EPFL

Thèse n° 8068

Efficient Analytical Query Processing on CPU-GPU Hardware Platforms

Présentée le 29 août 2022

Faculté informatique et communications
Laboratoire de systèmes et applications de traitement de données massives
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Periklis CHRYSOGELOS

Acceptée sur proposition du jury

Prof. M. Grossglauser, président du jury
Prof. A. Ailamaki, directrice de thèse
Prof. A. Pavlo, rapporteur
Prof. P. Boncz, rapporteur
Prof. S. Kashyap, rapporteur

École polytechnique fédérale de Lausanne

2022

Give me a place to stand, and I will move the earth.
— Archimedes

To my parents, Nikos and Polina,
and my partner, Georgia.

Acknowledgements

This thesis materialized only because I was fortunate enough to be surrounded by many great people who supported me and provided invaluable feedback.

First and foremost, I would like to thank my advisor, *Anastasia Ailamaki*. She has been a great teacher; her structured way of guidance and her encouragement to seek perfection and set ambitious goals, both in research and life, have been invaluable. Her patience, support, and understanding provide a second-family-like lab environment that fosters great research ideas. Further, she takes great care in providing us with all the necessary infrastructure and peculiar hardware configurations, without which this thesis would not be possible. I will always be grateful for the environment she created in the lab, her guidance, and the opportunity she gave me to be a member of the DIAS family.

I would also like to thank my jury members for their time investment and their constructive comments. Specifically, I would like to thank *Peter Boncz* and *Andy Pavlo* whose work highly influenced my thesis and their insightful questions and guidance helped in shaping it. Further, I am grateful to *Sanidhya Kashyap* for his very detailed comments, and for helping me improve the content of this thesis. Last but not least, I also thank *Matthias Grossglauser* for serving as my thesis committee president and creating a pleasant thesis exam environment.

I would also like to thank *Weiwei Gong* for being a great mentor during my internship at Oracle, and for her continuous support even beyond the internship, *Danica Porobic* for helping me discover this great team, and *Shasank Chavan, Jim Kearney, Tirthankar Lahir, Kantikiran Pasupuleti, Gary Smith, Hong Su,* and *Garret Swart* for their valuable feedback and support.

During my PhD, I had the pleasure to closely collaborate with multiple great people, namely *Manos Karpathiotakis, Raja Appuswamy, Aunn Raza, Panagiotis Sioulas, Viktor Sanca, Angelos Anadiotis, Ioannis Mytilinis, Odysseas Papapetrou, Anna Herlihy, Hamish Nicholson, Georgios Michas, Konstantinos Koukas, Vladimir Indjic,* and *Rubin Daija.* In addition to the direct collaborations, I also thank all the DIAS family for the great discussions, coffee breaks and in general the great time in the office: *Angelos, Anna, Aunn, Bikash, Christina, Darius, Diane, Dimitra, Eleni (x2), Erika, Fabienne, Foteini, Georgios, Hamish, Haoqiong, Ioannis, Konstantinos, Lionel, Manos, Margaret, Matt, Mirjana, Odysseas, Panagiotis, Srinivas, Stella, Tahir, Utku,* and *Viktor*. Likewise, I would like to thank the extended (alumni from before I joined, RAW

Labs SA, EPFL system admins, etc) DIAS family for a great time at lab gatherings, BBQs, and meetups: *Benjamin, Cesar, Danica, Ippokratis, Iraklis, Manos, Miguel, Pinar, Stéphane,* and *Thomas*. I am especially thankful to a set of people in DIAS: Manos Karpathiotakis and Raja Appuswamy with whom I collaborated in the first years of my PhD and they introduced me to research. In addition to being brilliant researchers, they are always there to discuss and provide guidance. Aunn Raza is always there both as a friend and as a colleague – including late at night and during countless early morning deadlines. Thank you for all the awesome, endless brainstorming sessions and for being the sounding board for all the technical absurdities. Viktor Sanca, eager to explore everything, help, brainstorm about any topic, and deep-dive into even the most obscure research ideas, has been a great, enthusiastic collaborator and a source of motivation. Angelos Anadiotis has been a great collaborator, withstood the hardness of time and accommodated countless incompatible schedules, from daylight incompatibilities to providing his nonstop support even in the absence of a viable desk or chair. Odysseas Papapetrou, while supposedly a hard shell, has been a constant source of joy as well as interesting research discussions and a reminder of more theoretic directions. Dimitra Tsaoussis, Erika Raetz, Margaret Church and Fabienne Ubezio provided tremendous assistance with many, often just-in-time, administrative and organizational tasks. Stéphane Ecuyer and Lionel Sambuc, our brilliant and passionate system administrators, were always available to assist and quickly fix every issue, provided tremendous assistance in planning the hardware configurations, coordinating with the suppliers, and helping satisfy all the niche requirements. Thank you all!

