Architecting a Pluggable Query Executor for Emerging Co-Processors

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

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

von M.Sc. Balasubramaninan Gurumurthy

geb. am 03.10.1992 in Neyveli

Gutachterinnen/Gutachter

Prof. Dr. rer. nat. habil. Gunter Saake
Prof. Dr.-Ing. Thilo Pionteck
Prof. i. R. Dr. Klaus Meyer-Wegener

Magdeburg, den 25.01.2024

Gurumurthy, Balasubramanian:
Architecting A Pluggable Query Executor for Emerging Co-Processors
Dissertation, Otto-von-Guericke University Magdeburg, 2023.

Abstract

CPUs are reaching their scaling limitations while data keeps growing rapidly. Developers of CPU-based applications are searching for an alternative processor to further improve their efficiency. DBMS being one such application, is always in need of high-performing processors to support the ever-growing data. Therefore, many database researchers investigate various co-processors available in the market to speed up query processing. As a result, many such co-processor accelerated DBMS engines are available both as commercial as well as research projects. Such a DBMS running on a co-processor is normally developed by tightly integrating the hardware-relevant code within the query execution engine. Or, the query engine is written with a common device-agnostic framework (like OpenCL) to support execution on different co-processor architectures. Out of these approaches, the former takes a lot of effort to develop, and the latter is not performance-portable. Additionally, with every new co-processor available in the market, an effort has to be made to develop a query engine on top of this device. Hence, to overcome these challenges, in this thesis, we explore a query executor that lies in the middle ground between the two approaches. Our query engine splits DBMS operators into primitives which are present in a task layer. This layer in addition to the device layer enables the pluggability of co-processors. Further, to reduce implementation effort, we also come up with a unified runtime, that handles query execution across any abstract co-processor. Overall, in this work, we explore a query engine that is capable of plugging in any new co-processor that comes up in the future, without losing the capability to have an optimal implementation over the device.

Zusammenfassung

Unter rapide steigenden Datenmengen stoßen CPUs an ihre Skalierungsgrenzen. Anwendungen, die auf herkömmlichen CPUs laufen, sind auf der Suche nach einem alternativen, spezialisierten Prozessoren, um ihre Effizienz weiter zu verbessern. Datenbankmanagementsysteme sind eine solche Anwendung, die immer leistungsstärkere Prozessoren benötigen, um die ständig wachsenden Datenmengen zu verarbeiten. Daher untersuchen viele Datenbankforscher verschiedene auf dem Markt erhältliche Co-Prozessoren, um die Abfrageverarbeitung zu beschleunigen. Infolgedessen sind viele solcher Co-Prozessor-beschleunigten Datenbank-Engines sowohl als kommerzielle als auch als Forschungsprojekte verfügbar. Ein solches Datenbankmanagementsystem, das auf einem Co-Prozessor läuft, wird normalerweise mit einer engen Integration des Hardware-Codes in die Anfrageausführungs-Engine entwickelt. Alternativ wird die Anfrage-Engine mit einem gemeinsamen geräteunabhängigen Wrapper (wie OpenCL) geschrieben, um die Ausführung auf verschiedenen Co-Prozessor-Architekturen zu unterstützen. Von diesen Ansätzen ist der erste sehr aufwändig in der Entwicklung und der zweite ist nicht leistungsfähig genug. Außerdem muss mit jedem neuen Co-Prozessor, der auf dem Markt verfügbar ist, eine Abfrage-Engine für dieses Gerät entwickelt werden. Um diese Herausforderungen zu überwinden, erforschen wir in dieser Arbeit eine Datenbank-Engine, die in der Mitte zwischen diesen beiden Ansätzen liegt. Unsere Abfrage-Engine teilt Datenbank-Operatoren in Primitive auf, die in einer Aufgabenschicht vorhanden sind. Diese Schicht ermöglicht zusammen mit der Geräteschicht das Plug'n'Play von Co-Prozessoren. Um den Implementierungsaufwand zu verringern, haben wir außerdem eine einheitliche Laufzeit entwickelt, die die Ausführung von Abfragen auf jedem abstrakten Co-Prozessor ermöglicht. Insgesamt entwickeln wir in dieser Arbeit eine Abfrage-Engine, die in der Lage ist, jeden neuen Co-Prozessor, der in der Zukunft auftaucht, einzubinden, ohne die Fähigkeit zu verlieren, eine optimale Implementierung für den Prozessor zu bieten.

Contents

List of Figures

List of Tables

Listings

1. Introduction

Today's co-processor landscape is broad and diverse, with numerous processors differing in their architectures, programming approaches, and processing capabilities, just to name a few. Such diversity in co-processors is necessary to fulfill the needs of many performance-hungry applications. Nowadays, these applications – Database Management Systems (DBMS) being one among them – running on a co-processor are often rewritten to utilize the processor's capabilities to match the performance needs. Such reworks are costly and require expert knowledge of both the hardware and the application. Hence, many researchers work on exploring ways to optimize an application to better fit a target co-processor.

Co-Processor Accelerated DBMS

The early 2000s saw increased use of GPUs (Graphical Processing Units) for query processing[1]: Ranging from offloading a few database operators [80] to a fully-fledged query processor [87, 91, 15]. A similar trend is seen currently with FPGAs (Field Programmable Gate Arrays), leading to research outcomes such as offloading operators [3], up to supporting fully-fledged queries [147]. Similarly, works are also undertaken to utilize other co-processors like APUs (Accelerated Processing Units), MICs (Many Integrated Cores), etc. Hence, many such works have already explored ways to use different co-processors for query execution [37, 154, 67]. Commonly, most of these works re-implement an existing query engine to run on its target co-processor. With new and diverse co-processors ready to hit the market on the horizon, we expect more query engines to be developed for these new ones as well. Such a trajectory will lead to varying implementation alternatives for a database operator across each of these co-processors. Hence, it is necessary to have a system that can manage these co-processor–implementation combinations to support any arbitrary co-processor.

Moreover, while exploring a new implementation for a database operator, one must invest time in developing and optimizing other co-processor driver code as well (like data management between host and co-processor). Such drivers, even though they

[1] DBMS with GPU support: https://dbdb.io/browse?hardware-acceleration=gpu

have no performance impact, are still necessary for execution. For example, to test kernel execution on a GPU, we must also implement the function to compile the kernel on the device. Thus, a query engine over a co-processor must also have these driver components integrated into it.

To sum up, a query engine over any co-processor is a tightly coupled mix of query execution components and co-processor components. Hence, we see diverse query engines tuned for each of the co-processors as they are developed, interspersing query engine code with co-processor code. However, such diversity meant that we had to rework a query processor from scratch for any new co-processor, which takes a considerable effort. Thus, even though many such co-processor-accelerated query engines exist, there is still a gap for a holistic query execution engine that can support any arbitrary co-processor.

In this work, we investigate a query engine that enables developers to plug in a co-processor without rebuilding a complete one. To avoid rework, we envision a layered architecture that separates device drivers from operator implementations so that we can reuse and modify the code without affecting other functionalities. Developing a co-processor-pluggable query engine comes with various challenges. To understand these, we must first study the implications of hardware awareness over DBMS and the current trends in co-processor acceleration.

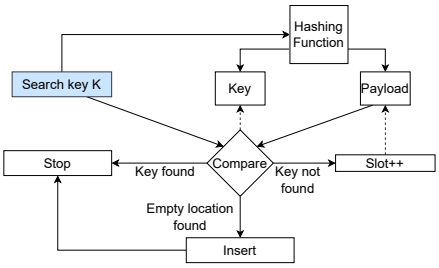
1.1 The Need for Hardware-Awareness in DBMS

Owing to the growing data sizes, DBMS are made to be hardware-aware to ensure a reasonable query execution time. This includes placing the complete table in the main memory (popular example being MonetDB [29]), cache conscious data access [31], hardware sensitive database operators [41], etc. Specifically, with a hardware-aware DBMS operator, many researchers study the underlying processor architecture to come up with a more 'architecture-friendly' operator to gain additional benefits. For example, utilizing SIMD registers in modern CPUs has been shown to increase the performance of DBMS operators. To understand the benefit of hardware-awareness, let us consider the example of hashing: Linear probing is a simple hashing technique as the algorithm in Figure 1.1-a and its execution is visualized in Figure 1.1-b. As we can see, the scalar execution—begins by reading the search key (blue block), and hashing it to identify the target location. Based on the existing values in the target, we have three outcomes: 1) the key is already present—which leads to incrementing the key's count, 2) the slot is empty—insert the search key and 3) a different key is present—increment the slot and insert again. Such workflow in SIMD becomes fairly complicated. As Figure 1.1-c shows, the search key must be replicated to fit the SIMD vector lane. Next, the keys in the target location are fetched and compared with the target key. Based on resultant comparison masks (M1 until M4), we follow three possible outcomes as listed above. We can see from comparing Figure 1.1-b & c, that a complex execution flow is needed for a comparatively simpler function. Still, the outcome of such re-implementation shows additional performance benefits, such as in the case of Figure 1.1-c, where vectorized is nearly twice as fast as the scalar implementation. This is one of the alternative solutions for group-by-aggregation over SIMD. Similar solutions are also available for other operators over SIMD and other processor components, and even further—over other co-processors.

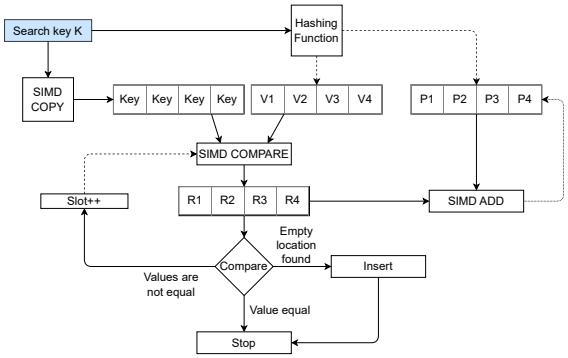
Algorithm 1: Linear probing

```
1 if HashTable full then
2     return false
3 key ← hash(input);
4 if bucket[key] is free then
5     bucket[key] ← input;
6 else
7     while current slot not empty do
8         goto next slot;
```

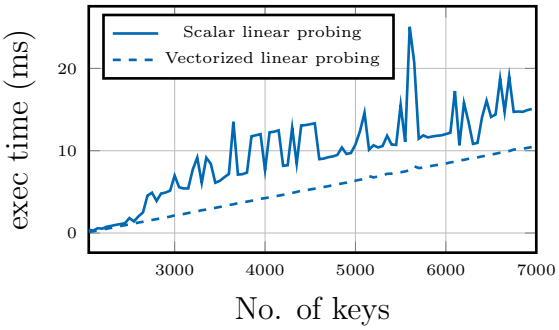
(a) Algorithm for simple linear probing



(b) Scalar workflow for linear probing



(c) SIMD workflow for linear probing



(d) Performance of vectorized linear probing with random distribution

Figure 1.1: Various realization of SIMD vectorization of hashing techniques showing impact over performance

However, developing such co-processor-aware solutions is not trivial. One needs to be an expert in the processor architecture to develop an operator implementation that utilizes the underlying device to its maximum extent. Even though developers are aware of the processor architecture, they must also know the instructions within the various SDKs available for the co-processor. This poses a challenge to developers as there are diverse co-processors, with each co-processor having diverse SDK support.

1.2 Challenges in Hardware & SDK Trends

Until recently, not many co-processors were present in the market to aid a host CPU. Similar is the SDK support for these co-processors to develop different applications. Only at the turn of the decade - 2010s - did we see a rise in different SDKs for each device. For example, previously, GPUs were mainly used for graphical applications. However, with the advent of GPGPUs (General-Purpose computing on Graphics Processing Units), other applications are also being supported over the device. Similarly, FPGA with its VHDL support is now one of the suitable devices to accelerate other applications, including DBMS.

However, developing such a cross-device application needs considerable effort. Efforts in terms of understanding a co-processor architecture, as well as knowing the SDKs (Software Development Kits) operate on a particular component in the underlying architecture. For example, recent GPU architectures support parallel aggregation functions - commonly called *atomics* - directly through a hardware component (more in Chapter 4)[2]. One has to study these SDKs and their relation to the target architecture to utilize the device effectively. Moreover, SDKs provide a varying level of specialized access to hardware components. The access ranges from completely abstracting hardware-related components to specialized access to individual components (more in Chapter 5). One can develop a database operator from any of these levels of abstraction, resulting in various implementation alternatives for even a single database operator.

1.3 DBMS over Heterogeneous Co-Processors

To sum up, on the one hand, we see a strong argument favoring the need for hardware-aware query processing. On the other hand, we have growing diversity in hardware and SDKs, resulting in various query processor realizations. Therefore, there is a potential to develop a query processor over each co-processor and SDK combination. However, such an approach is time-consuming. Furthermore, these solutions must be combined to support an overall best-performing query processor.

Hence, it is imperative to develop a query processor that reduces the implementation effort, providing DBMS support out-of-the-box for any abstract co-processor. To achieve such a holistic query processor, we need a system that supports plugging in a co-processor driver as well as its SDK implementation of DBMS operators. To this end, we explore in this thesis, the abstractions necessary to have such a pluggable architecture with a runtime supporting query execution out-of-the-box on the plugged device.

1.4 Goal of this Thesis

The ultimate goal of the thesis is to explore a query execution engine that supports plugging any arbitrary co-processor. Such a co-processor accelerated DBMS benefits from optimizations on co-processor drivers as well as database operator implementations. Therefore, an abstract pluggable query executor must also have interfaces that can be extended to add the above-mentioned two optimizations. In addition to the optimized code routines for the underlying co-processor, a query executor must also handle the query execution itself. Hence, it is imperative to explore a runtime that can handle query execution over any abstract co-processor. Thus, to achieve the goal of an abstract query executor, we must investigate three dimensions of the executor. Subsequently, we envision a three-tier architecture, with each addressing a particular challenge regarding pluggability.

[2]These are available in NVIDIA CUDA SDK as atomic_add()

Tier-0/1: Device layer - We explore the right level of abstraction for interfacing different co-processors.

Tier 1: Task layer - We explore the right level of abstraction for the database operators and realize them using the existing SDK.

Tier 2: Runtime layer - We explore ways to integrate pluggable interfaces into a common runtime. The runtime supports alternative execution models.

The first two layers explore ways to interface co-processor driver code and its corresponding DBMS operator implementations. Specifically, in the device layer, we focus on developing interfaces for data management that are integral for query execution. The task layer extends interfaces to include an operator implementation over the plugged-in co-processor. The runtime layer acts as the mediator between the query executor and our interfaces. We build our runtime layer using the interfaces designed above, therefore allowing us to have dynamic execution calls across various co-processors. The runtime takes in a query plan and handles execution of it across various co-processors.

1.5 Contributions: Three Tiers of a Pluggable Query Executor

Our pluggable architecture allows one to integrate co-processors without making any changes to the runtime. The architecture also enables one to encapsulate various alternative implementations of a single database operator using its task layer. A similar encapsulation of various SDKs is also possible via the device layer.

The detailed challenges and our proposed solutions for each of these layers are detailed below.

Tier 0/1: The Case for Hardware Sensitivity

At the start of this thesis, we make the case for the need for hardware sensitivity. Here, we explore the performance impact of hardware-aware implementation, as well as the effort invested in developing such an implementation. To this end, we conduct a case study on developing a *GPU-aware* group by aggregation. Specifically, we study sorting-based group by aggregation and modify it for the underlying GPU.

As GPUs are data parallel executors, they need synchronization for aggregation. In the case study, we synchronize aggregates using `atomics` in GPUs. Atomics are specialized serialization instructions executed using a dedicated hardware component in GPUs. Therefore, these functions make our implementation hardware-sensitive. Additionally, the function also exhibits varying performance with multiple tuning parameters. We evaluate these parameters and study the impact of hardware awareness on overall execution. Our evaluation shows these parameters have a significant impact in improving the execution - in some cases, with speed-up execution of up to 2x the naive approaches. Overall, our case study establishes the need for hardware-aware operators and shows their sensitivity to performance.

Tier 1: Pluggable Tasks

Though our case study shows clear performance implications of hardware awareness, we also see that considerable effort and expertise are needed to implement such a hardware-aware operator. Furthermore, even though many researchers have come up with efficient implementations, most of them are ad-hoc solutions [45, 101, 127, 128]. Integrating these ad-hoc solutions needs additional effort. Thus, a single holistic query executor merging these solutions needs to tackle two key challenges:

- Managing implementations across various levels of granularity. For example, sorting can be an ad-hoc implementation, like merge-sort for GPU, or is composed of various smaller functions. For example, Bingshen He and others compose their sort operator using split, map, prefix-sum [87]. Such variations lead to implementations across different levels of granularity present under a single roof (more in Chapter 5).
- An implementation can be realized in various levels of SDK abstractions, such as device-SDK, common wrapper, or expert-written library (more in Chapter 6).

Apart from these main challenges, we also need to consider the varying data representations in each of these implementations. Further, we can also see that some of the operators reuse common operator implementations. Hence, to capture these implementation alternatives, we need a common definition for the operator that spreads across different levels of realizations. Therefore, we survey the existing models and come up with definitions for primitives. To the challenge of varying dimensions of realizations, we place these primitives in different hierarchy levels. Using these definitions and signatures, we can include different operator implementations under a single implementation umbrella. Finally, to test the versatility of our primitive definitions, we realize them using different GPU libraries. By doing so, we study the usability of our task layer as well as the libraries.

Overall, within this tier, we study database operators to come up with granular primitives that encapsulate alternative realizations and test them with existing GPU libraries.

Tier 2: Query Execution Runtime

Once we have a holistic representation of primitives, we need a system that bundles these implementations across different devices. This also includes handling the device-related functionalities like data management, kernel execution, etc. as well as some DBMS-related components: like a proper execution model for the given query plan. For the former, we propose a device layer, which allows for plugging in any abstract co-processor. For the latter, we propose a runtime with alternative execution models. Since the execution model is one of the key components of a query executor, an efficient execution model improves the overall performance. Hence, we study the existing execution models for traditional CPUs, i.e. the vectorized and compiled execution models, and extend them to support co-processors.

Before we go ahead with establishing an execution model for co-processors, we first study the existing ones and their implications for the CPU. Since vectorized and

compiled executions are the prominent execution models, we explore their advantages and disadvantages. Based on our study, we find that compilation suffers from compile time, which contributes to a significant startup delay. To improve on this, we propose a hybrid execution model that follows an interpreted execution while the remaining query is being compiled. Our evaluations show that using such an execution model improves the overall performance by around 2-3x the traditional ones.

Since not all co-processors support reasonable compilation time, we stick with interpretation-based execution. Furthermore, one of the major concerns for databases over co-processors is the limited memory space. Due to this, scaling data sizes might not fit the device completely, and they suffer from executing such big data sizes. To support such execution, we propose a chunked execution model with 3 alternatives: 1) chunked, 2) pipelined, and 3) 4-phased. All these execution models are run on the host CPU, handling execution on the target co-processor.

Putting All Together

Thus, stacking these three layers together, we can have a co-processor pluggable query executor - **ADAMANT**. Since the query processor layers are loosely coupled, the functionalities within the layers can be freely substituted without any changes or awareness of other layers. Furthermore, we also evaluate the performance of using our execution models with GPU as a co-processor. Their performance against HeavyDB shows that we can support any arbitrary query, as they will not run out of memory space in the device. Overall, our ADAMANT query engine has a performance range from dropping up to 3x until gaining speed-up of up to 3x the HeavyDB.

To summarize, the contributions of this thesis are:

- Exploring the hardware-sensitive implementation of non-standard primitives
- A survey of existing database primitives that can be combined to execute complete database operators - developing a standard for our primitives
- Exploring GPU library implementation for standard primitives
- A hybrid execution model for traditional CPU-based query execution
- A scalable execution model for abstract co-processor execution
- A query executor with pluggable interfaces for easy co-processor integration

1.6 Corresponding Publications

The content of this dissertation is an extension of several peer-reviewed publications across workshops, conferences and journals. The following publications are listed in the order they appear in this dissertation.

- B. Gurumurthy, D. Broneske, M. Schäler, T. Pionteck and G. Saake, "An Investigation of Atomic Synchronization for Sort-Based Group-By Aggregation on GPUs," IEEE 37th International Conference on Data Engineering Workshops (ICDEW), Chania, Greece, 48-53(2021).

- B. Gurumurthy, D. Broneske, M. Schäler, T. Pionteck and G. Saake, "Novel insights on atomic synchronization for sort-based group-by on GPUs". Distributed Parallel Databases 41, 387–409(2023).
- B. Gurumurthy, D. Broneske, T. Drewes, T. Pionteck, G. Saake, "Cooking DBMS Operations using Granular Primitives". Datenbank-Spektrum: Vol. 18, No. 3. pp. 183-193(2018).
- H. K. H. Subramanian, B. Gurumurthy, G. C. Durand, D. Broneske and G. Saake, "Analysis of GPU-Libraries for Rapid Prototyping Database Operations: A look into library support for database operations," IEEE 37th International Conference on Data Engineering Workshops (ICDEW), Chania, Greece, 36-41(2021).
- H. K. H. Subramanian, B. Gurumurthy, G. C. Durand, D. Broneske and G. Saake, "Out-of-the-box library support for DBMS operations on GPUs". Distributed Parallel Databases 41, 489–509 (2023).
- B. Gurumurthy, I. Hajjar, D. Broneske, T. Pionteck, G. Saake, "When Vectorwise Meets Hyper, Pipeline Breakers Become the Moderator". In International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB),1-10 (2020).
- B. Gurumurthy, D. Broneske, G. C. Durand, T. Pionteck and G. Saake, "ADAMANT: A Query Executor with Plug-In Interfaces for Easy Co-processor Integration," IEEE 39th International Conference on Data Engineering (ICDE), Anaheim, CA, USA, 1153-1166(2023)

1.7 Outline of this Thesis

The thesis is structured as follows: The next chapter - Chapter 2 - gives an overview of the current state-of-the-art in hardware-accelerated DBMS. The chapter covers the different co-processors currently available, their SDK support, and the challenges in integrating DBMS over these co-processors. We follow these with the existing runtime systems that support heterogeneous co-processor execution in Chapter 3. Next, in Chapter 4 we elaborate on our case study on hardware-aware database operation. Here, we study the GPU-aware group-by-aggregation function to investigate its performance implications. Using the outcome of the case study, we survey the existing database operators and their granular functions in Chapter 5 and realize their implementations over GPU using libraries in Chapter 6. The remaining chapters explore the runtime for our query engine. Similar to our database primitives, we start with understanding the existing execution models from query engines. Based on our study, we developed a hybrid execution model for modern CPUs, which is detailed in Chapter 7. Finally, we put our concepts together into a co-processor pluggable query engine. The engine is detailed in Chapter 8. Finally, we conclude the thesis and postulate on the possible future work in Chapter 9.

2. Tier 0: Current Co-Processor Ecosystem

The consensus nowadays is that *the CPU has reached its scaling limitations* [160]. Yet, the performance demands from a CPU continue to increase, mainly due to the exploding data growth. [180] This is particularly challenging for a Database Management System (DBMS) as an end-user expects query results in a reasonable time. Hence, many DBMS researchers work on optimizing the query engine for the underlying hardware system (such as aligning memory layout, maximizing cache utilization, using in-memory databases, etc.) to have improved query execution. One of these explored solutions to speed up query processing is to use additional co-processors.

A co-processor is a specialized processor adjunct to a general-purpose CPU, mainly developed to accelerate a particular task or application. Even though a co-processor architecture is tailor-made for a target application, it can be used for other general-purpose computing as well. One can use these processors to execute a custom function – built using co-processor APIs. Currently, various APIs are available for an individual co-processor, each having its performance implications over execution. Hence, achieving optimal performance from a co-processor requires us to be aware of both the hardware architecture and the corresponding API that allows access to it.

Due to the variety of co-processors and their corresponding APIs, we can develop multiple implementation variants for a single database operator. Hence, to ease the development effort, we have varying support for programming co-processors ranging from programming paradigms to different systems to generate custom code for a co-processor. Though these systems ease the development effort, additional efforts are needed to handle multiple implementations. Such a handler is typically included within the runtime of the target application, which in our case is a query executor.

In a gist, a DBMS over a co-processor requires changes to its query execution runtime. Such integration opens up various challenges and opportunities for improvement in the query execution runtime. In this chapter, we start with an overview of different co-processor architectures and their ecosystem, which we use to later explore the challenges when integrating them into a DBMS. We use these fundamentals to build our abstract query executor in the subsequent chapters of this thesis.

Downfall of CPU Performance

Before the emergence of co-processors, the CPU has been the sole processor supporting any performance-hungry applications. As the performance requirements from applications increased, CPU capability scaled proportionally to support their needs. The CPU kept pace by cramming more components onto ICs (integrated circuits), whose count Gordan E. Moore has estimated to be doubling every 18 months [160]. However, as is the case with all exponential trends, after a certain limit, doubling components gives only a marginal performance benefit. There are several technological barriers to this behavior, with the most prominent being the power wall. Since simply doubling ICs would make them power-hungry, their power density has to be kept constant. Dennard observed that voltage and current should be proportional to the dimension of a transistor, which is termed Dennard scaling. Exploiting this behavior, shrinking transistors proportionally required less voltage and current, making the overall power density constant. This trend continued until recently when modern chips can no longer shrink the transistor sizes without current leaks[3]. Thus, with the end of Dennard scaling, ICs cannot pack more components without increasing the power density. Hereafter, CPUs were unable to utilize 100% of the transistors without compromising on power. To maintain a fixed power budget, CPUs started to intermittently cool down and run only on a partial transistor count. Thus, to keep up with the performance, multicore CPUs have been introduced.

Such a multicore solution came with the cost of managing parallelism across the cores. Furthermore, doubling CPU cores might not scale beyond a few hundred cores, as managing the cores and powering them will also become an overhead. Thus, scaling performance in a CPU has become more and more challenging.

One other alternative solution to flattening CPU performance is to develop a task-specific co-processor. The co-processor contains only the necessary hardware components for processing a specific application. These co-processors work in tandem with CPUs, where the CPU simply offloads the supported task to a co-processor.

2.1 Current Generation Co-Processors

A co-processor is loosely coupled with a host CPU, typically using a pluggable interface (most commonly via PCI-E). Any input to the co-processor is offloaded into a device's memory before its execution. Though such an execution routine is common across co-processors, they differ in their internal architecture. In this section, we give a brief overview of the commonly available co-processors and their internals.

[3]The leak is dissipated as heat

GPU—Graphics Processing Unit

Video rendering is challenging for traditional CPUs, as thousands of pixels must be constantly updated in real time. Though updating a single pixel might be trivial, it gets fairly complex to update the millions of pixels present in modern displays within a reasonable time. To support such a growing pixel count, CPUs must be fast enough to avoid any significant delays. However, modern CPUs are ill-equipped for such a task. Hence, GPUs are deployed to update the pixel values concurrently. Accordingly, a GPU architecture is equipped with thousands of `light-weight` cores, each executing in a Single-Instruction Multiple-Threads (SIMT) fashion. These cores execute potentially thousands of threads among them, hence supporting massive data-parallel execution. Unlike CPUs designed to have fast response time, GPUs are designed to have high throughput. Hence, the architecture of a GPU also reflects this goal. The SIMT cores deliver high computation throughput while assisted by the faster interconnect that provides high memory bandwidth complete with various levels of caching.

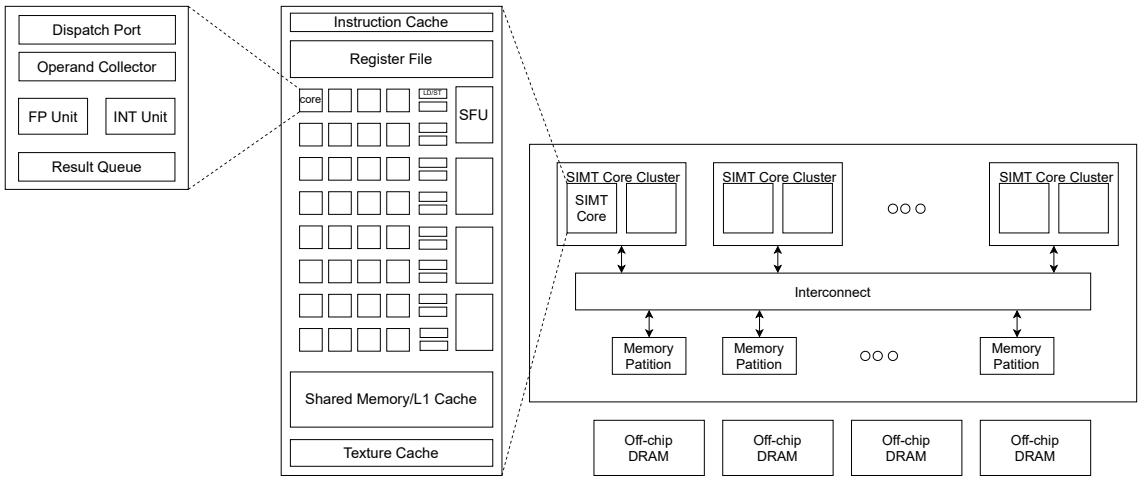


Figure 2.1: Basic components in a GPU

As Figure 2.1 shows, in an overview GPU has three main components: 1) off-chip memory—also known as global memory, which is the memory space for storing data from a host CPU, 2) crossbar—a data access component, that resolves the data access requests while execution, and 3) GPC or Graphics Processing Cluster—the processing core array. The GPC packs thousands of *lightweight* cores that work together in the data-parallel execution of an instruction. A set of GPU cores are clustered into SIMT clusters. The blocks on the top contain the cores. These are named *streaming multiprocessors* by NVIDIA and *compute units* by AMD. These cores share a common instruction cache as well as a data cache. This allows the device to reduce the overall latency within the system. Since numerous threads are working at any given instant, there will be lots of data access calls. These are resolved by an interconnect network (we explain this in detail in chapter 8). The interconnect links the cores with a dedicated off-chip memory in the device.

The single core of a GPC has a lower clock frequency as well as limited capability compared to a CPU core. For example, unlike a CPU, a GPU core is not equipped with branch predictors. It simply runs the else instructions following if instructions,

masking the non-qualifying threads in the corresponding branch. However, such small cores are packed together to support massive parallelism to provide extended performance. Further, memory accesses have high latency that needs to be hidden by processing. To this end, they spawn multiple threads for a given function and use context switching to hide the latency. This massive parallelism in GPUs well suits DBMS workloads. Hence, multiple works have investigated the use of GPUs for accelerating query execution (listed in Chapter 3). We extensively use a GPU to explore co-processor pluggability in this thesis. Hence, we give an in-depth understanding of a GPU and its architecture in this section.

GPUs execute their tasks using threads. The execution here is quite different from CPU threads. In a CPU, the threads are grouped by the core and are normally executed together. In the case of a GPU, a group of threads (defined as warps in CUDA) execute together, and there can be multiple warps in the system. Each warp executes a single instruction in a lockstep. Due to this, in case there is a divergence, threads in a single path are executed together and later the threads follow the alternative path. Hence, it is advised to avoid branching statements in the execution. One more feature of the warp is that threads within the warp have access to a shared local memory in the system.

The GPU supports coalesced access rather than sequential access when reading from global memory. This is again beneficial for the memory controller as the memory access triggered from a warp will be in bulk for all the threads as they are executing in a lock step.

Due to their architecture, GPUs have become the most popular co-processor to a CPU, such that almost all commercial PCs come with a GPU. These commodity-range GPUs, even as of 2023 – nearly a decade after it was first released, are aimed at supporting gaming systems[4]. This is closely followed by use cases in data centers and professional visualizations. There are several niche cases like automotive, healthcare, and logistics, where GPUs are also employed. Such versatility of the GPU is due to its architecture supporting massive data-parallel execution. Such a data-parallel execution is also suitable for the above-mentioned applications as well as many others, like HPC, machine learning, and even query execution in our case. Still, efficiently deploying query execution on GPUs is an ongoing topic and consequently, various GPU-based query executors are currently present and many are still in development. Some commonly known GPU-accelerated DBMS include CoGaDB [38], GPUDB [87], MapD [54] etc.

Though a GPU is also used for accelerating AI applications, its internal components are still not tuned for the application. Current generation GPUs try to support both graphics acceleration and AI using dedicated cores like Tensor cores from Nvidia [96]. However, there are still trade-offs necessary to support both domains. Hence, as a dedicated support for AI, TPU (Tensor Processing Unit) is developed.

TPU—Tensor Processing Unit

Tensor Processing Units are flagship devices from Google for supporting neural network applications. One of the main reasons for exploring TPUs is that Google

[4]NVIDIA investor report from 2023: https://s201.q4cdn.com/141608511/files/doc_presentations/2023/02/nvda-f4q23-investor-presentation-final.pdf

needed to double its data centers to work on machine learning workloads [185]. TPUs accelerate AI techniques (like deep neural network inferences), making them nearly 10x faster than their CPU and GPU counterparts. Similar to a GPU, the TPU contains the hardware components that are necessary for DNN (Deep Neural Network). Moreover, similar to a GPU, the TPU's instruction set also operates over a bulk of data [99].

A simplified block diagram for a TPU based on Jouppi et al. [100] is given in Figure 2.2. The bulk of the TPU space is allocated for the Matrix Multiply Unit. The unit contains 256×256 Multiple-Accumulators (MACs) capable of 8-bit multiplication followed by 16-bit accumulation, i.e., a MAC can process a 256-element sum per cycle. The next biggest space is reserved for a Unified Buffer for Local Activations. It stores the intermediate results in a 24 MiB unified buffer. Finally, the unified buffer has a dedicated DMA to control data access from host memory.

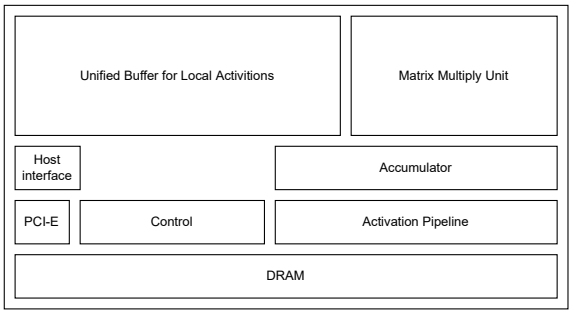


Figure 2.2: Basic components in a TPU

At present, TPUs are exclusively used to speed up machine learning workloads. Specifically, the TPU is well-equipped to run CNNs (convolutional Neural Networks). Though a TPU better fits such AI workload, there is ongoing research to use them for accelerating query processing [93, 96].

Though a TPU fits the AI workload, it also suffers from missing out on many CPU features. Hence, different devices are developed to bridge this gap. Below are two of the most commonly available devices that contain all CPU features while supporting massive data parallelism in a GPU.

MIC—Many Integrated Cores

MICs are the first of many core generations. Here, 10s of CPU cores are packed together to support massive parallel processing. MICs are mainly used in supercomputers and workstations. The first commercial MIC was Intel's Xeon Phi (code-named Knights Corner). it is capable of executing x86-compatible code out-of-the-box. In a nutshell, these are general-purpose CPUs put together to support high levels of parallelism. Unlike standard standalone CPUs, these are connected to a host CPU using PCI-E so that the host can offload some of its operations into the device.

In Figure 2.3, we depict the architecture of one of the MIC systems – Intel's Xeon Phi [170]. The architecture has: processing cores, caches, memory controllers, and a bidirectional ring interconnect. The cores are heavyweight, capable of processing complex operations, and even have a private L2 cache. They support in-order pipelines and up to 4 hardware threads, as well as SIMD vectorization. The whole MIC chip is cache coherent with MESIF[5] protocol [171].

[5]derived from 5 protocol states: *M*odified, *E*xclusive, *S*hared, *I*nvalid, and *F*orward

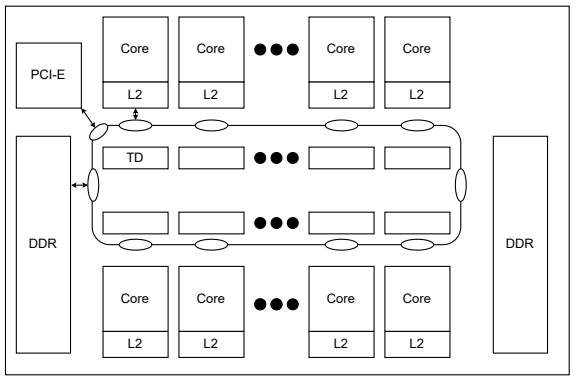


Figure 2.3: Basic components in a MIC

Due to the versatility of the cores in MICs, they are used for accelerating operations in a multitude of applications [125]. MICs are readily used in HPC [98], AI [151] and, naturally, in data processing [141, 161].

The main disadvantage of the MIC is again the managing of the complexity of parallelism. Specifically, communication becomes complex when more cores interact with each other, and complex data transfers take more time than processing using a single core. Hence, to partially solve this, APUs are introduced. These have both CPU and GPU in a single die.

APU—Accelerated Processing Unit

An APU, also known as an *iGPU* (short for integrated GPU) has both CPU and GPU in a single die, both accessing the same memory space (i.e., main memory). The main goal of such a fusion architecture (AMD names its APU platform as fusion architecture) is to have shared memory along with faster interconnects. Depending on the workload, the right device is selected. Usually, the CPU is suitable for serial workloads like web browsing, and cryptography, while the GPU handles data-intensive (vector processing favorable) workloads like real-time graphics rendering, data processing, etc. This avoids unnecessary data transfers and duplication (and subsequently no need for explicit coherency management, less energy consumption, and decreased latency). This also means we get fine-grained data sharing and a good spread of parallelism across both devices. In the next paragraphs, we give an overview of an APU architecture (focusing w.l.o.g on the AMD fusion architecture).

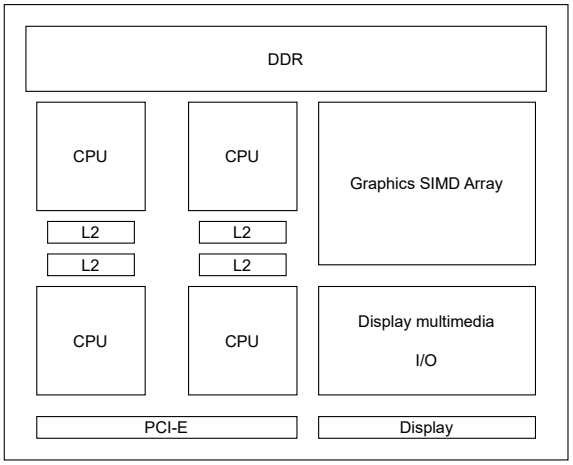


Figure 2.4: Basic components in an APU

Figure 2.4 shows the recreation of AMD Fusion APU: Llano architecture [34]. As we can see in the figure, there are multiple CPU cores with their set of shared caches as well as the graphics accelerator on the same die. Since these two lie on the same die, they are connected to the memory (DDR3) using a high-speed bus. Some of the previous generation of APUs supported discrete address spaces for these two processors. Here, any data from the CPU has to be moved to the GPU addressable space for processing. Yet, this is supported by faster block transfers to copy data from one space to the other.

Since both the CPU and the GPU share a single memory space, data transfer is fairly easy compared to standalone GPUs. This leads to a fine-grained execution of an application across the processors. Hence, APUs are frequently researched for accelerating mathematical routines, video processing, and among others, database systems [90, 184] as well. Though the APU supports both CPU and GPU workloads because of its limited chip size, the performance of these devices in isolation is sub-par compared to the individual devices.

FPGA—Field Programming Gate Arrays

Unlike the above devices, a Field Programming Gate Array (FPGA) supports synthesizing a custom digital circuit. Due to such a level of specialized implementation, FPGAs have gained popularity int he recent years as a suitable data processing device.

The architecture of an FPGA has two parts: Processor System (PS) and Programmable Logic (PL)[6]. The overview of the various components in an FPGA is given in Figure 2.5. The PS contains dual-core or single-core chips and its internal caches and other IO peripherals & interfaces. The PL part contains the Configurable Logic Blocks (CLB termed by Xilinx), routing resources, and IO for off-chip connections (commonly PCI-E to any possible interface). Additionally, the PL also contains DSP blocks and a small block RAM as well (of size 32 kB). Out of these, the key component in an FPGA is the CLB block, which can be configured to support any simple logic and storage functionality. These CLBs contain lookup tables (LUTs), through which one can ultimately design their custom circuit.

The next important feature of an FPGA is its routing architecture. Routing is responsible for feeding and fetching data bits across CLBs. They use wires and programming switches to route data bits across various CLBs. Putting these together, one can create any custom digital circuit leading to having a custom-built circuit for any software function, in our case any database operator. Overall, thousands of CLBs are present in a modern FPGA with a complex routing structure.

Due to such a large amount of configurable blocks available, we need a software stack to configure the device. A custom circuit in an FPGA can be built using a high-level Hardware Description Language (HDL) like VHDL (VHSIC—Very High-Speed Integrated Circuits program—Hardware Description Language). A synthesis tool (like Vivado) can use these VHDL descriptions to create a *netlist* of FPGA's configurable elements and their interconnections. Next, this netlist is mapped to the physical resources in the hardware.

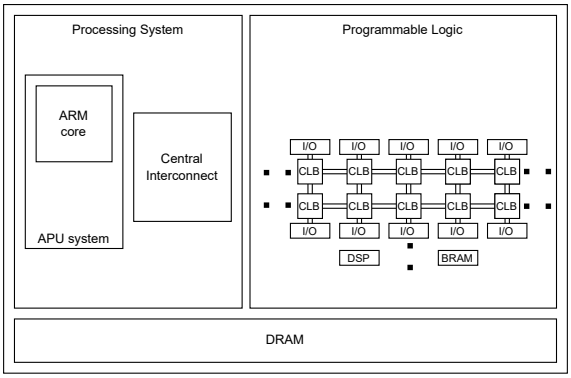


Figure 2.5: Basic architecture of an FPGA

[6]As per the architecture of Xilinx Zynq-7000: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview

In between, various sophisticated functions like placement, routing, and bitstream generation are also carried out depending on the support of various synthesizers. Hence, to ease development, FPGAs are programmed either using RTL (Register Transfer Level) languages like VHDL, and Verilog or via HLS (High-Level Synthesis)—where the circuits are extracted, for example from C or OpenCL code. This provides a platform that can be tuned to perfection for any given domain-specific operation, providing higher throughput. Thus, FPGAs support a flexible and fully customizable architecture that can be adapted to suit any program. The device can be used to build custom data paths, bit-width, and memory hierarchies, making it a completely free environment.

One of the primary benefits of employing FPGAs lies in their ability to support high line rates while processing, making them well-suited for data processing. As a result, FPGAs are synthesized with queries featuring deeper pipelines to harness performance advantages [53]. Furthermore, this pipeline can be replicated to have data parallelism during execution. Finally, depending on the placement of the FPGA in the system architecture, one can use FPGA as [67]: 1) a dedicated co-processor: query or partial query pipeline is pushed to an FPGA for execution, 2) a bandwidth amplifier: FPGA sits between the data source (HDD) and the main processor (CPU). It does data compression/de-compression or filtering to ease data transfer bottleneck, and 3) a near data store: FPGA stores partial data values and performs in-situ query processing before initiating transfer the the main processor. Among these, with the latter two cases, FPGAs are mostly used to process data in-stream while processing and a query is spread across the CPU and co-processor. In this thesis, we focus on the former approach, where the FPGA is tagged as a co-processor via interconnects.

FPGAs are used in data processing to accelerate the execution of individual operators, to running complete queries. Additionally, FPGAs also act as a *near-data* processor that aids in pruning initial results within the data source, so that only a fraction of data is transferred to the main processor.

Nonetheless, a drawback of employing an FPGA is the intricate process of circuit synthesis, which can be exceedingly time-consuming and lead to a complex circuit design. Additionally, despite FPGAs supporting line-rate processing, they often operate at a lower clock rate compared to CPUs or GPUs and are typically enhanced with multiple execution pipelines.

Other Devices

In addition to those previously mentioned, there exist other accelerators, including smart NICs (Network Interface Controller) and digital signal processors that have heterogeneous architecture and support a particular workload. Further, there is an emerging trend of accelerating AI tasks using hardware accelerators like a Vision Processing Unit (VPU). Nonetheless, these devices currently offer support for specific use cases and have yet to undergo significant enhancements to function effectively as dedicated co-processors.

Interconnects

While discussing co-processors, it is also worthwhile to understand the various interconnects used for data movement between the co-processor and the host CPU.

Out of the various interconnects available, the most well-known one is the PCI Express (short for Peripheral Component Interconnect Express[7]). It is one of the most widely used interconnects that has a robust but considerable transfer rate difference across their versions (for example, PCI-E 3.0 has a transfer rate of 8.0 GT/s and PCI-E 4.0 has 16.0 GT/s). Other than the PCI-E, many device-specific connectors are also present. For example, nvlink from NVIDIA supports faster communication compared to the PCI-E (Their performance comparison is discussed in Chapter 8). Furthermore, there are other interconnects like HyperTransport [105], Infinity Fabric [118], Intel QuickPath Interconnect [193] etc. providing competing transfer rates.

Overall, in this section, we emphasized the diversity in co-processor architectures and their performance implications. We showed the various components that make up each of the co-processors, which are relevant for improving performance. However, we must use an appropriate API to program a co-processor to develop an efficient application.

2.2 Programming Co-Processors

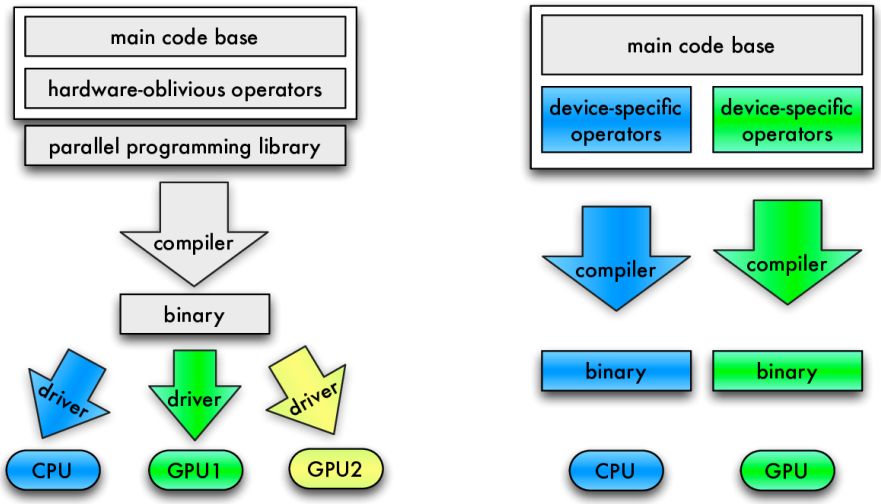
As we saw in the previous section, various internal components of a co-processor make up for its performance. A program puts to use these components efficiently to achieve maximal performance from a co-processor. This is enabled via various programming APIs and instruction sets (in virtual machines). In this section, we briefly present the API support for co-processors.

Since co-processor architectures are diverse, we cannot simply port a code from one device to another without changes in performance. Hence, diversity in co-processors also affects its programming support. At present, we see various programming APIs available across co-processors, each with varying levels of access to the co-processor components. In this section, we discuss the two paradigms and detail some currently available APIs for programming different co-processors.

2.2.1 Programming Paradigms

In a general sense, we can program for a co-processor in a hardware-oblivious way or hardware-sensitive way [91]. In the former approach, the developer needs only the portability of code, and performance comes secondary. The approach favors "write once; run anywhere" development, where the same code can be run in any co-processor. However, the performance may or may not be optimal depending on the underlying co-processor as well as the API used. Alternatively, a hardware-sensitive approach achieves good performance in terms of execution. However, the caveat here is the time for implementation, a co-processor needs a custom implementation to achieve the best performance.

[7] https://pcisig.com/



(a) Hardware oblivious **(b)** Hardware sensitive

Figure 2.6: Common paradigms for programming on a heterogeneous processors environment as given by Heimel et al.

Hardware Oblivious Approach

Hardware-oblivious programming APIs come with a layer in between the programmer and a target co-processor. This abstraction layer provides the necessary constructs for programming—which are compiled to the underlying device during the runtime. Hence, whenever the compiler supports a new co-processor, we can run our code onto this device without much rework. Some common APIs for hardware oblivious programming are OpenCL and OneAPI. They both have C++-like syntax that can be compiled into various co-processors like CPU, GPU, FPGA, etc.

Hardware Sensitive Approach

As an alternative to hardware-oblivious programming, one can also write code that is both aware and exploits the underlying hardware features. Such implementations require a developer to know these features of a target co-processor as well as the right instruction set to exploit them. However, the approach is highly time-consuming and requires considerable effort, depending on the skills of the developer. There are many examples of hardware-aware programming APIs: modern Intel CPUs support SIMD through SSE instructions for handling vectorized processing, OpenMP—a wrapper that enables threading, CUDA for developing programs on top of an NVIDIA GPU, tensorflow for programming TPUs, etc.

2.2.2 Programming APIs

Within the two paradigms mentioned above, we have a variety of API options to pick from for an underlying co-processor. These APIs grant varying levels of specialized access to the target device and its internals. Hence, we must pick the appropriate API based on the use case. Since our goal is to have an efficient query execution, we pick the ones that allow fine granular access to the underlying device components. Below is a detailed overview of the different APIs we use throughout this thesis.

APIs for Multithreading

Threading has become the most common model for enforcing parallelism in an execution. Here, the basic unit of execution is a thread[8] [17]. Nowadays, more or less all the popular programming languages support some levels of threading; ranging from fine-grained (such as spawning individual threads to managing their interactions) to more coarse-grained (such as dictating the parallelism). Many libraries support threading like Pthreads, Qt threads, WinThreads, etc. Here are some of the threading libraries.

Pthreads: Pthreads or POSIX threads have API implementations mainly for UNIX-like systems. It is designed to enable thread-level parallelism within a single node. It provides thread management constructs [9] that a user can use to define parallelism in their applications. Hence, it supports flexible implementation, which in turn leads to better performance. However, developing a highly efficient thread manager is a complex task.

OpenMP: OpenMP supports easy parallelization of loops. It is capable of parallelizing the execution of these loops because the instructions inside the loop can be executed in a highly parallel manner (embarrassingly parallel). This allows the developer to *sprinkle* predefined annotations within the loop and the wrapper takes care of thread creation, handling them and finally executing them in parallel.

```
#pragma omp parallel for num_threads(4) schedule(static,256)
for (int i = 0; i < size; ++i) {
        c[i] = a[i] + b[i];
}
```

Listing 2.1: Simple parallel arithmetic using OpenMP

A simple loop parallelization using OpenMP is depicted in Listing 2.1. As shown, adding annotations allows any user to manipulate thread-level parallelism. Specifically, in this case, we have defined the number of threads to be 4. Furthermore, we can also schedule the data access across threads. In this case, we have defined it to be static, where each thread accesses 256 values.

