree [283] | No | Overlap, Heterogeneous execution |
| Stehle et al. [298] | Yes | Overlap, Heterogeneous execution |
| Bøgh et al. [34] | No | — |
| G-Grid [177] | No | Heterogeneous execution |
| GAT [351] | Yes | Caching, Heterogeneous execution |
| Sioulas et al. [289] | Yes | Overlap |
| Gubner et al. [106] | No | Overlap |
| Lutz et al. [189] | Yes | Fast interconnect, Overlap |

in OpenCL. Ocelot [123], which proposed hardware-oblivious operator implementations based on OpenCL, targets any type of GPU. However, it appears that users rather program dedicated NVIDIA GPUs in CUDA because of its advanced features and better tool support.

### 2.8.3.4. Data size limitations

Fifteen of the 23 query processing systems targeting dedicated GPUs support out-of-core processing by horizontally partitioning the data. In principle, these systems can scale to arbitrarily large data sizes. GSS [33] and Bøgh et al. [34] perform a compute-intensive skyline or skycube computation on a small amount of data that is measured in a few hundred MBs. Similarly, G-Grid [177] performs a compute-intensive spatial query on a road network. The largest evaluated dataset is 3.5 GiB which comfortably fits into the memory of dedicated GPUs.

Even query processing systems that do not scale arbitrarily can process data that exceeds the GPU memory. For example, CoGaDB [42] partitions data vertically and is limited by columns that have to fit on the GPU. In Approximate & Refine [247], the available GPU memory limits the size of the approximate data set stored on the GPU. The HB$^+$-tree [283] stores inner nodes of an B$^+$ tree on the GPU but not its leaves. In Gubner et al. [106], the GPU stores a bloom filter which is much smaller than the corresponding key column. Finally, the statistical coprocessor [122] uses the GPU for selectivity estimation during query optimization.

### 2.8.3.5. Data transfer bottleneck mitigation

Overlapping data transfers with computation is the most popular technique to mitigate the data transfer bottleneck and is implemented by ten of the 23 systems targeting dedicated GPUs. This is not surprising, since this technique is well supported by GPU hardware and simplifies the algorithm design. Five of the eleven systems that implement individual query processing tasks on dedicated GPUs, design their heterogeneous processing strategy to reduce data transfers.

Only two systems employ lossless compression for arbitrary data types, i.e., GDB [80, 116], which is the oldest of the surveyed relational query processors, and Approximate & Refine [247, 249]. This is surprising, since GPUs are well suited for compression [80]. A possible reason may be that a CPU-optimized storage layout for compressed data is not directly suitable for GPUs [271]. He et al. [121] also employ compression, however, since this system targets integrated GPUs, we do not list it in Table 2.17. Furthermore, CoGaDB [42] employs dictionary encoding to store strings; SABER [163] employs frame-of-reference encoding to compress write positions for selection and join results; the HELLS join [148] employs bitmap compression for join matches; and Beier et al. [28] transform the sparse index lookup result matrix into a dense representation. Since these systems do not employ compression in a generalized way, we also do not list them in Table 2.17. Finally, Approximate & Refine [247] and Mega-KV [356] employ lossy compression, i.e., approximation, to reduce data transfers.

### 2.8.4. Summary

Our survey shows that the processor usage pattern and the type of GPU integration have the strongest influence on scheduling decisions. Query processing systems that use CPUs and GPUs as specialized processors for specific tasks generally execute a static schedule based on the nature of the task. In contrast, systems that use CPUs and GPUs as generic compute resources exhibit a large diversity in their scheduling decisions. Systems that target dedicated GPUs generally use coarse-grained tasks whereas systems that target integrated GPUs use fine-grained tasks.

Most query processing systems can scale to data sizes that exceed dedicated GPU memory. Compression is underused as a technique to reduce the data transfer bottleneck, especially since GPUs are well suited to it.

## 2.9. Key insights and open research problems

The key insight of our survey is that *heterogeneous systems with dedicated and integrated GPUs are different classes and place different demands on heterogeneous query processing*. In both cases, the GPUs have similar strengths and weakness, and are used for comparable tasks. However, as we discuss next, the key to achieve high performance is different for dedicated and integrated GPUs.

### 2.9.1. Performances guidelines for query processing on dedicated and integrated GPUs

*Dedicated GPUs should perform specialized coarse-grained tasks.* On these systems, the slow interconnect is the main performance bottleneck and data transfers should be avoided as much as possible. This bottleneck precludes fine-grained cooperation between the CPU and GPU, which relies on frequent and low-latency exchange of data. For example, instead of scheduling fine-grained primitives, the query processing system should schedule compiled query pipelines [54, 87]. It is worthwhile to investigate how to reformulate query processing problems to radically reduce transferred data, e.g., using techniques such as approximate processing on the GPU [247, 356].

*Integrated GPUs should cooperate closely with the CPU.* These systems are not constrained by the data transfer bottleneck. Instead, CPU and GPU compute units are connected by a fast and coherent on-chip fabric and can modify shared data structures simultaneously. This architecture makes it possible to schedule fine-grained tasks on the most suitable compute unit. For example, CPU compute units can prefetch sequential data [121], whereas GPU compute units can hide the latency of random accesses [355]. We are not aware of publications investigating how to accelerate hybrid transactional/analytical processing systems, or spatio-temporal queries through coprocessing on integrated GPUs. The architecture of integrated GPUs is similar to that of asymmetric multi-core processors and it is worthwhile to investigate heterogeneous query processing techniques [204] on these processors.

### 2.9.2. Open research problems

In addition to missing research into query processing applications on integrated GPUs, we also identified the following five open research areas regarding heterogeneous query processing on GPUs in general. These cover a lack of understanding of the performance of atomic operations on GPUs, GPU hardware features that have not yet been exploited for query processing, and the integration of CPU and GPU query processing models.

*(1) How do atomic operations influence GPU performance?* Fast single-pass GPU algorithms, as well as fine-grained cooperation between integrated CPU and GPU compute units, rely on atomics to modify shared data structures. So far, the benefits of these techniques have been shown experimentally but there is no theoretical model of the performance of atomics on GPUs. Such a model would help to analyze algorithms, characterize hardware, and drive scheduling decisions.

*(2) What is the tradeoff between data transfer and cooperation on fast coherent interconnects?* Fast dedicated GPU interconnects, i.e., NVLink 2.0 [225], Infinity Fabric [241], or Compute Express Link [284], offer higher bandwidth, lower latency, and coherent access. However, they are still slower than CPU main memory access and do not fully eliminate the data transfer bottleneck. We are only aware of work by Lutz et al. [189] who explore how these interconnects can be used for more fine-grained cooperation between CPUs and dedicated GPUs in the context of hash joins.

*(3) How can databases leverage dedicated hardware for matrix operations?* Recent GPUs contain dedicated hardware to accelerate matrix operations of machine learning workloads [30, 225]. These GPUs are promising processors to execute complex pipelines that integrate traditional query processing with machine learning, as well as to leverage machine learning for query optimization [3].

*(4) What is faster on GPUs, vectorization or query compilation?* State-of-the-art processing models such as vectorization [38] and query compilation [208] have been

implemented on GPUs [46, 87, 89, 121, 164, 244]. However, so far there is no systematic comparison between these processing models on GPUs similar to the work by Kersten et al. [151] and Gubner et al. [105] on CPUs. It is likely that the best processing model depends on the type of GPU integration. Query compilation creates coarse-grained tasks that are beneficial on dedicated GPUs. Vectorization supports efficient cooperation of fine-grained tasks on integrated GPUs.

*(5) How can we adapt the processing model and query plans to different processors?* Every relational query processor for heterogeneous CPU/GPU systems that we studied makes two simplifying decisions. (a) It uses the same query processing model on CPUs and GPUs. (b) It executes a physical query plan generated by a query optimizer that targets CPUs. However, as we discussed in the previous question, the fastest query processing model may depend on the processor. Furthermore, on the level of primitives, a different query plan may be faster on the GPU [9]. A heterogeneous query processor could use different processing models and query plans for coarse-grained tasks that run on each processor [103].

### 2.9.3. Conclusion

The query processing systems studied in our survey show that heterogeneous query processing can effectively leverage GPUs to improve query performance. However, our survey reveals that the research community often focuses on dedicated GPUs, and treats integrated GPUs as an afterthought. Since dedicated and integrated GPUs require different performance optimizations, the insights generated for dedicated GPUs are not always generalizable.

We also found that the implementation of relational heterogeneous query processors is very CPU-centric. Specifically, the processing model on the GPU is typically determined by the processing model chosen for the CPU. Further research is needed to identify which processing model is the fastest on GPUs and how to combine different processing models on CPUs and GPUs.

Fortunately, GPU vendors continue to innovate on the hardware capabilities of GPUs. This innovation generates exciting research opportunities, especially in the context of integrating machine learning with relational query processing.

# 3 Operator variant tuning on heterogeneous processors

## 3.1. Problem statement

In the previous chapter, we examined the architectural differences between CPUs and GPUs, and discussed the implications of a heterogeneous system architecture, which contains both CPU and GPU processors, on the design of a query processing system. We noted that the different hardware architectures of CPUs and GPUs require us to formulate programs in a processor-specific way to fully exploit their processing capabilities. For example, we discussed in Section 2.3.4.3 that on CPUs, each processing core should execute a single thread which works independently of the others, whereas on GPUs, we typically have to run many thousands of threads which cooperate with each other. Furthermore, low-level implementation details, such as the memory access pattern, branched or branch-free execution, or loop unrolling, have a different effect on CPUs and GPUs. It follows that we need to adapt the data processing code in a heterogeneous query processor to the specific processor it runs on. We call this problem the *operator variant selection problem on heterogeneous hardware*, and examine it in detail in this chapter.

A straightforward approach is to include dedicated implementations for each processor type, i.e., one operator implementation for CPUs and another one for GPUs. This approach is taken by GDB [116], which, to our knowledge, was the first full relational query processor for heterogeneous CPU/GPU systems. GDB includes an operator implementation written in CUDA [211] for the GPU and another one written in OpenMP [65] for the CPU. The advantage of this approach is that each operator implementation can be independently optimized for a specific processor architecture. However, the need to develop and optimize dedicated implementations for multiple processors is also a major disadvantage, as it increases development costs. Both implementations are functionally equivalent, but they require developers to know the details of different hardware architectures, and tuning takes time for each processor.

To reduce development costs, Heimel et al. [123] propose a design based on a single hardware-oblivious operator implementation, which is written in OpenCL [300] and runs on CPUs, GPUs, and other supported parallel processors. The OpenCL specification defines an abstract programming model, which is translated by a vendor-provided driver to a processor-specific binary, and allows developers to write scalable programs that run on a variety of parallel processors (see also Section 2.3.4). However, OpenCL only provides functional portability, but it does not ensure performance portability [272]. The specification guarantees that an OpenCL program will run on any supported device [317], but not necessarily at the best performance. Thus, hardware-oblivious operators written in OpenCL do not solve the problem that the operator implementation of a heterogeneous database has to be adapted to different processors, and optimized for each processor individually.

In this chapter, we propose that a data processing system exploits runtime performance feedback to automatically learn an operator implementation that is specifically adapted to the processor it runs on. Previous work exploits runtime performance feedback to select between different operator implementations to react to changing data characteristics during the execution of a query [258]. However, the diversity of heterogeneous processors makes this process much more challenging since the search space of possible operator implementations is considerably larger. We also want the query processing system to adapt its operator implementation to new parallel processor architectures, which are proposed from time to time, e.g., the Intel Xeon Phi [56] or the Cell Broadband Engine [141], and so we do not necessarily know which implementations are good candidates to choose from. Finally, as we will show in this chapter, GPUs are very sensitive to implementation parameters and can suffer a large performance penalty if the implementation is not carefully tuned to the specific GPU model.

## 3.2. Contributions

In this chapter, we experimentally motivate the operator variant selection on heterogeneous hardware, using selection and hash aggregation as two operator examples, and propose two algorithms towards solving it. Specifically, we make the following contributions:

(1) We show that we can generate a large number of implementations of the selection operator from a few implementation parameters, and perform an extensive experimental evaluation of these implementations on seven CPUs from AMD, IBM, and Intel; five GPUs from AMD, Intel, and Nvidia; and an Intel Xeon Phi coprocessor. Our evaluation illustrates the diversity of existing heterogeneous processor architectures with regard to the influence of implementation parameters on performance (Section 3.3).

(2) Digging deeper, we examine the influence of different execution parameters on GPU-accelerated hash aggregation on four NVIDIA and two AMD GPUs based on six different microarchitectures. We find that the optimal execution parameters are highly GPU-specific and that implementations optimized for a specific GPU are up to 21× slower on other GPUs, which indicates that previously derived heuristics for GPU-accelerated hash aggregation [149] cannot be generalized to other GPUs (Section 3.4).

(3) Based on our analysis, we develop an algorithm to learn fast operator implementations at runtime. The algorithm is based on Micro Adaptivity [258] but extends it with a search strategy to handle large search spaces. Our evaluation shows that it can significantly improve the performance of the selected operator implementation over a number of queries but that the absolute performance it achieves strongly depends on random initial conditions (Section 3.6).

(4) We develop a second algorithm to address some of these short-comings. It is based on a local search that follows a performance gradient in the search space, and extends it with a mechanism to handle performance plateaus and runtime variation. Crucially, it incorporates knowledge about the operator performance characteristics to restrict and guide the search (Section 3.7).

The key insight of this chapter is that processors differ greatly in how sensitive they are to deviations from optimal implementation parameters. On some of the processors we evaluated, such as the Intel Xeon Phi 7210, only a small number of combinations of implementation numbers result in fast operator implementations. Therefore, it is crucial to exploit available information about the performance characteristics of a particular processor, in order to restrict the search space and guide the search, and to avoid particularly bad operator implementations.

## 3.3. The operator variant selection problem on heterogeneous hardware

In this section, we experimentally motivate the operator variant selection problem on modern heterogeneous processors. In particular, we show that we can derive thousands of operator implementations from a small set of implementation parameters, which change implementation details, but do not change the semantics of the operator. Throughout this chapter, we use the term *variant* as a synonym for an operator implementation. Concretely, an operator variant is an assignment of a set of implementation parameters to specific values, which fully describes a concrete implementation of an operator.

We use a selection kernel as an example to illustrate how to construct operator variants. The selection kernel scans an array of input values, evaluates a predicate on each value, and constructs a bitmap which stores the result of the evaluation. Such a kernel is often used as a building block of a complete selection operator in GPU databases [123].

We construct variants of this selection kernel in three steps. First, we formulate a set of basic variants which specify the memory access pattern of the input array and how threads cooperate to construct the result bitmap. Second, we modify each basic variant by changing low-level implementation details, such as branch-free evaluation of the selection predicate, or unrolling kernel loops. Third, we vary how to distribute the workload on the threads running in parallel on the processor. In total, we construct variants for the selection kernel based on six implementation parameters.

After describing how to construct operator variants, we measure their performance on a diverse set of processors from different manufacturers, including seven CPUs, five GPUs, and an Intel Xeon Phi coprocessor. Our evaluation shows that there is no single variant which performs well on every processor. In fact, the fastest variant depends on the processor type, the manufacturer, and even the processor microarchitecture. Furthermore, we find that the implementation parameters influence performance in different processor-specific ways and that some processors are more tolerant to the selection of implementation parameters than others.

### 3.3.1. Basic selection kernel variants

We use six different basic variants for the selection kernel, which are illustrated schematically in Figure 3.1. These variants use different algorithms to produce the same result bitmap for a given input. Specifically, they differ in how they access the input array and how they cooperate to construct the result bitmaps.

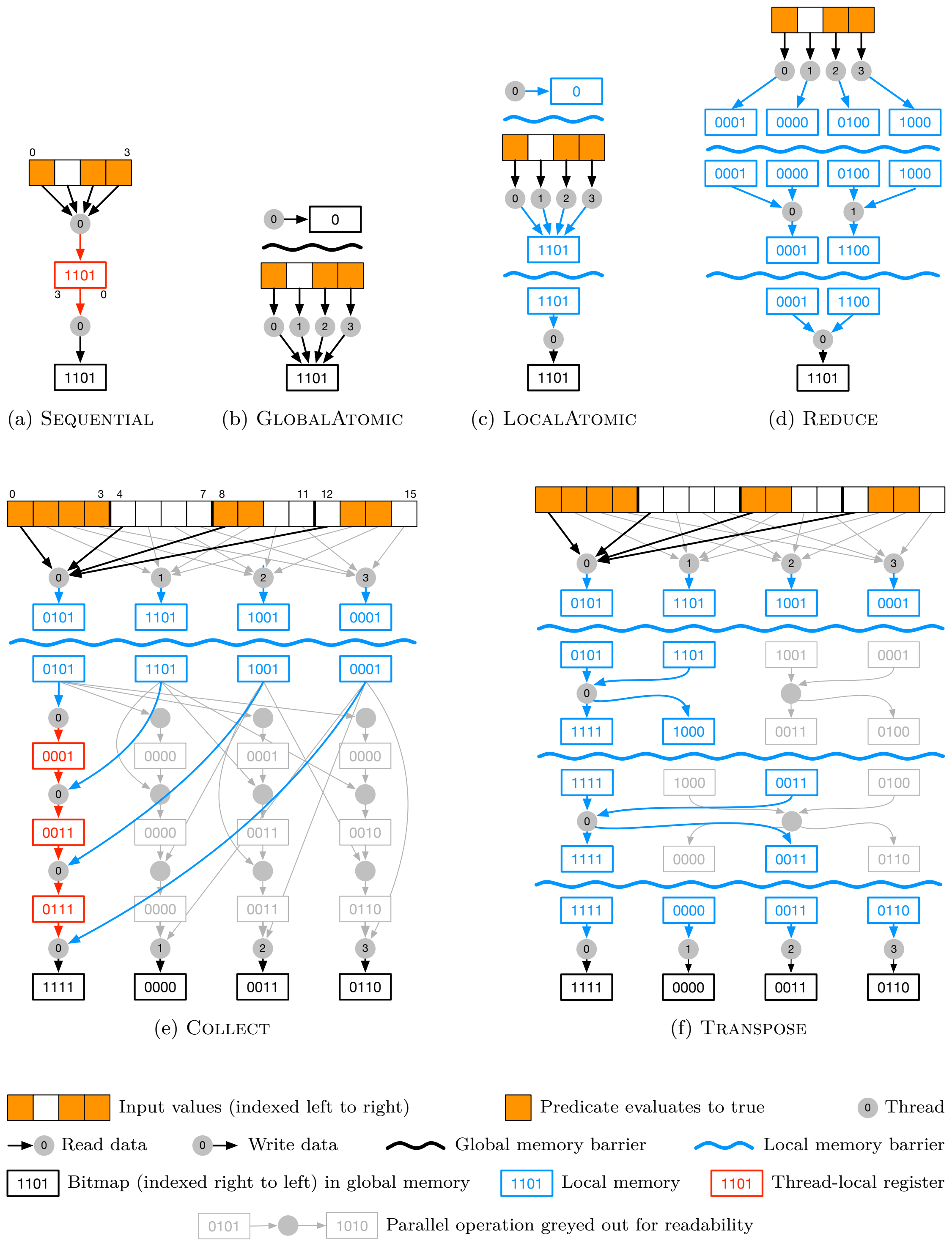


Figure 3.1.: Basic selection kernel variants.

Figure 3.1a shows the simplest possible variant, which we call SEQUENTIAL. Each thread works on a continuous region of the input array, independently of the other threads. The thread evaluates the predicate for a few consecutive values, creates the resulting bitmap element in a register, and writes it out to the corresponding global memory address. The memory access pattern of SEQUENTIAL does not utilize GPU hardware efficiently, because it does not provide opportunities to coalesce memory accesses from multiple threads to the same memory region. Instead, on GPUs, we want to interleave the access pattern of neighboring threads, so that they evaluate the selection predicate on neighboring input values [213, §9.2.1].

A straightforward interleaved variant, called GLOBALATOMIC, is shown in Figure 3.1b. The threads in a work group create the result bitmap by directly setting the corresponding bits in parallel, using atomic operations [154, §6.12.11]. In order to ensure correctness, the work group first has to zero-initialize the result memory, after which it has to synchronize the threads via a global memory barrier. LOCALATOMIC, shown in Figure 3.1c, is a slightly modified version of the previous variant. Here, the threads construct the bitmap in local memory. After synchronizing via a local memory barrier, the work group then copies the result to its final position in global memory. A major disadvantage of GLOBALATOMIC and LOCALATOMIC is their reliance on atomic operations. For predicates that are satisfied by a large percentage of the input data, these variants suffer from severe thread contention, as all atomic operations on the same address are serialized by the hardware [214, §4.1].

Figure 3.1d shows the REDUCE variant, which does not rely on atomics. Each thread evaluates the predicate for one value, creates a bitmap element with the corresponding bit set, and writes it to local memory. The threads in a work group then cooperate to create the result bitmap using a parallel reduction algorithm [130]. A drawback of REDUCE is its wastefulness with regard to local memory. Each thread only sets one bit, but stores a full bitmap element in local memory. This resource consumption limits the amount of threads that can run concurrently on the GPU.

The final two variants, COLLECT and TRANSPOSE, which are shown in Figure 3.1e and Figure 3.1f, respectively, use local memory resources more efficiently. The first part of the algorithm is the same for both variants. Similarly to SEQUENTIAL, each thread first evaluates the predicate for multiple tuples, creating a bitmap element in local memory. However, due to the interleaved memory access, the bit pattern in these elements will also be interleaved, forcing us to restore the correct bit order before writing to global memory. Essentially, the intermediate bitmaps can be interpreted as a square matrix which we need to transpose, and the variants differ in how this happens. Each thread of COLLECT builds one element of the result bitmap in a register by subsequently collecting the required bits using bit masks and bit shifts. This algorithm scales linearly with the number of bits per bitmap element and does not require memory barriers. TRANSPOSE uses the threads of the work group to cooperatively transpose increasingly larger tiles of the interleaved bitmap elements in local memory, again using bit masks and bit shifts [16]. This algorithm scales logarithmically, but it requires an additional memory barrier between each step.

### 3.3.2. Low-level implementation parameters

The algorithms described by the basic selection kernel variants in the previous section can be further adapted by modifying low-level implementation details. Again, changing these implementation details does not change the produced result. Concretely, we modify the following three implementation details.

#### 3.3.2.1. Result type

In Figure 3.1, we illustrated the basic variants for bitmap elements containing four bits. In actuality, we create bitmaps at the granularity of 8, 16, 32, or 64 bits, using the OpenCL types `uchar`, `ushort`, `uint`, and `ulong` [154, §6.1.1]. The size of the bitmap elements determines the number of steps required to reduce the initial 1-bit bitmaps in the REDUCE variant, and to fix the interleaved bit order in the COLLECT and TRANSPOSE variants.

#### 3.3.2.2. Loop unrolling

We optionally remove `for` loops entirely by replicating the loop body the required number of times.

#### 3.3.2.3. Branch-free evaluation

Instead of evaluating the predicate and setting the bit in the result bitmap using an `if` statement, we can also set bits unconditionally using branch-free evaluation. On CPUs, branch-free evaluation avoids costly branch mispredictions when the selectivity of the predicate is neither very low nor very high [258].

### 3.3.3. Workload distribution parameters

The abstract programming model of OpenCL requires developers to partition a problem into work groups, which are processed independently of each other on a dedicated compute unit of the parallel processor (see Section 2.3.4.2). Within a work group, the threads can cooperate to produce the result. Of the six basic selection kernel variants, LocalAtomic, Reduce, Collect, and Transpose cooperatively construct the result bitmap in local memory, before writing it out to global memory. Moreover, on a GPU, we have to schedule multiple work groups per compute unit to effectively hide memory access latency. Thus, we can further construct additional variants by varying the workload distribution using the following two parameters.

#### 3.3.3.1. Work group size

The work group size is an OpenCL parameter that is specified when a kernel is launched [154, §5.8]. It determines the number of work items in each work group, i.e., the number of work items that can cooperate to solve a subproblem.

#### 3.3.3.2. Elements per thread

To reduce the overhead for each thread, the variants are modified to produce additional result bitmap elements by processing a range of input values sequentially. This parameter indirectly determines the number of work groups per compute unit. Given a fixed input size and work group size, the more elements are processed by a thread, the smaller the number of work groups among which the input is partitioned.

To effectively distribute the work load on a parallel processor, there have to be at least as many work groups as there are independent processing units. Beyond, the ideal number of work groups is a processor-specific tradeoff. On Nvidia GPUs, more and smaller work groups provide the GPU scheduler with more flexibility to hide the memory access latency [213, §10.3]. In contrast, on the AMD Radeon HD 9650, larger work loads provide the compiler more opportunities to fill VLIW instructions [6, §3.6.3]. On Intel CPUs, having fewer work groups reduces scheduling overheads [133, §2.9].

### 3.3.4. Variant universe

Based on the six base variants, the three low-level implementation parameters, and the two workload partition parameters, we can generate close to six thousand different variants of the selection operator. Not all combinations of implementation parameters result in valid variants. For all basic kernel variants except Sequential, the number of work items must be a multiple of the number of bits in the result bitmap element. GlobalAtomic and LocalAtomic always use branched evaluation, because the result bitmap is zero-initialized at the beginning. Conversely, the Reduce variant always evaluates the predicate branch-free because this operation can be combined with clearing the other bits of the intermediate result bitmaps in local memory. GlobalAtomic and LocalAtomic also do not unroll loops, and OpenCL atomic operations are generally only defined on 32-bit integers, as well as on 64-bit longs on AMD CPUs.

The exact number of possible variants is device-specific. For example, the maximum work group size can range from as low as 256 on AMD GPUs to as high as 8192 on Intel CPUs and on the Intel Xeon Phi. Many OpenCL implementations also suggest a minimal work group size. AMD and Nvidia suggest a multiple of 64 or 32 work items, respectively, to fill warps on Nvidia GPUs [213, §10.3] and wavefronts on AMD GPUs [6, §3.6.3]. Intel suggests a multiple of eight or 16 work items to enable auto-vectorization on Intel CPUs [133, §2.8] or on the Intel Xeon Phi [132], respectively. However, this is not a hard rule, as in some cases other resources than processing cores, e.g., the TLB cache on GPUs, can constrain overall performance [149]. We therefore set the minimal work group size to one on CPUs and to eight on GPUs and the Intel Xeon Phi.

Most importantly, we have to set the workload parameters in a way which parallelizes the workload effectively on the available processing resources and does not leave compute units idle. Note that we cannot always retrieve this information from the OpenCL runtime in a straightforward manner. For example, on Nvidia GPUs and the AMD Radeon HD 6950, compute units correspond to independent streaming multiprocessors (SMs) or SIMD cores, respectively, and therefore the number of compute units provides a lower bound to the number of work groups that should be launched. However, on the Intel Iris 5100, the reported number of compute units correspond to execution units (EUs), which are grouped into subslices and slices and which are not independent from each other [139, §5.3]. On this processor, we should not derive the minimal number of work groups from the number of compute units. On Intel CPUs and on the Intel Xeon Phi, which support multiple hardware threads per physical core, the number of compute units corresponds to the number of logical cores.

In Table 3.1, we report the number of variants constructed on each processor, depending on the number of processing resources, the maximal work group size, and whether the OpenCL runtime supports 64-bit atomics. (Note that the number of variants differs from our previously published work [268] because we exclude variants which do not effectively parallelize the workload over the available processing cores, as described in the previous paragraph.)

Table 3.1.: Number of constructed variants based on properties of the processor and the OpenCL runtime.

| Processor | 64-bit atomics | Compute units | Independent cores/SMs/slices | Maximum work group size | Variants |
|---|---|---|---|---|---|
| AMD Opteron 2356[1] | ✓ | 8 | 8 | 1024 | 4974 |
| AMD Opteron 6128 HE[1] | ✓ | 16 | 16 | 1024 | 4806 |
| IBM 8231-E2B | | 48 | 12 | 1024 | 4980 |
| Intel Core i7-870 | | 8 | 4 | 8192 | 5884 |
| Intel Xeon E5620 | | 8 | 4 | 8192 | 5880 |
| Intel Xeon E5-2650 v2[1] | | 32 | 16 | 8192 | 5414 |
| Intel Core i7-4900MQ | | 8 | 4 | 8192 | 5880 |
| AMD Radeon HD 6950 | | 22 | 22 | 256 | 3104 |
| Intel Iris 5100 | | 40 | 2 | 512 | 3750 |
| Nvidia GeForce GTX 460 | | 7 | 7 | 1024 | 4160 |
| Nvidia Quadro K2100M | | 3 | 3 | 1024 | 4280 |
| Nvidia Tesla K40M | | 15 | 15 | 1024 | 3992 |
| Intel Xeon Phi 7120 | | 240 | 60 | 8192 | 3374 |

[1] Two-socket system.

## 3.3.5. Performance analysis on heterogeneous processors

In the following, we measure the performance of the selection operator variants constructed in the previous section on a diverse set of processors, including seven CPUs from AMD, IBM, and Intel; five GPUs from AMD, Intel, and Nvidia; as well as an Intel Xeon Phi coprocessor. Our goal is to determine the processor-specific influence of the six implementation parameters on the performance of the selection operator.

Using a custom code generator, we generate an OpenCL kernel for every valid combination of the six implementation parameters from the variant universe. The OpenCL kernel computes a simple less than predicate with a fixed selectivity, which we vary from 0 to 1 in steps of 0.1. We evaluate each variant on an array of 32 million random integers, i.e., the input size is 128 MB, which exceeds the largest cache on every processor. For each selectivity, we run each kernel ten times and compute the median run time.

The heat maps in Figure 3.2 summarize the results. The figure is arranged in a grid with four dimensions: the processor (columns in the plot grid), the basic kernel variant (rows in the plot grid), a combination of low-level implementation parameters (rows in individual heat map plots), and the predicate selectivity (columns in individual heat map plots). Each heat map tile shows the performance of the fastest variant given the four dimensions, i.e., the variant with optimal workload parameters. For each processor and selectivity, we also identify the fastest variant, and indicate it with the symbol ◘.

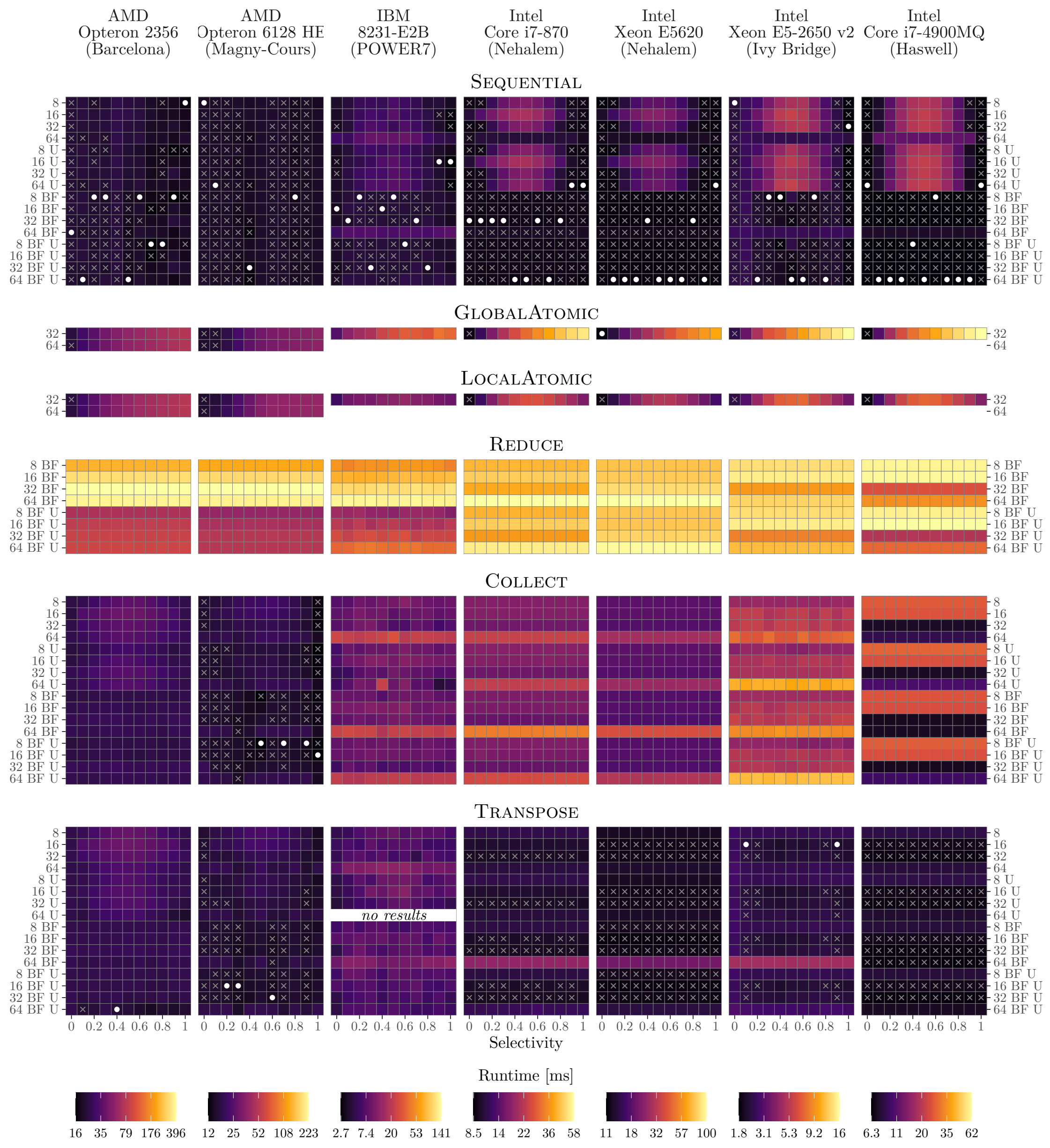


Figure 3.2.: Selection operator performance on different processors (CPUs). The labels on the y axis indicate the bit size of the result type and whether the variant is branch-free and/or unrolled. ● indicates the fastest variant for a given selectivity. ☒ indicates a variant that is at most 10% slower than the fastest. (Figure continues on next page.)

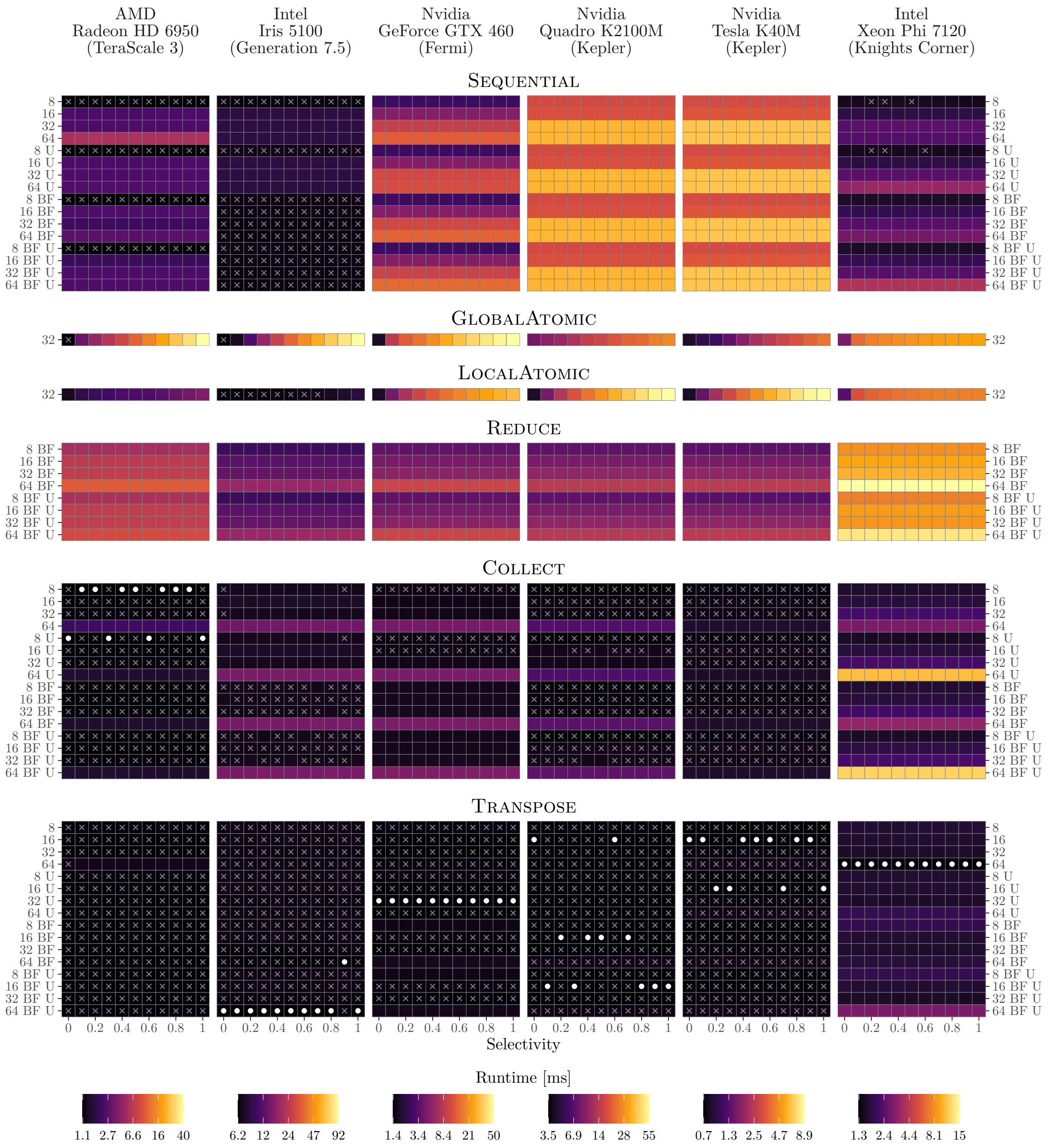


Figure 3.2.: Selection operator performance on different processors (continued from previous page, GPUs and Intel Xeon Phi).

In addition, we indicate variants that are at most 10% slower than the fastest with the symbol ☒. At the bottom of every processor column, we show the range of the kernel run times in milliseconds.

Note that not all combinations of basic kernel variant and low-level implementation parameters create valid variants, as we described in Section 3.3.4. Furthermore, when we verified the output of the selection kernel, we noticed that the 64-bit unrolled TRANSPOSE kernel produced the wrong results on the IBM 8231-E2B CPU.

### 3.3.6. Processor diversity

The heat maps in Figure 3.2 clearly show the diversity of the tested processors with regard to the performance of each variant. How well a variant performs compared to the others depends on the processor type, the manufacturer, and the microarchitecture.

#### 3.3.6.1. Differences between CPUs, GPUs, and the Xeon Phi

As expected, a simple SEQUENTIAL kernel is the fastest variant on CPUs. Furthermore, if the selectivity of the predicate is not either very low or very high, a branch-free kernel outperforms a branched kernel, which is consistent with previous research [48, 258]. Conversely, on GPUs and on the Xeon Phi, a more complex kernel, in which multiple threads cooperate to produce the result, is faster, i.e., COLLECT or TRANSPOSE.

#### 3.3.6.2. Differences between processor manufacturers

On GPUs, a COLLECT kernel is the fastest variant on the AMD Radeon HD 6950 GPU, whereas a TRANSPOSE kernel is fastest on the Intel Iris 5100, the Nvidia GPUs, and also the Intel Xeon Phi. Also note that on the Iris 5100, the LOCALATOMIC kernel is at most 10% slower than the fastest variant if the selectivity is below 0.7. On other GPUs, and on the Xeon Phi, the LOCALATOMIC kernels are much slower.

On CPUs, it is noteworthy that GLOBALATOMIC and LOCALATOMIC show the same performance on AMD GPUs. Conversely, on IBM and Intel CPUs, LOCALATOMIC is faster than GLOBALATOMIC as the selectivity approaches 1, even though CPUs do not have dedicated local memory.

#### 3.3.6.3. Differences between microarchitectures from the same manufacturer

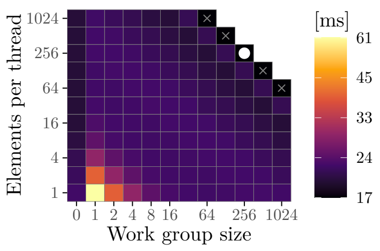
On the Nvidia GeForce GTX 460, which is based on the Fermi microarchitecture, a TRANSPOSE kernel with a 32-bit result size is the fastest variant, whereas a kernel with a 16-bit results size is the fastest on the two Nvidia GPUs based on the Kepler microarchitecture. Furthermore, the heat maps of the SEQUENTIAL kernel are noticeably different on both Nvidia microarchitectures, indicating that the low-level implementation parameters influence the performance of this kernel in different microarchitecture-specific ways. Similarly, the heat maps of the COLLECT kernel differ between the Nehalem, Ivy Bridge, and Haswell CPUs from Intel. Also note that on Kepler GPUs, LOCALATOMIC is slower than GLOBALATOMIC as the selectivity of the kernel approaches 1.

### 3.3.7. Influence of workload parameters

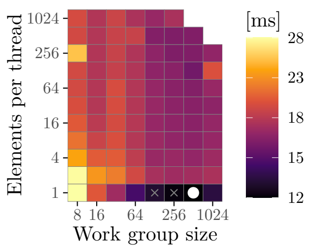
So far, we have analyzed the performance of different basic kernel variants and the influence of low-level implementation parameters. In the following we analyze the influence of workload parameters on the performance of variants. Figure 3.3 shows how the work group size and the number of elements per thread affect the runtime of the fastest variant from Figure 3.2 on page 79 at selectivity 0.5. The heat maps are irregular because some combinations do not represent valid variants, i.e., the number of work groups is smaller than the number of independent cores. Additionally, for kernels other than SEQUENTIAL, the number of bits in the result type determines the minimal work group size.

Again, the heat maps show the diversity of the processors with regard to the influence of the workload parameters on variant performance. On Intel CPUs, almost all variants are at most 10% slower than the fastest. The main exception is a diagonal line at the top-right corner which represents variants which do not utilize the available hyperthreads. Conversely, the AMD Opteron 2356 achieves the best performance only if the number of work groups precisely matches the number of hyperthreads. Deviating from the diagonal line results in a significant performance penalty. Similarly, on the AMD Opteron 6128 HE, the Intel Iris 5100, the Nvidia Tesla K40M, and the Intel Xeon Phi 7120, which use a COLLECT or TRANSPOSE kernel, only a small number of combinations of workload parameters result in optimal performance.

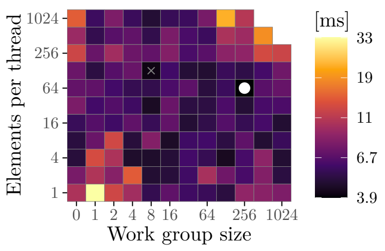
We can also see differences in the behavior of GPUs with regard to the number of work groups. On the AMD Radeon HD 6950 and the Nvidia GPUs, performance is maximized if the number of elements per thread is small and the number of work groups is correspondingly large. Having many active work groups is crucial to let the GPU scheduler hide the memory access latency [213, §10.3]. Conversely, on the Intel Iris 5100, performance is maximized if there are just four work groups, which correspond to the number of execution subslices [139, §5.3]. Note that this information is not available through the OpenCL runtime. If we restrict the number of work groups to the number of compute units, i.e., use fewer elements per thread, performance is significantly worse.



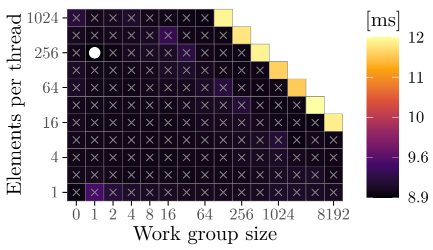
(a) Sequential 64-bit BF U
on AMD Opteron 2356



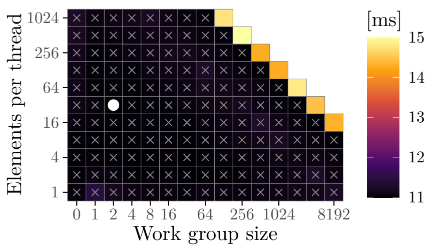
(b) Collect 8-bit BF U
on AMD Opteron 6128 HE



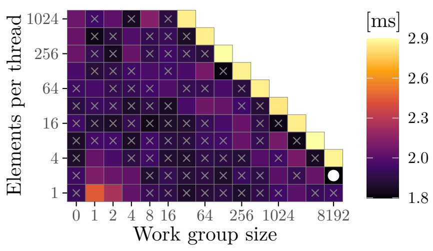
(c) Sequential 8-bit BF
on IBM 8231-E2B



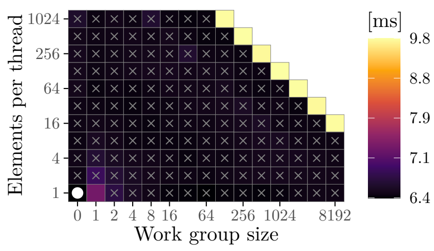
(d) Sequential 64-bit BF U
on Intel Core i7-870



(e) Sequential 64-bit BF U
on Intel Xeon E5620

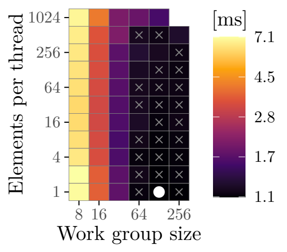


(f) Sequential 64-bit BF U
on Intel Xeon E5-2650 v2

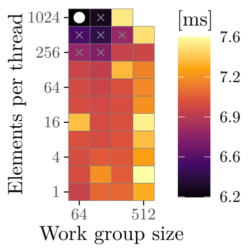


(g) Sequential 64-bit BF U
on Intel Core i7-4900MQ

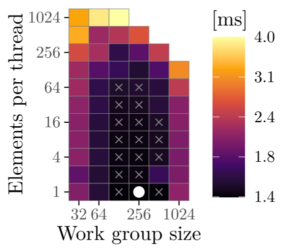
Figure 3.3.: Influence of workload parameters (CPUs). The heatmaps show the performance of the selected basic kernel variant and low-level implementation parameters at selectivity 0.5. ● indicates the fastest variant. ☒ indicates a variant that is at most 10% slower than the fastest. Note the different scales for the work group size on the x axis. (Figure continues on next page.)