I am thankful to all my friends who made my life enjoyable all these years: *Akhil, Angelos, Ankita, Anna, Antonia, Aunn, Batool, Christina, Eirini, Iro, Lionel, Panagiotis, Panayiotis, Panos, Stelios, Stella, Thanos,* and *Viktor*. In addition to being a great colleague, Aunn is a great friend, the go-to guy for everything, from board-game nights to excursions, and a saviour while waiting for the "insta"-people during the trips. Christina is a great, full of joy company, a source of countless what-if questions, and a tzatziki without tzatziki aficionado. Viktor is always eager to assist and a reliable source of information for all things, from approximate query processing to useful tips and tricks for life in Switzerland. Panayiotis has been of tremendous help settling in and feeling integrated in the first years. Anna and Stelios are a great company and I really enjoy our trips together – I am also deeply thankful to them for making me feel comfortable both in Cyprus and Chios. Thank you all for a great time!

I am grateful to *Georgia* for her love and support. Georgia is a highly motivated and persistent person. Not only has she endured multiple of my deadlines, but she has also been a continuous source of joy, relaxation, and many great excursions – a perfect escape from the Ph.D. hassles. Georgia's encouragement and faith in me have been a great motivation: trying to reach even close to these standards made me improve tremendously. I am deeply thankful to Georgia and look forward to all our future endeavors!

Last but not least, I would like to thank my parents, *Nikos* and *Polina*, for their great support, for taking care of me, and for providing me with two stellar examples to follow. I am deeply thankful for all their support, for teaching me how to be who I am today, and for encouraging me to push my limits. Thank you for everything!

This research has been supported by grants from the School of Computer and Communication Sciences, EPFL, the Swiss National Science Foundation, project No. 200021_178894/1, "Efficient Real-time Analytics on General-Purpose GPUs", the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa), the H2020 – UE Framework Programme for Research & Innovation (2014-2020), 2017 – ERC2017-PoC, grant agreement number 768910, ViDaR, and the European Union's Horizon 2020-ICT-2018-2020, SmartDataLake, project 825041.

Lausanne, July 29, 2022 P.C.

Abstract

Timely insights lead to business growth and scientific breakthroughs but require analytical engines that cope with the ever-increasing data processing needs. Analytical engines relied on rapid CPU improvements, yet the end of Dennard scaling stopped the free lunch and resulted in a *heterogeneous hardware* landscape that challenges existing analytical engines. First, each device has its own specialized execution model and architecture, impeding interoperability. Second, the diversity in device microarchitectures requires a diverse range of optimizations. Third, while the multitude of devices provides additional acceleration opportunities, moving the data across devices is costly. Finally, with networking bandwidths similar to intra-server device connections, the server boundaries are blurred, providing optimization opportunities and requiring careful orchestration to avoid wasting resources.

In this thesis, we aim for engines tailored for heterogeneous hardware: abstracting out hardware heterogeneity to enable efficient execution across the devices despite their diversity. To this end, we design and implement techniques that are i) scalable through accelerator-level parallelism, and ii) efficient through query execution customization to the underlying accelerators and data transfer paths.

Regarding scalability, we propose a unifying execution model and a throughput-oriented system view to enable on-the-fly multi-device orchestration without requiring knowledge about hardware specifics. In addition, by decoupling data and control flow, this thesis enables late and direct data transfers within and across servers.

Regarding efficiency, we provide an execution model that limits operator instances to specific devices, enabling operators to customize themselves to a single device without concern for multi-device effects. In addition, by providing interconnect-aware transfer methods, this thesis minimizes the cost of offloading operations across devices.