Since OpenMP favors tight loops, it is suitable for extending data parallelism in database operations. Similar support for GPU is provided via OpenACC [70]. However, this is not expressive enough to optimize for different devices. Alternatively, popularly GPUs are implemented with OpenCL or CUDA.

OpenCL

OpenCL, released in 2008, soon became the industry standard to have a common model for programming across various co-processors [130]. OpenCL supports varying co-processors like multicore CPUs, GPUs, FPGAs, etc. via a host-worker execution model. A program written in OpenCL can be executed on any of the supported devices, thus enabling software portability. Even though OpenCL makes its code portable, it still exposes the underlying device features for the end user to define

[8] It is defined as a lightweight process that enables asynchronous execution

[9] Management functions have a prefix posix_. A complete list of pthread functions is given in https://man7.org/linux/man-pages/man7/pthreads.7.html

execution routines over a particular device. Hence, even though OpenCL is hardware-oblivious, one can develop varying hardware-aware code based on the underlying device. OpenCL offers four models: platform, memory, execution, and programming. Each model exhibits functions to interact with an underlying heterogeneous system:

- **Platform**: describes features of the underlying devices
- **Execution**: offers configurations to execute target functions
- **Memory**: offers memory manipulation functions
- **Programming**: high-level stubs that can be orchestrated to define algorithms that will be executed in the device

Platform Model: Platform model is the lowest level of abstraction for a co-processor. OpenCL follows the host-worker model (which is followed throughout this thesis). It exposes characteristics of any OpenCL-supported co-processor, which is referred to as a *compute device*. Within a compute device, there are multiple Processing Elements (PEs) on which the target instructions are executed.

Execution Model: As the execution spans over both the host and worker, the execution model supports these two groups: host program and co-processor kernel. The host program is dedicated to defining interactions between the host and the co-processor. Whereas, kernels are used to define the instructions to be executed in the co-processor.

To make the execution flexible, the kernels are defined on the host. The host explicitly submits these kernels to a target co-processor, where they are ready for execution. Multiple instances of this compiled kernel – known as *work items* – are executed in parallel in the given co-processor. We can further group these work items into a *work group*, that gives shared access to the internal work items, they can share data among each other.

OpenCL supports writing once; run anywhere. One of the main goals of OpenCL is to enable portability along with exposing the hardware. OpenCL enables such an execution in the sequence below[10]. First, the end user must identify the various OpenCL-supported devices. These devices exhibit various characteristics based on their internal components. For example, GPUs exhibit execution with a massive number of threads, whereas CPUs have comparatively lower thread count. Next, the application kernel code is compiled for the target system, and their appropriate data blocks are prepared. Once all the necessary kernels and data blocks are ready—they can be executed and finally, results are collected back into the host memory space. Let us consider the simple addition from above. It will be implemented in OpenCL in the following steps. We start with preparing the input data as well as the space for the results. Such an example code is written as the one shown in Listing 2.2.

```
cl_mem _m_data_buffer = clCreateBuffer(_m_context,
    ↪ CL_MEM_USE_HOST_PTR, (_size) * _m * sizeof(T), _data,&
    ↪ _m_err);
clFinish(_m_device_queue);
```

Listing 2.2: Creating memory space using OpenCL

[10]For more information, the OpenCL programming guide by Aaftab Munshi is recommended

Once the data is ready, we can write the kernel code that performs the arithmetic operations. An example of such a kernel code is given in 2.3.

```
__kernel void arithAdd(__global int* a, __global int* b,
    ↪ __global int* c, ){
        size_t i = get_global_id();
        c[i] = a[i] + b[i];
}
```

Listing 2.3: Simple parallel arithmetic kernel using OpenCL

From the listing, we can see iterators are missing in the code. Instead of looping through the data, we can assign individual threads per data to be processed. The function `get_global_id()` allows the developer to get the thread ID. Capturing this ID allows one to access the corresponding data. Finally, to get this thread ID, we have to define the number of threads to spawn, which is done as shown in 2.4.

```
...
_m_err = clEnqueueNDRangeKernel(_m_device_queue, kernel_bin, 1,
    ↪ NULL, &_globalSize, &_localSize, 0, NULL, NULL);
...
```

Listing 2.4: Spawning threads for kernel execution

Here, the variables `_globalSize` and `_localSize` define the overall threads to spawn, and the threads packed together within a single workgroup respectively. Setting the _globalSize variable to the size of the input, we can spawn the necessary threads to process all the input data.

Thus, using these different functions, OpenCL abstracts a developer from underlying device details but still exposes the necessary features of the device. Next, we review CUDA, a similar framework that provides dedicated access to NVIDIA GPUs.

CUDA

Similar to OpenCL, CUDA (Compute Unified Device Architecture) is also one of the widely used frameworks for programming a GPU. CUDA is developed by NVIDIA to have dedicated support for their GPU devices. CUDA is structured similarly to OpenCL, with various functions each exposing various aspects of the underlying device. The functional groups in CUDA are:

- **Driver**: initialize the device and enable host access to the co-processor
- **Runtime**: manages execution and synchronization with the host
- **Device runtime**: handles kernel execution and related functions

Similar to the above, let's take the running example of executing arithmetic using CUDA to explain these different function groups.

First, similar to OpenCL, we have to create memory space in the underlying device for the input and output data. The basic constructs of creating a memory space and copying data from the host are given in Listing 2.5.

```
...
int* _data_create;
cudaError_t err = cudaMalloc(&_data_create, dataSize * sizeof(
    ↪ int));
...
```

Listing 2.5: Allocating memory space using CUDA

The syntax of CUDA is similar to C++ and fairly straightforward. Similar constructs are also available for other data management functions like deletion, memory deallocation, etc. In the next listing, we show the code for the arithmetic kernel.

```
__global__ void arith_sum(int *a,int *b, int *c) {

        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        c[tid] = a[tid] + b[tid];
}
```

Listing 2.6: Simple parallel arithmetic kernel using CUDA

As we have seen already in OpenCL, we don't have iterators to process the data in here as well. Once again, the iteration is replaced with spawning threads to process the bulk of data. We can spawn the threads as given in Listing 2.7.

```
__global__ void arith_sum(int *a,int *b, int *c) {
        ...
        err = cudaLaunchKernel(arith_sum,grid,block,(int*)a,(
            ↪ int*)b,(int*)c);
        ...
}
```

Listing 2.7: Spawning threads using CUDA

Here, the variables *grid*, *block* are used to represent the number of threads. These are similar to the global and local variables set in OpenCL. Block represents the number of threads spawned together, and grid defines the number of blocks to spawn.

So far, we have seen the various frameworks used to implement hardware-aware operations across different devices. These frameworks allow specialized implementation over their respective processors, as well as give options for tuning them on the fly to improve performance (more discussed in Chapter 4). As we can see, even with a simple CPU-GPU system, we will have two different frameworks for developing database operators. Hence, we would need a system to manage different frameworks. In the next section, we discuss the existing ways to abstract implementations across heterogeneous co-processors.

2.3 Abstraction Models

An implication of diverse processing architectures is the rise of multiple SDKs: ranging from low-level programming frameworks – like CUDA – to a plethora of

expert-written libraries – like Thrust (more about these are presented in Chapter 6). The availability of multiple SDKs results in numerous implementation alternatives for an operation, each offering competing performance benefits on their respective target devices. As a consequence, organizing this assortment of hardware-based implementations is a key challenge. Furthermore, it is also crucial to organize and manage these SDK alternatives without losing performance [25].

We have already discussed the low-level programming models in the previous section. Since many developers use low-level SDKs, resulting in a complex implementation, it is hard to maintain and update many SDK implementations together. To put it in another way – increasing performance usually tends to lower programmability [68]. Hence, we need a framework that simplifies programmability as well as offers optimal performance from co-processors. Such a solution is offered by abstract models.

As aptly named, abstract models hide architecture-specific details, exposing only the necessary components to develop an optimal code. These abstract models are defined based on the underlying system architecture and their interactions. In this work, we consider the system where co-processors act as workers to a host CPU, i.e., the CPU orchestrates the execution on the co-processor, while the co-processor executes a target function. For example, let us consider a simple vector addition using a GPU. Before executing the addition kernel, the host CPU must issue a data transfer to move input data from the main memory to the device memory. Once successful, the CPU issues kernel execution, the GPU executes the kernel, and signals back to the host on completion. Afterward, the CPU issues commands to read the results back into the main memory. In such a way, the CPU orchestrates execution while target functions are executed in a co-processor. Thus, we have two distinct modules working in tandem to support co-processor acceleration: 1) a CPU-based runtime and 2) co-processor kernels. To easily distinguish them, we call the modules in the host as `runtime` and the kernels as `tasks`.

Runtime: A runtime handles administrative functions like memory management, data transfer, scheduling operations, etc. These are functions necessary for the overall system but do not directly run on a co-processor (though some cases like [122] use co-processors to accelerate runtime components. However, these are still considered a part of the runtime). We discuss more about the runtime functionalities in Chapter 8. Overall, the runtime ensures the proper execution of a task in a co-processor.

Task: Tasks are popularized with OpenMP and CILK at the advent of multicore systems [46, 28]. A task is defined as "...a sequence of instructions that can be processed concurrently with other tasks in the same program, constrained by data and control-flow dependencies..." [178]. Current task-based systems support a wide array of features based on the underlying environment and use-case considered [178].

Therefore, a co-processor acceleration relies on the optimal implementation of both the runtime and kernels. Since developing a custom runtime as well as kernels for an application is highly complex, abstract models are proposed. An abstract model hides the underlying device features and encapsulates them as a high-level SDK. A developer uses such an SDK to architect their desired function, while the abstract model handles execution. Thereby, a developer is alleviated from the burden of mastering the underlying device-specific information. Thus, abstract models offer

better programmability in a co-processor ecosystem. Depending on their level of abstraction, the models are split into two paradigms: *skeleton-based* and *task-based* systems.

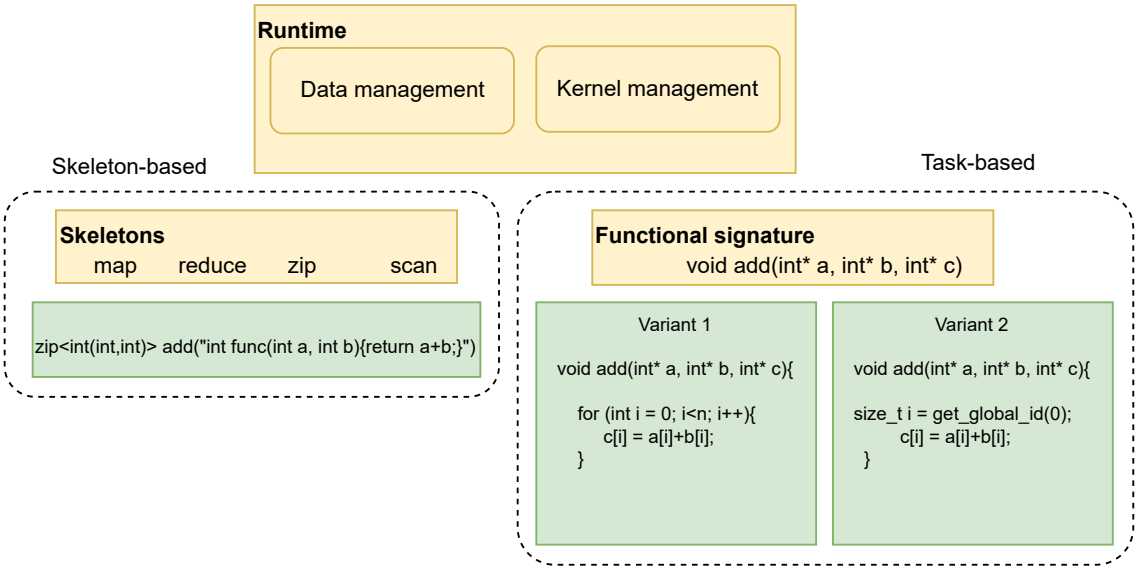


Figure 2.7: Example of abstraction model paradigms. Modules in yellow are offered out-of-the-box and green are provided by developers

Skeleton-based systems are an extension of *algorithmic-skeletons* proposed by M Cole [52] to "structure and simplify parallel programs". These skeletons offer high-level abstractions to model a parallel algorithm. A well-known algorithmic skeleton is `divide & conquer`—that is used to parallelize various functions like sort, aggregate, matrix multiplication and even discrete Fourier transforms. As we know, the skeleton divides a complex problem into sub-problems, solves them individually, and finally combines these results. Though the nature of the skeleton remains the same, the skeleton implementation varies across heterogeneous devices. For example, thread spawn and synchronization of merge-sort in a CPU would be entirely different from a GPU. Thus, skeletons offer a high-level abstraction to express parallelism, while hiding underlying parallel implementation complexities to the end-user. A well-defined set of skeletons is developed to ease development efforts that offer optimal performance across devices. A detailed description of these skeletons is given in the upcoming sections.

Task-based runtime supports parallel execution across heterogeneous cores using tasks itself as an abstraction. These systems define characteristics for programs or tasks running on a co-processor, abstracting them from host runtime. These task definitions are exposed to a developer, through which any custom implementation of the task can be added to the runtime. Since the tasks are predefined, any dependencies across tasks are resolved by the runtime itself. Thus, a task-based runtime allows one to plug in implementations that are internally executed over a target co-processor. For example, a user can add two vectors using a straight-forward serial loop or can use parallel execution using OpenCL code similar to the ones given in Figure 2.7 (right). Both these implementations support optimal performance over their appropriate processor,

which means we have to manage both these implementations. In addition, the developer has to also write a custom runtime, that executes the right implementation on the underlying device. This requires one to implement additional components, which in turn becomes highly complex to manage as well as time-consuming to develop. To avoid these, task-based runtime supports these additional functions out-of-the-box, along with allowing a user to add their custom implementations into the overall system. Thus, a developer is tasked with only developing the custom implementation of a target function optimized for an underlying device, while the runtime handles the execution of the function itself. More about these task-based systems is also detailed in the upcoming sections.

Both paradigms have their advantages and disadvantages. For example, a skeleton-based model offers high expressibility of a program, but the limited set of skeletons might restrict the realization of any custom algorithm. Similarly, task-based runtime supports integrating custom implementations, but with restricted data dependencies due to the predefined task definitions. Hence, we must select the appropriate model based on the use case and expertise level of the developer. In the upcoming sections, we briefly detail these two paradigms and argue their relevance in query execution.

2.3.1 Skeleton-Based Systems

As introduced prior, skeletons represent a higher-order function [173]: A function that takes one or more functions as arguments and returns another function. These skeletons hide hardware-specific implementation while exposing parallel characteristics of an implementation. These characteristics represent a collective operation, i.e., a function that operates over a vector of data, like map, reduce, broadcast, scatter, etc. These skeletons capture an execution pattern as well as communication behavior most commonly present in a parallel system. As the skeleton behaviors are already defined, they can be code-optimized for better execution on a target device. For example, let's say we use loop unrolling[11] to code-optimize a simple vector addition loop. This optimized execution depends on the unroll depth, which in turn depends on the underlying hardware device [43]. Thus, we have to change the unroll depth for the loop with every new device, which is highly time-consuming. To avoid such implementation work, skeletons handle these code optimizations while a user has to only provide the target operator specification. In the case of our running example, a *zip* primitive captures the execution of a binary operation over two operand vectors, producing a single resultant vector. When a user wants to perform vector addition, they can simply use this zip skeleton. The skeleton in turn can optimize the code for better execution, thereby reducing load to the user.

Earlier, these skeletons were employed to support parallelism (typically supporting data-parallel, task-parallel, or resolution-parallel execution) in a multicore system [173]. Hence, these skeletons are commonly classified depending on the parallelism exhibited as data-parallel, task-parallel, and resolution skeletons. of these, data-parallel skeletons have got traction as they are more suitable for various co-processors (a well-known example is the map and reduce skeletons) [68]. Due to the data

[11] expanding a tight for-loop to reduce the costly loop control checks, thereby improving performance.

parallel nature of these skeletons, they work over predefined data formats (usually a vector). As an implication, only a handful of skeletons support such parallelism (more discussed in Chapter 3) like map, reduce, overlap, zip, etc. Each of these skeletons is expanded into a hardware-aware code that provides optimal performance from the underlying device.

Case for Skeleton-Based System

As an example skeleton-based system, we explain the working of Hawk compiler [39]. Hawk supports a minimal set of database-aware skeletons that are combined to form a complete query. Figure 2.8 illustrates the transformed code of a skeleton function created using HAWK [39]. In the example, we have a query pipeline to filter values from a table building a hash table to be used in a hash join. These are represented using the skeletons Loop, Filter, Hash_Put, and Project. These are then transformed into a target code (on the right), which can also support additional code optimizations if needed.

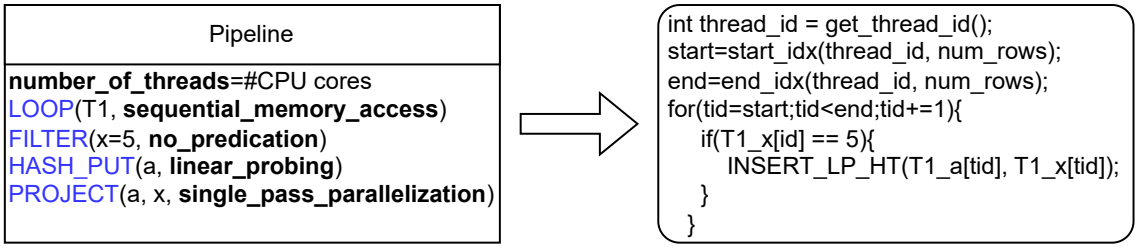


Figure 2.8: Example of code generated using skeletons offered by HAWK

Skeleton-based systems support automated `code optimizations` like loop fusion, loop unrolling, predication [39] etc., given the developer writes their target function using the offered skeletons. Thus, the user aware of skeletons can still develop fairly optimal code for the underlying device target. Thus, skeleton-based systems reduce the level of detail in programming as well as the number of instructions to use for developing an optimal code.

Drawbacks

Even though skeleton-based systems offer the above-mentioned benefits, their major disadvantage is that not all functions can be realized using them. Some functions need additional skeletons other than the existing ones in skeleton-based systems. For example, recursive functions are not possible with the mentioned ones above. Additionally, a given skeleton function can be transformed for optimal execution only on a single target at a time. This meant we had multiple code-optimized variants present for a single function across the different back-ends. This ultimately leads us to the problem of handling execution across co-processors. In such cases, we have to manually write our code and optimize it by hand.

To sum up, a skeleton-based system offers a limited set of skeletons to a developer who transforms into an optimal code for a target hardware. Additionally, some newer systems compile their skeleton functions on-the-fly, so that more co-processor-aware optimizations can be applied (more about these are explained in Chapter 3). Though

these systems reduce overhead for developers while supporting optimal execution, they still lack support for cross-device execution. For such cases, task-based systems are used.

2.3.2 Task-Based Systems

Since a plethora of SDKs are present, one can write multiple variants of a function across these SDKs. Availability of multiple variants[12] for a function that increases the difficulty of developing and maintaining software. Designing variants-rich software is highly complex, resulting in a dense software package[181]. To alleviate these difficulties, task-based runtime encapsulates variants of a function using high-level abstractions called *tasks*.

Since a task has predefined characteristics, it allows a user to incorporate their custom code into an existing task-based runtime. The runtime executes such custom functions implicitly, managing the data and functional dependencies. In addition to functional abstraction and dependency resolution, these task-based runtimes optionally also support task scheduling (or work stealing), task partitioning, synchronization, etc. Hence, in addition to concurrent execution, task definitions encapsulate multiple variants of a function across different co-processors—which is of interest in this work. Furthermore, the functionality of these runtimes (like scheduling, and data management) is closely tied to the underlying environment and use case under consideration. For example, task partitioning is useful with HPC workloads as they are compute-intensive[135]; whereas such execution leads to an additional synchronization phase with database workloads [49].

Moreover, a task manages only implementation variants and is not aware of its target devices explicitly. Thus, a developer typically provides the following to run a code in a target device: execution device/platform definitions, execution prerequisites, and then the kernel(s) itself. Though these are necessary for executing the target function, the same definitions and prerequisites can be shared across multiple functions, leading to a lot of code redundancy.

Case for Task-Based System

Elastic function is a task-based system that supports plugin implementation variants for a function via elastic function [186]. In the example Figure 2.9, we illustrate how the elastic computing framework allows a user to include multiple sort implementations through specialized adapters (their notion of a task). These adapters (bottom left) define the signature for a function, through which multiple variants are added into the runtime. In our case, the sort_adapter() allows a user to integrate multiple sort implementations (like sort_random, sort_sorted, and sort_reverse) using the IO description. Once these implementations are added, the elastic functions runtime picks the right variant for optimal execution depending on the current context. In this case, the runtime picks a variant based on the underlying data distribution.

[12]or flavor. Both keywords are used interchangeably. We use variants throughout this work.

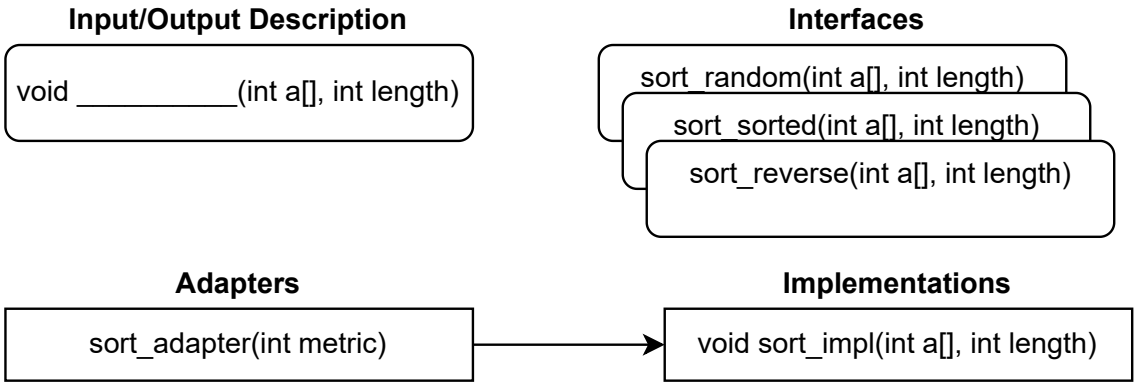


Figure 2.9: Example of variant handling offered by elastic functions

Drawbacks

On the one hand, task-based runtime offers abstractions—in the form of task definitions—for managing cross-device implementations, thereby easing implementation effort. On the other hand, it also needs additional rework to be aware of the execution environment. Specifically, a user is given the flexibility to implement a custom variant for a task pre-defined within the runtime. However, the functions within the runtime and the supported tasks are defined by the task-based system itself and cannot be easily updated. Specifically, these task-based systems only support a pre-defined set of hardware out-of-the-box and only variants can be developed for these by an end-user. Thus, one of the main drawbacks is that task-based runtime is not aware of both the workload and the underlying hardware.

Overall, both these paradigms support code management through abstractions, yet they have their inherent limitations. To support a query execution over various co-processors, we need an abstract runtime that has holistic support for query execution, which is not currently supported through these existing paradigms. Hence, we explore such a system that allows for plugging-in functions that are written for different co-processors. We focus on a holistic system that supports query execution, either for a single device or the system that facilitates runtime handling of executing operations across different devices using multiple data chunks.

To develop such a system, we must be aware of the current pitfalls present in query execution concerning cross-device systems. In the next section, we briefly discuss the various challenges of DBMS in a heterogeneous hardware system.

2.4 Challenges in DBMS with Co-Processors

The heterogeneity in a DBMS requires a highly adaptable system. This system must have all the necessary interfaces for adding new functionalities with fewer changes to the overall system. Though an adaptive DBMS provides various benefits, it also comes with additional complexities. To have a DBMS adaptable to both changing hardware and software, the following challenges have to be addressed.

2.4.1 Device Features

Adding DBMS operation to a new processing device requires novel ways to exploit the device without compromising the overall system design. Hence, one of the major challenges is to reorganize the processing functions based on the hardware features available and adapt to the underlying functions for efficient execution in the device.

We have already established that multiple devices have been developed for increasing the execution speed of specific functions. They require the DBMS operations to be modified for usage. This requires novel ways to exploit the device without compromising the overall system design.

2.4.2 Abstraction Hierarchy

We have already mentioned that database operations can be sped-up using device-specific parameter tuning [41]. However, adding this extra layer of optimization leads to an optimal operator for only the underlying device and cannot be ported to others. Also, these parameters must be tuned over every new device in-order to get better result. We see more on device-based parameter tuning in Chapter 4. Though parameter tuning leads to optimal execution, it also hinders performance portability. Due to this polarity in abstraction versus specialization between functions and devices, it is required that we find a good abstraction level for the operations that provides both an interface to write new functions and also exploits the hardware for optimal efficiency.

2.4.3 Parallelism Complexity

The growth of DBMS at both functional and hardware levels provides various parallelization opportunities. The presence of multiple devices creates an additional paradigm: cross-device parallelization. Using this type of parallelism, the given query is divided into granular parts based on the level of abstraction selected, and these functional primitives are distributed among the different processing devices for parallel processing. We detail the different types of parallelization in the subsequent sections.

Functional Parallelism

In multiple instances, the incoming queries have various sub-operations that run independently of each other. One common example is the availability of multiple selection predicates combined using logical operations. These predicates can be executed in parallel among the different devices, and the results are combined in the next steps. Thus, identifying and dissecting these parallel operations provides additional capabilities for simultaneous execution in the form of functional parallelism. The major challenge in this parallelism is the intermediate step of materialization of the results to be processed in the next operator in the pipeline. There is also a synchronization overhead present in this parallelism due to the differences in the execution time for different processing devices.

Data Parallelism

In contrast to functional parallelism, data parallelism does not split an operation into different functions, but executes the same operation on different partitions of the data concurrently. This method also has a similar synchronization overhead of waiting for all the devices to finish processing. The major disadvantage of this parallelism is the additional step to merge results from different devices.

Cross-Device Parallelism

The above-mentioned functional and data-level parallelism are decided after the selection of processing devices. As we mentioned earlier, each device has its perks and must be utilized to the maximum extent. Hence, it is necessary to decide on the implementation details for the given device that exploits the hardware for efficient execution. Moreover, the above-mentioned parallelization strategies can also be realized at the device level. In terms of device-level functional parallelism, it could be multiple operators running in parallel in different devices or a pipeline with communication within the devices. Similarly, data parallelism could also be realized via suitable cost functions for operations on devices.

2.4.4 Optimization Strategies

The different levels of parallelism for the execution of a query provide additional opportunities for fine-tuning the operations, but have the complexity of selecting the optimal execution path. As the decision of top-level parallelism influences the subsequent levels, the selection of the right execution path for a given query is critical. However, the important drawback of this multi-level parallelism model is the search space explosion. There are various options available for any given level, thereby having multiple combinations in total for selection. This search space of parallelism has to be traversed for finding the optimal execution path. Deciding the optimal path of a single operation in a query can be complex (e.g., join order optimization) which in addition to new dimensions of multiple devices increases the complexity further. Hence, newer methods for exploring the various optimization opportunities are to be determined.

2.5 Opportunities for Query Execution

The previously mentioned challenges can be solved using a new DBMS architecture that effectively handles diversity in terms of both functionality and underlying hardware features. Hence, based on the challenges, we have identified the key areas for designing an adaptive DBMS.

Example 1 (TPC-H Query 6)**.**

```
SELECT SUM(l_extendedprice * l_discount) AS revenue
FROM lineitem
WHERE l_shipdate >= '1994-01-01'
AND l_shipdate < '1995-01-01'
AND l_discount BETWEEN 0.05 AND 0.07
AND l_quantity < 24
```

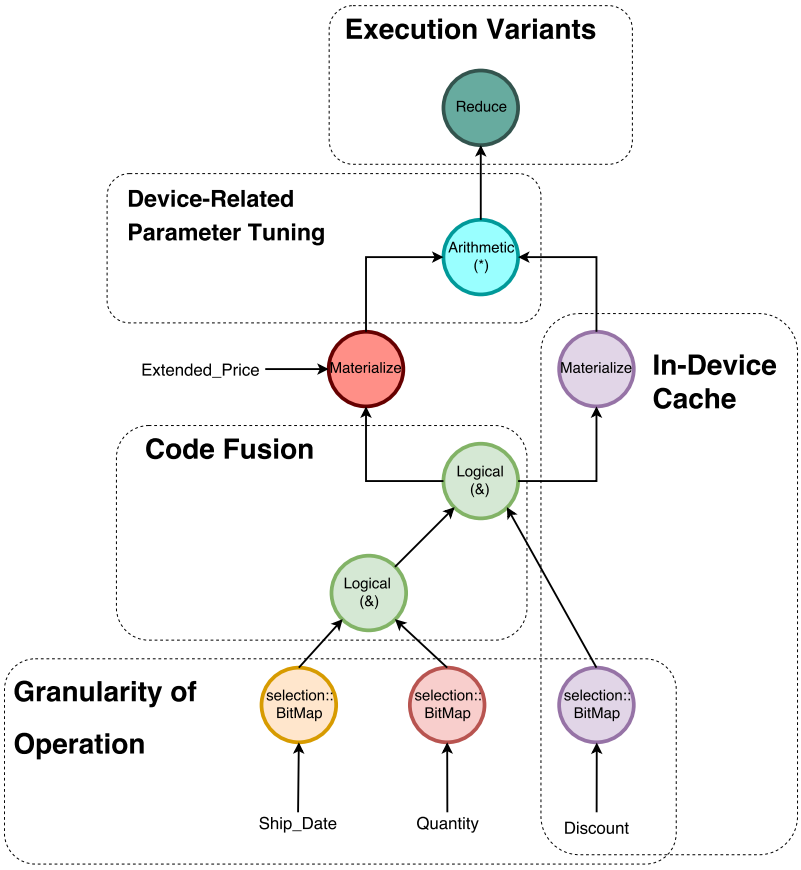


Figure 2.10: Possible cross-device optimization strategies using query execution plan of TPC-H Q6

To better explain these design options, we use the execution plan of TPC-H Query-6 as our motivating example (see the example above). The query reads data across multiple columns, filters them, multiplies the results, and outputs them after aggregation. These three operations are in turn executed using multiple granular primitive functions. The different primitives used for processing the given query are:

- **Selection** primitive selects the values from the given column. Bitmaps are used as output formats to reduce the data transfer size, as each bit carries the selection information of a single value.
- **Logical Operation** primitives perform logical functions on the bitmaps produced by the different selections.
- **Materialize** extracts the selected column values using the given bitmap input.
- **Arithmetic** performs arithmetic operations over column values.
- **Reduce** performs aggregation of values.

The execution of the query using the above presented primitives is given in Figure 2.10. The figure also illustrates different optimizations that can be done for a simple query like in the figure. We discuss the various optimization strategies in the subsequent sections.

2.5.1 Granularity of Operation

One of the main challenges in the proposed adaptable system is the level of granularity required for optimized processing. Based on the capabilities of the devices, we could either run a few complex operations or split them into more granular sub-operations, thereby executing them in parallel across different heterogeneous co-processors.

At the top level, each database operation acts as a set of primitives connected to provide a final result. The more granular a function is split, the more hardware sensitivity comes into play. For example, the access patterns in CPU and GPU are different for an optimal execution (i.e. sequential or coalesced access). Further, database operations are data-centric where every operation is applied to a massive amount of data. To aid parallel data processing, we propose the use of explicitly data parallel primitives to be combined into complete DBMS operations. There are many works on primitive-based DBMS query processing. He et al., propose multiple primitives such as Split, Filter, etc., for GPUs [87]. Other primitives such as prefix-sum and its variants, scatter, and gather are also proposed for efficient data-parallel execution [58]. This approach provides a fail-safe: when a newer device is added, the primitives could still run on them with minor changes to the functionality. This availability of different granular levels provides additional benefit, enabling a developer to replace the inefficient fine-granular primitives with custom coarse-granular ones.

2.5.2 Code Fusion

Implementing primitives in multiple granular levels becomes time-consuming. Hence, code could be generated at runtime for the given granularity level of the operation. This code for execution in an individual device is generated by combing the primitives for the corresponding device into a single execution process. This reduces the overhead of materializing data from intermediate steps. An extreme case for such code fusion is query compilation [131].

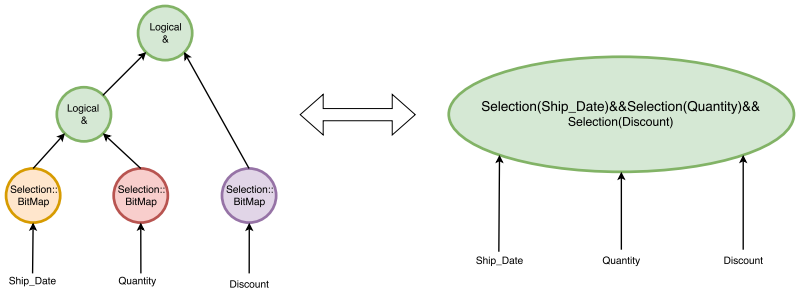


Figure 2.11: Example of code fusion using multi-column predicates

For example, three selection predicates as shown in Figure 2.11 can be either run on different devices (left) and the results are combined using the logical operations, or the predicates are all combined into a single execution (right).

2.5.3 In-Device Cache

The current data-transfer bottleneck is between the main memory and the processing devices themselves. The CPU has faster access than other devices as it is directly

linked to the main memory, whereas in the case of the co-processors, data must be transported via connections with higher latency and possibly more limited bandwidth, such as PCI-Express. Thus, even highly efficient GPUs can have suboptimal performance than CPUs due to limited access capabilities to main memory. Hence, using device memory as a data cache is crucial for high compute throughput. In contrast to this, these external devices have limited memory. Hence, it is not always possible to store all the necessary data on the device itself. Thus, the host system must determine the hot set of data to be stored in the device memory using the execution plan for the given query and monitoring the data transfer to the device.

2.5.4 Execution Variants

Each primitive selected for executing a given query can have different characteristics to choose from based on the executing device. For example, complex branching statements are handled efficiently by CPUs, whereas GPUs are capable of massive thread-level parallelism with less control flow. In addition, the data access pattern must be selected from the memory architecture of the given device. For example, coalesced data access provides efficient memory access in GPU. Finally, hardware-specific vectorization of DBMS operations (SIMD) is also an important parameter in database processing to exploit the hardware capabilities.

Also, on an abstract level, characteristics of the primitive itself can affect system throughput. The choice of output format and the number of intermediate steps are some of the characteristics that influence the overall system. For example, using bitmap results from selection in external devices will be generally more efficient than transferring complete columns.

2.5.5 Device-Related Parameter Tuning

Finally, once we have decided on the device and its corresponding function to execute, certain device-related parameters like global and local work group sizes have to be tuned for further improvement of the overall efficiency. These device-related parameters are tuned for efficiency by monitoring the performance of execution. There is a feedback loop from the devices, providing execution specific information used for tuning the primitive for higher efficiency.

On top of these above-mentioned challenges, one of the major challenges is to formulate an order for using the strategies to extract an efficient execution plan. Since all the strategies mentioned above are inter-dependent, the selection of one influences the other. To have a standardized execution flow, we propose an architecture that has all the necessary components for using the above strategies.

2.6 Summary

The field of co-processor acceleration gets wider with new processing architectures, each trying to accelerate a particular application. However, the co-processors also have SDKs on top to custom-write a target function to be executed on these new architectures. In such a growing ecosystem, we have the major challenge of testing whether a new device will fit our use case in hand—accelerating query execution.

The consensus is to either develop a hardware-oblivious executor that internally translates a user-written program into a target device or to write a hardware-aware executor that is written for a particular device. Though these two models are shown to be beneficial, they are not extendable with every new device and SDK present in the market. Therefore, the alternative is to have a system that handles multiple implementations and devices through a common abstract interface. To this end, many systems have been proposed under either skeleton-based or composition-based paradigms, out of which the latter fits our use case.

Now, to develop an abstract co-processor pluggable query executor, we must be aware of the various pitfalls when developing a cross-device query executor. To this end, we described the different challenges in terms of query execution such as operator granularity, runtime code fusion, code variants, etc. To overcome these challenges, in this thesis, we explore a query executor that is abstract enough to plugin various implementations yet allows a user to write optimal code.

In summary, we have presented the ecosystem for co-processor acceleration. Starting from different co-processor architectures, we have also seen the different support for programming and execution across such a heterogeneous environment. There are also various options to get better performance out of a device. Such a big options landscape makes it hard for a programmer to identify the optimal setup. Hence, it is being widely studied by various researchers to understand the best use of the ecosystem for optimal query execution. We review these state-of-the-art techniques in the next chapter, followed by our contributions to developing the abstract architecture in the later chapters.

3. Existing Unified Runtime

The previous chapter showed the different components developed to ease development over a heterogeneous hardware environment. Specifically, we have various APIs to have optimal implementation for a co-processor. However, as an implication, we now have the challenge of manually testing our implementation over these APIs, which is time-consuming. Alternatively, various runtimes are proposed to ease the implementation effort. We briefly discuss these existing systems in this chapter and show the need for a dedicated unified runtime for the query engine.

Using external accelerators for improved performance is becoming a common practice, but a plethora of programming and optimization strategies make their integration challenging. Specifically, integrating accelerators into an existing DBMS requires the developer to take care of a multitude of issues – like explicit data partitioning, data placement, optimal data transfer, synchronized execution, etc. Thus, even writing a simple operation for a target co-processor requires complex re-working of the existing query engine, which does not directly contribute to the performance of the operator. Therefore, it is necessary to study these challenges in developing a query engine that better exploits the underlying co-processor with less re-work.

Other than databases, High-Performance Computing (HPC) also uses co-processors for speeding up execution. HPC workloads are mostly computation-bound, unlike DBMS operations, which are data-bound. Hence, there are various runtimes proposed in HPC that support compute-bound workloads. We discuss these runtimes in detail w.r.t. query execution and show the need for an informed pluggable query engine.

We split this chapter into two main sections. First, we review the existing database systems tailor-made for various co-processors. Next, we detail the different unified runtimes that support easy co-processor integration. We split these runtimes into two based on the paradigms they support: component-based and skeleton-based systems (from Section 2.3). The component-based system requires a developer to provide the handwritten program for each of the devices in their supported platforms, and the runtime decides on the placement of these operations. On the other hand, skeleton-based systems require the user to provide their required execution in a

pseudocode-like construct given in a domain-specific language. The runtime then takes care of generating the optimal execution code for the underlying device. Both paradigms have different optimization opportunities, but both expect the user to provide information about the data manipulation operation and hide the information about the runtime. Let us now see the database engines developed over different co-processors.

3.1 DBMS On Co-Processors

One of the principal goals of a database engine is to have a fast response time. This is partially achieved for CPUs, with the query engine being aware of the underlying hardware characteristics [31]. As an extension, query engines are also developed that exploit the characteristics of various co-processors. These solutions range from a tailor-made operator implementation to a complete query engine developed for a particular co-processor. Here, we cover the systems that provide an extensible query processor that can be added with new co-processors. A more comprehensive overview of the query engines over appropriate hardware is presented in various co-processor-specific surveys, like [37] for GPUs or [67] for FPGAs.

The use of GPUs as DBMS accelerators is still under active research. Early implementations of GPU-based DBMSes supported partial query execution with dedicated operators implemented for the hardware. Solutions like GPUTeraSort [78] have dedicated implementations for individual database operators. However, with advancements in GPU programming, many alternative implementations have been developed to support various other operators. Furthermore, with similar advancements in programming options, we have systems that allow runtime code generation that compiles a query into the target device like Hawk (detailed later in Section 3.2.1) and Perseus [155]. These are the popular solutions currently available for query processing over a co-processor. More related work is discussed in each chapter in detail. Complete query engines are proposed for GPU like, GPUDB [87], OmniDB [192] etc. to have full-fledged query execution over the device. Early solutions like the ones from Govindaraju et al. [78] develop single table query processors successfully over GPUs. Similarly, GPUDB utilized the primitives to develop a CUDA-based solution for GPU. Here, they support both CPU and GPU-based query execution using these primitives. Various other solutions like [15], and CoGaDB [37] also support query execution over GPUs.

Other specialized devices like tensor cores are also equipped for query execution. Solutions like the systems by [93], TQP [11], TCUDB [96], and solution by [89] are some earlier exploration of the performance of tensor cores for query processing. Again, these systems explore the specialized capabilities of the device to increase query execution performance. The operator implementation and device calls within these systems can be extracted and integrated within our pluggable query engine.

Finally, FPGA-based solutions like Drewes et al. [60] and ReProVide [19] develop a complex infrastructure to support query execution within the device. These solutions develop a framework where other FPGA-based operator implementations can be included to support partial pluggability. These are a subset of the plethora of other works to support query processing over existing co-processors [37, 154, 67]. However,

these are all dedicated solutions that fit a particular co-processor, and they explore query execution using these architectures. When it comes to integrating these devices into query execution, they develop a dedicated hardware-aware solution (except Ocelot, which is hardware-oblivious supported via OpenCL [91]). Alternatively, there are also runtimes proposed that support plugging co-processors using minimal code rewrite. Some of these runtimes are reviewed in the upcoming section.

3.2 Existing Abstract Runtime

Above, we saw query engines tailor-made for individual co-processors. However, the current processor landscape is rapidly changing as well as there is a quick line-up of co-processor generations [25]. This makes it hard to manually develop a query engine on top of individual processors. Hence, in this work, we explore an abstract runtime that supports a unified runtime for easy co-processor integration. To develop such a system, we first review the various existing work and their features.

Many research projects tackle the challenge of co-processor integration to enhance productivity. Projects like PEPPHER [25], SARC [146], HyVM [163], Apple-Core [145], Sequoia [71], RapidMind [48] have developed solutions for various aspects of co-processor integration like scheduling, library integration, auto-compilers, memory hierarchies, etc. However, among these challenges, a common runtime has been ubiquitous across all these solutions. Hence, it is beneficial to review these existing solutions and contrast them with our work. We divide these systems into component-based and skeleton-based solutions (as mentioned in the previous chapter - cf. Section 2.3).

3.2.1 Skeleton-Based

These are a class of solutions, that speed up implementation time using an alternative programming model, simplifying the low-level processor APIs into DSLs (Domain-Specific Languages). These DSLs comprise skeletons commonly representing the *algorithmic patterns* present across commonly written functions (like loops, map, reduce, etc.). A user can write their custom function using the DSL, which is translated into hardware-aware implementations (possibly with necessary code optimizations). Thus, such DSL-based or skeleton-based solutions support portability as well as abstraction while enabling hardware-aware implementation.

Solutions like SkePU and its versions [63, 66, 65], Muesli [113, 64], SkelCL [173], FastFlow [56], SkeTO [62], Skandium [119], OSL [117] are the most popular general-purpose solutions present. They support generic constructs to realize any functional implementation. On the contrary, these systems lack the domain-specific information to tune for better performance. Thus, as an extension to these, several application-specific skeletons are proposed that leverage the application characteristics to improve performance. In our case, DBMS-based solutions like kernel-weaver [189], Hawk [39], flounder-IR [74] and its extension ReSQL [75], HorseQC [73], voodoo [140] are some of the solutions with skeletons that are aware of and exploiting query characteristics for better performance. Though many such runtimes are present, only a few directly relate to our use case. These related runtimes and their features are listed in Table 3.1.

Name	Supported skeletons	Written in	Supported devices	Data structure	Domain
SkePU [63]	map, reduce, mapreduce, mapoverlap,maparray	C++	CPU, GPU	Vector	Oblivious
Muesli [113]	map, zip, fold, mapIndexInPlace, permutePartition, farm, pipeline, divide&conquer, branch&bound	C++	CPU, GPU	vector, matrices, distributed array, distributed matrix	Oblivious
SkelCL [173]	map, zip, reduce, scan, mapoverlap, allpairs	OpenCL	multi-GPU	single, copy, block, Overlap	Oblivious
Kernel weaver [189]	project, product, select, set and join	CUDA	GPU	vector	DBMS
Hawk [39]	loop, filter, hash_put, hash_probe, cross_join, arithmetic, aggregate, hash_aggregate, project	CUDA	GPU	vector	DBMS
HorseQC [73]	select, project, aggregate, group-by, prefix sum, aligned write	CUDA	GPU	vector	DBMS
Voodoo [140]	load, persist, bitshift, logical, zip, project, upsert, scatter, gather, materialize, break, foldselect foldmax/min/sum/scan, range, cross	C++ & OpenCL	CPU, GPU	Structured vector	DBMS

Table 3.1: Common characteristics of skeleton-based systems

As the table summarizes, skeleton-based systems support only dedicated devices. These runtimes extend their own DSLs to later extend them with device-specific details for optimal performance. Below is a detailed explanation of individual systems.

SkePU: Modelled based on BlockLib [5], SkePU is a template library written in C++ that supports CPU (via C++ and OpenMP) and GPU (via CUDA and OpenCL) architectures. It abstracts memory management and handles it internally to avoid unnecessary data transfers. It exhibits different skeleton functions, as well as a container - vector for data, and supports user-defined functions via macros. The skeletons present in SkePU are: `map, reduce, mapreduce, mapoverlap, and mapArray`.

Furthermore, switching between target devices must be stated explicitly (The default target being a single-threaded CPU). Such targets are defined using SKEPU_*** within the existing code. The decision of the best execution platform is based on computation type, data characteristics, and system architectures. The runtime is written in C++, modeled after BlockLib for the IBM cell. It provides skeleton definitions supporting a single container implementation modeled after a vector (similar to an STL vector) container. This container is used to hide GPU memory management and uses lazy memory management. The processed result is copied back only when the host side needs to access this data. Additionally, the framework also provides an additional member function flush, which updates the vector from the device and de-allocates it from device memory.

Muesli: Muesli is short for the Muenster skeleton library. It is also written in C++. It supports multi-core CPUs via OpenMP and GPU via CUDA. In addition to algorithmic skeletons, it also supports data parallel and task parallel skeletons. Similarly, in

addition to vectors, they also support matrices as well and distributed data structures: distributed array and distributed matrix. The library offers data parallel skeletons like `map, zip, fold, mapindexinplace, permutepartition`. While task parallel skeletons include `farm, pipeline, divide and conquer` and `branch and bound`. Using these semantic definitions for skeletons, the system can effectively schedule execution.

SkelCL: SkelCL is written as an extension to OpenCL. Similar to the ones above, SkelCL also supports distributed data structures with four distribution types: single, copy, block, and overlap. Data within these distributions are copied according to the target device. They support six skeletons: `map, zip, reduce, scan, mapoverlap, and allpairs`.

Similar to the ones above, many other skeleton-based runtimes exist over different co-processor environments. However, as mentioned before, these are general-purpose systems that are not aware of the characteristics of execution. Unlike these skeletons, our ADAMANT system proposes query-aware primitives that represent a specific database operator. Hence to have better awareness during runtime, skeletons are developed based on particular use cases. Some of the skeletons that are developed specifically for DBMS are discussed below.

Kernel-weaver: Kernel-weaver is a DSL developed for Datalog queries[13]. The system is specifically developed to generate kernels that exploit the threading hierarchy of a GPU. However, unlike their system, our ADAMANT system can freely support any co-processor, trading off the implementation to the end-user. Hence, we can include kernel weaver for GPUs into our ADAMANT system.

Hawk: The major goal of HAWK is to auto-generate efficient code for the underlying hardware. To this end, it uses a minimal set of intermediate skeletons coupled to form a data processing pipeline - known as pipeline operations. These pipeline functions are translated into an efficient pipeline program that is then executed on the target device. Furthermore, the HAWK system also has a set of rules for defining the pipelines, where the first operation is a loop and the end is a project or hash_build or aggregate - or in short, a pipeline breaker. Our ADAMANT complements Hawk with similar primitive characteristics. We can directly integrate Hawk into our ADAMANT system to execute a query pipeline within a single co-processor.

Voodoo: Voodoo's IR represents the execution of an SQL in DAG format. The DAG clearly shows the different intermediate data being reused in different places of execution. For controlling parallelism, voodoo has come up with the notion of controlled folding and intent, an extent relationship where the number of inputs processed in a single thread can be controlled. Once the IR is devised, either by an end-user or by a parser, the runtime goes over the graph and creates bundles of operations that belong together in a single kernel. This is identified using the values of controlled folding and intent, the extent given to the operations. These isolated kernels are then generated together and executed. There are multiple data parallel patterns used by the voodoo framework for execution. Similar to the case of Hawk, we can also integrate Voodoo primitives within our ADAMANT to execute queries within a single co-processor.

[13]A declarative language that allows one to express database queries as first-order logic

Modular Extensible Compilers

These are a subset of skeleton-based systems that exposes a limited set of API that can also support plugging user-defined implementations. These frameworks support a modular API that can be combined to form custom skeletons, which are subsequently compiled into target code for a co-processor. These systems can be coupled with our ADAMANT framework to support runtime compiled execution on a target co-processor. Here are some of the popular systems in detail:

Delite [175]: Delite is an extended form of Lightweight Modular Staging (LMS) [152], that allows for multi-stage program generation. It allows an end user to use library calls as generative skeletons. Instead of a DSL for a particular domain, Delite comes with a general infrastructure for developing custom DSLs. This is supported by splitting the underlying compiler into multiple components. Delite supports multiple domains transparently using the internal components of the compiler. However, the DSLs developed as part of the Delite framework support GPUs and CPUs only.

PetaBricks [6]: PetaBricks is a compiler that supports plugging multiple implementations of an algorithm. The runtime auto-tunes implementations based on data distributions, and algorithm parameters. The system bridges the gap between skeletons for algorithm representation and pluggable task-like functions with alternative implementations. Thus, the system supports both auto-compilation to get the best implementation along with the portability of implementations with less re-work.

Qilin [120]: Qilin automatically maps computation to the right processing element in a processor. It also updates this mapping based on runtime characteristics. Qilin has various API options - such as stream and thread APIs - for users to program their functions. It also has two data types: QArray and QArryList to ship data. Once the user-defined function is ready, the runtime parallelizes it and compiles the code into either thread TBB or CUDA code - two of the currently supported low-level programs.

Limitations

However, with all these systems, the support for alternative co-processors is limited. Most of these systems support CPU, GPU, or a hybrid CPU-GPU system. To support a new system, its instruction sets must be adapted to fit the skeleton characteristic. Furthermore, we see very limited support for cross-device execution. Specifically, these skeleton-based systems focus on improving performance on a single device. Hence, we still need these skeletons to be integrated across multiple devices to have performance from a heterogeneous processing system. For such a case, component-based runtime is used. Plugging alternative implementations is one aspect of this component-based runtime. They also contain other aspects related to optimal execution like data management, task scheduling, etc. We review some of these runtimes and their attributes in the next section.