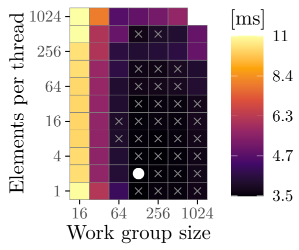
(h) Collect 8-bit
on AMD Radeon HD 6950



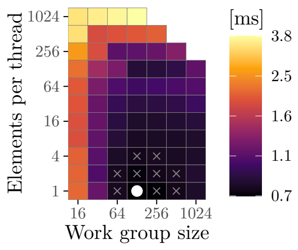
(i) Transpose 64-bit BF U
on Intel Iris 5100



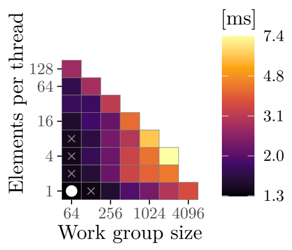
(j) Transpose 32-bit U
on Nvidia GeForce GTX 460



(k) Transpose 16-bit BF
on Nvidia Quadro K2100M



(l) Transpose 16-bit
on Nvidia Tesla K40M



(m) Transpose 64-bit
on Intel Xeon Phi 7120

Figure 3.3.: Influence of workload parameters (continued from previous page, GPUs and Intel Xeon Phi).

### 3.3.8. Number of competitive variants

In Figure 3.2 and Figure 3.3 we indicate competitive variants with the symbol ☒, i.e., those that are at most 10% slower than the fastest. 10% is an arbitrary threshold but it serves to highlight the importance of some implementation parameters over others and how they influence variant runtime.

For example, on Intel CPUs, if the selectivity is neither very low nor very high, it is important to select a branch-free Sequential kernel. The size of the bitmap result type and loop unrolling have a smaller, but not insignificant influence. Furthermore, we can choose a wide range of workload parameters, as long as we utilize all hyperthreads. Interestingly, on the Intel i7-870, the Xeon E5620, and the Core i7-4900MQ, Transpose kernels are also competitive. In these cases, the choice of the bitmap result type has a stronger influence on variant performance than loop unrolling and branch-free execution. The Intel OpenCL compiler is able to vectorize Transpose kernels whereas it does not vectorize Sequential kernels.

On GPUs, most Transpose kernels are competitive, and on the Intel Iris 5100, the Nvidia Quadro K2100M and the Nvidia Tesla K40M, any combination of low-level implementation parameters results in a competitive Transpose variant. Collect is also

Table 3.2.: Distribution of the variant runtime on the tested processors at selectivity 0.5. The columns show the deciles and the maximum of the normalized runtime, i.e., the absolute runtime divided by the minimal runtime for each processor and selectivity. Deciles which are within 10% of the fastest variant are highlighted in **bold**.

| Processor | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | Max |
|---|---|---|---|---|---|---|---|---|---|---|
| AMD Opteron 2356 | 1.26 | 1.30 | 1.40 | 1.62 | 2.1 | 2.6 | 3.7 | 5.7 | 9.1 | 41 |
| AMD Opteron 6128 HE | 1.47 | 1.54 | 1.55 | 1.56 | 1.70 | 2.1 | 3.0 | 4.6 | 7.4 | 33 |
| IBM 8231-E2B | 1.68 | 2.2 | 2.7 | 3.1 | 3.6 | 4.2 | 5.0 | 6.7 | 11 | 51 |
| Intel Core i7-870 | **1.02** | **1.10** | 1.30 | 1.79 | 2.1 | 2.3 | 2.4 | 3.3 | 6.0 | 33 |
| Intel Xeon E5620 | **1.01** | **1.07** | 1.21 | 1.68 | 2.0 | 2.1 | 2.4 | 3.1 | 7.8 | 32 |
| Intel Xeon E5-2650 v2 | 1.15 | 1.40 | 1.60 | 2.6 | 3.1 | 3.3 | 3.9 | 6.0 | 9.2 | 66 |
| Intel Core i7-4900MQ | **1.02** | **1.06** | 1.33 | 1.64 | 2.4 | 3.2 | 3.6 | 5.1 | 6.6 | 23 |
| AMD Radeon HD 6950 | **1.07** | 1.19 | 1.62 | 2.5 | 3.5 | 4.8 | 6.9 | 9.4 | 13 | 83 |
| Intel Iris 5100 | **1.10** | 1.13 | 1.17 | 1.26 | 1.75 | 2.2 | 2.7 | 3.5 | 8.2 | 73 |
| Nvidia GeForce GTX 460 | 1.19 | 1.46 | 1.95 | 3.2 | 4.3 | 7.3 | 9.9 | 16 | 22 | 219 |
| Nvidia Quadro K2100M | **1.07** | 1.20 | 1.55 | 2.3 | 3.6 | 6.3 | 11 | 14 | 20 | 320 |
| Nvidia Tesla K40M | 1.17 | 1.33 | 1.67 | 2.5 | 3.3 | 4.9 | 9.1 | 10 | 14 | 71 |
| Intel Xeon Phi 7120 | 1.58 | 1.97 | 2.5 | 3.3 | 4.4 | 5.8 | 7.2 | 8.9 | 13 | 147 |

a fast kernel on GPUs, but the choice of implementation parameters is more important. Furthermore, the range of fast workload parameters is smaller than on Intel CPUs.

Finally, the Intel Xeon Phi has very few competitive variants. The fastest variant is a 64-bit Transpose kernel. Both low-level implementation parameters and work-load parameters strongly influence variant performance.

Table 3.2 shows a more complete view of the variant runtime distributions on the tested processors. In it, we show the deciles of the distributions as well as the runtime of the slowest variant in the column *Max*. In order to compare the distributions across processors with different performance characteristics, we normalize the variant runtime by dividing it by the runtime of the fastest variant for each processor at selectivity 0.5. (Note that runtime distribution differs from our previously published work [268] because we exclude variants which do not effectively parallelize the workload over the available processing cores, as described in Section 3.3.4.)

The table shows how some processors have many competitive variants whereas others have few. For example, on the Intel Core i7-870, the Xeon E5620, and the Core i7-4900MQ, at least 20% of the variants are competitive, and on the AMD Radeon HD 6950, the Intel Iris 5100, and the Nvidia Quadro K2100M, at least 10% of the variants are competitive. We can conclude that on these processors it is comparatively easy to

Table 3.3.: Implementation parameters of the fastest variants for each selectivity (CPUs). Sel = selectivity, RT = result bitmap bit size, BF = branch-free evaluation, U = unrolled, E/T = elements per thread, WGS = work group size, WG = number of work groups. The most common choice for each parameter is shown in **bold**; ties are shown in *italics*. (Table continues on next page.)

(a) AMD Opteron 2356

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Sequential** | 64 | **BF** | | 256 | 32 | 64 | 20.6 |
| 0.1 | **Sequential** | 64 | **BF** | U | 512 | 64 | *16* | 19.5 |
| 0.2 | **Sequential** | **8** | **BF** | | 128 | 0 | – | 19.5 |
| 0.3 | **Sequential** | **8** | **BF** | | **1024** | **256** | *16* | 19.5 |
| 0.4 | Transpose | 64 | **BF** | U | 128 | 128 | 32 | 21.2 |
| 0.5 | **Sequential** | 64 | **BF** | U | 256 | **256** | *8* | 17.5 |
| 0.6 | **Sequential** | **8** | **BF** | | **1024** | 512 | *8* | 18.1 |
| 0.7 | **Sequential** | **8** | **BF** | U | **1024** | 512 | *8* | 15.6 |
| 0.8 | **Sequential** | **8** | **BF** | U | **1024** | **256** | *16* | 16.9 |
| 0.9 | **Sequential** | **8** | **BF** | | 256 | 1024 | *16* | 15.7 |
| 1 | **Sequential** | **8** | | | 512 | 1024 | *8* | 16.2 |

(b) AMD Opteron 6128 HE

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | *Sequential* | **8** | | | 2 | 128 | **16384** | 14.4 |
| 0.1 | *Sequential* | 64 | | **U** | **1** | 32 | **16384** | 14.5 |
| 0.2 | Transpose | 16 | **BF** | **U** | **1** | 128 | **16384** | 14.7 |
| 0.3 | Transpose | 16 | **BF** | **U** | **1** | 128 | **16384** | 13.8 |
| 0.4 | *Sequential* | 32 | **BF** | **U** | **1** | 64 | **16384** | 12.7 |
| 0.5 | *Collect* | **8** | **BF** | **U** | **1** | 512 | 8192 | 12.2 |
| 0.6 | Transpose | 32 | **BF** | **U** | **1** | 64 | **16384** | 14.0 |
| 0.7 | *Collect* | **8** | **BF** | **U** | **1** | **256** | **16384** | 14.4 |
| 0.8 | *Sequential* | **8** | **BF** | | **1** | **256** | **16384** | 13.6 |
| 0.9 | *Collect* | **8** | **BF** | **U** | **1** | **256** | **16384** | 13.6 |
| 1 | *Collect* | 16 | **BF** | **U** | **1** | **256** | 8192 | 12.2 |

(c) IBM 8231-E2B

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Sequential** | *16* | **BF** | | 4 | 64 | 8192 | 3.2 |
| 0.1 | **Sequential** | *32* | **BF** | | **128** | 4 | *2048* | 3.7 |
| 0.2 | **Sequential** | 8 | **BF** | | 1024 | **16** | *256* | 4.0 |
| 0.3 | **Sequential** | *32* | **BF** | U | **128** | **16** | 512 | 3.9 |
| 0.4 | **Sequential** | *16* | **BF** | | 8 | 512 | 512 | 3.7 |
| 0.5 | **Sequential** | 8 | **BF** | | 64 | 256 | *256* | 3.9 |
| 0.6 | **Sequential** | 8 | **BF** | U | **128** | **16** | *2048* | 3.8 |
| 0.7 | **Sequential** | *32* | **BF** | | 512 | 2 | 1024 | 3.8 |
| 0.8 | **Sequential** | *32* | **BF** | U | 32 | 128 | *256* | 3.5 |
| 0.9 | **Sequential** | *16* | | U | 512 | 2 | *2048* | 3.3 |
| 1 | **Sequential** | *16* | | U | 2 | 1024 | 1024 | 2.7 |

(d) Intel Core i7-870

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Sequential** | **32** | **BF** | | **128** | 2048 | **4** | 8.7 |
| 0.1 | **Sequential** | **32** | **BF** | | 256 | *1024* | **4** | 8.9 |
| 0.2 | **Sequential** | **32** | **BF** | | **128** | 2 | 4096 | 9.0 |
| 0.3 | **Sequential** | **32** | **BF** | | 32 | 8192 | **4** | 8.9 |
| 0.4 | **Sequential** | 64 | **BF** | U | **128** | *1024* | **4** | 8.8 |
| 0.5 | **Sequential** | 64 | **BF** | U | 256 | 1 | 2048 | 8.9 |
| 0.6 | **Sequential** | **32** | **BF** | | 1024 | *256* | **4** | 8.8 |
| 0.7 | **Sequential** | 64 | **BF** | U | 512 | *256* | **4** | 8.8 |
| 0.8 | **Sequential** | **32** | **BF** | | 1 | 0 | – | 8.9 |
| 0.9 | **Sequential** | 64 | | U | 512 | *128* | 8 | 8.9 |
| 1 | **Sequential** | 64 | | U | 1024 | *128* | **4** | 8.5 |

(e) Intel Xeon E5620

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | GlobalAtomic | 32 | | | **8** | 128 | *32768* | 10.7 |
| 0.1 | **Sequential** | **64** | **BF** | **U** | **8** | 8192 | 8 | 10.8 |
| 0.2 | **Sequential** | **64** | **BF** | **U** | 32 | **1** | 16384 | 10.8 |
| 0.3 | **Sequential** | **64** | **BF** | **U** | **8** | 2 | *32768* | 10.8 |
| 0.4 | **Sequential** | 32 | **BF** | | 1 | 256 | *4096* | 10.8 |
| 0.5 | **Sequential** | **64** | **BF** | **U** | 32 | 2 | 8192 | 10.8 |
| 0.6 | **Sequential** | **64** | **BF** | **U** | 4 | 32 | *4096* | 10.8 |
| 0.7 | **Sequential** | **64** | **BF** | **U** | 16 | **1** | *32768* | 10.8 |
| 0.8 | **Sequential** | 32 | **BF** | | 1024 | 256 | 4 | 10.5 |
| 0.9 | **Sequential** | **64** | **BF** | **U** | 128 | **1** | *4096* | 10.7 |
| 1 | **Sequential** | **64** | | **U** | 64 | 2048 | 4 | 10.7 |

(f) Intel Xeon E5-2650 v2

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Sequential** | *8* | | | 128 | *4* | **8192** | 2.3 |
| 0.1 | Transpose | 16 | | | 64 | *16* | 2048 | 2.3 |
| 0.2 | **Sequential** | *64* | **BF** | U | **1** | 64 | **8192** | 2.1 |
| 0.3 | **Sequential** | *8* | **BF** | | **1** | 512 | **8192** | 1.94 |
| 0.4 | **Sequential** | *8* | **BF** | | **1** | 32 | 131072 | 1.80 |
| 0.5 | **Sequential** | *64* | **BF** | U | 2 | 8192 | 32 | 1.82 |
| 0.6 | **Sequential** | *64* | **BF** | U | 32 | *4* | 4096 | 1.93 |
| 0.7 | **Sequential** | *8* | **BF** | | **1** | 256 | 16384 | 1.97 |
| 0.8 | **Sequential** | *64* | **BF** | U | 4 | 8 | 16384 | 2.1 |
| 0.9 | Transpose | 16 | | | 64 | *16* | 2048 | 2.1 |
| 1 | **Sequential** | 32 | | | 256 | 1 | 4096 | 1.78 |

(g) Intel Core i7-4900MQ

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Sequential** | **64** | | **U** | 32 | 2048 | 8 | 6.3 |
| 0.1 | **Sequential** | **64** | **BF** | **U** | **1** | 128 | 4096 | 6.4 |
| 0.2 | **Sequential** | **64** | **BF** | **U** | **1** | 64 | 8192 | 6.4 |
| 0.3 | **Sequential** | **64** | **BF** | **U** | **1** | 64 | 8192 | 6.4 |
| 0.4 | **Sequential** | 8 | **BF** | **U** | 8 | 32 | 16384 | 6.4 |
| 0.5 | **Sequential** | **64** | **BF** | **U** | **1** | **0** | – | 6.4 |
| 0.6 | **Sequential** | 8 | **BF** | | 8 | **0** | – | 6.4 |
| 0.7 | **Sequential** | **64** | **BF** | **U** | **1** | **0** | – | 6.4 |
| 0.8 | **Sequential** | **64** | **BF** | **U** | **1** | 64 | 8192 | 6.4 |
| 0.9 | **Sequential** | **64** | **BF** | **U** | **1** | 32 | 16384 | 6.4 |
| 1 | **Sequential** | **64** | | **U** | **1** | **0** | – | 6.3 |

Table 3.3.: Implementation parameters of the fastest variants for each selectivity (continued from previous page, GPUs and Xeon Phi).

(h) AMD Radeon HD 6950

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Collect** | **8** | | U | **1** | **128** | **32768** | 1.09 |
| 0.1 | **Collect** | **8** | | | **1** | **128** | **32768** | 1.09 |
| 0.2 | **Collect** | **8** | | | **1** | **128** | **32768** | 1.09 |
| 0.3 | **Collect** | **8** | | U | **1** | **128** | **32768** | 1.09 |
| 0.4 | **Collect** | **8** | | | **1** | **128** | **32768** | 1.09 |
| 0.5 | **Collect** | **8** | | | **1** | **128** | **32768** | 1.09 |
| 0.6 | **Collect** | **8** | | U | **1** | **128** | **32768** | 1.09 |
| 0.7 | **Collect** | **8** | | | **1** | **128** | **32768** | 1.09 |
| 0.8 | **Collect** | **8** | | | **1** | **128** | **32768** | 1.09 |
| 0.9 | **Collect** | **8** | | | **1** | **128** | **32768** | 1.09 |
| 1 | **Collect** | **8** | | U | **1** | **128** | **32768** | 1.09 |

(i) Intel Iris 5100

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.3 |
| 0.1 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.2 |
| 0.2 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.2 |
| 0.3 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.2 |
| 0.4 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.2 |
| 0.5 | **Transpose** | **64** | **BF** | **U** | **1024** | 64 | 8 | 6.2 |
| 0.6 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.2 |
| 0.7 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.2 |
| 0.8 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.2 |
| 0.9 | **Transpose** | **64** | **BF** | | **1024** | **128** | **4** | 6.3 |
| 1 | **Transpose** | **64** | **BF** | **U** | **1024** | **128** | **4** | 6.2 |

(j) Nvidia GeForce GTX 460

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.1 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.2 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.3 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.4 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.5 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.6 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.7 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.8 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 0.9 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |
| 1 | **Transpose** | **32** | | **U** | **1** | **256** | **4096** | 1.41 |

(k) Nvidia Quadro K2100M

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Transpose** | **16** | | | **2** | **128** | **8192** | 3.5 |
| 0.1 | **Transpose** | **16** | **BF** | U | 4 | **128** | 4096 | 3.5 |
| 0.2 | **Transpose** | **16** | **BF** | | 1 | **128** | 16384 | 3.5 |
| 0.3 | **Transpose** | **16** | **BF** | U | **2** | **128** | **8192** | 3.5 |
| 0.4 | **Transpose** | **16** | **BF** | | **2** | **128** | **8192** | 3.5 |
| 0.5 | **Transpose** | **16** | **BF** | | **2** | **128** | **8192** | 3.5 |
| 0.6 | **Transpose** | **16** | | | **2** | **128** | **8192** | 3.5 |
| 0.7 | **Transpose** | **16** | **BF** | | **2** | **128** | **8192** | 3.5 |
| 0.8 | **Transpose** | **16** | **BF** | U | **2** | **128** | **8192** | 3.5 |
| 0.9 | **Transpose** | **16** | **BF** | U | 4 | **128** | 4096 | 3.5 |
| 1 | **Transpose** | **16** | **BF** | U | **2** | **128** | **8192** | 3.5 |

(l) Nvidia Tesla K40M

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Transpose** | **16** | | | **1** | **128** | **16384** | 0.72 |
| 0.1 | **Transpose** | **16** | | | **1** | **128** | **16384** | 0.72 |
| 0.2 | **Transpose** | **16** | | U | **1** | **128** | **16384** | 0.72 |
| 0.3 | **Transpose** | **16** | | U | **1** | **128** | **16384** | 0.72 |
| 0.4 | **Transpose** | **16** | | | **1** | **128** | **16384** | 0.72 |
| 0.5 | **Transpose** | **16** | | | **1** | **128** | **16384** | 0.72 |
| 0.6 | **Transpose** | **16** | | | **1** | **128** | **16384** | 0.72 |
| 0.7 | **Transpose** | **16** | | U | **1** | **128** | **16384** | 0.72 |
| 0.8 | **Transpose** | **16** | | | **1** | **128** | **16384** | 0.72 |
| 0.9 | **Transpose** | **16** | | | **1** | **128** | **16384** | 0.72 |
| 1 | **Transpose** | **16** | | U | **1** | **128** | **16384** | 0.72 |

(m) Intel Xeon Phi 7120

| Sel | Variant | RT | BF | U | E/T | WGS | WG | [ms] |
|---|---|---|---|---|---|---|---|---|
| 0 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.32 |
| 0.1 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.32 |
| 0.2 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.33 |
| 0.3 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.34 |
| 0.4 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.32 |
| 0.5 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.32 |
| 0.6 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.34 |
| 0.7 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.32 |
| 0.8 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.32 |
| 0.9 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.30 |
| 1 | **Transpose** | **64** | | | **1** | **64** | **8192** | 1.32 |

select a fast variant, whereas on other processors, e.g., the Intel Xeon Phi, the choice is more difficult. Moreover, the table also shows that on every processors there is a large jump in the variant runtime when comparing the 90% decile to the slowest variant, from factor 3.5× on the Intel Core i7-4900MQ to 16× on the Nvidia Quadro K2100M. This large factor indicates that there are a few variants which are really slow and which we should avoid at all costs.

Nevertheless, on every processors, we can find certain implementation parameter choices that consistently outperform other choices. In Table 3.3, we show the implementation parameters of the fastest variant at every selectivity value. The results on GPUs and on the Intel Xeon Phi are particularly stable. On the Nvidia GeForce GTX 460 and the Intel Xeon Phi, a single combination of implementation parameters is the fastest variant across the range of tested selectivities. On the AMD Radeon HD 9650 and the Nvidia Tesla K40M, there is a slight variation with regard to the loop unrolling parameter. Furthermore, on the Intel Iris 5100 and the Nvidia Quadro K2100M, there is a slight variation with regard to the number of elements per threads and the work group size, however the range of selected workload parameters is small. Because of this stability across the entire selectivity range, we believe that these parameter configurations constitute a performance sweet spot.

### 3.3.9. Summary

Even for a simple selection kernel, and a small number of implementation parameters, we can generate thousands of operator variants. No single variant performs well on every processor. In fact, the fastest variant depends on the processor type, e.g., CPU or GPU, the manufacturer, and the concrete processor microarchitecture. The processors also differ with regard to the number of competitive variants, i.e., those which are at most 10% slower than the fastest. On some processors, e.g. the Intel Core i7-4900MQ or the Nvidia Quadro K2100M, many variants are competitive. On other processors, e.g., the Intel Xeon Phi, only a few specific combinations of implementation parameters result in a fast variant.

## 3.4. Performance analysis of hash aggregation on GPUs

In this section, we continue our investigation of the operator variant selection problem on heterogeneous hardware. However, in contrast to the previous section, in which we evaluated a diverse range of processors such as CPUs and GPUs, we now focus on a single processor type, i.e., GPUs, and examine the influence of the GPU microarchitecture on variant performance across different GPU vendors and models.

We conduct our analysis using a hash aggregation operator as an example of a stateful data processing primitive which can be effectively accelerated on GPUs [149, 201]. Hash aggregation is commonly used to implement the final aggregation in OLAP queries, to group the results of subqueries, or to eliminate duplicates. The performance of parallelized hash aggregation is mainly determined by the efficient use of processor caches [206] and by the amount of contention caused when multiple threads access a single hash table [58]. Both factors are directly related to the number of groups. Consequently, multiple *parallelization strategies* have been proposed that maximize cache efficiency and minimize the effects of contention depending on the group cardinality [58, 149, 206, 342]. Furthermore, as we discuss in Section 2.3.4, the performance of GPUs kernels is strongly influenced by their *thread configuration*, i.e., the work group size and the number of work groups per compute unit (CU). We refer to the parallelization strategy and the thread configuration as the *execution parameters* of the hash aggregation operator.

We first describe the implementation of a GPU-based hash aggregation operator and three parallelization strategies. Afterwards, we evaluate the performance of these parallelization strategies and the influence of different thread configurations on six GPUs based on different microarchitectures. Specifically, we look at four Nvidia GPUs based on the Kepler, Maxwell, Pascal, and Volta microarchitectures, as well as two AMD GPUs based on the 2nd and 3rd generation Graphics Core Next (GCN) microarchitectures. Our main finding is that the optimal execution parameters strongly depend on the executing GPU and that heuristics derived from the study of a single GPU cannot be generalized to other GPUs. Furthermore, we find that for a given parallelization strategy, the thread configuration search space has a single local minimum if we account for runtime variation.

### 3.4.1. GPU-accelerated hash aggregation

Our operator implementation is based on the scheme described by Karnagel et al. [149] with a few modifications to adapt it to GPUs by different manufacturers. We use the following SQL query as an example to describe the implementation in detail:

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

The query contains arithmetic operations both inside an aggregation function, i.e., `sum(`$(a-b)^2$`)`, as well as outside of the aggregation, i.e., it divides the result of `sum` by `count`.

#### 3.4.1.1. Preliminaries

We assume that the group cardinality $|g|$ is known so that we can size a hash table in advance and do not have to resize it during aggregation. Note that group cardinalities for arbitrary column combinations can be estimated with high accuracy and low overhead [84]. Furthermore, we assume that the hash table fits into GPU device memory. Current dedicated GPUs support up to 80 GB of device memory [218] which allows for very large group cardinalities. In contrast, the input table `R` is stored in main memory and does not necessarily fit into the device memory of a dedicated GPU. Integrated GPUs can access the input table directly in main memory, but dedicated GPUs require a data transfer of the input over a system bus, such as PCIe or NVLink.

We use multiply/shift [160] as the hash function and linear probing as the hashing scheme. These parameters achieve the highest throughput in an aggregation scenario, which consists only of insertions and successful lookups, if the load factor is below 90% [262]. Given the group cardinality $|g|$, and a desired load factor, an open addressing scheme such as linear probing also allows us to determine the exact size of the hash table at the beginning of the aggregation.

#### 3.4.1.2. Operator execution stages

The hash aggregation operator consists of multiple stages which are centered around three GPU kernels. In order to run the same code on AMD and Nvidia GPUs, we implement our kernels in OpenCL [300].

The operator first allocates sufficient memory on the GPU for the hash table and calls the INITIALIZE kernel. This kernel marks every hash bucket as empty and stores an initial value for each aggregation function, e.g., zero for `sum` and `count`.

Next, the operator processes the input in a block-wise fashion and calls the AGGREGATE kernel for each block. This kernel determines the hash bucket, performs computations inside aggregation functions, i.e., $(a - b)^2$ in our example, and updates all aggregates. It also tracks the number of non-empty hash buckets. We implement a software-managed staged pipeline (see Section 2.7.2.1) to explicitly overlap computation and data transfer, instead of relying on hardware-managed zero-copy access as in previous work [149]. This approach allows us to measure the raw execution speed of the AGGREGATE kernel.

Once the input has been processed, the operator allocates sufficient memory to store the final result based on the number of non-empty hash buckets determined by the AGGREGATE kernel. It then calls the FINALIZE kernel which iterates over the hash table, performs the computations outside of the aggregation functions, i.e., dividing `sum` by `count`, and materializes the result.

### 3.4.2. Parallelization strategies

The AGGREGATE kernel implements one of three parallelization strategies, which are illustrated schematically in Figure 3.4. These strategies have been shown to achieve high throughput on GPUs [149]. Two strategies, i.e., SHARED and INDEPENDENT, are also commonly used on multi-core CPUs [58, 342]. WORKGROUPLOCAL is specifically optimized to use fast local memory found on GPUs [149].

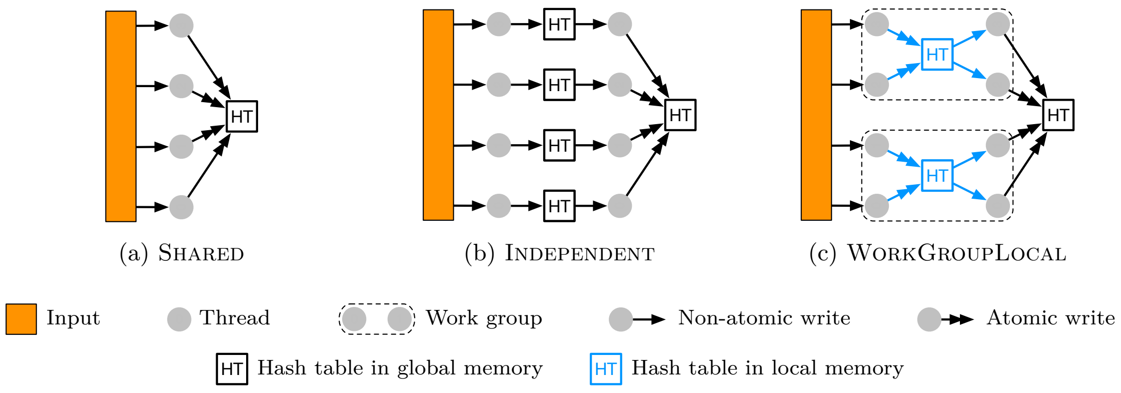


Figure 3.4.: Parallelization strategies.

In the Shared strategy, shown in Figure 3.4a, every thread aggregates into a single, shared hash table, which is placed in global GPU memory. Concurrent updates to the same hash bucket are resolved with atomic access primitives. For large group cardinalities and a uniform distribution of group values, contention is negligible because the chance of two threads accessing the same hash table bucket is small.

In the Independent strategy, shown in Figure 3.4b, each thread first aggregates into a thread-private hash table, which is also placed in global memory. Because each thread accesses its private hash table exclusively, there is no contention and no need to use atomics. Once an input block has been processed, the private tables are merged into a global table. Even though Independent still utilizes atomics to access the global hash table, the number of accesses is reduced by the ratio of the number of groups in the hash table and the number of tuples in the input block. However, because a GPU executes many threads, this strategy creates many hash table duplicates which have to fit into the L2 cache to minimize memory access latency. Therefore, this strategy is only feasible for very small group cardinalities.

In the WorkGroupLocal strategy, shown in Figure 3.4c, the threads of a work group cooperatively aggregate into a hash table that is placed in fast local memory. Concurrent accesses to this private hash table are resolved using atomic access primitives. Compared to Shared, contention is reduced because the accesses are distributed to multiple work group tables instead of a single global table. The number of work group tables, and the number of threads which try to access it concurrently, is determined by the thread configuration, i.e., the number of work groups per compute unit and work group size. Once a block has been fully processed, the intermediate result is merged into a table stored in global GPU memory. Note that the local memory region is relatively small, typically between 32 and 96 kB. Therefore, we can use this strategy only for small to medium group cardinalities.

### 3.4.3. Performance analysis on GPUs

In the following, we examine how hardware differences influence the performance of hash aggregation on GPUs. To this end, we perform five experiments on six different GPUs from AMD and Nvidia. (1) We evaluate the influence of the parallelization strategy and (2) the thread configuration on the performance of the Aggregate kernel. (3) We evaluate the performance penalty when executing an Aggregate kernel optimized for a specific GPU on other GPUs. (4) We analyze the shape of the thread configuration search spaces, i.e., we test if they have a single local minimum. (5) We analyze the degree of runtime variation and the influence of outliers on different GPUs.

In our evaluation, we focus on the effect of contention and cache efficiency on hash aggregation performance. Therefore, we use the following query with a single aggregate and no additional computation:

```
SELECT g, sum(v) FROM R GROUP BY g;
```

We vary the group cardinality $|g|$ by powers of two between 1 and $2^{28}$. The other evaluation parameters are as follows.

**Execution parameters.** For each group cardinality, we execute the three parallelization strategies described in Section 3.4.2. We vary the number of work groups per compute unit in powers of two, from 1 to 1024. Similarly, we vary the work group size in powers of two, from 1 to the maximum work group size, i.e., 256 on AMD GPUs and 1024 on Nvidia GPUs. In total, we evaluate up to 363 different combinations for each group cardinality. Depending on the group cardinality, some combinations are not possible because they exceed resource limitations.

**GPUs.** We run our experiments on the AMD A10-7850K (based on the 2nd generation GCN microarchitecture), the Radeon R9 Fury (GCN 3rd Gen.), the Nvidia Tesla K40m (Kepler), the GeForce GTX 980 (Maxwell), the GeForce GTX 1080 (Pascal), and the Tesla V100 (Volta). The A10-7850K is integrated with the host CPU. The Tesla V100 is connected over NVLink 2.0 and the others over PCIe 3.0. We list the memory configuration and additional properties of these GPUs in Table 3.4.

**Input data.** The input consists of two 32-bit integer values in columnar format. Each column is split into blocks of 16 MB. We process 32 blocks, so that the total input size is 1 GB. However, our analysis is fundamentally independent of the input

Table 3.4.: GPU properties.

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

size because we execute the Aggregate kernel on individual blocks and overlap kernel execution with data transfer. The group values are randomly generated from a uniform distribution.

**Measurement.** We measure the time to process a block with the Aggregate kernel using OpenCL profiling. We treat the input of 1 GB as a single sample consisting of 32 observations and compute the mean runtime per block. Some GPUs exhibit a high degree of runtime variation. Therefore, to verify our measurements, we collect three samples consisting of 32 observations each. Unless otherwise stated, we report the results of the first sample, which indicates that there are no differences between the samples. We only measure the Aggregate kernel because the Initialize and Finalize kernels are fixed costs that only depend on the hash table size and not on the input size.

### 3.4.4. Influence of the parallelization strategy

In the first experiment, we evaluate how the group cardinality influences the performance of the parallelization strategies on different GPUs. Figure 3.5 shows the throughput of the fastest thread configuration for each of the three parallelization strategies. The subplots have different scales on the $y$ axis because want to emphasize the relative differences for each individual GPU (absolute differences between GPUs are more than an order of magnitude). We report the number of processed input tuples per second on the left $y$ axis of each subplot and the derived throughput in GB/s on the right.

As long as the hash table fits into local GPU memory, WorkGroupLocal is the fastest parallelization strategy. The only exception is the Tesla K40m, where Independent is faster than WorkGroupLocal for small group cardinalities. This behavior is consistent with results reported by Karnagel et al. [149] who also evaluated a Kepler GPU. On this microarchitecture, atomic operations on local memory are implemented using a lock/update/unlock pattern that is slow when contention is high [229, §1.4.3.3].

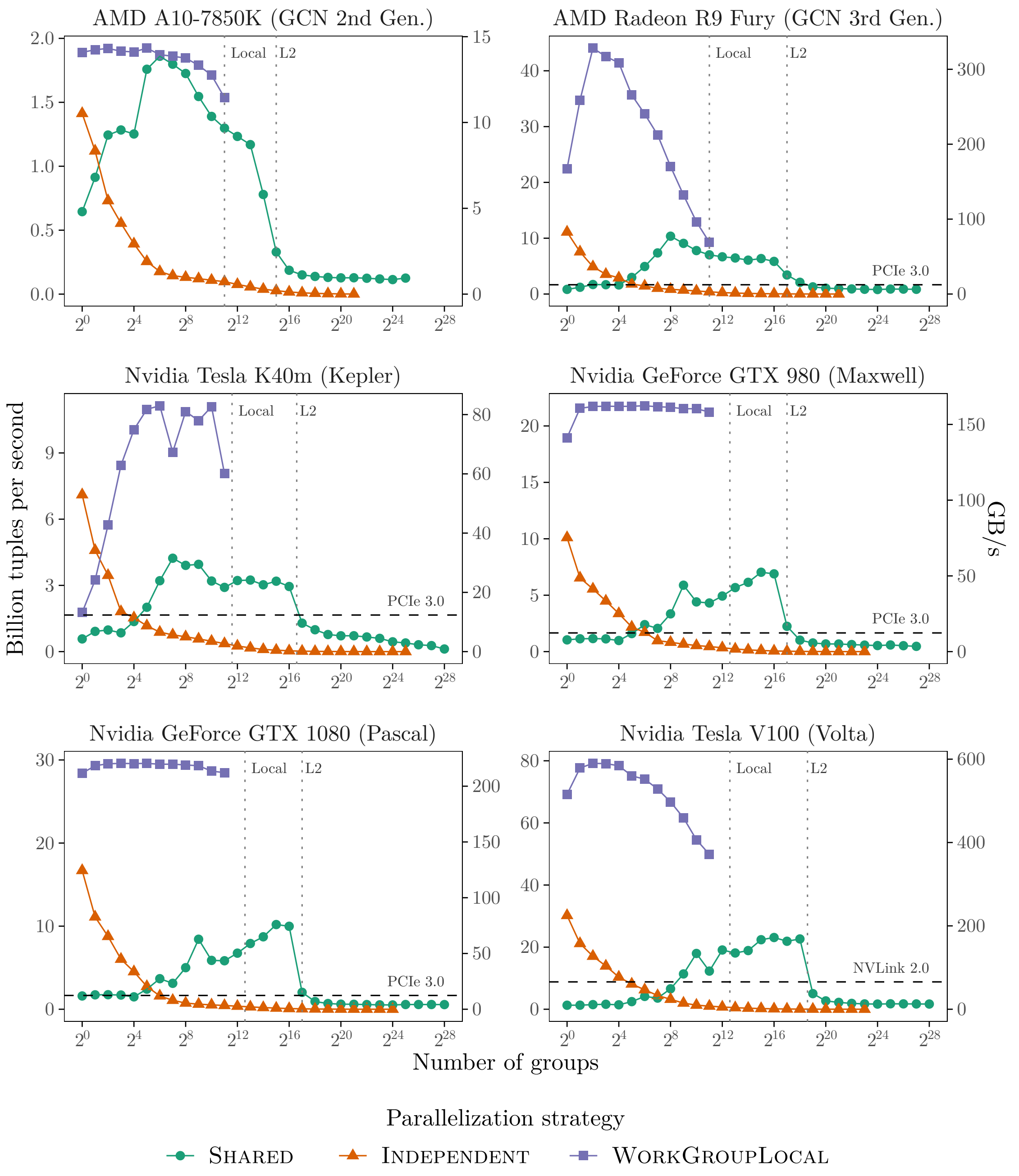


Figure 3.5.: Throughput of parallelization strategies depending on group cardinality. The vertical lines indicate the maximum number of groups so that the hash table fits in local memory when using WORKGROUPLOCAL aggregation, or in the L2 cache when using SHARED aggregation. The horizontal lines indicate the throughput of the system bus. Note the different scales on the $y$ axis.

Starting with the Maxwell microarchitecture, atomics on local memory are implemented with native instructions. Consequently, WorkGroupLocal is at least 1.3× faster than Independent on other GPUs.

When the hash table does not fit into local GPU memory, Shared is the fastest parallelization strategy. There is a steep drop in performance once the size of the hash table exceeds the L2 cache of the GPU. This behavior is consistent with reported results on CPUs [206].

The plots in Figure 3.5 show the raw performance of the Aggregate kernel without data transfers. The A10-7850K can access main memory directly, i.e., the plot shows the actual throughput of the hash aggregation operator. On dedicated GPUs, performance is limited by the data transfer bandwidth, indicated by the dashed lines in Figure 3.5, as long as the hash table fits into the L2 cache. However, for larger hash tables, the raw performance of the Aggregate kernel drops below the data transfer rate. For these hash tables, performance is limited by the global GPU memory latency.

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

### 3.4.5. Influence of the thread configuration

Having determined the fastest parallelization strategy for each group cardinality, we now evaluate which thread configurations yield the best performance on different GPUs. For our analysis, we multiply the number of work groups per compute unit and the work group size of the fastest thread configuration to determine the *optimal number of threads per compute unit*. The scatter plots in Figure 3.6 show the optimal number of threads of each parallelization strategy depending on the group cardinality, i.e., the number of threads that yields the fastest performance of the Aggregate kernel. We plot all three measured samples which is why in some plots there are multiple values per group cardinality and parallelization strategy. These multiple optimal thread configurations are an indication that the runtime of the Aggregate kernel has a high variation on some GPUs. We discuss the effects of this variation in Section 3.4.7 and analyze it in detail in Section 3.4.8.

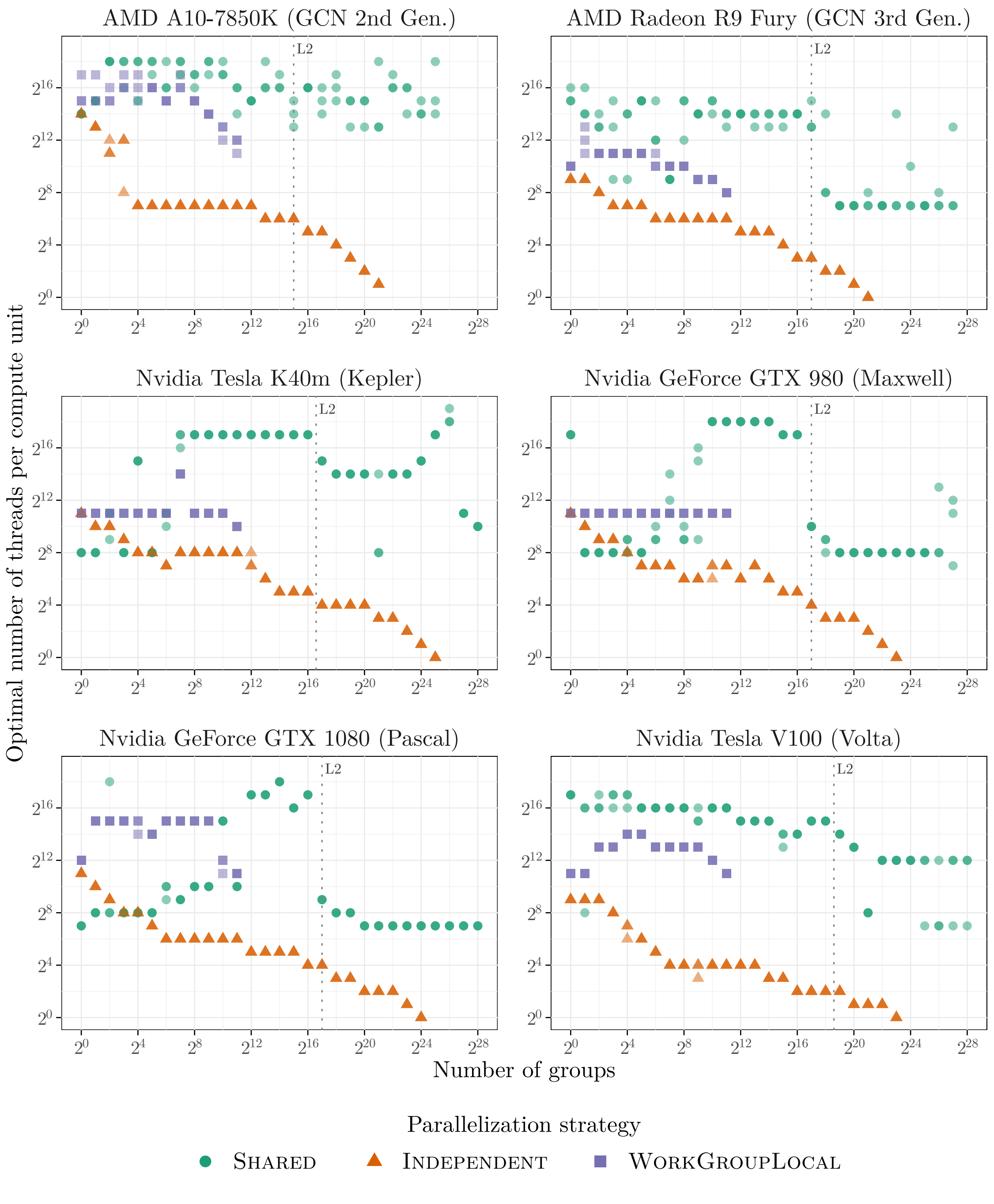


Figure 3.6.: Optimal number of threads depending on group cardinality. Darker shades indicate that more samples in the experiment selected a particular thread configuration. The vertical lines indicate the maximum number of groups so that the hash table fits in the L2 cache when using SHARED aggregation.

Every GPU exhibits a distinct profile in Figure 3.6 but we can identify three common patterns. (1) Independent aggregation shows a downward trend on every GPU. For this strategy, each thread requires a private copy of the hash table, straining GPU memory resources as the group cardinality increases. (2) For WorkGroupLocal aggregation, the optimal number of threads are clustered around GPU-specific values. The GeForce GTX 980 exhibits the least variation with 2048 threads over the entire range of groups. On the Tesla K40m, the fastest configurations also consist of 2048 threads but there are two outliers. The GeForce GTX 1080 and the Tesla V100 exhibit an inverted bowl-shaped pattern clustered around 32768 and 8192 threads, respectively. Finally, the two AMD GPUs show a downward trend clustered around 65536 and 2048 threads. (3) Shared aggregation exhibits the most variation. A common pattern is a change at the boundary of the L2 cache. This pattern is most pronounced on the GeForce GTX 980, the GeForce GTX 1080, and on the Radeon R9 Fury.