This thesis redesigns analytical engines to exploit hardware heterogeneity. Instead of trading hardware efficiency for accelerator-level scalability, this thesis embraces heterogeneity. Our design enables scalable analytics across CPU-GPU hardware and achieves the analytical performance of optimally combining a CPU- and a GPU-optimized engine. As a result, users benefit from faster insights without requiring large clusters of machines. The proposed

accelerator-centric design paves the way toward analytical engines that benefit from hardware improvements across the hardware spectrum – instead of relying on single-processor advancements.

Keywords: database management systems, analytical query processing, execution engine, code generation, heterogeneous hardware, accelerator-level parallelism, parallel execution, accelerators, GPU, RDMA

Résumé

Des informations opportunes mènent au développement commercial et aux avancées scientifiques, mais nécessitent des moteurs d'analyse capables de gérer les besoins de traitement des données toujours plus nombreuses. Les moteurs analytiques se basaient sur des améliorations rapides du processeur (CPU), pourtant la fin de la mise à échelle de Dennard a abouti à un environnement hétérogène du hardware qui défie les moteurs analytiques existants. Tout d'abord, chaque appareil a son propre modèle d'exécution et d'architecture spécialisées, entravant l'interopérabilité. Deuxièmement, la diversité des microarchitectures des appareils nécessite une gamme variée d'optimisations. Troisièmement, alors que la multitude d'appareils offre des opportunités d'accélération supplémentaire, le déplacement des données entre les appareils est coûteux. Enfin, avec des bandes passantes réseau similaires à des connexions intra-serveur des appareils, les limites du serveur sont floues, offrant des opportunités d'optimisation et nécessitant une orchestration minutieuse afin d'éviter le gaspillage des ressources.

L'objectif de cette thèse est l'adaptation des moteurs à l'hétérogénéité du hardware : abstraire l'hétérogénéité du hardware pour permettre une exécution efficace sur tous les appareils malgré leur diversité. A cette effet, nous concevons et mettons en œuvre des techniques qui sont : i) évolutives grâce au parallélisme au niveau de l'accélérateur, et ii) efficaces grâce à la personnalisation de l'exécution des requêtes sur les accélérateurs sous-jacents et les voies de transfert de données.

Concernant la scalabilité, nous proposons un modèle d'exécution unificateur et un vue système orientée rendement pour activer l'orchestration des nombreuses appareils à la volée sans nécessiter la connaissance des spécificités du hardware. De plus, en découplant les flux de données et de contrôle, cette thèse permet les transferts de données tardifs et directs au sein et entre les serveurs.

En ce qui concerne l'efficacité, nous fournissons un modèle d'exécution qui limite les instances de l'opérateur à des appareils spécifiques, permettant aux opérateurs de se personnaliser sur un seul appareil sans se préoccuper des effets multi-appareils. De plus, en fournissant des méthodes de transfert compatibles avec les interconnexions, cette thèse minimise le coût des opérations de déchargement entre les appareils.

Résumé

Cette thèse redéfinit les moteurs analytiques pour exploiter l'hétérogénéité du hardware. Au lieu d'échanger l'efficacité du hardware pour une scalabilité au niveau d'accélérateur, cette thèse embrasse l'hétérogénéité. Notre modèle permet des analyses évolutives sur le hardware CPU-GPU et atteint les performances analytiques d'une combinaison optimale d'un processeur et d'un moteur optimisé pour le GPU. En conséquence, les utilisateurs bénéficient d'informations plus rapides sans nécessiter de gros clusters de machines. Le modèle proposé centré sur les accélérateurs ouvre la voie à des moteurs analytiques qui bénéficient d'améliorations hardware sur l'ensemble du spectre hardware - au lieu de compter sur les avancées d'un seul processeur.

Mots clés : systèmes de gestion de base de données, traitement analytique des requêtes, moteur d'exécution, génération de code, l'hétérogénéité du matériel informatique, parallélisme au niveau de l'accélérateur, exécution parallèle, accélérateurs, GPU, RDMA

Contents

List of Figures

List of Tables

1 Introduction

The demand for real-time intelligence increases as businesses from a wide range of domains increasingly require interactive data analytics to make insightful decisions in fast-breaking situations. As the amount of data accumulated by applications continues to grow faster than Moore's law, achieving real-time intelligence requires a computational infrastructure that can sift through billions of rows in milliseconds.