3.2.2 Component-Based

Component-based systems, or commonly, task-based systems map high-level language constructs to user-written low-level implementations [23]. These systems expose

rigid task definitions that are realized by a developer. These systems handle the execution of these tasks, thereby striking a balance of express-ability as well as productivity through abstraction [23]. Similar to skeleton-based systems, depending on the level of support, various systems are proposed with their characteristics. Based on these characteristics (architectural - memory hierarchy, processor type, process management - work handler, resilience, etc.), Thoman et al. [178] have come up with a brief taxonomy of these task-based runtimes. The work lists some of the common general-purpose task-based runtime systems like PaRSEC [32], starPU [14], pipeflow [47], CPP-Taskflow [97]. These systems support the inclusion of any arbitrary task and execute them across hardware devices. However, as with general-purpose skeletons, these systems are also oblivious to the domain-specific characteristics and cannot exploit execution. We will first review some of these systems in detail and show the need for a query execution-aware unified runtime.

Name	Functional interface	Written in	Supports
StarPU [14]	codelets	C++	data management, scheduling
PaRSEC [32]	task	C	data-flow based scheduling
CPP-TaskFlow [97]	taskflow	C++	scheduling
Elastic computing [186]	elastic functions	C++	portability

Table 3.2: Common characteristics of component-based systems

StarPU: StarPU is one of the most popular runtimes specifically designed for data management. The runtime is introduced with a user-defined function via a `codelet` [13], which contains IO definitions of the function. The runtime uses this information to handle the subsequent memory management and data transfer functions internally. Additionally, the runtime also supports various scheduling mechanisms out-of-the-box like work stealing, eager (centralized queue), random, priority queue, HEFT [14]. Any special user-written schedulers can also be included in the StarPU runtime. Though the system supports handling multiple co-processors, it is application-agnostic and schedules tasks generically. Moreover, the system is designed to support compute-heavy tasks, which is not suitable for DBMS workloads.

PaRSEC: PaRSEC also handles data management tasks internally, while a user supplies the functional implementations. However, PaRSEC uses a custom data-flow model in the form of a directed acyclic graph (DAG) for scheduling tasks across heterogeneous hardware. Here, the relation between different tasks is mapped by a user to determine the overall runtime characteristics. Additionally, a user can also add execution hints (like data distribution) for informed task scheduling. Scheduling in PaRSEC is `event-driven` i.e. the runtime schedules after certain events (like task completion, data transfer), thus supporting lazy evaluation of task placement. This enables a more fine granular approach towards scheduling and placement. Again, the system is application agnostic, and execution characteristics are not exploited for better co-processor integration.

CPP-TaskFlow: It is also an abstract runtime that enables users to write parallel programs using task dependencies. The approach supports various levels of parallelism:

[14] More details given in:https://files.inria.fr/starpu/doc/html_web_basics/Scheduling.html

loop-level, functional level (graph placement), pipelined parallelism (incremental flows), as well as supports dynamic data structures. A user can introduce their executable functions via `Taskflow` object. The object internally hands over the tasks for execution to a unified runtime. It also gives users the ability to do static scheduling. Based on the schedule, the runtime executes them using work-stealing. However, this is limited to a CPU itself and cannot abstract various task implementations across different hardware.

Elastic Computing: The framework supports generic functions named as `elastic functions`. These functions have a single generic signature with alternative implementations across various processors. Using this information, the runtime does inform the scheduling of a target function. Though the work closely follows ours, it also has multiple shortcomings. Any co-processor-related implementations must be explicitly written by the user, without any reusable components. Also, similar to all of the above cases, the system is oblivious to the application characteristics, which requires the end-user to implement redundant functions.

Limitations

Though we have several runtimes that handle cross-device execution, they cannot fully support query execution due to their application-agnostic execution. Moreover, these systems mostly favor a compute-heavy workload, as the workload can benefit directly from heterogeneous hardware resources. Finally, the systems focus on plugging in an implementation, which means tightly coupling device drivers to the target function itself. Hence, to avoid these challenges and support a holistic query execution across any arbitrary co-processor, we need a query engine runtime that is both pluggable and aware of query execution features.

3.3 Summary

Overall, there are various DBMS implementations available across co-processors, each tuned towards improved execution. However, these are tailor-made solutions that fit only a particular device and cannot be extended with new ones in the future. To overcome this challenge, we propose an abstract runtime is proposed to support pluggability. This runtime increases productivity by abstracting either device-specific or runtime details. We saw some of the commonly available runtimes that either abstract underlying device features or runtime features. Though these systems improve productivity, they are too generic and do not extend query execution functionalities. Hence, there is still a gap in terms of a runtime that supports the pluggability of new hardware as well as is aware of query execution to better exploit it for efficient performance. In this work, we go about developing such a pluggable query engine. The query engine supports plugging devices without additional re-work and still executes the query without performance degradation.

As a first step towards developing such a system, we must study the need for hardware-aware programming in terms of performance. Specifically, we must understand the impact of hardware awareness in DBMS operators as it directly reflects on the overall query execution performance. To this end, we perform a case study experimenting with various hardware-aware options/implementations to understand their implication on group-by execution. Our case study is presented in the next chapter.

4. Tier 0/1: Crafting a Co-Processor Aware DBMS Operator

Database operators written based on the underlying hardware architecture have been shown to improve performance many-folds by numerous researchers [142, 41, 22, 80, 182]. Such benefits come from meticulously programming the operator with instructions that act over various hardware components of the underlying device [103]. A programmer must consider the order of execution to ensure that the instructions are executed seamlessly without encountering any hiccups in the execution chain of the underlying device. Hence, it is hard to code-optimize an operator for an underlying hardware. Therefore, we must understand the complexities of developing a hardware-aware database operator before we go about architecting an abstract query executor. To this end, in this chapter, we perform a case study developing sort-based aggregation to be GPU-aware.

Parts of this chapter have been based on the following publications:

- B. Gurumurthy, D. Broneske, M. Schäler, T. Pionteck and G. Saake, "An Investigation of Atomic Synchronization for Sort-Based Group-By Aggregation on GPUs," IEEE 37th International Conference on Data Engineering Workshops (ICDEW), Chania, Greece, 48-53(2021).
- B. Gurumurthy, D. Broneske, M. Schäler, T. Pionteck and G. Saake, "Novel insights on atomic synchronization for sort-based group-by on GPUs". Distributed Parallel Databases 41, 387–409(2023).

4.1 Need for HW-Awareness in Group-By

A simple group-by aggregation as shown in Example 2 is normally implemented either using hash-based or sort-based techniques [16]. We focus on the latter and implement a hardware-aware sort-based aggregation for current-generation GPUs. The rationale for studying this problem is twofold. First, compared to other database

operations (like joins) group-by operations are less affected by the data movement problem. The data movement problem occurs whenever data is shipped to or retrieved from a heterogeneous processing device. This may incur a major cost factor [9, 18, 36]. Secondly, computing the grouping and aggregate is highly compute intensive [30, 84, 22], and thus a perfect use case for parallelization.

Example 2 (SQL query with a simple grouped aggregation clause)**.**

```
SELECT COUNT(*), l_returnflag FROM lineitem
GROUP BY l_returnflag
ORDER BY l_returnflag;
```

Massively parallel grouping and subsequent aggregate is challenging – independent of the processing device. The complexity arises when data of one group is arbitrarily distributed within the input and aggregating them together requires some kind of synchronization. Such execution over a GPU increases the difficulties, the device is not designed for efficient inter-thread communication. To resolve this, a GPU supports a specialized hardware component for thread serialization - enabled via atomic operations.

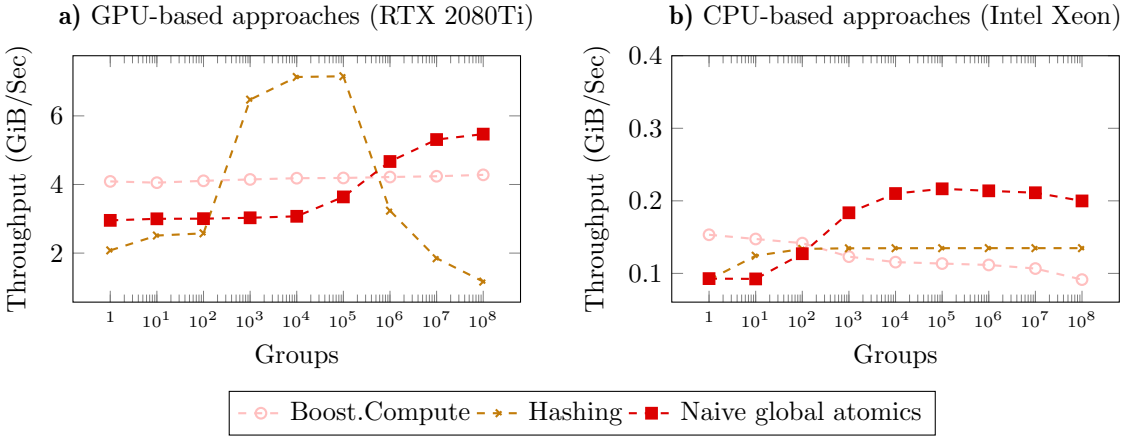


Figure 4.1: Throughput of different group-by approaches on a RTX 2080 Ti GPU and Intel Xeon CPU on 2^{27} integers with uniform random distribution. Note, the different scales of the y-axis.

Generally, group-by relies on sorting or hashing [87], with empirical results suggesting that hash-based approaches are generally superior [104, 22]. In Figure 4.1(a), we depict the throughput of a recent hash-based vs sort-based grouped aggregation (i.e., boost.compute). The results clearly show group cardinality impacts their performance. For reasonable group numbers between 10^2 and 10^6, hashing is best. For smaller numbers, boost.compute has the highest throughput. Adding a third approach, a naive sort-based aggregation using atomic operations (i.e., hardware-based), we observe that its throughput increases monotonically until each value is assigned uniquely to a group. From 10^6 distinct groups, it offers even the best performance. By contrast in Figure 4.1b, we depict the throughput when applying the same techniques to the CPU. We observe firstly that the performance pattern is entirely different, with the atomic-based approach being superior for a wide range of

group numbers. Secondly, the CPU versions are a magnitude slower, i.e., there is a substantial throughput benefit one can invest to move data to the GPU, in case it does not already reside there. Despite this remarkable result, our hypothesis is still that in current sort-based solutions, all threads aggregate data simultaneously and block each other. This is supposed to hold especially in the case of small group sizes. Hence, one does not fully exploit the massively parallel power that modern GPUs offer.

To this end, we first investigate if the synchronization overhead is a decisive bottleneck. Then, we aim to propose a solution that mitigates the issue, aiming at a throughput that is at least equal to – or even superior – to a hash-based solution or boost.compute depending on the number of groups. Our investigation has the following results and contributions:

1. An investigation of how the latest advances in GPU's architecture change the significance of our contributions w.r.t. state-of-the-art hash-based approaches. The key result is that the superiority of our atomic-based solution improves due to the larger number of available HW-based atomics processing components in the latest GPU generation.
2. We propose sort-based aggregation approaches that mitigate the synchronization overhead by reducing the amount of issued atomics. For instance, one approach requires 2 atomics per GPU thread independent of the data distribution. Afterward, we examine how the number of concurrent threads and chunk sizes affect the throughput of our approaches.
3. Our results suggest that atomics-based approaches are, in general, 3x faster than boost.compute and up to 2x faster than hash-based approaches for a reasonable number of groups, e.g., found in the TPC-H benchmark.
4. An examination of how different data distributions affect the performance of our contributions. The results suggest that the distribution has only a marginal effect, and thus our conclusions hold independent of the data distribution.
5. We put the GPU results into the context of results one can expect on present-day CPUs. The key insight is that our GPU-based solutions are on average by one order of magnitude faster.

The remainder of the chapter is structured as follows. We start with reviewing the related work in Section 4.2 Next, we explain the execution of atomics in a GPU in Section 4.3. Here, we present preliminaries on the execution of atomics particularly their performance. Afterward, we introduce several alternative approaches for using atomics for a sort-based group-by (Section 4.4). We first explain the three-step aggregation method, followed by our atomic-based approaches. In Section 4.5, we detail our extensive evaluation using microbenchmarks and a comparison of the full-fledged group-by-operator with state-of-the-art approaches. Finally, we summarize in Section 4.6.

4.2 Related Work

Since the usage of GPUs as general-purpose accelerators, many researchers use GPUs to accelerate DBMS operations. In the following, we list work that closely relates to our work.

Modeling performance of atomics: Hauck et al. propose to buffer atomic updates to reduce contention in a reduction [86]. However, their approach doesn't consider the different parameters affecting performance within the device. Similarly, Hoseini et al. explore the impact of atomics on CPUs [95] which is similar to our CPU-based system. We complement their work with a similar measure of atomics over GPU.

Sort-based aggregation on GPUs: Sort-based aggregation on a GPU was first devised by He et al. [87]. A similar method is followed by Bakkum et al. [15] using CUDA in SQLite. These are suitable for earlier GPU generations, where atomic operations are resolved via software-based serialization. These approaches follow a multi-step lock-free aggregation, which is costly compared to your direct atomic aggregation in modern GPUs.

Hash-based aggregation on GPUs: Alternatively to sort-based aggregation, hashing can be used for computing aggregates. Hence, there are several related approaches that tune hash-based aggregation for GPUs [22, 179].

Non-grouped aggregation on GPUs: Simple aggregation has the same execution pattern as grouped aggregation, where a single output location is accessed by all threads. To mitigate contention, there are various approaches [104, 115].

4.3 GPU and Atomic Functions

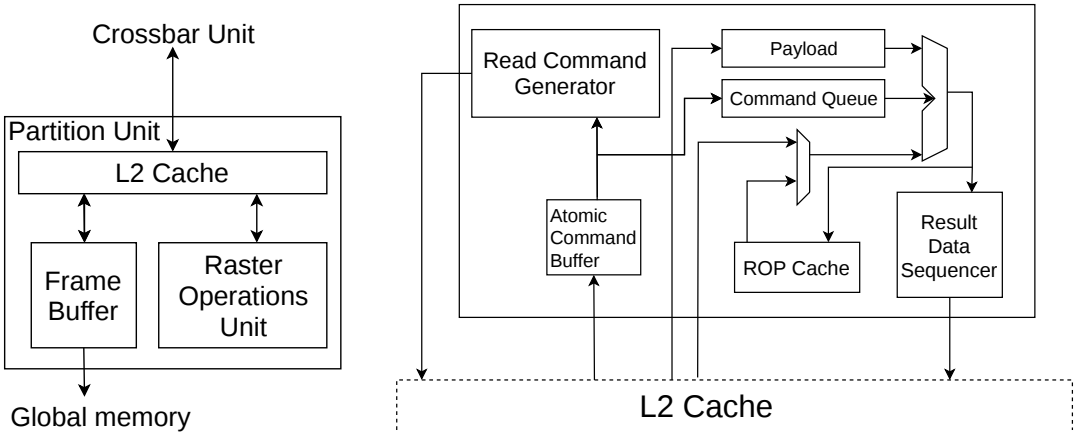
Although we have previously discussed the fundamental basic GPU architecture in Section 2.1, we will now go deeper into the architecture of the Nvidia 1050 Ti. This specific device is used in this section to examine hardware awareness and gain a better understanding of the hardware components that directly impact atomics. Specifically into the memory controller to study the flow of execution to resolve atomic operations. Specifically, we examine the components related to our hypothesis that sort-based group-by approaches suffer from the issue that all threads request synchronization simultaneously leading to lock congestion. To this end, we first investigate the various hardware components responsible for atomic functions. Note that, the newer generation GPUs have an updated version of this hardware-based atomics and would behave differently than the current implementation.

it is established already in Chapter 2 that GPUs favor better throughput instead of latency [133]. This is achieved using multiple Graphical Processing Clusters (GPCs), Memory Partition Units (MPUs), and an off-chip DRAM also known as the *global memory*. The cores access global memory and execute atomics over them using MPUs. In this section, we provide an overview of these components involved in atomic execution. Note, since the architecture of a GPU keeps varying, we explicitly refer to the work of Aamodt et al. and Glasco et al. [1, 77] for our work. We highly recommend these articles for more insights.

4.3.1 Architectural Components Involved

GPUs contain multiple `Memory Partition Units` (MPU) to handle upcoming data access requests. These MPUs favor coalesced memory accesses to hide memory latency for parallel threads to improve efficiency. Furthermore, within this component atomic operations are handled. Specifically, the MPU has three main parts (see Figure 4.2(a)): Frame buffer, L2 cache, and Raster Operation Unit (ROP). Here, ROP resolves atomics.

Whenever a thread encounters an atomic instruction, it sends an *atomic command* to the MPU. The command contains the target operation (add, sub, or exchange) and a payload value. This command is stored in a *command buffer* until the targeted shared data is fetched. Once fetched, the command buffer forwards the data and the atomic command to the raster operation unit (ROP) for execution (see Figure 4.2(b)).



(a) MPU components **(b)** Atomics in ROP

Figure 4.2: Components involved in global memory atomics

ROP Resolving Atomics

The forwarded atomic command is stored in an *atomic command buffer* - a FIFO queue to ensure serialized atomics. Using this queue, the ROP updates the shared result atomically. Afterward, as shown in Figure 4.2 (b), the atomic operator within the atomic command is sent to the *command queue* and its corresponding payload to the *payload buffer*. Finally, the atomic command buffer fetches the shared data using a *read command generator* module. This data is sent to the ALU to execute the atomic operation. Finally depending on the type of atomics, the result is either returned to the target thread (in case of increments, decrements, or addition commands) or simply stored in the global memory (min, max, or exchange commands).

4.3.2 Profiling Atomic Operations

Next, we study the negative impact of atomics on group-by aggregations, determining an upper bound or the worst case. This shall indicate the general potential we can expect when mitigating the synchronization overhead.

Upper Bound of Atomics Throughput

Normally, increasing the concurrency in a GPU improves the throughput. In contrast, increasing concurrency with atomics creates a backlog of threads waiting to access a memory location, adversely affecting throughput. Naturally, the severity of this backlog increases with increasing concurrency. Specifically, the severity is high when only one shared memory target is accessed, such as when the input contains a single group or `reduction operation`. The throughput of such an execution represents a worst case allowing us to measure the maximum negative impact of atomics on a GPU's throughput. Here we run a reduce operation with increasing concurrent threads. In the case of atomics, we observe a major bottleneck due to which throughput declines for high numbers of concurrent threads.

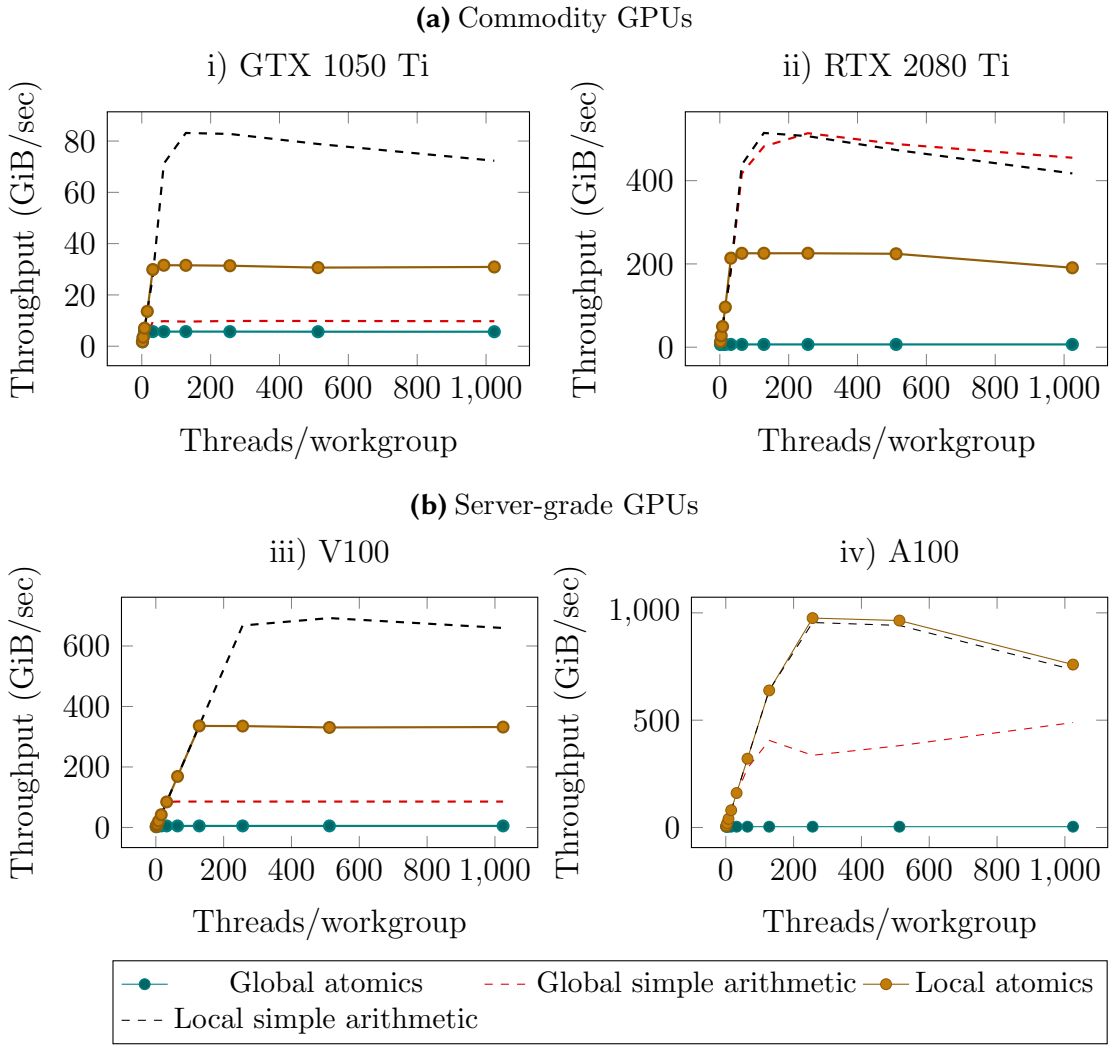


Figure 4.3: Throughput for naive atomics and arithmetics

Simple Arithmetic Operation as Optimal Throughput

To quantify the impact of atomic execution, we also execute a naive arithmetic operation on the same thread which incurs no synchronization. Since atomics has the same execution property as an arithmetic operation, this is a good way of quantifying

the impact of atomics. We consider both the global and local memory of the GPU for our experiment. The resultant throughput ranges across different GPU devices are plotted in Figure 4.3, using 2^{27} integers values as input. The results suggest three insights:

1) Comparing Figure 4.3 (i-iv), the throughput of the memory atomics of *local* in newer generations has significantly improved (instead of being 60% slower on GTX 1050 Ti, local atomics are only half as slow as local arithmetic on RTX 2080 Ti). A similar trend can also be seen in server-grade devices, with A100 having better local memory atomics. Overall, we see an increasing throughput with atomics over each newer generation.

2) The throughput difference for arithmetics and atomics is large with local atomics having a penalty of 2.0x to 2.6x on commodity GPUs and global atomics with up to 1.75x on GTX 1050 Ti and up to 77x on RTX 2080 Ti compared to their simple arithmetic counterparts. In the case of server-grade devices, V100 has a performance difference of 3x whereas, A100 has improved local memory atomics, even nearly the same as arithmetics. Hence, we need to mitigate this atomics penalty to unleash the full parallel power of present-day GPUs.

3) When using atomics, the best performance is reached with a small number of concurrent threads. In the case of commodity GPUs (GTX 1050Ti and RTX 2080Ti), we see the atomics throughput flat-line after the thread count reaches 16. With V100 and A100, the maximum atomic performance is reached at 128 and 256 thread counts respectively. Therefore, increasing the thread count after this critical threshold may reduce performance. This is the expected undesired behavior further indicating that one cannot exploit the massively parallel power GPUs offer.

These results, may at first sight suggest using local atomics. However, it is faster only with a limited thread count. Additionally, it relies on an extra synchronization step to get the final result.

4.4 Atomics within Sort-Based Aggregation

Based on our study of the architectural components, we identify that multiple components are involved with resolving atomics and these incur considerable overhead. The straightforward solution for reducing this overhead is to minimize the number of atomics issued. However as atomics are the key for aggregation, we cannot simply reduce them without developing a workaround for computing aggregates. In this section, we explain the ways to resolve atomics as well as ensure the correctness of aggregates. To this end, we first present the naive atomic aggregation and, afterward, introduce optimizations that we apply, which aim at reducing the amount of issued atomic operations in the subsequent sections.

4.4.1 Sort-Based Aggregation on a GPU: A Primer

A traditional (sequential) sort-based aggregation sorts the grouping attribute to identify the groups inside. This mechanism has two phases: The first phase sorts the input into clusters according to the group keys, which form a sequence of groups.

The second pass sequentially aggregates the groups present in the sorted input. To parallelize this processing for GPUs, additional phases are needed, as explained in the example of a COUNT aggregation below.

The sort-based aggregation on GPUs has three phases [87]: map, prefix-sum, and aggregate (excluding sort). First, the *map* phase compares two consecutive sorted-input values and returns 0 in case they match; 1 otherwise. As shown in the example in Figure 4.4, this phase marks the group boundaries of a given sorted input (with a 1). Next, the *exclusive prefix-sum* computes the target aggregate location for each group. As these two phases are commonly used in GPU, we use standard operators for them. The final *aggregation* phase aggregates the input values according to the target positions from the prefix-sum. Here, our atomic-based aggregation computes group-by results.

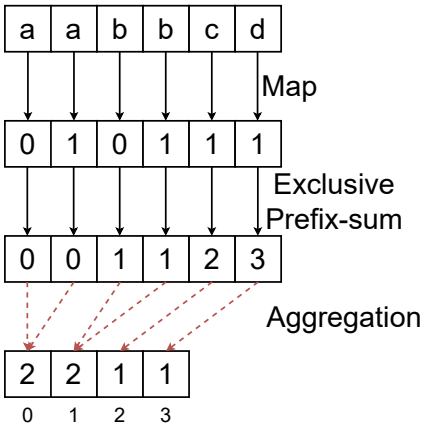


Figure 4.4: Three-phases for parallel aggregation

4.4.2 Minimizing Atomics Using Private Space

The naive sort-based aggregation issues one atomic per input value. Considering the load of executing atomics, it is reasonable to reduce the contention of threads by a more complex operator design. To this end, we exploit the fact that a sorted array has similar groups next to each other in sequence. Now, imagine the following hypothetical scenario, where we chunk the sorted data s.t. all values of a single group are assigned to a single thread. Hence, no synchronization issues can occur, removing the need for atomic operations and exploiting the full parallelism of GPUs. Of course, determining such perfect chunking creates a large overhead and leads to load imbalances. Nevertheless, as we will see, our solutions get fairly close to this ideal scenario.

The distinction of when and how to synchronize the partial result of a thread allows for proposing two algorithms: (1) using a private aggregate variable and (2) using a private aggregate array. Both versions are shown in Figure 4.5, where two threads aggregate their own chunk of three values.

The execution flow of both variants is roughly the same. In both, a thread sequentially reads its chunk of the prefix-sum and aggregates the corresponding input values within its private space until it encounters a group boundary. However, the variants differ in handling their partial aggregates and thus in the number of required atomics.

Single Private Variable Result Buffer

A thread using a private variable as a result buffer conducts an atomic operation whenever it encounters a group boundary because it only buffers the aggregate of a single group. Therefore, this variant issues as many atomics as there are groups in its input chunk. As a result, the best number of required atomics is 1, in case

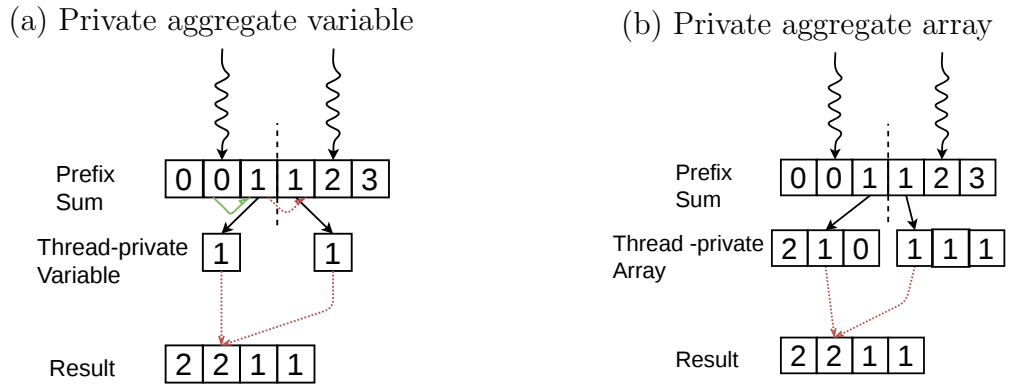


Figure 4.5: Using private address space in GPU for storing partial aggregates

there only is a single group per thread. The exact number of atomics and the time when they are issued depends on the data distribution. This is important, as this leads to the desired effect that, assuming group boundaries are evenly distributed, the number of concurrent atomics declines.

Private Array Result Buffer Variant

Instead of using a single variable as a buffer, this variant uses a private array to buffer the aggregates of all groups it processes. In the private array variant, a thread sequentially traverses its input and aggregates into the current result buffer position until a group border is found. Then, the next position is used for the next group aggregate. Since the arrays in a GPU are initialized statically, the result buffer must have the same size as the input data to cover the case that all input values belong to a distinct group. This limits the chunk size when the array is stored in local memory.

Once aggregated, the threads propagate their private result into the shared memory containing the overall result. To further mitigate the negative effects of excessive atomics usage, we use another optimization to reduce the number of required atomics per thread is exactly 2. This makes the number of required atomics independent of the data distribution depending only on the number of concurrent threads.

It works as follows: As the input data is sorted, synchronization issues may only arise for the first and the last group processed by a thread. The first group may have already begun in the prior thread's data input. The final group may continue in the next thread's data input. All other groups are only processed within the current thread. Thus, the approach pushes these results to global memory without synchronization having the *optimal* performance shown in Figure 4.3 (global arithmetic).

4.5 Experiments

In this section, we evaluate our approaches using micro benchmarks and comparison to state-of-the-art competitors. For both parts, we use the same setup: Since the GPU hardware has a direct influence on atomics, we profile our atomic-based aggregation on three GPU versions with varying degrees of usage - NVIDIA GTX 1050Ti, NVIDIA RTX 2080Ti, NVIDIA V100 and NVIDIA A100. All our experiments are executed on a Linux machine with GCC 6.5 and OpenCL 2.1. The input dataset contains 2^{27} (due

to Boost.Compute's data size limitation) randomly generated integers representing our group-by keys. While for the microbenchmark and the first comparison, data is presorted (i.e., sorting time is disregarded), the unordered data is used for fairness for the final competitor comparison. Each measurement is repeated 100 times and we present the average throughput for all variants. For brevity, we present results for count aggregation, but the result also holds for different aggregate functions and data sizes.

4.5.1 Micro Benchmark

The parameters affecting performance are (1) thread size per work group and (2) chunk size of input data per thread. To this end, we conduct experiments to examine their influence and find an optimal configuration used in the remainder.

Examining Optimal Thread Size for Naive Atomics

In this experiment, we identify the optimal thread size per workgroup for naive atomics serving as the baseline. Notably, the implementation of the naive atomics variant on global memory is straightforward (i.e., the aggregation step in Figure 4.4 uses an atomic operation on the global memory). However, the atomic variant on local memory needs an additional merging step. This step is to merge the partial aggregates inside the workgroup's local memory into the final result in the global memory. In this naive local variant, we perform merging similar to the approach used for our private array variant, where only the first and last positions are merged atomically. The throughput ranges for this experiment across GPU devices are depicted in Figure 4.6.

Our results are uniform across the devices and record best throughput ranges with large groups in input when spawned maximum number of threads. Such a behavior is because multiple threads efficiently hide memory latency. Furthermore, a higher number of groups (i.e., a larger spread of target locations in memory) creates less concurrency on atomic writes. The results also clearly show an improvement from using local memory as a cache for partial aggregates. Still, the penalties of an extra merging step are significant and thereby reduce the overall throughput. As an overall result, the best thread sizes are 256 for GTX 1050 Ti & V100 and 1024 for RTX 2080 Ti & A100, which we then use to compare naive atomics with our approaches and the competitors.

Best Thread and Chunk Size for Atomic Variants

In addition to the thread sizes, our variants using a private array/variable (either in local or global memory) are also influenced by the number of input values per thread (chunk size). Hence, we average the variants' throughput over all tested number of groups and plot the results in Figure 4.7[15]. Taking into account the influence of the size of the chunks on performance, we observe that the smaller to medium-sized chunks ($2^2 - 2^7$) are beneficial compared to the larger ones. In the latter case, the memory controller becomes the bottleneck, due to too many requests from threads that fetch input data from global memory and the execution of atomic operations.

[15]Note that not all combinations of chunks and threads are possible as they cross the physical limit of local memory that can be allocated.

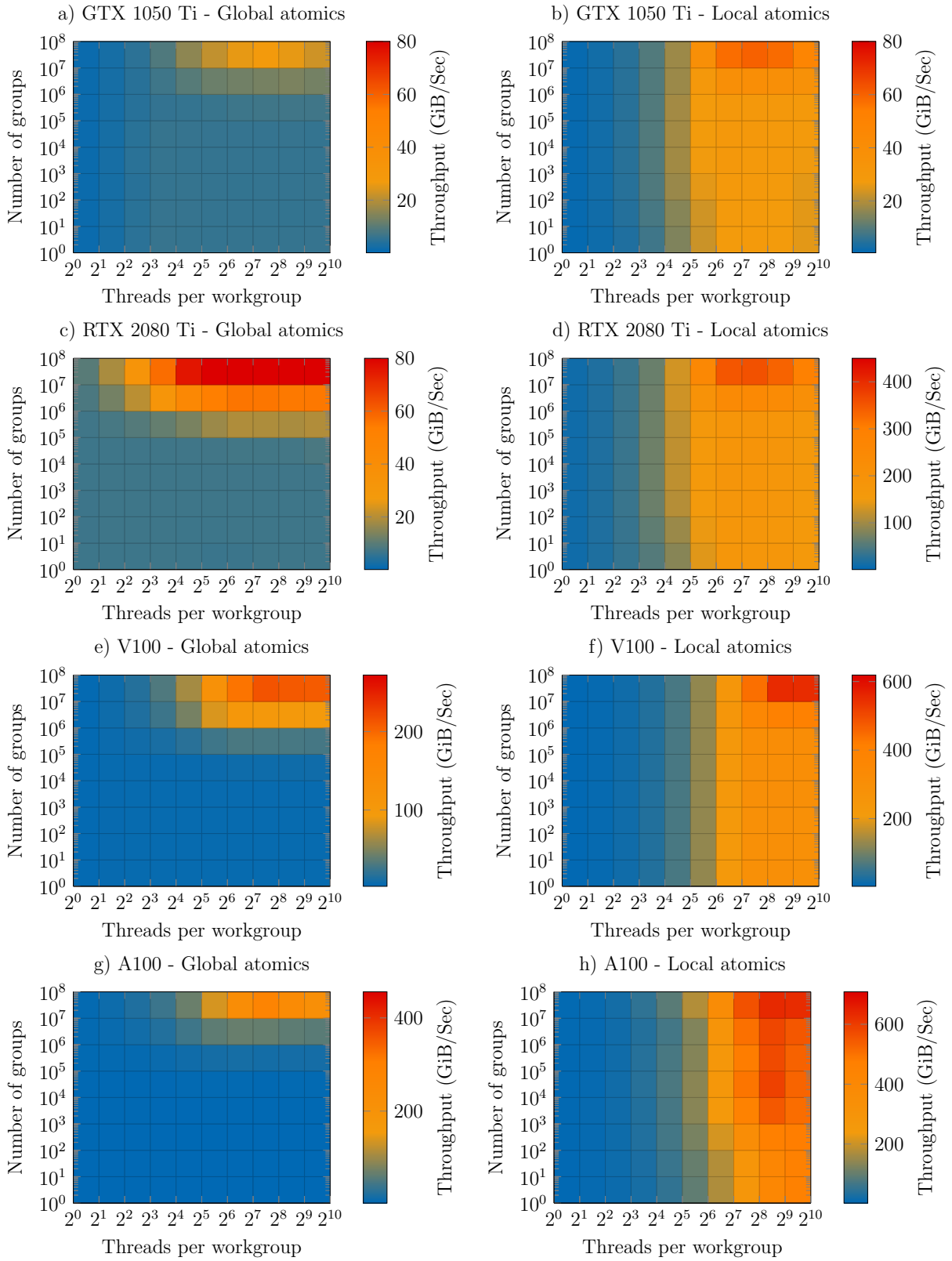
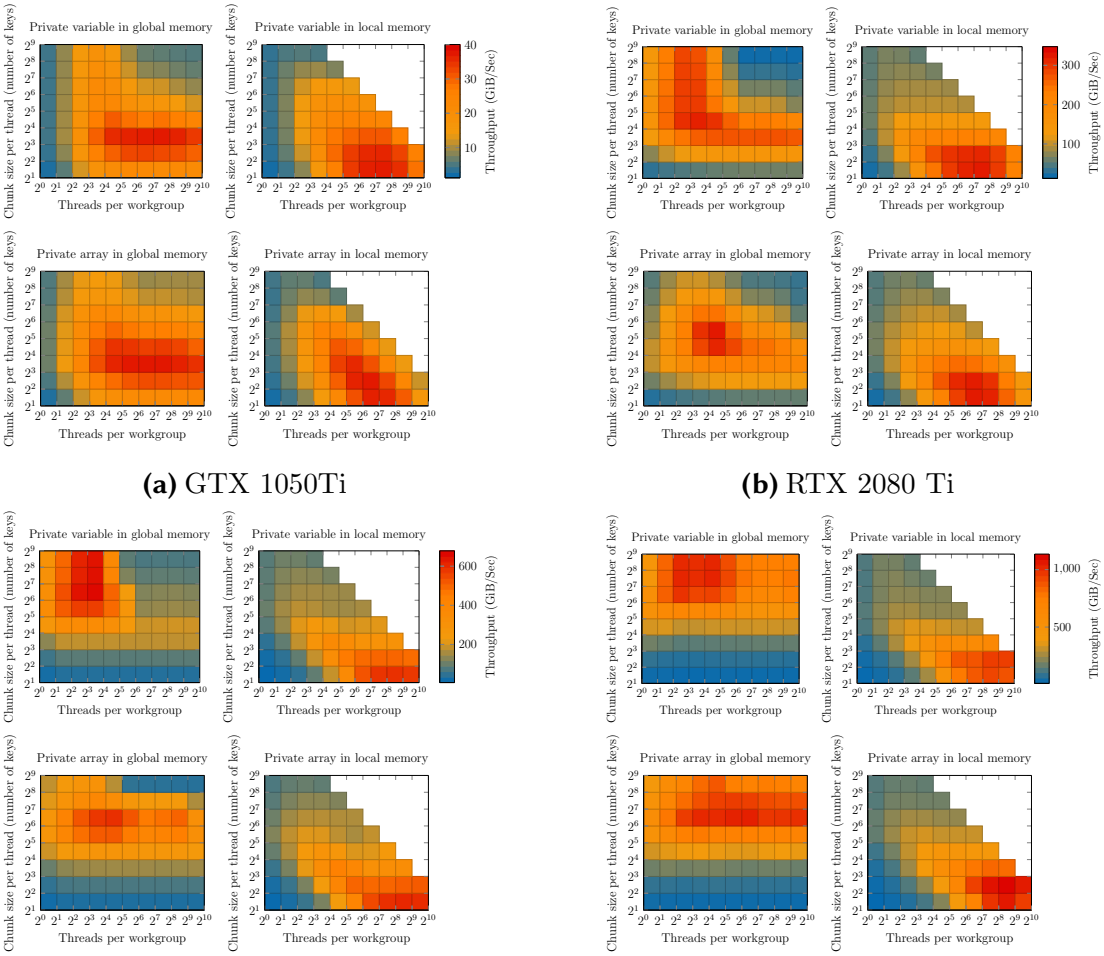


Figure 4.6: Throughput profile varying with changes to group and thread size across different NVIDIA GPU generations

Since the MPU incurs coalesced accesses, fetching bigger chunks of data for multiple threads requires multiple cycles, which degrades performance. In contrast, the local memory variants prefer very small chunk sizes ($2^1 - 2^3$), whereas global memory benefits from slightly larger ones ($2^2 - 2^7$). Unlike the naive atomics where larger numbers are beneficial, chunking improves the performance of even smaller thread sizes. Interestingly, there is only a small difference between using a private variable and a private array to store intermediate results. On the contrary, the throughput behavior changes w.r.t. the devices, since there is a wide spectrum of well-performing variants on GTX 1050 Ti, which shrinks for the RTX 2080 Ti. This indicates that variants are sensitive to the underlying hardware and need a smart variant tuning procedure [155].



(a) GTX 1050Ti

(b) RTX 2080 Ti

(c) V100

(d) A100

Figure 4.7: Impact of varying chunk and threads sizes over throughput

4.5.2 Comparative Experiments

Based on the inferences above, we can now tune the parameters of our variants for their optimal value. Now using these parameters, we can compare our variants to study their impact with varying group sizes. Such a comparison is studied in the next section. Later, we use these optimal executable variants for our subsequent experiments against the state-of-the-art group-by-aggregation mechanisms and other workload

alternatives. Specifically, we compare our variants against 1) hash-based aggregation for GPUs, 2) using alternate device - CPU, and 3) various data distributions, which are detailed later in this section.

Comparison of Atomic Variants

First, we identify the best variant of our approaches per device used for the final experiment. To this end, we compare the performance of the best-performing chunk- and thread-size combination of the two private aggregate variants with the naive atomic variants with an optimal thread size. The results are shown in Figure 4.8.

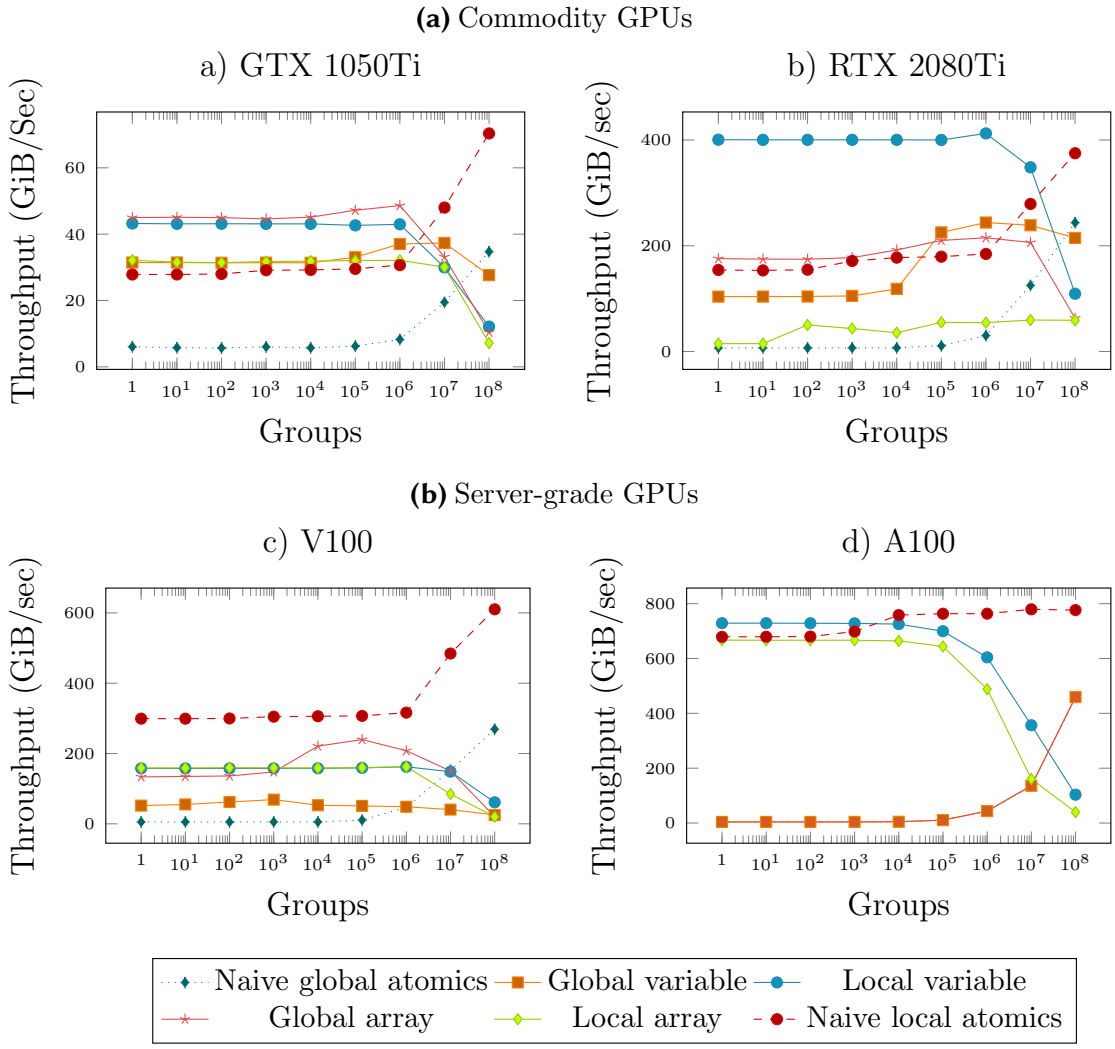


Figure 4.8: Performance comparison of atomic variants

The variants in Figure 4.8 are named as Naive representing the simple atomic aggregation without the variants, Global/local (target memory) Variable/array (variant from section 4.4.2). Our results show that the global array and local variable have higher throughput than the naive atomic variants for almost all numbers of groups (i.e., except a larger number of groups). This limitation of our variants is expected as a larger number of groups leads to multiple groups within a chunk. In this

case, a thread has to repeatedly insert the final result into global memory degrading its performance. We also see only a small improvement in using local memory for our variants on the GTX 1050 Ti, which in contrast is a huge improvement on the RTX 2080 Ti. This is consistent with Section 4.3.2. Finally, for very high amounts of groups, the overhead of internal synchronization for the private aggregate variants does not pay off. Hence, naive local atomics performs best in this case.

Summary: Our variants reach a speed up of 6x-12x to the naive atomics and 1.5 - 2.6x to the naive local memory atomics. For GTX 1050 Ti, the variant using a private array in global memory is optimal with a speed-up of 6x the naive atomics and 1.6x the naive local memory atomics. For RTX 2080 Ti, the variant using a local variable is clearly superior with a speed-up of about 12x the naive atomics and up to 2x the local memory atomics.

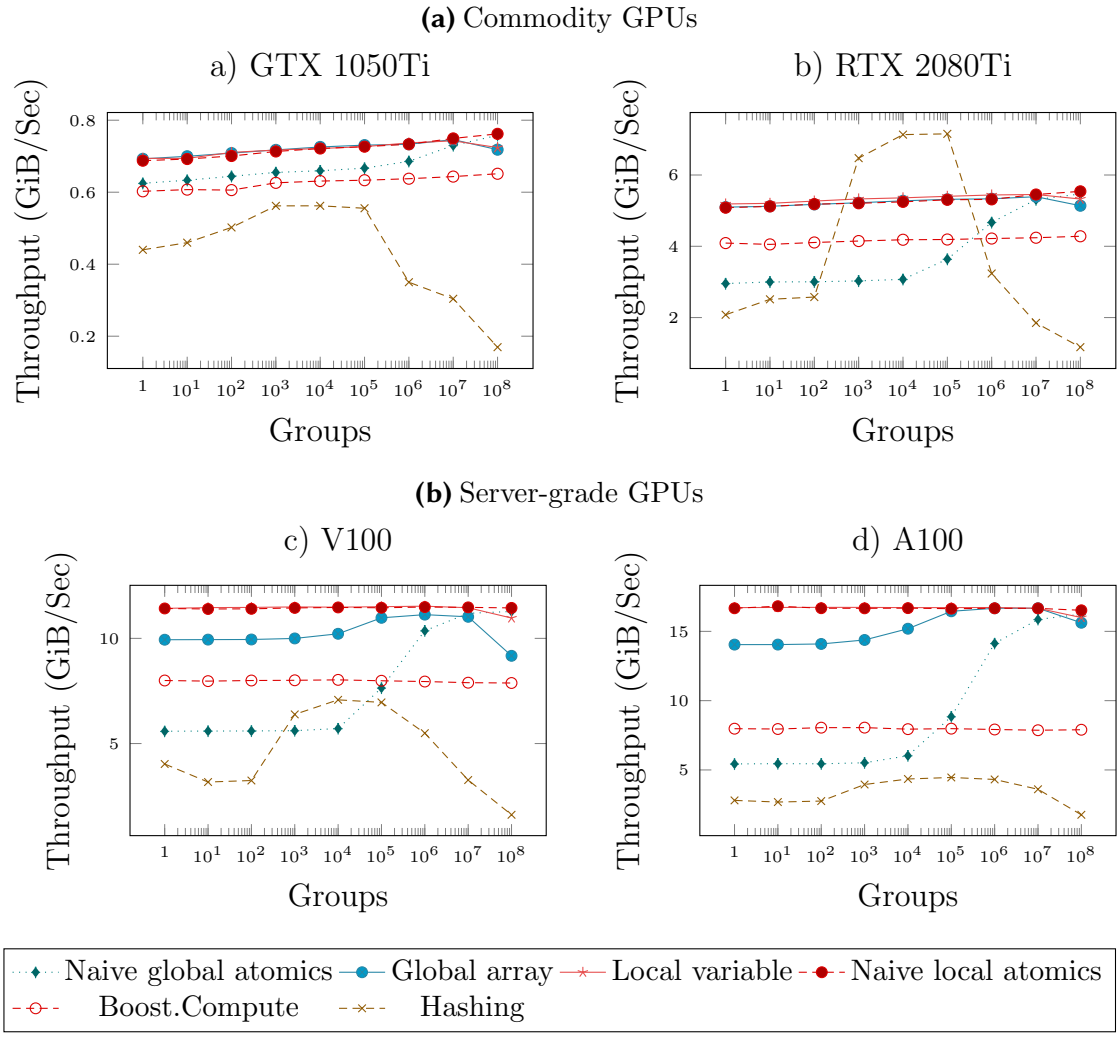


Figure 4.9: Overall comparison against state-of-the-art competitors. The performance of atomic variants now includes sorting.