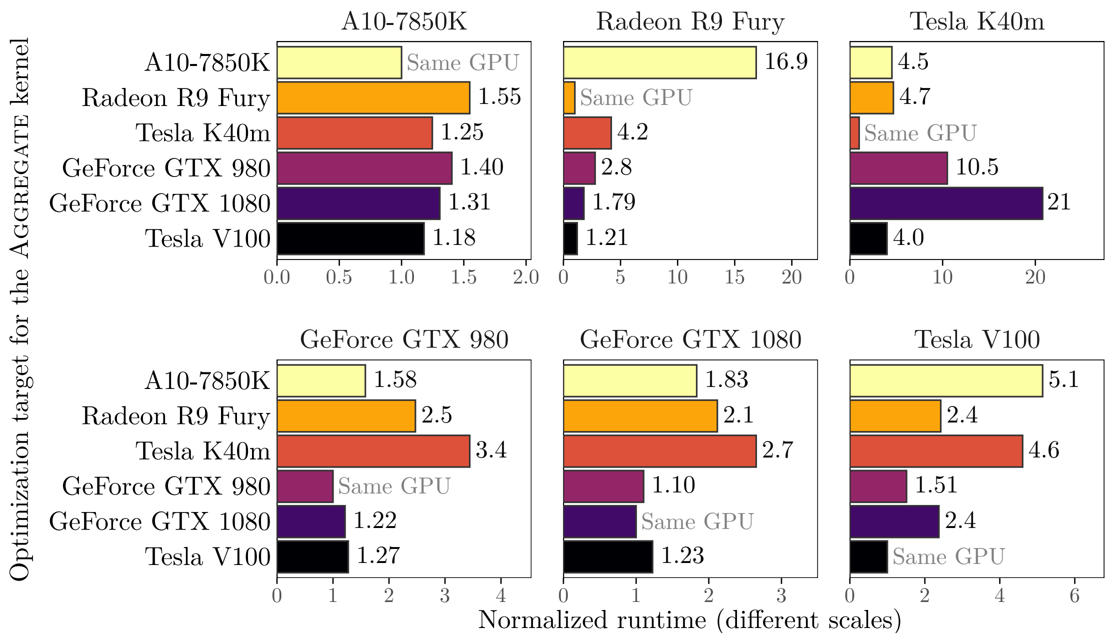
Note that different thread configurations can yield the same number of threads. For example, on the GeForce GTX 980, the fastest thread configurations consist of 2048 threads but the actual configurations vary between 2×1024, 4×512, and 32×64 threads, i.e., 32 work groups per compute unit and 64 work items per work group.

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

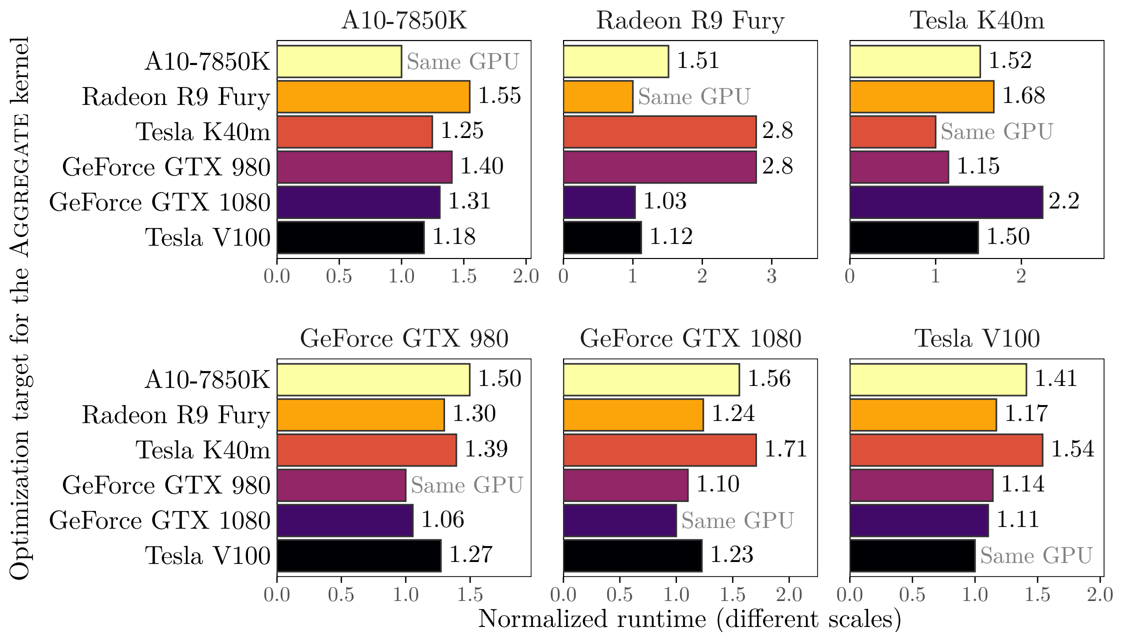
### 3.4.6. Performance penalty of incorrectly optimized Aggregate kernels

In this experiment, we demonstrate the importance of optimizing the execution parameters, i.e., the parallelization strategy and the thread configuration, for every individual GPU. For each GPU and group cardinality, we determine the performance penalty when executing the Aggregate kernel with execution parameters optimized for one of the other five GPUs. Figure 3.7 shows the maximum performance penalty, over all group cardinalities, for each GPU. (In Appendix B, we list the performance penalties for each group cardinality individually.) The subplots represent the GPU on which we execute the Aggregate kernel. The bars in each subplot represent the GPU for which we optimized the execution parameters. To compare the runtimes across group cardinalities, we normalize them relative to the fastest execution parameters for each GPU.

Figure 3.7a shows the maximum performance penalty when the input data is already placed in GPU memory. The performance penalty is particularly large, between 4.0× and 21×, on the Nvidia Tesla K40m. On this GPU, Independent aggregation is the



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



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

Figure 3.7.: Maximum performance penalty of AGGREGATE kernels, which are optimized for specific GPUs (bars), when executed on other GPUs (boxes).

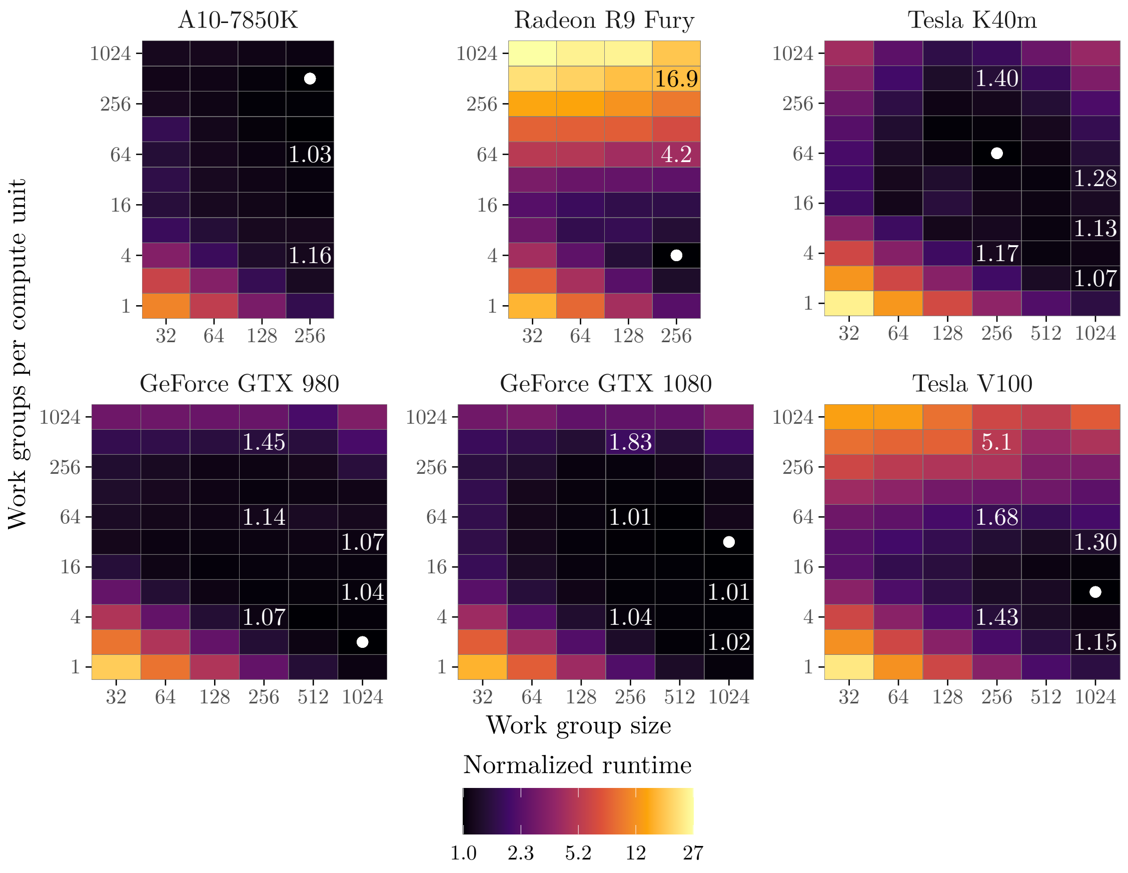


Figure 3.8.: Influence of the thread configuration on the performance of WorkGroupLocal aggregation at 128 groups. ● indicates the fastest thread configuration for a GPU. Labeled tiles indicate the normalized runtime of a thread configuration optimized for another GPU.

fastest parallelization strategy for small group cardinalities, whereas other GPUs use WorkGroupLocal aggregation. The AMD Radeon R9 Fury and the Nvidia Tesla V100 also exhibit large performance penalties of 16.9× and 5.1×, respectively, when executing an Aggregate kernel optimized for the AMD A10-7850K. On both GPUs, the slowdown is caused by a non-optimal thread configuration.

The effect of a non-optimized thread configuration on the performance of the Aggregate kernel is illustrated more clearly in Figure 3.8, which shows a heat map of the performance of WorkGroupLocal aggregation of 128 groups. For each GPU, the fastest thread configuration is indicated by the symbol ●, and the labeled tiles indicate the runtime of a tread configuration optimized for another GPU. On the AMD A10-7850K, the AMD Radeon R9 Fury, and the Nvidia Tesla K40m, i.e., the top row of

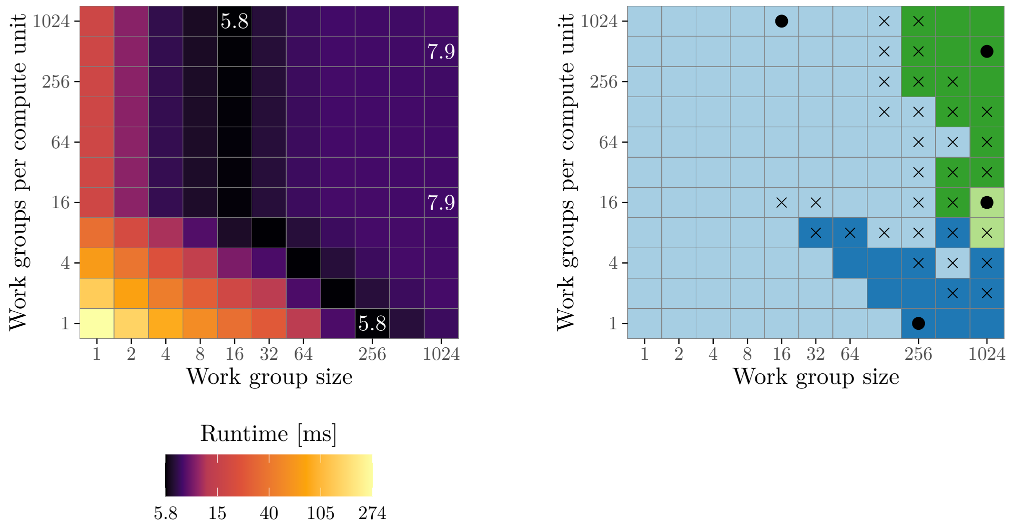


Figure 3.9.: Local minima of the thread configuration search space (left) and performance plateaus across influence regions of local minima (right) of SHARED aggregation on the Tesla K40m at $2^{21}$ groups. The symbol ● indicates a local minimum and the symbol × denotes a performance plateau at the border of two influence regions.

subplots, a work group size of 256 threads is the fastest. However, the optimal number of work groups per compute unit differs, and the wrong thread configuration results in a significant slowdown, particularly on the Radeon R9 Fury. A similar behavior can be observed on the Nvidia GeForce GTX 980 and 1080, and the Nvidia Tesla V100, for which a work group size of 1024 threads is the fastest.

Even when we account for the data transfer the performance penalty is up to 2.8×, as shown in Figure 3.7b. Note that the plots for the AMD A10-7850K are the same in both subfigures because this GPU is integrated with the processor. On other GPUs, Figure 3.7b shows the influence of a non-optimal thread configuration on SHARED aggregation for large group cardinalities (cf. Figure 3.5 on page 94).

To summarize, AGGREGATE kernels optimized for a specific GPU exhibit significant performance penalties when they are executed on another GPU.

### 3.4.7. Plateaus and minima in the thread configuration search space

In this experiment, we evaluate the properties of thread configuration search spaces when we fix the group cardinality and the parallelization strategy. As an example, we show in the left plot of Figure 3.9 the performance of different thread configurations for SHARED aggregation with $2^{21}$ groups on the Nvidia Tesla K40m. In contrast to the previous heat maps, we shift the gradient to increase the resolution at the low end of the scale and to emphasize the differences between fast runtimes.

The runtime behavior of the Tesla K40m is similar to the Nvidia GeForce GTX Titan, which is also based on the Kepler microarchitecture [228], and which was previously analyzed by Karnagel et al. [149]. The heat map appears convex at first glance, but there are multiple local minima, as indicated by the labeled tiles. Two of these local minima, at 1×256 and 1024×16 threads, are selected as the global minimum in different samples, as indicated by Figure 3.6 on page 95. Every local minimum is surrounded by a region of influence, i.e., the points in the thread configuration search space from which the gradient flows to the local minimum, which we illustrate in the left plot of Figure 3.9.

However, the search space also contains *performance plateaus*, i.e., regions where we cannot reliably determine which thread configuration is the fastest. We define that two thread configurations are part of a performance plateau when one of their runtimes is contained within an interval around the other. The extent of performance plateaus depend on the size of the interval we allow around each runtime. In our analysis, we set this interval to either a single standard deviation or 10% of the absolute value, whichever is greater, in each direction.

In particular, there are performance plateaus which span the borders of any two influence regions, which indicate by the symbol × in the right plot of Figure 3.9. This observation indicates that the existence of multiple local minima of the thread configuration search spaces is largely an artifact of the variation of the measured runtime of the aggregation kernel. If we account for this runtime variation, we find that only 2% out of 1143 tested search spaces have more than one local minimum.

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

### 3.4.8. Aggregate kernel runtime variation

As we mentioned in the previous two sections, the runtime of the Aggregate kernel exhibits a high degree of variation on some GPUs. It is necessary to take this variation into account when comparing the performance of different thread configurations, as we did in Section 3.4.7. Thus, in this experiment, we analyze the degree of variation of the Aggregate kernel runtimes on different GPUs. In the following, we first quantify the degree of variation for each GPU and then analyze the influence of outliers.

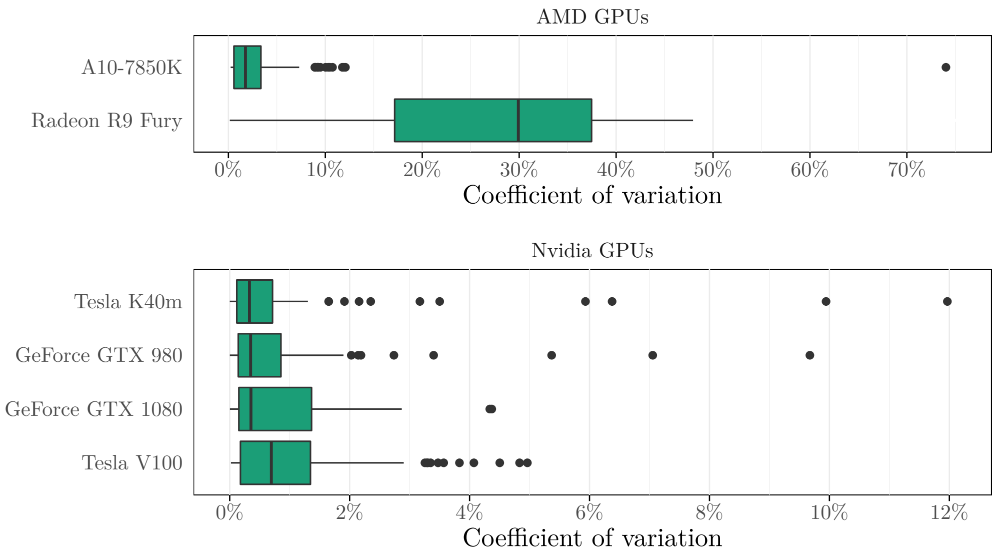


Figure 3.10.: Degree of variation of Aggregate kernel runtimes (different scales on the $x$ axis for AMD and Nvidia GPUs).

**Degree of variation.** To quantify the degree of variation, we use the *coefficient of variation*, i.e., the ratio of the standard deviation and the mean, of each sample. This relative metric captures the fact that the degree of variation must be understood in the context of the measured data. The same standard deviation may indicate a low degree of variation for slow kernels and a high degree of variation for fast kernels. In Figure 3.10, we summarize the coefficient of variation for every Aggregate kernel. Note that we use a different scale on the $x$ axis for AMD and Nvidia GPUs. For clarity, we only show the variation of the fastest thread configuration for each parallelization strategy and group cardinality. This filtering biases the plot towards fast Aggregate kernels but it resembles the plot which includes all thread configurations, except for outliers. However, it is more important to consider the degree of variation when comparing fast Aggregate kernels, as we are interested in finding these.

Our main observation is that Nvidia GPUs exhibit a low degree of runtime variation, with a median coefficient of variation below 0.7%. In contrast, the AMD Radeon R9 Fury exhibits a substantial degree of runtime variation. On AMD GPUs, the outliers of the measured kernel runtimes often exhibit a higher magnitude than the outliers we encounter on Nvidia GPUs. In addition, the samples collected on the Radeon R9 Fury have significantly more outliers than those collected on other GPUs.

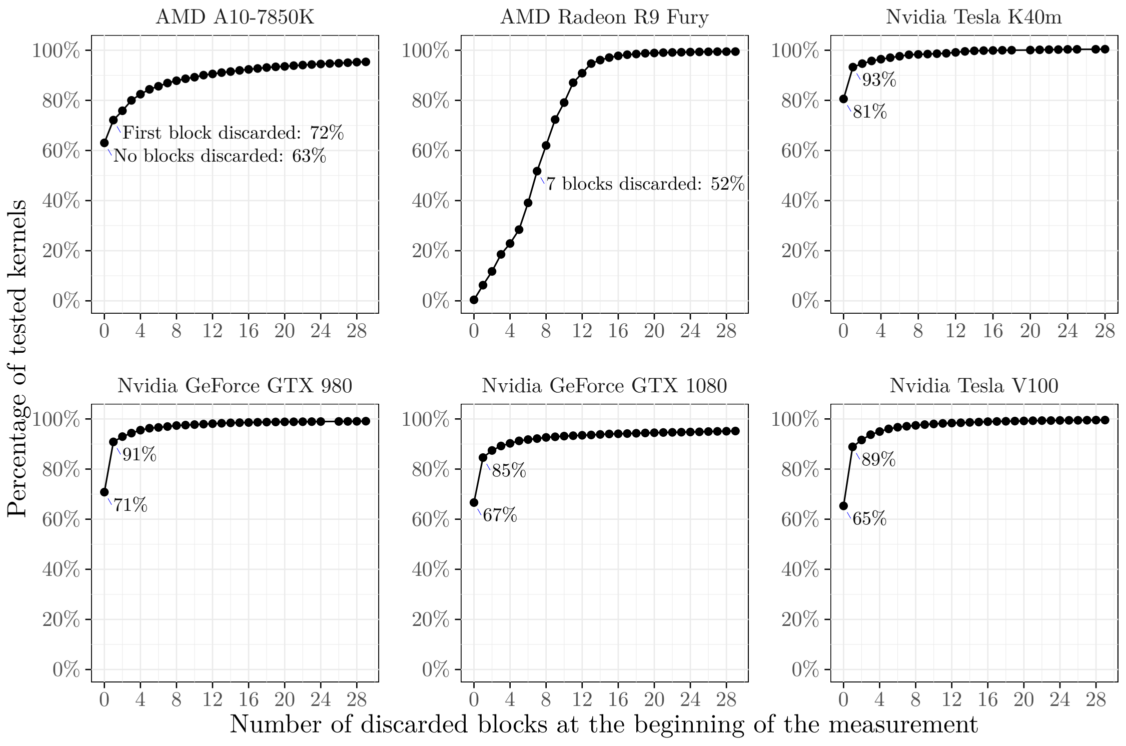


Figure 3.11.: Influence of outliers at the beginning of the measurement on the degree of variation of Aggregate kernels. The points show the percentage of kernels for which the average of the first window of three blocks, after removing outlier blocks in the beginning of the measurement, is at most 1.05% slower than any window of three blocks of the same kernel.

**Influence of outliers.** The outliers of the measured execution runtimes are not uniformly distributed. Instead, they are typically clustered at the beginning of each sample. This behavior is demonstrated in Figure 3.11. The figure shows the percentage of tested Aggregate kernels for which the coefficient of variation, when it is computed over a sliding window of three blocks, drops below a threshold value when we discard outliers in the beginning of the measurement. For this analysis, we choose 1.05% as the threshold. This value corresponds to the 99th percentile of the minimal coefficient of variation computed over any consecutive window of three blocks for every tested Aggregate kernel.

We observe three key points. (1) For a majority of kernels on the AMD A10-7850K, as well as on the Nvidia GPUs, there are no significant outliers in the beginning of our measurements. Concretely, after processing the first window of three blocks, the coefficient of variation is smaller than the threshold for at least 63% of the Aggregate kernels. (2) If necessary, discarding just a single outlier substantially reduces the degree of variation on Nvidia GPUs, as indicated by the steeply rising curves. (3) On AMD GPUs, the influence of outliers is more pronounced, as indicated by the slowly rising curves. Especially on the Radeon R9 Fury, even after we discard the first seven blocks as outliers, the coefficient of variation is smaller than the threshold for only 52% of the Aggregate kernels.

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

### 3.4.9. Summary

We derive five key insights from our analysis. First, Independent aggregation is not competitive on newer Nvidia GPUs which implement fast atomics on local memory. Instead, WorkGroupLocal should be used whenever the hash table fits into local memory. Second, the fastest thread configuration is highly GPU-specific. A thread configuration optimized for a specific GPU is up to 16.9× slower on other GPUs when input data is already placed in GPU memory, and up to 2.8× slower when the input has to be transferred to the GPU. Taken together, these two findings show that *previously formulated heuristics, which are derived from the study of a specific Nvidia Kepler GPU [149], are not generalizable to other GPUs.*

Third, our analysis shows that the thread configuration search space for a specific parallelization strategy and group cardinality is *nearly convex*, i.e., it has a single local minimum when we account for runtime variation. Forth, NVIDIA GPUs exhibit a low degree of runtime variation whereas AMD GPUs exhibit a higher degree of variation.

Finally, we show that the performance of the Aggregate kernel is bounded by global GPU memory latency and not by the data transfer, when the hash table exceeds the L2 cache of the GPU.

## 3.5. Tuning operator variants for a specific processor

As we have seen in the previous two sections, we can construct hundreds or even thousands of variants for simple database operators by modifying a few implementation or workload parameters. The fastest variant is highly dependent on the specific processor on which it is executed, and a non-adapted variant can be an order of magnitude

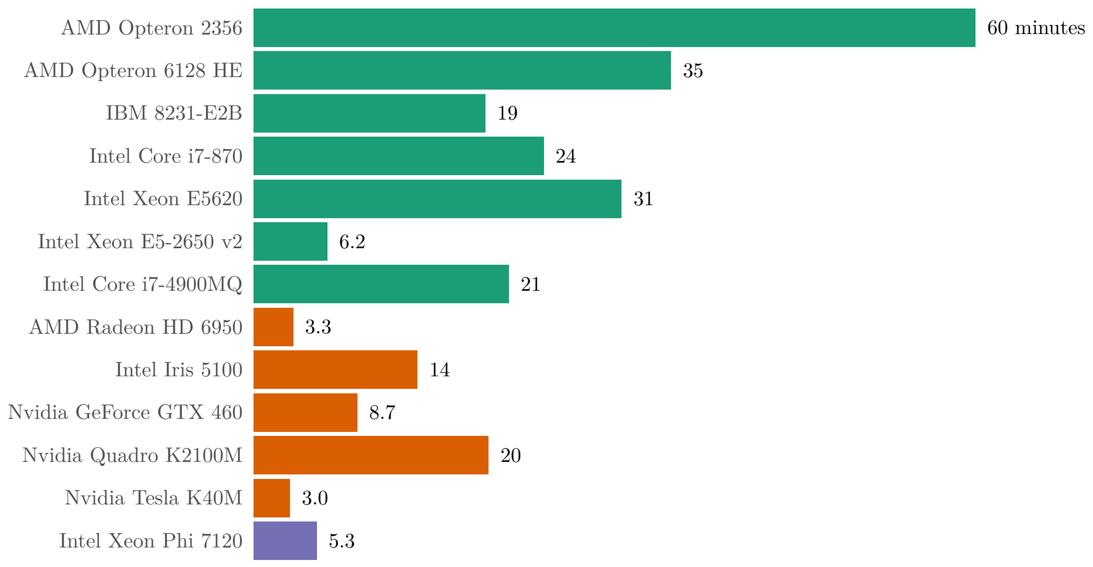


Figure 3.12.: Evaluation time in minutes of the selection kernel variants listed in Table 3.1 on page 78.

slower than the fastest. Given these results, how can we select the fastest, or at least a competitively fast, variant for a given processor?

In the next two sections, we describe two algorithms to solve this problem. Both algorithms use performance feedback gathered during query execution to learn a fast variant at runtime. They differ in the type of information they rely on, in addition to performance feedback, to traverse the variant search space. In the following, we motivate our choice to use an online learning method and discuss the potential benefit of using only runtime performance feedback to guide the variant search.

### 3.5.1. Offline vs. online search

A straightforward approach is to run a training phase during database setup in which we evaluate all possible variants to identify the fastest one. For example, linear algebra libraries such as ATLAS [336], evaluate a large number of auto-generated variants of basic linear algebra subprograms (BLAS) during installation, to find an optimal implementation for a particular environment. However, this offline learning approach has a number of important drawbacks.

First, an offline exploration of database operator variants is very time-consuming. The exploration of just one operator and a few implementation and workload parameters requires a significant amount of time. For example, in Figure 3.12, we show the time required for our exhaustive evaluation of a few thousand selection variants in Section 3.3; on some CPUs, this process requires more than half an hour.

Second, assuming we can search the full variant space reasonably fast, we still face potential data and workload dependencies [258]. Therefore, the initial exploration should be based on a representative query workload, which is often hard to facilitate. Even for a single query, there might not exist a single optimal operator variant, and it is beneficial to switch the operator implementation while executing the query [258]. Furthermore, a representative workload also does not account for potential interference from other workloads on separate processing cores [19, 261, 355].

Third, in cloud-based database-as-a-service applications, the hardware running the database might change at any given moment because of machine migrations. Therefore, we require a flexible strategy that allows us to quickly adapt the selected variant to the new environment.

Given these limitations of offline strategies, we decided to investigate online methods, which rely on performance feedback generated during normal query execution to select operator variants. These methods allow us to start making progress on actual queries immediately, without waiting for an initial training phase, and to react to data and workload changes. The drawback of online methods are varying execution times of similar queries, especially in the beginning, when there is little information about variant performance.

### 3.5.2. Incorporating information about processor characteristics

One advantage of a learning method that relies only on performance feedback gathered during query execution is that it makes no further assumption about the processor. Consequently, it can be applied to very diverse processors, including processors about which we know very little. For example, our evaluation in Section 3.3 includes familiar CPUs and GPUs, but also the Intel Xeon Phi, which is difficult to characterize. On the one hand, it is highly parallel like a GPU; on the other hand, its architecture includes a branch predictor, which is commonly used in CPUs [293]. Indeed, in our evaluation it behaves more like a GPU: The fastest variant is a kernel in which multiple threads cooperate to generate the final result.

Conversely, a large body of prior work has characterized various processor types and determined best practices [6, 132, 213]. Exploiting this information in addition to performance feedback lets us immediately discard a large part of the search space and avoid evaluating potentially slow variants.

### 3.5.3. Outlook

In the next two sections, we describe two algorithms to learn fast operator variants at runtime. The first algorithm is based on micro adaptivity [258], but extends it to handle a large number of candidate variants with unknown performance characteristics. This algorithm relies only on performance feedback generated during the search. In contrast, the second algorithm searches a limited search space consisting of variants that have been previously identified as beneficial on GPUs. It is based on a local search, which relies on our analysis in Section 3.4.7 of the nearly convex nature of the thread configuration search space, but extends it to deal with runtime variation during query execution.

## 3.6. Candidate selection in large variant spaces

In this section, we describe a framework which enables a database to learn a fast operator implementation for any kind of processor. This framework relies only on performance feedback gathered at runtime during query execution and does not consider additional information to reduce the search space based on specific processor characteristics.

Our framework builds upon the concept of *micro adaptivity* [258], which is used in Vectorwise [359] to automatically select an operator variant that is optimized for the actual machine which runs the query and the specific data processed by a query. By itself, micro adaptivity works best when the number of candidate variants is small. However, as we have shown in Section 3.3, we can easily generate thousands of operator variants, even for a simple selection kernel and a small number of implementation and workload parameters. Therefore, our framework contains an additional step which periodically generates a small pool of variants containing the candidates that are considered and evaluated during query execution. We propose two search strategies to generate this variant pool, a simple greedy search and a genetic algorithm which creates a new pool based on the fitness of the candidates in the previous pool.

Our evaluation shows that the ability of our framework to select a fast operator variant depends on the number of competitive variants on a given processor, which we analyzed in Section 3.3.8. On an Intel Xeon E5620, which has many competitive variants, we consistently find a variant that is only 1.12× slower than the fastest. Conversely, on an Intel Xeon Phi 7120, which has few competitive variants, the median performance of the selected variant after ten queries is 3.4× slower than the fastest.

### 3.6.1. Micro adaptivity and the VW-GREEDY algorithm

Micro adaptivity leverages the vectorized, or block-at-a-time, processing model [38], in which data of a query is split into cache-sized blocks and one or more query processing primitives are called repeatedly on those blocks to evaluate a query. Every primitive exists in multiple variants (called flavors in the original paper [258]), which produce the same result but differ in their implementation details, e.g., a branched and a branch-free implementation of a selection primitive. The goal of micro adaptivity is to select for each block the variant which is expected to be the fastest, given the data characteristics of the block, the query parameters, as well as the properties and state of the machine on which the query is executed.

To select the expected fastest variant, micro adaptivity uses the VW-GREEDY algorithm [258], which casts the problem as a multi-armed bandit [173]. When a primitive is called for the first time, i.e., at the beginning of the first query, each variant is called on a number of blocks to learn about its performance. VW-GREEDY then periodically switches between *exploitation* and *exploration* phases. In an exploitation phase, VW-GREEDY selects the expected fastest variant, based on previously observed performance, and executes it on a number of blocks. During the processing of these blocks, the observed performance of the selected variant might change, and VW-GREEDY incorporates this new information when selecting a variant for subsequent exploitation phases. However, over time, the knowledge about the performance of variants that are not chosen becomes stale and, as the data and workload changes, the selected variant might no longer be the fastest. To adapt, vw-greedy periodically enters an *exploration* phase, choosing a random variant and evaluating its current performance on a small number of blocks. Afterwards, it uses its updated knowledge about the variant performance to select the expected fastest variant in the next exploitation phase. The behavior of VW-GREEDY, e.g., the length of the exploration and exploitation phases, is determined by four parameters, which we describe in Table 3.5.

VW-GREEDY has three beneficial properties which make it suitable as a foundation for our learning framework. First, it has very low overhead because its book-keeping costs are amortized over the tuples contained in a block. Second, each block provides performance information about the currently chosen variant and the periodic exploration phase allows it to update its knowledge about variants that were not chosen recently. Consequently, VW-GREEDY can quickly adapt to data and workload changes. Third, slow variants will only affect the performance of a few blocks during exploration instead of slowing down the entire query.

Table 3.5.: VW-GREEDY configuration parameters, and their values used in this section.

| Parameter | Description |
|---|---|
| $p_{explore} = 512$ | Number of blocks between the start of two exploration phases. |
| $p_{exploit} = 8$ | Number of blocks on which the expected fastest variant is evaluated before selecting a new variant during exploitation. |
| $l_{explore} = 2$ | Number of blocks to determine the runtime of an explored variant. |
| $l_{skip} = 2$ | Number of blocks to skip when determining the runtime of an explored variant to reduce the influence of instruction cache misses. |

## 3.6.2. Variant pool size limitation

By itself, VW-GREEDY is only able to handle cases where the number of existing variants is comparatively small. To show this limitation, we evaluate a selection query on the Nvidia Tesla K40m over a 128 GB column partitioned into 1024 blocks of 128 MB, using randomly selected variant pools of increasing sizes. For each pool size, we repeat the experiment 300 times, and for each run, we construct a new variant pool by randomly selecting variants from the universe of selection operator implementations described in Section 3.3. Note that for this experiment, we also include variants with workload parameters that do not fully utilize the GPU resources, i.e., which have a work group size of less than eight and use fewer work groups than the available 15 compute units of the Nvidia Tesla K40m; in total, there are about 4800 variants.

At the start of the query, we evaluate each variant in the pool on four blocks ($l_{skip} + l_{explore}$) to learn about its performance. After this initial exploration, we select a variant for the remaining chunks using either VW-GREEDY with the configuration parameters described in table Table 3.5, or by picking a random variant for each block as a baseline.

The box plots in Figure 3.13 show the results. A pool of size one can contain either a fast, a slow, or an average variant, resulting in the large spread of the measured runtimes. As the pool size increases, the pool will contain a mix of fast, slow, and average variants. When selecting variants at random, the influence of fast and slow variants is reduced and the runtime distributions contracts and converges towards the mean runtime of all the variants in the universe, as indicated by the dotted line.

In contrast, VW-GREEDY reduces the influence of slow variants in the pool but increases the influence of fast variants. Therefore, the spread of the runtime distribution is contracted even further and the average runtime is shifted towards the runtime of the fastest variant, as indicated by the dashed line. However, VW-GREEDY also limits the practical size of the variant pool to around eight to sixteen variants. With larger pool

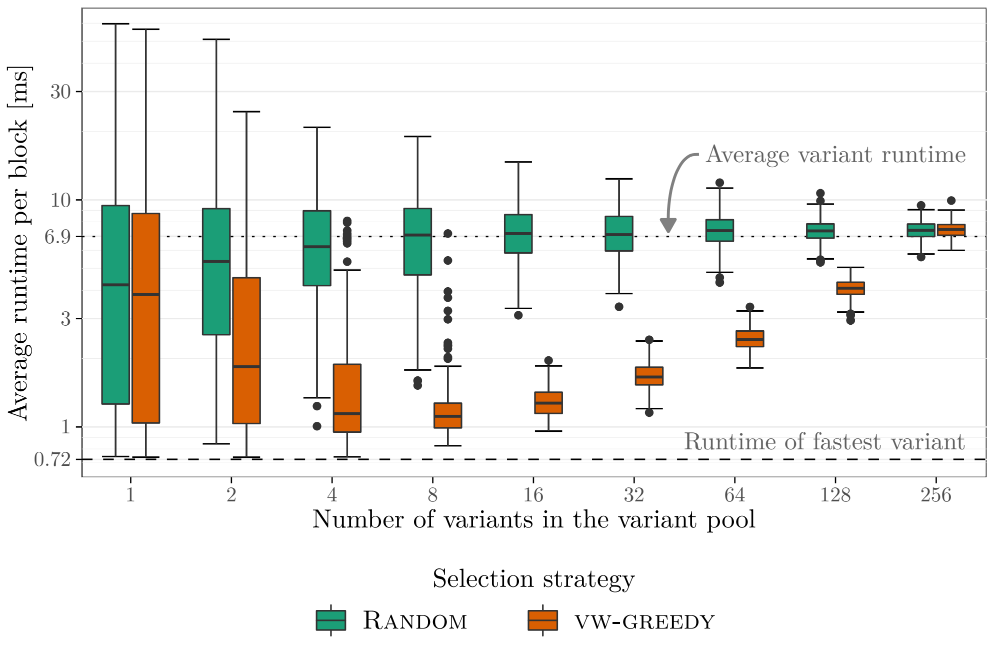


Figure 3.13.: Average variant runtime per block depending on the size of the variant pool on the Nvidia Tesla K40m. After an initial exploration of each variant in the pool, the remaining blocks are processed either by selecting the expected fastest variant using VW-GREEDY or by a randomly selected variant.

sizes, the initial exploration phase, used to determine the performance of each variant in the pool, dominates query execution time. Because we evaluate the performance of each variant on four blocks, at a pool size of 256 variants, the 1024 blocks that make up the query are used in the initial exploration, and there are no blocks left for the exploitation phase. The initial exploration of a large variant pool effectively degenerates to a full evaluation of the entire variant space, which we want to avoid. Thus, the main challenge our learning framework needs to solve is how to select a comparatively small variant pool from a universe that can potentially contain thousands of possible variants.

### 3.6.3. Learning framework overview

Figure 3.14 gives an overview of our learning framework. A parameterized *variant generator* is used to produce variants of an operator or a primitive for a given processor. Individual variants are identified through a *configuration*, which is a predefined collection of implementation parameters, and allowed values for them, which modify the behavior of the variant generator. For example, in Figure 3.15 we summarize the implementation

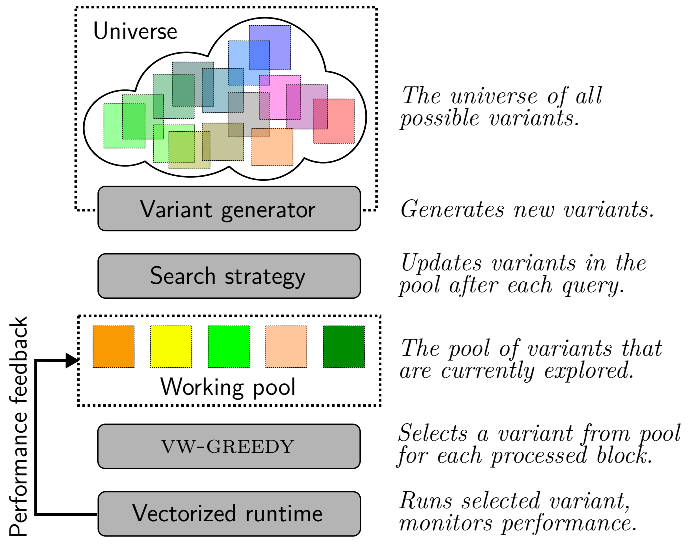

Universe
The universe of all possible variants.
Variant generator
Generates new variants.
Search strategy
Updates variants in the pool after each query.
Working pool
The pool of variants that are currently explored.
VW-GREEDY
Selects a variant from pool for each processed block.
Vectorized runtime
Runs selected variant, monitors performance.
Performance feedback

Figure 3.14.: Overview of the variant learning framework.

`kernel_type`: SEQUENTIAL, GLOBALATOMIC, LOCALATOMIC, REDUCE, COLLECT, TRANSPOSE
`result_type`: `uchar`, `ushort`, `uint`, `ulong`
`branched`: true, false
`unrolled`: true, false
`elements_per_thread`: $1, 2, 4, 8, \ldots, 1024$
`local_size`: $0, 1, 2, 4, 8, \ldots,$ *maximum device-specific local size*

Figure 3.15.: Implementation parameters of the selection kernel and their possible values.

parameters of the selection kernel described in Section 3.3, and their possible values, which make up the variant universe of this primitive.

Instead of instantiating the full universe of all possible implementations, we only use a small *working pool* of around eight to sixteen active variants during the evaluation of a query. Queries are evaluated using a vectorized runtime [38] which allows us to make fine-grained performance measurements of the variants, and then use VW-GREEDY [258] to select the expected fastest variant from the working pool for each block.

In between queries, or after a fixed number of queries, we use a *search strategy* to update the pool based on the collected performance feedback. The search strategy replaces badly performing variants in the pool by newly selected ones. This process continuously improves the quality of the variants in the working pool, bringing the overall performance of the system closer to the optimum in each step. Our learning framework is fairly flexible, allowing us to plug-in different search strategies that differ in how they decide which variants to select next.

### 3.6.4. Search strategies

As discussed in the previous section, the goal of the search strategy is to periodically update the active variant pool, based on the collected performance feedback, by replacing underperforming variants with newly selected ones. We use two strategies in our framework, a greedy search and a genetic search. Both search strategies are initialized with a pool consisting of randomly chosen variants.

#### 3.6.4.1. Greedy search

For the Greedy search, we keep the two fastest variants of the current pool and replace the remaining ones by selecting random variants from the variant universe. This strategy essentially corresponds to a random sampling of the variant universe.

#### 3.6.4.2. Genetic search

For the Genetic search, we also keep the two fastest variants of the current pool. The remaining variants are replaced by following a genetic propagation protocol that generates new variants by combining the configurations of two parents from the current pool. Parents are selected randomly, with a probability proportional to their observed performance, i.e., faster variants have a higher chance of passing on their configurations. For every implementation parameter, both parents have an even chance to pass on their value to the offspring. In order to add genetic diversity and avoid getting stuck in local minima, we also introduce mutations, i.e., we select a random value for each implementation factor with a probability of 0.2.

### 3.6.5. Experimental evaluation

To evaluate our learning framework, we run a series of selection kernels and measure how the query runtime changes over time.

#### 3.6.5.1. Methodology

Each experimental series consists of ten selection queries with a fixed selectivity of 0.5. The queries scan a 16 GB column in 1024 blocks of 16 MB each, and we measure the average runtime per block for each query. After each query, we use a search strategy to update the current working pool, which contains eight variants. In addition to the two search strategies Greedy and Genetic, we also use the following baseline: In the None strategy, the working pool is initialized with random variants and then remains

Table 3.6.: Number of competitive variants, and performance of the slowest variant, of the selection kernel at selectivity 0.5. A competitive variant is at most 10% slower than the fastest.

| Processor | Number of variants | Competitive variants | Ratio | Maximum slowdown |
|---|---|---|---|---|
| Intel Xeon E5620 | 5880 | 1370 | 23% | 32 |
| Nvidia Tesla K40M | 4696 | 129 | 2.7% | 136 |
| Intel Xeon Phi 7120 | 3886 | 6 | 0.15% | 147 |

constant for all ten queries. Each experimental series is repeated 100 times to control for random effects, initializing a random working pool before each repetition.

#### 3.6.5.2. Processors

We run the experiment on three processors, the Intel Xeon E5620 CPU, the Nvidia Tesla K40M GPU, and the Intel Xeon Phi. These processors differ in the number of competitive variants, as we describe in Section 3.3.8.

In Table 3.6, we summarize the properties of the variants search space of these processors when the selectivity is set to 0.5. The Intel Xeon E5620 has many competitive variants; 23% of all variants are at most 10% slower than the fastest. In contrast, on the Nvidia Tesla K40M, only 2.7% of all variants are competitive. The Intel Xeon Phi 7120 is the least competitive processor; only six variants, i.e., 0.15%, are competitive. Compared to the other processors, the Xeon E5620 is a much easier target to optimize for, since there is a 23% chance of selecting a competitive variant at random. Therefore, we expect to clearly see different learning behavior between the three processors.

Note that for this experiment, we include all possible variants in the variant universe, including those that which do not utilize all compute resources, and which use a local size of less than eight on the Nvidia Tesla K40M and Intel Xeon Phi 7120.

#### 3.6.5.3. Results

Figure 3.16 shows the results of the experiment. The runtimes on each processor are normalized relative to the fastest variant at selectivity 0.5, which we determined in Section 3.3.5.

**Intel Xeon E5620.** Let us first take a closer look at the Intel Xeon E5620. Even without using any search strategy, i.e., when using the NONE baseline, VW-GREEDY alone produces a very competitive performance. Except for a single outlier, which is 2.3×

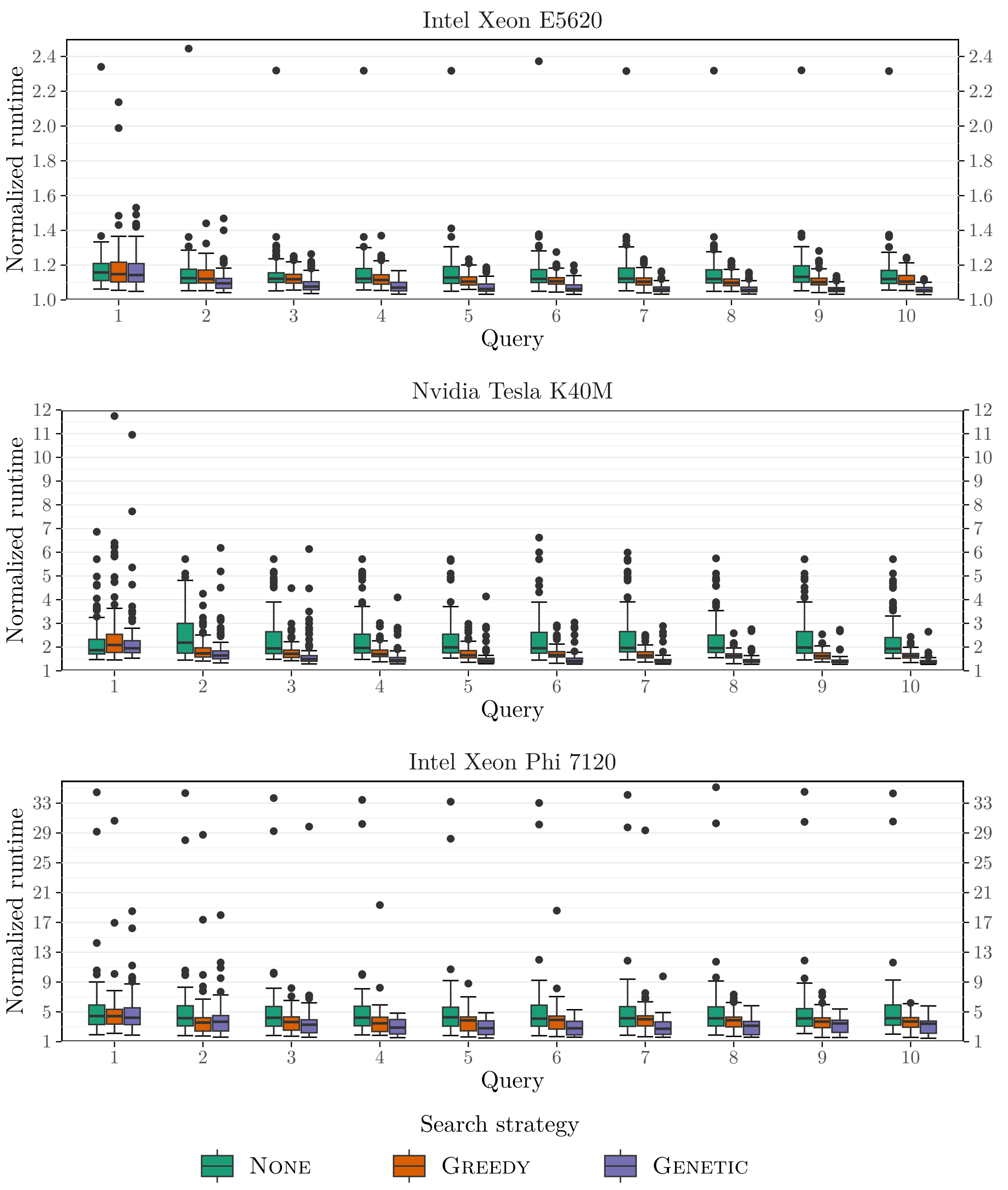


Figure 3.16.: Influence of search strategies on the average runtime per block over a series of queries. The runtimes are normalized relative to the fastest variant for each processor at selectivity 0.5. Note the different scales on the $y$ axis.

slower than the fastest variant, the average runtime per block is at most 1.37× slower than the fastest, and the median normalized runtime per block is 1.16×. This behavior is caused by the runtime performance distribution of the selection kernel variants on the Intel Xeon E5620. As discussed, 23% of all variants are competitive, i.e., they are at most 10% slower than the fastest variant. Consequently, a randomly initialized working pool of size eight, as we use in this experiment, has a 88% chance of containing a competitive variant. Since VW-GREEDY masks the occurrence of slow variants in the working pool, even the NONE strategy produces competitive results.