To optimize query execution, analytical query engine designs became hardware-conscious and specialized to the homogeneous multi-CPU architectures [6, 15, 47, 51, 60, 63, 78, 95]. State-of-the-art analytical engines use hardware-conscious algorithms [14, 78, 90, 92] that match the CPU microarchitecture. Techniques like *vector-at-a-time execution* [15] and *just-in-time code generation* [45, 46] reduce the query execution overheads, while the Exchange [30] operator and HyPer's Morsels [51] parallelize query execution in multi-core and multi-CPU configurations. Such parallelization techniques [30, 51] rely on the efficiency of CPU characteristics like cache coherence, efficient system-wide atomic operations, and shared memory.

Still, such approaches fall short in the new era of heterogeneous CPU-GPU hardware: The introduction of loosely coupled server architectures consisting of multiple accelerators, such as GPUs, breaks the traditional assumption of homogeneous CPU resources – bringing new challenges for hardware-efficient analytical processing. The heterogeneous devices come with a mix of shared and shared-nothing regions, even more acute NUMA effects, various programming models, and the closing bandwidth difference between intra- and inter-server interconnects breaks the traditional locality assumptions.

Figure 1.1 shows the throughput (working set size over time) of different systems when running all the SSB [71] queries[1]. We use two scale factors, SF=1000 for a working set that is bigger than the available GPU memory and SF=100 for a dataset that fits in the aggregated GPU memory. The performance of GPU-based DBMS highly depends on whether the data can fit in the GPU memory and the engine implementation. For example, the GPU commercial system is faster

[1] Exact experimental setup in Section 4.4

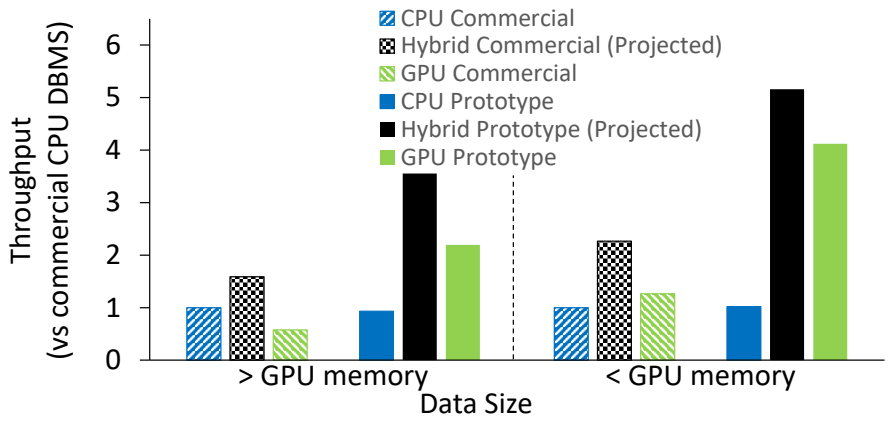


Figure 1.1: Relative throughput of different systems and the unrealized potential of hybrid CPU-GPU engines. The reported throughputs are normalized, per data size, over the corresponding throughput of the CPU commercial DBMS.

than the CPU-based one when the data reside in the GPU memory. As a result, the preferred underlying devices depend on both the engine efficiency and the current system state, e.g., the data placement at query start time.

More importantly, CPU-only and GPU-only engines underutilize the hardware as they leave idle resources: the black bars represent the expected throughput of an ideal CPU-GPU integration. We observe that there is a 20%-80% throughput waste just due to idle resources.

To leverage the processing capabilities of such heterogeneous servers, the analytical query engine must 1) be flexible to *scale* across the diverse hardware resources and 2) *efficiently* use each of the underlying hardware devices, but without requiring manual optimization of the full stack for each target architecture. A key ingredient to achieving the aforementioned flexibility and efficiency is using accelerator-level parallelism [38] and customizable operators. For accelerator-level parallelism, multi-device execution must be decoupled from the internal device characteristics to enable arbitrary combinations of accelerators; for portable but efficient pipelines to be realized, the engine must abolish static hardware-dependent implementations in favor of dynamic pipeline specialization to the underlying hardware.

1.1 Motivation

Existing analytical engine designs rely on specific, system-wide available features to achieve efficient execution on CPUs. As a result, 1) their design inherently fails to exploit the available accelerator-level parallelism, and 2) the fate of analytical performance improvements depends on continuous CPU improvements. This section discusses how these two artifacts generalize to the heterogeneous hardware case.