Comparison With Hashing

As a final evaluation, we compare the performance of our atomics-enabled sort-based aggregation with another state-of-the-art mechanism i.e. hash-based aggregation. Since, hash-based aggregation processes unsorted input, we also include the sorting

step along with the best-performing atomic variants. The hashing technique is realized based on the mechanism proposed by Karnagel et al. [104]. Additionally, we also compare against a baseline sort-based aggregation: Boost.Compute library to visualize the improvement from using atomics. Our results in Figure 4.9 show that our variants have performance benefits across different devices and group cardinalities. In a gist, our local atomics variant has the best performance overall. On the GTX 1050 Ti, we reach on average 20 % speed-up over naive global atomics and Boost.Compute, while it reaches nearly 2x the speed of hash-based aggregation. We see a similar speed-up on the RTX 2080 Ti except for the local variable variant that reaches up to 1.25x the performance of Boost.Compute. Interestingly, hash-based aggregation is the best for groups between 1,000 and 100,000. Here, smaller group sizes lead to a synchronization when accessing the shared global hash table, a larger group sizes need a hash table beyond a manageable memory size.

Comparison With CPU

To compare against the CPU, we run the same atomic-based aggregation and hash-based techniques in the Intel Xeon CPU and compare its throughput against the A100 GPU. Due to a high degree of parallelism, GPU in general has higher throughput than CPU. As we see in Figure 4.10, the throughput ranges of GPU is in order 10x faster than that of a CPU. Additionally, the throughput of aggregation in GPU has a clear difference in throughput across atomics and other techniques, whereas the hashing and boost. compute-based aggregation in CPU are competitive with each other. In general, aggregation runs orders of magnitude faster, even with atomics, due to the efficient serialization of aggregation from parallel threads.

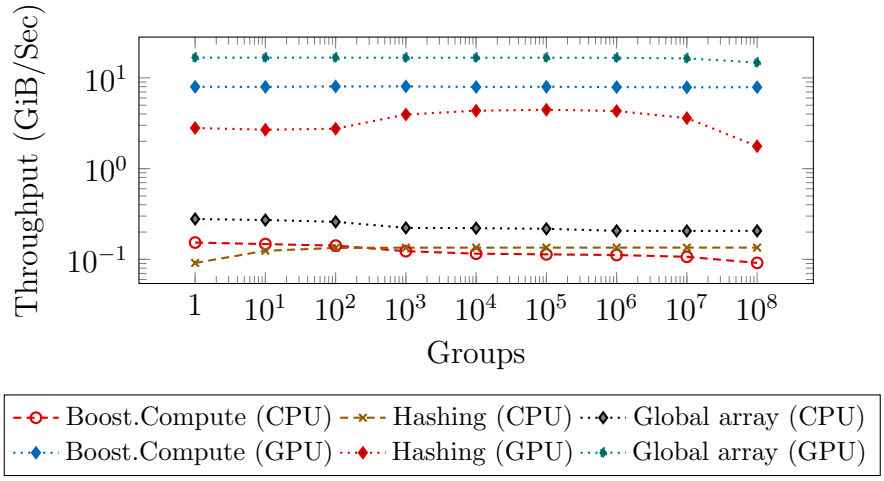


Figure 4.10: Throughput comparison of grouped aggregation in CPU (Intel Xeon) and GPU (A100)

Comparison Across Data Distributions

As our final evaluation, we experiment with various data distributions. Once again, we consider input to be 2^{27} integer values. The distributions considered are heavy

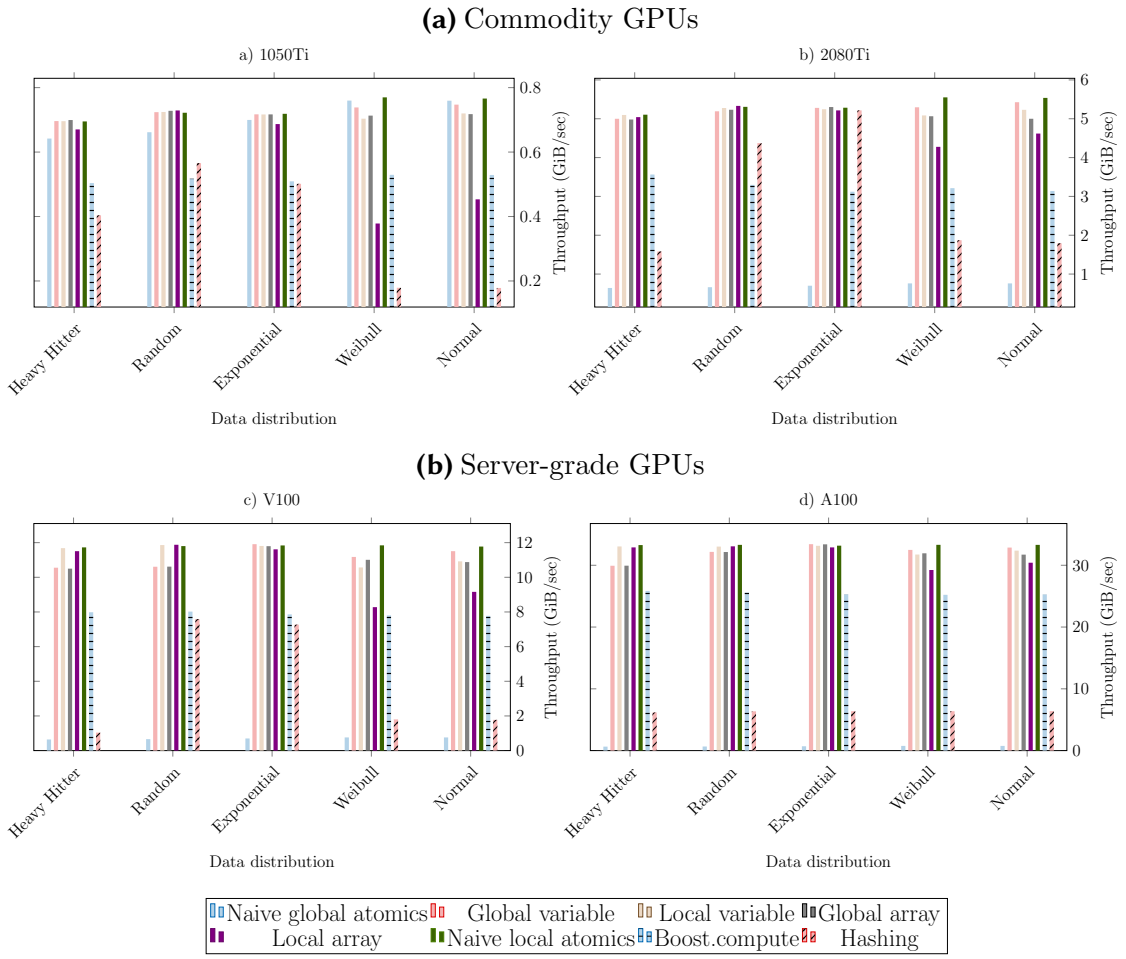


Figure 4.11: Performance of aggregation techniques across various data distributions

hitter (90% of input is a single group), random, exponential(lambda = 0.05), Weibull (a=2.0 & b=4.0), and normal. The corresponding throughput ranges across the different GPU devices are given in Figure 4.11.

From the results, we see that naive local atomics is consistently performing better than other techniques for all distributions as well as across all devices. Similarly, naive global memory atomics is consistently the poor-performing variant of all atomic-based aggregation techniques. remaining techniques vary in performance depending on the distribution and device.

In addition, results for server-grade GPUs are consistent with each other in terms of their relative performance. The throughput ranges are higher for A100 compared to V100 due to its higher CUDA core count. However, in the case of commodity GPUs, the hashing is comparatively faster for RTX 2080Ti than for GTX 1050Ti when subjected to random distribution.

Discussion

In summary, we can see that for the common use case of up to some hundred groups[16], a sort-based aggregation using atomics is the superior variant to be used. This is

[16]For instance in the TPC-H, 11 out of 16 queries do a group-by on less than 500 groups. Seven of them operate on less than 10 groups.

remarkable, as usually hashing is the best variant [104, 22]. We argue for a change of this general assumption for the following three reasons:

- There are a lot of circumstances where presorted data is grouped (due to sort-merge join or a clustered index) or data has to be sorted after executing the grouping (due to an order-by statement). In these cases, it would be the natural option to also employ a sort-based grouping.
- Although the sorting time dominates the throughput of our variant in Figure 4.9 (making up 80% of the execution time), it is still the most stable strategy on the GPU across the group sizes. The reason is a more cache-friendly access pattern and a better fit for the SIMT processing model of the GPU [108].
- Due to the increased local memory performance of modern GPUs, the overhead of atomic operations can be effectively mitigated.

4.6 Summary

GPUs with their massively parallel processing have been used for more than a decade now to accelerate compute-intensive database operators. One such compute-intensive database operator is a grouped aggregation. Although, up to now, hashing is the predominant technique for grouped aggregations even on the GPU, a sort-based grouped aggregation is an important alternative to be considered – especially with an improved performance of atomics.

In this chapter, we investigate how far we can tune a sort-based grouped aggregation using atomics in the aggregation step. To this end, we design two alternative variants using a private variable or array and investigate their performance improvement when using local or global memory followed by an atomic-based propagation of private aggregates.

Our results show that our variants speed up grouped aggregation compared to a naive usage of atomics by a factor of 1.5 to 2, when well configured. Furthermore, a sort-based grouped aggregation using atomics can outperform a hash-based aggregation by 1.2x to 2x for most used group sizes.

Overall, we see in this chapter the importance of using a hardware-sensitive operator. Though the benefits are lucrative, it also takes a considerable implementation effort. Extending such with multiple co-processors present in the market, many implementations of a single operator will be available with considerable human hours spent perfecting them. Handling such alternative implementations across different hardware further increases the complexity of development. To reduce this effort, we can enforce a standard signature for these implementations so that we can encapsulate them into a single functional entity. To this end, we survey the existing database operations to come up with reusable functions that can be combined to run a complete query. Our survey covers the standard data processing operators, that are examined to develop granular functions. The next chapter covers the survey with the various identified functions. We also explain in the Chapter the combinations of these functions to run a different database operator and by extension a complete query.

5. Tier 1: Primitive Definitions for Interfacing Operators

Executing queries over a heterogeneous processor needs re-implementation of database operators for the processor. Such re-implementations end up with multiple `variants` of a single operator available across each of the processing architectures. Usually, these implementations are ported to a new co-processor using a hardware-oblivious implementation translating the code according to the underlying device. However, we already explained that direct translations might not be necessarily performance-efficient (refer to Section 2.2.2). Hence the alternative is to write optimal implementations using device-specified language constructs for a given co-processor. This approach is time-consuming and requires complex testing. To overcome these disadvantages, in this chapter, we propose the use of granular building blocks for database operators called *primitives*. The name *primitive* is quite influential and, hence, a primitive in literature can range from describing a database operator to the finest granular function (e.g., a parallel loop with a single instruction).

In this chapter, we aim to shed some light on the current state of the art in primitive for database operations. We explore the available primitives with a comprehensive survey. Using the survey, we define a minimal set of primitives necessary for implementing a given database operator and present different tuning opportunities for these primitives to the underlying hardware. Finally, our proposed classification of primitives spans multiple levels of granularity, which requires changes in the query execution engine. We discuss different implementation strategies that can be used for primitive-based evaluation by the query engine and its components. We also provide an overview of the other factors impacting the efficiency of the primitives.

Parts of this chapter has been based on the following publication:

> B. Gurumurthy, D. Broneske, T. Drewes, T. Pionteck, G. Saake, "Cooking DBMS Operations using Granular Primitives". Datenbank-Spektrum: Vol. 18, No. 3. pp. 183-193(2018).

We split this chapter into four main parts discussing the primitives. We start with defining a hierarchical model for the primitives based on their level of granularity in Section 5.1. Next, we list the different primitives from our survey and place them in the given hierarchy. Each level of the hierarchy is detailed in its section: Section 5.2, Section 5.3. These sections elaborate on the individual primitives and possible implementation alternatives to realize these primitives. Further in Section 5.4, we briefly discuss the different factors affecting primitive performance. We discuss integrating these primitive definitions into a query engine in Section 5.5. Finally, we summarize the chapter in Section 5.6

5.1 Defining Primitives

Modern CPU architectures provide multiple code optimization opportunities to fine-tune a given database operation. Based on the efficiency criteria (e.g., cache utilization, response time), we have different variants for a single operation. In addition to these criteria, the code optimization strategies also vary according to the used processor. Thus, we have a plethora of different variants available for evaluating a single operation. Implementation of all these variants is time-consuming and not manageable. As an alternative, these operations are evaluated by coupling database primitives.

A database primitive is a basic building block that can be re-arranged to execute a part of a given DBMS operation. Primitives aim to minimize the implementation time and also improve the re-usability of code. Furthermore, they usually are easily parallelizable. Exploiting this level of parallelism improves the throughput of overall computation. Based on the level of granularity multiple primitives have been proposed for DBMS operations [87, 140, 29]. On a conceptual level, the DBMS operations itself serve as primitives that are combined to form the user-defined query. We detail this hierarchy in Figure 5.1.

As shown in Figure 5.1, primitives are present across various hierarchical levels. At the lowest granular level, the primitives are just stand-alone functions with multiple variants for efficient execution. These primitives are combined with operation-specific components forming coarse-granular or *composed primitives*. These provide additional functionality by adding a tailor-made component with the atomic primitives. Finally, the desired algorithm for executing an operation is built using these primitives. In the next sections, we provide an overview of the different primitives available and discuss how they are used to form a DBMS operator.

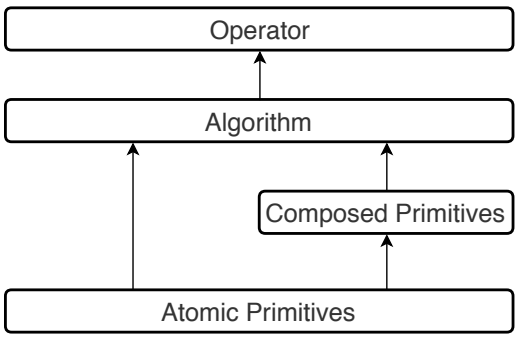


Figure 5.1: Hierarchy in realizing primitives

5.2 Atomic Primitives

Atomic primitives are fine-granular functions that can be combined to form a larger operation. These primitives are data parallel in nature and can be tuned for efficiency based on the underlying processor. Furthermore, an atomic primitive consists of a single loop with a body that cannot be split further (i.e., it contains a single operation executed on a vector of data). In this section, we present the atomic primitives *map, scan, reduce* and *scatter & gather*, as well as code optimization strategies available for each of them. An overview of their properties can be found in Table 1.

Table 5.1: Properties of atomic primitives

Primitive	Input Size	Working Space	Output Size	Access pattern	Multi-step
MAP	N	-	N	Sequential	N
REDUCE	N	$\leq N$	1	Random	Y
SCAN	N	$W + N$	N	Random	Y
SHUFFLE	N	-	N	Random	N

N - Vector Size; W - Work group size;

5.2.1 Map

The map primitive applies a function f (e.g., a *user-defined function (UDF)*) to all values in the given input vector. This is a non-blocking primitive that applies f to each input independent of the other.

Naive Implementation

In a basic implementation, a map is simply a loop applying f to all input values. Depending on the definition of the function f, a map either takes two columns, or one column plus an additional constant as input. Frequently used functions include:

- **Arithmetic operations:** Arithmetic operations perform an arithmetic function (e.g., addition, multiplication) on either vector-vector or vector-scalar arithmetic operation.
- **Logical operation:** Logical operations combine vectors on a logical/bitwise operation on the vector values.
- **Comparison:** A comparison tests whether an input value in the vector passes a given condition. It returns boolean vectors of `true` or `false` as a result, which can also be bit-packed.
- **Bit-shift:** A bit-shift takes two inputs – value and shift count – and shifts bits inside the given value by the shift count value.
- **Zip:** A zip pairs values from two vectors into a single vector.
- **Project:** A project forwards the required set of values from the given vector to the next function in the process.

Possible Code Optimizations

Since a map is implemented in a loop, loop-based code optimization can be applied. Loop unrolling is used to reduce pipeline stalls. Stalls are reduced by unrolling a tight loop to the desired depth. This depth is decided based on the underlying hardware used [92]. Another optimization commonly used is loop fission or fusion. Depending on the execution environment, a single map function can be divided into multiple smaller loops or multiple maps can be combined into a single loop [136]. Other than these strategies, various other factors also influence the efficiency of a map such as access patterns, data structures and so on which we discuss later in this Chapter.

5.2.2 Scan

A scan is proposed as a fundamental primitive for vector processing by Guy Blelloch [27]. This primitive is mainly used for determining the index for storing values in a parallel processing environment. Additionally, this primitive is further extended into a segmented scan. A segmented scan performs the scan operation on arbitrary segments and can be used to sort a vector.

Naive Implementation

The scan takes a single vector as input and performs a given binary operation $\oplus$ over all the values between the initial and current index (c.f. Equation 5.1). Some of the common binary functions applied with scan are sum, min, max, and multiply.

$$[v_0, v_1, ..., v_n] = [v_0, v_0 \oplus v_1, v_0 \oplus v_1 \oplus v_2, ...] \tag{5.1}$$

The parallel implementation presented by Blelloch has the input and output data found in the leaf nodes of a balanced binary tree and computes results in two phases. The phases are up-sweep and down-sweep.

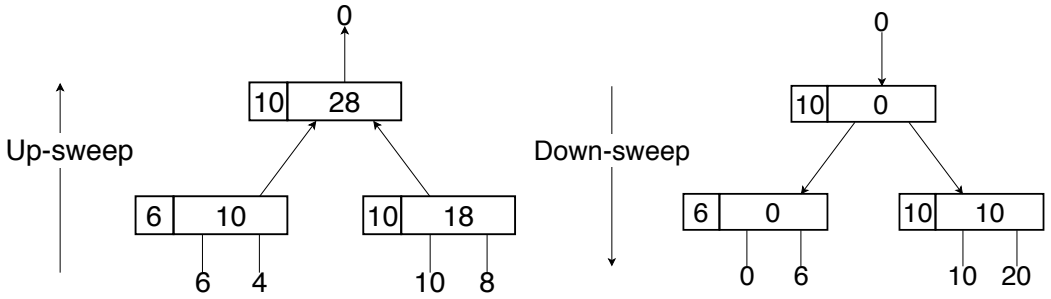


Figure 5.2: Example of prefix-sum phases

Up-sweep: During up-sweep, the binary operation is carried out on the child node, and the resultant values are stored in their corresponding parent nodes. Along with the result, the value of the left node is also stored in the parent.

Down-sweep: In the down-sweep phase, the binary operation is applied to the left node stored in memory and the corresponding parent node value and the result is stored in the right child. The value of the parent node is stored in the left child of the current node. Note, that the initial value for the root node is given as zero for further propagation.

In Figure 5.2, we show an example of the prefix-sum calculation using the two phases. The result from the root to the leaf is carried down and the binary operation is performed again. The result is stored in the right child thereby computing the result or output vector. Overall, this algorithm has the complexity of $O(n \log(n))$ with the final results present in the leaf nodes.

Segmented Scan

As mentioned previously, the simple scan operation is extended to perform a segmented scan. These segments are marked using flags. The common functions used with segmented scans are arithmetic and comparison. These two variants of scan are used as the primitive function for many composed operations such as enumerate, distribute, etc. Their implementations are detailed by Blelloch et al. [27]. The prefix-scan composed database operations are discussed in Section 5.3.

Possible Code Optimizations

Since prefix sum is a multi-fold operation (where the result is calculated in multiple stages), data parallelism and task parallelism provide additional benefits by reducing latency. In CPU implementations, this is realized using threading and pipelining of instructions. Whereas, a GPU with its massively parallel processing capabilities is used as an ideal candidate for computing prefix-scan results. One of the common implementations of prefix scan in GPU is given by Horn et al. [94]. Whereas, Senguptha et al., provide a GPU implementation of segmented scan [165].

5.2.3 Reduce / Aggregate

Reduce computes the aggregate of values in a given vector. A reduce evaluates multiple input values and returns a single output. Hence, the result is not forwarded until all the corresponding input values are evaluated. This operation is a pipeline-breaker as the result is not forwarded until all the corresponding values are processed [131].

Naive Implementation

The implementation of a reduce is similar to that of a map. However, unlike the map, the intermediate result from a single value is reused by the given function f for computing the next result.

Apart from using a UDF, all aggregate operations carried out by SQL are implemented using reduce (e.g., min, max, sum).

Possible Code Optimizations

Similar to other primitives, parallel execution of a reduce improves its throughput. Horn et al., provide a two-phase algorithm for computing reduce [94]. Since the final result depends on the intermediate results of applying the function f, a parallel reduce needs at least two passes to compute the final result. In the first pass, a local aggregate is calculated within each of the thread groups with an intermediate output size equal to the available work-group size. In the second pass, the intermediate results are aggregated to compute the final result. Since reduce is similar to map, all the optimization criteria of the map can be applied to reduce such as SIMD acceleration (data parallelism) and instruction pipelining for faster processing. Rosenfeld et al. have given an extensive analysis report on the impact of these optimizations for reduce computation [155].

5.2.4 Scatter & Gather

Scatter and gather are data movement primitives that perform an indexed read/write on given data. These primitives are used to rearrange the given dataset. These two different primitives are discussed below.

- **Scatter:** Writes data into a vector from the input on the index given.
- **Gather:** Read data from the input vector at the given index.

These are the atomic primitives required to compose the coarse granular primitives or a complete DBMS operation. In the next section, we detail the composed primitives developed by combining atomic primitives.

5.3 Composed Primitives

The next layer of DBMS primitives from Figure 5.1 are composed primitives. Although composed primitives can be implemented by pipelining atomic primitives, they can also be implemented from scratch by fusing the code of atomic primitives.

5.3.1 Filter

A filter is used for selecting values from a given input vector that satisfies a certain condition. This primitive is used for selection and returns a subset of the given input vector.

Composing Filter

A primitive-based filter implementation is given by He et al. [88]. The pipeline is depicted in Figure 5.3.

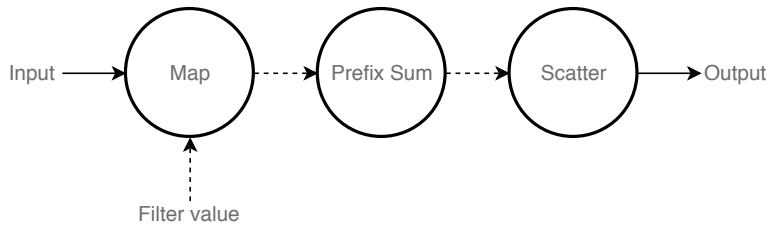


Figure 5.3: Composing filter

Since the result size is not known in advance, memory space may not be allocated for it or else have multiple empty spaces unused. Hence, the filter pipeline requires a map to determine the results followed by a prefix scan to allocate the required space and finally scatter to store the results.

Possible Code Optimizations

A filter can also be implemented from scratch using code optimization strategies for efficient execution. There are three major improvements available for a selection [43]:

- **Branched logical:** The execution breaks as soon as a condition fails. It has the disadvantage of costly multiple branches (if the data distribution is highly selectivity).
- **Branched bitwise:** It evaluates all the conditions in a bit-wise manner and a final branching statement will check the result.
- **Predicated:** The predicated version converts control dependencies into data dependencies thereby omitting branches.

Along with different implementations, the conjunction order of selections also provides additional throughput. Ross et al. provide a comprehensive analysis of filtering using conjunctive selections [157], while Broneske et al. [41] and Zeuch et al. [190] show the impact of hardware sensitivity on selections. A GPU implementation of select conditions is given by Sitaridi et al. [169]. Finally, selection over compressed vectors provides additional advantages and many works also investigate the impact of the compression in filtering [187, 188, 144]. Selections over compressed columns require an additional step of generating the column values from the intermediate result. This additional step is called materialization, which we detail in the next section.

5.3.2 Materialize

Materialize is mainly used in synthesizing column values from encoded results. Abadi et al. have done an extensive analysis in materialization [2]. Materialization is commonly used in heterogeneous computing environments since data transfer is the bottleneck. The selected results are encoded into bitmap or position list values and transferred, thereby reducing the transfer bottleneck. A materialize is then used to retrieve the column values from the generated intermediate results.

Composing Materialize

Materialize is executed using a map and a gather primitive based on the encoded data.

Bitmap: For filter-materialize pipelines with a bitmap, a map compares the input and generates a series of 1s or 0s as a result (either bit-packed or byte-packed). In this case, materialize uses a method similar to filtering, where a prefix scan is used to identify the location and a map to store the results (refer to Figure 5.3).

Position list based: In the case of performing materialize with a position list, a map function compares the input vectors followed by a gather primitive to collect the values. The composing primitives are constrained only on the order of execution and there can be other primitives executed between their execution. We detail the execution of filter-materialize with position list in Figure 5.4.

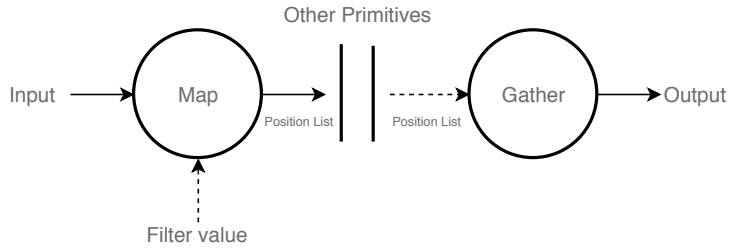


Figure 5.4: Composing materialize

Possible Code Optimizations

Since materialize is a special functionality of map and gather, their optimizations can be applied to it. Abadi et al. provide a cost model for placement of the gather primitive early or late in the pipeline [2], calling it early materialization and late materialization, respectively.

5.3.3 Hash Build

The hash build takes a value and based on the underlying hash function and technique stores the value in its respective bucket. A hash table is built using a map along with a tailor-made hash put component for the underlying hashing technique. Note: we represent these tailor-made components using double circles.

Composing Hash Build

To implement a hash build, a map is used with a hash function (e.g., multiplicative hashing [109]) to provide the index of a bucket to store the value in. Since the underlying techniques are prone to collision, the hashing technique defines a collision-resolution mechanism as hash put. The primitive composition of the hash build is shown in Figure 5.5

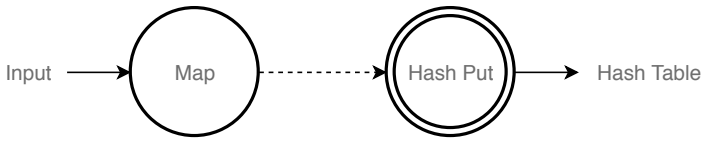


Figure 5.5: Composing hash build

There are various hashing techniques available and based on the technique, a different collision resolution mechanism is used. An extensive list of the hash functions and available hashing techniques are given by Richter et al. [150].

Possible Code Optimizations

Hashing from scratch combines hash build with a probe phase. The code optimization on the hash build is applied to the hash probe as well. Hardware sensitive optimization such as SIMD optimized cuckoo hashing is given by Ross et al. [156]. Polychroniou et al. has given an abstract overview of using SIMD on hashing techniques [142].

5.3.4 Hash Probe

Hash probing is used to retrieve values from a given hash table. This along with hash build is used to evaluate join and aggregation queries. Probing performs a search of a value from a computed index in the hash table. Since hashing has collisions, the given search value may have to be filtered from possible candidates.

Composing Hash Probe

Probing is highly dependent on the hashing technique used to insert a value into the hash table. Hence, the hash put component is modified to provide the possible indexes to probe. We name this as hash get. Values in the indexes given by hash get are fetched using a gather and finally, a map is used to compare the value at the determined indexes. The flow of the hash probe is given in Figure 5.6.

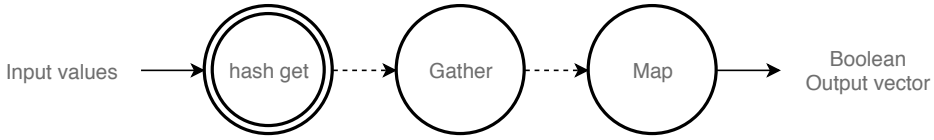


Figure 5.6: Composing hash probe

5.3.5 Split

A split or partition divides the given array of values into multiple arrays. Split gets in an input vector along with a flag vector and based on the flag, the given input is split into multiple chunks.

Composing Split

He et al. have devised a way to compile split using their data parallel primitives [88]. Based on their interpretation, the split is compiled using a map, prefix-sum, gather, and scatter primitives. Their interpretation is lock-free and uses a histogram to compute the required space. This is given in Figure 5.7.

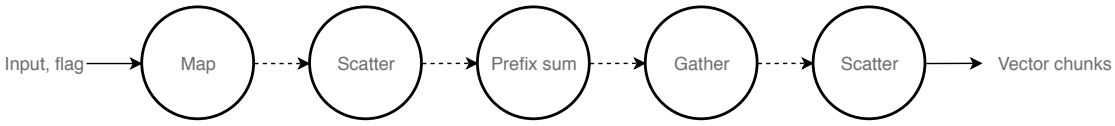


Figure 5.7: Composing split

Split is one of the complex operations and requires five steps. At first, a map is used to prepare the histogram of values per bucket in each thread. Then these histogram values are scattered into an array and a prefix sum is performed. The results from the prefix sum are scattered back to the threads marking the write locations for the threads. Finally, the values are scattered based on the indexes.

Possible Code Optimizations

The impact of code optimizations in radix partitioning is extensively researched in various works. Polychroniou et al. give a complete overview of different partitioning variants based on cache efficiency and provide details on the impact of hardware on these algorithms [143]. Schuhknecht et al. provide a guided approach for optimizing partitioning [162].

5.3.6 Sort

Finally, similar to split a sort is also implemented using the atomic primitives. Specifically, He et al, have presented a way to perform quick sort using atomic primitives [88].

Composing Sort

To perform a quick sort, He et al. divide the given input by arbitrarily chosen pivots until a partition fits local memory. Here, the split primitive is used to divide the vector into chunks. Sorting a local memory chunk is done using bitonic sort networks.

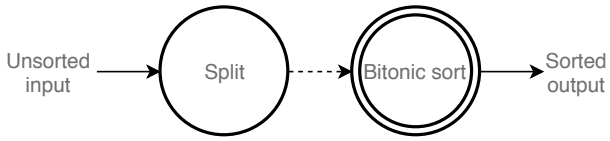


Figure 5.8: Composing sort

Possible Code Optimizations

Similar to split, the sort can also be implemented from scratch with code optimization strategies. As sorting requires a high availability of data in memory, cache efficiency improves the processing speed. To this end, Govindaraju et al. provide an implementation of a cache-efficient sorting algorithm for GPUs [79]. To further improve the availability of data, Albutiu et al. proposed a NUMA-aware algorithm for sorting [4].

A DBMS operation is executed by combining the above atomic and composed primitives. Other than code optimization strategies mentioned for each of the primitives, considering certain external factors also improves the performance of the primitives. We discuss some of these factors in the next section.

5.4 Other Impact Factors

Other than the primitive-specific code-optimization strategies, there are several factors influencing the processing efficiency of the overall query. Some of these influencing factors are the selected access pattern, parallelism, as well as data structure for underlying data. We briefly discuss these in this section.

5.4.1 Access Pattern

In a heterogeneous system, a device-specific access pattern (e.g., sequential or coalesced memory access) is used for additional efficiency. Rosenfeld et al. analyzed the impact of the access patterns on various devices and show that the right access pattern improves efficiency on the underlying hardware [155].

5.4.2 Parallelism Mode

One major advantage of splitting DBMS operations into granular primitives is to enable concurrent execution. Based on the parallelism type – data or instruction parallelism – two different functionalities are required. For instruction parallel execution, the result of concurrent primitives must be materialized before being used in the next primitive. This is processed using a custom materialize function based on the used primitives. In the data parallel approach, the given data is divided into multiple chunks and is executed in parallel. As a final step, these locally processed results are combined by performing the same operation over the final results. We call this secondary combine as *defer*. Similar to the materialize primitive, defer is also based on the underlying function. Thus, based on the parallelism method an additional hidden function is used. This also influences evaluating a complete query.

5.4.3 Data Structure

Though sequential access provides additional advantages of prefetching and data locality, different data structures are used for efficient processing of selective operations [149]. Powerful indexes such as CSB-tree [148], FAST [107], k-ary search trees [191], or Elf [42] can be used for efficient selection.

5.5 Primitive-Based Execution in a Query Engine

Since the DBMS operations are divided into primitives, the query engine is also modified to execute them efficiently. The first step is to split the given relational operation into granular primitives. Since multiple levels of primitives are available, a DBMS operation is either a set of atomic primitives or a combination of the predefined composed primitives and the atomic primitives. We depict the necessary primitives for composing a complete operation in Figure 5.9.

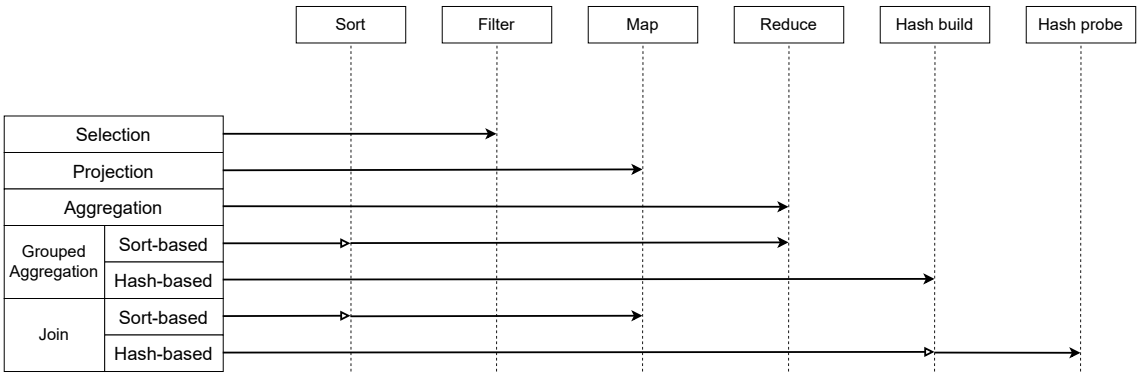


Figure 5.9: Composing DBMS operators from primitives

For example, as we know now, a grouped-aggregation operation can be executed using two techniques: sort-based or hash-based. To perform a hash-based grouped aggregation, first, the given input is grouped using a hash build followed by a reduce primitive to perform the aggregation. Similarly, based on the selected operation, a different sequence of primitives can be selected for execution.

These primitives are executed either as stand-alone functions or as a pipeline of primitives. Once a query plan with different granularities of primitives is determined, fusing the primitives improves the performance. There are common patterns available in fusing these primitives available and we discuss them in the subsequent section.

5.5.1 Pipeline Patterns

The flow of execution among the primitives is one of the factors affecting overall processing efficiency. The traditional iterator model suffers from the overhead of multiple function calls [81]. One of the alternative models is the compiled-query execution model where the given query is compiled from granular functions [131]. This model reduces the working size as the processing data resides in the device register rather than in memory. Using this execution model, multiple primitives are combined into a single coarse-granular function. There are various pipeline patterns proposed for compiling the primitives [57, 189, 149]. They include *project*, *product*, *select*, *set relation*, and *join*. In Figure 5.10, we show the different pipeline patterns available.

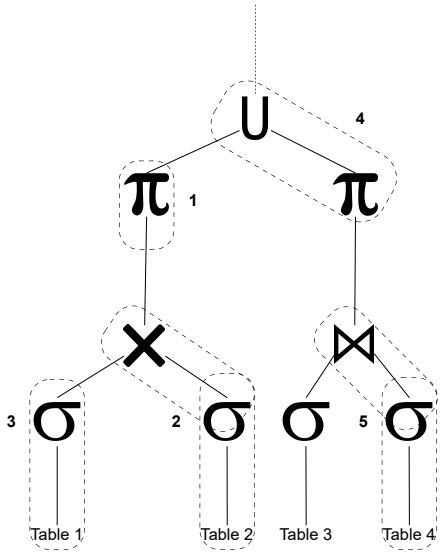


Figure 5.10: Different pipeline patterns

Project The project pattern is the simplest of all patterns, as shown in Figure 5.10(1), it is used for pipelines that only project an input on a set of attributes. Since there is no blocking operation, all primitive functions can be combined in a single tight loop.

Product: In this model, the result is the product of the number of entries in the input relations. This is also a non-blocking pipeline and hence, no intermediate storage of data is required (c.f. Figure 5.10(2)).

Select: The selection pipeline consists of two passes. The passes are similar to the filter passes with the first pass computing the local results and their histograms. Then, prefix-sum is computed over the histogram to determine the total size and index positions. In the second pass, the data are stored in their respective indexes. The selection pipeline is depicted in Figure 5.10(3).

Set Relation: This pipeline is specifically designed for set operations. We show the this pipeline in Figure 5.10(4). It is comprised of three major stages. First, the data is partitioned into smaller chunks. Second, these chunks are processed with the given set operation. Finally, the local resultant values are merged providing the complete result. Additionally for efficient processing, Wu et al. argue the use of combining set operations into a single operation for additional throughput. They call this fusing of operations as kernel fusion stage [189].

Join: As the name suggests, this involves a join operation in the pipeline (c.f. Figure 5.10(5)). Join pipelines are evaluated using custom join operations [57].

Pipeline Operators

Apart from using the pipeline skeletons to assemble an operator chain, Breß et al. list a set of operators used for developing a compiled execution pipeline [39]. These additional operators can be combined with the required pipelines discussed above. Further, Rauhe et al. have given a two-phase model based on thread-level execution [149]. In this model, the first phase is called *compute phase*, splits the input horizontally into logical partitions. After the compute phase, the threads are synchronized in the so-called *accumulate phase* by merging them to produce final results.

5.6 Summary

A DBMS using primitives to implement operations has multiple advantages. Granular functions are combined to form a complete operation and tuning one of the available primitives for a device provides efficiency in multiple operators. We show in this work the different levels of granularity available among these primitives and discuss hardware-based tuning for the finest granular level of primitives. Finally, we discuss the impact of these primitives in the design of a query engine.

We have shown three levels of primitives available for compiling a DBMS operation. The atomic-level primitives are present at the lowest granular level. These are stand-alone functions that are coupled into all the primitives in the upper layers. The atomic-level primitives also have multiple implementation variants based on the underlying hardware used. In the next layer, an abstract level of primitives is presented known as composed primitives. Composed primitives are coarse-grained and need other components for executing a given DBMS operation. Finally, the implementation of a database operator itself is present at the top level.

Though various code optimization strategies are present for implementing these primitives, we also show the various external factors impacting the performance of primitives. We also show the various possible ways to combine the primitives into efficient pipelines for better execution. We give a brief overview of these pipeline patterns available for executing the primitives and the underlying operators needed for forming the overall pipeline. Overall in this chapter, we discuss the following.

1. The primitives in different granularity combined to realize complete database operators
2. Multiple code optimizations available for these primitives
3. Ways to compose primitives into a complete database operation
4. Other influencing factors affecting performance
5. Ways to combine these primitives to realize a complete pipeline.

Though the comprehensive list of primitives can be used to realize the common database operations, it is still not complete. We can extend the list with primitives for other data structures (like graph structures) as currently, we have restricted this survey to relational DBMSs.

Even though these primitives are minimal and even reusable in multiple operations, there also comes the problem of handling multiple implementations. Also, these implementations are necessary as each variant is implemented to suit well with the underlying hardware. In the next few chapters, we will have a deeper look into the performance implication of a hardware-sensitive primitive. Specifically, we study the performance of primitive implementations over the co-processor GPUs. As a start, the next chapter studies the performance of using expert-written GPU libraries for database operations.

6. Tier 1: Task Layer - Realizing Standard Primitives

Though co-processors are ideal candidates to offload particular tasks, we see from previous chapters that it is not a trivial task. Co-processors have their programming models, tunable parameters, and other impact factors that can affect performance. Hence to have optimal performance, we need to either manually tune (as we did with sort-based aggregation in Chapter 4) or use expert-written functions packed as libraries. We explore such libraries written for co-processors in this chapter, studying their performance benefits for query execution. Similar to the previous chapters, we focus our study on GPU.

Numerous researchers have conducted extensive studies to achieve an optimal implementation of a database operator on a GPU. Implementations like group-by [104],[22], selections [155, 15], joins [168, 101], or whole engines [35, 91, 87] have been already discussed in earlier chapters.

As explained, developing such tailor-made implementations requires a developer to be an expert of the underlying device [8]. This makes the approach highly time-consuming but leads to the best performance [41]. As an alternative, many expert-written libraries are available that can be included in a system needing only minimal knowledge about the underlying device.

Parts of this chapter have been based on the following publications:

- H. K. H. Subramanian, B. Gurumurthy, G. C. Durand, D. Broneske and G. Saake, "Analysis of GPU-Libraries for Rapid Prototyping Database Operations: A look into library support for database operations," IEEE 37th International Conference on Data Engineering Workshops (ICDEW), Chania, Greece, 36-41(2021).
- H. K. H. Subramanian, B. Gurumurthy, G. C. Durand, D. Broneske and G. Saake, "Out-of-the-box library support for DBMS operations on GPUs". Distributed Parallel Databases 41, 489–509 (2023).

6.1 GPU Libraries within DBMS

Libraries for GPUs are either written by hardware experts [85] or are available out-of-the-box from device vendors [24]. In this chapter, we survey the existing libraries and identify more than 40 libraries for GPUs each packing a set of operators commonly used in one or more domains. The common benefits of these libraries are that they are constantly being updated to perform the best, repeatedly tested to support newer GPU versions, and their predefined interfaces offer high portability and faster development time compared to handwritten operators. This makes the libraries a suitable match for many commercial database systems to offer GPU support easily. Some examples of systems using libraries for GPU support are SQreamDB using Thrust [172], BlazingDB using cuDF [26], Brytlyt using the Torch library [44].

Since these libraries are an integral part of GPU-accelerated query processing, it is imperative to study them in detail. To this end, we investigate existing GPU-based libraries w.r.t. their out-of-the-box support of usual column-oriented database operators and analyze their performance in query execution. Hence, we survey available GPU libraries and focus on the three most commonly used GPU libraries: Thrust, boost.compute, and ArrayFire to study their support for database operators. Specifically, we explore available operators to determine the library's level of support for database operators, and we present which library operators can be used to produce the usual database operators. Using these implementations, we benchmark the libraries based on individual operator performance as well as their execution of a complete query. Overall within this chapter, we explore the usefulness of GPU libraries in two directions to assess their overall impact.

- **Usefulness:** We look for libraries with tailor-made implementations for database operators. As a result, we can assess the ad-hoc fit of the libraries for database system implementation (cf. Table 6.2).
- **Usability:** We analyze the performance of the different library-based database operators in isolation as well as for queries from the TPC-H benchmark. This is a key criterion for deciding which library to use for a developer's database system (cf. Section 6.5).
- **Portability:** We experiment across two different grades of GPU to see the impact of libraries from the underlying hardware.

The chapter is structured as follows: In Section 6.2, we classify existing languages and libraries for heterogeneous programming. We review existing GPU libraries and identify how to use them to implement database operators in Section 6.3. Next, we detail the way to plug-in various library implementations in Section 6.4. In Section 6.5, we compare the performance of library-based database operators. Finally, we summarize the chapter in Section 6.6.

6.2 Levels of Programming Abstractions

For more than a decade now, the database community has been investigating how to use GPUs for database processing [69]. The interest in GPU acceleration is mainly due to the advancements in its processing capabilities as well as the maturity of programming interfaces and libraries. However, for most practitioners, it is hard to assess the impact of choosing a specific interface or library. To shed some light on the matter, we compare and review current programming interfaces and libraries. As we have seen already in Chapter 2, we broadly categorize them w.r.t. their abstraction level: languages, wrappers, and libraries. We place them as a hierarchy since each entity in a level is developed using the lower-level constructs. Our Figure 6.1 shows examples of these identified levels, which we characterize in the following.

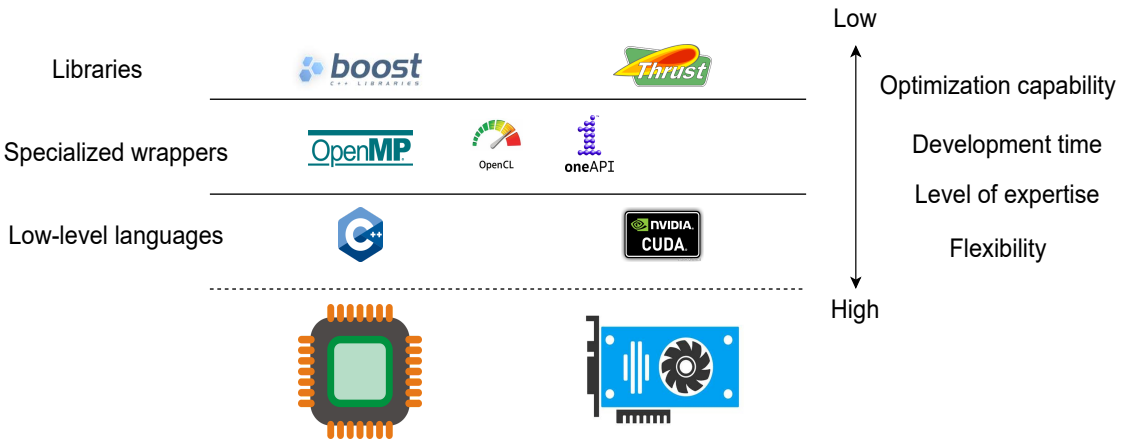


Figure 6.1: Hierarchy of abstraction levels characterizing languages, wrappers, and libraries for heterogeneous computing

We have already discussed the low-level languages and specialized wrappers in Chapter 2 (in Section 2.2.2). In a gist, using such low-level languages (like CUDA) might improve efficiency but comes with the drawback of the high development cost (including usually large size of program code) and requires expertise on the device features. The specialized wrappers (like OpenCL) in the next level offer abstractions to significantly reduce the implementation effort compared to low-level languages, but are susceptible to device changes.

Libraries

At last, there is a plethora of pre-written libraries developed by domain and hardware experts for different devices [126]. Using a library, all internal details of different operator implementations are hidden behind a set of predefined interfaces. Hence, the developer must simply make the right function call based on the underlying scenario. This requires only minimal knowledge of the underlying hardware and implementation. Some examples of libraries include the boost libraries in C++ and the Thrust library for GPUs. Even though these libraries are developed by experts, they are not tailor-made for one underlying use case. Hence, although a generic implementation of operators suits multiple use cases, they can be suboptimal compared to handwritten use-case-specific implementations. Furthermore, due to the predefined interfaces

for operators, one cannot freely combine them for a custom scenario. Instead, we have to chain multiple library calls, leading to unwanted intermediate data movements. Thus, libraries provide high productivity in development with only small necessary knowledge about the underlying device (plus, minimal lines of code) but they come with the drawback of potentially sub-optimal performance from the operator implementations.

Used Abstraction Levels in Database Systems

Various GPU-accelerated database systems are developed using the concepts of different levels. Considering low-level languages, GPUQP [69], CoGaDB [35], and the system of Bakkum et al. [15] use CUDA. For wrappers, Ocelot [91], HAWK [39] are implemented in OpenCL. Finally, many commercial database systems use libraries to implement operators, such as SQreamDB [172] or BlazingDB [26], mainly for their robustness and strong vendor support.

Disregarding their low flexibility, libraries give considerable advantages to the ad-hoc development of a GPU-accelerated database system, reducing its development cost to an acceptable limit. However, with multiple GPU libraries being available, the question remains what library has the best support for the rapid prototyping of database operators, and which library implementation achieves the best performance.

6.3 Implementing DBMS Operators With Libraries

Here is a review of different GPU libraries and their ad-hoc usability for implementing database operators. To this end, from the selected libraries, we discuss the level of support and the offered functions to implement database operators using these GPU libraries.

6.3.1 Review of GPU Libraries

To collect available GPU libraries, we conducted an extensive survey using Google Scholar, and the CUDA website[17]. Generally, there are four different frameworks/languages used by libraries over a GPU namely: CUDA, OpenCL, ROCm, and oneAPI. However, ROCm has been not widely adopted and its performance is similar to that of OpenCL [176]. Next, oneAPI is still in its early stages of development and not all GPUs are currently supported [12]. This shortens our search over OpenCL and CUDA. Between these two frameworks, we found 43 libraries that provide GPU-accelerated operators for various domains. The library details are listed in Table 6.1.