Building upon the behavior of VW-GREEDY during the execution of a single query, the GREEDY search strategy improves query runtimes even further. After each query, six of the eight variants in the pool are replaced, and we have a chance of about 79% to choose a competitive variant. As a result, after seven queries, the median average runtime of the working pool is competitive and the worst runtime is only 1.23× slower than the fastest. However, note that the performance can degrade somewhat in subsequent queries, due to the initial exploration of a particularly slow variant at the beginning of a query.

The GENETIC search strategy achieves an even stronger convergence towards the optimum. Here, after two queries the median average runtime is competitive, and after ten queries, the median average runtime is 1.05× slower than the fastest, and the worst runtime per block is 1.12× slower. Consequently, the GREEDY and GENETIC search strategies effectively improve the average runtime per block over a series of queries, and they eliminate the influence of particularly bad variants.

**Intel Xeon Phi 7120.** On the Intel Xeon Phi 7120, VW-GREEDY alone cannot achieve as good a performance as on the Intel Xeon E5620. Here, only about six variants, or 0.15%, are competitive, meaning there is only around a 1.2% chance that a random, eight-variant pool will contain a competitive variant. Furthermore, the worst variant on the Xeon Phi is 147× slower than the fastest. Compared to other processors, there are not only few competitive variants but also some *very* slow variants, which are likely to cause a strong performance degradation during the exploration phase. Consequently, when using the NONE strategy, the median average runtime is 4.4× slower than the fastest, and the worst runtime per block is 35× slower.

While the search strategies are able to improve the performance on the Intel Xeon Phi 7120, it is not as impressive as on the Intel Xeon E5620. After ten queries, the GREEDY strategy can improve the average runtime to 3.71× in the median case and to 6.2× in the worst case. However, there are still substantial outliers in the queries series, where the worst case performance is slower than 10×, e.g., for query two, four, six and seven.

The Genetic strategy shows a slightly stronger benefit and is able to improve the average runtime after ten queries to 3.4× in the median case and 5.8× in the worst case. It can also suppress severe outliers effectively after three queries. However, the best median performance of Genetic strategy, a slowdown of factor 2.7×, is achieved after seven queries, but degrades afterwards. This degradation is most likely caused by the initial exploration of very slow variants at the beginning of a query.

**Nvidia Tesla K40M.** On the Nvidia Tesla K40M, 2.7% of all variants are competitive, which is an order of magnitude less than on the Intel Xeon E5620 but an order of magnitude more than on the Intel Xeon Phi 7120. Consequently, the query performance that our variant learning framework can achieve on the Nvidia Tesla K40M is also between the other two processors.

When the None strategy is used, the median average runtime is 1.87× slower than the fastest, and the worst runtime per block is up to 6.9× slower. The box plots indicate a substantial variation in the average runtime per block during the experimental series compared to the other processors; i.e., the average runtime strongly depends on which variants are randomly chosen in the initial working pool.

The Greedy and Genetic search strategies are able to considerably reduce this variation. When using the Greedy strategy, the average runtime per block after ten queries is improved to 1.63× in the median case and 2.4 in the worst case. The Genetic strategy can further improve the average runtime per block after ten queries to 1.37× in the median case. Similarly to the Intel Xeon Phi 7120, the median and worst case runtime fluctuates during the query series due to the initial exploration of slow variants at the beginning of each query.

### 3.6.6. Summary

Our analysis of micro adaptivity shows that it effectively reduces the impact of slow variants only when the number of variants with unknown performance is small. Otherwise, the initial exploration of a large number of unknown variants causes the runtime per block to converge to the average runtime of all variants. Essentially, micro adaptivity degrades to a full evaluation of the variant universe in this case.

To overcome this limitation, we propose a learning framework which incorporates a search strategy to periodically update a small working pool of active variants which are used during query execution. Our experimental evaluation shows that both the Greedy and the Genetic search strategies can effectively improve the performance of the variants in the working pool, which improves average query runtime. However, their effectiveness largely depends on the number of competitive variants in the variant universe of a particular processor. If there are many competitive variants, e.g., on the Intel Xeon E5620, our learning framework reliably finds them after a small number of queries. However, if there are only few competitive variants, e.g., on the Intel Xeon Phi 7120, the performance depends on random variants contained in the initial working pool.

## 3.7. Runtime variation-adaptive local search

In this section, we describe an alternative algorithm to find fast operator implementations at runtime. In contrast to the learning framework described in the previous section, which is agnostic of the operator and the processor, the design of this algorithm incorporates knowledge about the behavior of a specific operator on a specific class of processors to guide the search. Concretely, the algorithm optimizes the implementation parameters of Aggregate kernels during hash aggregation on GPUs, based on the performance analysis of this kernel which we conducted in Section 3.4. Focusing on GPUs allows us to restrict the search space according to well-known best practices, e.g., use coalesced memory access [213] and GPU-optimized parallelization strategies [149]. Focusing on hash aggregation allows us to incorporate one of the main findings of our previous analysis into the design of the algorithm, namely, that the thread configuration search space of the Aggregate kernels is *nearly complex*, i.e., it has a single local minimum when we account for runtime variation (see Section 3.4.7).

The idea of the algorithm is straightforward and is based on a local search of the thread configuration search space. Given a group cardinality and a parallelization strategy, it starts from an initial thread configuration and follows the performance gradient to a local minimum. During the search, the algorithm has to take special care to handle the effects of runtime variation, i.e., performance plateaus and measurement outliers. If the runtimes of two configurations are similar, i.e., if they form a potential performance plateau, it explores the search space from both thread configurations, effectively forking the path taken through the search space. If the runtimes of the measurements in a sample differ too much, it repeats the measurement until the variation is reduced.

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

```
Input: An initial thread configuration $c_0$; a similarity coefficient $s$; a prune factor $p > 1$.
$t(c_0) \leftarrow$ ProcessBlocksWith($c_0$)
$c_f \leftarrow c_0$                                        ▷ Initialize fastest known configuration.
$Q \leftarrow \{c_0\}$
while $Q \neq \varnothing$ do
    $c_i \leftarrow$ Peek($Q$)
    if $t(c_i) > p \times t(c_f)$ then
        Pop($Q$)                                            ▷ Prune slow reference configurations.
    else
        $N \leftarrow$ UntestedNeigborhoodOf($c_i$)
        if $N \neq \varnothing$ then
            $c_j =$ Pop($N$)                                ▷ Evaluate neighbor of current configuration.
            $t(c_i) \leftarrow$ ProcessBlocksWith($c_j$)
            if $t(c_i) \sim_s t(c_j)$ then                  ▷ Keep record of performance plateaus.
                Push($Q, c_j$)
            else if $t(c_j) < t(c_i)$ then                  ▷ Follow gradient in search space.
                ReplaceFirst($Q, c_j$)
            end
            if $t(c_j) < t(c_f)$ then
                $c_f \leftarrow c_j$                        ▷ Update fastest configuration.
            end
        else
            Pop($Q$)                                        ▷ Backtrack from local minimum.
        end
    end
end
```

Algorithm 3.1: Dynamic selection of thread configurations.

### 3.7.1. Algorithm overview

Algorithm 3.1 shows the pseudocode of the local search algorithm. In the following, we describe its key aspects in detail.

#### 3.7.1.1. Notation and definitions

We use $c_i$ to represent a thread configuration and $t(c_i)$ to express its runtime. We define the runtimes of $c_i$ and $c_j$ as similar if one of them is within the interval determined by a similarity coefficient $s$ around the other:

$$t(c_i) \sim_s t(c_j) \iff (1-s)t(c_i) \leq t(c_j) \leq (1+s)t(c_i).$$

#### 3.7.1.2. Inputs, initial steps, and main optimization loop

The inputs of Algorithm 3.1 are an *initial thread configuration* $c_0$, a *similarity coefficient* $s$, and a *pruning factor* $p$. The similarity coefficient is used by our algorithm to identify two runtimes as part of a performance plateau. The pruning factor is used to exclude parts of the search space when exploring multiple branches from performance plateaus. During execution, our algorithm maintains a FIFO queue $Q$ containing reference positions $c_i$ from which it explores parts of the search space. It also tracks the fastest thread configuration $c_f$ it has encountered so far.

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

#### 3.7.1.3. Handling performance plateaus

When comparing the runtimes of two thread configurations $c_i$ and $c_j$, the algorithm distinguishes three results of the comparison to handle performance plateaus.

**Case (1)** If $c_i$ and $c_j$ have similar runtimes with regard to the similarity coefficient $s$, i.e., if they are part of a performance plateau, both are added to the top of the queue (lines 13–14). In subsequent loop iterations, the algorithm follows the performance gradient in the search space from $c_i$ and $c_j$ independently, until one or both of the branches are pruned.

**Case (2)** Otherwise, if $c_j$ is strictly faster than $c_i$, then $c_i$ is replaced with $c_j$ in the queue (lines 15–16).

**Case (3)** Otherwise, if $c_j$ is strictly slower than $c_i$, the algorithm tries out a different untested neighbor $c'_j$ of $c_i$ in the next loop iteration.

If there are no more neighbors of $c_i$, the algorithm has reached a local minimum. It removes $c_i$ from the queue and backtracks to a previously encountered $c'_i$ inside a performance plateau (line 22). Note that by setting the similarity coefficient $s$ to 0, the algorithm ignores performance plateaus and strictly follows the performance gradient in the search space.

#### 3.7.1.4. Handling measurement outliers

When measuring the runtime of a thread configuration, the algorithm has to satisfy two conflicting requirements. On the one hand, we want to reduce the influence of slow thread configurations. On the other hand, we want to reduce the influence of any outliers in the beginning of the measurement. To this end, the algorithm executes a thread configuration on three blocks. It then determines the degree of variation and compares it to a threshold value $v_{max}$. If the variation is below the threshold, the algorithm returns the mean runtime as the measurement. Otherwise, the algorithm discards the first measured value and executes the thread configuration on another block. The algorithm continues until the variation drops below the threshold or it has processed $b_{max}$ blocks. This process is encapsulated by the function `ProcessBlocksWith(`$c_i$`)` in Algorithm 3.1.

Instead of computing the coefficient of variation, as we do in Section 3.4.8, we compute the range between the minimal and maximal measured values and divide it by the mean. This approach significantly reduces overhead by eliminating a costly square root operation that is part of computing the standard deviation. We set $v_{max} = 0.019$, which corresponds to the 99th percentile of the normalized range coefficient computed over windows of three blocks on all GPUs, based on our analysis in Section 3.4.8. In other words, we expect the algorithm to process more than three blocks in only one case out of a hundred.

#### 3.7.1.5. Optimizing multiple parallelization strategies

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

#### 3.7.1.6. Initial thread configurations

The number of loop iteration of our algorithm, and therefore its runtime, depends on the initial thread configuration $c_0$. Based on our analysis in Section 3.4.3, we determine for each parallelization strategy the thread configuration with the lowest normalized runtimes, averaged over all GPUs and group cardinalities.

If WorkGroupLocal aggregation is possible, i.e., if the group cardinality is between one and 2048, we use 4×512 threads as the initial configuration. On the Nvidia Tesla K40m, we also use Independent aggregation with 1×256 threads as an additional initial configuration, but exclude it on newer GPUs because it not competitive. For larger group cardinalities, we use SHARED aggregation with 1×512 threads as the initial configuration. Note that AMD GPUs only support 256 work items per work group, so we adjust the initial configurations accordingly.

### 3.7.2. Support for other operators

The algorithm described in this section can optimize any operator that satisfies the following two requirements. First, the thread configuration search space has to be (nearly) convex, since the algorithm exploits this property to efficiently search it. Second, the algorithm must be able to change the implementation of the operator for each processed block without loosing the progress made by processing previous blocks.

For example, hash aggregation satisfies the second requirement because every parallelization strategy merges the results of processing a block into a single, globally shared aggregation table. This shared hash table encapsulates the global state of the aggregation. Many important database operations satisfy the second requirement, e.g., any operator that materializes its output, such as selections and partitioned joins [46, 87, 258, 268].

Note that some operator implementations may use local state that is incompatible with other implementations, as long as it can be discarded once the block is processed. For example, in our analysis, the number of private hash table copies used by INDEPENDENT and WORKGROUPLOCAL aggregation differs. We can also process blocks with different hash functions, hashing schemes, and load factors.

### 3.7.3. Evaluation

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

#### 3.7.3.1. Experimental setup

We vary the similarity coefficient $s$ from 0 to 0.1, in steps of 0.01, i.e., the runtimes of two measurements can differ by at most 10% for the measurements to be considered part of a performance plateau. Similarly, we vary the pruning factor $p$ from 1 to 1.1, also in steps of 0.01. We set the maximum degree of variation $v_{max} = 0.019$, so that the expected number of processed blocks per measurement is three in 99% of all measurements. We set the maximum number of processed blocks per measurement $b_{max} = 10$.

Each combination of hyperparameters is evaluated on the three measurement samples collected in Section 3.4.3 to control for random runtime variation. To speed up the evaluation, we implement the algorithm in Python and inject the execution runtimes collected in Section 3.4.3 into the algorithm.

#### 3.7.3.2. Metrics

We evaluate Algorithm 3.1 with three metrics. The first metric indicates the *quality of the found thread configuration*. Let $c_f$ be the thread configuration found by our algorithm, $t(c_f)$ its runtime, and $t(c_F)$ the runtime of the fastest thread configuration for a given processor and group cardinality, which we determined in Section 3.4.3. Then the quality of $c_f$ is its normalized runtime relative to $c_F$:

$$quality(c_f) = \frac{t(c_f)}{t(c_F)}.$$

The second metric indicates the *cost expended for optimization* in terms of the number of blocks the system could have processed with the fastest thread configuration. Let $n$ be the number of blocks the algorithm processed until it converged to the found thread configuration $c_f$, and let $c_i$ be the thread configuration which was used to process the block $i$. Then the cost of $c_f$ is the difference between the cumulative normalized runtime and the number of blocks:

$$cost(c_f) = \sum_{i=1}^{n} \frac{t(c_i)}{t(c_F)} - n.$$

This formula takes into account that thread configurations that are marginally slower than the fastest only add a small penalty, regardless of how many thread configurations are tested.

As a final metric, we determine the amount of time the algorithm requires to converge to $c_f$ compared to a full evaluation of the search space.

#### 3.7.3.3. Results

In general, the quality of the found runtime $c_f$ improves, and the cost increases, as we increase the similarity coefficient $c_f$ and pruning factor $p$. For clarity, we report the results of two scenarios.

**Scenario (1)** In the *strict runtime comparison* scenario the algorithm ignores performance plateaus, i.e., $s = 0$ and $p = 0$. This scenario serves as our baseline.

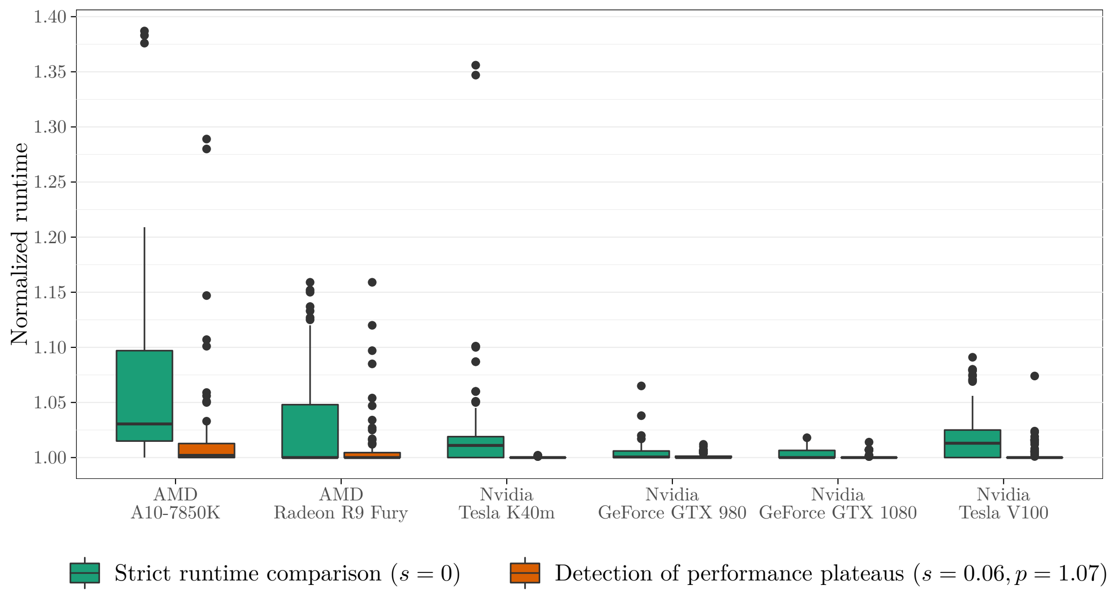


Figure 3.17.: Quality of found configuration.

**Scenario (2)** In the *detection of performance plateaus* scenario, we use a similarity coefficient $s = 0.06$ to detect performance plateaus and a pruning factor $p = 1.07$ to remove slow thread configurations during the search. Increasing these values further does not result in an improvement of the runtime of the found thread configuration in the worst case.

**Quality of found configuration.** Figure 3.17 shows the runtime of the found AGGREGATE kernel relative to the fastest configuration per group cardinality. By simply following the gradient of the thread configuration search space, the algorithm finds the fastest configuration in 36% of our experiments, even if it ignores performance plateaus. When treating thread configurations with similar runtimes as performance plateaus, the algorithm finds the fastest configuration in 62% of our experiments. In fact, in 90% of the time, the found configuration is at most 1% slower than the fastest. The worst-case performance of the found configuration improves from a factor of 1.39× to 1.29×.

The outliers in Figure 3.17 are caused by two factors. In one third of the cases, the algorithm finds a thread configuration that is a local minimum but not the global minimum in the thread configuration search space. For example, on the AMD A10-7850K, it finds a thread configuration that is 1.15× slower than the fastest for a group cardinality of $2^{11}$. In this case, the similarity coefficient and the pruning factor are not aggressive enough to detect this local minimum as part of a performance plateau.

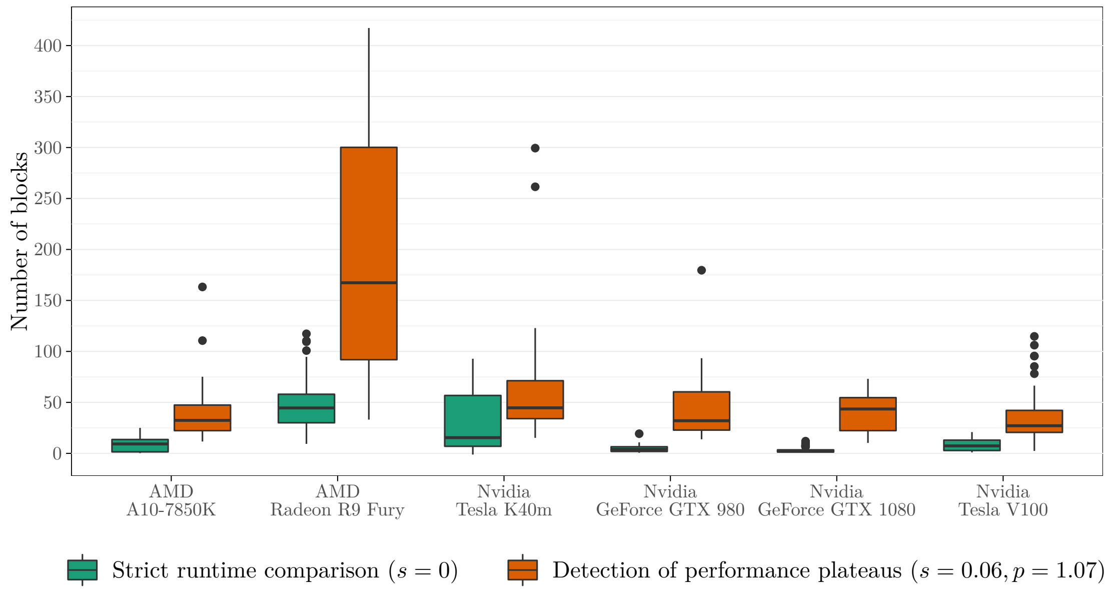


Figure 3.18.: Cost expended for optimization.

In the other two thirds of the cases, the algorithm terminates because the first few blocks of a sample are much faster than the average runtime of the thread configuration in the sample. For example, again on the AMD A10-7850K, the first few blocks of the thread configuration found for a group cardinality of $2^{21}$ are as fast as the fastest thread configuration on this processor. However, subsequent blocks are much slower, so that the average performance of this thread configuration is $1.29\times$ slower than the fastest.

In both cases, the underlying cause of outliers is the runtime variation of the AGGREGATE kernel, which we analyzed in Section 3.4.8.

**Cost expended for optimization.** Figure 3.18 shows how many additional blocks the database could have processed in the time our algorithm converges to a thread configuration, if it had known the fastest configuration from the beginning. This cost increases as the algorithm treats configurations with similar runtimes as performance plateaus and processes additional blocks to reduce the influence of outliers. However, this work is not wasted as the algorithm still makes progress towards the result. When the algorithm detects performance plateaus, the median cost is between 33 and 54 blocks, which in our setup corresponds to 1.06 GB and 1.73 GB of data, respectively. The median cost on the Radeon R9 Fury, which exhibits a high degree of runtime variation, is 172 blocks.

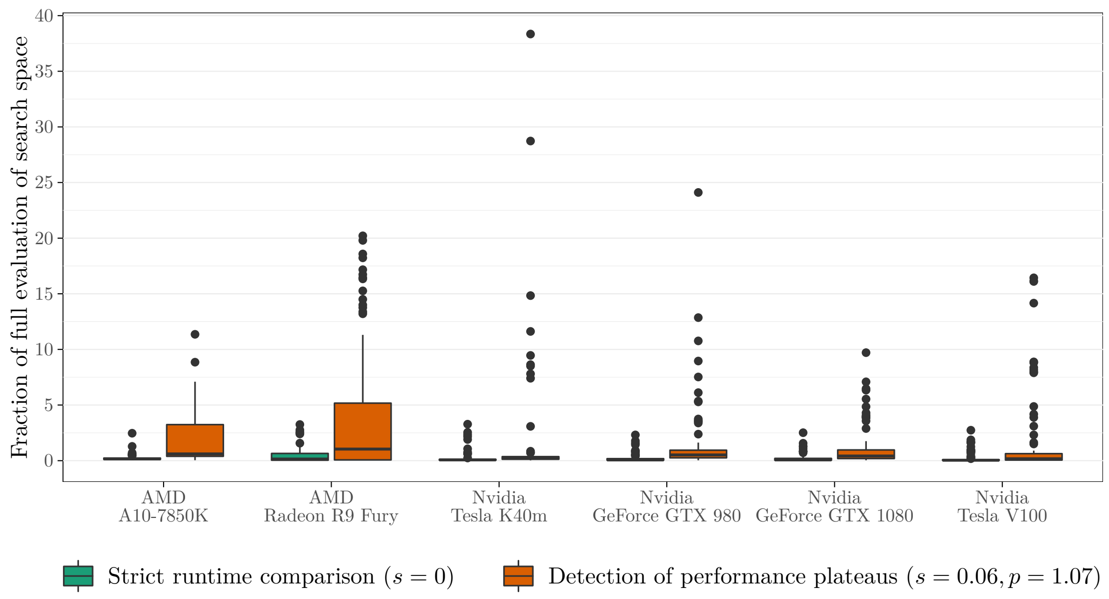


Figure 3.19.: Optimization effort compared to full evaluation.

**Comparison to full evaluation.** Figure 3.19 shows the time our algorithm requires to converge to a fast variant as a fraction of the time required for a full evaluation of the search space. On every GPU, the median runtime is below 1.05% of a full evaluation. There are a few outliers when the algorithm requires more than 20%. In these cases, the thread configuration search space contains many configurations with similar runtimes which form large performance plateaus. However, since these configurations are only marginally slower than the fastest, they do not add significant overhead to query execution.

### 3.7.4. Summary

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

## 3.8. Related Work

In this section, we describe related work, which we group into four categories: (1) works that analyze the impact of implementation variants on heterogeneous hardware, (2) hardware adaption of computational kernels through auto-tuning, (3) auto-tuning databases, and (4) adaptive query processing. The two systems most closely related to our work are Hawk [46], which adapts operator implementations to the processor through auto-tuning, and Excalibur [104], which adapts to the processor during query processing; we describe these systems in detail in Section 3.8.3.1 and Section 3.8.4.1, respectively. For query processing systems which are built on top of OpenCL, we refer to Section 2.4.5 and Section 2.6.2.1.

### 3.8.1. Impact of implementation variants on heterogeneous hardware

Database research on GPUs often compares specially GPU-adapted operator implementations to well-known CPU implementations, e.g., for join [190] or aggregations [149]. However, we are only aware of few works, which compare the impact of multiple implementation parameters to a heterogeneous set of processors.

Rul et al. [272] evaluate the impact of loop unrolling, SIMD vectorization, and the number of work items per work group on OpenCL kernels on an Intel CPU, on GPUs from ATI and Nvidia, and on the Cell Broadband Engine [141]. They observe that the impact of individual implementation parameters varies on the evaluated architectures, that the ATI GPU is much more sensitive to specific implementation parameters than the Nvidia GPU, and that no combination of implementation parameters is optimal for every architecture. Consequently, they conclude that OpenCL is not performance portable.

Broneske et al. [48] evaluate the impact of loop unrolling, branch-free execution, SIMD vectorization, and parallelization on the scan operator on multiple Intel CPUs. They conclude that there is no optimal scan variant for all CPUs.

Gubner et al. [105] evaluate different flavors of data-centric [208] and vectorized [38] execution, including prefetching [199], on ARM, POWER, RISC-V, and x86 CPUs. Importantly, they find that the well-known heuristic that vectorized execution outperforms data-centric execution on workloads with many parallel memory accesses, which is derived from experiments on x86 CPUs [151], does not necessarily hold on other architectures, e.g., the ARM Graviton 1 or the POWER9 CPU. This observation strengthens our conclusion that heuristics derived from the study of a particular hardware architecture, i.e., Kepler GPUs in our work, do not necessarily translate to other architectures. Accordingly, Gubner et al. also identify the need to adapt the implementation of a database engine to the underlying hardware [105].

### 3.8.2. Auto-tuning computational kernels

Auto-tuning has been used effectively to adapt computational kernels to the hardware environment on which they are executed. For example, ATLAS [336] optimizes linear algebra kernels on different CPU architectures; MAGMA aims to do the same on GPUs and heterogeneous CPU/GPU systems [24, 180]. FFTW [85] and SPIRAL [253] optimize discrete Fourier transform and other signal processing algorithms on different CPU architectures. PATUS [53] optimizes stencil computations on CPUs and GPUs.

These frameworks use a variety of search algorithms, including exhaustive search, random search, heuristics, dynamic programming, hill climbing, or evolutionary search. In their discussion on OpenTuner [18], Ansel et al. note that a suitable representation of the search space is domain-specific, since implementation parameters may be independent of each other or strongly coupled. Indeed, some of our implementation parameters are strongly coupled, e.g., the number of work items and the bitmap element size for selection kernels. Good search strategies are also domain-specific, since a search strategy that finds optimal results in one domain may miss good solutions in other domains [18]. The optimal search strategy also depends on the time budget available for the search [253].

Seo et al. [282] investigate the influence of the work group size on OpenCL kernel performance and describe a model-based algorithm to dynamically pick a work group size that minimizes cache misses and improves load balancing on multi-core CPUs. In contrast to GPUs, the thread configuration on CPUs is only determined by the number of work groups and not the work group size.

### 3.8.3. Database auto-tuning

Auto-tuning also has a long history in industry databases and data management research, e.g., learning optimizers [194, 195, 299], index advisors [11, 326, 341], or knob tuning [328, 354, 358]. These systems either train on a representative workload or observe the behavior of different configurations during query processing to improve the performance of *future queries*. Recently, approaches based on deep learning have received widespread interests [167, 178, 194, 195]. However, deep learning approaches that aim to replace traditional database optimizers require a long training time, have poor tail performance, and require retraining when the workload changes [194]. While query execution time depends on the hardware environment, learning optimizers generally focus on data characteristics, e.g., estimated cardinalities [299]. However, systems for knob tuning in cloud environments explicitly take changes to the hardware enviroment into account [328, 354, 358].

#### 3.8.3.1. Hawk

Hawk [46] is an auto-tuning query compiler for heterogeneous processors, i.e., CPUs, GPUs, and the Intel Xeon Phi. Hawk first translates a query into a set of *pipeline programs*, an intermediate representation of a parameterized operator pipeline. Pipeline programs are adapted to different processors with pipeline or operator-specific *modifications*, including (1) processing the data in a single pass, or using lock-free algorithms with multiple passes [116, 118]; (2) selecting between shared or independent aggregation [58], as well as the specific hash table implementation, i.e., linear probing or Cuckoo hashing [237]; and (3) low-level code transformations, e.g., predication, loop unrolling, SIMD vectorization, or memory access coalescing [213]. The modified pipeline programs are then compiled to OpenCL and passed to a vendor-specific OpenCL driver.

To determine which modifications are beneficial for a particular processor, Hawk performs an offline training phase based on a representative query workload; in contrast, our work tries to find fast operator implementations at runtime. Starting with the modifications that have the highest expected impact on performance, Hawk determines the value of each modification dimension which yields the lowest execution time, independently of the other modification dimensions. This strategy does not account for interdependencies between different types of modifications; indeed, the found combination of modifications may not even be a local minimum in the search space. To counteract this deficiency, Hawk repeats this process multiple times from different initial starting points in the search space.

### 3.8.4. Adaptive query processing

Our work is closely related to adaptive query processing (AQP), in which the query execution plan is modified while the query is running. Deshpande et al. [69] provide a survey on early work on AQP, focussing on cardinality estimation and join ordering. In recent works, AQP has been used to rearrange the predicate evaluation order [101, 200, 277, 350], change the join order [200, 277, 324, 334], or switch between latency and throughput-oriented execution backends [152, 162]. Notably, Grizzly [101] is able to switch between shared and independent aggregation at runtime and also specializes the hash table data structure depending on observed data characteristics. Similarly, Permutable Compiled Queries [200] specializes hash aggregation for heavy hitters. When AQP is used together with query compilation, the cost of compiling the query multiple times can be mitigated through fast compilation backends [88, 90, 152], or through patching, i.e., reordering precompiled code fragments [200, 277]. In general, these works also focus on reacting to non-optimal query plans due to data characteristics, and they do not evaluate large search spaces. Nevertheless, they provide valuable contributions to a query processing system that adapts to heterogeneous hardware.

Of note is also the work by Zhang et al. [357], who adapt a general purpose compiler, i.e., Graal [339], to apply micro adaptivity [258] to data-intensive for-loops in generic programs. During compilation, the compiler creates variants to reorder predicates, change their evaluation strategy according to Ross [270], and apply SIMD vectorization if there are few data-dependent conflicts. It then selects the best variant at execution time. Similarly to our work, the authors identify the problem of finding good candidate variants in a large search space. To this end, the compiler performs an exhaustive evaluation over a range of selectivities and computational load estimates of predicates at installation time.

#### 3.8.4.1. Excalibur

Excalibur [104] is a query execution virtual machine, which JIT-compiles data processing primitives to code fragments and adapts them to different processors. Excalibur first segments a query into a set of pipelines, consisting of operators expressed in the VOILA domain-specific language [103]. Every VOILA primitive can be compiled into a vectorized code fragment, which by default results in a vectorized execution [38] of the query. From this baseline, Excalibur adapts the generated executable code using *mutation rules*. These rules include low-level operator implementation parameters similar to ours, i.e., loop unrolling, as well as setting a specific SIMD size to prevent down-clocking of the processor. In addition, there are mutation rules for more high-level plan changes, i.e., the inclusion of bloom filters, swapping operations to reorder filter predicates, and the inlining and JIT-compilation of multiple VOILA primitives into a single code fragment. The latter rule allows Excalibur to seamlessly mix vectorized and data-centric [208] query execution for different parts of a query plan.

Similarly to our work, Excalibur tries to find beneficial variants (called *flavors*) at runtime, and casts the tradeoff between exploiting the fastest known variant vs. exploring the search space as a multi-armed bandit problem. The authors evaluate three exploration strategies, i.e., (1) a random traversal of the search space, (2) a heuristic search based on well-known query plan optimization rules, and (3) an adaptation of Monte Carlo Tree Search (MCTS) [288]. Of these, the heuristic search and MCTS yield the best results, depending on the size of the processed data. This result is inline with our observation that a random exploration of the search space does not quickly converge to a fast variant, and that we have to guide the exploration of the search space using information about the processor characteristics and operator behavior. Furthermore, Excalibur limits the amount of exploration time by a risk budget, which results in most of the exploration being done at the beginning of the query. This behavior is based on the insight that a fast variant that is found at the end of the execution of a query has less chance to pay off its exploration cost. In contrast, we explore the search space between different queries, but also periodically try out every variant in the variant pool during the execution of a single query in order to react to changes in the data distribution [258].

Excalibur only targets query execution on CPUs; to also target GPUs, a number of adaptations are necessary. For example, it should include techniques to mitigate branch divergence [89, 243], and adapt the heuristic search strategy to incorporate best practices for GPU programs [213].

## 3.9. Discussion and conclusion

In the following, we summarize the main insights and results of our investigation of our two research questions for this chapter, i.e., (1) how sensitive are processors to changes in the operator implementations, and (2) how can a query processing systems learn fast operator implementations automatically, without manual tuning? We then discuss open problems and future work, i.e., how to integrate our learning algorithms in a query processing system, how we can deal with data and query changes, and how to efficiently generate code for different operator implementations.

### 3.9.1. Processor sensitivity

To answer the first research question, we implemented variants of two data processing primitives, a selection kernel and a hash aggregation kernel. We evaluated the selection kernel on thirteen CPUs, GPUs, and an Intel Xeon Phi coprocessor, and the hash aggregation kernel on six AMD and Nvidia GPUs, which are based on different microarchitectures. As a measure of sensitivity, we determined the distribution of the variant performance on each processor, and classified a variant as competitive, if it is at most 10% slower than the fastest. We can derive four main results from our analysis.

First, we substantiated the operator variant selection problem on heterogeneous hardware. Even for simple operations and a small number of implementation parameters, we can generate thousands of possible implementation variants. No single variant performs well on every processor, and the fastest variant depends on the type of processor, the vendor, and the specific microarchitecture.

Second, the number of competitive variants varies by two orders of magnitude, from more than 20% on the three single-socket Intel CPUs, to just 0.15% on the Intel Xeon Phi 7120. This behavior makes it much harder to automatically find a fast operator variant on the Xeon Phi.

Third, the influence of implementation parameters is also processor-specific. For example, we found that the number of OpenCL threads has little influence on the performance of the selection kernel on Intel CPUs, as long as all logical processing cores are utilized. In contrast, on an AMD CPU, the number of threads has to precisely match the number of processing cores. Similarly, on the Intel Iris 5100, the Nvidia Tesla K40m, and the Intel Xeon Phi 7120, only very specific thread configurations are competitive.

Fourth, our analysis of hash aggregation on six GPUs showed that previously derived heuristics, which were based on the analysis of a single Nvidia GPU, do not generalize to newer GPUs, which support fast atomic access to local GPU memory in hardware. Even if we take this capability into account, we found that thread configurations that are optimized for one GPU incur significant performance penalties on other GPUs.

Taken together, these findings indicate the need to adapt the operator implementation individually for every processor.

### 3.9.2. Auto-tuning operator implementations

To answer our second research question, we devised two algorithms to automatically adapt the operator implementation to the current processor. The algorithms differ in the type of information they incorporate to guide the search.

The first approach is based on micro adaptivity [258] and relies only on performance feedback gathered at runtime. Our analysis showed that the ability of micro adaptivity to hide the impact of slow variants degrades as the number of variants with unknown performance increases. However, in our setting, i.e., with thousands of variants and unknown processor behavior, we do not know which variants are good candidates to consider. Our solution is to restrict micro adaptivity to a small working pool, and use a search strategy based on a genetic algorithm to periodically update the pool with new candidates to explore. This approach significantly improves the average performance of the fastest variant in the working pool, which micro adaptivity is then able to select in the exploitation phase. However, the absolute performance that we can achieve strongly depends on the random selection of the variants in the initial working pool. Consequently, this approach works well on processors with many competitive variants, e.g., the Intel Xeon E5260, where the median runtime of the variant selected after ten queries is just 1.05× slower than the fastest variant. On the Nvidia Tesla K40m, which has fewer competitive variants, the median runtime after ten queries is 1.37× slower than the fastest, and on the Intel Xeon Phi 7210, which has very few competitive variants, the median runtime after ten queries is 3.4× slower than the fastest.

If the goal of query optimization is to avoid bad query processing strategies, then we unnecessarily handicap ourselves by just relying on runtime performance feedback to find fast operator variants. Therefore, in our second approach, we restrict the search space to a coalesced memory pattern on GPUs and only consider parallelization strategies for the hash aggregation operator which are known to perform well on each GPU and group cardinality. We traverse the remaining search space with a specialized local search strategy, which follows the performance gradient but accounts for runtime variation and possible plateaus in the search space by branching the search when the runtime of two variants is very similar. This approach finds the fastest variant in 62% of the time and the worst case performance is just 1.29× slower than the fastest. The local search relies on our analysis that the thread configuration search space is nearly convex, i.e., it has a single local minimum if we account for runtime variation. Even though we provided experimental evidence for this behavior of the hash aggregation kernels on the six GPUs that we tested, we cannot assume that other GPUs or operators exhibit the same behavior. However, if we cannot assume the existence of a single local minimum, we can probe the search space from multiple points, and use a scheme similar to micro adaptivity to switch between phases in which we explore the search space or exploit previously collected performance data.

One motivation of just relying on runtime performance to guide the search, was to find fast variants on processors with unknown performance characteristics. If we consider the niche Intel Xeon Phi 7120 as an example of such a processor, then this approach did not produce the desired results. The genetic algorithm which we used to generate candidates for the micro adaptivity working pool was not able to quickly find a competitive variant. Therefore, we conclude that we should exploit the available information about a processor, or invest time in microbenchmarks to determine processor characteristics, in order to restrict the variant search space.

### 3.9.3. Dealing with data and query changes

In this chapter, we optimized the variants for a fixed query and constant data characteristics, e.g., a single selection predicate over uniform random data, or a single sum aggregation over a data with fixed group cardinality. In principle, it is possible to start the learning process from scratch if we encounter a new query with new data characteristics. However, this approach has two disadvantages. First, we ignore the information about the processor that we gained during the optimization of variants for prior queries. Second, storing the runtime information about the evaluated variants for each query, in order to reuse them if we encounter the query again, would be infeasible if there are a lot of ad hoc queries. Instead, we should learn a model about the processor behavior based on the information we collect during the evaluation of different variants for a query.

A promising method to learn such a model is Bayesian optimization [47]. Bayesian optimization works well for optimization problems where the objective function, i.e., the runtime of a variant on a block of data, is expensive to evaluate. Based on a prior, i.e., an initial starting variant, Bayesian optimization predicts a new variant to evaluate. This decision is a trade-off between exploration, i.e., testing variants with uncertain runtimes, and exploitation, i.e, choosing a variant that is expected to be fast, and it aims to minimize the number of evaluations. Furthermore, Bayesian optimization can predict multiple new samples [292] to form a working pool for a query from which we can select the fastest variant using micro adaptivity. However, Bayesian optimization is impractical to predict new variants during the evaluation of a query because it is expensive to evaluate compared the execution of a variant on a block of data. In our initial experiments with Spearmint [114], which is geared toward the optimization of hyperparameters for expensive machine learning algorithms, we observed prediction times of about one second. This overhead is acceptable for a set of queries but not for an individual block of data.

### 3.9.4. Variant and code generation

So far, we have omitted from the discussion the generation of executable code for different variants. Because many variants are evaluated on a few small blocks of data and then discarded, the overhead of generating a variant, including the compilation of OpenCL kernel code to the target processor by the OpenCL driver, must be low.

To generate the selection kernel variants in Section 3.3, we enumerate all possible variants which can be generated from the specified implementation parameters, compile the corresponding OpenCL source code, and cache the binary kernel code in the filesystem. However, this approach is not scalable because in order to eliminate the compilation latency during query execution, we move the compilation of variants to the installation time, and compile variants that may not be evaluated. In contrast, the upfront compilation of the variants for the hash aggregation kernels in Section 3.4 is feasible because the generated code depends on a single implementation parameter, the parallelization strategy. Since the thread configuration is set dynamically at runtime, we can evaluate a variants with a different thread configurations without overhead.

However, recent work has made the dynamic generation of variants more practical. Specialized intermediate representations (IRs) and query compilers, such as Flounder IR [88, 90] or Umbra IR/Flying Start [152], which are optimized for relational queries, greatly reduce the query compilation time compared generic query compilers, which are based on LLVM IR [174]. Specifically, Umbra [209] requires a comparable amount of time to prepare the execution of a query as interpreter-based systems, such as MonetDB [37] and DuckDB [256], which do not incur the query compilation overhead [152]. This result makes query compilation feasible for short-running queries, which is essentially what a short-lived variant is. Query compilation time can be further amortized over multiple variants, by compiling code blocks for multiple variants into a single executable and then synthesizing a concrete variant during query execution [277]. A similar strategy can synthesize variants from precompiled primitives during query execution in interpreter-based systems [200].

### 3.9.5. Integration into a query processing system

We can now sketch the design of a complete query processing system which automatically adapts the executed data processing code to the processor it runs on. Such a system first reduces the search space by excluding variants which are expected to be slow, based on processor-specific best practices and/or the results of microbenchmarks. During query planning, the system picks a small number of variants to constructs a working pool. These variants are transformed into executable code by a low-latency query compiler or synthesized from precompiled code fragments in interpreter-based systems. During query execution, the system employs micro adaptivity to pick the fastest variant from the working pool, and to adapt to changing data characteristics. The performance feedback gathered during query execution, as well as query and data characteristics, are fed back into a learning model, which refines the search space in the background, to guide the selection of the variants in the working pool for subsequent queries.

# 4 Processing Java UDFs in a C++ Environment

## 4.1. Problem statement

In the previous two chapters, we investigated the effect of heterogeneous processors on the implementation of query processing systems. In this chapter, we shift our focus from hardware heterogeneity to software heterogeneity, and investigate how to integrate different software components in a diverse data analytics ecosystem, in order to achieve both high programmer productivity and application performance.

Many popular data analytics frameworks, e.g., Hadoop [309], Spark [313], and Flink [308], are written in Java, or in another language that uses the Java Virtual Machine (JVM). The reasons for this state of affairs are partly historical, but Java also offers clear advantages, e.g., the rich ecosystem of languages and tools that are built on top of the JVM, as well as an abundance of developers trained in Java. A key feature these data analytics frameworks is an execution model centered on *user-defined functions* (UDFs). UDFs are used as arguments to second-order functions such as *map* and *reduce* (e.g., in Hadoop) or within declarative queries (e.g., in SparkSQL [20]). By supporting arbitrary code inside UDFs, these systems achieve a high degree of expressiveness and versatility.