Specializing for a single type of processor, e.g., CPUs, seems the most straightforward approach for analytical engines, especially given the CPU ubiquity. However, this creates three main problems for analytical engines. First, there is up to 20%-80% performance penalty due to idle resources, as seen in Figure 1.1. Second, most of the recent hardware improvements are on specialized hardware, threatening a stagnation of the analytical engine performance, despite the continuous increase in data processing needs. Third, specialization to a single processing unit or feature relies on the survivability of a specific technology, risking invalidating today's specialization attempts if hardware vendors abandon this feature. In summary, analytical engines that rely on the features of a single processor risk underutilizing the available hardware resources, seeing limited performance improvements compared to the increased demand for analytics and observing technological lock-in.

Aiming for universal portability is the next straightforward alternative for analytical engines. In this case, the engine avoids specific hardware features and operates under minimal assumptions about hardware features. However, the pessimistic viewpoint toward hardware characteristics leaves significant performance unexploited. First, in contrast to CPU-only hardware-oblivious approaches, specialized hardware, like GPUs, imposes a higher penalty for deviating from its intended use or not considering its peculiarities. Second, such oblivious approaches have to rely on a minimal set of hardware characteristics: if, for example, one device does not support system-wide cache coherence or shared memory, then the entire engine has to operate without it, even if some of the available hardware devices support it. Third, the set of hardware characteristics supported by all the devices shrinks as the engine targets more types of processors. In summary, analytical engines that aim for universal portability observe diminishing returns as they have to abandon optimizations to support more accelerator types.

This thesis investigates the impact of hardware heterogeneity on analytical engines through the use case of GPU acceleration. GPUs provide an architectural platform that is relatively generic and ubiquitous but still has significant differences from the traditional CPU architectures. These differences range from microarchitectural and memory hierarchy differences to how they connect with the rest of the server processors, making it an ideal testbed. In the rest of this thesis, we use the terms device/accelerator interchangeably and to mean "either a CPU or a GPU", except if otherwise stated. This naming conversion aligns with our approach of treating heterogeneous devices equally for inter-device execution, independently of whether they are CPUs or GPUs.

1.2 Pitfalls of Hardware Heterogeneity

Databases are typically optimized for specific hardware – most commonly for the CPU – and make static assumptions about the system architecture, such as the existence of specific hardware features, the collocation of the fast memory tier with the processing units, and even the

meaning of data locality. However, the presence of heterogeneous hardware complicates the hardware behavior of modern servers, and existing static designs threaten to leave significant hardware resources underutilized.

Parallelization over heterogeneous CPU-GPU hardware. Modern server hardware is increasingly heterogeneous as hardware accelerators, such as GPUs, are used together with multicore CPUs to meet the computational demands of modern data-intensive workloads. Unfortunately, query parallelization techniques used by analytical database engines are designed for homogeneous multicore servers, where query plans are parallelized across CPUs to process data stored in cache-coherent shared memory. Thus, these techniques are unable to fully exploit accelerator-level parallelism available across heterogeneous devices, where one needs to combine the task-parallelism of CPUs and data-parallelism of GPUs for processing data stored in a deep, non-cache-coherent, disaggregated memory hierarchy with widely varying access latencies and bandwidth.

Hardware-consciousness across heterogeneous CPU-GPU hardware. Existing analytical engines are highly optimized for the CPU architecture to provide efficient and fast analytics: hardware-conscious operators use advanced CPU features, like vectorization, and are specialized for the CPU cache hierarchy and task-parallel architecture. However, GPUs come with a different programming model as well as different microarchitectural features. From their single instruction multiple threads execution model to the cache hierarchy, and their in-order instruction processing, GPUs fundamentally differ from CPUs in a variety of performance-critical features that are core to many hardware-conscious algorithms and state-of-the-art execution models. Furthermore, even hardware-oblivious analytical engine proposals rely on advanced CPU features absent in GPUs, such as data prefetching, big per-thread caches, and out-of-order execution. As a result, existing analytical engine solutions introduce a sharp tradeoff between portability and performance: either reoptimize and manually specialize the analytical engines for each device type or waste significant analytical performance relying on suboptimal but generic operations.