As GPUs are fundamentally graphics machines, their parallel processing is perfect for number crunching. Hence, as shown in Figure 6.2 many libraries focus on image processing (7) and math operations (13). Since GPUs were recently adopted for machine learning workloads[18], only a few libraries are currently present. With databases, libraries that support database operators explicitly are relatively few (5) compared to those supporting general vector operations (such as tensor operations

[17] https://developer.NVIDIA.com/CUDA-zone
[18] https://developer.NVIDIA.com/tensor-cores

Library	Wrapper/Language	Use case	Reference
AmgX	CUDA	Math	https://developer.NVIDIA.com/amgx
ArrayFire	CUDA & OpenCL	Database operators	https://developer.NVIDIA.com/arrayfire
boost.compute	OpenCL	Database operators	[177]
CHOLMOD	CUDA	Math	https://developer.NVIDIA.com/CHOLMOD
cuBLAS	CUDA	Math	https://developer.NVIDIA.com/cublas
CUDA math lib	CUDA	Math	https://developer.NVIDIA.com/cuda-math-library
cuDNN	CUDA	Deep learning	https://developer.NVIDIA.com/cudnn
cuFFT	CUDA	Math	https://developer.NVIDIA.com/cuFFT
cuRAND	CUDA	Math	https://developer.NVIDIA.com/cuRAND
cuSOLVER	CUDA	Math	https://developer.NVIDIA.com/cuSOLVER
cUSPARSE	CUDA	Math	https://developer.NVIDIA.com/cuSPARSE
cuTENSOR	CUDA	Math	https://developer.NVIDIA.com/cuTENSOR
DALI	CUDA	Deep learning	https://developer.NVIDIA.com/DALI
DeepStream SDK	CUDA	Deep learning	https://developer.NVIDIA.com/deepstream-sdk
EPGPU	OpenCL	Parallel algorithms	[116]
FFmpeg	CUDA	Image and video	https://developer.NVIDIA.com/ffmpeg
Goopax	OpenCL	Parallel algorithms	https://www.goopax.com/
Gunrock	CUDA	Others - Graph processing	https://github.com/gunrock/gunrock
HPL	OpenCL	Parallel algorithms & Math	https://github.com/fraguela/hpl
IMSL Fortran Numerical Library	CUDA	Math	https://developer.NVIDIA.com/imsl-fortran-numerical-library
Jarvis	CUDA	Deep learning	https://developer.NVIDIA.com/NVIDIA-jarvis
MAGMA	CUDA	Math	https://developer.NVIDIA.com/MAGMA
NCCL	CUDA	Communication libraries	https://developer.NVIDIA.com/nccl
nvGRAPH	CUDA	Parallel algorithms	https://developer.NVIDIA.com/nvgraph
NVIDIA Codec SDK	CUDA	Image and video	https://developer.NVIDIA.com/NVIDIA-video-codec-sdk
NVIDIA Optical Flow SDK	CUDA	Image and video	https://developer.NVIDIA.com/opticalflow-sdk
NVIDIA Performance Primitives	CUDA	Image and video	https://developer.NVIDIA.com/npp
nvJPEG	CUDA	Image and video	https://developer.NVIDIA.com/nvjpeg
NVSHMEM	CUDA	Communication libraries	https://developer.NVIDIA.com/nvshmem
OCL-Library	OpenCL	Database operators	https://github.com/lochotzke/OCL-Library
OpenCLHelper	OpenCL	Others - wrapper	https://github.com/matze/oclkit
OpenCV	CUDA	Image and video	https://developer.NVIDIA.com/opencv
SkelCL	OpenCL	Database operators & Parallel algorithms	[173]
TensorRT	CUDA	Deep learning	https://developer.NVIDIA.com/tensorrt
Thrust	CUDA	Database operators	[24]
Triton Ocean SDK	CUDA	Image and video	https://developer.NVIDIA.com/triton-ocean-sdk
VexCL	OpenCL	Others - vector processing	https://github.com/ddemidov/vexcl
ViennaCL	OpenCL	Math	http://viennacl.sourceforge.net/

Table 6.1: Libraries and their properties based on our survey

offered by VexCL or Eigen tensor). Even from the available libraries, skelCL and OCL-Library are *boilerplates* to OpenCL without any pre-written functions [173]. These have no direct functions available for implementing database operations. Therefore, we select the remaining ones: boost.compute, Thrust, and ArrayFire for further analysis built over OpenCL, CUDA, and both, respectively. Among these, ArrayFire uses lazy evaluation while boost.compute transforms high-level functions into OpenCL kernel programs, and Thrust operators are transformed into CUDA C functions.

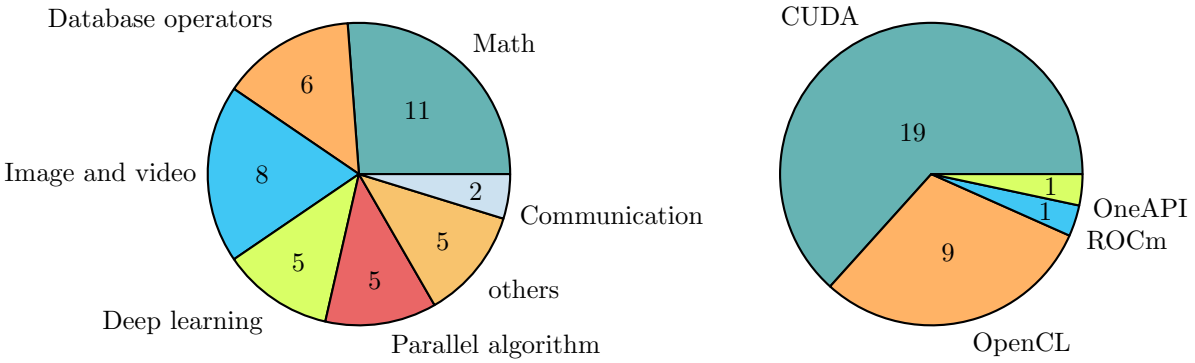


Figure 6.2: Proportion of GPU libraries. Left: proportion of libraries across various application domains. Right: Proportion of GPU libraries and their underlying implementation language.

6.3.2 Operator Realization

Since GPUs are predominantly used for column-oriented analytical queries [7, 139], we consider the operators: projection, (conjunctive) selection, join, aggregation, grouping, and sorting (sort-by-key) for our study. Besides these, we also study the parallel primitives: prefix-sum, scatter, and gather, commonly used for materializing final values. The level of support (i.e., usefulness) and the possible library call for a database operator in the three libraries are listed in Table 6.2. The level of support is determined by the simplicity of the usage of library operators for implementing a database operator. The full support operators have the least interoperability costs and programming effort because they have a direct functional implementation available in the library. In the case of partial support (~), several function calls are needed to implement an operator. Hence, additional effort is required to pass the intermediate results from one function to another before retrieving the final result. Detailed information on the functional support from these libraries is given in the Function-column of Table 6.2, where we map library functions to the database operators.

6.3.3 Summary of Library Usefulness

Overall, when compared to ArrayFire, the other two libraries- boost.compute, and Thrust have multiple alternative implementations for selection. Specifically, ArrayFire does not directly support prefix-sum, nested-loop join, scatter, and gather operations. Regarding functional implementations, it is notable that ArrayFire returns a position list for selections, whereas Thrust and Boost.compute return bitmaps.

Join implementation: A major limitation is that all the libraries lack a custom implementation for specialized joins. They lack direct support for hash tables or merge join of sorted results. Hence, these important implementations must be developed from scratch, or support needs to be added to the libraries. However, the database community has shown great performance for hash-based joins [121] and, hence, these libraries should be extended by custom operators for hashing in future versions.

Database operators	ArrayFire		boost.compute		Thrust	
	Support	Function	Support	Function	Support	Function
Selection	+	where(operator())	∼	transform() & exclusive_scan() & gather()	∼	transform() & exclusive_scan() & gather()
Nested-Loops Join	–	–	+	for_each_n()	+	for_each_n()
Merge Join	–	–	–	–	–	–
Hash Join	–	–	–	–	–	–
Grouped Aggregation	+	sumByKey(), countByKey(),	+	reduce_by_key()	+	reduce_by_key()
Conjunction & Disjunction	+	setIntersect(), setUnion()	+	bit_and<T>(), bit_or<T>()	+	bit_and<T>(), bit_or<T>()
Reduction	+	sum<T>()	+	reduce()	+	reduce()
Sort by Key	+	sort()	+	sort_by_key()	+	sort_by_key()
Sort	+	sort()	+	sort()	+	sort()
Prefix Sum	–	–	+	exclusive_scan()	+	exclusive_scan()
Scatter & Gather	–	–	+	scatter(), gather()	+	scatter(), gather()
Product	+	operator*()	+	transform() & multiplies<T>()	+	transform() & multiplies<T>()

+ full support; ∼ partial support; – no support;

Table 6.2: Mapping of library functions to database operators

6.4 A Connecting Framework for Library Operators

As a next step towards rapid prototyping, it is necessary to execute library operators in a common environment. This is an important step since we want to assess their runtime without any side effects and also allow for interoperability between operators of different libraries if the performance difference is significant. In the following, we describe our generalized task model and adapter pattern that we use for interfacing the libraries.

6.4.1 Task Model

Our task model manages the implementation of an operator within a unified interface for all libraries. In our case, we support ArrayFire and Thrust through CUDA, while Boost.Compute is implemented in C++. We employ these libraries to demonstrate that our task model can achieve cross-platform execution of GPU libraries (both CUDA and C++) within a single codebase. Furthermore, our framework can include a new library or hand-written code with a simple additional wrapper. To support extensibility, we make use of the adapter design pattern. The detail of the programming structure is explained in the next section.

6.4.2 Adapter Pattern

Since the libraries differ in container and operator arguments, our framework needs an easy way to interface with these library operators. A promising feature is that they support the same data type - a vector. In Figure 6.3, we depict the

adapter design pattern that we use for interfacing the library operators. The idea is that the end-user interacts with the target: an interface without implementation. Each library implementation consists of an adapter and an adaptee. As a result, the adapter bridges the incompatibility between the target and the adaptee. For example, the target of the container is a C++ STL vector, which can be converted into a thrust::device_vector<T>, a boost::compute::vector<T>, and an af::array in corresponding adapters. As a result, we can easily switch between operator implementations of different libraries. To this end, the adapter performs library-specific data conversions and includes library-specific additional arguments.

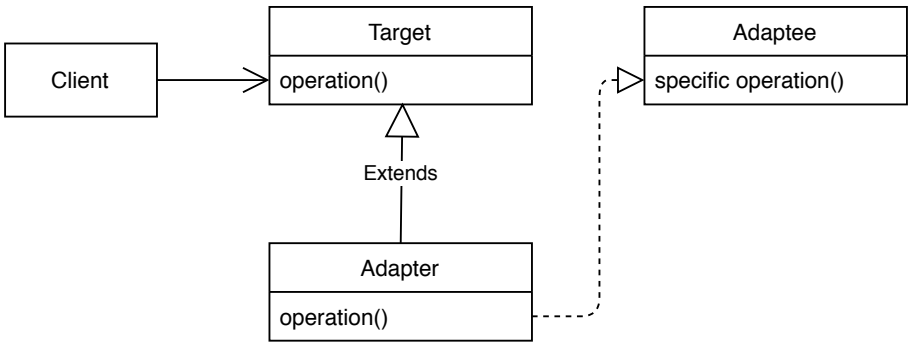


Figure 6.3: Adapter design pattern used for plugging libraries

6.5 Performance Comparison

An essential requirement for using library operators for a database system is that they deliver acceptable performance (i.e., usability). Hence, in this section, we study the libraries' performance for different database workloads. We split our evaluation into two main sections: first, we benchmark the performance of individual operators in micro-benchmarks using a synthetic dataset. Afterward, we measure the overall performance of the libraries with complete TPC-H queries.

Experimental setup: All our experiments are conducted on a commodity - NVIDIA GeForce RTX 2080 Ti with 10 GB memory and server-grade - NVIDIA V100 with 32 GB memory GPU respectively. We use the following library versions: boost.compute-v1.71, ArrayFire-v3.7.2, Thrust-v11.0. All these libraries run on top of OpenCL 1.2 and CUDA 10.1. Our evaluation framework is written in C++ and compiled with GCC 9.3.0 running on Ubuntu 18.04[19].

Dataset: We synthesize datasets for our micro-benchmark. Our synthetic dataset consists of 2^{28} randomly generated integer values unless specified otherwise and the TPC-H dataset is generated with a scale factor of 10. Note: This is the maximum scale factor up to which the execution across libraries is supported. Any larger scale factors are not executed due to space limitations from boost.compute.

[19]The source code is available here: https://github.com/harish-de/cross_library_execution

6.5.1 Transfer Time

Since each library has a custom wrapper for accessing the data present in the GPU, they incur different overhead when transferring data to the GPU. Hence, we analyze the data transfer rate of the individual libraries before analyzing the actual operator performance. We test the transfer time with input sizes ranging from 2^{20} integer values (5 MB) up to 2^{30} integer values (5 GB) and plot it in Figure 6.4.

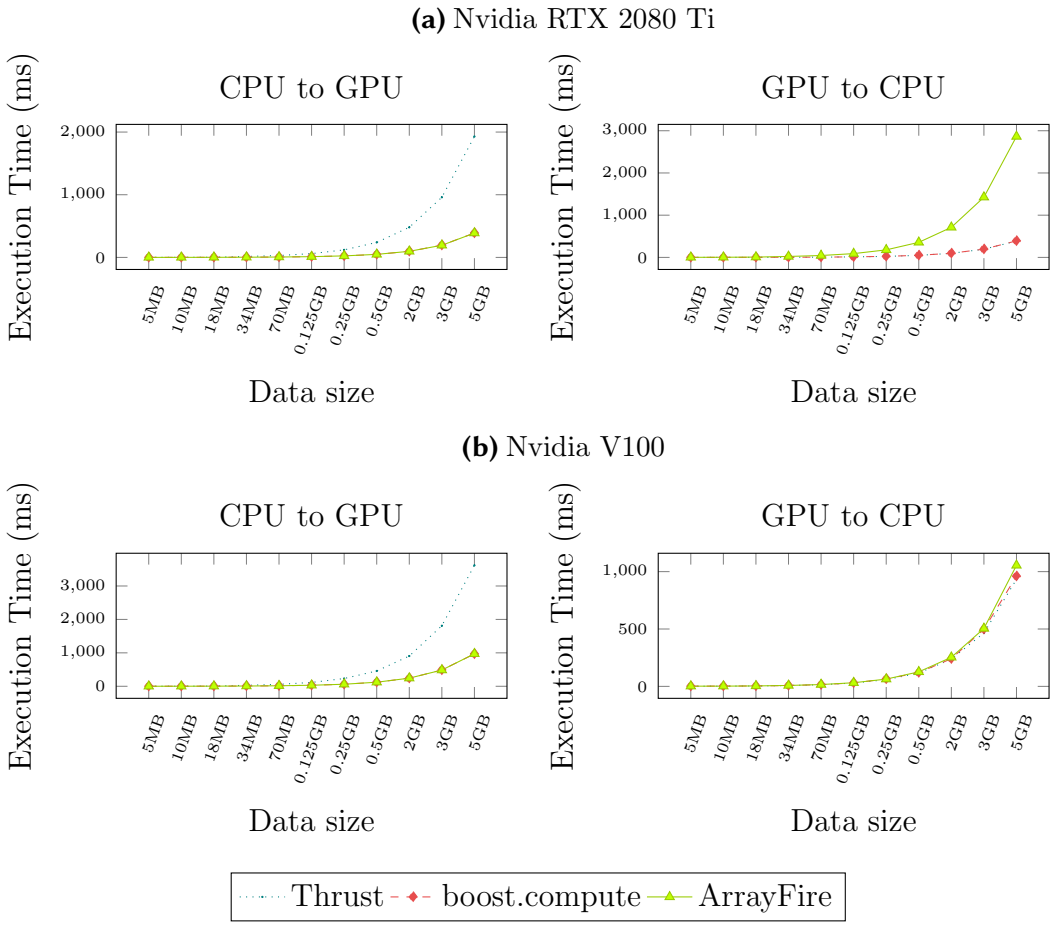


Figure 6.4: Transfer times for different libraries

Foremost, the results show a considerable overhead when transferring data to the GPU using Thrust when compared to boost.compute or ArrayFire on both devices. However, while transferring back, ArrayFire shows poor transfer rates. We believe that the additional steps taken by these libraries in allocating / de-allocating the data lead to such poor performance. Even though transfer rates are significant, buffering input columns can easily avoid this overhead. Furthermore, intermediate or final query results that need to be retrieved from the GPU are usually significantly smaller than the input and, hence, this overhead is mostly negligible. Finally, we see that the transfer rates for V100 are considerably faster than RTX 2080 Ti, even though these two systems use the same PCI-e 3.0 standards for data transfer. However, we see an identical profile for CPU to GPU transfer in both devices - Thrust takes more time to transfer data. We believe this is mainly due to the implementation of the copy operator in this library. Whenever a host-to-device copy is made (which can be

achieved using a simple assignment operator, =), it calls CUB's uninitialized_copy() function that allocates data space followed by data transfer, which leads to poor performance. However, when copying results back, it simply does a data copy on the pre-allocated memory in the CPU space.

6.5.2 Micro-Benchmark: Individual Operators

In this section, we measure the performance impact of operator-specific parameters on the different library implementations. Due to space limitations, we focus on the most common and complex database operators. We exclude sorting, prefix-sum and map as there are already several papers that analyze the performance of these operators [165, 167].

6.5.2.1 Selection

As the selection operator is sensitive to the selectivity of the incoming predicate, we evaluate the libraries by varying the selectivity from 1% to 100%. The final result of our selection is the materialized column of matching values. Since Thrust and Boost.compute create a bitmap and need an additional prefix-sum for materialization, we also show their single performance for creating the bitmap.

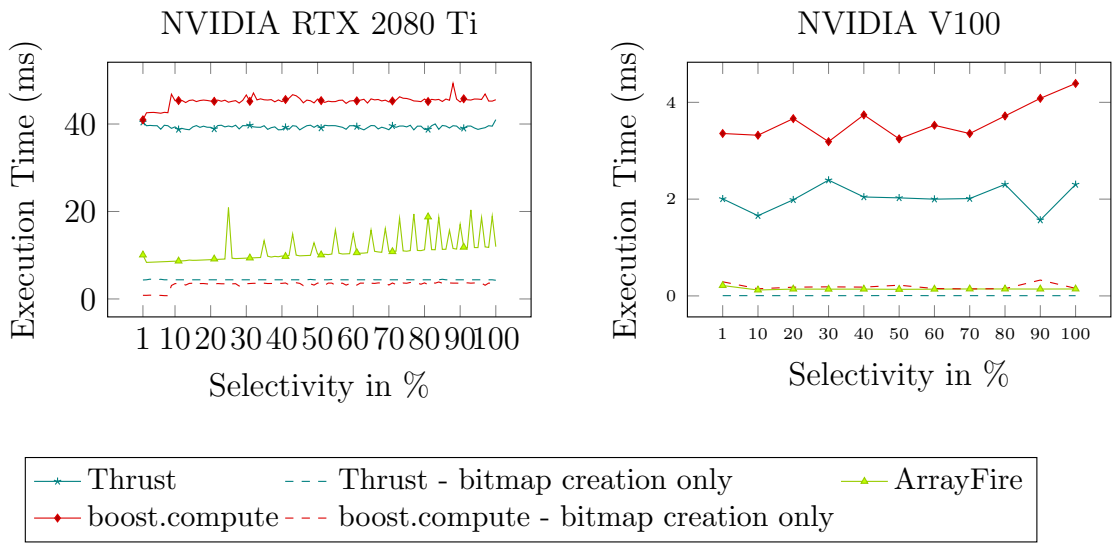


Figure 6.5: Performance for selection with varying selectivity

The results in Figure 6.5 show that the performance of ArrayFire is far better than the performance of Thrust and boost.compute for a materialized filtered column (solid line) across both devices. The main benefit of ArrayFire is that it can directly generate filtered results without additional prefix-sum and gather steps to arrive at the final results. Instead, ArrayFire generates position lists from which we can directly materialize the result. Interestingly, boost.compute has the best performance when creating a bitmap, but is the worst when materializing the result. This is due to the bad performing gather implementation, which is consistent with our following results.

As a result, Thrust and boost.compute are the best choice for multiple predicates on the same table, because combining bitmaps is faster than intersecting position lists. For single predicates and if subsequent operators work with position lists or materialized columns, ArrayFire should be chosen.

6.5.2.2 Group By

In this experiment, we focus on group-by-aggregation, where the performance varies according to the spread of groups. We use a uniform distribution of input values and vary the group size from 1% to 100% where 1% has nearly all values belonging to the same group and 100% contains one group per input value.

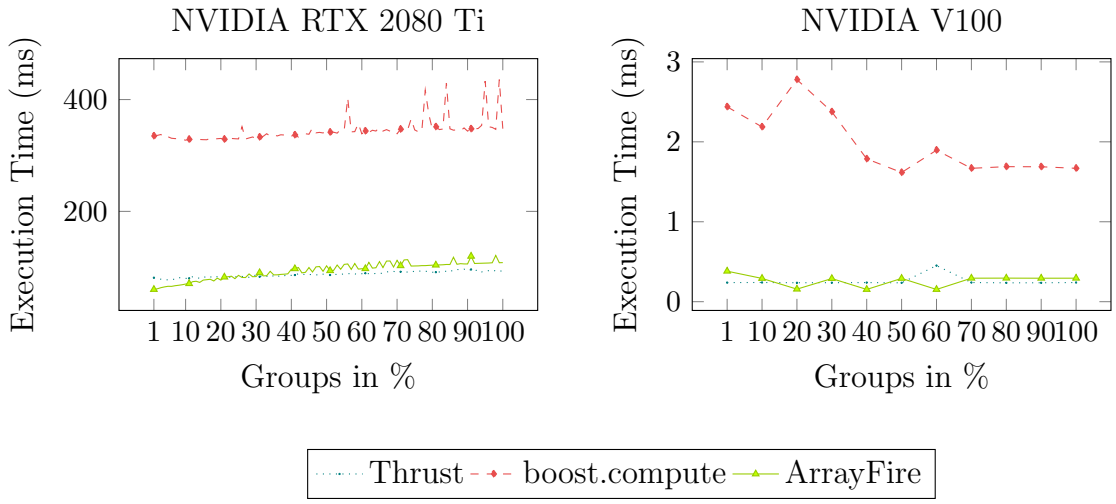


Figure 6.6: Performance for Group-by with varying group sizes

The performance in Figure 6.6 shows that ArrayFire and Thrust have the best performance. Nevertheless, the superior method changes according to the number of groups with ArrayFire performing best for a small number of groups and Thrust performing best for many groups. Further, the performance of V100 shows a drop after a group size of 30%. This shows that V100 can manage multiple data writes efficiently when repeatedly accessing a single location.

6.5.2.3 Joins

Joins being complex operator, generally requires a considerable time for execution. In the case of libraries, we can only support nested loop joins (cf. Table 6.2). Our nested loop join uses for_each() - a function to parallelize an operation based on the given input size.

We measure the performance of join implementations varying the cardinality of the left table (|R|) in a range of 2^1 to 2^{19} using a uniform distribution[20]. We vary the input size, as it directly impacts the degree of parallelism during the execution of a join. Additionally, the execution also depends on the size of the right-side table. Therefore, we keep the size of the right table (|S|) as 2^{28}. Finally, only Thrust and Boost.compute support join operations in the form of a nested loop join. Since ArrayFire does not offer a custom for_each() function, it is not part of the evaluation.

[20]Note that 2^{19} is the maximum data size until which we get a reasonable execution time.

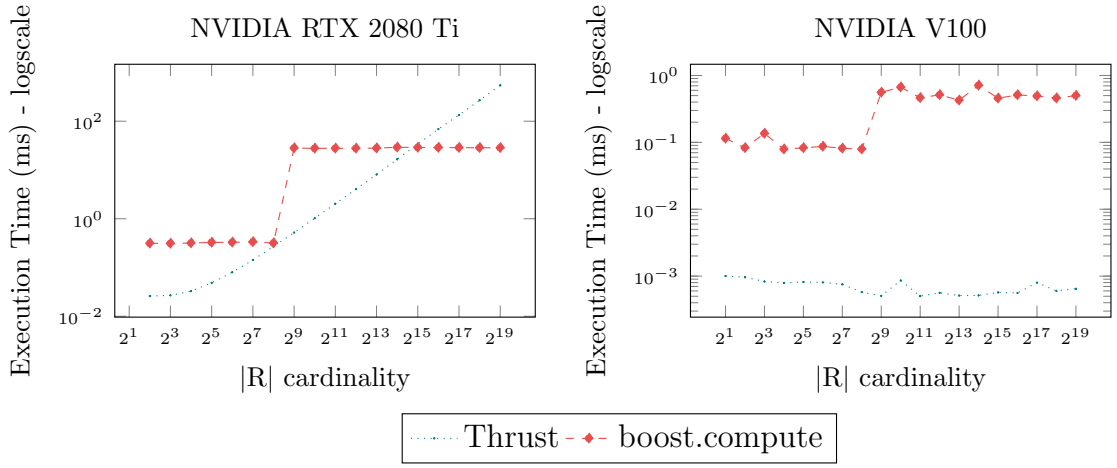


Figure 6.7: Performance for join with varying R-table size

We plot the execution time for joins over the two devices in logscale in Figure 6.7. From the results, we see that boost.compute is comparatively better in terms of parallelization, as its results are linear for a range of inputs for RTX 2080 Ti. However, even with its linear increasing execution time, Thrust is considerably better for smaller input sizes. In contrast, in case of bigger data sizes, boost.compute is superior. Considering the results on the V100, Thrust is clearly the winner as we can see a huge difference in the runtime of the two libraries in Figure 6.7.

Furthermore, results pertaining to boost.compute shows a near-constant growth in performance. This is mainly due to the way the execution spawns threads for executing the custom function. Until a data size of 2^9, the framework uses a different number of threads to spawn, while for any data size greater they process the data using multiple iterations.

6.5.2.4 Scatter & Gather

Our final micro-benchmark is to measure the performance of scatter and gather operations, as they are useful in realizing a hashing operation. Hence, we evaluate the performance of scatter and gather giving as positions the results of multiplicative hashing of the input items. We chose multiplicative hashing as it is a function that is commonly used for scattering/gathering keys into hash tables.

The performance comparison in Figure 6.8 shows clearly that Thrust has a better scatter and gather time compared to boost.compute for RTX 2080 Ti. Here, the poor performance in boost.compute is due to the additional kernel compilation time, whereas Thrust does not have this additional time. Considering the V100 results, we see that scatter operations take longer than gather operations for both libraries. Such poor behavior for scatter shows the overhead of executing random memory accesses on global memory. This behavior indirectly represents the bottleneck in the memory controller of V100 in resolving memory accesses to the global memory.

6.5.2.5 Summary

Overall, we see that there is no one good library that gives consistent performance benefits for all database operations. For selection, ArrayFire is the clear winner

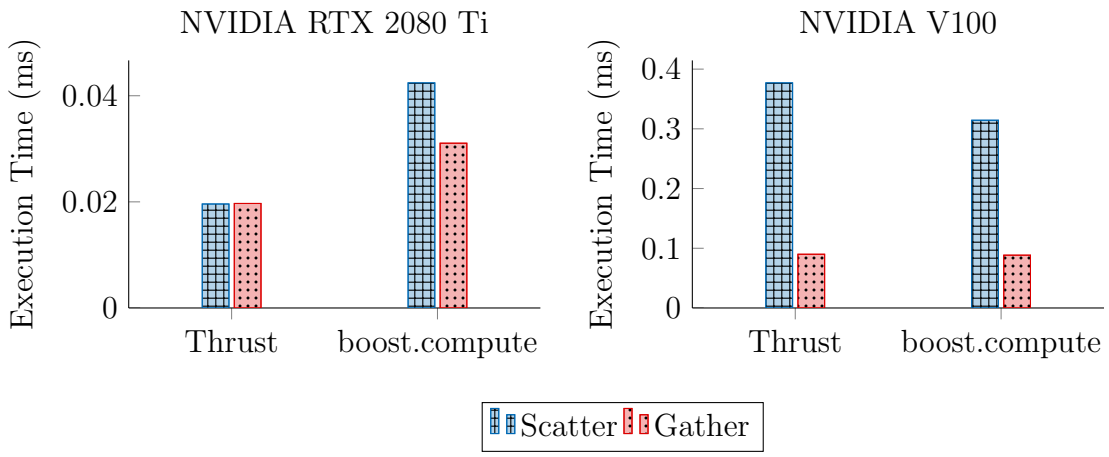


Figure 6.8: Performance for scatter & gather

being nearly 2x faster than the other libraries. In the case of group-by, both Thrust and ArrayFire perform similarly. In case of joins, boost.compute gives constant performance across various table sizes; however, Thrust gives a better performance compared to boost.compute in commodity GPUs. Finally, with scatter and gather operations, Thrust's performance is better in commodity GPU and both Thrust and Boost.compute behave the same in server-grade GPUs. Based on these results, we can now devise the execution of TPCH queries with a single library as well as cross-library calls.

6.5.3 TPC-H Performance

Extending the previous experiments with individual operators, in this section we use the operator implementations to execute complete queries. We use the TPCH dataset with scale factor 10 (SF 10) for executing two query types: group-by (Q1, Q6) and join (Q3, Q4) queries. In the case of the join queries, we substitute ArrayFire with Thrust implementation as the former does not support joins. Finally, we experiment with two different scenarios: single-library and cross-library executions. The results from the execution are explained in the sections below.

6.5.3.1 Single Library Performance

In this experiment, we execute TPCH queries using single homogeneous library calls. The resultant execution time across the considered devices is depicted in Figure 6.9. The results depict only the time taken to execute the operator (as in Table 6.2) and exclude the data transfer time into the device memory.

Group-By: Depending on the cardinality and complexity of the operator clauses (like multiple group-by or multiple conjunctive predicates), the execution characteristics vary. This is evident from the results of Q1 with more time invested in computing group-by aggregates whereas Q6 has most of its time spent in the selections. Though the rank of the fast-performing libraries remains the same across RTX 2080 Ti and V100, there is a significant difference in their performance profile. Specifically, we see that ArrayFire performs significantly better in V100. This is in accordance with its performance difference from group-by experiments (cf. Figure 6.6). Since the V100

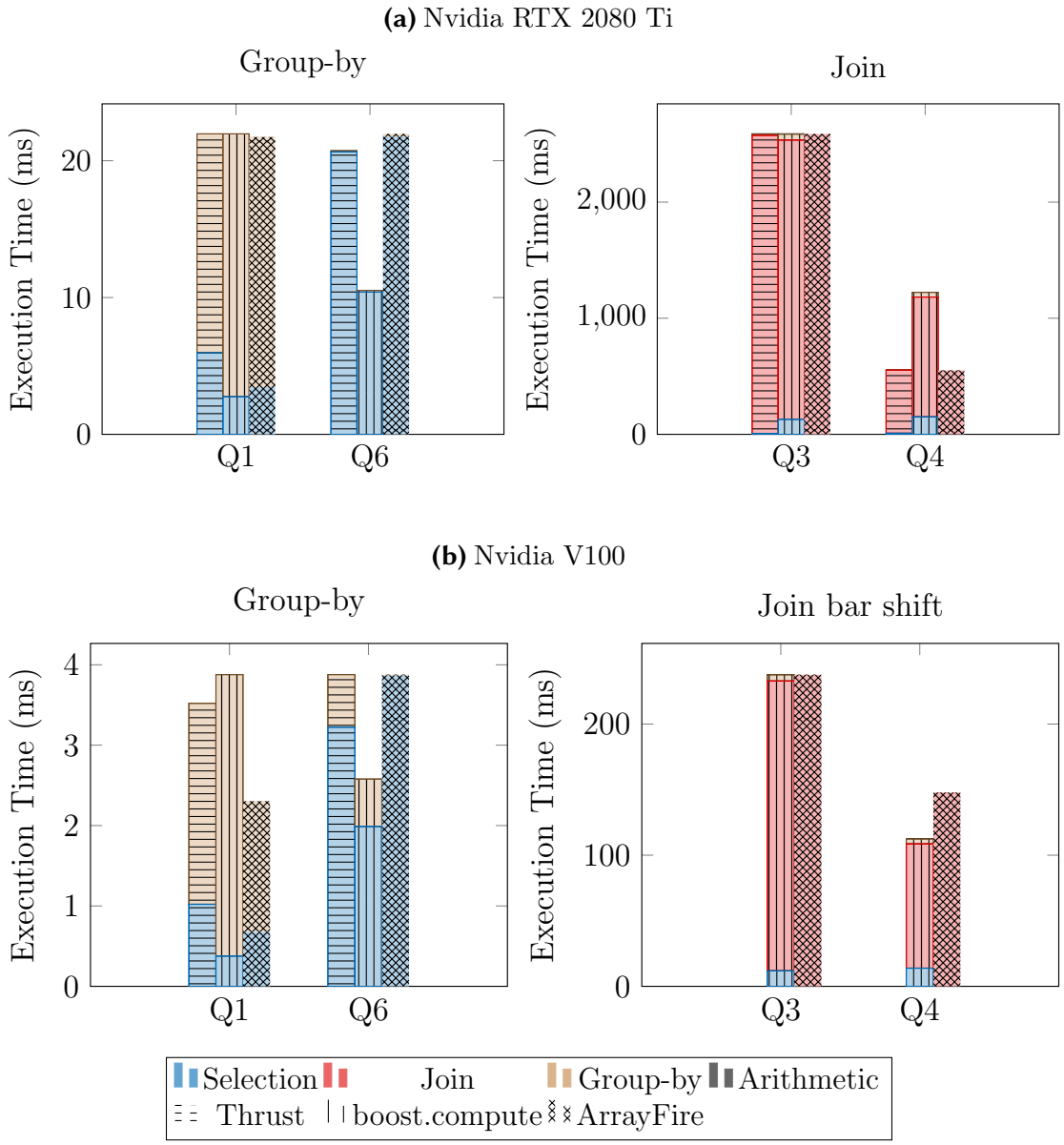


Figure 6.9: Performance of TPC-H Queries

is equipped with much more cuda cores, a larger data size fits the execution in the device, and more aggregates are resolved at a given timespan. Overall, we see that boost.compute performs well with selection operations whereas ArrayFire works well with group-by aggregation.

We see that with Q1 in Figure 6.9 (c), ArrayFire shows an improved performance for group-by aggregation compared to other approaches. However, the throughput profile for Q6 remains the same for the libraries. Still, we see an increase in execution time for group-by aggregation on Thrust and boost.compute for the device. We believe such behavior is due to hardware sensitivity during execution. Specifically, the longer group-by duration is again from the random access to the global memory while grouping the input. As we have seen in Figure 6.8 for V100, random access to global memory is a bottleneck, which is also the case with group-by queries here.

Join: As we have seen earlier, joins are considerably more expensive than other database operations when executed using libraries. This is also reflected in the query results. Almost 90% of the overall time is invested in executing join operations. Unlike with group-by queries, changing devices reflects in the performance profile across the libraries. This is again mainly due to the increase in CUDA cores in V100. Overall, we see that V100 increases the performance by about 10x compared to RTX 2080 Ti. Clearly, we see that libraries are not a suitable solution for executing join operations. The current solutions in research [158][138] are faster than the naive library counterparts. However, we also believe this is mainly due to the lack of support for more sophisticated join algorithms like hash joins and sort-merge joins.

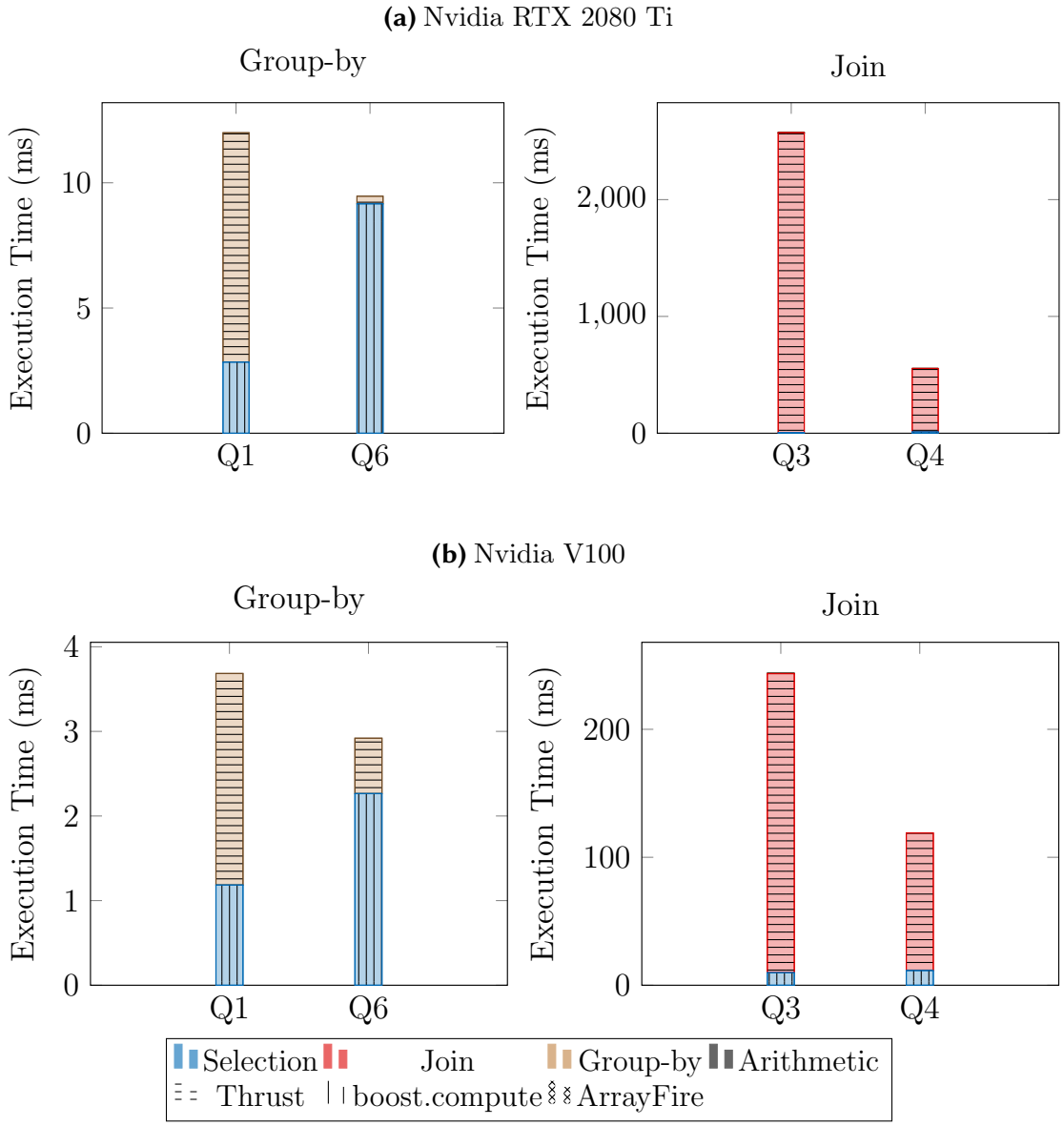


Figure 6.10: Performance of TPC-H queries using inter-library execution

The execution over V100 also has the same performance characteristics as above, except, Q4 shows differences in execution time compared to RTX 2080 Ti. Here,

ArrayFire performs poorly compared to other libraries. Again, we believe the difference in performance arises due to the penalty of random access in V100.

Summary: For group-by queries over RTX 2080 Ti (cf. Figure 6.9 (a), we see a similar performance across libraries. However, with conjunctive predicates, boost.compute is the fastest, followed by Thrust and finally ArrayFire. Since conjunctive predicates in boost.compute and Thrust use bitmaps as intermediate values, the conjunction of these predicates is considerably faster. However, boost.compute's better selection performance for Q1 is compensated by its bad aggregation performance, as we have already seen in Figure 6.6. However, this is not the case with the V100 device. Overall, boost.compute is better for queries with conjunctive selections, whereas for single predicates Thrust or ArrayFire should be used. ArrayFire is better for group-by operations and, finally, the nested-loops join operation is quite expensive on all the libraries. Finally, the libraries are also sensitive to the underlying device (even to the generations).

6.5.3.2 Cross Library Performance

As our final experiment, we evaluate the combined performance of executing TPCH queries across various libraries. To this end, we evaluate the two TPCH queries Q1 and Q3 as samples for group-by and join queries respectively. The execution plan considers the best-performing library for the different operators in the query.

From our previous experiments, we identify that selection is best executed with boost.compute and join & group-by with Thrust. Hence, a reasonable goal is to enable cross-library execution by using our adapter pattern, which we described in Section 6.4.2. Additionally, using mixed libraries for execution also introduces the overhead of translating data from one library format to another. However, this overhead is negligible, as we switch the logical data format instead of physically moving the data. The result of TPCH execution with this mixed library execution is given in Figure 6.10. The overall performance, when compared with the ones in Figure 6.9, shows a considerable decrease in execution time. A decrease of around 25% for Q1 and Q6 can be observed while for join queries, the improvement is not that significant due to the overhead of executing joins.

6.6 Summary

GPUs are more often integrated into database processing both academically and commercially. However, building a system from scratch to support database operators is highly time-consuming and requires expert knowledge. Therefore, we review different expert-written libraries to be used for faster prototyping of a GPU-accelerated database system. Based on our review, we identify 43 GPU libraries out of which 6 support database operators. From these, we study in-depth the support for DBMS considering the following three libraries built over CUDA and OpenCL: Thrust, boost.compute, and ArrayFire. Based on our study, we show that not all database operators are supported out-of-the-box by these libraries and one requires additional re-work for operator realization. Our evaluation shows there is no single library that provides the best performance for all supported database operators. Each of the libraries has its advantages & disadvantages and their functions must be combined

in query execution. we see a lack of support for joins from these libraries making the operator the most time-consuming one. As a final observation, we see a change in the performance of the libraries across different GPU generations. Based on our observations, we conclude the following:

- **Usefulness**: The usefulness of libraries for DBMS is fairly restrictive. Not all database operations are supported out-of-the-box through these libraries. During our study, we especially identified hash-based or sort-based joins to be a pain point, which calls for future work in library implementations.
- **Usability**: Based on our evaluations, not all library functions are performance efficient. Apart from the obvious deficiency of joins, the performance of other operators across libraries varies heavily. Hence, to reach the best performance, users would need to test all libraries and combine their operators based on the query. Since interfacing between libraries is still manual work, future work needs to create a solution for inter-library execution and automatic library/operator selection.
- **Portability**: Libraries can be executed across various devices out-of-the-box with fewer rework. However, our evaluation shows that new devices have a different performance profile for the same operator. Hence, this poses another challenge to the library/operator selection problem.

Overall in this chapter, we use our task layer to realize the various primitives. Still, we cannot directly use these realizations to execute a complete query. For that, we need a query execution model that dictates the flow of execution to gather the final results. To define such an execution model, we must first review the existing ones and understand their internals. Based on this knowledge, we later define our execution model(s) that are suitable for query execution over an abstract co-processor. Therefore in the next chapter, we elaborate on the commonly available execution models for CPU and use their properties to improve their efficiency by developing a hybrid execution model.

7. Tier 2: Runtime Layer - Developing an Execution Model

A key for better execution in a query executor is selecting the right execution model. Hence, we have to define an appropriate execution model for efficient query processing over any co-processor. However, before we define such an execution model, we must study the common models currently being used. To this end, we study the execution models for CPUs, especially the ones used within in-memory systems - compiled and vector-at-a-time execution.

Since these execution model have unique ways of processing queries, they have their advantages and disadvantages. In this chapter, we explore these systems and build the case for a hybrid execution model that merges these two. To achieve this, we would need a special mechanism to cross over partial results from one execution model to another. We solve this problem with the use of our primitives defined in the early chapter. Specifically, we use the pipeline breaker primitives (i.e. hash-probe, aggregate, and hash-aggregate) to facilitate the switch from vectorized to compiled execution. These are illustrated in this chapter as well as the overall performance gain from using our hybrid model.

Parts of this chapter has been based on the following publication:

> B. Gurumurthy, I. Hajjar, D. Broneske, T. Pionteck, G. Saake, "When Vectorwise Meets Hyper, Pipeline Breakers Become the Moderator". In International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB),1-10 (2020).

7.1 Introduction

Compiled execution engines (e.g., Hyper) have an execution time close to hand-written code [132]. However, its faster execution time comes with the overhead of compiling the given query. Even though the compilation cost is negligible compared to the query processing time on large datasets, it could be an overhead for small datasets, as

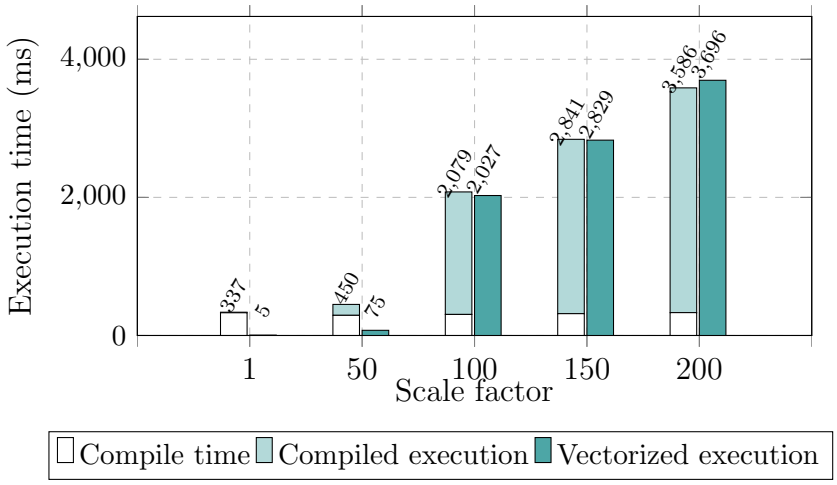


Figure 7.1: Single threaded performance for TPC-H Q6 using vectorized and compiled-code execution

shown in Figure 7.1. Therefore, it is imperative to address the compilation overhead to improve query processing performance.

Even with efficient compilation techniques, there is still an idle window between the time when a query is issued and the start of its execution. Depending on the complexity of a query, this window can be as high as 100ms [132]. Therefore to overcome the cost of compilation completely, it is advisable to use interpreted execution that hides compilation of the query [110]. We follow up on this idea and hide compilation using an interpretation-based execution engine. A vectorized engine with its execution processing on cache-resident vectors is a natural candidate for efficient interpreted execution. Therefore, we focus on hiding the compilation overhead using vectorized execution.

Hiding compilation cost using a vectorized engine (or rather its operators) requires partial results of vectorized execution to be forwarded to the compiled execution. Since compiled execution does not read/write partial results of intermediate operators in a pipeline, we propose to use their natural materialization points i.e., *pipeline breakers* as exchange points from vectorized to compiled execution.

Alternatively, we use the pipeline breakers: aggregation, hash join, and hash aggregation (i.e., group by) for forwarding results from vectorized to compiled execution. Based on the framework by Kersten et al. [106], we present how to adapt the two query engines to inter-operate in query execution. Specifically, we show how compiled execution uses the intermediate results of the vectorized execution engine.

Our approach shows a performance improvement of up to a factor of three by hiding compilation times and several further interesting details in the result.

In summary, we show that with a pipeline breaker as a connector, we can hide the compilation cost using vectorized execution. Overall, this chapter elaborates on:

- *Tether* - a hybrid execution engine that allows for vectorized data processing while query code is compiled.

- Investigation of different pipeline breakers to forward partial results from vectorized to compiled execution after the query has been compiled.
- Evaluation analysis of different TPC-H queries and shows significant benefits of hybrid execution that go far beyond the performance gains of purely compiled or interpreted (vectorized) execution.

The remainder is structured as follows. First, we review related work in Section 7.2. Next, we detail the two state-of-the-art execution models in Section 7.3. The features of our Tether execution model are explained in Section 7.4. We also present in this section, details of using pipeline breakers as switching points between the execution models. We conduct several experiments comparing our Tether framework to standalone compiled and vectorized engines using the standard TPC-H benchmark in Section 7.5. Based on the experimental results, we discuss the key performance factors and the next steps for our work in Section 7.6. Finally, we draw our conclusions in Section 7.7.

7.2 Related Work

To improve on LLVM compilation overhead, Kohn et al. [110] interpret the LLVM IR of a query directly using a custom interpreter. Though similar to us, the work also blends interpreted and compiled execution but still has to generate the LLVM IR to execute a query (which takes about 0.7 ms from their measurements). In our work, we use the query plan and directly execute a query using the pre-compiled operators. Furthermore, the LLVM IR interpreter is fine-grained containing hundreds of custom-written operators for efficient processing. Though this is efficient to switch between the execution modes, they show that the bytecode interpreter is three times slower than the unoptimized machine code. Our work mitigates this overhead by using coarse-grained operator implementation.

Another work on improving performance using materialization and vectorization in compiled execution is explored by Menon et al. [124]. Their relaxed operator fusion model improves performance by introducing staging points to materialize the results of intermediate operators in a compiled code engine. Our work can benefit from these staging points to decide on a more granular level to switch between the execution paradigms.

Due to execution time being comparable with hand-written code, compiled execution is widely adapted, our hybrid system can complement these works by improving their response time [131, 137, 112].

Kersten et al. [106] have built the vectorized and compiled query processing engines using compatible implementations for a fair comparison of these two processing models. Their implementation serves as the base for our work.

There are different models present that combine push and pull-based approaches [55, 183]. However, these systems can also benefit from our hybrid technique to avoid compilation latency.

Other than traditional DBMSs, Spark SQL contains the catalyst optimizer which supports code generation during runtime [10]. Our work can provide improved execution time for this system.

7.3 Preliminaries on In-Memory Execution Models

Query engines are the chassis of a DBMS. All other components revolve around how a query is being processed. The current DBMS engines can be split into either pull-based or push-based engines. The more traditional pull-based engine or the volcano model provides high portability of operations for the cost of materializing all intermediate data [82]. This penalty of materialization was later improved by the vectorized processing engine that uses cache resident data to improve performance [196]. A push-based or code-generation engine avoids such unnecessary materialization of values by keeping the data within registers as long as possible. However, this advantage comes at the cost of compilation time [131]. In this section, we detail these two execution models.

7.3.1 Vectorized Execution

Vectorized processing follows batched execution for query processing. It is an interpreted execution engine where each operator consumes a vector of input values for processing. The size of the vector thereby depends on the cache size of the CPU. Though the operators work on cache resident data, when it comes to operations like group-by, aggregation, or hash-join, intermediate data are materialized into the memory before executing the next operation in the pipeline. This forceful materialization of partial results leads to poor performance. Apart from its materialization overhead, vectorized processing also suffers from the overhead of function calls. For example, a conjunctive selection repeatedly executes selections on each of the columns and combines the results either using a relaxed selection or a conjunction operation [106].

7.3.2 Compiled Execution

To improve data as well as code locality, a compiled query engine generates code directly for a given SQL query [131]. In the generated code, database operators are fused into one pipeline, improving execution time [59]. To facilitate code generation in this model, producer, and consumer functions are included in these operators. Since we are compiling operators together before the start of the execution, we can have arbitrary combinations of input (e.g., a conjunctive selection can be custom-built for the selection criteria based on the input query).

Though the compiled query execution model provides improved data and code locality along with faster execution time, it suffers from the overhead of compiling an SQL query before execution [132]. This compilation time can vary depending on the underlying compiler. Therefore, a poor compiler might lead to a compilation time that is higher than the pure execution time of certain queries.

7.4 Tether: A Hybrid Query Execution Engine

From the introduction to query compilation, we know that compilation time has an important impact on performance. To hide the compilation overhead, we propose to start query execution concurrently with query compilation using an interpreted query engine. Vectorized execution suits this scenario well with its memory-friendly

interpreted processing engine. Furthermore, vectorized engines have been robust, well-maintained, and well-established engines for several years now [195]. Hence, we propose a hybrid engine with both execution models. In the following, we first present the overall workflow of our hybrid query engine –*Tether*– and afterward, we introduce how to switch query execution given the pipeline breakers aggregation, hash join, and hash aggregation.