However, users may want to implement parts of a data analytics pipeline in a compiled programming language, such as C++, in order to utilize machine resources better, and achieve higher performance, than JVM-based systems [349]. We refer to code which is compiled down to machine code and executed directly by the CPU without an intervening interpreter or JIT compiler as *native code*. UDFs pose a particular challenge in this scenario because there is no straightforward way to link Java bytecode with native code. That is because the JVM abstracts from its internal representation of Java objects and how Java bytecode is executed by the JVM. The Java Native Interface (JNI) [234] provides a mechanism to bridge this gap by allowing Java methods to call native functions and vice versa. Through the JNI, native code can even instantiate an *embedded JVM* to execute Java code. Unfortunately, a JNI call crossing the boundary between native code and the JVM has a considerable runtime overhead compared to a call that does not cross this boundary. Because UDFs are typically written to process an individual tuple or a row at a time, a naive implementation that simply invokes the UDF through JNI for each row incurs this overhead for each row.

Previous solutions to combine UDF-centric frameworks with native execution either reimplement the entire software stack in native code (e.g., TupleWare [62]) or force the user to write UDFs directly in C (e.g., Impala [165]). Consequently, they forego a large amount of existing code and sacrifice interoperability with JVM-based languages. In this chapter, we aim to investigate how to preserve interoperability with the JVM and optimize execution of Java-based UDFs inside a native code environment.

A rather obvious remedy is to amortize the overhead of the JNI call over more than one row, i.e., instead of invoking the UDF for one row at a time, a single JNI call processes a batch of rows. We call such a batch a *stride* and this form of UDF invocation *strided execution*. Existing systems (e.g., Vertica [170]), require users to program against a specific interface to enable strided execution of UDFs that operate on a single row at a time. In contrast, our goal in this chapter is to execute scalar UDFs in a strided fashion transparently without exposing a new interface to the user writing the UDF.

## 4.2. Contributions

In this chapter, we describe how efficiently execute arbitrary scalar Java UDFs in a data processing engine written in C++ without the overhead of executing the UDF over JNI. Specifically, we make the following contributions.

(1) We describe how to use *just-in-time (JIT) compilation* of a *strided execution wrapper* to move the critical loop that invokes the scalar UDF from the C++ engine into an embedded JVM. Additionally, we use Java's *direct byte buffers* to pass data between the engine and the JVM without creating unnecessary copies in many cases. We implement these techniques in the research prototype Wildfire [26], which integrates a C++ columnar engine with Spark (Section 4.4).

(2) We compare our approach of strided execution inside an embedded JVM against executing the UDF directly in Spark or executing an equivalent SQL predicate without a UDF (Section 4.5).

(3) Additionally, we describe how to compile Java UDFs directly to machine code and dynamically link them with the Wildfire engine, in order to avoid the JVM altogether (Section 4.6).

Our evaluation shows that executing Java UDFs inside an embedded JVM in Wildfire is consistently faster, at least 1.12×, than executing them in Spark. The overhead of a UDF-based execution is small. Evaluating a predicate using a UDF is 1.27× slower than an evaluating an equivalent SQL predicate without a UDF. Furthermore, we find that strided execution inside an embedded JVM is generally faster than compiling Java UDFs to machine code, and comparable to hand-written code. Overheads compared to hand-written versions are due to constraints of the Java programming model, e.g., that Java strings are immutable.

## 4.3. Background

In this section, we briefly discuss how Java UDFs are represented in Spark, quantify the overhead of JNI calls, and describe Wildfire, the research prototype we use for our analysis.

### 4.3.1. Scalar UDFs in SparkSQL

Users have to register UDFs with Spark before referring to them in SparkSQL [20] queries, as shown below in Listing 4.1.

```
var offset = 10
sqlContext.udf.register("add_offset", (i: Int) => i + offset)
sqlContext.sql("SELECT add_offset(i) FROM table").show()
```

Listing 4.1: SparkSQL UDF registration and usage.

Typically, SparkSQL UDFs are specified as Scala lambda functions, which are closures, i.e., they capture any free variables used in their definition. In the example above, the `add_offset` UDF adds to its input the value in the variable `offset` that is specified outside of the UDF. If a captured variable is mutable, such as `offset` above, and its value changes between SparkSQL queries, each query will use the latest value. This property can be used to configure more complex UDFs, e.g., machine learning models.

Internally, Scala lambda functions are represented as anonymous Java classes. Listing 4.2 below shows the class definition of the `add_offset` UDF that is generated by the Scala compiler.

```
public final class SparkProgramm$$anonfun$run$1
    extends scala.runtime.AbstractFunction1$mcII$sp
    implements scala.Serializable {
  public SparkProgram$$anonfun$run$1(scala.runtime.IntRef);
  public final int apply(int);
  ...
}
```

Listing 4.2: Lambda function class definition.

We want to point out the following details. First, free variables are captured by the UDF by passing them as an argument to the class constructor (line 4). The variable `offset` is represented internally by the Scala compiler as an `IntRef` type because it is mutable and captured by a lambda function. Consequently, changes to `offset` outside of the UDFs are visible when the UDF is evaluated. If `offset` were immutable, it would be passed as a simple primitive `int` type and would be captured by value.

Second, the actual lambda function is executed by calling the `apply` method of the class (line 5). It adds the value that is passed as an argument to the method to the value of `offset` which was captured by the class constructor.

Finally, note that the name of the UDF, `add_offset`, is not stored inside the Java class. The class name has no connection to the UDF's name.

### 4.3.2. The Java Native Interface

To facilitate interoperability between Java and code that does not run on the JVM, the Java specification defines the Java Native Interface (JNI) [234]. The JNI provides an API to start a JVM, to create and manipulate Java objects inside it, and to execute Java methods on these objects. However, as Figure 4.1 shows, executing Java methods via a JNI call incurs a significant overhead. In order to highlight the JNI overhead, we use a Java method that performs the least possible amount of work, i.e., by returning a constant integer. We invoke the method one billion times from a loop in C++. The figure shows that implementing the loop in C++ code and calling the method via JNI (dashed red line) is two orders of magnitude slower than implementing the complete loop in Java and, thus, avoiding the JNI call altogether (dotted green line).

In order to reduce this overhead, we must move the loop from C++ into the JVM. To this end, we break up the loop into strides. We loop over the strides in the C++ code and execute one JNI call per stride. Instead of calling the Java method directly, we invoke a wrapper method that loops over the rows of a stride (solid blue line). As

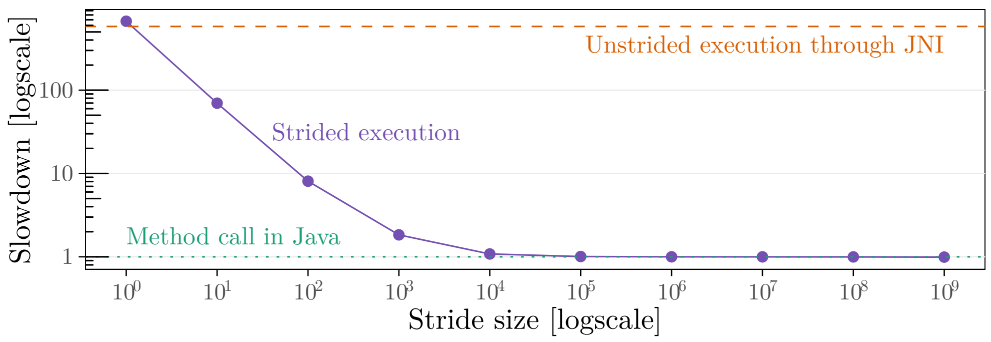


Figure 4.1.: Calling a method through JNI compared to calling a method directy in Java.

we can see, a comparatively small stride size of 10000 rows effectively amortizes the JNI call overhead. Given the low computational complexity of the UDF, this value presents an upper bound on the stride size required to amortize the cost of JNI calls.

### 4.3.3. Wildfire

Wildfire [26] is a research prototype of a distributed, hybrid transactional and analytics (HTAP) system. It is designed to handle very high rates of OLTP traffic while, at the same time, supporting OLAP queries on the latest data.

#### 4.3.3.1. System architecture

The Wildfire architecture [26], shown in Figure 4.2, leverages Spark as a front end for analytical queries. It takes advantage of the existing ecosystem for big data analytics, machine learning, and graph processing. Users can submit analytical queries to Wildfire using SparkSQL or Spark's DataFrame API. The Wildfire engine speeds up analytical queries (solid arrows in Figure 4.2), handles OLTP traffic (dotted arrows), and provides access to newly ingested data before it is stored in the shared file system through a background process (dashed arrows).

In order to move computation to the data, SparkSQL plans are pushed down to the Wildfire engine as much as possible. The well-defined semantics of SQL make this process fairly straightforward, with the exception of UDFs because they allow the execution of arbitrary code. Executing UDFs outside of the Wildfire engine moves the computation away from the data and leads to costly data transfers. Therefore, it is desirable to also move the execution of UDFs from the Spark front end to the Wildfire back end as part of the plan push-down. However, there's a conflict of implementation languages. SparkSQL UDFs are typically written in Scala whereas the Wildfire engine is implemented in C++.

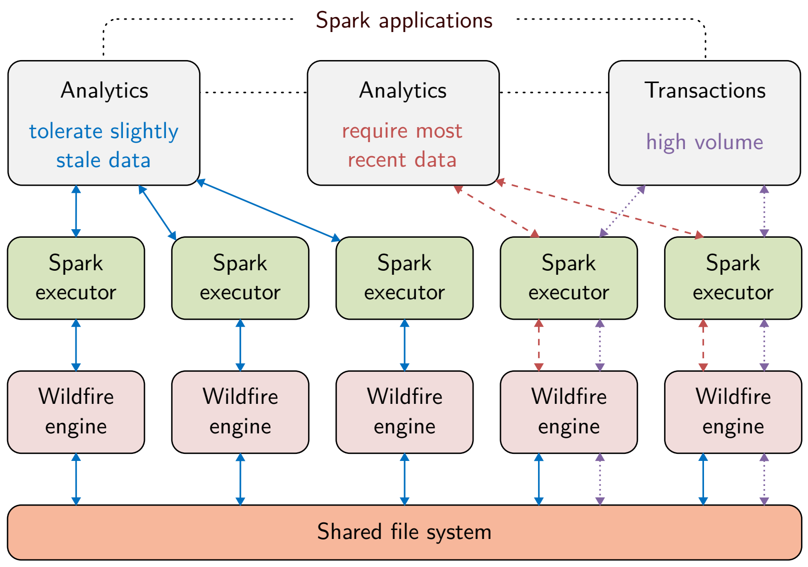


Figure 4.2.: Wildfire architecture.

#### 4.3.3.2. Query engine

Wildfire uses a columnar query engine similar to DB2 with BLU acceleration [260]. It executes queries in a pipelined [208], block-at-a-time [38] fashion. The operator tree of a query is split into individual pipelines. Each pipeline scans a single input table and consists of operators that perform filters, hash table probes for joins, aggregations, etc. Furthermore, pipelines operate on strides of rows. Each operator consumes strides of rows consisting of one or more input columns and produces one or more output column strides. An input stride is fully consumed by a pipeline before the next input stride is processed.

#### 4.3.3.3. Data storage and representation

On disk, data processed by Wildfire is stored as Parquet files [312]. The columnar representation of the Parquet format is kept during execution, i.e., the data of each column is stored as a contiguous memory region. Fixed-sized values are simply stored as an array. Variable-sized values are stored using two data structures, as illustrated in Figure 4.3. A fixed-sized array contains offsets into a variable-sized auxiliary buffer. In this auxiliary buffer, each value is prefixed by its length. We refer to the fixed-sized arrays collectively as *input/output buffers*.

Because the size of input/output buffers does not change during the execution of different blocks of the same query, they are only allocated once at the beginning of the

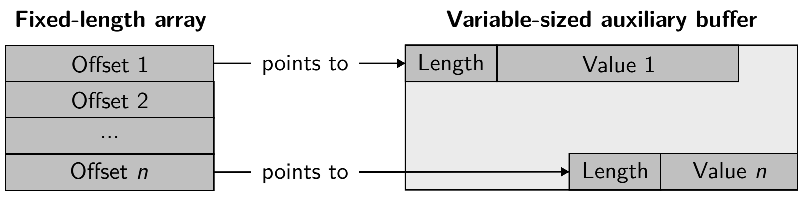


Figure 4.3.: Storage of variable-sized values.

execution of the query and then reused. Wildfire reads the data of each block into the same input buffer. Similarly, the memory region allocated for the output buffer is also reused. However, memory allocated for auxiliary buffers is not guaranteed to be reused because the size of the buffer can change from one stride to the next due to variable-sized values. When a stride's auxiliary buffer has a larger size than the previous one, the memory is reallocated for that buffer instead of reusing the one from the previous stride.

## 4.4. UDF execution in an embedded JVM

We focus our analysis on executing SparkSQL UDFs inside an embedded JVM on all Wildfire engine nodes. An overview of this process, and some of the necessary steps, are shown in Figure 4.4. Notably, we have to transfer the UDF's bytecode and its class dependencies to the engine nodes when the UDF is registered in Spark. We inject the bytecode into the embedded JVM, generate and compile a strided execution wrapper for the UDF, and register it under the name of the UDF. When the UDF is called inside a SparkSQL query, we also need to transfer the current instance of the generated UDF class to execute the UDF with the correct state. During the UDF's execution, we need to pass data from the engine's data buffers to the embedded JVM. These steps are described in more detail in the following sections.

### 4.4.1. Bytecode extraction and transfer

As a first step, we have to transfer the bytecode of the UDF class and all its dependencies to the Wildfire engine nodes when the UDF is registered in Spark. Given their class names, the bytecode of these classes can be retrieved as resources from the Java classloader. Once retrieved, we parse the bytecode to enumerate all referenced types, and repeat this process recursively. Rather than computing a complete transitive closure of all referenced types, we stop the recursion when we encounter a Java or Scala common type, as we do not want to transfer them. Java types are present in the embedded

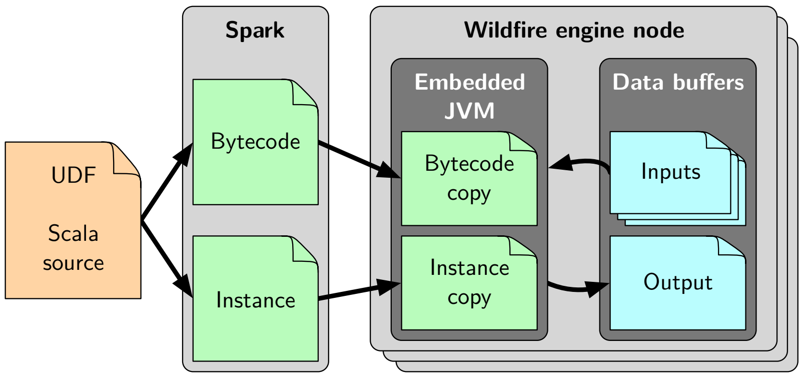


Figure 4.4.: UDF execution inside an embedded JVM.

JVM by default and we also unconditionally load the Scala language library into the embedded JVM.

## 4.4.2. Bytecode injection

Once the bytecode of the UDF and its dependencies has been received by a Wildfire engine node, we have to make the embedded JVM aware of it. The JNI provides the function `DefineClass` to inject a class into a JVM. However, once defined, a class with a particular name cannot be changed. This poses a potential problem, as UDFs defined in the Spark front end can reuse the class names of previous UDF classes. For example, the lifecycles of the Spark front end and the Wildfire engine nodes are independent, and a user could change the implementation of a UDF between different Spark jobs. Similarly, multiple users may use different UDFs with the same Java class name.

Fortunately, different classes with the same name can be isolated in a JVM if they are loaded from different class loaders. Class dependencies, and the strided execution wrapper described below, must be loaded from the same class loader as the UDF class itself. We, therefore, inject the bytecode of the UDF and its dependencies into the embedded JVM by initializing a custom class loader with a mapping of class names to byte arrays containing the respective bytecode as described in the previous section. This class loader can define classes from the stored bytecode as they are loaded by the JVM.

To support UDFs with the same name from different Spark front end session, we have to store a reference to the UDF-specific class loader in a session-specific engine catalog. When a session is terminated, or the UDF is otherwise unregistered from the engine, the class loader can be released and garbage-collected. However, in our prototype, we store it in a global session catalog for simplicity.

### 4.4.3. Passing data

As described in Section 4.3.3.3, input/output buffers are fixed-size, contiguous memory regions. For variable-sized values, another contiguous memory region is used as an auxiliary buffer.

The JNI can wrap memory regions inside a Java direct ByteBuffer object. The contents of this byte buffer can be accessed from Java classes with typed getter and setter methods, e.g., `get` to read a 8-bit byte or `setInt` to write a 32-bit integer. The JVM will make a "best effort" to avoid unnecessary data copies when accessing the contents of direct byte buffer [235]. By wrapping input/output buffers, as well as auxiliary buffers, inside byte buffers, we can pass the data processed by a UDF as opaque memory blocks from the Wildfire engine to the embedded JVM, minimizing JNI call overhead as well as unnecessary copies. Inside the JVM, input buffers that wrap primitive data types are accessed using the typed getter methods and are essentially treated as typed arrays. Buffers that wrap variable-sized values, e.g., strings, need to construct a Java object from the data contained in the auxiliary buffer. Unfortunately, this object construction requires the data to be copied at least once into a temporary Java byte array.

Because Wildfire reuses the input buffers and replaces their contents during the execution of subsequent blocks of a single query, they only have to be wrapped once as a direct byte buffer for each query. Similarly, output buffers only have to be wrapped once. However, as stated in Section 4.3.3.3, auxiliary buffers for input and output are not guaranteed to be reused. Therefore, they have to be wrapped within new direct byte buffers for each query block.

### 4.4.4. Strided execution

To overcome the overhead associated with JNI calls for each input tuple, we automatically generate the Java code for a custom strided execution wrapper for each UDF when it is registered, compile it using the Janino compiler [325], and inject it into the UDF-specific class loader. We need to distinguish two cases. For fixed-size outputs, only one JNI call per stride to the wrapper is necessary. For variable-sized outputs, multiple calls may be necessary if the output exceeds the allocated size of the auxiliary buffer.

The fixed-size version of the wrapper follows the template shown in Listing 4.3. It exports a standardized method call, which takes a UDF class instance, the number of rows, and the input/output and auxiliary buffers as parameters. The wrapper iterates over the input rows and calls the `apply` method of the UDF class for each input tuple, unpacking the input byte buffers in the process using the appropriate typed getter

```
public class StridedExecutionWrapper {
  public static void wrapUdf(
      UdfClass udfInstance,
      int numRows,
      ByteBuffer output,
      ByteBuffer auxiliaryOutput,
      ByteBuffer[] inputs,
      ByteBuffer[] auxiliaryInputs) {
    for (int i = 0; i < numRows; ++i) {
      output.putX(udfInstance.apply(
          inputs[0].getX(),...,inputs[N].getX()));
    }
  }
}
```

Listing 4.3: Strided UDF wrapper template.

methods, i.e., `get`$X$ in line 10. The output is then stored in the output buffer using the appropriate setter method, i.e., `put`$X$ in line 10. Note that the example assumes that input types are primitive; if strings are used, a temporary String object needs to be constructed from the auxiliary buffer. This process is omitted in the template of the wrapper.

The variable-sized version extends the template in Listing 4.3 to handle multiple invocations, and accept and return internal state for each invocation. The state consists of the index of the last processed row in the current stride and the result of the last UDF invocation wrapped inside a Java object. It is used as a signal that the engine has to resize the auxiliary buffer and is otherwise opaque to the Wildfire engine. In our prototype, we initialize the auxiliary buffer with the size of the largest buffer seen so far since the start of the query, and double its size whenever a stride exceeds it.

Figure 4.5 illustrates this process using an example where the auxiliary buffer needs to be resized once to fit the results of the UDF. Initially, the wrapper is invoked with an empty state, i.e., passing `null` to it. The wrapper then loops over input rows as shown in Listing 4.3. If adding the result of the current row to the auxiliary buffer would exceed its capacity, the row's index and the current result is stored in the state object which is returned to the engine. The engine then extends the memory region containing the output auxiliary buffer, wraps it inside a new direct byte buffer, and calls the wrapper again with the previously returned state, i.e., the previously computed result of the UDF and the index of the corresponding row. The wrapper adds the stored result to the new auxiliary buffer and continues processing of the block after the row stored in the state

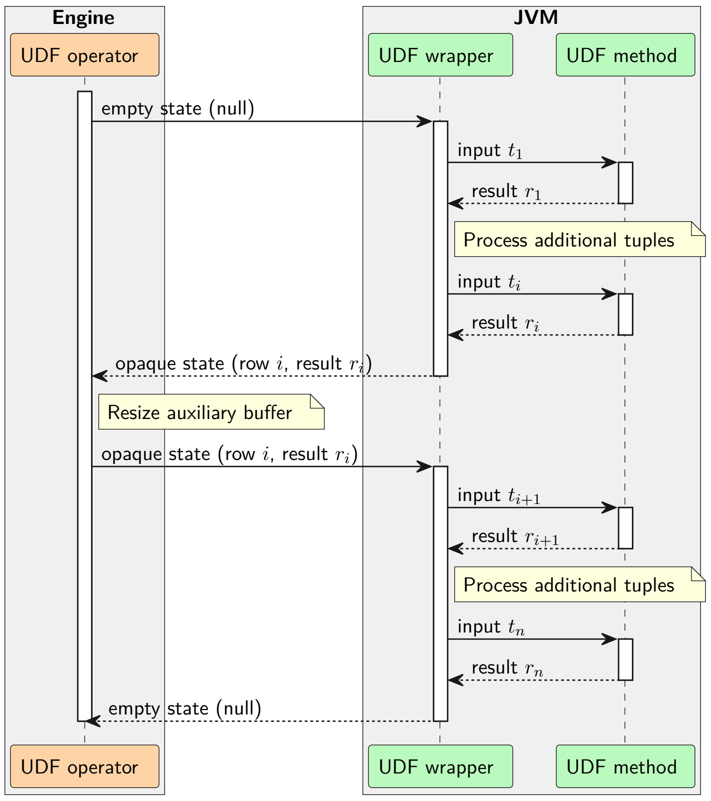


Figure 4.5.: Control flow for variable-sized outputs.

object. Once all rows have been processed, the wrapper signals the end of processing by returning an empty state.

### 4.4.5. Detailed architecture

Figure 4.6 depicts our architecture to execute UDFs in an embedded JVM in detail. The workflow is broken down into three parts, which are implemented by three operators evaluated by the Wildfire engine.

#### 4.4.5.1. Register UDF operator

The first operator is invoked by the Spark front end when the UDF is registered. On the Spark front end, we retrieve the UDF bytecode from the Spark class loader, parse it for dependencies, retrieve the bytecode of dependent classes, and transfer it over the network to the Wildfire engine nodes. The `Register UDF` operator accepts the bytecode of all involved classes and injects them into the embedded JVM using a UDF-specific class loader. It then compiles a strided execution wrapper for the UDF and injects it

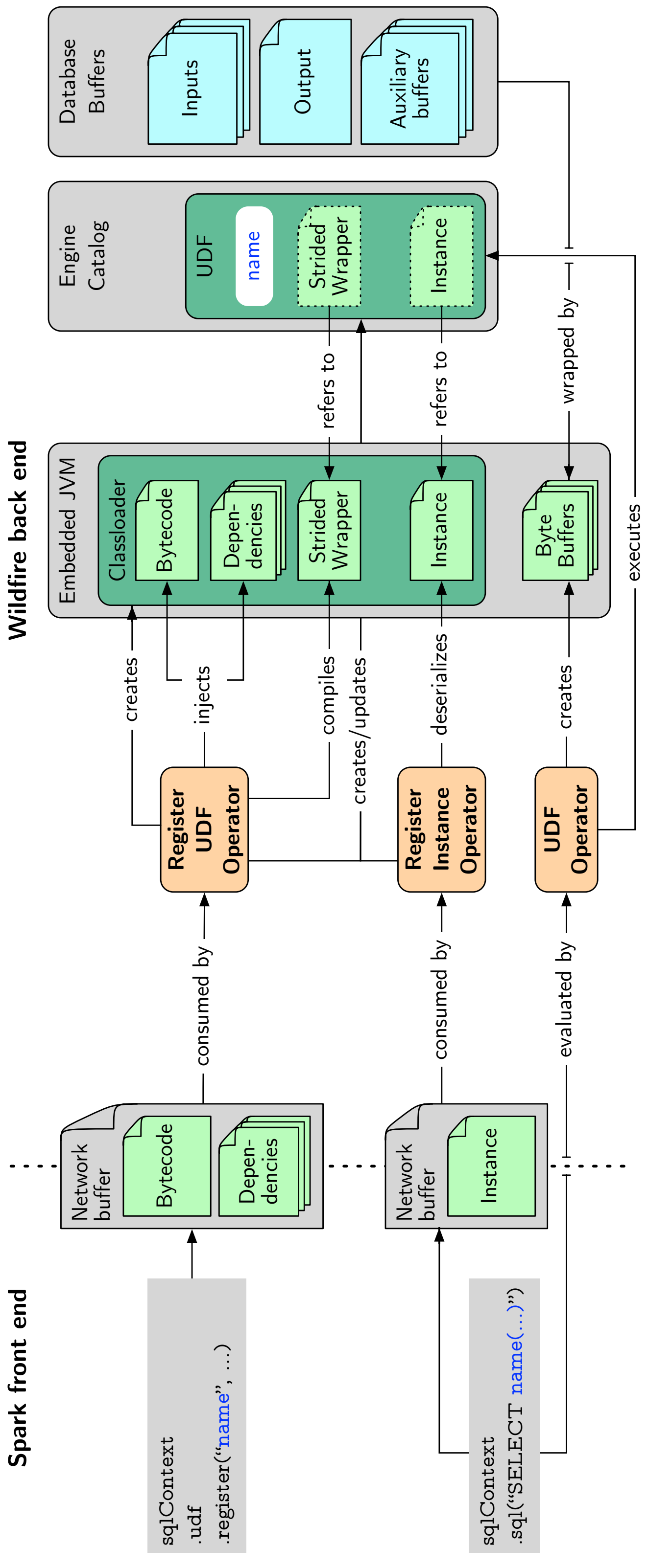


Figure 4.6.: Embedded JVM architecture.

into the JVM. Finally, a reference to the strided execution wrapper is stored under the UDF's name in the engine's catalog.

#### 4.4.5.2. Register UDF instance operator

The `Register UDF instance` operator is invoked by the Spark front end when a UDF is encountered in a SparkSQL query. On the Spark front end, we serialize the UDF instance and all captured variables using the Java serialization API. The serialized data is then transferred over the network to the Wildfire engine nodes. The Wildfire operator deserializes the UDF class instance using the UDF-specific class loader and stores it under the UDF's name in the engine's catalog.

#### 4.4.5.3. Evaluate UDF operator

Finally, the `Evaluate UDF` operator is inserted into the operator tree constructed by the Wildfire Catalyst component to evaluate a SparkSQL query. When it is invoked on the first query block, it wraps the engine's input/output buffers as Java direct byte buffers. It also wraps auxiliary buffers for each block if necessary. The operator retrieves the UDF class instance and the UDF-specific strided execution wrapper from the engine's catalog and calls the wrapper, passing the UDF class instance and the input/output and auxiliary buffers as required. For variable-sized outputs, the wrapper is called until the input has been processed entirely.

### 4.4.6. User-defined types

In our prototype, we support UDFs with primitive types and strings as parameters. As stated in Section 4.4.3, primitive types can be accessed from Java's direct byte buffers without making an extra copy. However, to process string parameters, we need to construct a Java String object and also copy the data into a Java byte array.

Similarly to string parameters, user-defined types (UDTs) also require us to create an object of that type. If the UDT can be decomposed into a fixed number of primitive types or strings, constructing it is a straightforward task that can be accomplished by calling an appropriate constructor. Thus, the cost of supporting such simple UDTs is similar to supporting strings, i.e., creating the object and copying the data internally in the constructed instance.

A UDF parameter can also be a more complex nested data structure, containing optional and/or repeatable components. The Parquet format, which Wildfire uses as its storage scheme, supports nested structures through so-called *repetition* and *definition levels*, which are stored as additional columns alongside the decomposed components [198]. However, supporting UDTs and nested structures as UDF parameters is outside the scope of this prototype.

### 4.4.7. Security considerations

Database management systems that support UDFs generally have to take measures so that buggy or malicious UDFs cannot crash or otherwise harm the database process. A popular technique is fencing, whereby UDFs run in separate processes and communicate with the database through some form of interprocess communication. Running UDFs inside an embedded JVM can provide a similar mode of separation. We assume that the JVM implementation itself is robust, such that a UDF cannot crash it, and, therefore, also not the process that embeds it. Instead, errors such as dereferencing null values or buffer overflows will throw an appropriate exception. These errors can then be handled gracefully by the calling code.

It is indeed possible to purposefully crash a JVM through the JNI interface, e.g., by calling JNI functions with parameters of the wrong type. However, we can reasonably guard against this contingency since we only call our runtime-compiled UDF wrapper, which has a fixed signature, and not arbitrary methods.

Another consideration is the execution of UDFs that do not crash the database process but are otherwise malicious, e.g., by accessing the file system or opening a network socket. To guard against such UDFs, we can use Java's security manager mechanism [236] to restrict access to specific Java APIs. A UDF that exceeds the permissions set by the security manager will throw an exception which can be handled by the calling code. However, such behavior is currently not implemented in our prototype.

### 4.4.8. Summary

In this section, we showed how to execute a SparkSQL UDF inside an embedded JVM in a strided fashion, in order to reduce the JNI call overhead. Specifically, we can automatically generate a strided execution wrapper, which supports fixed-sized and variable-sized inputs and outputs, as well as primitive types and arbitrary Java objects. Furthermore, by wrapping the Wildfire engine input/output buffers in Java direct byte buffers, we can reduce data copies between the engine nodes and the embedded JVM, especially if the UDF works on primitive types.

## 4.5. Evaluation

In this section, we evaluate the techniques described in Section 4.4 using a set of queries containing scalar UDFs, which are representative of different use cases and resource requirements. Our analysis focuses on the effect of UDF execution on query performance, and excludes other influences, such as disk I/O costs. This is a fair evaluation since many data analysis workloads consist of multiple Spark tasks and intermediate results are kept resident in memory to improve performance [345]. Therefore, we devise a number of microbenchmarks that model UDFs with different computational requirements, ranging from simple UDFs that are bound by data movement, to more complex UDFs that are compute-bound. We also investigate the effect of the Java type system, i.e., using primitive types vs. Java objects, on UDF performance.

Specifically, we use a simple *range predicate* to evaluate the cost of UDF invocation of UDFs that are bandwidth-bound; a *fixed-point iteration* to evaluate compute-heavy UDFs; a *word length* UDF that takes a variable-sized Java object as input; and an *upper case* UDF, that additionally produces a variable-sized Java object as output.

### 4.5.1. Test system and setup

Before discussing the evaluated UDFs in detail, we want to describe our test environment and experimental setup. Our test machine runs Ubuntu 16.04 LTS and the Oracle JDK 112. Every experiment is executed on a system with four Intel Xeon X7560 processors running at 2.27 GHz. Except for a thread scaling experiment described in Section 4.5.6, our experiments execute on a single thread. The machine contains 512 GB of memory. On disk, the data is stored in uncompressed Parquet files, using a RLE/PLAIN encoding. Note that on-disk data is accessed from the Linux buffer cache. We observed virtually no data being read from the disk during the execution of the experiments. Also note that we use full table scans and do not build any indexes on the data.

We run each experiment ten times and discard the first result. We report the mean value of the run times of the remaining nine results. We also show one standard deviation as error bars. We measure the wall-clock time required to issue the SparkSQL query and materialize the result in the Spark front end. In particular, we include the time required to transfer the UDF class instance from Spark to Wildfire, including serialization and deserialization, because this cost is incurred for each query using the UDF. However, we do not include the time required to transfer the bytecode of the UDF class and its dependencies, nor the compilation of the strided execution wrapper, as these two steps are only performed during the registration of the UDF. In general, we compare the

Table 4.1.: UDF stride sizes.

| UDF | Stride size |
|---|---|
| Range | 4096 |
| Distance | 512 |
| Word length | 1024 |
| Upper case | 1024 |

execution time of the strided execution wrapper to two baselines, (1) the execution of the SparkSQL query in Spark 2.0.2 without the involvement of Wildfire, and (2) the unstrided execution where we invoke the original UDF method for each tuple through JNI. We report normalized wall-clock time relative the execution of the queries in Spark, i.e., our first baseline.

For each UDF, we have determined the minimal required stride size, so that doubling it does not yield a further improvement. We list these stride sizes for each UDF in Table 4.1. Note that determining the correct stride size for a query does not only depend on the UDF but also on other aspects of the execution environment, such as cache sizes and tuple width. Such an analysis is outside of the scope of our prototype.

### 4.5.2. Range UDF

The first query uses a range predicate on an integer column. Such a predicate can be expressed natively in SQL, allowing us to evaluate the overhead of formulating it as an UDF instead. Furthermore, by setting the range so that all values are selected, we can also measure the total cost of UDF invocation.

Consequently, we evaluate three versions of the query on Wildfire which are shown in Listing 4.4. The first contains no predicate, i.e., it selects all rows from the table. The second version formulates the range query as a standard SQL predicate. The third version encapsulates the range predicate inside a UDF. Each version of the query has a selectivity of one since the first version does not contain any predicate. Therefore, in order to minimize the amount of data that is transferred from the Wildfire engine nodes back to the Spark front end, and highlight the cost of the UDF, we wrap the results of each query in an aggregation function.

This UDF is designed to measure UDF invocation overhead. It has a low computational load, i.e., the influence of UDF invocation is maximized compared to the actual computation. The difference between the run times of the first and second query yields the cost of applying the predicate as a native SQL expression, whereas the difference

```
SELECT sum(a), count(a) FROM R
```

(a) No predicate

```
SELECT sum(a), count(a) FROM R WHERE min < a AND a < max
```

(b) SQL predicate

```
-- def filter(a: Int, low: Int, high: Int) = low < a && a < high
SELECT sum(a), count(a) FROM R WHERE filter(a, min, max)
```

(c) UDF predicate

Listing 4.4: Range predicate queries.

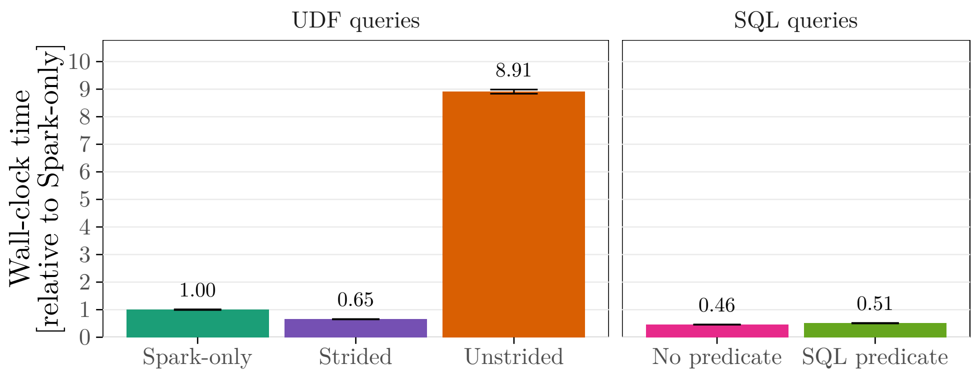


Figure 4.7.: Run time of the strided execution of the range UDF, compared to execution in Spark and unstrided execution, and equivalent SQL queries; single-threaded execution.

between the second and third query yields the cost of formulating the predicate as a UDF compared to native SQL.

We execute the queries on a 10 GB table with one column containing approximately 2.5 billion random integers. For the strided execution we use a stride size of 4096, which is a conservative value. The results are shown in Figure 4.7. As stated, the UDF is computationally lightweight, i.e., the difference between the query without a predicate and with a SQL predicate is small. Consequently, there is an order of magnitude difference between strided and unstrided execution. Evaluating the predicate inside a Java UDF is 1.27× slower than evaluating it directly in SQL. However, evaluating the UDF inside an embedded JVM in Wildfire is 1.54× faster than evaluating it in Spark directly.

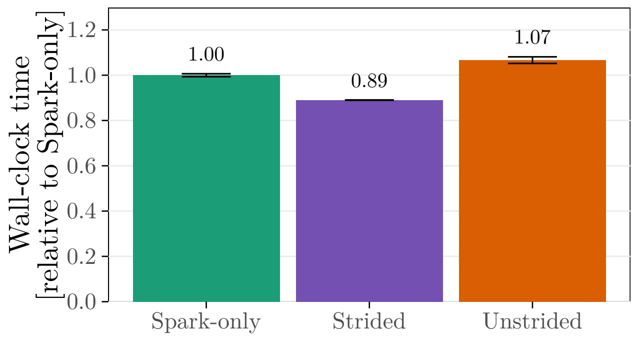


Figure 4.8.: Run time of the distance UDF query.

### 4.5.3. Distance UDF

The second query, shown below in Listing 4.5, contains an example of a computationally heavy UDF. It uses the inverse Vincenty's formulæ [331] to compute the distance between two points on an oblate spheroid, such as Earth. This algorithm contains several trigonometric functions and is based on a fixed-point iteration. While it can be implemented directly in SQL using recursive common table expressions, it is much more straightforward to implement inside a UDF.

```
-- distance: UDF implementing Vincenty's formula
SELECT sum(lat), count(lat) FROM R
WHERE distance(lat, lon, 0, 0) > 100000;
```

Listing 4.5: Computationally intensive distance UDF query.

We evaluate the UDF on a 1 GB table containing approximately 64 million random points uniformly distributed on a sphere. The UDF computes the distance between each point and a point we arbitrarily choose at latitude and longitude 0°. On average, the UDF requires five iterations to converge. To eliminate data transfer overheads, we again wrap the results in an aggregation function, and minimize its overhead by choosing the selectivity of the query so that no point is actually selected. Note that the distance UDF processes forty times fewer rows and ten times less data than range UDF. For the strided execution we use a stride size of 512.

The results are shown in Figure 4.8. Contrary to the range query, the JNI call overhead is dwarfed by the computational complexity of the UDF itself. Consequently, unstrided execution is only 1.2× slower than strided execution. In general, the necessity of strided execution to achieve good performance diminishes as the computational load of the UDF increases. The UDF runs 1.12× faster in Wildfire than in Spark.

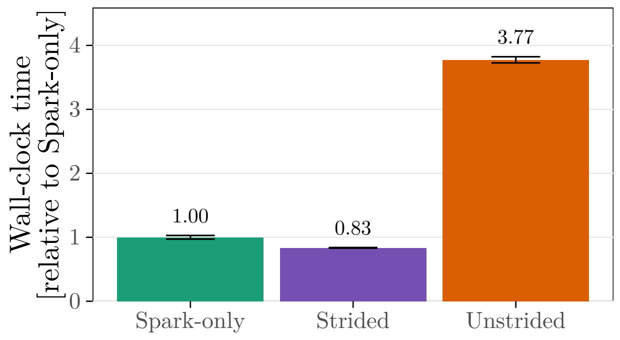


Figure 4.9.: Run time of the word length UDF query.

### 4.5.4. Word length UDF

The third query, shown below in Listing 4.6, returns the length of an input string.

```
-- def length(s: String) = s.length
SELECT sum(length(word)), count(word) FROM R
```

Listing 4.6: Word length UDF query.

The query evaluates the execution of UDFs that operate on Java objects as input, using variable-sized strings as an example. Unlike primitive types, which can be read directly from the direct byte buffers used to pass data between the engine and the embedded JVM, Java objects used as inputs to the UDF first need to be constructed. This involves copying the data from the direct byte buffer into the data structures that back the Java object, which increases the data movement cost of the UDF.

We evaluate the UDF on a data set of about 80 million randomly generated variable-sized words. The word length follows a Poisson distribution with $\lambda = 9.7$, the average length of distinct English words [291]. The approximate size of the data set is 1 GB. For the strided execution we use a stride size of 1024.

The results of this experiment are shown in Figure 4.9. As the UDF is also computationally lightweight, there is a significant difference, about a factor of 4.5, between strided and unstrided execution. However, due to the memory access entailed in constructing String objects, which is required in both versions, the difference is not as big as for the range query. The UDF runs 1.20× faster in Wildfire than in Spark.

### 4.5.5. Upper case UDF

The final query, shown below in Listing 4.7, transforms an input string into upper case.

```
-- def upper(s: String) = s.toUpperCase
SELECT count(upper(word)) FROM R
```

Listing 4.7: Upper case UDF query.

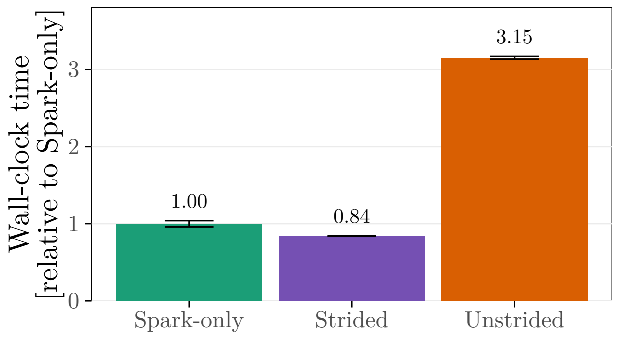


Figure 4.10.: Run time of the upper case UDF query.

This query evaluates the generation of variable-sized auxiliary output buffers. Since the UDF is opaque to the Wildfire engine, the size of the auxiliary buffer is not known when processing of a block starts. The engine needs to resize the buffer when the actual output exceeds the size estimation. As stated in Section 4.4.4, we use a simple strategy that always allocates the size of the largest output block seen so far since the start of the query, and doubles the size estimation if a block exceeds it.

We evaluate the UDF on the same data set used in the word length UDF and show the results in Figure 4.10. Compared to the word length UDF, the upper case UDF requires two additional passes over the string, one to transfer its content into upper case and a second to write the string as an array into the output auxiliary buffer. Consequently, the UDF is dominated by String data copies, and each of the strided, unstrided, and Spark versions take about twice as long as the previous UDF (not visible due to normalization in Figure 4.10).

### 4.5.6. Thread scaling

So far, we have run all UDFs in a single-threaded environment. In a final experiment, we want to evaluate how our embedded JVM approach scales with multiple threads. Our test system is a four-way NUMA machine with eight physical cores per socket that support hyper-threading. In total, there are 64 logical cores in the system. We run the range UDF query from Section 4.5.2 in Wildfire and increase the number of threads from one to 64.

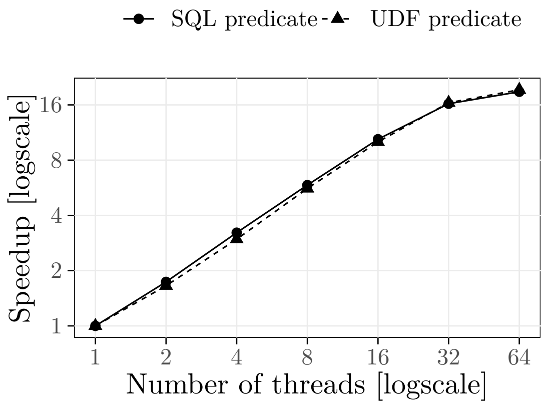


Figure 4.11.: Scaling with multiple threads.

We show the results of two versions of the range query in Figure 4.11. Dots represent the query where the predicate is evaluated directly in SQL and triangles represent the query with a UDF predicate. For each version we show the speedup compared to single-threaded execution. As we can see, the system scales almost linearly with the number of threads. In particular, the speedup of the query with the UDF predicate closely follows the query with the SQL predicate, indicating that the embedded JVM does not introduce overheads that limit parallelization.

The drop-off in the curve between 32 and 64 threads is caused by the way data is partitioned on disk into Parquet row groups. During import, Wildfire partitioned the 10 GB into 161 row groups. Most of these row groups are 64 MB large, except for the first and the last which are smaller. Each row group is processed by a different thread in a work-stealing fashion. Since 161 is almost evenly divisible by 32 but not by 64, some threads are starved of data when we use 64 threads. Half of the threads process three row groups whereas the other half only process two.

### 4.5.7. Summary

Our approach of embedding a JVM inside the Wildfire engine nodes can effectively bridge Java code and native code. A naive execution strategy, which uses a JNI call to invoke the UDF for each tuple, incurs a large overhead for computationally lightweight UDFs. The key to overcome this overhead is to compile a UDF-specific strided execution wrapper at run time and move the critical loop from outside the JVM into it. The strided execution wrapper has the additional benefit of allowing us to pass opaque data buffers between the Wildfire engine and the embedded JVM using direct byte buffers, thereby

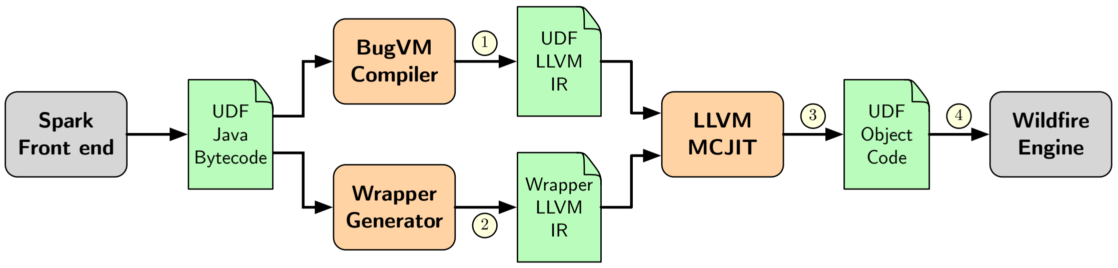


Figure 4.12.: Compiling Java UDFs to machine code.

further reducing JNI call overhead to handle UDF arguments. The benefit of strided execution diminishes as the computational complexity and memory bandwidth of the UDF increases.