Data access on heterogeneous CPU-GPU hardware. The shared-memory, cache-coherent multi-socket CPU architecture has been a cornerstone for efficient analytics and their parallelization. However, diverse accelerators introduce additional processor boundaries. The devices are interconnected with a variety of special- or general-purpose interconnects. While these interconnects have seen significant bandwidth improvements over recent years, they are still orders of magnitude slower than accessing device-local memory, amplifying the NUMA effects. Furthermore, the device heterogeneity has, counterintuitively, changed a fundamental assumption on the importance of sending data over interconnects: traditionally, the processor homogeneity has provided 1) little reason for crossing slow interconnects without a functional need and 2) little processing gain, as the processor on the other side had similar capabilities. However, the advent of GPUs and their on-chip high-bandwidth memory (HBM) creates a different landscape: crossing a slow connection can provide significant analytical throughput

gains due to the processor on the other side. As a result, existing NUMA-aware solutions designed with the implicit assumption that crossing a slow connection should be avoided are now challenged by the new architecture that shifts the trade-off in a per-query case.

Disaggregation on heterogeneous CPU-GPU hardware. Current analytical engines scale out of a single machine as a necessity originating from limited single-server resources. Furthermore, scaling out has a performance penalty: whenever significant data-shuffling is required, the low network bandwidth and the relatively similar computational resources on the other side of the network highly penalize small-scale clusters compared to scale-up solutions. As a result, analytical engines face an uncanny valley where small numbers of machines could result in worse performance than scale-up alternatives. GPU-acceleration, however, combined with high-bandwidth networking solutions, fundamentally changes the prior assumptions about the networking overheads and the importance of data local computations as 1) the bandwidth of accessing remote data and processors is close to the intra-server bandwidth, and 2) significant analytical power is available remotely. However, existing designs for scale-out analytical engines fundamentally rely on servers as the unit of operation, providing poor scalability and load-balancing capabilities in the case of internally heterogeneous CPU-GPU servers.

1.3 Thesis Statement and Contributions

Hardware heterogeneity significantly limits the efficiency and scalability of existing analytical engine solutions. Further, as hardware improvements shift from general-purpose advances to hardware specialization, existing analytical query processing techniques see reduced benefits. This thesis redesigns the analytical query execution engine so that execution across multiple, heterogeneous devices can be efficiently composed and orchestrated, with minimal effort for porting operator code across devices. The end goal is to enable efficient and scalable analytics by harnessing the available accelerator-level parallelism while maintaining the operator specification device-independent.

Thesis Statement

As hardware heterogeneity is increased, the trade-off between portability and performance of analytical query engines becomes more acute. Monolithic execution models are unable to fully exploit compute and memory devices. We can achieve both portability and performance by decoupling the execution model from device-specific characteristics while hiding the modularity and inter-connectivity overhead.

1.3.1 The End Goal: Efficient and Scalable Analytics on CPU-GPU Hardware

Composability and orchestration of query execution on CPU-GPU hardware. While diverse hardware architectures bring new features into a hardware platform, they also (i) reduce the set of features supported by all the system processors and (ii) limit device interoperability. However, this clashes with existing parallelization techniques that rely on advanced CPU features, such as techniques that depend on efficient system-wide atomic operations, shared memory, or independent task execution. We present a parallelization framework that encapsulates hardware heterogeneity to enable efficient orchestration of the available CPU-GPU hardware resources while at the same time allowing subparts of the execution to use advanced processor-specific capabilities.

Portability and customization of query execution to underlying CPU-GPU hardware. We present a system design that bridges the gap between operator portability and operator efficiency. We leverage the hardware and pipeline boundaries to enable the analytical engine to customize execution pipelines for each targeted microarchitecture independently. Furthermore, we provide operator portability by delegating operator specialization to device-specific backends and leveraging code generation to erase this modularity cost. As a result, we provide an intra-device execution model tailored to each device while minimizing the necessary manual operator specialization.

Data volume reduction and device cooperation on CPU-GPU hardware. While the hardware heterogeneity provides an adaptive execution environment where the workload is distributed across the devices depending on the query requirements, the interconnectivity of such devices limits the actual performance improvements and offloading flexibility. We provide a solution that efficiently uses the available interconnects by combining the byte-addressability provided by inter-device interconnects and the pre-filtering capabilities of near-data processing power. As a result, we build analytical engines that operate on datasets well beyond the GPU-memory capacity, with efficient bandwidth utilization, despite such architectures' highly NUMA setup.