7.4.1 Hiding Compilation Overhead With Vectorization

The main task for Tether as the connecting engine between compiled and interpreted (i.e., vectorized) processing is to take an input query and transform it into a hybrid query that is started on the vectorized engine and finalized on the compiled engine. For a more detailed description, we use an exemplary query shown in Figure 7.2. The example query has: 2 selections, 1 simple aggregation, 1 group-by aggregation, and 2 equi-joins, which leads to five possible pipelines labeled P_1 to P_5. Given this query, a question arises: how to do a minimal-invasive switching between both execution models as they execute queries differently (cf. Section 7.3). This makes switching hard since a common way of handshaking is needed. There are two important considerations:

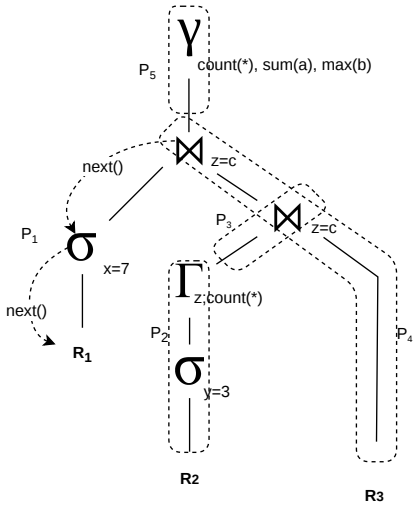


Figure 7.2: A sample hybrid query execution plan in *Tether* using TPCH Q3

we have to decide *how* and *when* to share intermediate results between both execution models. In the following, we answer the *how* by using pipeline breakers and the *when* after finalizing the currently processed vector.

Pipeline Breakers as Switching Points

A vectorized engine materializes intermediate results after each operator. However, these intermediate results cannot be simply shared with a compiled query at an arbitrary point in the query pipeline. This is because compiled execution, as a push-based model, does not consume/materialize intermediate data explicitly from every single operator.

One exception to the *no-materialization rule* of compiled execution is a pipeline breaker [132]. Pipeline breakers force compiled execution to materialize their intermediate values into memory (marking the end of the pipelines P_1 to P_4) before executing the next operator of the pipeline. Hence, the key idea of our hybrid processing model is to use these natural materialization points of a query for switching between vectorized and compiled query execution. From the TPC-H benchmark, we have identified three pipeline breakers commonly present in a query that are useful in connecting vectorized execution with compiled execution: aggregation, hash aggregation (i.e., group-by), and hash join. Hence, by implementing a common data structure for these operations, we can easily switch execution from vectorized to compiled execution.

Inter-Pipeline Switching

With pipeline breakers at both ends of a pipeline, our hybrid processing engine processes at least one pipeline of a query *partially* using vectorized execution. In our example query from Figure 7.2, this is P_1, which materializes results in a hash table for each vector. However, depending on the compilation overhead and the input size, more than one pipeline might be processed by vectorized execution. Hence, we have to compile the pipelines such that their processing can start at any pipeline breaker.

When vectorized execution hits a pipeline breaker and the compilation is done, a switch can happen. The intermediate results processed by the vectorized execution are then materialized and forwarded to the compilation execution. This provides minimal waiting time for compiled execution after the compilation is finished.

Tether's Workflow

The general workflow of our hybrid execution system Tether consists of three phases as given in Algorithm 2. First, we start by compiling the pipelines for compiled execution (`CmplPipeline`) using a compiler thread. The thread adds the instructions for merging the partial results of vectorized execution to the corresponding pipeline before compiling them. Once compiled, the thread sets the `CompilationDone` flag as TRUE.

Second, while compiling, we execute the vectorized operations in the pipeline P_1, which processes values until the following pipeline breaker. In the running example, we materialize the hash table built for the join operation ($z = c$). Additionally, the current index of the input scanned from relation R_1 is also remembered (`ProcessedData`) so that the compiled execution can continue building the partial hash table of vectorized execution.

Finally, once the `CompilationDone` flag is set, we interrupt the vectorized execution, forward the pointers of the hash table and relation R_1 plus the index of already-processed tuples to compiled execution. Afterward, the compiled execution finishes the materialization of the pipeline breaker values based on the current shared index and continues with the next pipelines (P_2 to P_5) producing the final result.

Algorithm 2: Tether execution flow.

```
Data: CmplPipeline,VecPipeline
Result: result
1 bool CmplDone = FALSE;
2 func* CmplFunc = CmplThread.spawn(CmplPipeline);
3 VecPipeline→PipelineBreaker.open();
4 ProcessedData = 0;
5 do
6 |   ProcessedData += VecPipeline.start.read();
7 |   VecPipeline→PipelineBreaker.materialize();
8 while (!EndOfStream & !CmplDone);
9 CmplFunc(VecPipeline→PipelineBreaker.data, ProcessedData);
```

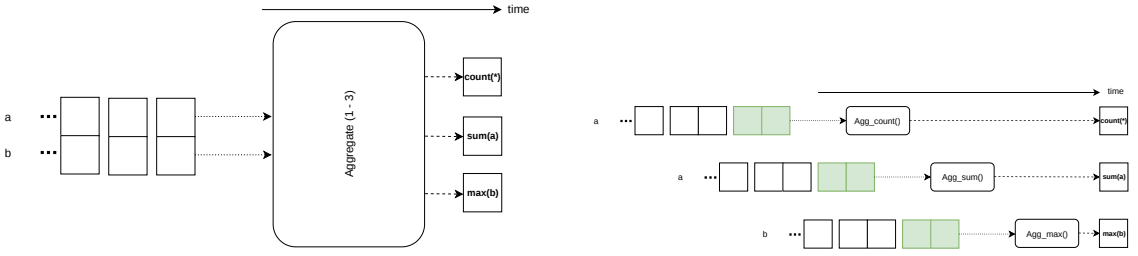
In the following sections, we detail how each pipeline breaker is realized in the two paradigms and how they are executed in our hybrid execution engine Tether. We base the operator implementations of the three pipeline breakers on those from the framework of Kersten et al. [106], as they have proven to be a sophisticated baseline for both models.

7.4.2 Switching via Direct Aggregation

Direct aggregation is the least challenging of all the pipeline breakers we have considered. The aggregate results are computed after a single pass over the input columns. Hence, while switching, only partial results might have been processed by vectorized execution. Hence, we have to forward these partial aggregates from vectorized execution along with the last processed index to compiled execution for continuing aggregation. Notably, a query can have more than one independent aggregation to be computed, resulting in multiple partial results being forwarded from vectorized to compiled execution. Such a query with multiple independent aggregates (e.g., TPC-H Q1 without group by clauses) is computed differently by vectorized and compiled execution.

Compiled Aggregation

Compiled execution generates a custom aggregation function to aggregate all column inputs simultaneously. Therefore at any given instant, as depicted in Figure 7.3(a), all tuple values of the respective columns (`a` & `b` in the figure) are read and aggregated one after another. At any point in time, we obtain the partial results of all the independent aggregates (`count(*), sum(a), max(b)`). For example, the aggregates in the pipeline - P_5 of Figure 7.2 will be computed together.



(a) Compiled execution **(b)** Vectorized execution

Figure 7.3: Direct aggregation with execution models

Vectorized Aggregation

In contrast to compiled execution, vectorized execution aggregates one vector at a time. Therefore, for our running example, partial aggregates will be produced for each processed vector creating the currently aggregated `count(*)`, the aggregated `sum(a)` as well as the aggregated `max(b)` as shown in Figure 7.3(b).

Hybrid Aggregation

Connecting the execution paradigms using aggregation requires that the partial results from vectorized execution can be forwarded to compiled execution. Since compilation can be ready at any point during vectorized execution, predicting the exact number of independent aggregates computed by vectorized execution is a complex task. Instead, our idea is to simply update these partial results in compiled execution. To this end, we add the information about the current column and its last visited index along with the partial aggregates. Figure 7.4 shows the partial aggregates (`p_count`, `p_sum`, `p_max`) of vectors (in green) computed. These partial aggregates are the input for the compiled execution engine, which updates them with the aggregates of the remaining input in a tuple-at-a-time fashion.

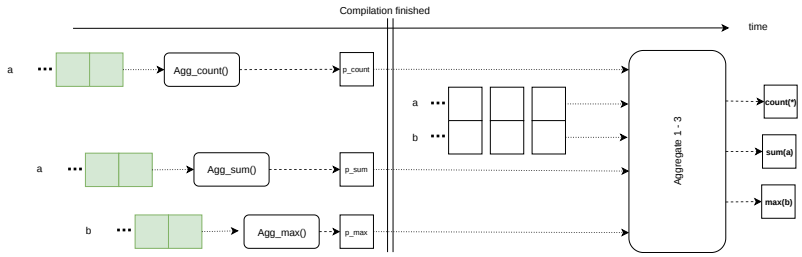


Figure 7.4: Hybrid aggregation.

7.4.3 Switching via Hash Join

A hash join breaks a pipeline by materializing input values into a hash table. Once the hash table is built, it is probed by the next pipeline for join pairs. In our hybrid engine, the hash join becomes the connector between the execution engines. To make the hash join compatible across these paradigms, we keep the same hash table implementation as well as the same hash function. This is mainly to have a common set of methods to insert and probe a key within the hash table. Since the execution switches from vectorized to compiled execution, the hash function that favors efficient processing in compiled execution is chosen. Such a suitable hash table and hash function combination are shown to be chained hashing with the *CRC32* hash function [106]. Even though we have common techniques across the execution engines, we still have variations in the way the runtime populates the hash tables to have the best performance.

In the following, we explain the two possible states that exist when switching from one engine to the other. In the first possible state – i.e., a hybrid hash build, the build phase is only partially done by vectorized execution when compilation finishes. Compiled execution follows up by building a separate hash table for the remaining tuples. The separation of hash built between the engines is to avoid any unnecessary penalties of populating a vectorized hash table in compiled execution. The main issue here is that the engines populate the hash tables differently. On the one hand, threads in hyper cooperatively populate a single hash table. On the other hand, threads in the vectorized engine populate their private hash table first followed by a global merge step. In the second possible state – i.e., a hybrid hash probe, vectorized execution has already finished building the hash table, and now the compiled execution probes the hash table.

Hybrid Hash Build

During switching, the vectorized execution might have only built a partial hash table. Inserting into vectorized hash tables from compiled execution is not efficient. In this case, compiled execution first reads the remaining tuples aggregating them in a separate hash table.

From our example query in Figure 7.2, considering the switching point is at the join $R1.z = R3.c$, $R1.z$ column values are present in two different hash tables. Once built, both hash tables (the vectorized and compiled hash table) will be probed (on $R3.c$ from our example) for each tuple. Therefore, compiled execution must include code to probe the hash table of vectorized execution to find join pairs.

Hybrid Hash Probe

Vectorized execution, due to its working granularity of a whole vector, requires a sequence of steps for hash probing. First, the hash values are computed for input vectors. Next using the hash values, the target match locations for vectors are identified. Finally, the values in the target locations are compared to identify join pairs. We circumvent these steps in compiled execution by directly computing the hash value of a given key and probing through the table using the traversal functions of chained hashing.

```
if (ht.contains(o_orderkey[i]) && (
    ↪ name = ht1.findOne(o_custkey[
    ↪ i]))) {
entries.emplace_back( ... );
}
```

Listing 7.1: Compiled hash probe

```
/* hybrid Code */
if(ht.contains(o_orderkey[i]) {

// Vectorized probe
runtime::CRC32Hash h1;
uint64_t output = h1(o_custkey[i]);
for (auto entry = vwHT.find_chain(output);  entry!=runtime::Hashmap::
    ↪ end();   entry = entry->next) {
        if (entry->o_custkey == o_custkey[i]) {
                entries.emplace_back( ... );
        }
}

// Compiled probe
if(ht0.contains(o_custkey[i])) {
        entries.emplace_back( ... );
}
}
```

Listing 7.2: Hybrid hash probe

The inclusion of the additional probe steps of the vectorized hash table increases the lines of code compared to the naive code given in Listing 7.1. In Listing 7.2, we depict all hybrid probing steps. For every input key, we first look into the partial hash table of the vectorized engine followed by probing the remaining values in the hash table of the compiled engine.

7.4.4 Switching via Hash Aggregation

The final pipeline breaker that we consider in our work is a hash aggregation (`z; count(*)` in running the example). Like the hash join, a hash aggregation does two passes over the input data for computing the results.

In the first pass, the input is hashed and values are grouped in the hash table. In the second pass, the aggregates for each of the groups are calculated. Furthermore, with multi-threaded execution, each thread has to do these two passes plus an additional merge stage, which is necessary to aggregate the partial results from all threads.

Hybrid Hash Aggregation

To connect vectorized with compiled execution, we have to forward the partitioned group values from vectorized to compiled execution to aggregate them. Since hash aggregation follows a similar execution of hash join, the compiled execution also starts with building its hash table for the remaining input values along with aggregating them. Finally, once the results for compiled execution are ready, we update them with the partial results from vectorized execution. Additionally, in the case of multi-threaded execution, compiled execution also takes care of merging the partial results from all the individual threads of vectorized execution.

7.5 Experiments

In this section, we compare the performance of our hybrid system Tether with a stand-alone compiled and vectorized execution for different data sizes and present our observations. To this end, after a short introduction of the experimental setup, we first discuss the incurred compilation overhead due to our added switching points (e.g., from Listing 7.2). Afterward, we investigate the benefits and drawbacks of Tether on simple and complex queries with several pipeline breakers.

7.5.1 Experimental Setup

We conduct our experiments on an Intel® Xeon® Gold 6130 CPU. The machine runs an Ubuntu 18.04. with CLANG version 6.0. For our execution, we parallelized the queries using 16 threads.

Comparable Implementations

As mentioned earlier, the compatibility of data structures and operator implementations between compiled and vectorized execution is a key factor for our system. To this end, we use the operator implementations of *Tectorwise* and *Typer* with their common data structures[21]. Our Tether as a hybrid engine uses the vectorized operator implementations of Tectorwise and the compiled pipeline implementations of Typer and adds custom code for the switching points. The stand-alone engine of Typer does not contain direct code generation or compilation. Hence, we also directly compile the LLVM code of a target query and link it with our execution in runtime, and record the time as compilation time. To this end, we use clang 6.0 for compilation (with the -O3 flag) and for building the machine code. Similar to Hyper, we consider only the time taken to compile the code for our experiments.

[21] https://github.com/TimoKersten/db-engine-paradigms

Workload Description

We use the TPC-H benchmark with scale factors ranging between 100-200 and its queries Q1, Q3, Q6, and Q18 for our experiments. These queries specifically contain the different pipeline breakers that we discussed earlier in the exemplary query plan. we use the query plan of compiled execution for our execution [22]. For the comparison, we measure the execution time for our system and compare it to the runtimes of vectorized and compiled executions. Please note, we *always* include compilation times into the reported execution times of compiled execution and our hybrid system.

Experiments

From the selected TPC-H queries, we derive three important experiments that show the benefits and drawbacks of our hybrid engine Tether. In the first experiment, we are interested in the additional compilation overhead due to our compiled switching points that add extra code to the compiled engine. In the second experiment, we investigate simple queries like Q1 and Q6 that contain a single pipeline and compare Tether's performance to the performance of the other two standard execution engines. Since these two queries have rather small and short pipelines, switching between both models should add considerable overhead, and we are interested in whether we can still benefit from hybrid execution. In the third experiment, we look at queries with several pipeline breakers, i.e., Q3 and Q18, and investigate good switching points for Tether in order to outperform single-engine performance.

7.5.2 Hybrid Compilation Overhead

Since Tether includes additional compiled code to merge the partial results of its vectorized engine into its compiled engine, we first investigate the resulting overhead for compiling the queries at the first pipeline breaker. Subsequently, we investigate how the compilation time changes when compiling the query more cleverly at later pipeline breakers.

Assessing Compilation Time

The TPC-H queries we presented above have different pipeline breakers in their initial pipelines: Q1 contains hash aggregate, Q6 has a direct aggregation and Q3 & Q18 use hash joins. The comparison of compilation times for naive (i.e., Typer), hybrid (i.e., Tether's compilation time when the vectorized engine runs concurrently), and hybrid without any concurrent execution for these queries are depicted in Figure 7.5. We use a single thread for compilation and use clang for compiling the C++ pipeline code[23]. We see that concurrent processing has a minor impact on the compilation time, and this is mainly due to the overhead of handling the compilation thread. We can also see that the overhead of compiling code with additional merge instructions is between 14ms to 42ms. The worst compilation time is recorded for Q1 or in other terms for merging results of the hash aggregation. This is mainly due to the

[22] Plans provided by hyper-DB interface: https://hyper-db.de/interface.html

[23] Our experiments have shown that, of course, different compilers will lead to different compilation times. However, the ratio of compilation times between the stand-alone and hybrid engines is always the same.

additional aggregation step required to merge the partial aggregates of vectorized execution with the aggregates of compiled execution. Furthermore, due to parallel execution, we have multiple partial results from vectorized threads which have to be merged with the results of compiled threads.

On the other hand, Q6 (i.e., direct aggregation) leads to the smallest compilation overhead. In this case, we simply perform one additional aggregation step to merge the results of compiled execution with vectorized execution. The merge of the remaining queries (Q3 and Q18) simply includes the probe instructions of vectorized execution along with the probing of compiled execution, which incurs only a small overhead for compilation.

Although we have additional costs for compilation, we will in the following hide the compilation totally with vectorized execution. Furthermore, depending on the data size and pipelines in a given query, vectorized execution might process more than one pipeline before we finalize compilation. Therefore with such workloads, we refrain from compiling the pipelines that are completely processed by vectorized execution. This reduces the overall execution time from the naive compilation of a complete query.

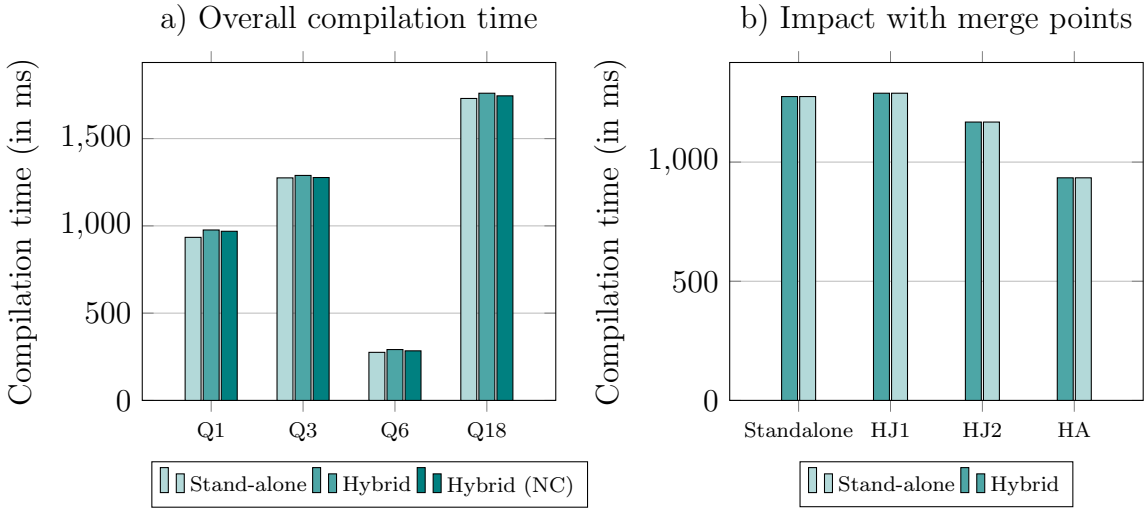


Figure 7.5: Stand-alone vs. hybrid compilation time.

Compilation Time Comparison of Pipelines With Partial Merge Instructions

To understand the impact of compiling only partial pipelines of a query, let us consider the pipelines of Q3. In total there are three pipelines and pipeline breakers in the query: two hash joins (HJ1, HJ2) followed by a hash aggregation (HA) before producing the results. By using these different pipeline breakers as the switching points, we incur different compilation times. The compilation time w.r.t. the different pipelines for Q3 is given in Figure 7.5. By compiling partial pipelines, we outperform the naive compilation significantly leading to better exploitation of compiled query performance. However, this in turn means that we have to delegate a complete pipeline to vectorized execution. Hence, the question that we want to answer in Experiment 3 is: which of these pipelines provide the best trade-off between compilation time and vectorized execution.

7.5.3 Single-Pipeline Queries

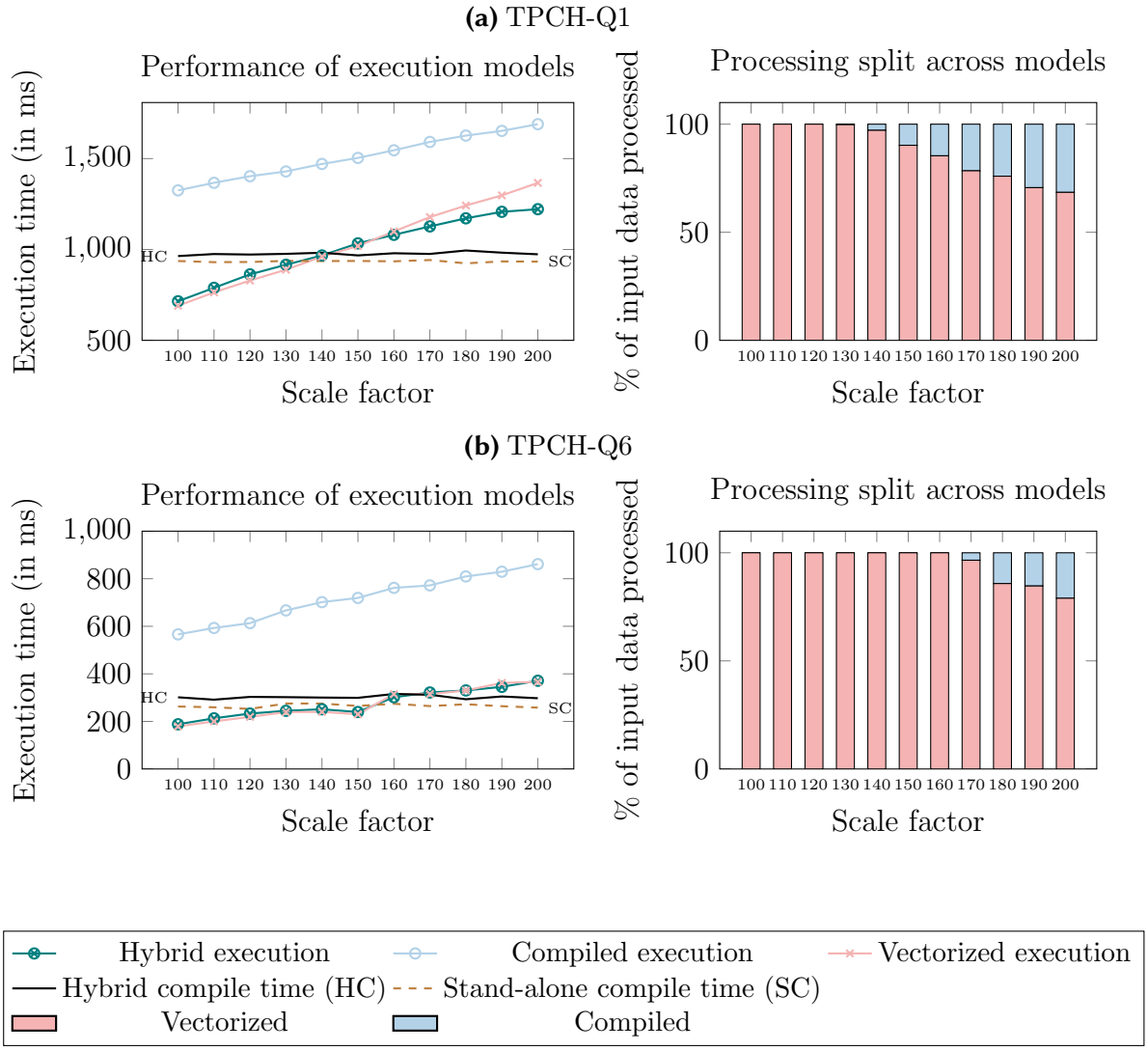


Figure 7.6: Execution profile for single-pipeline queries

In this experiment, we compare the execution times of our hybrid system with the stand-alone compiled and vectorized execution engine for single-pipeline queries. We chose queries Q1 and Q6 because they have only a single pipeline to process. Therefore, we have only one possible merging point, i.e., we merge the partial results of vectorized execution into the partial ones from compiled execution for the complete query. For these queries, a high compilation time would lead to the circumstance that vectorized execution processes the input data completely before we could switch the processing engines. Such characteristics are visible in the results for the queries in Figure 7.6. In Figure 7.6(left), we compare the runtime of the three different engines for different scale factors of the TPC-H benchmark. Furthermore, we depict the compilation time for all queries, which is rather stable across different scale factors. In Figure 7.6(right), we break the execution time of Tether down to show the ratio of processed tuples in the vectorized engine compared to the remaining tuples that are processed with the compiled engine of Tether.

Due to the compilation time, we see that the switching points for the queries in Figure 7.6(a) & (c) are around scale factor 140 and 160, respectively. As expected, our system follows the performance of vectorized execution until the switching point. After the switching point, it is clearly visible that our hybrid system deviates from vectorized execution due to an increased fraction of tuples that are processed in the better-performing compiled engine. For Q1, the change in performance is slightly affected by the additional merge step. However, this additional impact is negligible considering the data shared among the two systems. We see from Figure 7.6(b) that even with processing only 30% of data in the compiled engine, we already outperform both engines. A similar case can be also seen for Q6 (cf., Figure 7.6(c)). Since, it is a comparatively simple query, compiled execution processes only up to 20% for the chosen scale factors of input data at the moment. However, with increasing the scale factor, the amount of data processed by compiled execution will also increase and, hence, improve Tether's performance. Since only a fraction of the total input is processed by compiled execution after the cut-off point, our execution time improves accordingly. Overall, our Tether approach is 2.3 times faster than stand-alone compiled execution and 1.2 times faster than vectorized execution after the cut-off point. Now that we have seen the performance of single pipelines, we experiment on the impact of several switching points in a query in the next section.

7.5.4 Informed Switching Points

Queries with multiple pipelines require a closer look for selecting the right switching point. To better analyze the importance of the switching points, we show the performance difference of Q3 executed with all possible switching points in Figure 7.7.

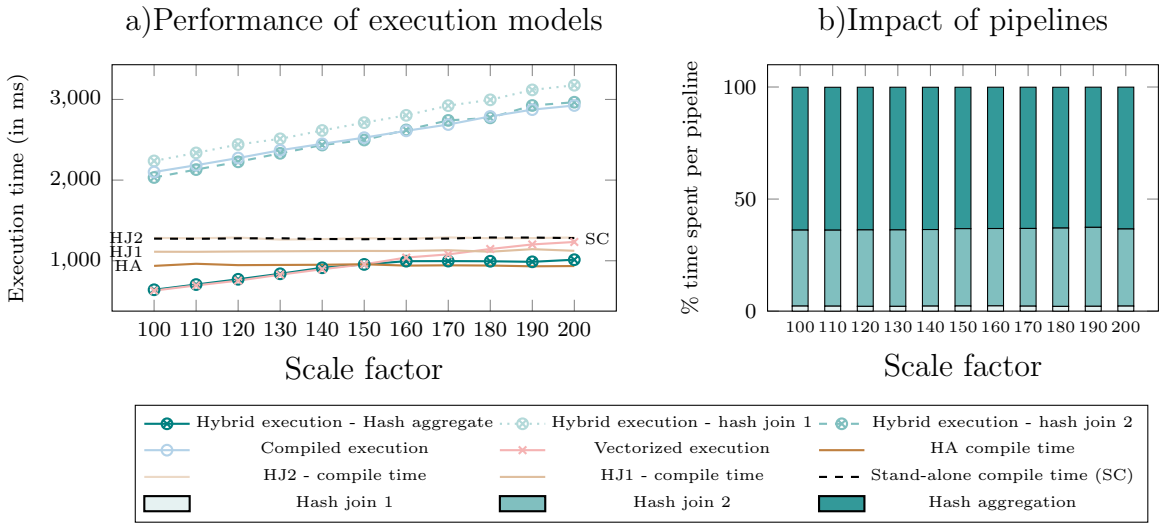


Figure 7.7: Illustrating the impact of switching points using TPCH-Q3.

As discussed, Q3 is composed of three pipelines. Therefore, depending on the data size, vectorized execution could have finished some of the pipelines before compilation was complete. By injecting the switching point at the hash join, we see that we are either worse than or on par with the stand-alone compiled performance. This shows two implications: 1) a hybrid hash probe in the compiled engine of Tether incurs a performance penalty, and 2) the vectorized engine of Tether is not

completely busy until the compilation has finished. To have a better picture of the most performance-critical pipeline, we measured the execution time for individual pipelines and presented them in Figure 7.7(b). We see that 60 % of Q3 is spent on the final pipeline (i.e., the hash aggregation). Using this pipeline as the switching point, we see a drastic improvement in performance. Since hash aggregation is the final pipeline in Q3, the execution characteristics are similar to those of Q1 and Q6. When compilation takes longer than vectorized execution, Tether follows the performance of vectorized execution. However, if the compilation finishes before vectorized execution, Tether outperforms the performance of vectorized execution due to compiled execution. Thus, similar to the case of Q1 and Q6, our overall improvement by Tether is about a factor of 2 compared to compiled execution and 1.5 to vectorized execution after the cut-off point.

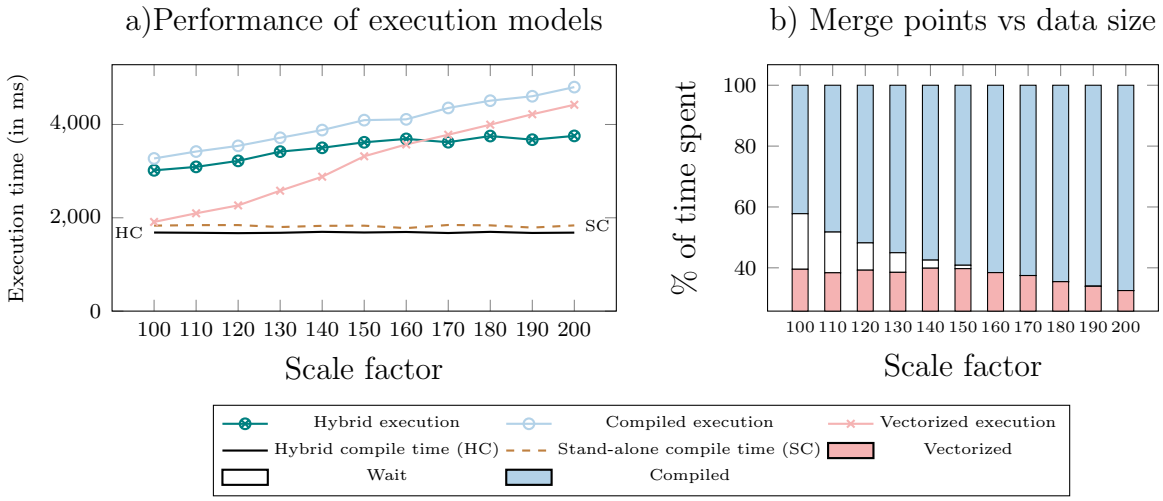


Figure 7.8: Illustrating wait time while switching using TPCH-Q18

As a final and most challenging query, we investigate Q18 with 5 pipelines to show the impact of changing the switching points between the engines. From Figure 7.8(a), we see that even with less than scale factor 100, compilation time is less than the total execution time of vectorized execution. Hence, during the transition from vectorized to compiled execution, Tether processes only a partial result of some internal pipeline. This connection pipeline, similar to that of Q3, has to be identified.

With compilation time being static for all the datasets, depending on the data size, vectorized will be processing a different part of the query pipeline (the bigger the amount of input data, the fewer pipelines have been already processed at the time of switching). Below are the different pipelines available within Q18:

1. Build customer hash table
2. Group by lineitem
3. Build lineitem hash table
4. Probe orders over customer and lineitem tables and build the final table
5. Probe lineitem and compute aggregates

Out of these, the first pipeline on the customer table has only a few values to process. Therefore the second pipeline with grouping of the lineitem table is taken as our cut-off pipeline-breaker. The execution of this query plan shows that we have the performance improvement from both engines only after SF 160. To better understand the impact of the switching points, we depict the percentage of time spent on each of the execution engines in Figure 7.8(b).

For scale factors 100 to 150, vectorized execution is able to process its given pipeline completely before the hybrid query is compiled. Therefore for these scale factors, we have a worse execution time than vectorized execution. With an increasing data size, this gap reduces as vectorized execution has to process more data.

One way to reduce the wait time is to move the switching point to the next pipeline, i.e., building the hash table of the lineitem table. However, this will not be an optimal choice. The subsequent pipeline, i.e., the order's probing pipeline, probes over both the built hash tables of lineitem aggregates and customer from vectorized execution. With the first three pipelines executed by vectorized execution, compiled execution has to issue another probe call to the built customer table as well as a partially built lineitem table. These probe calls, as the results of Q3 show, are way too expensive. Hence, we keep using the second pipeline breaker as a switching point. Thus, with a penalty of smaller wait times, our Tether execution outperforms compiled execution by a factor of 1.5. Once we bridge the execution gap, we outperform vectorized execution by a factor of 1.2.

In summary, complex pipelined queries require a complete analysis to decide on the right switching points. Therefore, such queries could benefit from an optimizer providing these switching points during runtime. We consider this as our future work.

7.6 Discussion

From our experiments, we show that our approach has achieved the overall best execution time. Our system is the fastest or it is on par with the execution time of vectorized execution, which completely hides the compilation time. On average, we are three times faster than the combined execution time for compilation and compiled query execution. We also outperform vectorized execution after switching to compiled execution by up to a factor of two for bigger datasets. Based on the results, we have the following observations and discussion points:

Hiding compilation: An overall positive result of our experiments is that Tether effectively hides compilation time. Hence, since starting the processing concurrent with compilation reduces the overall amount of data to be processed, the overall performance of Tether is above the performance of stand-alone systems.

Merging overhead: Despite the impressive performance of Tether, its main drawback is the inclusion of an extra merging step in the pipeline breaker. This leads to a sometimes increased compilation time and adds a processing overhead to Tether's compiled execution engine. Such overhead is visible in the performance difference between stand-alone compiled versus Tether's compiled execution in Q1 (cf., Figure 7.6(a)).

Hence, one significant optimization space is the development of efficient merging strategies, that allow both engines to handshake more efficiently on switching.

Furthermore, especially when using the hash probe as a switching point, it is visible that the functional invocation of vectorized primitives inside compiled execution affects performance negatively. Therefore, important future work is to implement custom pipeline breakers that can be optimally executed in vectorized execution and read without overhead within compiled execution.

Performance critical pipelines: Another observation from Q3 is that different pipelines contribute to a different extent to the overall execution time. Pipelines that are composed of complex operators and process a large input dataset are promising for being started in vectorized execution. By identifying these pipelines and sharing their results, we can improve the overall performance significantly.

Selection of the right pipeline breaker: For the best performance of our hybrid system Tether, an optimal switching point is required. Such an optimal switching point is present at the cross-point of the performance impact of vectorized processing of the input query, input data size, and compilation time. Hence, vectorized execution may process more than one pipeline. As a consequence, an optimizer should choose the right pipeline breaker to switch between both engines of Tether.

Data size vs. compilation time: For datasets with considerably smaller data sizes, vectorized execution might complete processing the workload before compilation finishes. In this case, it is not beneficial to start compilation. Therefore, the query optimizer should decide on the right execution model based on the input sizes and the query compilation time.

7.7 Summary

In this chapter, we aim to solve the biggest pain point of compiled execution: compilation times. To this end, we investigate whether its competing query engine, a vectorized engine, can help in effectively hiding compilation times. This leads to our hybrid engine Tether which starts query processing in its vectorized engine and after compilation is finished, continues query processing in its compiled engine.

For switching between the execution paradigms, we use pipeline breakers as a natural switching point. We realize such a data sharing between the execution engines using the pipeline breakers: aggregation, hash aggregation, and hash join operators.

From our results, we show that by switching from vectorized to compiled execution we reach the best performance compared to both, vectorized and compiled execution. Hiding compilation time using vectorized execution can improve performance by up to a factor of three compared to stand-alone vectorized or compiled query performance. Therefore, our *Tether* framework shows that query processing can be significantly improved when combining the features of the two state-of-the-art approaches. However, our results also show that choosing the right pipeline breaker for switching between the engines is of significant importance. Hence, the query optimizer has to be extended in future work to incorporate these design decisions.

Thus, in this chapter we investigate the two popular execution models to adapt them for our cross-device execution. However, these models have inherent disadvantages which can be solved with a hybrid model. Still, this solution is tailor-made for CPU-based systems and cannot be easily extended to a cross-device system. Specifically, we are at a disadvantage in supporting compiled query execution for any co-processor (for example: compilation time for FPGA is very poor). Similarly, we are also at a disadvantage with co-processors in terms of available on-chip memory. Usually, these memory spaces are quite limited and do not hold a complete database (more explained in the next chapter). Hence, we explore a solution that considers these problems while supporting query execution over an abstract co-processor. Such a runtime is discussed in the next chapter.

8. Tier 2: ADAMANT – A Pluggable Query Executor

Thus far, we have looked into primitives, their use in isolating functions, and existing execution models. However, to develop a complete co-processor pluggable query engine, we need to combine these two along with a pluggable interface for co-processor drivers. In this chapter, summarize the solutions so far, and extend them to have a complete pluggable query engine. Specifically, we use the primitives as the means for plugging operators and the existing execution models to develop various execution models. Parts of this chapter has been based on the following publication:

> B. Gurumurthy, D. Broneske, G. C. Durand, T. Pionteck and G. Saake, "ADAMANT: A Query Executor with Plug-In Interfaces for Easy Co-processor Integration," IEEE 39th International Conference on Data Engineering (ICDE), Anaheim, CA, USA, 1153-1166(2023)

8.1 Query Executor On Co-Processors - A Primer

Turning to co-processors, there is now a broad consensus that these diverse domain-specific processors are here to stay and many new ones will be available in the market shortly. For example, the prominent co-processor - GPU - has many supporting SDKs, both hardware-sensitive as well as hardware-oblivious. We have already seen this in Chapter 6, where we list a plethora of libraries written for GPUs. We could see such a trend branching towards other co-processors as well. New generation CPUs with SIMD acceleration [41, 142] and FPGAs with query execution in line-rate, and most importantly with good SDK support for these co-processors (OpenCL, OneAPI, CUDA, Verilog, etc.), has renewed the interest in co-processor acceleration for query execution. Such a problem poses a new challenge: frequent adaptation of query executors[24] for the different SDKs (on top of the co-processors). With various combinations of co-processors and SDKs, it is often the case that engineers have

[24]We define the query executor as the component handling the execution of a query, which calls the kernels to be executed and is responsible for data provisioning on the processing devices.

to develop multiple versions of query executors. This is challenging, as with every SDK, the existing query executor has to be updated. As we briefly addressed in Chapter 2 (See Figure 2.6, a query executor can be extended with co-processors in two ways [91]: 1) using hardware-oblivious SDKs or 2) using hardware-aware SDKs. The former is portable across SDK but has poor performance; whereas, the latter offers good performance with high re-work cost. Though both these approaches are valid, as the co-processor landscape gets broader with different accelerators, the disadvantages in these approaches are ever more noticeable. Therefore, we need a query executor that is extensible while not compromising performance. hence, we propose a unified query executor – ADAMANT – that supports the free plug-and-play use of any SDK/co-processor, without reworking the execution modules (like the one in Figure 8.1).

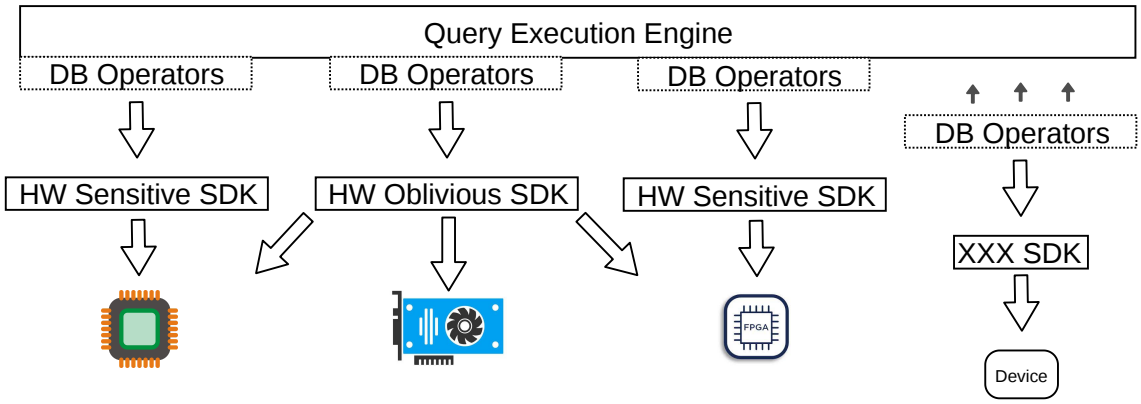


Figure 8.1: A common pluggable executor for any type of SDK.

To realize the functionality of ADAMANT, we add to the characteristics of the primitives, two additional concepts: a set of device-pluggable interfaces and a unified runtime. These parts address two key challenges. 1) **Multiple implementation alternatives:** With multiple SDKs per co-processor (e.g., a GPU has OpenCL, CUDA, oneAPI, etc.), one has to capture multiple versions of database operations. In addition, there may be multiple versions for an algorithm with a single SDK specialized for a specific workload (c.f. Chapter 6). Here, the device-pluggable interfaces define signatures for database operators so that any new implementation can be plugged into the system. Using the interfaces, we can freely couple any SDK together with its operator implementation. 2) **Handling co-processor execution:** With each co-processor, a runtime has to manage the data transfer across the device, as well as the execution itself. These functionalities are highly SDK dependent, such that updating SDK calls for one device might affect the functionalities for another. We overcome this challenge with a unified runtime that supports abstract execution models. These models handle query execution over any co-processor that is plugged in. Additionally, the models support `larger-than-memory` data sizes, i.e., processing data that does not fit completely in the co-processor memory. We implement a chunked execution model to transfer data without putting pressure on device memory.

The novelty of our approach can be considered by seeing existing related approaches that support co-processor acceleration with varying degrees of extensibility [14,

76]. Even though these systems support execution over heterogeneous processors (providing scheduling, data placement, etc.), their support for query execution is limited. Unlike these, our unified runtime expresses execution models that support query execution on arbitrary co-processors. Overall, in this paper, we architect a pluggable query engine to couple a new co-processor or API with an existing co-processor without reworking the complete query engine. Our main results and contributions are as follows.

- A query executor that supports easy operator and device plugin
- Alternative execution models for co-processor acceleration, that include operator-at-a-time, chunked execution, pipelined execution, and 4-phase pipelined execution.
- An experimental study with two heterogeneous processors (CPU, GPU) and three different API implementations (OpenCL, OpenMP, and CUDA) shows the versatility as well as shortcomings of our query executor.

The remainder of this chapter is structured as follows. First, in Section 8.2, we provide further context to our study by reviewing related work. Next, we present the necessary background for co-processor accelerated query processing (Section 8.3). In Section 8.4, we present the different tiers in our query execution engine and list available interfaces that enable the pluggability of co-processors. Next, in Section 8.5, we explain the different execution models incorporated in the runtime. Section 8.6 covers the details of our experimental study of ADAMANT, with various heterogeneous operator implementations. Finally, we conclude this chapter in Section 8.7.

8.2 Related Work

In this section, we compare our approach with other existing techniques for co-processor acceleration.

Query engines over co-processors - Based on the underlying co-processor capability, many DBMS systems have been developed [37, 67]. Most prominently, various query engines are available for CPU-GPU coupled systems: Systems like GDB [87], Ocelot [91], CoGaDB [35], OmniDB [192], Saber [111], works in SQLite by Bakkum et al. [15] and many more (BlazingSQL, PG-Storm, etc.) [51] are examples of DBMS engines over GPU. Similar works on DBMS over FPGAs include work by Ziener et al. [194], DoppioDB [166], dbX [164], AxelDB [159]. Additionally, query engines also exist over emerging co-processors like TPUs [93], and tensor cores [96]. These systems focus on optimal execution over the underlying co-processor, and they need extra support in terms of portability. Our system supports such extensions with its plugins. Components from all these systems, mainly operator implementations and device drivers, can be freely interchanged in our ADAMANT system and make use of the scalable execution models. Further, one can also combine operator implementation across these systems together, as well as link them to an optimal device driver for improved query execution.

Cross-device execution models - Recent works like HAPE [50], HetExchange [49], Fluidic co-processing [83], work by Lutz et al. [121] explore an optimal query execution model of running GPU in tandem with CPU. Unlike these, our work focuses on execution models for processing over only co-processors.

Other than DBMS engines, there are also abstract runtimes that support general-purpose processing over a co-processor. Popular systems like StarPU [14], fluidic kernels [135], elastic computing [186], Kokkos [61], DAGuE [33], Parsec [32] support cross-device execution. However, these systems are not aware of the DBMS workload. Ours complement these with primitives and execution models to be DBMS conscious. Still, our ADAMANT can benefit from concepts in these systems to have an improved execution over a co-processor.

Task model and query compilers - Finally, As mentioned previously, we extend the definitions of our primitives based on existing works. Some such works we closely relate to are Hawk [39], work of He et al. , Voodoo [140], HorseQC [73]. These systems can be integrated into our ADAMANT system given the proper primitive signatures are maintained.

Other than these, many works have a layered architecture for extensible query processing environments such as He..roDB [129] which also supports co-processor acceleration, Apache calcite [21], that extends pluggable interface for any data sources. Overall, ADAMANT varies from the existing works in terms of enabling a co-processor pluggable query engine with a pre-existing abstract query execution model.

8.3 Diversity in Programming Abstractions

Before we dive into our execution engine implementation, we quickly summarize co-processor acceleration and programming abstractions once again. We use this knowledge to develop our suitable query engine in later sections. Today, co-processors are being deployed across various domains (e.g., GPUs are used for gaming, data mining, deep learning, etc.) [134, 114]. As a consequence, there is a steady rise of SDK alternatives, as well as libraries, for co-processors [40]. In this section, we briefly explore these programming abstractions for co-processors in the context of query execution.

Co-processor SDKs give access to specialized hardware components. For example, Intel's SSE instruction set[25] gives access to SIMD features of CPUs. Based on SDKs, we identify three access levels while integrating a query engine with a co-processor. At the lowest level, vendor-specified SDKs give access to almost all hardware components of the co-processor. They offer the best performance, at the cost of poor code portability [102]. Next, some wrappers cover SDKs, abstracting device-specific details (e.g., OpenCL). They have standard functional abstractions for all supported hardware. Finally, co-processors have libraries with pre-written functions. These are written by device experts using one of the low-level SDKs, abstracting all key implementation details from an end user (e.g., OpenBLAS, cuDNN) [174]. After providing a context for SDKs and co-processors, in the upcoming sections, we explain our query engine architecture and its components for query execution in co-processors.

[25] https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

8.4 A Query Executor to Plug-in Co-Processors

Since the execution routines of a co-processor are normally handled by a host CPU (briefly detailed in Chapter 2), our architecture depicted in Figure 8.2 has a unified runtime running on a host CPU. The runtime interacts with the plugged co-processor using predefined device interfaces. These interfaces act as functional boundaries, separating the query engine from co-processor SDKs. Finally, we introduce an intermediate task layer to handle alternative implementations of a database operator across these SDKs. Overall, our architecture is split into three loosely coupled layers. At their core, these layers are responsible for:

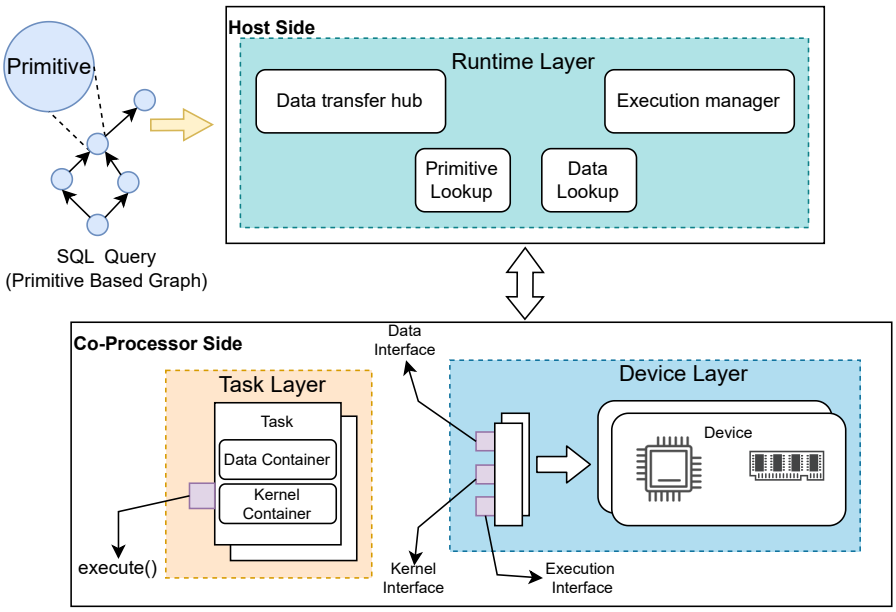


Figure 8.2: Architecture with a unified runtime and interfaces (purple blocks) to interact with plugged components.

- **Device Layer** represents the implementation of the driver on the target.
- **Task Layer** links runtime handlers to database operators on the underlying device driver.
- **Runtime Layer** is at the host, and handles the execution across multiple devices.

As shown in Figure 8.2, our runtime takes a query plan (generated from any existing optimizer) translated into a primitive graph with annotations, which mark the target device. Using these annotations, the custom execution models at the runtime layer (see Section 8.5) process the primitives using the interfaces in the device and task layers, respectively. Thus, our executor is split into a host-dependent runtime layer interacting with flexible co-processor-dependent device and task layers. Below, we first explain these interfaces in detail and show how to construct arbitrary execution models for co-processor acceleration.

8.4.1 Device Layer

Since a device can support multiple SDKs, the overall performance from the device varies based on the SDK being used [123]. For example, profiling the bandwidth range of OpenCL and CUDA in Figure 8.4 shows variations in transfer bandwidths. Generally, results show a lower bandwidth range for OpenCL compared to CUDA. This difference arises from OpenCL's translation overhead. Such a minor yet significant difference affects the overall query execution considerably. Similar performance differences can also be observed in other functionalities of the wrappers (e.g., during kernel launch, memory allocation etc.). Therefore, it is expected that if a newer and more efficient SDK is available, these functions will have to be rewritten. Therefore, for such re-work, we propose a device layer, which we use to pack SDK functions into two groups: 1) kernel management and 2) data management. We split the kernel functions separately and make them optional, as not all the SDKs support runtime compilation of kernels. On the contrary, the interface functions for data management are mandatory to be able to plug in a co-processor, as data management needs to be handled explicitly at runtime. These data management tasks include allocation/freeing memory space in the device, and transferring data into the allocated space.