## 4.6. Compilation to machine code

The range query shows that the execution of a UDF inside an embedded JVM still incurs a small but significant overhead compared to the execution of the semantically equivalent SQL query. In this section, we investigate whether it is possible to improve the performance of Java UDFs by compiling them to machine code at run time. The approach is sketched in Figure 4.12. It works as follows: (1) When the Spark front end registers a UDF, we first translate the Java bytecode into LLVM [174] intermediate representation (IR). For this step, we use the BugVM compiler [49], which is described in more detail below. (2) Based on the Java UDF, we also generate LLVM IR for a wrapper function that calls the UDF for each tuple. (3) We then link, optimize, and compile the UDF and wrapper IR fragments to object code using LLVM's MCJIT. (4) The object code is dynamically loaded by Wildfire and the wrapper is executed for each block.

### 4.6.1. BugVM

BugVM is an ahead-of-time compiler for JVM-based languages [49]. It provides a compiler that translates Java bytecode to LLVM IR and a custom implementation of the Java runtime environment. BugVM leverages LLVM to produce machine code from the generated IR. This machine code is linked against the custom Java runtime resulting in a fully contained native binary.

The BugVM compiler uses Soot [327] to translate Java bytecode into a typed three-address IR in static single assignment (SSA) form. Instead of following the JVM's stack-based semantics, this IR is already register-based and can be easily translated to

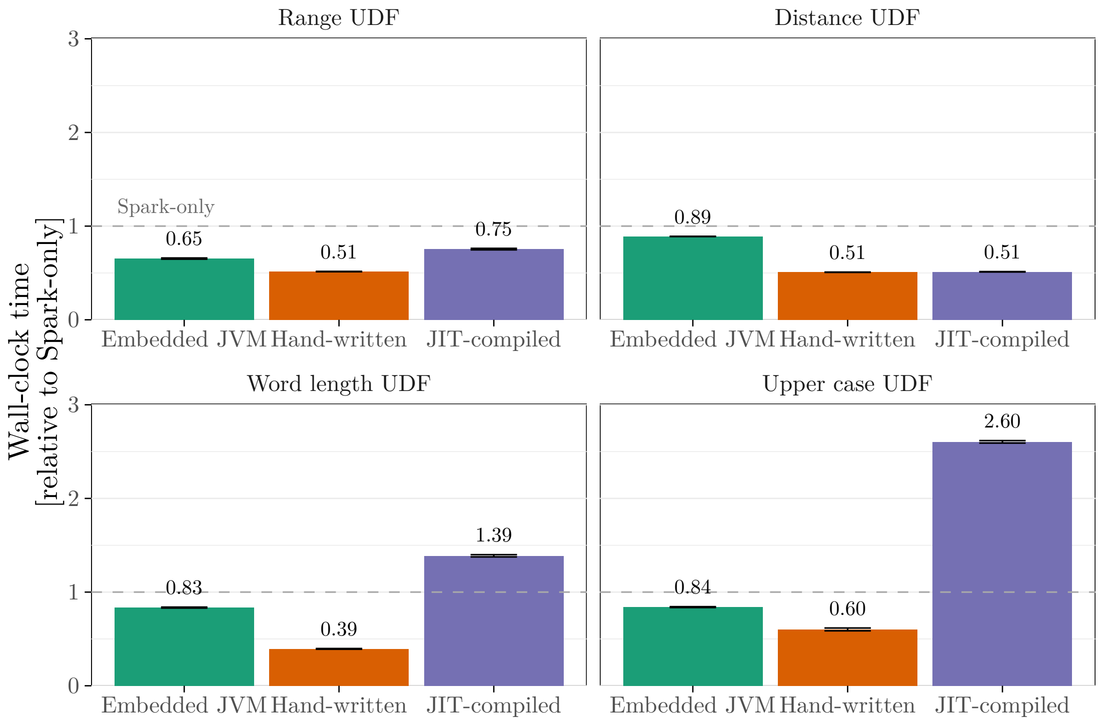


Figure 4.13.: Performance of Java UDFs that are JIT-compiled to machine code.

LLVM IR. Special JVM bytecode constructs, e.g., `invokevirtual` to call a polymorphic method, are implemented as functions for which BugVM provides the implementation. These special functions are inlined by the LLVM optimizer when appropriate to improve performance. The BugVM JVM uses the Boehm-Demers-Weiser garbage collector [31, 32] for memory management.

### 4.6.2. JIT-compiled UDF performance

Figure 4.13 shows the performance of UDFs that are JIT-compiled from Java bytecode to machine code, and compares them against strided execution inside an embedded JVM, as well as against versions of the UDF hand-written in C++ that are statically linked with the Wildfire engine. To allow comparison with the measurements reported in Section 4.5, we again normalize the wall-clock time to the execution time of the queries in Spark. We exclude the compilation time from our analysis because it is a one-time cost incurred when the UDF is registered with the engine. Note that each execution strategy

```
Input   : Input buffer input with strings.
Output: Output buffer output of string lengths.
1 for $i \leftarrow 1$ to size of input stride do
2     $javaString \leftarrow$ CreateJavaString($input_i$)
3     $output_i \leftarrow$ WordLengthUdf($javaString$)
4     CheckForJavaException()
5     ReleaseJavaObject($javaString$)
6 end
```

Algorithm 4.1: Word length UDF wrapper.

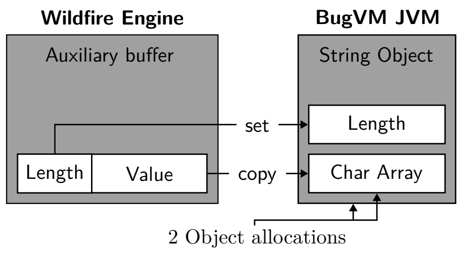
evaluates the UDF in a strided fashion, using the stride sizes reported in Section 4.5.1 for each UDF.

The results clearly indicate overheads in the embedded JVM approach, as the UDFs which are hand-written in C++ outperform the Java UDFs executed inside the JVM. The word length UDF shows the biggest difference, a reduction by a factor of 2.1. However, as we will see below, the hand-written version exploits knowledge about the specific task performed by the UDF, as well as the storage scheme used by Wildfire, to achieve this speedup. The JIT-compiled version of the computationally heavy distance UDF is as fast as the hand-written version and 1.75× faster than executing the UDF in an embedded JVM. In contrast, the hand-written versions of the range UDF and the upper case UDF are only marginally faster than executing the UDF inside the embedded JVM, whereas JIT-compiled versions are considerably slower. We discuss the reasons for this slowdown and possible remedies in the next section.

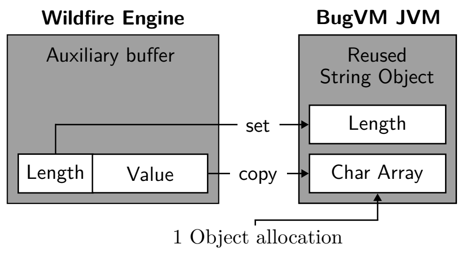
### 4.6.3. String construction optimizations

We now describe optimizations that we can apply to JIT-compiled machine code UDFs, using the word length UDF as a case study. The wrapper executing the UDF on a single stride is shown in Algorithm 4.1 in pseudocode.

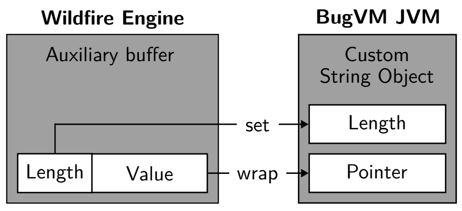
A few operations deserve scrutiny. First, for each input string, we need to construct a Java String object (line 2). In Java, each string is represented by its own immutable String object, which in turn contains a reference to a Java Char array. Thus, to create a Java string from a string in a Wildfire input buffer, a data copy is needed in addition to two object allocations, one for the String object and one for the referenced Char array, as shown in Figure 4.14a. Second, after the UDF has been called, we need to check if the UDF raised a Java exception (line 4). Finally, we have to explicitly release the Java string object that was allocated outside of the JVM, otherwise it cannot be reclaimed by the garbage collector (line 5).



(a) Construction of String objects.



(b) Reuse of String objects.



(c) Wrapping of engine buffer.

Figure 4.14.: Java String creation and optimizations.

The hand-written UDF is much simpler. Most conveniently, the length for each word is already stored in the auxiliary buffer, as shown in Figure 4.3 in Section 4.3. Thus, the Wildfire engine can evaluate the UDF without accessing the contents of the string at all. Wildfire also does not have to do any input-specific error handling. In other words, lines 2 to 5 are replaced by just copying the word length value from the auxiliary buffer to the output buffer.

In order to address the overheads in the generated wrapper, we investigate four optimizations: (1) Reusing String objects to reduce object construction; (2) Modifying the JVM's String objects to wrap data from the engine's input buffers to minimize data copies; (3) Relaxing the JVM's exception handling; and (4) Removing a memory fence normally required by the Java memory model.

#### 4.6.3.1. Eliminate String object construction

Since the word length UDF does not store a reference to the input string, we can reuse the existing String object between UDF calls and simply replace the internal Char array. This optimization saves one object allocation as shown in Figure 4.14b, and, therefore, puts less pressure on the garbage collector. Note that this optimization requires knowledge about the inner workings of the UDF and cannot be applied in general. In particular, if the UDF were to store (a reference to) the input string internally, the contents of the stored string would change in-between each UDF call, violating the immutability of Java strings.

#### 4.6.3.2. Eliminate data copies.

To eliminate the object allocation of the internal Char array, we can implement a custom String class that wraps a string stored in a memory region outside of the JVM by storing a pointer to it, as shown in Figure 4.14c. This also removes the need to copy the string contents from outside the JVM into it. However, we now have to take care of distinguishing String objects created inside Java and those that wrap a memory region outside of the JVM. The latter are owned by the engine and cannot be automatically reclaimed by the garbage collector.

#### 4.6.3.3. Relax exception handling.

In general, Java UDFs submitted by Spark can have side effects, such as changing global state or performing I/O. Consequently, we have to check for error conditions after each UDF invocation, otherwise side effects can continue to occur after an error, resulting in an inconsistent state of the application.

In Java, errors are typically signaled by raising an exception. If an exception was raised inside a JNI call, it will remain active until it is explicitly cleared. Therefore, for a general UDF, we have to check for the presence of an exception, take appropriate measures, and clear it. However, we know that the word length UDF is free of side effects. Furthermore, given our previous optimizations, we can also rule out an out-of-memory error since there are no more object allocations. Hence, we can move the exception handling code out of the loop.

#### 4.6.3.4. Remove memory fence.

The Java memory model guarantees that final fields of objects will be correctly initialized after an object is constructed [93, §17.5]. This guarantee, which restricts possible variable assignment reorderings by the compiler, applies to the internal Char array of the String object. To satisfy this guarantee in a multi-threaded environment, the compiler has to insert a memory fence instruction after the String object has been constructed. However, since the String object is not referenced by other code (and, therefore, also not by other threads), we can remove the memory fence from the generated code without violating this guarantee.

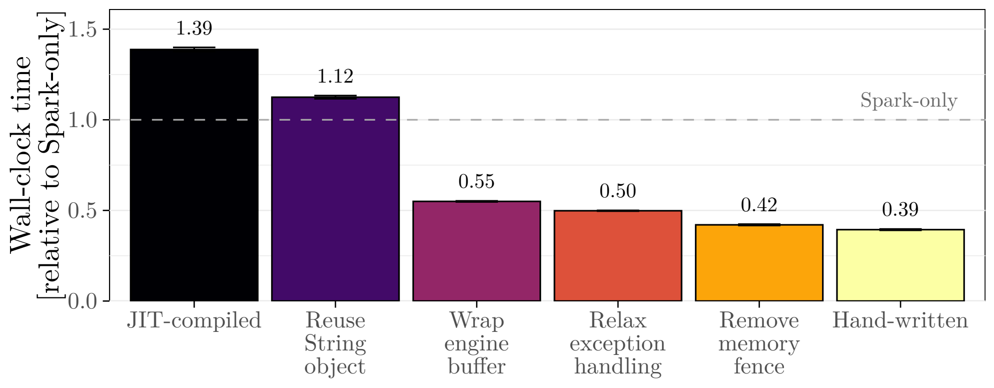


Figure 4.15.: Effect of optimizations on JIT-compiled code for the word length UDF.

### 4.6.4. Effect of optimizations

Figure 4.15 shows the effect of the optimizations described above, with the original JIT-compiled UDF without optimization and the UDF hand-written in C++ as a reference. The measured wall-clock times are again normalized to execution time in Spark. Each optimization is applied on top of the previous optimizations in the order they are mentioned above. The fully optimized UDF is almost as fast as the hand-written version. Eliminating object allocation and data copies has the biggest effect on performance. Further optimizations have diminishing returns.

While the exception handling code has a small influence on the performance of the word length query, for the range query it is precisely responsible for the gap between the hand-written and the JIT-compiled version of the query in Figure 4.13. In fact, if we eliminate this check, LLVM's JIT compiler produces exactly the same machine code as for the hand-written version, which is inlined and makes use of the processor's SIMD instructions. For the distance UDF, there is virtually no difference between the execution time of the JIT-compiled and hand-written version because the time required for exception handling is negligible compared to the time spent in the UDF itself.

As we have seen, when we apply all of the proposed optimizations to the JIT-compiled word length query, it is almost as fast as hand-written code. Unfortunately, we have broken Java in the process because we have changed core language semantics.

### 4.6.5. Applicability of optimizations

The Java language specification mandates that strings are immutable. Three of the optimizations violate this critical guarantee. The issue arises when a UDF *leaks* the string reference, i.e., if it stores the reference in a global variable as part of state it maintains across invocations. For regular Java strings, such leaking is unproblematic because strings are immutable. However, if we reuse the String object across UDF invocations and exchange the underlying Java Char array, the leaked string will change from one invocation to the next. Similarly, if we use a custom String object that wraps a memory region owned by the engine, the leaked string also changes when the engine modifies the memory region. Finally, if we remove the memory fence and the UDF passes the String reference to a different thread, this thread could see the uninitialized contents of the String object.

For the word length UDF, we can apply these optimizations safely since we know that no String reference is leaked. In general, these optimizations are safe if the UDF is free of side effects.

UDFs that keep state across invocations may lead to unpredictable behavior in Spark. There is no way to directly exchange state information between Spark executors. Furthermore, the result depends on how the Spark job is distributed across executors. The same applies to the embedded JVMs in the Wildfire engine nodes.

Even without knowledge of the inner workings of the UDF, it is possible to apply many optimizations with the help of static and dynamic code analysis. For example, by checking the reference count of the String object after the UDF has finished, we can verify that no String reference has been leaked and reuse the object for the next iteration.

### 4.6.6. Summary

As we have seen, computationally heavy UDFs that do not create objects can benefit from JIT-compilation to machine code, leading to a performance increase by a factor of 2.1. While the Java programming model limits many possible automatic optimizations related to object creation overheads, it is still possible to apply these optimizations through static and dynamic code analysis. Thus, it is not surprising that the Oracle HotSpot VM is quite good at dynamically optimizing Java code. Consequently, UDFs evaluated inside an embedded JVM in a strided fashion achieve a performance that is comparable to machine code for many use cases.

## 4.7. Related work

Most, if not all, popular database systems support UDFs to let users express an algorithm in an imperative way. Consequently, complex UDFs are prevalent in real-word database workloads [108]. However, common wisdom discourages the use of UDFs due to their slow execution speed, which is caused by the impedance mismatch between declarative

SQL queries and procedural UDFs. Whereas queries are optimized by a query optimizer, UDFs are treated as a black box, and, in the worst case, called for every tuple.

Foufoulas et al. [82] give an overview of recent work to speed up the execution of UDFs in database engines. They group the work into three categories: (1) systems that translate UDFs to SQL, (2) systems that lower relational operators and UDFs to a common IR, and (3) systems that embed an execution environment for the UDF into the database engine. Our work falls into the third category. In the following, we discuss the advantages and disadvantages of each approach, and how it relates to our work.

### 4.7.1. Translation to SQL

In this approach, a procedural UDF is translated into an equivalent SQL expression, and SQL queries with UDFs are rewritten as plain SQL queries. Note that the UDF itself can contain relational statements. The main advantage is that the rewritten query can now be fully optimized by the query optimizer, which has numerous benefits: (1) It eliminates context switches between relational execution and the UDF [72, 81, 259], (2) it avoids the materialization of the results of SQL query called from the UDF [107], and (3) it eliminates repeated calls to the query planner and optimizer, if a SQL query is used multiple times inside a UDF, or if the UDF calls itself recursively [50, 71]. Furthermore, the techniques typically do not depend on the specific UDF language [107, 259] and can be applied to any application that embeds SQL queries [81, 107]. However, translation to SQL is typically restricted to a pure subset of the UDF language [29, 81, 259, 353] or to a limited set of third-party libraries [109, 278].

Froid [259] translates scalar UDFs written in T-SQL into nested subqueries, and is integrated with Microsoft SQL Server. It supports branches and recursion but does not support loops. The authors observe that for complex recursive or nested UDFs, the translated SQL query can become too large for the optimizer to handle efficiently; therefore, they reduce the maximal depth of the generated algebraic tree. Aggify [107], which is also prototyped on Microsoft SQL Server, translates CURSOR loops into equivalent custom aggregates. The CURSOR can either be created inside a T-SQL UDF or in a Java application, i.e., a JDBC ResultSet.

Duta et al. and Burghardt et al. translate PL/pgSQL UDFs with iterative [72] and recursive [50, 71] control flow into a single SQL:1999 query with recursive common table expressions (CTE). ByePy [81] uses the same approach to translate Python functions with embedded SQL queries to SQL:1999 queries with recursive CTEs. ByePy supports complex control flow (but no recursion) and a limited set of built-in Python types and functions.

Grizzly [109] translates Python pandas [212] code to SQL queries with nested subqueries. However, Python UDFs called by pandas code are shipped as is to the database and still require the invocation of a Python interpreter. Schüle et al. [278] translate Python pandas [212] and scikit-learn [245] code to SQL but do not support UDFs.

CLIS [353] translates Spark SQL queries containing Scala UDFs to plain Spark SQL. Notably, it supports the capture of (immutable) free variables inside the UDF in the same way as we do. However, it does not support arbitrary Scala code, e.g., reflection, virtual dispatch, or user-defined types; or functionality without a built-in Spark SQL equivalent, e.g., matrix multiplication or encryption.

### 4.7.2. Common intermediate representation

In this approach, relational operations, user-defined code [102] or other data processing libraries [238, 239] are mapped internally to a unified internal representation (IR). A query compilation framework can then apply optimization steps, across operator boundaries e.g., loop fusion, loop unrolling, function call inlining, vectorization, and common subexpressions elimination [102, 238]. Furthermore, it can specialize data access to eliminate data conversion or copies between operations [77, 102]. However, this approach is limited to the extend that UDF language features or third-party libraries are mapped to the common IR [102, 238, 239].

Flare [77] is query compilation backend for Spark SQL which targets scale-up machines, with the goal of retaining Spark SQL as a popular data analysis front end while improving the integration other data analysis frameworks, e.g., TensorFlow [5]. Flare employs lightweight modular staging [265] in Scala to specialize the Spark SQL query and generate a C program, which is then compiled and executed. When calling TensorFlow UDFs, Flare specializes its internal data structures in a way that eliminates data layout modifications or copies. Scala UDFs in Spark SQL queries need to be modified so that they will also be specialized during query compilation However, this is a straightforward process that can be done mechanically.

Weld [238, 239] is a common runtime for data analytics frameworks and libraries. To integrate a system with Weld, its operations have to be reimplemented to generate Weld IR to construct a common operator graph. The Weld runtime then applies query compilation optimizations across the entire graph and generates an executable program via LLVM [174]. Furthermore, the Weld runtime supports adaptive execution to switch between branched or branch-free selection and independent or shared aggregation [238]. Weld is integrated with Spark SQL and speeds up execution of Scala UDFs in Spark SQL queries by vectorizing UDF calls and eliminating data conversions [239].

Babelfish [102] combines relational operators written in Java with polyglot UDFs and executes them on the GraalVM [233]. It relies on Truffle [337] to support UDF languages such as Python or JavaScript, in addition to Java UDFs. Babelfish specializes the generated code, inlines function calls, and rewrites data access operations to eliminate data conversion and data copies between relational operators and UDFs.

### 4.7.3. Embedded execution environment

In this approach, the database engine includes a fully featured execution environment for the UDF language. Both of the techniques we describe in the Chapter fall into this category; we either embed a standard JVM in Wildfire or we link the machine code generated from the UDF against the specialized BugVM JVM [49]. The main advantage is that the UDF can use all of the functionality of the UDF language, including third-party libraries [159]. However, switching between relational execution and UDF execution incurs a large overhead, especially for scalar UDFs applied to every tuple. To mitigate this overhead, the database engine can vectorize the execution of scalar UDFs [83, 159, 257] or fuse multiple UDFs together [83]. The database engine also has to copy and/or convert data between its internal representation and that of the UDF language. Columnar database engines can sidestep in this issue, if the UDF is able to process C-style arrays; in this case, only a small amount of metadata has to converted [64, 83, 257]. Alternatively, both the database engine and the UDF can use a common data representation, e.g., Apache Arrow [306]. Finally, a buggy or malicious UDF can crash the database server. To protect against this issue, a database engine can preprocess the UDF code [159], execute the UDF in a separate process [159], or even inside a containerized environment [276].

AIDA [64] and Raasveldt et al. [257] integrate MonetDB [37] with NumPy [111]. Since NumPy is an array-based programming framework, the UDF is automatically executed in a vectorized fashion and there is no data conversion.

YeSQL [83] integrates Python UDFs with MonetDB or SQLite [296]. YeSQL applies many of the same techniques that we use to improve scalar UDF execution. (1) It automatically generates a vectorized execution wrapper; (2) it compiles the generated code with PyPy [36], a tracing JIT compiler; and (3) it passes strings as memory buffers to UDFs that support this representation, without copying data. Furthermore, YeSQL fuses nested scalar UDFs, which enables PyPy to optimize longer execution traces.

Impala [165] executes Hive UDFs written in Java inside an embedded JVM one tuple at a time. The Impala manual recommends C++ UDFs for performance reasons, stating that C++ UDFs are often 10× faster than equivalent Java UDFs [315]. To reduce redundant work across multiple invocations, Impala allows the UDF to store invariant data in a thread-safe memory area. In our implementation, a UDF can trivially replicate this optimization by storing state inside the UDF instance.

## 4.8. Conclusion

In this chapter, we analyzed the overheads of executing Java UDFs within a database engine written in native code, and which does not use the JVM. Contrary to common belief, we find that using an embedded JVM does not necessarily lead to disastrous performance. As expected, a strided execution pattern effectively minimizes call overheads commonly associated with JNI calls, and the strided execution wrapper can be generated automatically by the engine, making the entire process transparent to users. The necessity of strided execution diminishes as the computational load of the UDF increases.

By using Java direct byte buffers, we can pass entire data blocks from the engine to the embedded JVM. If the UDF uses primitive types in its interface, the data encapsulated in these buffers can be accessed directly without incurring an additional copy. However, Java objects in the UDF interface require the data to be copied from input buffers to the Java object's data structures, or from the Java object to output buffers, which dominates UDF execution. Consequently, primitive types should be used whenever possible.

We also show that running UDFs inside an embedded JVM compares well with UDFs hand-written in C++. JIT-compiling computationally heavy UDFs, that do not use Java objects, to machine code, can improve performance by about a factor of two. Other UDFs do not benefit from JIT-compilation to machine code because of the overheads associated with the guarantees of the Java language, namely the immutability of Strings, exception handling, and a particular memory model.

We were able to apply a number of optimization to the generated machine code because our evaluated UDFs were free of side effects. We would argue that it is good practice to use side effect-free functions for scalar UDFs in any case. However, we also advocate to integrate these optimizations inside an existing JVM's JIT compiler instead of essentially duplicating the HotSpot VM.

# 5 Conclusion and research outlook

In this thesis, we investigated how heterogeneous hardware and software impacts query processing. The heterogeneous nature of today's computing systems is a direct consequence of efforts to continue to improve computing performance in the face of restrictive physical constraints, and to provide data analysis tools to a wide range of users with diverse requirements. Thus, this heterogeneity represents a challenge to query processing systems, but also an opportunity to continue to improve performance and democratize data analysis. Since a thesis can only cover a selection of problems in this very broad research area, we focused on three specific scenarios, which cover both hardware and software heterogeneity.

## 5.1. Hardware heterogeneity

In Chapter 2, we conducted a survey of query processing systems which utilize both CPUs and GPUs for query processing. We developed a classification scheme to categorize how these systems distribute query processing tasks each processor, and reviewed techniques to reduce the implementation complexity of such systems and to mitigate the data transfer bottleneck. Our study revealed a key difference between computing systems with dedicated GPUs, and those in which CPU and GPU compute units are integrated on the same chip. Systems with a dedicated GPU, when it is connected over slow system bus, are constrained by the data transfer bottleneck. Thus data transfers should be avoided as much as possible, which can be achieved by offloading specialized coarse-grained tasks to the GPU, which execute independently of the tasks assigned to the CPU. In contrast, on systems with integrated GPUs, both CPU and GPU compute units can exchange data frequently and access shared data structures simultaneously. A query processing system can thus divide the workload into more fine-grained cooperative tasks and has more opportunities to match the characteristics of each task with the capabilities of the CPU and GPU compute units.

However, the recent emergence of fast, coherent interconnects, such as NVLink 2.0 [225], Infinity Fabric [241], or Compute Express Link [284] changes the status quo. These fast interconnects reduce the difference between accessing data in main memory from the CPU and from a dedicated GPU, and also allow both processors to access shared data structures simultaneously, thus likely moving the trade-off from coarse-grained, independent tasks to fine-grained, cooperative tasks. Lutz et al. have studied the effect of NVLink 2.0 on hash joins [189, 190] and showed that the main bottleneck is shifted from the interconnect to other resources. Consequently, the effects of fast interconnects on other query processing tasks, e.g., aggregations, indexing, or transactional processing, present a large and likely fruitful open research area. Furthermore, simultaneous access to shared data structures depends on atomic operations. We believe it is worthwhile to model the performance of atomics based on workload characteristics, and while we have explored this topic in a Master's thesis [279], it remains an open research question.

Modern GPUs also increasingly incorporate specialized hardware for matrix operations [30, 225], which are ideal candidates to speed up machine learning tasks in complex data analysis pipelines. A heterogeneous query processing system has to take data movement into account when assigning machine learning and other query processing tasks to the GPU, which is an application scenario that we did not include in our survey. Alternatively, matrix cores on GPUs can be used to estimate data distributions and correlations with machine learning-based approaches [171]. We expect the capability of the GPU to process larger models in a shorter amount of time to lead to an improvement of the query optimizer, in line with previous research [122, 156].

With regard to application scenarios, we noticed that most research into query processing on GPUs focuses on dedicated GPUs at the expense of integrated GPUs; probably, because their raw processing power makes them an enticing research target. However, we would argue that integrated GPUs are becoming increasingly important, due to their prevalence in consumer hardware [21], power-constrained mobile devices [255], and embedded applications [290]. To the best of our knowledge, a number of important application scenarios, e.g., hybrid transactional/analytical processing or spatio-temporal workloads, have not yet been studied on integrated GPUs. Since, as we noted above, integrated GPUs require a different approach to heterogeneous query processing, the application design to handle these workloads on integrated GPUs remains an open research question.

In Chapter 3, we focused on a specific aspect of a heterogeneous query processing system, its low-level operator implementation. Our goal was to let the system automatically adapt its operator implementation to the processor it runs on, instead of having to fine-tune it manually to multiple target processors. To this end, we performed an extensive performance analysis of two common data processing operators, selection and hash aggregation, on various multi-core CPUs, GPUs, and an Intel Xeon Phi processor. A key insight of our work is that GPUs (and also the Xeon Phi) are very sensitive to specific implementation parameters; so much so that previously derived heuristics from a single GPU micro architecture do not yield good results on other GPUs. Even on GPUs created by the same manufacturer, the optimal implementation parameters strongly depend on the particular micro architecture, and wrongly optimized implementations incur large performance penalties. Based on this analysis, we developed two algorithms to learn fast operator implementations at runtime. Here, our main insight is that it is essential to restrict the search space by incorporating prior knowledge about the target processor, in order to efficiently find a fast operator implementation.

At the end of Chapter 3, we sketched how we would integrate our algorithms into a complete query compressing system which adapts its operator implementation to different target processors. Building such a system presents many interesting research challenges. One open task is to formulate a set of micro benchmarks to characterize relevant processor properties, in order to restrict the search space without relying on an extensive analysis, as we have done. Another open question, which we started to investigate in a Master's thesis [169], is how to use the data we gain during the evaluation of various operator implementations during query processing, to build or a refine a model about the behavior of the processor.

The low-level operator implementation is only one aspect of the implementation of a query processing system; another is the query processing model. Whereas the relative benefits of vectorization and query compilation have been investigated on CPUs [151, 294], there is so far, to our knowledge, no systematic comparison between these processing models on GPUs. We expect that the relative advantages of these processing models depend on the type of GPU integration, since query compilation creates coarse-grained tasks that are beneficial to on dedicated GPUs, whereas vectorization allows for more fine-grained tasks that target integrated GPUs. Furthermore, every heterogeneous relational query processor that we studied uses the same query processing model on both CPUs and GPUs, and also executes a physical query plan that was created by an optimizer targeting CPUs. Consequently, it is an open question how to adapt the processing model and query plans to different processors in a heterogeneous CPU/GPU system. The main challenge in such a query processing system is to keep its complexity in check.

## 5.2. Software heterogeneity

In Chapter 4, we focused on integrating two software systems, with the overarching goal of keeping a familiar query interface for users, while improving the speed and capabilities of the system. Specifically, we investigated how to execute SparkSQL Java UDFs [20] inside the C++ engine of Wildfire [26], a hybrid transactional/analytical processing engine. Contrary to our initial expectations, executing the UDFs inside an embedded JVM showed very good performance; it was faster than executing the UDFs in Spark alone, and competitive to equivalent hand-written C++ code. By calling the UDFs in a strided fashion, we eliminated call overheads between C++ code and the JVM; and by employing Java direct byte buffers, we eliminated copies of primitive types. We also investigated compiling Java UDFs directly to machine code but found that this approach is often slower than running the UDFs inside an embedded JVM, especially, if they create Java objects, such as Strings. Statically compiling Java UDFs to machine code foregoes the runtime optimizations of the Java HotSpot VM, such as method inlining and monomorphic dispatch [78]; it is easier to execute UDFs inside an embedded JVM than to replicate these optimizations in a query processing system.

Since our work, the research community has formulated new approaches to combine Java UDFs with native code written in C++. For example, TornadoVM [86] transpiles Java bytecode to OpenCL [300], which enables the exciting capability [340] of accelerating Flink UDFs [51] on multi-core CPUs, GPUs, or FPGAs. Babelfish [102] takes the opposite approach and utilizes Truffle [337] to translate polyglot queries, in which a relational operators written in Java are combined with UDFs written in Python, JavaScript, or other languages, to a common intermediate representation, which is then executed on GraalVM [233]. However, the seamless integration of UDFs in different languages, which can also be accelerated on GPUs, remains an open problem. TornadoVM still requires data marshaling, including padding and endianness conversion [340]. And while it is possible to integrate Babelfish with a query compilation engine, calling precompiled functionality provided by the query engine from an UDF, e.g., a specialized data structure implementation, still presents an optimization boundary.

## 5.3. Common challenges

We end this thesis by pointing out the similarities when dealing with heterogeneous hardware and software. At first glance, the task of distributing computation on multiple heterogeneous processors appears to be very different from integrating various software tools in a complex data analysis ecosystem. However, in both cases we encounter similar challenges. For example, just as we have to match query processing tasks to the different capabilities of CPUs and GPUs (Chapter 2), we have to assign tasks to an appropriate data processing platform in cross-platform data processing [10, 143]; often, we use cost models to do so. We also have to be aware of the cost of data movement and aim to reduce it, e.g., when exchanging data between different software environments (Chapter 4), between different processors (Chapter 2), or between different systems over the network [91]. Finally, in both cases there is a need for declarative programming abstractions; e.g., just as OpenMP [65, 179] and OpenACC [319] allow programmers to declaratively offload parts of a C++ program to GPUs or other accelerators, Emma [14] allows users to write declarative data analysis programs that run on Spark [313] or Flink [51]. Thus, techniques to address the heterogeneity of today's computing systems, are applicable in a wide range of research areas and engineering tasks in query processing. In this thesis, we have developed new techniques, as well as systematically categorized existing approaches, to let query processing system exploit this heterogeneity, in order to extract high performance and support a diverse user base.

# A

# Descriptions of surveyed CPU/GPU query processing systems

In this appendix, we discuss the scheduling decisions of selected heterogeneous query processing systems, which we surveyed in Chapter 2.

## GDB

GDB [116] is the first GPU-based relational query processor described in literature. It implements relational operators on the GPU using previously described data-parallel primitives [115, 118, 281]. In a follow-up work, these primitives were revised to support extended precision [186]. The GPU-based operators are combined with a previously described cache-oblivious CPU implementation [117] to create a heterogeneous query processor.

Thus, GDB can execute the operators of a query plan on either processor. It can also execute operators on both processors by partitioning the data. During query optimization, GDB decides for each operator where to execute it and how to partition the data based on processor-specific cost-models. For the CPU, it uses the generic database cost model by Manegold et al. [191]. For the GPU, it uses a custom cost model which treats primitives as black boxes and incorporates computation time, memory stalls, and data transfer time based on microbenchmarks. GDB handles limited device memory by partitioning the input data and out-of-core processing. It does not support overlapping computation with transfer. Follow-up work added compression to the system [80].

Using CPU/GPU coprocessing, the authors found a moderate speedup of up to 20% for TPC-H queries with scale factor 1. For larger scale factors, overall system performance was dominated by disk I/O.

## Approximation and refinement

Pirk et al. propose the *Approximate & Refine* processing model for heterogeneous CPU/GPU systems [247]. In this processing model, attributes are bitwise decomposed and the partitions are stored non-redundantly on different devices. The major bits of each value are stored in GPU memory and the residual minor bits are stored in CPU main memory. In essence, the GPU stores a lossily compressed version of the data on which an approximate query result can be calculated. Together with the residuals, this approximate result can be refined on the CPU to produce an exact result. Consequently each relational operator is expressed as a pair of an approximation operator that runs on the GPU and a refinement operator that runs on the CPU.

This processing model fits the strengths and weaknesses of a heterogeneous CPU/GPU system with a dedicated GPU very well. On the one hand, the approximation on the GPU effectively acts as a filter on the data that has to be processed on the CPU to produce an exact result. It benefits from high computational power and memory bandwidth of the GPU. On the other hand, the limited size of GPU memory and the slow system bus are not bottlenecks in this approach. First, by adapting the number of bits stored on the GPU, the approach can handle data sets that greatly exceed GPU memory. The authors show that predicate selectivity has a much bigger influence on query performance than the number of bits stored on the GPU. Second, input data only has to be transferred during the partitioning phase but not for each query. During query execution, only approximate results have to be transferred over the system bus. (The authors do not discuss updates.) Furthermore, the approximation and refinement phases of different relational operators can be interleaved. Therefore, both phases can largely run in parallel on the GPU and CPU.

In their evaluation, the authors achieve a speedup of 3.9× for spatial queries and between 1.7× and 6.5× for relational queries compared to MonetDB [37]. They also note that the CPU is often underutilized during query processing and available for other tasks.

## CoGaDB

CoGaDB [42] is an in-memory relational database that supports multi-core CPUs, dedicated and integrated GPUs, and other heterogeneous processors. It also supports multiple processing models, e.g., operator-at-a-time execution [42] as well as query compilation [46].

In its original design [42], CoGaDB performs static scheduling based on black-box cost models in combination with processor utilization. The scheduler integrates its decisions into the compilation of a physical query plan. For each operator, it determines both the algorithm to implement the operator as well as the processor to run the algorithm. To achieve a balanced execution, the scheduler takes into account the estimated runtime of the operators that are already scheduled on each processor.

In a follow-up work [44], the authors implement a data-driven scheduling heuristic to reduce data transfers. The system analyzes the workload in the background and copies frequently accessed columns to the GPU. The scheduler subsequently schedules operations on the GPU only if all its inputs are present in GPU memory. Accordingly, these schedules do not contain on-demand transfers of input data to the GPU. Intermediate results are transferred only when execution moves from the GPU to the CPU.

The authors show that data-driven operator placement in combination with dynamic scheduling gracefully handles use cases when the working set does not fit into the limited device memory.

## He et al. [120, 121]

He et al. [120, 121] describe a relational query processing system optimized for integrated GPUs, which is based on the close cooperation of the CPU and the GPU through fine-grained coprocessing. The authors break down complex data processing operations, e.g., hash joins, into primitive processing steps and measure the throughput of each processing step on the CPU and the GPU. Based on these measurements, a cost model determines task-specific data partitions for each processing step in order to statically schedule tasks on the most suitable processor without leaving processing resources idle [120]. Intermediate results of each processing step are materialized in the processor cache if the next step in the processing pipeline executes on a different processing core. In a follow-up work [121], the authors also incorporate decompression and prefetching steps in order to improve the effective memory bandwidth and reduce stall times on the GPU. Whereas the decompression and relational query processing steps can be scheduled on either the CPU or the GPU, the data prefetching step always runs on the CPU.

The authors first evaluate the performance of the system when executing hash joins on an AMD Llano processor [41]. They show that the data partition ratios for each step vary greatly and that a workload distribution which divides the data into task-specific partitions for each step is 28% faster than a single partitioning of the data for all steps [120]. In a follow-up work, they evaluate the system on a subset of TPC-H queries [323] on an AMD Kaveri processor [40], showing that decompression and CPU-based prefetching can further improve performance by up to 40% compared to a task-specific data partitioning alone [121].

## Statistical coprocessor

Heimel et al. propose to use the GPU as a *statistical coprocessor* during query optimization [122]. Specifically, the database maintains a sample of the tables on the GPU and computes a kernel density estimator (KDE) on this sample to predict the selectivity of a query with multiple predicates. Kiefer et al. adapt this approach to also estimate join selectivities over multiple base tables [156].

This approach fits the strengths and weaknesses of a heterogeneous CPU/GPU system with a dedicated GPU very well. On the one hand, the computation of the KDE model is compute-intensive but embarrassingly parallel. By offloading this computation to the GPU, a more involved model can be processed in a fixed time budget, which improves the selectivity estimate and indirectly the query performance. On the other hand, the slow system bus to the GPU is not a bottleneck in this approach. Only the query predicates, the computed estimates, and new rows for sample maintenance have to be transferred. Furthermore, the separation of query optimization and query processing on distinct processors has additional useful consequences. For example, once the predictions have been communicated to the CPU, the CPU can proceed with query processing while the GPU performs sample maintenance operations in parallel.

## Mega-KV

Mega-KV [356] is a GPU-assisted in-memory key-value store (IMKV). The authors profile a state-of-the-art CPU-based IMKV and identify index operations as the main bottleneck. Depending on the size of key-value pairs, these operations take up to 75% of the total request time. Because the large index does not fit into the CPU cache, the CPU often stalls on random main memory accesses. However, index operations fit the GPU architecture well. They consist of simple computations and are highly data parallel. Therefore, the authors propose to perform these operations on the GPU.

Mega-KV stores actual key-value pairs in CPU main memory. The GPU index maps lossily compressed key signatures to the location of the key-value pair. Depending on the average size of a key-value pair, the limited size of GPU device memory can index orders of magnitude more data in main memory. Storing compressed key signatures and value locations instead of full key-value pairs also mitigates the data transfer bottleneck. By assigning a dedicated GPU to a specific logical partition of the data, Mega-KV scales well with additional GPUs. The CPU in Mega-KV performs preprocessing tasks, e.g., request parsing and key signature compression, as well as post-processing tasks, e.g., key-value pair lookup and response construction. The system executes a pipeline in which CPU tasks, GPU tasks, and data transfers between the two processors are efficiently overlapped. The authors report a speedup of 1.4× to 2.8× over a state-of-the-art CPU-based IMKV.

## SABER

SABER [163] is a window-based SQL stream processor. It is written in Java but also includes OpenCL-based operator implementations. Therefore, it can run operators either on the CPU or on the GPU.

SABER uses a dynamic scheduling scheme that combines a global queue with a black-box cost model based on historical performance. As the system receives input data, it is split into fixed-size batches. Each batch is bundled with a query-specific function to create a *query task*. These query tasks make up the basic scheduling unit and are placed into a single global queue. SABER schedules each query task on the processor that achieves the highest throughput according to its cost model. However, if this preferred processor is busy executing other tasks, SABER schedules the query task on another processor if it can finish the task before the originally preferred processor becomes available. This way, SABER generally schedules query tasks on the most suitable processor but avoids underutilization.

The authors show that hybrid execution always yields better performance than executing queries on a single processor. However, the combined throughput is less than the sum of the throughputs on individual processors due to contention when dispatching tasks and assembling results. They report that SABER is up to 6× faster than Spark streaming [346].

## DB2 with BLU acceleration

DB2 with BLU acceleration can offload sorting and grouped aggregation operations to the GPU [201]. It schedules operations based on GPU availability and heuristics. These heuristics take into account the number of input tuples, as well as the number of aggregation functions and the expected number of groups for aggregation queries. For example, if the input size is small, it is faster to execute operations on the CPU because the data transfer dominates execution time.

The authors show that offloading sorting and aggregation to the GPU can speed up complex queries by up to 20%. However, offloading operations to the GPU also frees up the CPU to perform other tasks. Thus, the system can process a complex workload consisting of multiple queries in half the time required by a CPU-only implementation.

## Caldera

Caldera [19] is an HTAP system for heterogeneous CPU/GPU systems. The design of Caldera is based on two observations. (1) The different characteristics of OLTP and OLAP workloads are aligned with the processor properties of CPU and GPUs. (2) Emerging heterogeneous many-core systems do not provide system-wide cache coherence across all processing cores. The authors introduce the *archipelago* abstraction and partition the system into a task-parallel archipelago consisting of CPU cores that execute latency-critical OLTP queries, and a data-parallel archipelago consisting of GPUs that that execute data-intensive OLAP queries. Caldera therefore uses a static schedule based on the nature of the two high-level tasks. (The authors also propose a hybrid design in which CPU cores dynamically migrate between the task-parallel and the data-parallel archipelago but they do not implement this design in the paper.) To address the lack of cache coherence, Caldera implements a message passing protocol to ensure that the results of transactions are visible to all cores in the task-parallel archipelago. Furthermore, Caldera employs software-assisted copy-on-write to ensure that OLAP queries operate on fresh data.

The authors evaluate Caldera using a custom workload consisting of TPC-H query 6 for OLAP and a read-modify-write query on random records for OLTP. They observe that even though the OLAP and OLTP queries run on different processors, they interfere with each other because the software-assisted copy-on-write competes for main memory bandwidth with OLAP queries. This interference can be reduced by limiting the percentage of hot data that is processed by OLTP queries and by reducing the freshness of the data for OLAP queries. The authors also evaluate the impact of the data layout on OLAP performance and compare the n-ary storage model (NSM) with the decomposition storage model (DSM) [61] and PAX [12]. On newer NVIDIA Maxwell GPUs [221], DSM and PAX show the same performance. However, NSM is 14× slower when data is accessed from CPU main memory but only 2× slower when data is GPU-resident, indicating that non-sequential reads over PCIe 3.0 are a bigger bottleneck than non-coalesced reads in GPU device memory.

## DIDO

DIDO [355] is an in-memory key-value store that is optimized for integrated GPUs. The authors first evaluate the fixed three-stage pipeline of Mega-KV [356] which is optimized for dedicated GPUs (see Mega-KV). They find that, depending on the workload, the pipeline is often unbalanced and processor utilization is low.

To improve processor utilization, the coarse-grained pipeline of Mega-KV is split into fine-grained tasks. DIDO dynamically assigns these fine-grained tasks to different pipeline stages on the CPU or the GPU. DIDO also uses separate pipelines for search queries, inserts and deletes because the different workload characteristics interfere with each other. These assignment decisions are driven by a white-box cost model. The computation cost is derived from the number of instructions of different tasks and theoretical peak IPC of each processor. The memory cost is derived from the expected number of memory and cache accesses based on the workload. To further improve processor utilization, DIDO also lets an idle processors opportunistically steal tasks that were assigned to the other processor.

The authors show that the dynamic pipeline is up to 4× faster than the fixed pipeline of Mega-KV on an integrated GPU, especially for small key-value pairs. DIDO cannot match the raw throughput of Mega-KV running on a dedicated GPU but has a better price-performance ratio.

## HEterogeneous Resource Optimizer (HERO)

The HEterogeneous Resource Optimizer (HERO) [145] is a virtualization layer to perform operator placement on multiple processors in databases. It is implemented as an OpenCL driver which manages all OpenCL-capable processors in a heterogeneous CPU/GPU system. To a database, HERO presents itself as a single virtual processor.

However, the OpenCL code that is executed on this virtual processor is transparently scheduled on all of the actual processors in the system.

HERO uses a hybrid scheduling scheme that combines static and dynamic scheduling. It estimates operator execution time using a cost model based on historical runtimes [147]. To overcome the impact of incorrect cardinality estimates, HERO splits the query execution plan into disjoint phases. Each phase ends with an operation which produces an intermediate result of unknown cardinality, e.g., a selection or a join. Therefore, the placement of the operators of a phase is triggered at runtime just before the phase is executed and the cardinalities of all inputs are known. The remaining operations of a phase produce intermediate results with known cardinality, e.g., projections. Therefore, the placement of all of the operators that make up a phase is decided statically before the phase is executed. To efficiently schedule the operators of an individual phase, HERO performs a greedy optimization with multiple restarts to avoid getting stuck in a local optimum [146].

The authors use HERO to integrate heterogeneous scheduling in Ocelot [123] and gpudb [343]. They show that heterogeneous execution on multiple processors is up to 2× faster than executing a query on the fastest single processor [147]. Since HERO does not support concurrent execution on multiple processors, this speedup is based solely on an improved operator placement.

## HetExchange

HetExchange [54] is a framework to encapsulate parallelism across multiple heterogeneous processors in a physical relational query plan. It is modeled after the Exchange operator [97] which encapsulates parallelism in the classic Volcano processing model. The authors identify four interesting properties of the data flow represented by a query plan that are related to parallel processing on heterogeneous CPU/GPU systems: the target device, the data locality, the degree of parallelism, and the data packing. They then introduce four corresponding operators which modify one of these properties. Handling each property by an individual operator produces three desired effects.

First, the operators remain oblivious of the other properties. For example, the *router* operator parallelizes execution across multiple processing CPU cores and GPUs but it is oblivious to the nature of these processors. The *device crossing* operator moves execution from the CPU to the GPU or back. By combining both operators, a query plan can parallelize execution across multiple CPUs and GPUs. Similarly, by encapsulating data transfer into the *mem-move* operator, other operators remain oblivious of the data location.

Second, traditional relational operators, e.g., joins, also remain oblivious to the hardware heterogeneity. To specialize execution on CPUs and GPUs, HetExchange fuses multiple operators into hardware-agnostic query pipelines which are then JIT-compiled to hardware-specific code.

Third, due to their restricted nature, the HetExchange operators can be easily integrated with existing query optimizers which reason over data flow properties. In the current implementation, the heterogeneous query plan is generated statically based on heuristics. During query execution, the router operators distribute data based on processor utilization.

The authors evaluate HetExchange on the Star Schema Benchmark [230] with scale factor 1000. They show that, on average, heterogeneous execution on multiple CPU sockets and GPUs achieves 88.5% of the sum of the throughputs of CPU-only and GPU-only execution.

## Raza et al. [261]

Raza et al. [261] describe a heterogeneous HTAP system that combines an OLAP engine based on HetExchange [54] with a custom OLTP engine called Aeolus. The design is similar to that of Caldera [19] (see Caldera) and runs the OLAP engine exclusively on the GPU and the OLTP engine exclusively on the CPU. The system therefore uses a static schedule based on the nature of the two high-level tasks. In contrast to Caldera, the system does not use copy-on-write to ensure that OLAP queries operate on fresh data. Instead, OLTP updates are collected in a delta log and applied periodically to OLAP data.

The authors evaluate the system with the CH-benchmark [60], which combines TPC-C [322] as the transactional workload and TPC-H [323] as the analytical workload, with a scale factor of 100. They observe that the segregation of the workloads on different processors results in a 2-7% reduction of OLTP throughput. The sequential access of OLAP queries does not starve the random accesses of the OLTP workload because the transfer over the system bus only consumes 16% of the DRAM bandwidth in case of PCIe 3.0 and 50% of the DRAM bandwidth in case of NVLink 2.0. The interference of OLTP queries on the performance of the OLAP workload depends on type of system bus and the strategy employed to overlap data transfers with computation on the GPU. Concurrent OLTP queries result in virtually no reduction in OLAP performance when data is pushed to the GPU in a software-managed pipeline over PCIe 3.0. However, when data is pulled by the GPU using zero-copy, concurrent OLTP queries reduce OLAP performance by up to 2×. When data is transferred over NVLink 2.0, queries are up to 30% slower, regardless of the transfer strategy.

## FineStream

FineStream [352] is a window-based SQL stream processor targeting integrated GPUs. It is written in OpenCL and can run operators either on the CPU or GPU compute units. FineStream groups operators into pipeline stages and assigns a variable number of compute units to each stage to maximize the bandwidth utilization of CPU and GPU compute units. This assignment is based on a black-box cost model based on historical data. Crucially, FineStream computes multiple candidate plans to group operators and assign them to compute units. During query execution, FineStream continuously monitors the stream ingestion rate, the size of intermediate results, and the overall performance. If these metrics change, FineStream can either assign more or fewer compute units to an operator group or switch to an entirely different plan.

The authors also reimplement SABER [163], which assigns entire queries to either a CPU or a dedicated GPU, and compare their implementation with FineStream on an integrated GPU. They report that a fine-grained assignment of operators to the compute units of an integrated GPU improves throughput by 52% and latency by 36% on average.

# B Performance penalties of incorrectly optimized AGGREGATE kernels

In Table B.1 on the following pages, we provide additional data for the experiments performed in Section 3.4. For each GPU and group cardinality, we list the optimal execution parameters of the AGGREGATE kernel. We also show the performance penalty when executing the AGGREGATE kernel with these execution parameters on the other five GPUs. This data is summarized in Figure 3.7 on page 98.

Table B.1.: Performance penalty of Aggregate kernels optimized for a specific GPU and group cardinality, when executed on other GPUs. WG/CU = work groups per compute unit, WGS = work group size. For each GPU, the worst performance penalty, which corresponds to the value shown in Figure 3.7a on page 98, is highlighted in **bold**. (Table continues on next page.)

(a) Optimized for AMD A10-7850K

| Groups | Parallelization strategy | WG/CU | WGS | Threads | Radeon R9 Fury | Tesla K40m | GeForce GTX 980 | GeForce GTX 1080 | Tesla V100 |
|---|---|---|---|---|---|---|---|---|---|
| $2^{0}$ | WorkGroupLocal | 256 | 128 | 32768 | 1.25 | **4.5** | 1.12 | 1.02 | 1.83 |
| $2^{1}$ | WorkGroupLocal | 128 | 256 | 32768 | 1.21 | 2.2 | 1.04 | 1.01 | 1.36 |
| $2^{2}$ | WorkGroupLocal | 256 | 256 | 65536 | 1.65 | 1.74 | 1.05 | 1.02 | 1.58 |
| $2^{3}$ | WorkGroupLocal | 256 | 256 | 65536 | 1.93 | 1.63 | 1.04 | 1.01 | 1.95 |
| $2^{4}$ | WorkGroupLocal | 256 | 256 | 65536 | 3.0 | 1.38 | 1.04 | 1.02 | 3.5 |
| $2^{5}$ | WorkGroupLocal | 256 | 256 | 65536 | 3.7 | 1.36 | 1.04 | 1.02 | 3.8 |
| $2^{6}$ | WorkGroupLocal | 256 | 128 | 32768 | 7.6 | 1.08 | 1.04 | 1.02 | 4.1 |
| $2^{7}$ | WorkGroupLocal | 512 | 256 | 131072 | **16.9** | 1.40 | 1.45 | **1.83** | **5.1** |
| $2^{8}$ | WorkGroupLocal | 128 | 256 | 32768 | 10.4 | 1.20 | 1.07 | 1.04 | 2.7 |
| $2^{9}$ | WorkGroupLocal | 64 | 256 | 16384 | 8.8 | 1.51 | 1.17 | 1.06 | 1.79 |
| $2^{10}$ | WorkGroupLocal | 32 | 256 | 8192 | 6.1 | 3.2 | 1.31 | 1.19 | 2.1 |
| $2^{11}$ | WorkGroupLocal | 8 | 256 | 2048 | 2.8 | 3.5 | **1.58** | 1.46 | 1.59 |
| $2^{12}$ | Shared | 128 | 256 | 32768 | 1.01 | 1.01 | 1.01 | 1.01 | 1.13 |
| $2^{13}$ | Shared | 1024 | 256 | 262144 | 1.27 | 1.02 | 1.00 | 1.01 | 1.56 |
| $2^{14}$ | Shared | 512 | 256 | 131072 | 1.08 | 1.00 | 1.01 | 1.01 | 1.26 |
| $2^{15}$ | Shared | 64 | 256 | 16384 | 1.15 | 1.03 | 1.01 | 1.01 | 1.10 |
| $2^{16}$ | Shared | 512 | 128 | 65536 | 1.04 | 1.00 | 1.00 | 1.00 | 1.11 |
| $2^{17}$ | Shared | 512 | 128 | 65536 | 1.07 | 1.16 | 1.07 | 1.12 | 1.10 |
| $2^{18}$ | Shared | 1024 | 128 | 131072 | 1.25 | 1.24 | 1.25 | 1.22 | 1.47 |
| $2^{19}$ | Shared | 64 | 128 | 8192 | 1.48 | 1.33 | 1.39 | 1.30 | 1.14 |
| $2^{20}$ | Shared | 256 | 128 | 32768 | 1.04 | 1.41 | 1.38 | 1.35 | 1.14 |
| $2^{21}$ | Shared | 64 | 128 | 8192 | 1.01 | 1.37 | 1.39 | 1.36 | 1.21 |
| $2^{22}$ | Shared | 1024 | 128 | 131072 | 1.45 | 1.44 | 1.42 | 1.42 | 1.21 |
| $2^{23}$ | Shared | 128 | 128 | 16384 | 1.51 | 1.52 | 1.50 | 1.52 | 1.28 |
| $2^{24}$ | Shared | 256 | 128 | 32768 | 1.09 | 1.16 | 1.41 | 1.52 | 1.28 |
| $2^{25}$ | Shared | 1024 | 256 | 262144 | 1.38 | 1.02 | 1.40 | 1.56 | 1.41 |

(b) Optimized for AMD Radeon R9 Fury

| Groups | Parallelization strategy | WG/CU | WGS | Threads | A10-7850K | Tesla K40m | GeForce GTX 980 | GeForce GTX 1080 | Tesla V100 |
|---|---|---|---|---|---|---|---|---|---|
| $2^{0}$ | WorkGroupLocal | 16 | 64 | 1024 | 1.23 | **4.7** | 1.07 | 1.30 | 1.06 |
| $2^{1}$ | WorkGroupLocal | 32 | 256 | 8192 | 1.10 | 2.1 | 1.06 | 1.01 | 1.05 |
| $2^{2}$ | WorkGroupLocal | 8 | 256 | 2048 | 1.18 | 1.43 | 1.02 | 1.01 | 1.04 |
| $2^{3}$ | WorkGroupLocal | 8 | 256 | 2048 | 1.15 | 1.28 | 1.02 | 1.01 | 1.12 |
| $2^{4}$ | WorkGroupLocal | 8 | 256 | 2048 | 1.13 | 1.00 | 1.02 | 1.02 | 1.15 |
| $2^{5}$ | WorkGroupLocal | 8 | 256 | 2048 | 1.20 | 1.11 | 1.02 | 1.02 | 1.14 |
| $2^{6}$ | WorkGroupLocal | 4 | 256 | 1024 | 1.14 | 1.46 | 1.07 | 1.04 | 1.49 |
| $2^{7}$ | WorkGroupLocal | 4 | 256 | 1024 | 1.16 | 1.17 | 1.07 | 1.04 | 1.43 |
| $2^{8}$ | WorkGroupLocal | 4 | 256 | 1024 | **1.55** | 1.28 | 1.07 | 1.05 | 1.43 |
| $2^{9}$ | WorkGroupLocal | 2 | 256 | 512 | 1.17 | 2.0 | 1.36 | 1.29 | 1.83 |
| $2^{10}$ | WorkGroupLocal | 2 | 256 | 512 | 1.18 | 2.1 | 1.38 | 1.34 | 1.71 |
| $2^{11}$ | WorkGroupLocal | 1 | 256 | 256 | 1.38 | 2.8 | **2.5** | **2.1** | **2.4** |
| $2^{12}$ | Shared | 512 | 32 | 16384 | 1.07 | 1.32 | 1.01 | 1.01 | 1.20 |
| $2^{13}$ | Shared | 1024 | 16 | 16384 | 1.13 | 1.67 | 1.01 | 1.01 | 1.31 |
| $2^{14}$ | Shared | 512 | 16 | 8192 | 1.32 | 1.89 | 1.06 | 1.05 | 1.16 |
| $2^{15}$ | Shared | 512 | 16 | 8192 | 1.08 | 1.67 | 1.02 | 1.04 | 1.15 |
| $2^{16}$ | Shared | 512 | 32 | 16384 | 1.07 | 1.44 | 1.02 | 1.04 | 1.13 |
| $2^{17}$ | Shared | 256 | 128 | 32768 | 1.00 | 1.16 | 1.08 | 1.14 | 1.07 |
| $2^{18}$ | Shared | 256 | 64 | 16384 | 1.03 | 1.12 | 1.30 | 1.24 | 1.05 |
| $2^{19}$ | Shared | 2 | 64 | 128 | 1.06 | 1.58 | 1.04 | 1.00 | 1.17 |
| $2^{20}$ | Shared | 1 | 128 | 128 | 1.09 | 1.57 | 1.03 | 1.00 | 1.05 |
| $2^{21}$ | Shared | 1 | 128 | 128 | 1.03 | 1.41 | 1.02 | 1.00 | 1.02 |
| $2^{22}$ | Shared | 2 | 64 | 128 | 1.05 | 1.45 | 1.02 | 1.00 | 1.02 |
| $2^{23}$ | Shared | 2 | 64 | 128 | 1.15 | 1.68 | 1.02 | 1.00 | 1.06 |
| $2^{24}$ | Shared | 4 | 32 | 128 | 1.24 | 1.59 | 1.03 | 1.00 | 1.10 |
| $2^{25}$ | Shared | 2 | 64 | 128 | 1.03 | 1.02 | 1.01 | 1.00 | 1.01 |
| $2^{26}$ | Shared | 2 | 64 | 128 |  | 1.03 | 1.01 | 1.00 | 1.02 |
| $2^{27}$ | Shared | 2 | 64 | 128 |  | 1.04 | 1.06 | 1.00 | 1.02 |

Table B.1.: Performance penalty of Aggregate kernels optimized for a specific GPU and group cardinality, when executed on other GPUs (continued from previous page).

(c) Optimized for Nvidia Tesla K40m

| Groups | Parallelization strategy | WG/CU | WGS | Threads | A10-7850K | Radeon R9 Fury | GeForce GTX 980 | GeForce GTX 1080 | Tesla V100 |
|---|---|---|---|---|---|---|---|---|---|
| $2^{0}$ | Independent | 2 | 1024 | 2048 | | | 1.92 | 1.71 | 3.3 |
| $2^{1}$ | Independent | 2 | 512 | 1024 | | | **3.4** | **2.7** | **4.6** |
| $2^{2}$ | WorkGroupLocal | 16 | 128 | 2048 | 1.16 | 1.07 | 1.02 | 1.03 | 1.01 |
| $2^{3}$ | WorkGroupLocal | 16 | 128 | 2048 | 1.15 | 1.06 | 1.02 | 1.03 | 1.03 |
| $2^{4}$ | WorkGroupLocal | 16 | 128 | 2048 | 1.14 | 1.13 | 1.02 | 1.02 | 1.17 |
| $2^{5}$ | WorkGroupLocal | 16 | 128 | 2048 | 1.16 | 1.08 | 1.02 | 1.03 | 1.20 |
| $2^{6}$ | WorkGroupLocal | 16 | 128 | 2048 | 1.13 | 1.31 | 1.03 | 1.02 | 1.18 |
| $2^{7}$ | WorkGroupLocal | 64 | 256 | 16384 | 1.03 | **4.2** | 1.14 | 1.01 | 1.68 |
| $2^{8}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.00 | 1.01 | 1.17 |
| $2^{9}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.00 | 1.02 | 1.10 |
| $2^{10}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.00 | 1.00 | 1.18 |
| $2^{11}$ | WorkGroupLocal | 1 | 1024 | 1024 | | | 1.05 | 1.01 | 1.10 |
| $2^{12}$ | Shared | 256 | 512 | 131072 | | | 1.01 | 1.00 | 1.17 |
| $2^{13}$ | Shared | 256 | 512 | 131072 | | | 1.01 | 1.00 | 1.18 |
| $2^{14}$ | Shared | 1024 | 128 | 131072 | 1.01 | 1.14 | 1.01 | 1.01 | 1.45 |
| $2^{15}$ | Shared | 256 | 512 | 131072 | | | 1.00 | 1.00 | 1.17 |
| $2^{16}$ | Shared | 1024 | 128 | 131072 | 1.00 | 1.12 | 1.00 | 1.00 | 1.50 |
| $2^{17}$ | Shared | 1024 | 32 | 32768 | 1.09 | 1.08 | 1.01 | 1.03 | 1.20 |
| $2^{18}$ | Shared | 512 | 32 | 16384 | 1.08 | 1.23 | 1.09 | 1.13 | 1.09 |
| $2^{19}$ | Shared | 1024 | 16 | 16384 | 1.06 | 1.12 | 1.11 | 1.12 | 1.00 |
| $2^{20}$ | Shared | 1024 | 16 | 16384 | 1.09 | 1.19 | 1.13 | 1.18 | 1.02 |
| $2^{21}$ | Shared | 1024 | 16 | 16384 | 1.04 | 1.12 | 1.10 | 1.15 | 1.10 |
| $2^{22}$ | Shared | 1024 | 16 | 16384 | 1.06 | 1.17 | 1.15 | 1.22 | 1.08 |
| $2^{23}$ | Shared | 1024 | 16 | 16384 | 1.11 | 1.19 | 1.15 | 1.24 | 1.10 |
| $2^{24}$ | Shared | 1024 | 32 | 32768 | **1.25** | 1.44 | 1.28 | 1.39 | 1.15 |
| $2^{25}$ | Shared | 256 | 512 | 131072 | | | 1.39 | 1.55 | 1.39 |
| $2^{26}$ | Shared | 256 | 1024 | 262144 | | | 1.39 | 1.71 | 1.54 |
| $2^{27}$ | Shared | 1024 | 2 | 2048 | | 2.8 | 1.00 | 1.09 | 1.01 |
| $2^{28}$ | Shared | 1024 | 1 | 1024 | | | | 1.13 | 1.16 |

(d) Optimized for Nvidia GeForce GTX 980

| Groups | Parallelization strategy | WG/CU | WGS | Threads | A10-7850K | Radeon R9 Fury | Tesla K40m | GeForce GTX 1080 | Tesla V100 |
|---|---|---|---|---|---|---|---|---|---|
| $2^{0}$ | WorkGroupLocal | 4 | 512 | 2048 | | | **10.5** | 1.02 | 1.00 |
| $2^{1}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 6.7 | 1.03 | 1.03 |
| $2^{2}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 4.2 | 1.02 | 1.04 |
| $2^{3}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 3.2 | 1.02 | 1.16 |
| $2^{4}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 2.1 | 1.02 | 1.19 |
| $2^{5}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.90 | 1.02 | 1.21 |
| $2^{6}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.48 | 1.02 | 1.17 |
| $2^{7}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.07 | 1.02 | 1.15 |
| $2^{8}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.00 | 1.01 | 1.17 |
| $2^{9}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.00 | 1.02 | 1.10 |
| $2^{10}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.00 | 1.00 | 1.18 |
| $2^{11}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.03 | 1.00 | 1.00 |
| $2^{12}$ | Shared | 256 | 1024 | 262144 | | | 1.06 | 1.00 | 1.27 |
| $2^{13}$ | Shared | 256 | 1024 | 262144 | | | 1.06 | 1.01 | 1.27 |
| $2^{14}$ | Shared | 256 | 1024 | 262144 | | | 1.07 | 1.00 | 1.26 |
| $2^{15}$ | Shared | 1024 | 128 | 131072 | 1.06 | 1.14 | 1.00 | 1.00 | **1.51** |
| $2^{16}$ | Shared | 1024 | 128 | 131072 | 1.00 | 1.12 | 1.00 | 1.00 | 1.50 |
| $2^{17}$ | Shared | 8 | 128 | 1024 | **1.40** | 1.08 | 1.05 | 1.04 | 1.06 |
| $2^{18}$ | Shared | 2 | 256 | 512 | 1.04 | 1.22 | 1.01 | 1.01 | 1.17 |
| $2^{19}$ | Shared | 1 | 256 | 256 | 1.01 | 1.02 | 1.08 | 1.00 | 1.05 |
| $2^{20}$ | Shared | 2 | 128 | 256 | 1.02 | 1.04 | 1.09 | 1.02 | 1.02 |
| $2^{21}$ | Shared | 1 | 256 | 256 | 1.02 | 1.01 | 1.00 | 1.01 | 1.00 |
| $2^{22}$ | Shared | 1 | 256 | 256 | 1.02 | 1.03 | 1.07 | 1.03 | 1.04 |
| $2^{23}$ | Shared | 4 | 64 | 256 | 1.02 | 1.04 | 1.15 | 1.04 | 1.10 |
| $2^{24}$ | Shared | 1 | 256 | 256 | 1.03 | 1.05 | 1.05 | 1.05 | 1.14 |
| $2^{25}$ | Shared | 1 | 256 | 256 | 1.01 | 1.04 | 1.04 | 1.03 | 1.03 |
| $2^{26}$ | Shared | 1024 | 8 | 8192 | | 1.08 | 1.02 | **1.10** | 1.04 |
| $2^{27}$ | Shared | 1024 | 2 | 2048 | | **2.8** | 1.00 | 1.09 | 1.01 |

Table B.1.: Performance penalty of Aggregate kernels optimized for a specific GPU and group cardinality, when executed on other GPUs (continued from previous page).

(e) Optimized for Nvidia GeForce GTX 1080

| Groups | Parallelization strategy | WG/CU | WGS | Threads | A10-7850K | Radeon R9 Fury | Tesla K40m | GeForce GTX 980 | Tesla V100 |
|---|---|---|---|---|---|---|---|---|---|
| $2^{0}$ | WorkGroupLocal | 4 | 1024 | 4096 | | | **21** | 1.02 | 1.04 |
| $2^{1}$ | WorkGroupLocal | 256 | 128 | 32768 | 1.01 | 1.31 | 1.58 | 1.03 | 1.44 |
| $2^{2}$ | WorkGroupLocal | 256 | 128 | 32768 | 1.02 | **1.79** | 1.16 | 1.03 | 1.42 |
| $2^{3}$ | WorkGroupLocal | 32 | 1024 | 32768 | | | 3.6 | 1.06 | 1.12 |
| $2^{4}$ | WorkGroupLocal | 16 | 1024 | 16384 | | | 2.3 | 1.04 | 1.06 |
| $2^{5}$ | WorkGroupLocal | 32 | 512 | 16384 | | | 1.48 | 1.06 | 1.00 |
| $2^{6}$ | WorkGroupLocal | 64 | 512 | 32768 | | | 1.31 | **1.22** | 1.60 |
| $2^{7}$ | WorkGroupLocal | 32 | 1024 | 32768 | | | 1.28 | 1.07 | 1.30 |
| $2^{8}$ | WorkGroupLocal | 64 | 512 | 32768 | | | 1.14 | 1.20 | 1.69 |
| $2^{9}$ | WorkGroupLocal | 32 | 1024 | 32768 | | | 1.22 | 1.08 | **2.4** |
| $2^{10}$ | WorkGroupLocal | 4 | 1024 | 4096 | | | 1.03 | 1.03 | 1.00 |
| $2^{11}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.03 | 1.00 | 1.00 |
| $2^{12}$ | Shared | 512 | 256 | 131072 | 1.02 | 1.07 | 1.01 | 1.01 | 1.26 |
| $2^{13}$ | Shared | 512 | 256 | 131072 | 1.04 | 1.09 | 1.00 | 1.01 | 1.26 |
| $2^{14}$ | Shared | 256 | 1024 | 262144 | | | 1.07 | 1.00 | 1.26 |
| $2^{15}$ | Shared | 1024 | 64 | 65536 | 1.07 | 1.14 | 1.05 | 1.00 | 1.23 |
| $2^{16}$ | Shared | 512 | 256 | 131072 | 1.00 | 1.04 | 1.00 | 1.00 | 1.26 |
| $2^{17}$ | Shared | 1 | 512 | 512 | | | 1.01 | 1.05 | 1.17 |
| $2^{18}$ | Shared | 1 | 256 | 256 | 1.01 | 1.01 | 1.08 | 1.00 | 1.60 |
| $2^{19}$ | Shared | 1 | 256 | 256 | 1.01 | 1.02 | 1.08 | 1.00 | 1.05 |
| $2^{20}$ | Shared | 1 | 128 | 128 | 1.09 | 1.00 | 1.57 | 1.03 | 1.05 |
| $2^{21}$ | Shared | 2 | 64 | 128 | 1.03 | 1.00 | 1.41 | 1.00 | 1.03 |
| $2^{22}$ | Shared | 1 | 128 | 128 | 1.04 | 1.00 | 1.45 | 1.01 | 1.02 |
| $2^{23}$ | Shared | 1 | 128 | 128 | 1.14 | 1.00 | 1.66 | 1.02 | 1.05 |
| $2^{24}$ | Shared | 2 | 64 | 128 | **1.31** | 1.03 | 1.58 | 1.04 | 1.11 |
| $2^{25}$ | Shared | 1 | 128 | 128 | 1.02 | 1.01 | 1.01 | 1.01 | 1.01 |
| $2^{26}$ | Shared | 1 | 128 | 128 | | 1.01 | 1.02 | 1.01 | 1.01 |
| $2^{27}$ | Shared | 1 | 128 | 128 | | 1.00 | 1.04 | 1.06 | 1.02 |
| $2^{28}$ | Shared | 1 | 128 | 128 | | | 2.2 | | 1.03 |

(f) Optimized for Nvidia Tesla V100

| Groups | Parallelization strategy | WG/CU | WGS | Threads | A10-7850K | Radeon R9 Fury | Tesla K40m | GeForce GTX 980 | GeForce GTX 1080 |
|---|---|---|---|---|---|---|---|---|---|
| $2^{0}$ | WorkGroupLocal | 16 | 128 | 2048 | 1.15 | 1.03 | **4.0** | 1.01 | 1.05 |
| $2^{1}$ | WorkGroupLocal | 16 | 128 | 2048 | **1.18** | 1.02 | 1.42 | 1.01 | 1.05 |
| $2^{2}$ | WorkGroupLocal | 64 | 128 | 8192 | 1.14 | **1.21** | 1.18 | 1.07 | 1.01 |
| $2^{3}$ | WorkGroupLocal | 32 | 256 | 8192 | 1.06 | 1.06 | 1.42 | 1.06 | 1.01 |
| $2^{4}$ | WorkGroupLocal | 32 | 512 | 16384 | | | 1.56 | 1.06 | 1.00 |
| $2^{5}$ | WorkGroupLocal | 32 | 512 | 16384 | | | 1.48 | 1.06 | 1.00 |
| $2^{6}$ | WorkGroupLocal | 8 | 1024 | 8192 | | | 1.55 | 1.04 | 1.00 |
| $2^{7}$ | WorkGroupLocal | 8 | 1024 | 8192 | | | 1.13 | 1.04 | 1.01 |
| $2^{8}$ | WorkGroupLocal | 8 | 1024 | 8192 | | | 1.06 | 1.04 | 1.01 |
| $2^{9}$ | WorkGroupLocal | 8 | 1024 | 8192 | | | 1.07 | 1.03 | 1.01 |
| $2^{10}$ | WorkGroupLocal | 4 | 1024 | 4096 | | | 1.03 | 1.03 | 1.00 |
| $2^{11}$ | WorkGroupLocal | 2 | 1024 | 2048 | | | 1.03 | 1.00 | 1.00 |
| $2^{12}$ | Shared | 32 | 1024 | 32768 | | | 1.06 | 1.01 | 1.01 |
| $2^{13}$ | Shared | 32 | 1024 | 32768 | | | 1.04 | 1.01 | 1.00 |
| $2^{14}$ | Shared | 32 | 1024 | 32768 | | | 1.06 | 1.02 | 1.01 |
| $2^{15}$ | Shared | 32 | 512 | 16384 | | | 1.05 | 1.01 | 1.00 |
| $2^{16}$ | Shared | 32 | 512 | 16384 | | | 1.05 | 1.01 | 1.01 |
| $2^{17}$ | Shared | 32 | 1024 | 32768 | | | 1.17 | 1.08 | 1.12 |
| $2^{18}$ | Shared | 32 | 1024 | 32768 | | | 1.26 | **1.27** | **1.23** |
| $2^{19}$ | Shared | 1024 | 16 | 16384 | 1.06 | 1.12 | 1.00 | 1.11 | 1.12 |
| $2^{20}$ | Shared | 1024 | 8 | 8192 | 1.04 | 1.10 | 1.11 | 1.08 | 1.09 |
| $2^{21}$ | Shared | 1 | 256 | 256 | 1.02 | 1.01 | 1.00 | 1.00 | 1.01 |
| $2^{22}$ | Shared | 1024 | 4 | 4096 | 1.06 | 1.04 | 1.35 | 1.08 | 1.08 |
| $2^{23}$ | Shared | 1024 | 4 | 4096 | 1.06 | 1.03 | 1.50 | 1.06 | 1.07 |
| $2^{24}$ | Shared | 1024 | 4 | 4096 | 1.04 | 1.07 | 1.29 | 1.04 | 1.06 |
| $2^{25}$ | Shared | 1024 | 4 | 4096 | 1.05 | 1.05 | 1.03 | 1.05 | 1.08 |
| $2^{26}$ | Shared | 1024 | 4 | 4096 | | 1.04 | 1.01 | 1.01 | 1.08 |
| $2^{27}$ | Shared | 1024 | 4 | 4096 | | 1.04 | 1.03 | 1.00 | 1.08 |
| $2^{28}$ | Shared | 1024 | 4 | 4096 | | | 1.32 | | 1.07 |

# List of Figures

# List of Tables

# List of Listings

# List of Algorithms

# Bibliography

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: a data stream management system. In *Proc. of SIGMOD'03*, 666. DOI: 10.1145/872757.872855.

[2] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Ré, D. Suciu, M. Stonebraker, T. Walter, and J. Widom. 2016. The Beckman report on database research. *Commun. ACM*, 59, 2, 92–99. DOI: 10.1145/2845915.

[3] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan, L. Dong, M. J. Franklin, J. Freire, A. Halevy, J. M. Hellerstein, S. Idreos, D. Kossmann, T. Kraska, S. Krishnamurthy, V. Markl, S. Melnik, T. Milo, C. Mohan, T. Neumann, B. Chin Ooi, F. Ozcan, J. Patel, A. Pavlo, R. Popa, R. Ramakrishnan, C. Ré, M. Stonebraker, and D. Suciu. 2020. The Seattle report on database research. *SIGMOD Rec.*, 48, 4, 44–53. DOI: 10.1145/3385658.3385668.

[4] D. Abadi, S. Madden, and M. Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proc. of ACM SIGMOD'06*, 671–682. DOI: 10.1145/1142473.1142548.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proc. of USENIX OSDI'16*, 265–283.

[6] Advanced Micro Devices, Inc. 2015. *AMD APP SDK OpenCL Optimization Guide*. (1.0 ed.). http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/.

[7] Advanced Micro Devices, Inc. 2021. More about how ROCm uses PCIe atomics. Retrieved May 16, 2021 from https://rocmdocs.amd.com/en/latest/Installation_Guide/More-about-how-ROCm-uses-PCIe-Atomics.html.

[8] Advanced Micro Devices, Inc. 2021. Welcome to AMD ROCm platform. Retrieved May 7, 2021 from https://rocmdocs.amd.com/en/latest/.

[9] A. Agbaria, D. Minor, N. Peterfreund, E. Rozenberg, and O. Rosenberg. 2017. Overtaking CPU DBMSes with a GPU in whole-query analytic processing with parallelism-friendly execution plan optimization. In *Proc. of ADMS/IMDM@VLDB'17*, 57–78.

[10] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, and A. Troudi. 2018. Rheem: enabling cross-platform data processing: may the big data be with you! *Proc. VLDB Endow.*, 11, 11, 1414–1427. DOI: 10.14778/3236187.3236195.

[11] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *Proc. of VLDB'00*. Vol. 2000, 496–505.

[12] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. 2001. Weaving relations for cache performance. In *Proc. of VLDB'01*, 169–180.

[13] J. Ajanovic. 2009. PCI Express 3.0 overview. In *Proc. of IEEE HCS 21*, 1–61. DOI: 10.1109/HOTCHIPS.2009.7478337.

[14] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. 2015. Implicit parallelism through deep language embedding. In *Proc. of ACM SIGMOD'15*, 47–61. DOI: 10.1145/2723372.2750543.

[15] Amazon Web Services. 2021. Amazon EC2 instance types. Retrieved Apr. 14, 2021 from https://aws.amazon.com/ec2/instance-types/.

[16] S. E. Anderson. 2005. Interleave bits by binary magic numbers. Retrieved Aug. 9, 2022 from https://graphics.stanford.edu/~seander/bithacks.html#InterleaveBMN.

[17] J. Andrews and N. Baker. 2006. Xbox 360 system architecture. *IEEE Micro*, 26, 2, 25–37. DOI: 10.1109/MM.2006.45.

[18] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. In *Proc. of PACT '14*, 303–316. DOI: 10.1145/2628071.2628092.

[19] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. 2017. The case for heterogeneous HTAP. In *Proc. of CIDR'17*.

[20] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. 2015. Spark SQL: relational data processing in Spark. In *Proc. of ACM SIGMOD'15*. SIGMOD, 1383–1394. DOI: 10.1145/2723372.2742797.

[21] S. Arora, D. Bouvier, and C. Weaver. 2020. AMD next generation 7nm Ryzen™ 4000 APU "Renoir". In *Proc. of IEEE HCS 32*, 1–30. DOI: 10.1109/HCS49909.2020.9220414.

[22] P. J. Ashenden. 2008. *The Designer's Guide to VHDL*. (3rd ed.). Morgan Kaufmann.

[23] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1, 2, 97–137. DOI: 10.1145/320455.320457.

[24] M. Baboulin, J. Demmel, J. Dongarra, S. Tomov, and V. Volkov. 2008. Enhancing the performance of dense linear algebra solvers on GPUs [in the MAGMA project]. *Poster at Super Computing*.

[25] N. Bandi, C. Sun, D. Agrawal, and A. E. Abbadi. 2004. Hardware acceleration in commercial databases: a case study of spatial operations. In *Proc. of VLDB'04*, 1021–1032. DOI: 10.1016/B978-012088469-8.50089-9.

[26] R. Barber, C. Garcia-Arellano, R. Grosman, R. Mueller, V. Raman, R. Sidle, M. Spilchen, A. Storm, Y. Tian, P. Tözün, et al. 2017. Evolving databases for new-gen big data applications. In *Proc. of CIDR'17*.

[27] T. Behrens, V. Rosenfeld, J. Traub, S. Breß, and V. Markl. 2018. Efficient SIMD vectorization for hashing in OpenCL. In *Proc. of EDBT'18*, 489–492. DOI: 10.5441/002/edbt.2018.54.

[28] F. Beier, T. Kilias, and K.-U. Sattler. 2012. GiST scan acceleration using coprocessors. In *Proc. of DaMoN'12*, 63–69. DOI: 10.1145/2236584.2236593.

[29] M. Blacher, J. Giesen, S. Laue, J. Klaus, and V. Leis. 2022. Machine learning, linear algebra, and more: is SQL all you need. In *Proc. of CIDR'22*, 1–6.

[30] D. Blythe. 2020. The Xe GPU architecture. In *Proc. of IEEE HCS 32*, 1–27. DOI: 10.1109/HCS49909.2020.9220591.

[31] H.-J. Boehm, A. J. Demers, and S. Shenker. 1991. Mostly parallel garbage collection. *SIGPLAN Not.*, 26, 6, 157–164. DOI: 10.1145/113446.113459.

[32] H.-J. Boehm and M. Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18, 9, 807–820. DOI: 10.1002/spe.4380180902.

[33] K. S. Bøgh, I. Assent, and M. Magnani. 2013. Efficient GPU-based skyline computation. In *Proc. of DaMoN'13*. DOI: 10.1145/2485278.2485283.

[34] K. S. Bøgh, S. Chester, D. Šidlauskas, and I. Assent. 2017. Template skycube algorithms for heterogeneous parallelism on multicore and GPU architectures. In *Proc. of ACM SIGMOD'17*, 447–462. DOI: 10.1145/3035918.3035962.

[35] M. Bohr. 2007. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE SSCS Newsletter*, 12, 1, 11–13. DOI: 10.1109/N-SSC.2007.4785534.

[36] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proc. of ICOOOLPS'09*, 18–25. DOI: 10.1145/1565824.1565827.

[37] P. A. Boncz, M. L. Kersten, and S. Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM*, 51, 12, 77–85. DOI: 10.1145/1409360.1409380.

[38] P. A. Boncz, M. Zukowski, and N. Nes. 2005. MonetDB/X100: hyper-pipelining query execution. In *Proc. of CIDR'05*.

[39] S. Borkar and A. A. Chien. 2011. The future of microprocessors. *Commun. ACM*, 54, 5, 67–77. DOI: 10.1145/1941487.1941507.

[40] D. Bouvier and B. Sander. 2014. Applying AMD's Kaveri APU for heterogeneous computing. In *Proc. of IEEE HCS 26*, 1–42. DOI: 10.1109/HOTCHIPS.2014.7478810.

[41] A. Branover, D. Foley, and M. Steinman. 2012. AMD Fusion APU: Llano. *IEEE Micro*, 32, 2, 28–37. DOI: 10.1109/MM.2012.2.

[42] S. Breß. 2014. The design and implementation of CoGaDB: a column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14, 3, 199–209. DOI: 10.1007/s13222-014-0164-z.

[43] S. Breß, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. 2013. Efficient co-processor utilization in database query processing. *Information Systems*, 38, 8, 1084–1096. DOI: 10.1016/j.is.2013.05.004.

[44] S. Breß, H. Funke, and J. Teubner. 2016. Robust query processing in co-processor-accelerated databases. In *Proc. of ACM SIGMOD'16*, 1891–1906. DOI: 10.1145/2882903.2882936.

[45] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. 2014. GPU-accelerated database systems: survey and open challenges. *Trans. Large Scale Data Knowl. Centered Syst.*, 15, 1–35. DOI: 10.1007/978-3-662-45761-0_1.

[46] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27, 6, 797–822. DOI: 10.1007/s00778-018-0512-y.

[47] E. Brochu, V. M. Cora, and N. de Freitas. 2010. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599. arXiv: 1012.2599.

[48] D. Broneske, S. Breß, and G. Saake. 2015. Database scan variants on modern CPUs: a performance study. In *In Memory Data Management and Analysis*, 97–111. DOI: 10.1007/978-3-319-13960-9_8.

[49] BugVM Project. [n. d.] BugVM. https://github.com/bugvm/bugvm.

[50] T. Burghardt, D. Hirn, and T. Grust. 2022. Functional programming on top of SQL engines. In *Practical Aspects of Declarative Languages*, 59–78.

[51] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. 2015. Apache Flink: stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 36, 4. http://sites.computer.org/debull/A15dec/issue1.htm.

[52] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proc. of USENIX OSDI'18*, 578–594.

[53] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proc. of IEEE IPDPS'11*, 676–687. DOI: 10.1109/IPDPS.2011.70.

[54] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.*, 12, 5, 544–556. DOI: 10.14778/3303753.3303760.

[55] P. Chrysogelos, P. Sioulas, and A. Ailamaki. 2019. Hardware-conscious query processing in GPU-accelerated analytical engines. In *Proc. of CIDR'19*, 9.

[56] G. Chrysos. 2012. Intel® Xeon Phi coprocessor (codename Knights Corner). In *Proc. of IEEE HCS 24*, 1–31. DOI: 10.1109/HOTCHIPS.2012.7476487.

[57] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. 2010. Single-chip heterogeneous computing: does the future include custom logic, FPGAs, and GPGPUs? In *Proc. of ACM/IEEE MICRO 43*, 225–236. DOI: 10.1109/MICRO.2010.36.

[58] J. Cieslewicz and K. A. Ross. 2007. Adaptive aggregation on chip multiprocessors. In *Proc. of VLDB'07*, 339–350.

[59] Cloudera, Inc. [n. d.] Flink and map reduce compatibility. Retrieved May 5, 2022 from https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/dataset/hadoop_map_reduce/.

[60] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. 2011. The mixed workload CH-BenCHmark. In *Proc. of DBTest'11*. DOI: 10.1145/1988842.1988850.

[61] G. P. Copeland and S. N. Khoshafian. 1985. A decomposition storage model. In *Proc. of ACM SIGMOD'85*, 268–279. DOI: 10.1145/318898.318923.

[62] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. 2015. An architecture for compiling UDF-centric workflows. *Proc. VLDB Endow.*, 8, 12, 1466–1477.

[63] G. Cugola and A. Margara. 2012. Processing flows of information: from data stream to complex event processing. *ACM Comput. Surv.*, 44, 3. DOI: 10.1145/2187671.2187677.

[64] J. V. D'silva, F. De Moor, and B. Kemme. 2018. AIDA: abstraction for advanced in-database analytics. *Proc. VLDB Endow.*, 11, 11, 1400–1413. DOI: 10.14778/3236187.3236194.

[65] L. Dagum and R. Menon. 1998. OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5, 1, 46–55. DOI: 10.1109/99.660313.

[66] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang. 2018. Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38, 1, 82–99. DOI: 10.1109/MM.2018.112130359.

[67] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9, 5, 256–268. DOI: 10.1109/JSSC.1974.1050511.

[68] P. J. Denning and T. G. Lewis. 2016. Exponential laws of computing growth. *Commun. ACM*, 60, 1, 54–65. DOI: 10.1145/2976758.

[69] A. Deshpande, Z. Ives, and V. Raman. 2007. Adaptive query processing. *Foundations and Trends® in Databases*, 1, 1, 1–140. DOI: 10.1561/1900000001.

[70] H. Doraiswamy, H. T. Vo, C. T. Siva, and J. Freire. 2016. A GPU-based index to support interactive spatio-temporal queries over historical data. In *Proc. of IEEE ICDE'16*, 1086–1097. DOI: 10.1109/ICDE.2016.7498315.

[71] C. Duta and T. Grust. 2020. Functional-style SQL UDFs with a capital 'F'. In *Proc. of ACM SIGMOD'20*, 1273–1287. DOI: 10.1145/3318464.3389707.

[72] C. Duta, D. Hirn, and T. Grust. 2020. Compiling PL/SQL away. In *Proc. of CIDR'20*.

[73] S. A. Dyer and B. K. Harms. 1993. Digital signal processing. In vol. 37. Elsevier, 59–117. DOI: 10.1016/S0065-2458(08)60403-9.

[74] Ècole Polytechnique Fédérale Lausanne (EPFL). [n. d.] The Scala Programming Language. https://www.scala-lang.org/.

[75] A. Eldawy and M. F. Mokbel. 2016. The era of big spatial data: a survey. *Foundations and Trends® in Databases*, 6, 3-4, 163–273. DOI: 10.1561/1900000054.

[76] G. Elder. 2002. Radeon 9700. In *Proc. of ACM SIGGRAPH/Eurographics'02 Tutorials*.

[77] G. Essertel, R. Tahboub, J. Decker, K. Brown, K. Olukotun, and T. Rompf. 2018. Flare: optimizing Apache Spark with native compilation for scale-up architectures and medium-size data. In *Proc. of USENIX OSDI'18*, 799–815.

[78] B. Evans. 2014. Understanding Java JIT compilation with JITWatch, part 1. Retrieved Aug. 28, 2023 from `https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html`.

[79] J. Fang, Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal*, 29, 1, 33–59. DOI: `10.1007/s00778-019-00581-w`.

[80] W. Fang, B. He, and Q. Luo. 2010. Database compression on graphics processors. *Proc. VLDB Endow.*, 3, 1-2, 670–680. DOI: `10.14778/1920841.1920927`.

[81] T. Fischer, D. Hirn, and T. Grust. 2022. Snakes on a plan: compiling Python functions into plain SQL queries. In *Proc. of ACM SIGMOD'22*, 2389–2392. DOI: `10.1145/3514221.3520175`.

[82] Y. Foufoulas and A. Simitsis. 2023. Efficient execution of user-defined functions in SQL queries. *Proc. VLDB Endow.*, 16, 12, 3874–3877. DOI: `10.14778/3611540.3611574`.

[83] Y. Foufoulas, A. Simitsis, L. Stamatogiannakis, and Y. Ioannidis. 2022. YeSQL: "you extend SQL" with rich and highly performant user-defined functions in relational databases. *Proc. VLDB Endow.*, 15, 10, 2270–2283. DOI: `10.14778/3547305.3547328`.

[84] M. J. Freitag and T. Neumann. 2019. Every row counts: combining sketches and sampling for accurate group-by result estimates. In *Proc. of CIDR'19*.

[85] M. Frigo and S. Johnson. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93, 2, 216–231. DOI: `10.1109/JPROC.2004.840301`.

[86] J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, J. Clarkson, and C. Kotselidis. 2019. Dynamic application reconfiguration on heterogeneous hardware. In *Proc. of ACM VEE'19*, 165–178. DOI: `10.1145/3313808.3313819`.

[87] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proc. of ACM SIGMOD'18*, 1603–1618. DOI: `10.1145/3183713.3183734`.

[88] H. Funke, J. Mühlig, and J. Teubner. 2020. Efficient generation of machine code for query compilers. In *Proc. of DaMoN'20*. DOI: `10.1145/3399666.3399925`.

[89] H. Funke and J. Teubner. 2020. Data-parallel query processing on non-uniform data. *Proc. VLDB Endow.*, 13, 6, 884–897. DOI: `10.14778/3380750.3380758`.

[90] H. Funke and J. Teubner. 2021. Low-latency compilation of sql queries to machine code. *Proc. VLDB Endow.*, 14, 12, 2691–2694. DOI: `10.14778/3476311.3476321`.