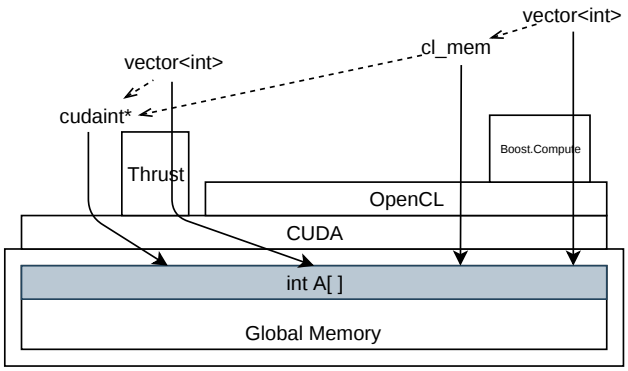


Figure 8.3: Data types across SDKs. Data type (Solid) used by a developer & in SDKs (dotted)

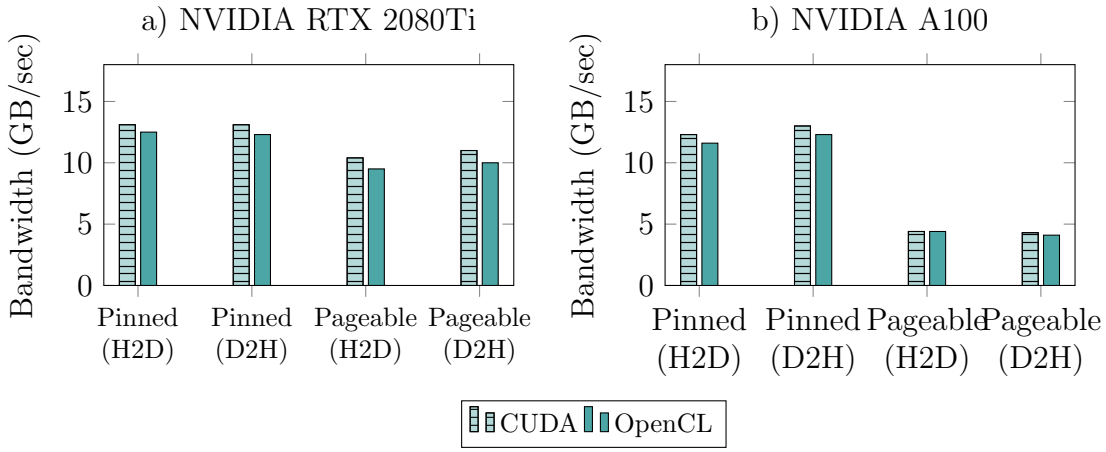


Figure 8.4: Data transfer bandwidths using CUDA and OpenCL across GPUs. H2D: Host to device, D2H: Device to host

Furthermore, we dedicate an interface to explicitly transform data from one SDK's data type to another. To understand data transformation complexity, consider a GPU using libraries Thrust & Boost.compute and SDKs CUDA and OpenCL. Each interprets a GPU's memory space in its data type, as shown in Figure 8.3.

Here the host is unaware of the relation between SDKs, so in a naive case the data is transferred into the host, transformed into the target format, and transferred back into the device. Such unwanted transfers to and from a co-processor can be avoided with a transform function, which internally transforms the memory objects from one representation to another without actual data movement. Based on the criteria, we have defined ten interfaces for the device layer:

- **place_data**(data, size, offset) - push data to the device.
- **retrieve_data**(id, size, offset) - receive data from the device.
- **prepare_memory**(size) - allocate memory in the device.
- **transform_memory**(source, target) - convert data type from source to target.
- **delete_memory** (id) - de-allocates memory in the device.
- **prepare_kernel**(name, location) - compile functions in the device.
- **initialize**() - set relevant properties for the co-processor.
- **create_chunk**(ID, chunk size, offset) - access a subset of data in the device.
- **add_pinned_memory**(ID, chunk size, offset) - reserve host-accessible memory.
- **execute**() - execute any task tagged to the device.

With any new SDK developed for an existing or new co-processor, one can realize these interfaces to couple them with our runtime. Next, let us see a sample integration of an OpenCL-programmed GPU into the ADAMANT system.

8.4.1.1 Case Study - Integrating a GPU

To showcase our ADAMANT's ease of use, we take the OpenCL GPU wrapper as a case-study and show how it can be integrated into our ADAMANT system. A similar integration is also possible with CUDA and other GPU wrappers, as well as for other systems. For this case study, we use the example code shown in Section 2.2.2 integrating them into the device layer interfaces.

To place a column into device memory, we use the `place_data()` interface. It takes an array of values as input, along with its size and optionally the starting index of the data. The corresponding OpenCL wrapper code looks like the one given in Listing 8.1, where we show the called functions for buffer creation and memory transfer.

```
int OpenCLDevice::place_data(unsigned int* data, size_t size,size_t
    ↪ start_idx) {
...
 _m_data_buffer = clCreateBuffer(_m_context, NULL, (size) * sizeof(T),
    ↪ NULL,&_m_err);
 _m_err = clEnqueueWriteBuffer(_m_device_queue, _m_data_buffer, CL_TRUE
    ↪ , 0, size * sizeof(T),data, 0, NULL, NULL);
...
}
```

Listing 8.1: OpenCL code for transferring data to a GPU

Additionally, one can include support for unified memory if possible. In case of GPU supporting unified memory, it is added in the `add_pinned_memory()` as shown in Listing 8.2.

```
int OpenCLDevice::add_pinned_memory(short alias, size_t size,size_t
    ↪ start_idx) {
...
 _m_data_buffer = clCreateBuffer(_m_context,CL_MEM_ALLOC_HOST_PTR, (
    ↪ _size) * _m * sizeof(T), NULL, &_m_err);
...
}
```

Listing 8.2: OpenCL code to allocate space in unified memory

We explicitly define these pinned memory functions to take advantage of fast data transfer. We use this memory space to transfer chunks onto the device, while utilizing the dedicated device memory to store intermediate results. More details on using the pinned memory is given in Section 8.5. Finally, we clear these allocated memory spaces as shown in Listing 8.3.

```
int OpenCLDevice::delete_data(short alias) {
 cl_int err = clReleaseMemObject(_m_data_buffer);
}
```

Listing 8.3: OpenCL code to delete space

Now that the data management functions are present, we focus on integrating the kernel compiler and its corresponding execution functions. First, we compile a primitive kernel as shown in Listing 8.4.

```
int OpenCLDevice::prepare_kernel(short alias, string _kernelSrc) {
 cl_program _m_program = clCreateProgramWithSource(_context, 1, &
    ↪ _kernelSrc, NULL, NULL);
cl_int _m_err = clBuildProgram(_m_program, 1, &_device, _cmdAgrs.c_str
    ↪ (), NULL, NULL);
_m_kernel = clCreateKernel(_m_program, _kernelName, &_m_err);
}
```

Listing 8.4: OpenCL code to compile a kernel

Our system compiles all the pre-existing kernels during initialization. These compiled binaries are tagged to their corresponding tasks (detailed in the next section), therefore, `task->execute()` performs the current task. This `execute()` in OpenCL can be implemented as in Listing 8.5.

```
int OpenCLDevice::execute() {
for (int i = 0; i < _m_args_size; i++)
_m_err |= clSetKernelArg((*_m_iter).second, i, sizeof(cl_mem), &
    ↪ _m_argument_buffer);
for (int i = 0; i < _m_param_size; i++)
_m_err |= clSetKernelArg((*_m_iter).second, i + _m_args_size, sizeof(
    ↪ int), &_param[i]);
_m_err = clEnqueueNDRangeKernel(_m_device_queue, _kernel, wd, NULL, &
    ↪ _globalSize, &_localSize, 0, NULL, NULL);
}
```

Listing 8.5: OpenCL code for kernel execution

8.4.1.2 Integration of Other Co-Processors

Other than GPUs, we can also integrate FPGAs and other co-processors into our system. For the case of FPGA, we can consider generating binary files from input for transferring into the device as `place_data()` and reading back from binary to be `retrieve_data()`. Since FPGAs are commonly used to directly execute operations as soon as the data arrives into the device, the DMA function for data transfer will act as `execute()`. However, this `execute()` must be capable of targeting the right task. This depends on the implementation of these tasks. In case of sophisticated mechanisms such as creating a runtime configurable overlay [20], the device driver must be capable of handling the execution.

Limitations: One of the caveat of our system is the integration of near data processors such as smart NICs that sits between a host and a co-processor / data store. Still, one can support such smart NICs extending the `execute()` interface. Here, the custom driver for `execute()` must be capable of differentiating between the NIC and the target executing the operators individually in them. Here, the smart NIC and the co-processor are represented as a single co-processing entity to our ADAMANT system. Thus, our ADAMANT can be pluggable with any new device wrapper as well as a co-processor without any changes to other execution system components.

8.4.2 Task Layer

The task layer encapsulates multiple implementations of a database primitive. Once again, their performance varies according to the implementation. For example, a straightforward map and reduce will vary in their performance based on the SDK in which they are implemented. OpenCL (represented with a circular mark in the graph in Figure 8.5) and device-aware implementations (CUDA, OpenMP) show mostly the same performance. However, more complex operations will have clear variations in their performance (see Section 8.6). Apart from the results in the experiments, the implementation approaches can be: 1) hand-written, 2) library, or 3) generated on the fly or in other words compiled during runtime. Therefore, this layer collects implementations (task model) and enforces functional signatures of database operations (primitive definitions).

8.4.2.1 Task Model

As discussed above, a task reduces the complexity of including a new implementation variant, without making changes to the device interface. Now, to handle the execution of a task or even a series of tasks, we propose the use of two containers.
1) **Kernel Container**: This is a simple adapter with additional runtime information required for executing a custom-written function. In the case of runtime compilation, the kernel string or generator is present in the container.
2) **Data Container**: Manages data formats for a task. Internally, a lookup table is used for data transformations. Using these, our runtime is capable of handling data transfer and execution on different devices. With this generic task model, we can introduce database-specific operations defining the signatures of individual database primitives.

Primitive definition	Description
MAP(NUMERIC in[n],NUMERIC out[n])	Does one-to-one mapping operation e.g. arithmetic operation.
AGG_BLOCK(NUMERIC in[n],(NUMERIC out)†	Does reduce operation on input (in) into result space - out.
HASH_AGG(NUMERIC in1[n],NUMERIC in2[n],HASH_TABLE hashTable[m])†	Does group-by aggregation of in2 based on groups in in1. In case of *COUNT* in2[n] is not required.
HASH_BUILD(NUMERIC in[n],HASH_TABLE hashTable[m])†	Populates the hashTable with the input - in.
HASH_PROBE(NUMERIC in[n],HASH_TABLE hashTable[m],JOINLEFT left[n],JOINRIGHT right[n])	Returns joins pairs in left and right respectively based on input in probing over hashTable.
SORT_AGG(NUMERIC in[n],PREFIX_SUM pxsum[n],NUMERIC aggregates[m])†	Does group-by aggregation over sorted data, with positions pointed by computing their prefix-sum.
FILTER_BITMAP(NUMERIC in[n],BITMAP bitmap[k],NUMERIC parameter)	Filters input in based on the parameter mentioned and stores the results in form of bitmap. Here, k = n/b - where b is the size of the bits packed per unit data.
FILTER_POSITION(NUMERIC in[n],POSITION position[k],NUMERIC parameter)	Filters similarly like FILTER BITMAP, but returns the position of selected input. The size of the result is estimated.
PREFIX_SUM(NUMERIC in[n],PREFIX_SUM pxsum[n])†	Computes prefix sum for a sequence of sorted input or input with series of 1s and 0s.
MATERIALIZE(NUMERIC in[n],BITMAP bitmap[k],NUMERIC output[m])	Returns the column values in input based on the bitmap.
MATERIALIZE_POSITION(NUMERIC in[n],POSITION position[k],NUMERIC output[m])	Returns the column values in input based on the position list.

Table 8.1: Primitive definitions for encapsulating multiple database operator implementations

8.4.2.2 Primitive Definitions

As we have detailed in Chapter 5, there exist multiple levels of primitives in RDBMS. Most of these primitives were found by surveying related work [140, 87, 39]. Additionally, we also define the I/O signatures for these primitives. Therefore, any custom implementation of primitives can be included in the system given that they adhere to the I/O semantics. This also helps include varying implementations of a primitive in our ADAMANT system. Further, with our I/O semantics, we can freely combine implementations of primitives from different wrappers together: like an OpenCL implementation of arithmetic followed by a reduce implementation using CUDA for a single device.

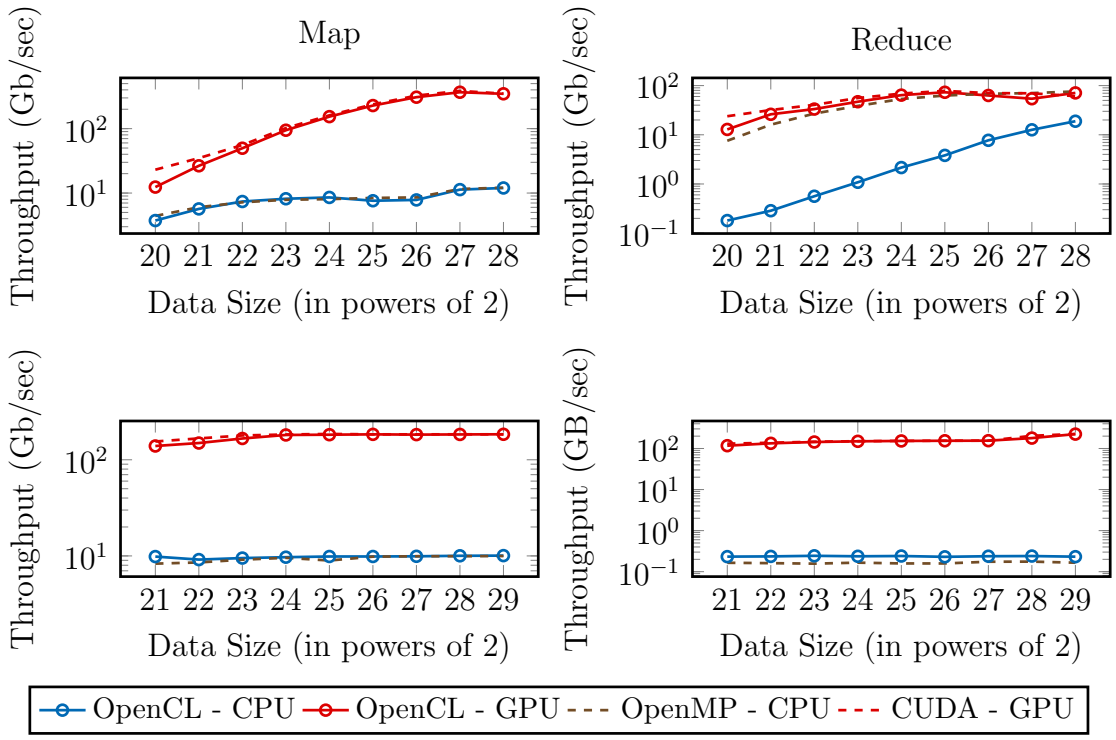


Figure 8.5: Performance of map and reduce depends on the underlying implementation, as well as the device. (The results are measured on top: NVIDIA RTX 2080Ti and Intel core i7-8700 & bottom: NVIDIA A100 and Intel Xeon Gold 5220R).

Query Pipelines: Other than the awareness of primitive signatures, our system is also aware of the characteristics of these primitives. Specifically, ADAMANT is aware of pipeline breakers (denoted with †) and materializes their intermediate results into the device memory. These pipeline breakers mark the end of a query pipeline. Thus, given a query with several pipeline breakers (for example, Q3 of TPCH), our system splits into an equal number of pipelines. These pipelines are considered an execution group and all primitives are executed together (more details on execution are given in Section 8.5). Since a query is processed pipeline-wise, our framework can also work with compiled operators as they are also forced to generate code until a pipeline-breaker before the next operator in the query can be executed.

8.4.2.3 I/O Definitions

Other than the primitive functional definitions, we explicitly define the I/O definitions to call an appropriate primitive further down the execution pipeline. For example, a selection primitive might return bitmaps or even a position list instead of column values to reduce the transfer load. However, if the materialization primitive is not aware of the incoming data scheme, it might generate wrong results, (or might even run into system exceptions). Therefore, we encode some of the common I/O semantics of the above-mentioned primitives in the data edges. Hence, when a selection produces its results as a bitmap, the corresponding materialize can be executed. Moreover, the result of a primitive might be forwarded to different primitives in the plan. For example, in a hash join, the results of the left and right tables will be materialized separately. Based on these scenarios, we define the following I/O semantics:

- **NUMERIC** - Any numeric or column values.
- **BITMAP** - A bit-packed result. These are the results of a FILTER primitive.
- **POSITION** - A position list. These are the results of a FILTER primitive.
- **PREFIX_SUM** - results of PREFIX_SUM primitive. Useful with SORT_AGG.
- **HASH_TABLE** - result of HASH_BUILD or HASH_AGG.
- **GENERIC** - Any custom data semantic (e.g. a specialized tree structure for filtering).

Using these three components, one can form a query execution plan using primitives. This enables a developer to include any custom implementation into the runtime and subsequently into the query execution.

8.4.3 Runtime Layer

The runtime layer interprets a query execution plan, executing it in the target devices. Below are the components present in the layer.

Primitive Graph: It models a query execution plan with primitives (Section 8.4.2.2) as nodes, and the data flow across primitives as edges. The graph encodes additional data information at the edges. Below are some details on the information encoded:

- **data ID** - unique ID for the data path.
- **device ID** - data location across devices.
- **processed until** - index until which the data have been processed so far.
- **fetched until** - index until which the input is transferred into device memory.

Using the data ID and device ID, we infer the type of transfer necessary for the target device. Pointers defined as processed & fetched-until allow for parallelism in query execution.

Data Transfer Hub: The data transfer hub has three main tasks:

- **load_data()**: loads data to the target device before execution. This includes either loading the complete data into the device or incurring overhead from partial loads. Internally, this function calls `place_data()` to load the input.
- **router()**: Handles all SDK-to-SDK and device-to-device data transfers. This function iterates over all the incoming edges to a primitive and loads the data to the target device. Internally, the function calls interfaces : `place_data()`, `retrieve_data()`,`transform_memory()`.
- **prepare_output_buffer()**: It estimates and creates a result space for a given primitive. It also handles data semantics based on the primitive.

In summary, the runtime couples operator implementation with device interfaces. Thus, the three layers enable query execution of any of the plugged co-processors. Such an out-of-the-box query execution is possible because of multiple execution models present in the runtime. In the next section, we describe these execution models for plugging devices.

8.5 Execution Model Alternatives for Co-Processors

Execution models are the key to query execution. They define the process and data flow within the system while processing a query. Hence, defining a suitable execution model for co-processor acceleration in turn characterizes the execution flow of our runtime. In this section, we explore the scalability limitation in the operator-at-a-time execution model with examples. To overcome the limitation, we propose an abstract chunk-based execution model capable of supporting any co-processor. Additionally, we modify this chunked execution model to be hardware-aware using GPU as a case study.

8.5.1 Limitations in Operator-At-A-Time Execution in Co-Processors

Operator-at-a-time (OAAT) is one of the common execution models for query execution in co-processors, where complete database tables are stored within device memory to avoid costly transfers. However, this is not a scalable solution as co-processors usually have smaller on-chip memory compared to the main-memory sizes. To illustrate this, we plot the data size of the input for different queries in the TPCH benchmark and the complete TPCH dataset against the memory capacity on various GPUs (cf. Figure 2.10). From the results, we can observe that only some of the TPCH queries can be executed on a device with input data completely in the device memory. As a normal OLAP query requires only a few columns from the complete dataset, storing the complete dataset reduces the space to store intermediate results. For example, the query plan for TPCH query 6 (given in Figure 2.10) has the memory footprint as shown in Figure 8.7 during execution. Thus, storing the complete dataset in co-processor memory reduces the space available for the intermediate results of a query. To reduce such memory pressure on the co-processor, we need an alternative execution model.

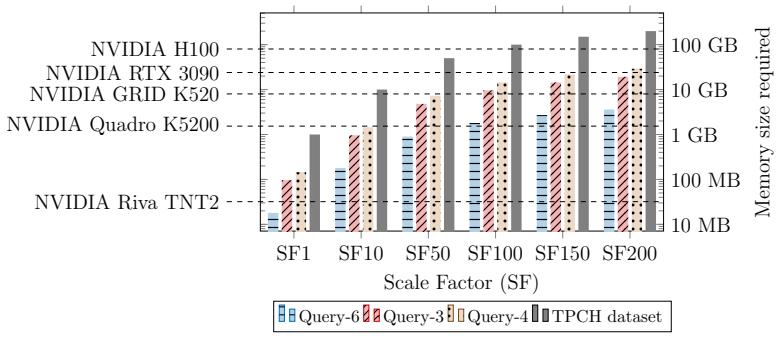


Figure 8.6: Memory capacity in GPU devices vs memory required for processing TPCH data

As an alternative to the operator at a time, a scalable chunked execution model is already available [73]. We construct a similar chunked execution model using the interfaces discussed in the previous sections. Using this execution model, our runtime can scale query execution over any arbitrary co-processor.

8.5.2 Chunked Execution for Arbitrary Co-Processors

Even with chunked execution, a long query execution plan might generate multiple intermediate results utilizing the complete memory space of a co-processor. Therefore,

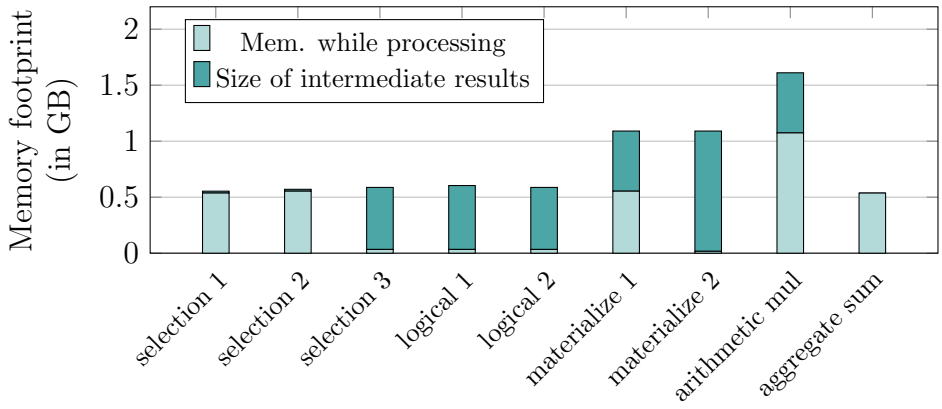


Figure 8.7: Memory footprint of individual primitives in TPCH-Query 6

our execution plan executes a query pipeline-wise to reduce both the memory load, as well as processing load in the device.

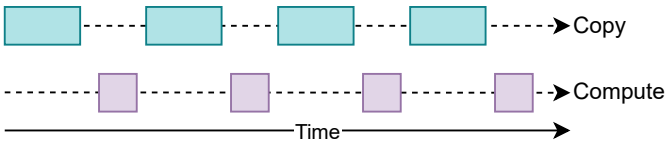


Figure 8.8: Process flow in chunked execution model

Algorithm 3: Chunked execution

```
1 foreach chunk C of input do
2     foreach Primitive P in pipeline (QEP) do
3         Edge ie = incoming_edges(dag[P]);
4         router(ie.source_device_ID, ie.target_device_ID,c,chunk_size);
5         available_device[target_device_ID]→ prepare_memory(output_size);
6         available_device[target_device_ID]→ execute();
```

The chunked execution constructed using our interfaces is given in Algorithm 3. The execution starts with a chunk of input transferred to the co-processor. This chunk is processed through a complete pipeline and the intermediate result of the final pipeline operator is persisted, while others are overwritten by the results of processing the next chunk. The overall memory consumed for intermediate results depends on the chunk size; therefore, only a fraction of the memory is utilized. Since a chunk has to be processed until the end of a pipeline, the next chunk is transferred only the current chunk is processed. Here, the transfer waits for the execution to compete before transferring the next chunk. Even though the execution model works with arbitrary data sizes, its performance might not be optimal due to constant data transfers. Such transfer delays are hardware-dependent and, therefore, can be improved only using a hardware-aware approach.

Since improving chunked execution is hardware-centric, we take a current generation GPU as a case study and utilize its components in improving our execution model.

8.5.3 Case Study: Pipelined Execution in GPUs for Concurrent Execution with Data Transfer

Since data transfer is a bottleneck, we hide the transfer time with concurrent execution (cf. Figure 8.9). However, the transfer delay is so high that hiding it with a single primitive execution will not be beneficial. Hence, we hide the transfer of a data chunk with the execution of a complete pipeline. We incorporate this `copy-compute` routine into our runtime using separate threads for data transfer and pipeline execution on a co-processor.

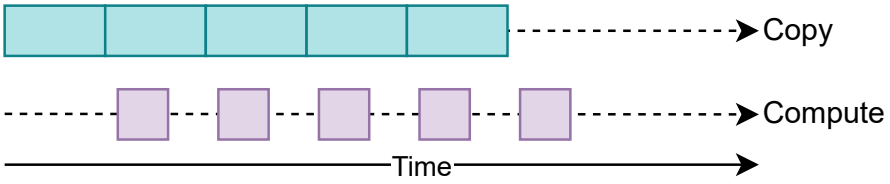


Figure 8.9: Process flow in pipelined execution model

Algorithm 4: Pipelined execution

```
1  foreach Primitive P in pipeline(QEP) do
2      thread transfer = spawn_thread(transfer_data() );
3      foreach i=0 until input/chunk do
4          Edge ie = incoming_edges(dag[P]);
5          wait_until(ie.fetched_until ≤ ie.processed_until); available_device[target_device_ID]→
             execute();
6          ie.processed_until+=chunk_size
7  transfer_data(){
8  foreach Chunk C of input do
9      foreach Edge ie in incoming_edges(dag[P]) do
10         router(ie.source_device_ID, ie.target_device_ID,c,chunk_size);
11         available_device[target_device_ID]→ prepare_memory(output_size);
12         ie.fetched_until+=chunk_size;
13     }
```

We track the processed chunks to effectively synchronize the threads. The execution thread keeps track of the amount of data processed using a counter `processed_until`. A similar counter - `fetched_until` - is used to track data transferred so far. If the fetched_until value is smaller than processed_until, the execution thread waits for the data to be transferred by the transfer thread, and vice versa. The threads also synchronize at the end of each pipeline breaker and start with the next pipeline.

Even though the execution model hides execution with transfer, the overall transfer overhead is still the same as that of naive chunked execution. We improve this further using a four-phase approach, described in the next section.

4-Phase Pipelined Execution With Memory Reuse

Since transfer is a severe bottleneck, improving it should in turn increase performance. We optimize the transfer delay in this execution model, as shown in Figure 8.10, with four different phases.

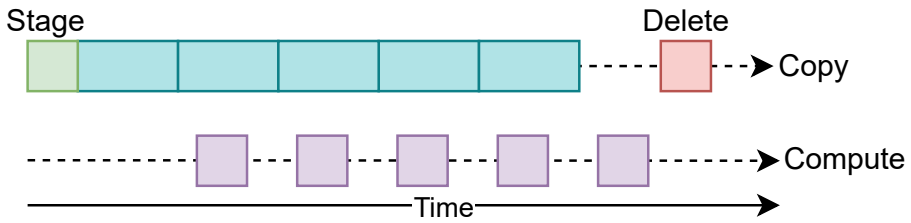


Figure 8.10: Process flow in 4-phase pipelined execution model

The detailed execution of the four-phase query execution is given in Algorithm 5. As the memory transfers for a query are predefined, we use *pinned memory* (cf. Figure 8.4) for faster transfer. As pinned memory is accessible to both the host and the co-processor, the host directly transfers data here while the co-processor executes its primitives. One problem arises here with copy-compute, that the copy phase might overwrite the data currently being executed. However, we ensure the data is not overwritten using the `processed-until` index.

Algorithm 5: 4-phase pipelined execution

```
1  //Stage Phase
2  foreach Primitive P in pipeline(QEP) do
3      foreach edge E in P do
4          if Source(E) is Data_Scan then
5              alias1 = available_device[target_device_ID] →add_pinned_memory(memory_size);
6              data_dictionary.insert(E,alias1); alias2=available_device[target_device_ID]
                 →add_pinned_memory(memory_size);
7              data_dictionary.insert(E,alias2);
8          else
9              alias=available_device[target_device_ID] →prepare_memory(memory_size);
                 data_dictionary.insert(alias,E);
10 //Copy-compute phase
11 foreach Primitive P in pipeline(QEP) do
12     thread transfer = spawn_thread(transfer_data() );
13     foreach i=0 until input/chunk do
14         Edge ie = incoming_edges(dag[P]);
15         wait_until(ie.fetched_until ≤ ie.processed_until); available_device[target_device_ID]→
             execute();
16         ie.processed_until+=chunk_size
17 //Delete phase
18 foreach Primitive P in pipeline(QEP) do
19     foreach entry in data_dictionary do
20         available_device[target_device_ID] →delete_data(entry.alias);
```

To avoid such scenarios, we create two identical memory spaces to alternate execution and transfer. The transfer and execution threads alternate between these memories that access the chunks, as shown in Figure 8.11. Additionally, the intermediate results of any pipeline breaker are also transferred back to the host using pinned memory. All other intermediate results are stored in the device memory itself. Once the execution is complete, we deallocate these memory locations. In summary, execution starts by creating pinned memory spaces - stage phase - over which the

chunks are copied - copy phase. Once a chunk is copied, the compute phase processes these data. After execution, the deletion phase clears memory for the next query.

Replacing `add_pinned_memory()` with `add_data()`, we make the above execution models support any arbitrary co-processor. However, the performance still depends on the implementations of the device driver. To this end, we have developed custom drivers for GPU and CPU using CUDA and OpenMP, respectively, in addition to a common implementation in OpenCL. We use these drivers to evaluate the performance of our ADAMANT framework, and the results are discussed in the next section.

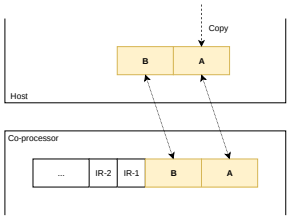


Figure 8.11: Dual memory spaces for concurrent transfer-execution

8.6 Experiments

We evaluate in this section the performance of our custom primitive implementation, the overhead from our abstraction layers, and the performance of using the above-mentioned execution models, the latter with special consideration for larger-than-memory processing. For our evaluations, we use the device drivers: OpenCL (for CPU), OpenMP, OpenCL (for GPU), and CUDA, running on top of two different environments. Details about the environments are given in Table 8.2. The primitives over the evaluated drivers follow semantically similar implementations. All these implementations are written in C++[26]. For the baseline, we consider HeavyDB (formerly MapD) for its compiled execution model and compare it with our proposed execution models.

	Setup 1	**Setup 2**
CPU	Intel(R) Core(TM) i7-8700	Intel Xeon Gold 5220R
GPU	GeForce RTX 2080 Ti	Nvidia A100
GCC	9.3.0	8.4.0
OpenCL	2.1	2.1
CUDA	11.0	10.1
OS	Ubuntu 18.04	Ubuntu 20.04

Table 8.2: Device setup used in evaluating ADAMANT engine

8.6.1 Profiling Primitives

We profile the throughput of our primitives using 2^{28} integers values (1GB) in random distribution. Our filter operator can perform early and late materialization with bitmaps as an intermediate data type. We measure the performance of both

[26]code available at: https://git.iti.cs.ovgu.de/dead-ops/ADAMANT/-/tree/master

approaches. Similarly, as the hash join has both the hash-build phase and the hash-probe phase, we measure them individually. We use linear probing as the underlying hashing technique. The hash table is placed in a global memory and all the threads compete to place their data in a bucket, which is resolved using atomic operations. The performance profile of these primitives is shown in Figure 8.12 for two hardware configurations.

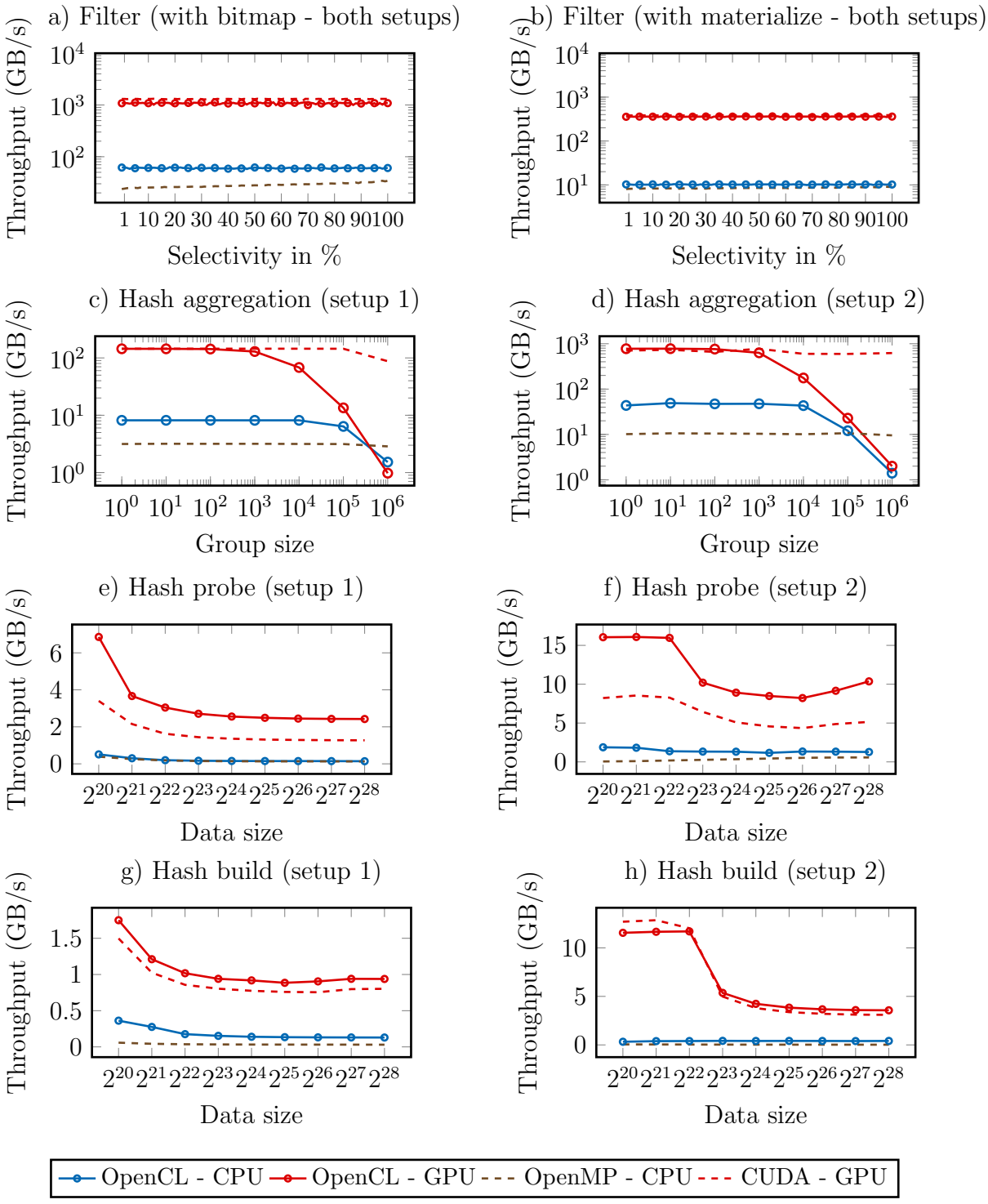


Figure 8.12: Profile of primitives in OpenCL, OpenMP and CUDA

Filter (bitmap): The results in Figure 8.12(a) are nearly constant for the different devices, as we perform a bitwise comparison of the values in an array. Since each comparison of input takes roughly the same amount of time irrespective of their selection, the performance graph is similar to that of a map in these devices (cf. Figure 8.5). However, on both devices, OpenCL performs better than OpenMP on the CPU but is equivalent to CUDA on the GPU. In our case, the OpenMP variant suffers from data movement overheads, as the hardware threads are explicitly scheduled, whereas OpenCL handles them internally. Perhaps further evaluation with varying thread sizes could improve this.

Filter (with materialize): Comparing Figure 8.12 (a & b), we see that adding materialization leads to a significant performance drop in a GPU - about 30% the performance from using only bitmap. This is mainly from the time taken to extract bits from a bitmap in a GPU. Since we pack results from multiple inputs into a single bitmap, GPU threads must cooperatively extract their respective bitmap input, leading to performance degradation. However, such an impact of materialization is very small for CPUs, as threads are scheduled with a sequence of 32 input values, avoiding data sharing among threads.

Hash aggregation: Our hash aggregation uses a single global hash table for aggregation. Based on the results in Figure 8.12 (c & d), we see that the performance of OpenCL decreases drastically with increasing group sizes. As the data from SIMT threads is served through a common memory controller, inserting multiple data in parallel requires more time.

We also see the profile of CUDA is not deteriorating with larger group sizes than that of OpenCL. We believe that this is due to the static scheduling of threads in OpenCL. Finally, the high performance of GPUs comes from its faster internal bandwidth from memory controllers.

Hash build: Similar to hash aggregation, hash build also has a shared hash table for insertion. We see that the hash build performance drops with larger data sizes. This is mainly due to the repeated data insert calls from threads for larger data sizes. On the other hand, the CPU performance is still the same. Again, the threads spawned in a workgroup lead to minor performance differences across OpenCL and OpenMP execution. Additionally, by comparing the performance with hash probing, we can easily identify the overhead of insertion using a single shared hash table. Here, we use atomics for insertion, which serializes the threads during insertion. The difference in performance indicates this serialization overhead of atomics.

Hash probe: Since a hash probe follows nearly the same execution path as a hash build, the reflected performance also has similar characteristics to a hash build. However, the performance from CUDA is affected by the probe, compared to OpenCL. This again might be due to the influence of the order of threads accessing the global memory.

8.6.2 Impact of Abstraction Layers

As our abstraction layers are loosely coupled, they incur overhead in query execution. To understand this overhead, we measure the difference between the overall execution

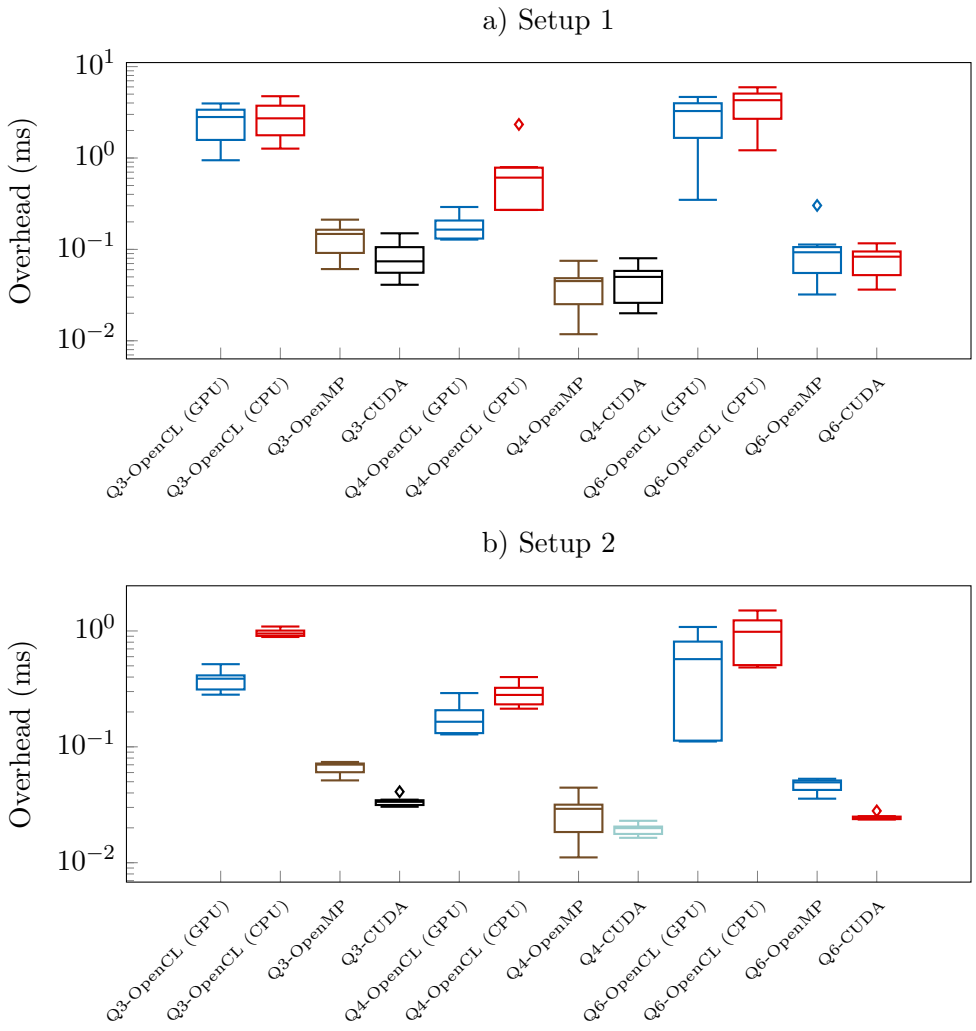


Figure 8.13: Overhead of abstraction layers

time and the total sum of processing time of the individual primitives of a query. The results in Figure 8.13 show maximum overheads for OpenCL wrappers, compared to CUDA and OpenMP. This overhead arises from explicit data mapping to a target kernel, whereas OpenMP and CUDA do not need one such data mapping explicitly. Based on the results, we see that the abstraction layers and the overhead of our execution model are quite small compared to direct execution. Furthermore, by comparing the performance across devices, we see that the hardware-sensitive implementation for the primitives plays a major role in performance. In general, our OpenCL implementation incurs significant delays due to explicit data mapping.

8.6.3 Performance of Execution Models

Our execution models focus on co-processor acceleration of larger-than-memory datasets, as described in Section 8.5. In this section, we evaluate the performance of these execution models with larger TPC-H scale factors (with total input size for queries varying from 2GB (2^{29} integer values) to 3.5GB ($2^{29.7}$ 32-bit integer values). Since multiple TPC-H queries have similar patterns, we consider queries Q3 (multiple joins), Q4 (subquery), and Q6 (heavy aggregation) for our evaluation. We consider

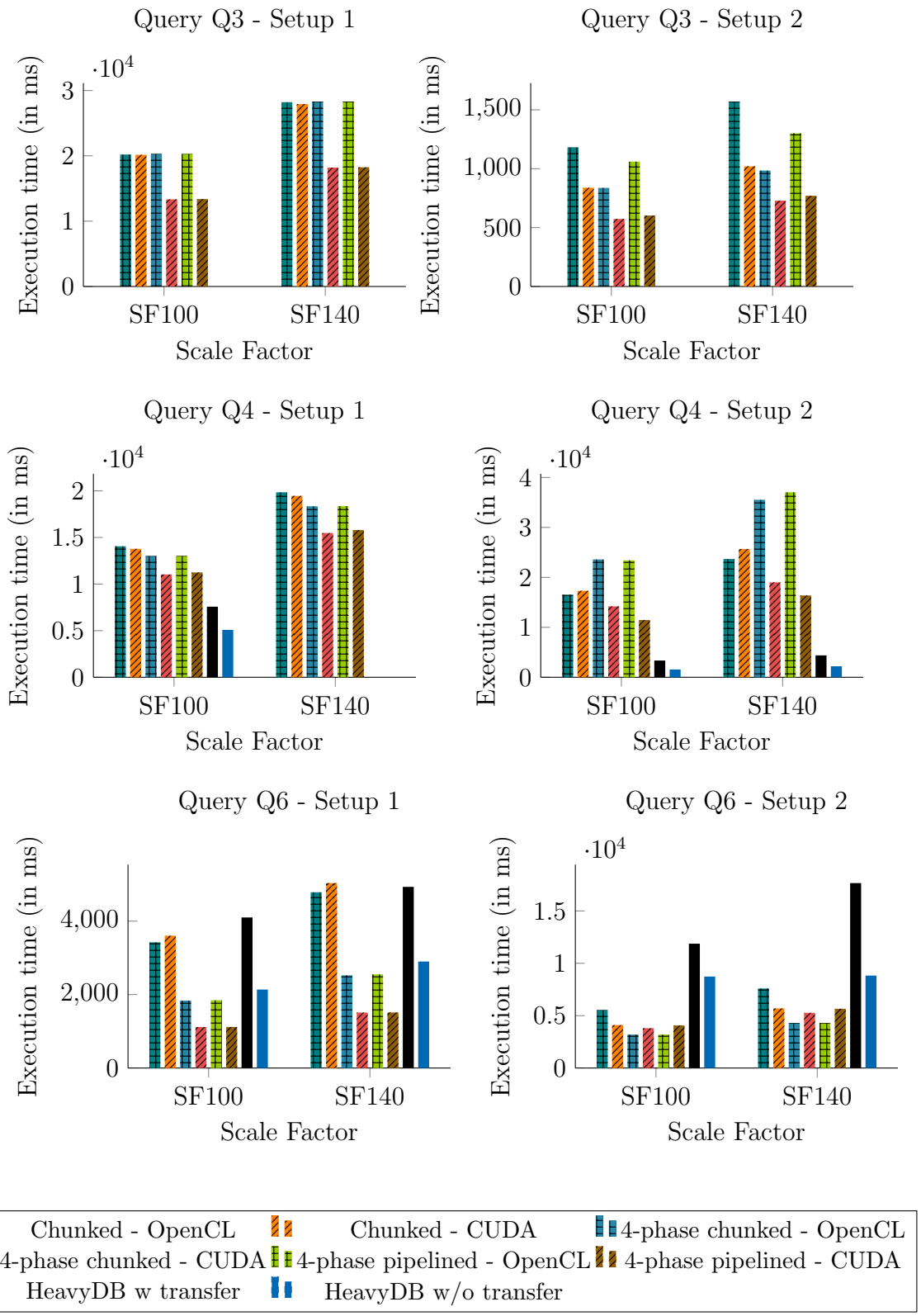


Figure 8.14: Performance of the execution models versus HeavyDB across various scale factors

the size of chunks to be 2^{25} ints across all the queries. This chunk size is found to be optimal for the underlying GPU based on the available space in the device.

Overall, the plots in Figure 8.14 show the overall execution time for the execution model, with the underlying SDK and even the query being executed. In general, results show that 4-phase execution is faster than naive chunked execution in both OpenCL and CUDA. Mixing pinned memory with normal transfer consistently leads to better performance compared to the alternatives. Additionally, the results show that 4-phase pipelining produces similar improvements when compared with four-phase chunked execution. This shows that the execution time of a query is so small that hiding it with transfer only provides minimal benefit. The rationale for the performance difference is as follows.

In particular, the query being executed on the target has the largest impact on performance. For example, Q4 has an adverse performance with 4-phase execution in OpenCL. This behavior shows that using pinned memory for multiple data transfers (which is the case with Q4) to a GPU degrades its performance. Since the query starts with building a hash table, there is no other operation between data transfer and hash build that has a considerable execution time for the transfer-time hiding. Therefore, the overall time for the pipeline is nearly the same as the data transfer time, leading to no performance benefit from attempting to hide execution time. Due to such poor execution behavior, 4-phased execution for OpenCL is nearly 2x slower than chunked execution. However, CUDA can overcome this issue and has up to 1.5x speed-up when compared to chunked execution. This shows that the overhead of handling query execution using OpenCL incurs additional effort, which degrades overall performance. For Q3 and Q6 4-phased execution is faster than chunked execution. We see that the execution is nearly 2x faster for CUDA and 1.5x faster for OpenCL. Since these queries have a pipeline comparatively deeper than Q4, the 4-phase execution is beneficial. Therefore, depending on the query and its operators, the pipelined execution is found to be beneficial. For the most part, four-phased execution has a speed-up of 3x (in the best case - Q6) until 1.3x (the worst case - Q3) over chunked execution. This performance difference is subject to change with newer GPUs. Next, we see that OpenCL performs worse in general compared to CUDA. As seen in Figure 8.12 and Figure 8.13, the difference in execution of individual primitives as well as the overhead of handling execution for OpenCL leads to a higher execution time.

Comparison with HeavyDB: We compare our execution models with HeavyDB[27] (formerly MapD [153]) both with a cold start (HeavyDB w transfer - with data transfer) and pure execution (HeavyDB w/o transfer). We use larger scale factors SF 100,120,140 datasets for our evaluation. Since HeavyDB works with in-place tables in the GPU, Q3 cannot be executed for the given scale factors, as the hash table size exceeds the maximum capacity. For the other queries, we see that the in-place execution of HeavyDB is comparable with our chunked execution, whereas cold start is quite slower than our execution models. In the case of Q4 and Q6, our execution models show a performance improvement of up to 2x for in-place and up to 4x for cold start execution. This behavior can be associated with the delay in transferring a

[27] https://github.com/heavyai/heavydb

complete table to the device memory, whereas we only transfer chunks of the column necessary for execution. The transfer delay within HeavyDB can also be inferred from the difference in performance between cold and hot start.

In summary, using pinned memory for costly transfer and using device memory for intermediate results improves query execution. The results show such a benefit with up to 3x the performance of naive chunked execution. However, the benefits of such execution still depend on the query (and its pipelines). Furthermore, the execution of pipelining with transfer has a small impact, since the transfer time dominates the execution of the overall query. Therefore, the hiding execution time only improves a little. Finally, our results show that there is indeed a performance difference between OpenCL and CUDA that hardware-sensitive implementation is highly necessary for better performance.

Evaluation Summary: Overall, with our architecture, multiple SDKs can be plugged in, which allows us to have a common performance comparison module for database operations (Figure 8.12). Furthermore, we measure the overhead of handling these SDKs (Figure 8.13). Finally, our results from Figure 8.14 show that scaling data sizes requires an efficient execution model that utilizes both pinned memory and normal transfer for data management.

8.7 Summary

Hardware architectures are increasingly heterogeneous and many database engines are trying to utilize their capabilities for faster query execution. In this chapter, instead of developing a query engine from scratch over each of these devices or SDK choices, we propose a pluggable architecture that can plug in multiple devices and SDKs, with a low overhead. Our proposed architecture, for ADAMANT, has three layers that handle execution in a co-processor using granular database primitives. Along with our architecture, we also propose an execution model for scaling execution to larger-than-memory datasets. Based on our evaluation of CPU (OpenMP, OpenCL) and GPU (OpenCL, CUDA) prototypes, we observe that there is a marked overhead in handling OpenCL execution compared to OpenMP and CUDA. Furthermore, we also identify that our four-phased execution can be employed for significant performance improvements over chunked execution of up to 3x. Our experimental evaluation emphasizes the complex optimization space of queries in a co-processor environment, which other than traditional optimization, spans further execution models, operator placement, and primitive implementation (including micro-optimizations or SDK choices), among more parameters. We believe our ADAMANT prototype can facilitate the study of this optimization space for database systems. Overall, our ADAMANT query engine can be used to build an efficient query processing system around new hardware, with less effort.