[91] H. Gavriilidis, K. Beedkar, J.-A. Quiané-Ruiz, and V. Markl. 2023. In-situ cross-database query processing. In *Proc. of IEEE ICDE'23*, 2794–2807. DOI: `10.1109/ICDE55515.2023.00214`.

[92] Google Cloud Platform. 2021. Machine types. Retrieved Apr. 14, 2021 from `https://cloud.google.com/compute/docs/machine-types`.

[93] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. 2015. The Java language specification (Java SE 8 edition). https://docs.oracle.com/javase/specs/jls/se8/html/index.html.

[94] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. of ACM SIGMOD'06*, 325–336. DOI: 10.1145/1142473.1142511.

[95] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. 2004. Fast computation of database operations using graphics processors. In *Proc. of ACM SIGMOD'04*, 215–226. DOI: 10.1145/1007568.1007594.

[96] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. 2005. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proc. of ACM SIGMOD'05*, 611–622. DOI: 10.1145/1066157.1066227.

[97] G. Graefe. 1990. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of ACM SIGMOD'90*, 102–111. DOI: 10.1145/93597.98720.

[98] G. Graefe and L. D. Shapiro. 1991. Data compression and database performance. In *Proc. of IEEE SAC'91*, 22–27. DOI: 10.1109/SOAC.1991.143840.

[99] C. Gregg and K. Hazelwood. 2011. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *Proc. of IEEE ISPASS'11*, 134–144. DOI: 10.1109/ISPASS.2011.5762730.

[100] E. Grochowski, R. Ronen, J. Shen, and H. Wang. 2004. Best of both latency and throughput. In *Proc. of IEEE ICCD'04*, 236–243. DOI: 10.1109/ICCD.2004.1347928.

[101] P. M. Grulich, B. Sebastian, S. Zeuch, J. Traub, J. v. Bleichert, Z. Chen, T. Rabl, and V. Markl. 2020. Grizzly: efficient stream processing through adaptive query compilation. In *Proc. of ACM SIGMOD'20*, 2487–2503. DOI: 10.1145/3318464.3389739.

[102] P. M. Grulich, S. Zeuch, and V. Markl. 2021. Babelfish: efficient execution of polyglot queries. *Proc. VLDB Endow.*, 15, 2, 196–210. DOI: 10.14778/3489496.3489501.

[103] T. Gubner and P. Boncz. 2021. Charting the design space of query execution using VOILA. *Proc. VLDB Endow.*, 14, 6, 1067–1079. DOI: 10.14778/3447689.3447709.

[104] T. Gubner and P. Boncz. 2022. Excalibur: a virtual machine for adaptive fine-grained JIT-compiled query execution based on VOILA. *Proc. VLDB Endow.*, 16, 4, 829–841. DOI: 10.14778/3574245.3574266.

[105] T. Gubner and P. A. Boncz. 2021. Highlighting the performance diversity of analytical queries using voila. In *Proc. of ADMS@VLDB'21*, 47–54.

[106] T. Gubner, D. Tomé, H. Lang, and P. Boncz. 2019. Fluid co-processing: GPU bloom-filters for CPU joins. In *Proc. of DaMoN'19*. DOI: 10.1145/3329785.3329934.

[107] S. Gupta, S. Purandare, and K. Ramachandra. 2020. Aggify: lifting the curse of cursor loops using custom aggregates. In *Proc. of ACM SIGMOD'20*, 559–573. DOI: 10.1145/3318464.3389736.

[108] S. Gupta and K. Ramachandra. 2021. Procedural extensions of SQL: understanding their usage in the wild. *Proc. VLDB Endow.*, 14, 8, 1378–1391. DOI: 10.14778/3457390.3457402.

[109] S. Hagedorn, S. Kläbe, and K.-U. Sattler. 2021. Putting pandas in a box. In *Proc. of CIDR'21*.

[110] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. 2005. Qpipe: a simultaneously pipelined relational query engine. In *Proc. of ACM SIGMOD'05*, 383–394. DOI: 10.1145/1066157.1066201.

[111] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. 2020. Array programming with NumPy. *Nature*, 585, 7825, 357–362. DOI: 10.1038/s41586-020-2649-2.

[112] M. Harris. 2004. General-purpose computation using graphics hardware. In *Eurographics'04 Tutorials*. DOI: 10.2312/egt.20041034.

[113] M. Harris, S. Sengupta, and J. D. Owens. 2007. Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*. Addison Wesley.

[114] Harvard Intelligent Probabilistic Systems Group. [n. d.] Spearmint. Retrieved Mar. 25, 2023 from https://github.com/HIPS/Spearmint.

[115] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. 2007. Efficient gather and scatter operations on graphics processors. In *Proc. of ACM SC'07*. DOI: 10.1145/1362622.1362684.

[116] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. 2009. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34, 4. DOI: 10.1145/1620585.1620588.

[117] B. He and Q. Luo. 2008. Cache-oblivious databases: limitations and opportunities. *ACM Trans. Database Syst.*, 33, 2. DOI: 10.1145/1366102.1366105.

[118] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. 2008. Relational joins on graphics processors. In *Proc. of ACM SIGMOD'08*, 511–524. DOI: 10.1145/1376616.1376670.

[119] B. He and J. X. Yu. 2011. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4, 5, 314–325. DOI: 10.14778/1952376.1952381.

[120] J. He, M. Lu, and B. He. 2013. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *Proc. VLDB Endow.*, 6, 10, 889–900. DOI: 10.14778/2536206.2536216.

[121] J. He, S. Zhang, and B. He. 2014. In-cache query co-processing on coupled CPU-GPU architectures. *Proc. VLDB Endow.*, 8, 4, 329–340. DOI: 10.14778/2735496.2735497.

[122] M. Heimel, M. Kiefer, and V. Markl. 2015. Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In *Proc. of ACM SIGMOD'15*, 1477–1492. DOI: 10.1145/2723372.2749438.

[123] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6, 9, 709–720. DOI: 10.14778/2536360.2536370.

[124] J. L. Hennessy and D. A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM*, 62, 2, 48–60. DOI: 10.1145/3282307.

[125] J. L. Hennessy and D. A. Patterson. 2017. *Computer Architecture: A Quantitative Approach*. (6th ed.). Morgan Kaufmann.

[126] T. Hey, S. Tansley, K. Tolle, et al. 2009. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Vol. 1. Microsoft Research.

[127] M. Hilbert and P. López. 2011. The world's technological capacity to store, communicate, and compute information. *Science*, 332, 6025, 60–65. DOI: 10.1126/science.1200970.

[128] M. D. Hill and M. R. Marty. 2008. Amdahl's law in the multicore era. *Computer*, 41, 7, 33–38. DOI: 10.1109/MC.2008.209.

[129] P. Holanda and H. Mühleisen. 2019. Relational queries with a tensor processing unit. In *Proc. of DaMoN'19*. DOI: 10.1145/3329785.3329932.

[130] D. Horn. 2005. Stream reduction operations for GPGPU applications. In *GPU Gems*. Vol. 2. Addison-Wesley, 573–589.

[131] IEEE Spectrum. 2020. Interactive: the top programming languages. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020.

[132] Intel Corporation. 2013. OpenCL design and programming guide for the Intel Xeon Phi coprocessor. Retrieved Aug. 30, 2022 from https://web.archive.org/web/20130226045033/https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor.

[133] Intel Corporation. 2011. *Writing Optimal OpenCL Code with Intel OpenCL SDK*. https://web.archive.org/web/20150627133253/https://software.intel.com/sites/default/files/m/d/4/1/d/8/Writing_Optimal_OpenCL_28tm_29_Code_with_Intel_28R_29_OpenCL_SDK.pdf.

[134] Z. István, K. Kara, and D. Sidler. 2020. FPGA-accelerated analytics: from single nodes to clusters. *Foundations and Trends® in Databases*, 9, 2, 101–208. DOI: 10.1561/1900000072.

[135] C. S. Jensen, T. B. Pedersen, and C. Thomsen. 2010. Multidimensional databases and data warehousing. *Synthesis Lectures on Data Management*, 2, 1, 1–111. DOI: 10.2200/S00299ED1V01Y201009DTM009.

[136] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza. 2019. Dissecting the NVIDIA Turing T4 GPU via microbenchmarking. *CoRR*, abs/1903.07486. arXiv: 1903.07486.

[137] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826. arXiv: 1804.06826.

[138] N. P. Jouppi, C. Young, N. Patil, and D. Patterson. 2018. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61, 9, 50–59. DOI: 10.1145/3154484.

[139] S. Junkins. 2014. The Compute Architecture of Intel® Processor Graphics Gen7.5. Tech. rep. Intel Corporation. https://www.intel.com/content/dam/develop/external/us/en/documents/compute-architecture-of-intel-processor-graphics-gen7dot5-aug2014-541960.pdf.

[140] Kaggle. 2020. State of Machine Learning and Data Science 2020. Tech. rep. https://www.kaggle.com/kaggle-survey-2020.

[141] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. 2005. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49, 4.5, 589–604. DOI: 10.1147/rd.494.0589.

[142] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. 2012. GPU join processing revisited. In *Proc. of DaMoN'12*, 55–62. DOI: 10.1145/2236584.2236592.

[143] Z. Kaoudi and J.-A. Quiane-Ruiz. 2018. Cross-platform data processing: use cases and challenges. In *Proc. of IEEE ICDE'18*, 1723–1726. DOI: 10.1109/ICDE.2018.00223.

[144] T. Karnagel, T. Ben-Nun, M. Werner, D. Habich, and W. Lehner. 2017. Big data causing big (TLB) problems: taming random memory accesses on the GPU. In *Proc. of DaMoN'17*. DOI: 10.1145/3076113.3076115.

[145] T. Karnagel, D. Habich, and W. Lehner. 2017. Adaptive work placement for query processing on heterogeneous computing resources. *Proc. VLDB Endow.*, 10, 7, 733–744. DOI: 10.14778/3067421.3067423.

[146] T. Karnagel, D. Habich, and W. Lehner. 2015. Local vs. global optimization: operator placement strategies in heterogeneous environments. In *Proc. of EDBT'15 Workshops*, 48–55.

[147] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. 2014. Heterogeneity-aware operator placement in column-store DBMS. *Datenbank-Spektrum*, 14, 3, 211–221. DOI: 10.1007/s13222-014-0167-9.

[148] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. 2013. The HELLS-Join: a heterogeneous stream join for extremely large windows. In *Proc. of DaMoN'13*. DOI: 10.1145/2485278.2485280.

[149] T. Karnagel, R. Müller, and G. M. Lohman. 2015. Optimizing GPU-accelerated group-by and aggregation. In *Proc. of ADMS@VLDB'15*, 13–24.

[150] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. 2014. Managing GPU concurrency in heterogeneous architectures. In *Proc. of IEEE/ACM MICRO 47*, 114–126. DOI: 10.1109/MICRO.2014.62.

[151] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11, 13, 2209–2222. DOI: 10.14778/3275366.3284966.

[152] T. Kersten, V. Leis, and T. Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal*, 30, 5, 883–905. DOI: 10.1007/s00778-020-00643-4.

[153] J. Kessenich, G. Sellers, and D. Shreiner. 2016. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.5 with SPIR-V*. (9th ed.). Addison-Wesley Professional.

[154] Khronos OpenCL Working Group. 2012. The OpenCL specification version 1.2. https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[155] Khronos OpenCL Working Group. 2013. The OpenCL specification version 2.0. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf.

[156] M. Kiefer, M. Heimel, S. Breß, and V. Markl. 2017. Estimating join selectivities using bandwidth-optimized kernel density models. *Proc. VLDB Endow.*, 10, 13, 2085–2096. DOI: 10.14778/3151106.3151112.

[157] G.-H. Kim, S. Trimi, and J.-H. Chung. 2014. Big-data applications in the government sector. *Commun. ACM*, 57, 3, 78–85. DOI: 10.1145/2500873.

[158] B. Kitchenham and R. Carn. 1990. Research and practice: software design methods and tools. In *Psychology of Programming*. Academic Press, 271–284. DOI: 10.1016/B978-0-12-350772-3.50022-7.

[159] S. Kläbe, R. DeSantis, S. Hagedorn, and K.-U. Sattler. 2022. Accelerating Python UDFs in vectorized query execution. In *Proc. of CIDR'22*.

[160] D. E. Knuth. 1998. *The Art of Computer Programming: Sorting and Searching*. (2nd ed.). Vol. 3. Addison Wesley.

[161] R. Koduri. 2019. Exascale for everyone. In *Intel HPC Developer Conference '19*.

[162] A. Kohn, V. Leis, and T. Neumann. 2018. Adaptive execution of compiled queries. In *Proc. of IEEE ICDE'18*, 197–208. DOI: 10.1109/ICDE.2018.00027.

[163] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. 2016. SABER: window-based hybrid stream processing for heterogeneous architectures. In *Proc. of ACM SIGMOD'16*, 555–569. DOI: 10.1145/2882903.2882906.

[164] M. Körber, J. Eckstein, N. Glombiewski, and B. Seeger. 2019. Event stream processing on heterogeneous system architecture. In *Proc. of DaMoN'19*. DOI: 10.1145/3329785.3329933.

[165] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. 2015. Impala: A modern, open-source SQL engine for Hadoop. In *Proc. of CIDR'15*.

[166] C. Kotselidis, I. Komnios, O. Akrivopoulos, S. Bress, K. Doka, H. Mohammed, G. Mylonas, V. Spitadakis, D. Strimpel, J. Fumero, F. S. Zakkak, M. Papadimitriou, M. Xekalaki, N. Foutris, A. Stratikopoulos, N. Koziris, I. Konstantinou, I. Mytilinis, C. Bitsakos, C. Tsalidis, C. Tselios, N. Kanakis, C. Lutz, V. Rosenfeld, and V. Markl. 2020. Efficient compilation and execution of JVM-based data processing frameworks on heterogeneous co-processors. In *Proc. of DATE'20*, 175–179. DOI: 10.23919/DATE48585.2020.9116246.

[167] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. 2018. The case for learned index structures. In *Proc. of ACM SIGMOD'18*, 489–504. DOI: 10.1145/3183713.3196909.

[168] G. Krishnan, D. Bouvier, and S. Naffziger. 2016. Energy-efficient graphics and multimedia in 28-nm carrizo accelerated processing unit. *IEEE Micro*, 36, 2, 22–33. DOI: 10.1109/MM.2016.24.

[169] B. Kumar. 2020. *Machine Learning Based Automatic Tuning of Hash Aggregation on GPUs*. MA thesis. Technische Universität Berlin.

[170] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. 2012. The Vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5, 12, 1790–1801.

[171] H. Lan, Z. Bao, and Y. Peng. 2021. A survey on advancing the DBMS query optimizer: cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6, 1, 86–101. DOI: 10.1007/s41019-020-00149-7.

[172] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. 2016. Data Blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proc. of ACM SIGMOD'16*, 311–326. DOI: 10.1145/2882903.2882925.

[173] T. Lattimore and C. Szepesvári. 2020. *Bandit Algorithms*. Cambridge University Press.

[174] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *Proc. of IEEE CGO'04*, 75–86. DOI: 10.1109/CGO.2004.1281665.

[175] V. Leis, P. Boncz, A. Kemper, and T. Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proc. of ACM SIGMOD'14*, 743–754. DOI: 10.1145/2588555.2610507.

[176] O. Lempel. 2011. 2nd generation Intel® Core processor family: Intel® Core i7, i5 and i3. In *Proc. of IEEE HCS 23*, 1–48. DOI: 10.1109/HOTCHIPS.2011.7477509.

[177] C. Li, Y. Gu, J. Qi, J. He, Q. Deng, and G. Yu. 2018. A GPU accelerated update efficient index for kNN queries in road networks. In *Proc. of IEEE ICDE'18*, 881–892. DOI: 10.1109/ICDE.2018.00084.

[178] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. 2021. OpenGauss: an autonomous database system. *Proc. VLDB Endow.*, 14, 12, 3028–3042. DOI: 10.14778/3476311.3476380.

[179] K. Li. [n. d.] OpenMP Accelerator Support for GPUs. https://www.openmp.org/updates/openmp-accelerator-support-gpus/.

[180] Y. Li, J. Dongarra, and S. Tomov. 2009. A note on auto-tuning GEMM for GPUs. *Computational Science*, 5544, 884–892. DOI: 10.1007/978-3-642-01970-8_89.

[181] Y. Lin and V. Grover. 2018. Using CUDA warp-level primitives. Retrieved May 19, 2021 from https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[182] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro*, 28, 2, 39–55. DOI: 10.1109/MM.2008.31.

[183] E. Lindholm, M. J. Kilgard, and H. Moreton. 2001. A user-programmable vertex engine. In *Proc. of ACM SIGGRAPH'01*, 149–158. DOI: 10.1145/383259.383274.

[184] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. 2015. The Java® virtual machine specification. Java SE 8 edition. https://docs.oracle.com/javase/specs/jvms/se8/html/index.html.

[185] LLVM Project. [n. d.] The LLVM target-independent code generator. Retrieved May 7, 2021 from https://www.llvm.org/docs/CodeGenerator.html.

[186] M. Lu, B. He, and Q. Luo. 2010. Supporting extended precision on graphics processors. In *Proc. of DaMoN'10*, 19–26. DOI: 10.1145/1869389.1869392.

[187] J. Luitjens. 2014. Faster parallel reductions on Kepler. Retrieved Nov. 7, 2020 from https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/.

[188] C. Lutz, S. Breß, T. Rabl, S. Zeuch, and V. Markl. 2018. Efficient k-means on GPUs. In *Proc. of DaMoN'18*. DOI: 10.1145/3211922.3211925.

[189] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. 2020. Pump up the volume: processing large data on GPUs with fast interconnects. In *Proc. of ACM SIGMOD'20*, 1633–1649. DOI: 10.1145/3318464.3389705.

[190] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. 2022. Triton join: efficiently scaling to a large join state on GPUs with fast interconnects. In *Proc. of ACM SIGMOD'22*, 1017–1032. DOI: 10.1145/3514221.3517911.

[191] S. Manegold, P. Boncz, and M. L. Kersten. 2002. Generic database cost models for hierarchical memory systems. In *Proc. of VLDB'02*, 191–202.

[192] M. Mantor. 2019. 7nm "Navi" GPU - a GPU built for performance and efficiency. In *Proc. of IEEE HCS 31*, 1–28. DOI: 10.1109/HOTCHIPS.2019.8875649.

[193] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. 2011. Big data: the next frontier for innovation, competition, and productivity. Retrieved Mar. 12, 2021 from https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/big-data-the-next-frontier-for-innovation.

[194] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. 2021. Bao: making learned query optimization practical. In *Proc. of ACM SIGMOD'21*, 1275–1288. DOI: 10.1145/3448016.3452838.

[195] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. 2019. Neo: a learned query optimizer. *Proc. VLDB Endow.*, 12, 11, 1705–1718. DOI: 10.14778/3342263.3342644.

[196] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. 2003. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22, 3, 896–907. DOI: 10.1145/882262.882362.

[197] C. Mayr, S. Höppner, and S. B. Furber. 2019. SpiNNaker 2: A 10 million core processor system for brain simulation and machine learning. *CoRR*, abs/1911.02385. arXiv: 1911.02385.

[198] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. 2011. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54, 6, 114–123. DOI: 10.1145/1953122.1953148.

[199] P. Menon, T. C. Mowry, and A. Pavlo. 2017. Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11, 1, 1–13. DOI: 10.14778/3151113.3151114.

[200] P. Menon, A. Ngom, L. Ma, T. C. Mowry, and A. Pavlo. 2020. Permutable compiled queries: dynamically adapting compiled queries without recompiling. *Proc. VLDB Endow.*, 14, 2, 101–113. DOI: 10.14778/3425879.3425882.

[201] S. Meraji, B. Schiefer, L. Pham, L. Chu, P. Kokosielis, A. Storm, W. Young, C. Ge, G. Ng, and K. Kanagaratnam. 2016. Towards a hybrid design for fast query processing in DB2 with BLU acceleration using graphical processing units: a technology demonstration. In *Proc. of ACM SIGMOD'16*, 1951–1960. DOI: 10.1145/2882903.2903735.

[202] Microsoft Corporation. 2020. Sizes for virtual machines in Azure. Retrieved Apr. 14, 2021 from https://docs.microsoft.com/en-us/azure/virtual-machines/sizes.

[203] S. Mittal and J. S. Vetter. 2015. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.*, 47, 4. DOI: 10.1145/2788396.

[204] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Kemper, and T. Neumann. 2014. Heterogeneity-conscious parallel query execution: getting a better mileage while driving faster! In *Proc. of DaMoN'14*. DOI: 10.1145/2619228.2619230.

[205] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli. 2016. A comprehensive performance analysis of HSA and OpenCL 2.0. In *Proc. of IEEE ISPASS'16*, 183–193. DOI: 10.1109/ISPASS.2016.7482093.

[206] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. 2015. Cache-efficient aggregation: hashing is sorting. In *Proc. of ACM SIGMOD'15*, 1123–1136. DOI: 10.1145/2723372.2747644.

[207] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proc. of ACM SIGCOMM'18*, 327–341. DOI: 10.1145/3230543.3230560.

[208] T. Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4, 9, 539–550. DOI: 10.14778/2002938.2002940.

[209] T. Neumann and M. J. Freitag. 2020. Umbra: a disk-based system with in-memory performance. In *Proc. of CIDR'20*.

[210] B. Nichols, D. Buttlar, and J. P. Farrell. 1996. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. (1st ed.). O'Reilly & Associates, Inc.

[211] J. Nickolls, I. Buck, M. Garland, and K. Skadron. 2008. Scalable parallel programming with CUDA. *Queue*, 6, 2, 40–53. DOI: 10.1145/1365490.1365500.

[212] NumFOCUS, Inc. [n. d.] pandas. https://pandas.pydata.org/.

[213] NVIDIA Corporation. 2020. *CUDA C++ Best Practices Guide*. (v11.1 ed.). https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.

[214] NVIDIA Corporation. [n. d.] *CUDA C++ Programming Guide*. Retrieved Feb. 1, 2020 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[215] NVIDIA Corporation. 2021. CUDA LLVM compiler. Retrieved May 7, 2021 from https://developer.nvidia.com/cuda-llvm-compiler.

[216] NVIDIA Corporation. [n. d.] *CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/index.html.

[217] NVIDIA Corporation. 2020. NVIDIA A100 Tensor Core GPU Architecture. Tech. rep.

[218] NVIDIA Corporation. 2022. NVIDIA A100 tensor core GPU data sheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf.

[219] NVIDIA Corporation. 2007. *NVIDIA CUDA Programming Guide*. (Version 1.0 ed.).

[220] NVIDIA Corporation. 2012. NVIDIA GeForce GTX 680. Tech. rep.

[221] NVIDIA Corporation. 2014. NVIDIA GeForce GTX 980. Tech. rep.

[222] NVIDIA Corporation. [n. d.] NVIDIA GPUDirect. Retrieved May 29, 2021 from https://developer.nvidia.com/gpudirect.

[223] NVIDIA Corporation. 1999. NVIDIA launches the world's first graphics processing unit: GeForce 256. Retrieved Apr. 30, 2021 from https://web.archive.org/web/20000301052845/http://www.nvidia.com/Geforce256.nsf.

[224] NVIDIA Corporation. 2016. NVIDIA Tesla P100. Tech. rep.

[225] NVIDIA Corporation. 2017. NVIDIA Tesla V100 GPU Architecture. Tech. rep.

[226] NVIDIA Corporation. 2018. NVIDIA Turing GPU Architecture. Tech. rep.

[227] NVIDIA Corporation. 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Tech. rep.

[228] NVIDIA Corporation. 2014. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210. Tech. rep.

[229] NVIDIA Corporation. 2017. Tuning CUDA applications for Maxwell. https://docs.nvidia.com/cuda/maxwell-tuning-guide/.

[230] P. O'Neil, E. O'Neil, and X. Chen. 2009. Star schema benchmark-revision 3. http://www.cs.umbo.edu/~poneil/StarSchemaB.PDF.

[231] Oak Ridge National Laboratory. 2019. Frontier spec sheet. Retrieved Nov. 18, 2020 from https://www.olcf.ornl.gov/wp-content/uploads/2019/05/frontier_specsheet.pdf.

[232] I. Ohmura, G. Morimoto, Y. Ohno, A. Hasegawa, and M. Taiji. 2014. MDGRAPE-4: a special-purpose computer system for molecular dynamics simulations. *Phil. Trans. R. Soc. A.*, 372, 2021. DOI: 10.1098/rsta.2013.0387.

[233] OpenJDK Community. 2012. Graal project. https://openjdk.org/projects/graal/.

[234] Oracle Corperation. [n. d.] Java native interface specification. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[235] Oracle Corporation. [n. d.] Java™ Platform Standard Ed. 8: Class ByteBuffer. Direct vs. non-direct buffers. Retrieved June 11, 2023 from https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html#direct.

[236] Oracle Corporation. [n. d.] Java™ Platform Standard Ed. 8: Class SecurityManager. Retrieved June 11, 2023 from https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html.

[237] R. Pagh and F. F. Rodler. 2004. Cuckoo hashing. *Journal of Algorithms*, 51, 2, 122–144. DOI: 10.1016/j.jalgor.2003.12.002.

[238] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, S. Madden, and M. Zaharia. 2018. Evaluating end-to-end optimization for data analytics applications in Weld. *Proc. VLDB Endow.*, 11, 9, 1002–1015. DOI: 10.14778/3213880.3213890.

[239] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, and M. Zaharia. 2017. Weld: a common runtime for high performance data analytics. In *Proc. of CIDR'17*. DOI: 1721.1/137425.

[240] I. E. Papazian. 2020. New 3rd gen Intel® Xeon® scalable processor (codename: Ice Lake-SP). In *Proc. of IEEE HCS 32*, 1–22. DOI: 10.1109/HCS49909.2020.9220434.

[241] M. Papermaster. 2020. Future of high performance. Retrieved Oct. 28, 2020 from https://ir.amd.com/news-events/analyst-day.

[242] D. A. Patterson. 2004. Latency lags bandwith. *Commun. ACM*, 47, 10, 71–75. DOI: 10.1145/1022594.1022596.

[243] J. Paul, B. He, S. Lu, and C. T. Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on gpus. *Proc. VLDB Endow.*, 14, 2, 202–214. DOI: 10.14778/3425879.3425890.

[244] J. Paul, J. He, and B. He. 2016. GPL: a GPU-based pipelined query processing engine. In *Proc. of ACM SIGMOD'16*, 1935–1950. DOI: 10.1145/2882903.2915224.

[245] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

[246] M. Petre. 1990. Expert programmers and programming languages. In *Psychology of Programming*. Academic Press, 103–115. DOI: https://doi.org/10.1016/B978-0-12-350772-3.50012-4.

[247] H. Pirk, S. Manegold, and M. Kersten. 2014. Waste not... efficient co-processing of relational data. In *Proc. of IEEE ICDE'14*, 508–519. DOI: 10.1109/ICDE.2014.6816677.

[248] H. Pirk, O. Moll, M. Zaharia, and S. Madden. 2016. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9, 14, 1707–1718. DOI: 10.14778/3007328.3007336.

[249] H. Pirk, T. Sellam, S. Manegold, and M. Kersten. 2012. X-device query processing by bitwise distribution. In *Proc. of DaMoN'12*, 48–54. DOI: 10.1145/2236584.2236591.

[250] F. J. Pollack. 1999. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)(abstract only). In *Proc. of IEEE/ACM MICRO 32*, 2.

[251] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer. 2019. Data science through the looking glass and what we found there. (2019). arXiv: 1912.09536 [cs.LG].

[252] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K.-U. Sattler. 2015. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Proc. of TPCTC'14*, 97–112. DOI: 10.1007/978-3-319-15350-6_7.

[253] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. 2004. SPIRAL: a generator for platform-adapted libraries of signal processing alogorithms. *The International Journal of High Performance Computing Applications*, 18, 1, 21–45. DOI: 10.1177/1094342004041291.

[254] Python Software Foundation. [n. d.] Python. https://www.python.org/.

[255] Qualcomm Technologies, Inc. 2020. Snapdragon 888 5G mobile platform. Retrieved Apr. 14, 2021 from https://www.qualcomm.com/media/documents/files/qualcomm-snapdragon-888-mobile-platform-product-brief.pdf.

[256] M. Raasveldt and H. Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proc. of ACM SIGMOD'19*, 1981–1984. DOI: 10.1145/3299869.3320212.

[257] M. Raasveldt and H. Mühleisen. 2016. Vectorized UDFs in column-stores. In *Proc. of SSDBM'16*. DOI: 10.1145/2949689.2949703.

[258] B. Răducanu, P. Boncz, and M. Zukowski. 2013. Micro adaptivity in Vectorwise. In *Proc. of ACM SIGMOD'13*, 1231–1242. DOI: 10.1145/2463676.2465292.

[259] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. 2017. Froid: optimization of imperative programs in a relational database. *Proc. VLDB Endow.*, 11, 4, 432–444. DOI: 10.1145/3186728.3164140.

[260] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. 2013. DB2 with BLU acceleration: so much more than just a column store. *Proc. VLDB Endow.*, 6, 11, 1080–1091. DOI: 10.14778/2536222.2536233.

[261] S. M. A. Raza, P. Chrysogelos, P. Sioulas, V. Indjic, A. C. Anadiotis, and A. Ailamaki. 2020. GPU-accelerated data management under the test of time. In *Proc. of CIDR'20*.

[262] S. Richter, V. Alvarez, and J. Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9, 3, 96–107. DOI: 10.14778/2850583.2850585.

[263] P. Rogers. 2013. Heterogeneous system architecture overview. In *Proc. of IEEE HCS 25*, 1–41. DOI: 10.1109/HOTCHIPS.2013.7478286.

[264] P. Rogers, B. Ander, B. Gaster, and I. Bratt. 2013. Heterogeneous system architecture (HSA): overview and implementation. In *Proc. of IEEE HCS 25*, 1–41. DOI: 10.1109/HOTCHIPS.2013.7478286.

[265] T. Rompf and M. Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proc. of GPCE'10*, 127–136. DOI: 10.1145/1868294.1868314.

[266] V. Rosenfeld, S. Breß, and V. Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Comput. Surv.*, 55, 1. DOI: 10.1145/3485126.

[267] V. Rosenfeld, S. Breß, S. Zeuch, T. Rabl, and V. Markl. 2019. Performance analysis and automatic tuning of hash aggregation on GPUs. In *Proc. of DaMoN'19*. DOI: 10.1145/3329785.3329922.

[268] V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl. 2015. The operator variant selection problem on heterogeneous hardware. In *Proc. of ADMS@VLDB'15*, 1–12.

[269] V. Rosenfeld, R. Mueller, P. Tözün, and F. Özcan. 2017. Processing Java UDFs in a C++ environment. In *Proc. of ACM SoCC'17*, 419–431. DOI: 10.1145/3127479.3132022.

[270] K. A. Ross. 2004. Selection conditions in main memory. *ACM Trans. Database Syst.*, 29, 1, 132–161. DOI: 10.1145/974750.974755.

[271] E. Rozenberg and P. Boncz. 2017. Faster across the PCIe bus: a GPU library for lightweight decompression. Including support for patched compression schemes. In *Proc. of DaMoN'17*. DOI: 10.1145/3076113.3076122.

[272] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere. 2010. An experimental study on performance portability of OpenCL kernels. In *Proc. of SAAHPC'10*, 1–3. DOI: 1854/LU-1016024.

[273] K. Rupp. 2020. 48 years of microprocessor trend data. Retrieved Feb. 9, 2021 from https://github.com/karlrupp/microprocessor-trend-data.

[274] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. 2017. IBM Power9 processor architecture. *IEEE Micro*, 37, 2, 40–51. DOI: 10.1109/MM.2017.40.

[275] N. Sakharnykh. 2018. Everything you need to know about unified memory. In *GPU Tech Conference 2018*.

[276] K. Saur, T. Mirmira, K. Karanasos, and J. Camacho-Rodríguez. 2022. Containerized execution of UDFs: an experimental evaluation. *Proc. VLDB Endow.*, 15, 11, 3158–3171. DOI: 10.14778/3551793.3551860.

[277] T. Schmidt, P. Fent, and T. Neumann. 2022. Efficiently compiling dynamic code for adaptive query processing. In *Proc. of ADMS@VLDB'22*.

[278] M. E. Schüle, L. Scalerandi, A. Kemper, and T. Neumann. 2023. Blue elephants inspecting pandas. In *Proc. of EDBT'23*.

[279] C. Schulze. 2021. *Modeling Memory Contention on GPUs*. MA thesis. Technische Universität Berlin.

[280] Science Staff. 2011. Special online collection: Dealing with data. Challenges and opportunities. *Science*, 331, 6018, 692–693. DOI: 10.1126/science.331.6018.692.

[281] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. 2007. Scan primitives for GPU computing. In *Proc. of ACM SIGGRAPH/Eurographics'07 Workshop*. DOI: 10.2312/EGGH/EGGH07/097-106.

[282] S. Seo, J. Lee, G. Jo, and J. Lee. 2013. Automatic OpenCL work-group size selection for multicore CPUs. In *Proc. of IEEE PACT'13*, 387–398. DOI: 10.1109/PACT.2013.6618834.

[283] A. Shahvarani and H.-A. Jacobsen. 2016. A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In *Proc. of ACM SIGMOD'16*, 1523–1538. DOI: 10.1145/2882903.2882918.

[284] D. D. Sharma and S. Tavallaei. 2020. Compute Express Link™ 2.0 White Paper. Tech. rep. Compute Express Link Consortium. Retrieved Aug. 18, 2023 from https://www.computeexpresslink.org/_files/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf.

[285] D. E. Shaw et al. 2021. Anton 3: twenty microseconds of molecular dynamics simulation before lunch. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. DOI: 10.1145/3458817.3487397.

[286] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua. 2018. Graph processing on GPUs: a survey. *ACM Comput. Surv.*, 50, 6. DOI: 10.1145/3128571.

[287] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop distributed file system. In *Proc. of IEEE MSST'10*, 1–10. DOI: 10.1109/MSST.2010.5496972.

[288] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, 7587, 484–489. DOI: 10.1038/nature16961.

[289] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. 2019. Hardware-conscious hash-joins on GPUs. In *Proc. of IEEE ICDE'19*, 698–709. DOI: 10.1109/ICDE.2019.00068.

[290] A. Skende. 2016. Introducing "Parker": next-generation Tegra system-on-chip. In *Proc. of IEEE HCS 28*, 1–17. DOI: 10.1109/HOTCHIPS.2016.7936207.

[291] R. Smith. 2012. Distinct word length frequencies: distributions and symbol entropies. *Glottometrics*, 23, 7–22.

[292] J. Snoek, H. Larochelle, and R. P. Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*. Vol. 25, 2951–2959.

[293] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. 2016. Knights Landing: second-generation Intel Xeon Phi product. *IEEE Micro*, 36, 2, 34–46. DOI: 10.1109/MM.2016.25.

[294] J. Sompolski, M. Zukowski, and P. Boncz. 2011. Vectorization vs. compilation in query execution. In *Proc. of DaMoN'11*, 33–40. DOI: 10.1145/1995441.1995446.

[295] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter. 2012. The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *Proc. of ACM CF'12*, 103–112. DOI: 10.1145/2212908.2212924.

[296] [n. d.] SQLite. https://www.sqlite.org/.

[297] W. Starke and B. Thompto. 2020. IBM's POWER10 processor. In *Proc. of IEEE HCS 32*, 1–43. DOI: 10.1109/HCS49909.2020.9220618.

[298] E. Stehle and H.-A. Jacobsen. 2017. A memory bandwidth-efficient hybrid radix sort on GPUs. In *Proc. of ACM SIGMOD'17*, 417–432. DOI: 10.1145/3035918.3064043.

[299] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. 2001. LEO – DB2's learning optimizer. In *Proc. of VLDB'01*. Vol. 1, 19–28.

[300] J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12, 3, 66–73. DOI: 10.1109/MCSE.2010.69.

[301] M. Stonebraker and U. Cetintemel. 2005. "one size fits all": an idea whose time has come and gone. In *Proc. of IEEE ICDE'05*, 2–11. DOI: 10.1109/ICDE.2005.1.

[302] D. Suggs, M. Subramony, and D. Bouvier. 2020. The AMD "Zen 2" processor. *IEEE Micro*, 40, 2, 45–52. DOI: 10.1109/MM.2020.2974217.

[303] C. Sun, D. Agrawal, and A. El Abbadi. 2003. Hardware acceleration for spatial selections and joins. In *Proc. of ACM SIGMOD'03*, 455–466. DOI: 10.1145/872757.872813.

[304] H. Sutter. 2005. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30, 3. https://web.archive.org/web/20230707143149/https://www.gotw.ca/publications/concurrency-ddj.htm.

[305] J. Teubner and L. Woods. 2013. Data processing on FPGAs. In *Synthesis Lectures on Data Management*. Number 2. Vol. 5. Morgan Claypool Publishers, 1–118. DOI: 10.2200/S00514ED1V01Y201306DTM035.

[306] The Apache Software Foundation. [n. d.] Apache Arrrow. https://arrow.apache.org/.

[307] The Apache Software Foundation. [n. d.] Apache Avro. https://avro.apache.org/.

[308] The Apache Software Foundation. [n. d.] Apache Flink. https://flink.apache.org.

[309] The Apache Software Foundation. [n. d.] Apache Hadoop. https://hadoop.apache.org.

[310] The Apache Software Foundation. [n. d.] Apache Hive. https://hive.apache.org/.

[311] The Apache Software Foundation. [n. d.] Apache Mahout. https://mahout.apache.org/.

[312] The Apache Software Foundation. [n. d.] Apache Parquet. https://parquet.apache.org.

[313] The Apache Software Foundation. [n. d.] Apache Spark. https://spark.apache.org.

[314] The Apache Software Foundation. [n. d.] Apache Zookeeper. https://zookeeper.apache.org/.

[315] The Apache Software Foundation. [n. d.] Impala User-Defined Functions (UDFs). Retrieved Oct. 10, 2023 from https://impala.apache.org/docs/build/html/topics/impala_udf.html.

[316] The Economist. 2010. Data, data everywhere. A special report on managing information. Retrieved Nov. 20, 2020 from https://www.economist.com/special-report/2010/02/27/data-data-everywhere.

[317] The Khronos Group Inc. 2022. OpenCL conformant products. Retrieved Aug. 30, 2022 from https://www.khronos.org/conformance/adopters/conformant-products/opencl.

[318] The Khronos Group Inc. [n. d.] The open standard for parallel programming of heterogeneous systems. Retrieved Mar. 22, 2020 from https://www.khronos.org/opencl/.

[319] The OpenACC Organization. [n. d.] OpenACC. https://www.openacc.org/.

[320] D. E. Thomas and P. R. Moorby. 2008. *The Verilog® Hardware Description Language*. (5th ed.). Springer Science & Business Media.

[321] Top500. [n. d.] Top500 list – November 2020. Retrieved Apr. 14, 2021 from https://www.top500.org/lists/top500/2020/11/.

[322] TPC. 2010. TPC-C. http://www.tpc.org/tpcc/.

[323] TPC. 2021. TPC-H version 2 and version 3. http://www.tpc.org/tpch/.

[324] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. 2019. SkinnerDB: regret-bounded query evaluation via reinforcement learning. In *Proc. of ACM SIGMOD'19*, 1153–1170. DOI: 10.1145/3299869.3300088.

[325] A. Unkrig et al. [n. d.] Janino Compiler. Retrieved Aug. 8, 2017 from https://janino-compiler.github.io/janino.

[326] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. 2000. DB2 advisor: an optimizer smart enough to recommend its own indexes. In *Proc. of IEEE ICDE'00*, 101–110. DOI: 10.1109/ICDE.2000.839397.

[327] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. 1999. Soot – a Java bytecode optimization framework. In *Proc. of CASCON'99*.

[328] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proc. of ACM SIGMOD'17*, 1009–1024. DOI: 10.1145/3035918.3064029.

[329] X. Vera. 2020. Inside Tiger Lake: Intel's next generation mobile client CPU. In *Proc. of IEEE HCS 32*, 1–26. DOI: 10.1109/HCS49909.2020.9220443.

[330] J. Veselý, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt. 2018. Generic system calls for GPUs. In *Proc. of ACM/IEEE ISCA'18*, 843–856. DOI: 10.1109/ISCA.2018.00075.

[331] T. Vincenty. 1975. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review*, 23, 176, 88–93. DOI: 10.1179/sre.1975.23.176.88.

[332] D. W. Wall. 1993. Limits of Instruction-Level Parallelism. Tech. rep. Western Research Laboratory. Digital Equipment Corp.

[333] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. 2012. Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems. *Proc. VLDB Endow.*, 5, 11, 1543–1554. DOI: 10.14778/2350229.2350268.

[334] Z. Wei and I. Trummer. 2022. SkinnerMT: parallelizing for efficiency and robustness in adaptive query processing on multicore platforms. *Proc. VLDB Endow.*, 16, 4, 905–917. DOI: 10.14778/3574245.3574272.

[335] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. 2000. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29, 3, 55–67. DOI: 10.1145/362084.362137.

[336] R. C. Whaley, A. Petitet, and J. J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27, 1–2, 3–35. DOI: 10.1016/S0167-8191(00)00087-9.

[337] C. Wimmer and T. Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proc. of SPLASH'12*, 13–14. DOI: 10.1145/2384716.2384723.

[338] W. A. Wulf and S. A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23, 1, 20–24. DOI: 10.1145/216585.216588.

[339] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. 2013. One VM to rule them all. In *Proc. of Onward!'13*, 187–204. DOI: 10.1145/2509578.2509581.

[340] M. Xekalaki, J. Fumero, A. Stratikopoulos, K. Doka, C. Katsakioris, C. Bitsakos, N. Koziris, and C. Kotselidis. 2022. Enabling transparent acceleration of big data frameworks using heterogeneous hardware. *Proc. VLDB Endow.*, 15, 13, 3869–3882. DOI: 10.14778/3565838.3565842.

[341] R. Yadav, S. R. Valluri, and M. Zaït. 2023. AIM: a practical approach to automated index management for SQL databases. In *Proc. of IEEE ICDE'23*, 3349–3362. DOI: 10.1109/ICDE55515.2023.00257.

[342] Y. Ye, K. A. Ross, and N. Vesdapunt. 2011. Scalable aggregation on multicore processors. In *Proc. of DaMoN'11*, 1–9. DOI: 10.1145/1995441.1995442.

[343] Y. Yuan, R. Lee, and X. Zhang. 2013. The yin and yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.*, 6, 10, 817–828. DOI: 10.14778/2536206.2536210.

[344] E. T. Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freire. 2017. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *Proc. VLDB Endow.*, 11, 3, 352–365. DOI: 10.14778/3157794.3157803.

[345] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2.

[346] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *Proc. of ACM SOSP'13*, 423–438. DOI: 10.1145/2517349.2522737.

[347] M. Zahran. 2019. *Heterogeneous Computing: Hardware and Software Perspectives*. Association for Computing Machinery.

[348] C. Zeller, R. Fernando, M. Wloka, and M. Harris. 2004. Programming graphics hardware. In *Proc. of Eurographics'04 Tutorials*. DOI: 10.2312/egt.20041034.

[349] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. 2019. Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.*, 12, 5, 516–530. DOI: 10.14778/3303753.3303758.

[350] S. Zeuch, H. Pirk, and J.-C. Freytag. 2016. Non-invasive progressive optimization for in-memory databases. *Proc. VLDB Endow.*, 9, 14, 1659–1670. DOI: 10.14778/3007328.3007332.

[351] B. Zhang, Y. Shen, Y. Zhu, and J. Yu. 2018. A GPU-accelerated framework for processing trajectory queries. In *Proc. of IEEE ICDE'18*, 1037–1048. DOI: 10.1109/ICDE.2018.00097.

[352] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du. 2020. Finestream: fine-grained window-based stream processing on CPU-GPU integrated architectures. In *Proc. of USENIX ATC'20*, 633–647.

[353] G. Zhang, Y. Xu, X. Shen, and I. Dillig. 2021. UDF to SQL translation through compositional lazy inductive synthesis. *Proc. ACM Program. Lang.*, 5, OOPSLA. DOI: 10.1145/3485489.

[354] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proc. of ACM SIGMOD'19*, 415–432. DOI: 10.1145/3299869.3300085.

[355] K. Zhang, J. Hu, B. He, and B. Hua. 2017. DIDO: dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In *Proc. of IEEE ICDE'17*, 671–682. DOI: 10.1109/ICDE.2017.120.

[356] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. 2015. Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8, 11, 1226–1237. DOI: 10.14778/2809974.2809984.

[357] W. Zhang, J. Kim, K. A. Ross, E. Sedlar, and L. Stadler. 2021. Adaptive code generation for data-intensive analytics. *Proc. VLDB Endow.*, 14, 6, 929–942.

[358] X. Zhang, H. Wu, Z. Chang, S. Jin, J. Tan, F. Li, T. Zhang, and B. Cui. 2021. ResTune: resource oriented tuning boosted by meta-learning for cloud databases. In *Proc. of ACM SIGMOD'21*, 2102–2114. DOI: 10.1145/3448016.3457291.

[359] M. Zukowski and P. Boncz. 2012. Vectorwise: beyond column stores. *IEEE Data Engineering Bulletin*, 35, 1, 21–27.

[360] M. Zukowski, S. Héman, N. Nes, and P. Boncz. 2006. Super-scalar RAM-CPU cache compression. In *Proc. of IEEE ICDE'06*, 59–59. DOI: 10.1109/ICDE.2006.150.