9. Conclusion

It has been evident that processor architectures are going to be diverse in the future. Consequently, the processor ISAs will also get diverse, leading to an abundance of options to develop an optimal application. This also meant, painstakingly testing each of these options to identify the best among them, which is quite hard and time-consuming. Hence, it is necessary to come up with a system that increases productivity in terms of integrating co-processors.

Diverse changes to co-processor architecture are a ubiquitous challenge across various domains, and many have come up with various solutions to adapt to the architectural changes. On one hand, many hand-made solutions are developed for DBMSs over a particular co-processor. On the other hand, abstract runtimes are proposed to hide implementation-specific details for faster integration. Though these solutions have their benefits, there is still a need for a holistic query engine that supports easy co-processor integration, still supporting extended performance from the device. In this work, we explored such a query execution engine.

Case-Study: Sort-Based Aggregation

As a first step towards realizing our system, we performed a case study to understand the importance of hardware-aware implementation. We consider the case of optimizing group-by operator that is aware of the underlying GPU. Specifically, we implement the operator with the support for `atomic operations` - a special operator executed directly in a GPU component. We investigate how to tune sort-based grouped aggregation using atomics and see its implication in execution. Furthermore, we also designed two alternative variants using a private variable or array and investigated their performance on various GPU memory spaces. Our results show that our variants speed up grouped aggregation compared to a naive usage of atomics by a factor of 1.5 to 2, when well configured. Furthermore, a sort-based grouped aggregation using atomics can outperform a hash-based aggregation by 1.2x to 2x for most used group sizes. Based on these results, we conclude that hardware-awareness indeed improves performance. However, direct operator implementation is still time-consuming. Hence, we split the implementations into various smaller functions called primitives.

Primitives

A DBMS operator can be realized by coupling various primitives. We surveyed the existing works to come up with a comprehensive list of primitives, that can support various standard database operators. We have also shown that these primitives work at different levels of granularity, and discuss hardware-based tuning for the finest-granular level of primitives. Finally, we discuss the impact of these primitives in the design of a query engine. Even though these primitives are minimal and even reusable in multiple operations, we have the problems of realizing these primitives and handling them across devices.

Realizing Primitives

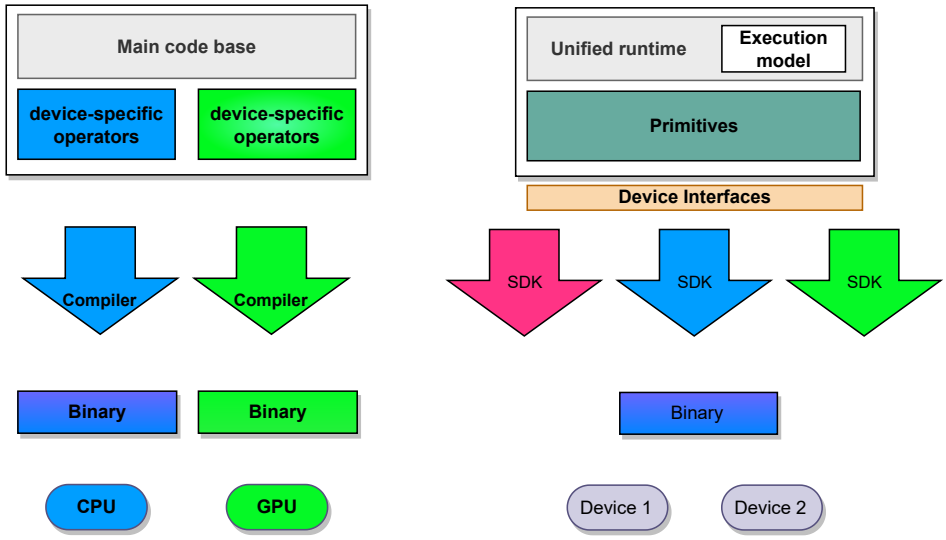
For primitive realization, we consider the different GPU libraries that support DBMS operators. To this end, we reviewed different expert-written libraries to be used for faster prototyping of a GPU-accelerated database system. Based on our review, we identified 43 GPU libraries, out of which 6 support database operators. Though we have quite a handful of libraries for GPUs, only three of those support database operators directly. Hence, we studied their performance implication in query execution and summarized their level of usability and usefulness. Now that the operator implementation is available, the next challenge addressed is to couple them with query execution.

Understanding Execution Models

Query execution relies heavily on the execution model used. Hence, we have studied the existing execution models in CPU: compiled & vectorized to develop one for our query engine. Based on our review of the execution models, we identify that compilation time has a huge impact on query performance. Hence, to overcome this issue, we integrated vectorized with compiled execution. Specifically, we hide compilation time with vectorized execution. This prototype - named Tether - first starts execution using a vectorized model concurrently with compilation. Once compilation is done, the system switches to compiled execution via pipeline breakers. These pipeline breakers are specialized primitives that enforce the intermediate results to be materialized completely before the next primitive in the execution pipeline can be processed. Thus, our system can successfully utilize the time for compilation into processing partial results, thereby improving performance. Based on our experiments, we saw that we get benefits up to 3x the baseline approaches. Though the approach is beneficial, it is still centered around CPU-based systems. Hence, to have abstract co-processor support, we developed a unified query engine runtime.

ADAMANT Architecture

Our proposed architecture - ADAMANT - has three layers that handle execution in a co-processor using granular database primitives. Along with our architecture, we also propose an execution model for scaling execution to larger-than-memory datasets. Based on our evaluation of CPU (OpenMP, OpenCL) and GPU (OpenCL, CUDA) prototypes, we observe that there is a marked overhead in handling OpenCL execution compared to OpenMP and CUDA. Furthermore, we also identify that our



(a) Hardware-aware query engine

(b) Hardware-pluggable query engine

Figure 9.1: An updated query engine for plugging in an arbitrary co-processor

four-phased execution can be employed for significant performance improvements over chunked execution of up to 3x. Our experimental evaluation emphasizes the complex optimization space of queries in a co-processor environment, which other than traditional optimization, spans further execution models, operator placement, and primitive implementation (including micro-optimizations or SDK choices), among more parameters. We hope that our ADAMANT prototype can facilitate the study of this optimization space for database systems. Overall, our ADAMANT query engine can be used to build an efficient query processing system around new hardware, with less effort.

Thus, here we explored developing a query engine for emerging co-processors. We extend the current hardware-aware query executor architecture, like in Figure 9.1 (a), to be pluggable. Such a hardware-aware architecture is time-consuming, which we have showed in our case study. Our extension to the hardware-aware architecture is to have pluggable components, so that we can support any co-processor through any one of its corresponding SDKs (as given in Figure 9.1 (b)). These extensions span over three main components: First, we showed in this chapter the need for hardware awareness and used them for developing abstract primitives - forming the *task layer*. Additionally, we also realized these abstract primitives using GPU-based libraries. Second, we explore the existing execution models, optimizing them with a hybrid execution model - forming the *runtime layer*. Finally, we develop an abstract query engine that supports the pluggability of any co-processor- using the *device layer*. Thus, we have a unified runtime that supports plugging any co-processors that increase productivity as well as support increased performance.

Future Work

Our pluggable query engine so far supports co-processor integration specifically implemented for query execution. However, optimal execution needs various optimizations to be carried out. These can be the natural next step for the work.

First, we need an optimizer that does informed placement of database operations onto target co-processors. To this end, we envision a two-level optimizer. On the global level, the optimizer decides to place operations into a target co-processor. Once placed, a local optimizer must then identify the best implementation to have improved performance. These two optimizers must be in symbiosis to attain the best performance out of the overall system.

Next, as an extension to our current execution, we can support parallelism across processors. The current execution engine supports only co-processor acceleration while the host CPU is idle. Hence, adding parallelism constructs in execution should enable concurrent execution across both the host and co-processor. The parallelism can enable functional parallel, data parallel as well and pipelined execution across the devices.

Finally, our work can be used to develop primitives for other database models (like graphs, and key-value stores). These primitives along with their supported execution models can be explored to develop a unified runtime that supports query execution across other database paradigms.

Moreover, within the scope of each of the chapters, we propose the following future work:

Sort-Based Aggregation

Since sort-based aggregation has been ubiquitous for query execution, we took it as a case for exploring hardware-sensitive implementations. The natural next step is to replicate the same optimizations for sort-merge joins.

Additionally, we focused only on atomics as a hardware-sensitive option, and we tuned it to obtain better performance. However, other parameters like loop-unrolling and predication can also be integrated into the existing code to further improve performance.

Finally, the current setup only explores atomics-based aggregation in isolation from other steps like sorting. Hence, a hybrid sort-aggregation setup could be explored – where we generate partial aggregates with sorted runs to improve performance.

Primitives

As mentioned above, the straightforward next step for primitives is to explore the ones present for operators running over other data models (like graphs). Similar primitives can also be identified for other database functionalities, especially query optimizations.

Realizing Primitives

For our future work, we can extend our approach with other libraries built on top of other low-level wrappers like OneAPI and do a comprehensive study of all libraries w.r.t. their support for database operators. Furthermore, building an optimizer that chooses the best-performing library-based operator during runtime is another important tuning task.

Hybrid Execution Model

Since merging vectorized and compiled execution benefits execution, there are various directions to improve the current performance. Tuning options like identifying optimal cross-over points, storing existing compiled plans for future re-use, and a dedicated optimizer to find the optimal compilation paths. One other key area to explore is ways to debug these two different execution models without increasing complexity.

Optimization

Ultimately, the next step in advancing our query engine is developing an appropriate optimizer. Our query engine functionalities like primitive placement, variant selection, and execution island selection require optimizations to have an efficient query execution.

Such a variety of optimizations must work together to have the best execution. For example, based on the operator placed on a particular co-processor, its corresponding optimal variant must be selected across the different implementations from different SDKs. Hence, we must progressively optimize for best execution. Our possible next steps in this scenario are to first identify execution islands - a set of query operators that act as a single execution unit. Next, we need to place this execution island onto a co-processor. Once placed, we then can identify the optimal variant for faster execution. To achieve such an optimal query execution plan, we need a special optimizer that is aware of the co-processor characteristics that are also pluggable through interfaces like the ones in our query engine.

A. Appendix

Here we provide the various code snippets and additional references to the concepts presented in this work. We first start with queries we used throughout this work, followed by the code snippets used in realizing the functions / operators used in our experiments.

A.1 Benchmark Queries

Throughout this work, we use TPC-H extensively for benchmarks and evaluation. It is a decision support benchmark developed and curated by Transaction Processing Performance Council (TPC). They have defined a database with eight tables. The records in the database can be generated with different scale factors, with a 1 scale factor (1SF) equalling 1 GB of data. These tables are used to benchmark decision support systems (DSS) via 22 benchmark queries. Out of these, we use the following in this work:

Query 1

```
SELECT
        l_returnflag,   l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
FROM
        lineitem
WHERE
        l_shipdate <= date '1998-12-01' - interval '90' day
GROUP BY
        l_returnflag,   l_linestatus
ORDER BY
        l_returnflag,   l_linestatus;
```

Query 3

```
SELECT
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        l_orderkey,     o_orderdate, o_shippriority
FROM
        customer, orders, lineitem
WHERE
        c_mktsegment = 'BUILDING'
        AND c_custkey = o_custkey
        AND l_orderkey = o_orderkey
        AND o_orderdate < date '1995-03-15'
        AND l_shipdate > date '1995-03-15'
GROUP BY
        l_orderkey,     o_orderdate, o_shippriority
ORDER BY
        revenue desc, o_orderdate
LIMIT 20;
```

Query 4

```
SELECT
        o_orderpriority, count(*) as order_count
FROM
        orders
WHERE
        o_orderdate >= date '1993-07-01'
        and o_orderdate < date '1993-10-01'
        and exists (
                SELECT * FROM lineitem
                WHERE l_orderkey = o_orderkey and l_commitdate < l_receiptdate )
GROUP BY o_orderpriority
ORDER BY o_orderpriority
```

Query 6

```
SELECT
        sum(l_extendedprice * l_discount) as revenue
FROM
        lineitem
WHERE
        l_shipdate >= date '1994-01-01'
        AND l_shipdate < date '1994-01-01' + interval '1' year
        AND l_discount between 0.06 - 0.01 AND 0.06 + 0.01
        AND l_quantity < 24;
```

Query 18

```
SELECT
        c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice,
        sum(l_quantity)
FROM
        customer, orders, lineitem
WHERE
        o_orderkey in (
        SELECT l_orderkey FROM lineitem
        GROUP BY
                l_orderkey having sum(l_quantity) > 300
        )
        and c_custkey = o_custkey and o_orderkey = l_orderkey
GROUP BY
        c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
ORDER BY
        o_totalprice desc, o_orderdate
LIMIT 100
```

We use these (and in few cases, tiny variations of these) queries to evaluate the performance of our systems. These queries cover the behaviors of other TPCH queries as well, hence the performance profile would be similar for others.

A.2 Code Snippets for Sort-Based Aggregation

In the case of sort-based aggregation, atomic operations play a key role in overall performance. Hence, we develop a simple atomic-based aggregation mechanism that can work with sorted results to clearly see the impact of these functions over execution. To make it a fair comparison, we also developed a fairly straightforward hashing mechanism (linear probing in this case) that uses atomics for aggregation. Here are the code snippets for them:

Simple atomic aggretation over sorted values on global memory

```
__kernel void atomic_aggregate(  __global unsigned int* ps,
__global unsigned int* res){

        atomic_add(&res[ps[get_global_id(0)]],1);
}
```

The Listing A.2 shows a straightforward atomic based aggregation mechanism. In this case, we execute count aggregation, however it is easy to update with other aggregation operations like sum, max and min.

Simple atomic aggretation over sorted values on local memory

```
__kernel void atomic_aggregate_local(  __global unsigned int* ps,
__global unsigned int* res,
__local unsigned int* localRes){

/*
* Initialize local memory and required parameter
*/
size_t pos = get_local_id(0);
localRes[pos] = 0;
barrier(CLK_LOCAL_MEM_FENCE);

unsigned int firstPS,lastPS;
firstPS = ps[get_group_id(0)*get_local_size(0)];
lastPS = ps[((get_group_id(0)+1)*get_local_size(0)) - 1];

/*
* Variant processing step for aggregation
*/
int insertPos = ps[get_global_id(0)] - firstPS;
atomic_add(&localRes[insertPos],1);
barrier(CLK_GLOBAL_MEM_FENCE);

/*
* Push results to global memory from local
*/
//Fill in first and last position
if(get_local_id(0) == 0){
        atomic_add(&res[ps[get_global_id(0)]],localRes[get_local_id(0)]);

        if(lastPS - firstPS > 1){
                for (int i = 1; i < lastPS - firstPS; i++)
                res[firstPS + i] += localRes[i];
        }
        return;
}
else if(get_local_id(0) == get_local_size(0) - 1){

        if(lastPS!=firstPS)
        atomic_add(&res[lastPS],localRes[ps[get_global_id(0)] - firstPS]);
        return;
}
}
```

To store partial aggregates in the local memory, we have to initialize the local memory. The Listing A.2 initializes local memory in lines 1-3. Next, we have to store the partial results onto global memory which is also given in the Listing.

Private variable aggretation over sorted values on global memory

```
void kernel branched_aggregate(
global const int* PS,
global int* res
) {

int local_res=1;
size_t pos = get_global_id(0) * ITERATOR;
for (unsigned int i = 0;i < ITERATOR-1; ++i){

        if((PS[pos + i] - PS[pos + i + 1])){
                atomic_add(&res[PS[pos + i]],local_res);
                local_res=1;
        }
        else
        local_res++;
}
atomic_add(&res[PS[pos + ITERATOR - 1]],local_res);
}
```

Private array aggretation over sorted values on global memory

```
void kernel private_array_sequential_aggregate(
global int* PS,
global int* res
){

int local_ps[ITERATOR];
int local_res[ITERATOR];
size_t pos = get_global_id(0) * ITERATOR;
int lc = 0;

local_ps[lc] = PS[pos];
local_res[lc] = 1;

for (unsigned int i = 0;i < ITERATOR - 1 ; ++i){
        if((PS[pos + i] - PS[pos + i + 1])){
                lc++;
                local_ps[lc] = PS[pos + i +1];
                local_res[lc] =1;
        }
        else
        local_res[lc]++;
}
atomic_add(&res[local_ps[0]],local_res[0]);

for(int i = 1;i < lc;i++){
        res[local_ps[i]]+=local_res[i];
}

if(lc!=0)
        atomic_add(&res[local_ps[lc]],local_res[lc]);
}
```

As mentioned, since atomics is the key factor influencing the execution, we use the same for aggregation via hashing. In our case, we use simple linear probing has our hashing technique as shown in Listing A.2.

Group by aggregation using linear probing - atomics variant

```
#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable
#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable

__kernel void hashAggregation(__global int* input, __global int* result, unsigned
    ↪ int HASH_TABLE_SIZE){

int insert_val = input[get_global_id(0)];
if(!insert_val)
return;

size_t position = 1300000077*input[get_global_id(0)]%HASH_TABLE_SIZE;
for(size_t i = 0; i< HASH_TABLE_SIZE; i++){
        int test_val = atomic_cmpxchg(&result[position*2],0,insert_val);
        if ((test_val==0) ||(test_val == insert_val)){
                atomic_add(&result[position*2+1],1);
                break;
        }

        position = (position+1)%(HASH_TABLE_SIZE);
}
}
```

A.3 Tether - Linking Vectorwise with Hyper

One of the critical component in our Tether framework is the function that links vectorwise with hyper. This is essentially the function that allows the execution to switch from interpreted mode to compiled execution, thereby triggering faster query execution. We achieve this using threading where - thread 1 starts compilation; thread 2 executes query using interpreted execution. The code for this execution is presented below.

Group by aggregation using linear probing - atomics variant

```
// Execute using hybrid approach
Relation q6_hybrid(Database& db, size_t nrThreads, size_t vectorSize,
const std::string& path_to_lib_src, bool fromLLVM,
bool verbose) {
using namespace vectorwise;
using namespace std::chrono_literals;

// 1. START COMPILING Q6 IN TYPER
std::atomic<hybrid::SharedLibrary*> typerLib(nullptr);
std::thread compilationThread([&typerLib, &path_to_lib_src, &fromLLVM,
&verbose] {
        const std::string& path_to_lib =
        hybrid::CompilationEngine::instance().linkQueryLib(path_to_lib_src,
        fromLLVM);
        // open library
        typerLib = hybrid::SharedLibrary::load(path_to_lib + ".so");
        auto end = std::chrono::steady_clock::now();
});

// 2. WHILE Q6 IS COMPILING, START TECTORWISE
vectorwise::SharedStateManager shared;
WorkerGroup workers(nrThreads);
GlobalPool pool;
std::atomic<int64_t> aggr(0);
std::atomic<size_t> processedTuples(0);
workers.run([&] {
// init query
auto query = queryBuilder.getQuery();
// get top operator
```

```
std::unique_ptr<vectorwise::FixedAggr> topAggr( static_cast<vectorwise::FixedAggr
    ↪ *>(query->rootOp.release()));

// while children provide tuples, compute the aggregate
for (auto pos = topAggr->child->next(); pos != EndOfStream && !typerLib;
pos = topAggr->child->next()) {
        topAggr->aggregates.evaluate(pos);
        processedTuples.fetch_add(vectorSize);
}
// store result from this thread
aggr.fetch_add(query->aggregator);
});

// 3. PROCESS REMAINING TUPLES WITH TYPER
compilationThread.join();
size_t nrTuples = db["lineitem"].nrTuples;
if (processedTuples.load() > nrTuples) { processedTuples.store(nrTuples); }
// load library
if (!typerLib) {
        throw hybrid::HybridException("Could␣not␣load␣shared␣Typer␣library!");
}
// get compiled function
const std::string& funcName = "_Z15hybrid_typer_q6RN7runtime8DatabaseEmml";
hybrid::CompiledTyperQ6 typer_q6 = typerLib.load()->getFunction<hybrid::
    ↪ CompiledTyperQ6>(funcName);
// compute typer result
Relation result =
typer_q6(db, nrThreads, processedTuples.load(), aggr.load());
end = std::chrono::steady_clock::now();
return result;
}
```

Bibliography

[1] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. *General-Purpose Graphics Processor Architectures*. Morgan & Claypool Publishers, 2018. (cited on Page 46)

[2] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented DBMS. In *International Conference on Data Engineering (ICDE)*, pages 466–475, 2007. (cited on Page 67 and 68)

[3] Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J Halstead, Walid A Najjar, and Vassilis J Tsotras. FPGA-accelerated group-by aggregation using synchronizing caches. *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–9, 2016. (cited on Page 1)

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment (VLDB)*, 5(10):1064–1075, June 2012. (cited on Page 70)

[5] Markus Ålind, Mattias V Eriksson, and Christoph W Kessler. Blocklib: a skeleton library for cell broadband engine. In *Proceedings of the international workshop on Multicore software engineering (IWMSE)*, pages 7–14, 2008. (cited on Page 38)

[6] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. *ACM Special Interest Group on Programming Languages notices (SIGPLAN)*, 44(6):38–49, 2009. (cited on Page 40)

[7] I. Arefyeva, D. Broneske, M. Pinnecke, M. Bhatnagar, and G. Saake. Column vs. row stores for data manipulation in hardware oblivious cpu/gpu database systems. In *GI-Workshop Grundlagen von Datenbanken (GvDB)*, pages 24–29. CEUR-WS, 2017. (cited on Page 80)

[8] I. Arefyeva, G. Campero Durand, M. Pinnecke, D. Broneske, and G. Saake. Low-latency transaction execution on graphics processors: Dream or reality? In *Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2018. (cited on Page 75)

[9] Iya Arefyeva, David Broneske, Gabriel Campero, Marcus Pinnecke, and Gunter Saake. Memory management strategies in CPU/GPU database systems: A

survey. In *Proceedings of the International Conference Beyond Databases, Architectures and Structures (BDAS)*, pages 128–142. Springer, September 2018. (cited on Page 44)

[10] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1383–1394, 2015. (cited on Page 95)

[11] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, et al. Share the tensor tea: how databases can leverage the machine learning ecosystem. *arXiv preprint arXiv:2209.04579*, 2022. (cited on Page 36)

[12] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. Data parallel c++ enhancing sycl through extensions for productivity and performance. In *Proceedings of the International Workshop on OpenCL (IWOCL)*, pages 1–2, 2020. (cited on Page 78)

[13] Cédric Augonnet and Raymond Namyst. A unified runtime system for heterogeneous multi-core architectures. In *International European Conference on Paralleland Distributed Computing Workshops (Euro-Par)*, pages 174–183. Springer, 2009. (cited on Page 41)

[14] Cédric Augonnet, Samuel Thibault, et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2), 2011. (cited on Page 41, 113, and 114)

[15] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. *Proceedings of the Workshop on General–Purpose Computation on Graphics Processing Units*, pages 94–103, 2010. (cited on Page 1, 36, 46, 75, 78, and 113)

[16] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment (VLDB)*, 7(1):85–96, 2013. (cited on Page 43)

[17] Gerassimos Barlas. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014. (cited on Page 19)

[18] Andreas Becher, Lekshmi B.G., et al. Integration of FPGAs in database management systems: Challenges and opportunities. *Datenbank-Spektrum*, 2018. (cited on Page 44)

[19] Andreas Becher, Achim Herrmann, Stefan Wildermann, and Jürgen Teich. Reprovide: Towards utilizing heterogeneous partially reconfigurable architectures for near-memory data processing. *Database Systems for Business, Technology and Web Workshops (BTW)*, 2019. (cited on Page 36)

[20] Andreas Becher, Achim Herrmann, Stefan Wildermann, and Jürgen Teich. Reprovide: Towards utilizing heterogeneous partially reconfigurable architectures for near-memory data processing. In Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke, editors, *Database Systems for Business, Technology and Web Workshops (BTW)*, pages 51–70. Gesellschaft für Informatik, Bonn, 2019. (cited on Page 119)

[21] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2018. (cited on Page 114)

[22] Tobias Behrens, Viktor Rosenfeld, Jonas Traub, Sebastian Breß, and Volker Markl. Efficient SIMD Vectorization for Hashing in OpenCL. *International Conference on Extending Database Technology (EDBT)*, pages 3–6, 2018. (cited on Page 43, 44, 46, 59, and 75)

[23] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A survey of high-level parallel programming models. *Heriot-Watt University, Edinburgh, UK*, 1(2):2–2, 2013. (cited on Page 40 and 41)

[24] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*. Elsevier, 2012. (cited on Page 76 and 79)

[25] Siegfried Benkner, Sabri Pllana, Jesper Larsson Traff, Philippas Tsigas, Uwe Dolinsky, Cedric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov. Peppher: Efficient and productive usage of hybrid computing systems. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 31(5):28–41, 2011. (cited on Page 23 and 37)

[26] BlazingDB. High Performance GPU Database for Big Data SQL. 2015. (cited on Page 76 and 78)

[27] Guy E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, 1990. (cited on Page 64 and 65)

[28] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM Special Interest Group on Programming Languages notices (SIGPLAN)*, 30(8):207–216, 1995. (cited on Page 23)

[29] P. A. Boncz and M. L. Kersten. Mil primitives for querying a fragmented world. *Proceedings of the VLDB Endowment (VLDB)*, 8(2):101–119, Oct 1999. (cited on Page 2 and 62)

[30] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013. (cited on Page 44)

[31] Peter A Boncz, Martin L Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008. (cited on Page 2 and 36)

[32] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013. (cited on Page 41 and 114)

[33] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012. (cited on Page 114)

[34] Alexander Branover, Denis Foley, and Maurice Steinman. Amd fusion apu: Llano. *Ieee Micro*, 32(2):28–37, 2012. (cited on Page 14)

[35] Sebastian Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated dbms. *DBS*, 14(3):199–209, 2014. (cited on Page 75, 78, and 113)

[36] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1891–1906, 2016. (cited on Page 44)

[37] Sebastian Breß, Max Heimel, et al. GPU-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data and Knowledge-Centered Systems (TLDKS)*, pages 1–35. Springer, 2014. (cited on Page 1, 36, and 113)

[38] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-accelerated database systems: Survey and open challenges. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, pages 1–35, 2014. (cited on Page 12)

[39] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating custom code for efficient query execution on heterogeneous processors. *Proceedings of the VLDB Endowment (VLDB)*, 27(6):797–822, December 2018. (cited on Page 26, 37, 38, 73, 78, 114, and 120)

[40] Robert A Bridges, Neena Imam, and Tiffany M Mintz. Understanding GPU power: A survey of profiling, modeling, and simulation methods. *ACM Computing Surveys (CSUR)*, 49(3), 2016. (cited on Page 114)

[41] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward hardware-sensitive database operations. *International Conference on Extending Database Technology (EDBT)*, pages 1–6, 2014. (cited on Page 2, 29, 43, 67, 75, and 111)

[42] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating multi-column selection predicates in main-memory - the Elf approach. In

International Conference on Data Engineering (ICDE), pages 647–658. IEEE, 2017. (cited on Page 71)

[43] David Broneske, Andreas Meister, and Gunter Saake. Hardware-sensitive scan operator variants for compiled selection pipelines. In *Database Systems for Business, Technology and Web (BTW)*, pages 403–412, 2017. (cited on Page 25 and 67)

[44] Brytlyt. World's most advanced GPU accelerated database. 2013. (cited on Page 76)

[45] Daniel Cederman and Philippas Tsigas. GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. *Journal of Experimental Algorithmics (JEA)*, pages 4:1.4—-4:1.24, jan 2010. (cited on Page 6)

[46] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001. (cited on Page 23)

[47] Cheng-Hsiang Chiu, Tsung-Wei Huang, Zizheng Guo, and Yibo Lin. Pipeflow: An efficient task-parallel pipeline programming framework using modern c++. *arXiv preprint arXiv:2202.00717*, 2022. (cited on Page 41)

[48] Iris Christadler and Volker Weinberg. Rapidmind: Portability across architectures and its limitations. *Facing the multicore-challenge: aspects of new paradigms and technologies in parallel computing*, pages 4–15, 2010. (cited on Page 37)

[49] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. Technical report, 2019. (cited on Page 27 and 114)

[50] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *Conference on Innovative Data Systems Research (CIDR)*, number CONF, 2019. (cited on Page 114)

[51] Hawon Chu, Seounghyun Kim, Joo-Young Lee, and Young-Kyoon Suh. Empirical evaluation across multiple gpu-accelerated dbmses. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–3, 2020. (cited on Page 113)

[52] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989. (cited on Page 24)

[53] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96. IEEE, 2018. (cited on Page 16)

[54] T.Mostak C.Root. Mapd: A GPU-powered big data analytics and visualization platform. *Special Interest Group on Computer Graphics and Interactive Techniques Conference (SIGGRAPH)*, pages 73–74, 2016. (cited on Page 12)

[55] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An architecture for compiling UDF-centric workflows. *Proceedings of the VLDB Endowment (VLDB)*, 8(12):1466–1477, 2015. (cited on Page 95)

[56] Marco Danelutto and Massimo Torquati. Structured parallel programming with "core" fastflow. *Central European Functional Programming School (CEFP)*, pages 29–75, 2015. (cited on Page 37)

[57] G Diamos, H Wu, A Lele, J Wang, and et al. Efficient relational algebra algorithms and data structures for GPU. Technical report, Georgia Institute of Technology, 2012. (cited on Page 72 and 73)

[58] Yuri Dotsenko, Naga K Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast Scan Algorithms on Graphics Processors. In *International Conference on Supercomputing (ICS)*, pages 205–213, 2008. (cited on Page 32)

[59] Markus Dreseler, Jan Kossmann, Johannes Frohnhofen, Matthias Uflacker, and Hasso Plattner. Fused table scans: Combining AVX-512 and JIT to double the performance of multi-predicate scans. In *International Conference on Data Engineering Workshops (ICDEW)*, pages 102–109. IEEE, 2018. (cited on Page 96)

[60] Tobias Drewes, Jan Moritz Joseph, and Thilo Pionteck. An fpga-based prototyping framework for networks-on-chip. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7. IEEE, 2017. (cited on Page 36)

[61] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014. (cited on Page 114)

[62] Kento Emoto and Kiminori Matsuzaki. An automatic fusion mechanism for variable-length list skeletons in sketo. *International Journal of Parallel Programming*, 42:546–563, 2014. (cited on Page 37)

[63] Johan Enmyren and Christoph W Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *International workshop on High-level parallel programming and applications (HLPP)*, pages 5–14, 2010. (cited on Page 37 and 38)

[64] Steffen Ernsting and Herbert Kuchen. A scalable farm skeleton for heterogeneous parallel programming. *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, 25:72, 2014. (cited on Page 37)

[65] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. Skepu 3: Portable high-level programming of heterogeneous systems and hpc

clusters. *International Journal of Parallel Programming*, 49(6):846–866, 2021. (cited on Page 37)

[66] August Ernstsson, Lu Li, and Christoph Kessler. Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46:62–80, 2018. (cited on Page 37)

[67] Jian Fang, Yvo TB Mulder, et al. In-memory database acceleration on FPGAs: a survey. *Proceedings of the VLDB Endowment (VLDB)*, 29(1), 2020. (cited on Page 1, 16, 36, and 113)

[68] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing (CCF THPC)*, 2:382–400, 2020. (cited on Page 23 and 25)

[69] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. GPUQP: Query co-processing using graphics processors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1061–1063, 2007. (cited on Page 77 and 78)

[70] Rob Farber. *Parallel programming with OpenACC*. Newnes, 2016. (cited on Page 19)

[71] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 83–es, 2006. (cited on Page 37)

[72] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. In *Conference on Programming language design and implementation (PLDI)*, pages 212–223, 1998. (cited on Page)

[73] Henning Funke, Sebastian Breß, et al. Pipelined query processing in coprocessor environments. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2018. (cited on Page 37, 38, 114, and 123)

[74] Henning Funke, Jan Mühlig, and Jens Teubner. Efficient generation of machine code for query compilers. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–7, 2020. (cited on Page 37)

[75] Henning Funke and Jens Teubner. Low-latency compilation of sql queries to machine code. *Proceedings of the VLDB Endowment (VLDB)*, 14(12):2691–2694, 2021. (cited on Page 37)

[76] Thierry Gautier, Joao VF Lima, et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013. (cited on Page 113)

[77] David B. Glasco, Peter B. Holmqvist, George R. Lynch, Patrick R. Marchand, Karan Mehra, and James Roberts. Cache-based control of atomic operations in conjunction with an external alu block, March 13 2012. US Patent 8,135,926. (cited on Page 46)

[78] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 325–336, 2006. (cited on Page 36)

[79] Naga Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, University of North Carolina, June 2005. (cited on Page 70)

[80] Naga K Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, page 215, 2004. (cited on Page 1 and 43)

[81] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–170, 1993. (cited on Page 72)

[82] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 6(1):120–135, 1994. (cited on Page 96)

[83] Tim Gubner, Diego Tomé, Harald Lang, and Peter Boncz. Fluid co-processing: Gpu bloom-filters for cpu joins. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–10, 2019. (cited on Page 114)

[84] Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero Durand, and Gunter Saake. SIMD vectorized hashing for grouped aggregation. In *Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 113 – 126, 2018. (cited on Page 44)

[85] Mark Harris, John Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson. CUDPP: CUDA data parallel primitives library, Dec 2016. (cited on Page 76)

[86] M. Hauck, M. Paradies, and H. Fröning. Software-based buffering of associative operations on random memory addresses. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 943–952, May 2019. (cited on Page 46)

[87] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):1–39, 2009. (cited on Page 1, 6, 12, 32, 36, 44, 46, 50, 62, 75, 113, and 120)

[88] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 511–524. ACM, 2008. (cited on Page 66, 69, and 70)

[89] Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. Query processing on tensor computation runtimes. *arXiv preprint arXiv:2203.01877*, 2022. (cited on Page 36)

[90] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache query co-processing on coupled cpu-gpu architectures. *Proceedings of the VLDB Endowment (VLDB)*, 8(4):329–340, 2014. (cited on Page 15)

[91] Max Heimel, Michael Saecker, et al. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment (VLDB)*, 6(9), 2013. (cited on Page 1, 17, 37, 75, 78, 112, and 113)

[92] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011. (cited on Page 64)

[93] Pedro Holanda and Hannes Mühleisen. Relational queries with a tensor processing unit. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–3, 2019. (cited on Page 13, 36, and 113)

[94] Daniel Horn. Stream reduction operations for GPGPU applications. In Matt Pharr, editor, *GPU Gems*, volume 2, pages 573–589. Addison-Wesley, 2005. (cited on Page 65)

[95] Fazeleh Hoseini, Aras Atalar, and Philippas Tsigas. Modeling the performance of atomic primitives on modern architectures. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 28:1–28:11, New York, NY, USA, 2019. ACM. (cited on Page 46)

[96] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. Tcudb: Accelerating database with tensor processors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, SIGMOD '22, page 1360–1374, New York, NY, USA, 2022. Association for Computing Machinery. (cited on Page 12, 13, 36, and 113)

[97] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Cpp-taskflow: Fast task-based parallel programming using modern c++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 974–983. IEEE, 2019. (cited on Page 41)

[98] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *international conference on cloud computing*

technology and science (CloudCom), pages 159–168. IEEE, 2010. (cited on Page 14)

[99] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 38(3):10–19, 2018. (cited on Page 13)

[100] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017. (cited on Page 13)

[101] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU join processing revisited. *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 55–62, 2012. (cited on Page 6 and 75)

[102] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint*, 2010. (cited on Page 114)

[103] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. Big Data Causing Big (TLB) Problems: Taming Random Memory Accesses on the GPU. *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, (2):6:1—-6:10, 2017. (cited on Page 43)

[104] Tomas Karnagel, René Müller, and Guy M. Lohman. Optimizing GPU-accelerated group-by and aggregation. *Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–12, 2015. (cited on Page 44, 46, 57, 59, and 75)

[105] Chetana N Keltcher, Kevin J McGrath, Ardsher Ahmed, and Pat Conway. The amd opteron processor for multiprocessor servers. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 23(2):66–76, 2003. (cited on Page 17)

[106] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment (VLDB)*, 11(13):2209–2222, 2018. (cited on Page 94, 95, 96, 99, and 100)

[107] Changkyu Kim et al. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 339–350. ACM, 2010. (cited on Page 71)

[108] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment (VLDB)*, page 1378–1389, 2009. (cited on Page 59)

[109] Donald E Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison Wesley Longman Publishing Co., Inc., 3rd edition, 1997. (cited on Page 68)

[110] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. In *International Conference on Data Engineering (ICDE)*, pages 197–208. IEEE, 2018. (cited on Page 94 and 95)

[111] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, pages 555–569, 2016. (cited on Page 113)

[112] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *International Conference on Data Engineering (ICDE)*, pages 613–624. IEEE, 2010. (cited on Page 95)

[113] Herbert Kuchen. A skeleton library. In *International European Conference on Paralleland Distributed Computing Workshops (Euro-Par)*, pages 620–629. Springer, 2002. (cited on Page 37 and 38)

[114] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008. (cited on Page 114)

[115] Tobias Lauer, Amitava Datta, Zurab Khadikov, and Christoffer Anselm. Exploring Graphics Processing Units As Parallel Coprocessors for Online Aggregation. In *Proceedings of the ACM International Workshop on Data Warehousing and OLAP*, pages 77–84. ACM, 2010. (cited on Page 46)

[116] O. S. Lawlor. Embedding OpenCL in C++ for expressive GPU programming. In *International Workshop on Domain-Specific Languages and High-Level Frameworks (WOLFHPC)*, 2011. (cited on Page 79)

[117] Joeffrey Legaux, Frédéric Loulergue, and Sylvain Jubertie. Osl: an algorithmic skeleton library with exceptions. *Procedia Computer Science*, 18:260–269, 2013. (cited on Page 37)

[118] Kevin Lepak, Gerry Talbot, Sean White, Noah Beck, Sam Naffziger, SENIOR FELLOW, et al. The next generation amd enterprise server product architecture. *IEEE hot chips*, 29, 2017. (cited on Page 17)

[119] Mario Leyton and José M Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 289–296. IEEE, 2010. (cited on Page 37)

[120] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55, 2009. (cited on Page 40)

[121] Clemens Lutz, Sebastian Breß, et al. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2020. (cited on Page 80 and 114)

[122] Andreas Meister, Sebastian Breß, and Gunter Saake. Toward gpu-accelerated database optimization. *Datenbank-Spektrum*, 15:131–140, 2015. (cited on Page 23)

[123] Suejb Memeti, Lu Li, et al. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC)*, 2017. (cited on Page 116)

[124] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment (VLDB)*, 11(1):1–13, 2017. (cited on Page 95)

[125] Sparsh Mittal. A survey on evaluating and optimizing performance of intel xeon phi. *Concurrency and Computation: Practice and Experience*, 32(19):e5742, 2020. (cited on Page 14)

[126] Sparsh Mittal and Jeffrey S Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015. (cited on Page 77)

[127] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, and Norman May. Comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, 2020. (cited on Page 6)

[128] Mahmoud Mohsen, Norman May, Christian Färber, and David Broneske. Fpga-accelerated compression of integer vectors. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery. (cited on Page 6)

[129] Michael Müller, Thomas Leich, Thilo Pionteck, Gunter Saake, Jens Teubner, and Olaf Spinczyk. He..ro db: A concept for parallel data processing on heterogeneous hardware. In André Brinkmann, Wolfgang Karl, Stefan Lankes, Sven Tomforde, Thilo Pionteck, and Carsten Trinitis, editors, *Architecture of Computing Systems (ARCS)*, Cham, 2020. (cited on Page 114)

[130] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011. (cited on Page 19)

[131] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment (VLDB)*, 4(9):539–550, 2011. (cited on Page 32, 65, 72, 95, and 96)

[132] Thomas Neumann and Viktor Leis. Compiling database queries into machine code. *IEEE Data Engineering Bulletin*, 37(1):3–11, 2014. (cited on Page 93, 94, 96, and 97)

[133] John Owens. Gpu architecture overview. *Special Interest Group on Computer Graphics and Interactive Techniques Conference (SIGGRAPH)*, 10, 2007. (cited on Page 46)

[134] John D Owens, David Luebke, et al. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, 2007. (cited on Page 114)

[135] Prasanna Pandit and R Govindarajan. Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices. In *International Symposium on Code Generation and Optimization (CGO)*, pages 273–283, 2014. (cited on Page 27 and 114)

[136] Styliani Pantela and Stratos Idreos. One loop does not fit all. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 2073–2074, 2015. (cited on Page 64)

[137] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment (VLDB)*, 11(6):663–676, 2018. (cited on Page 95)

[138] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Revisiting hash join on graphics processors: A decade later. *Distributed and Parallel Databases*, 38:771–793, 2020. (cited on Page 89)

[139] Marcus Pinnecke, David Broneske, Gabriel Campero Durand, and Gunter Saake. Are databases fit for hybrid workloads on GPUs? A storage engine's perspective. In *International Conference on Data Engineering (ICDE)*, pages 1599–1606, 2017. (cited on Page 80)

[140] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment (VLDB)*, 9(14), 2016. (cited on Page 37, 38, 62, 114, and 120)

[141] Constantin Pohl and Kai-Uwe Sattler. Joins in a heterogeneous memory hierarchy: Exploiting high-bandwidth memory. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–10, 2018. (cited on Page 14)

[142] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2015. (cited on Page 43, 68, and 111)

[143] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 755–766, 2014. (cited on Page 69)

[144] Orestis Polychroniou and Kenneth A. Ross. Efficient lightweight compression alongside fast scans. *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, 2015. (cited on Page 67)

[145] Raphael Poss, Mike Lankamp, Qiang Yang, Jian Fu, Michiel W van Tol, I Uddin, and C Jesshope. Apple-core: harnessing general-purpose many-cores with hardware concurrency management. *Microprocessors and Microsystems*, 37(8):1090–1101, 2013. (cited on Page 37)

[146] SARC European Project. Parallel programming models for heterogeneous multicore architectures. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 30(5):42–53, 2010. (cited on Page 37)

[147] J. Teubner R. Müller and G. Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment (VLDB)*, 2(1):910–921, 2009. (cited on Page 1)

[148] Jun Rao and Kenneth Ross. Making B^+-Trees cache conscious in main memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 475–486. ACM, 2000. (cited on Page 71)

[149] Hannes Rauhe, Jonathan Dees, Kai-Uwe Sattler, and Franz Faerber. Multilevel parallel query execution framework for CPU and GPU. In *Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 330–343. Springer, 2013. (cited on Page 71, 72, and 73)

[150] Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proceedings of the VLDB Endowment (VLDB)*, 9(3):96–107, 2015. (cited on Page 68)

[151] Krzysztof Rojek and Roman Wyrzykowski. Performance and scalability analysis of ai-accelerated cfd simulations across various computing platforms. In *International European Conference on Paralleland Distributed Computing Workshops (Euro-Par)*, pages 223–234. Springer, 2022. (cited on Page 14)

[152] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *International conference on Generative programming and component engineering (GPCE)*, pages 127–136, 2010. (cited on Page 40)

[153] Christopher Root and Todd Mostak. MapD: A GPU-powered big data analytics and visualization platform. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference (SIGGRAPH)*. 2016. (cited on Page 132)

[154] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)*, 55(1), 2022. (cited on Page 1 and 36)

[155] Viktor Rosenfeld, Max Heimel, Christoph Viebig, and Volker Markl. The operator variant selection problem on heterogeneous hardware. *Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2015. (cited on Page 36, 54, 65, 70, and 75)

[156] K. A. Ross. Efficient hash probes on modern processors. In *International Conference on Data Engineering (ICDE)*, pages 1297–1301, 2007. (cited on Page 68)

[157] Kenneth A. Ross. Selection conditions in main memory. *ACM Transactions on Database Systems (TODS)*, 29(1):132–161, 2004. (cited on Page 67)

[158] Ran Rui, Hao Li, and Yi-Cheng Tu. Join algorithms on gpus: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2541–2550. IEEE, 2015. (cited on Page 89)

[159] Behzad Salami, Gorker Alp Malazgirt, Oriol Arcas-Abella, Arda Yurdakul, and Nehir Sonmez. Axledb: A novel programmable query processing platform on fpga. *Microprocessors and Microsystems*, 51:142–164, 2017. (cited on Page 113)

[160] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997. (cited on Page 9 and 10)

[161] Michael Scherger. Design of an in-memory database engine using intel xeon phi coprocessors. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2014. (cited on Page 14)

[162] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. On the surprising difficulty of simple things. *Proceedings of the VLDB Endowment (VLDB)*, 8(9):934–937, 2015. (cited on Page 69)

[163] Karsten Schwan, Ada Gavrilovska, and Sudha Yalamanchili. HyVM-hybrid virtual machines-efficient use of future heterogeneous chip multiprocessors. In *Architecture of Computing Systems (ARCS)*, pages 1–1. Springer, 2010. (cited on Page 37)

[164] Todd C Scofield, Jeffrey A Delmerico, Vipin Chaudhary, and Geno Valente. Xtremedata dbx: an fpga-based data warehouse appliance. *Computing in Science & Engineering*, 12(4):66–73, 2010. (cited on Page 113)

[165] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for GPU computing. In *Graphics Hardware*, pages 97–106. Eurographics Association, 2007. (cited on Page 65 and 84)

[166] David Sidler, Zsolt István, et al. doppiodb: A hardware accelerated database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017. (cited on Page 113)

[167] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of GPU based sorting algorithms. *International Journal of Parallel Programming*, 46(6):1017–1034, 2018. (cited on Page 84)

[168] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on GPUs. In *International Conference on Data Engineering (ICDE)*, 2019. (cited on Page 75)

[169] Evangelia A Sitaridi and Kenneth A Ross. Optimizing select conditions on GPUs. *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–8, 2013. (cited on Page 67)

[170] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, 2016. (cited on Page 13)

[171] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 36(2):34–46, 2016. (cited on Page 13)

[172] SQream Technologies. GPU based SQL database. 2010. (cited on Page 76 and 78)

[173] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl-a portable skeleton library for high-level gpu programming. In *International Symposium on Parallel and Distributed Processing (IPDPS) Workshops and Phd Forum*, pages 1176–1182. IEEE, 2011. (cited on Page 25, 37, 38, and 79)

[174] Harish Kumar Harihara Subramanian, Bala Gurumurthy, et al. Analysis of GPU-libraries for rapid prototyping database operations: A look into library support for database operations. In *International Conference on Data Engineering Workshops (ICDEW)*, 2021. (cited on Page 114)

[175] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–25, 2014. (cited on Page 40)

[176] Yifan Sun, Saoni Mukherjee, Trinayan Baruah, Shi Dong, Julian Gutierrez, Prannoy Mohan, and David Kaeli. Evaluating performance tradeoffs on the radeon open compute platform. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 209–218. IEEE, 2018. (cited on Page 78)

[177] J. Szuppe. Boost.Compute: A parallel computing library for C++ based on OpenCL. *International Workshop on OpenCL (IOWCL)*, 15:1–39, 2016. (cited on Page 79)

[178] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018. (cited on Page 23 and 41)

[179] Diego G Tome, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A. Boncz. Optimizing Group-By and Aggregation using GPU-CPU Co-Processing.

International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, pages 1–10, 2018. (cited on Page 46)

[180] Chun-Wei Tsai, Chin-Feng Lai, Han-Chieh Chao, and Athanasios V Vasilakos. Big data analytics: a survey. *Journal of Big data*, 2(1):1–32, 2015. (cited on Page 9)

[181] Hans Vandierendonck, Polyvios Pratikakis, Dimitrios S Nikolopoulos, et al. Parallel programming of general-purpose programs using task-based programming models. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2011. (cited on Page 27)

[182] Marten Wallewein-Eising, David Broneske, and Gunter Saake. Simd acceleration for main-memory index structures – a survey. In *Proceedings of the International Conference Beyond Databases, Architectures and Structures (BDAS)*, sep 2018. accepted for publication. (cited on Page 43)

[183] Skye Wanderman-Milne and Nong Li. Runtime code generation in Cloudera Impala. *IEEE Data Engineering Bulletin*, 37(1):31–37, 2014. (cited on Page 95)

[184] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. In *Proceedings of the VLDB Endowment (VLDB)*, volume 5, page 1543. NIH Public Access, 2012. (cited on Page 15)

[185] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019. (cited on Page 13)

[186] John Robert Wernsing and Greg Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. *ACM Special Interest Group on Programming Languages (SIGPLAN)*, 45(4):115–124, 2010. (cited on Page 27, 41, and 114)

[187] Thomas Willhalm, Yazan Boshmaf, Hasso Plattner, Nicolae Popovici, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment (VLDB)*, 2(1):385–394, 2009. (cited on Page 67)

[188] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. Vectorizing database column scans with complex predicates. In *Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–12, 2013. (cited on Page 67)

[189] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 107–118, 2012. (cited on Page 37, 38, 72, and 73)

[190] Steffen Zeuch and Johann-christoph Freytag. Selection on modern CPUs. *Proceedings of the International Workshop on In-Memory Data Mangement and Analytics (IMDM)*, pages 1–8, 2015. (cited on Page 67)

[191] Steffen Zeuch, Johann-Christoph Freytag, and Frank Huber. Adapting tree structures for processing with SIMD instructions. In *International Conference on Extending Database Technology (EDBT)*, pages 97–108, 2014. (cited on Page 71)

[192] Shuhao Zhang, Jiong He, et al. Omnidb: Towards portable and efficient query processing on parallel CPU/GPU architectures. *Proceedings of the VLDB Endowment (VLDB)*, 6(12), 2013. (cited on Page 36 and 113)

[193] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6. IEEE, 2010. (cited on Page 17)

[194] Daniel Ziener, Florian Bauer, et al. FPGA-based dynamically reconfigurable SQL query processing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(4), 2016. (cited on Page 113)

[195] Marcin Zukowski and Peter A Boncz. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin*, 35(1):21–27, 2012. (cited on Page 97)

[196] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. Vectorwise: A vectorized analytical dbms. In *International Conference on Data Engineering (ICDE)*, pages 1349–1350. IEEE, 2012. (cited on Page 96)

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

Magdeburg, 25th October 2023