Scalability and hierarchical composability on disaggregated CPU-GPU hardware. Analytical engines rely on scaling out to overcome single-server deployments' limited memory capacity and analytical throughput. But, GPU acceleration challenges existing CPU scale-out solutions. Specifically, scalable analytics on CPU-GPU clusters require efficient coordination and direct point-to-point data transfers to exploit the disaggregated hardware. However, the server boundaries introduced by shared-nothing multi-server architectures conflict with the coordination and direct transfer requirements of such intra-server asymmetric architectures. We introduce a hierarchically scalable analytical engine design that reconciles distributed analytics with hardware accelerators by decomposing the orchestration decisions into a multi-step, throughput-optimized process. Lastly, we show how this architecture repurposes CPUs into a near-data processor in the presence of GPUs and high-bandwidth, RDMA-enabled NICs.

1.3.2 Thesis Roadmap

The rest of the thesis is organized as follows:

- Chapter 2 introduces the necessary background.
- Chapter 3 proposes a decomposition of the analytical engine design space that drives the rest of the thesis.
- Chapter 4 proposes a parallelization framework that encapsulates the hardware heterogeneity to enable composable scale-up execution and CPU-GPU orchestration.
- Chapter 5 introduces an analytical engine design that optimizes intra-device query execution by providing portable, specialized operators coupled with an efficient intra-device execution model.
- Chapter 6 presents an optimization technique that reduces the inter-device data transfer volume. Specifically, it exploits byte-addressability and in-CPU pre-filtering opportunities to avoid wasteful transfer over the interconnect.
- Chapter 7 investigates the impact of shared-nothing architectures and high-bandwidth networking on GPU-accelerated analytics. Specifically, it shows how high-bandwidth networking distorts the traditional data locality definition and its effect on the role of CPUs and GPUs in data analytics. Then, it proposes a new analytical engine design that achieves scalable query execution and hierarchical composability by reconciling inter-server device orchestration.
- Chapter 8 concludes the thesis and presents future directions.

2 Background

In this chapter, we overview the typical hardware setup of today's heterogeneous compute servers and summarize related work on parallelizing query execution, hardware-conscious operators to set the context for our work.

2.1 Heterogeneity in Modern Servers

Modern servers incorporate numerous accelerators – typically multiple GPUs, connected to each CPU socket via a PCIe, NVLink, or similar interconnect. Servers can also increase the number of available devices per socket using PCIe/NVLink switches. However, if the server relies on switches to connect multiple GPUs to a CPU socket, the per-switch GPUs have to share the PCIe bandwidth whenever they concurrently trigger PCIe traffic.

As a result, besides hardware heterogeneity, modern servers must cope with non-uniform memory access topologies. CPUs experience additional memory access latency when accessing the memory of another socket – a phenomenon dubbed NUMA (non-uniform memory access) [50]. Introducing GPUs exacerbates NUMA effects. When GPUs access CPU memory, they transfer data through the aforementioned interconnects, whose bandwidth is limited compared to CPU's local DRAM bandwidth (a few 100s of GBps) and to the bandwidth of a GPU's device memory (up to a few TBps). Specifically, a 16-lane PCIe 3.0 connection – typical for CPU-GPU connectivity – offers ~12.8GBps data bandwidth, while PCIe 4.0 used by more recent GPUs offer ~25GBps. IBM Power 8 and 9 CPUs support CPU-GPU communication through NVLink – an NVIDIA-specific interconnect that achieves up to 75 GBps CPU to GPU communication.

2.2 Parallel Query Execution on CPUs

Volcano and Exchange. When a query reaches a database system, it is processed by a query planner / optimizer, resulting in an algebraic query execution plan. This plan, is generally expressed in the form of a tree, where each node is an operator, and it was traditionally interpreted using the Volcano iterator model [30]. Every operator of the plan exposes a general API, consisting of *open()*, *next()* and *close()* functions. When an operator's *next()* method is called, a request for a new tuple is sent to the operator's children.

The Exchange operator introduced in Volcano has been the standard approach for parallelizing a query plan. The Exchange operator encapsulates all three different types of parallelism (horizontal, vertical, and bushy) by exposing the same (iterator) interface as other operators in an interpreted query plan. Inserting an Exchange operator in a query plan splits it into two parts, with the sub-plan above the Exchange becoming the consumer and the sub-plan below being its producer. The Exchange operates as an asynchronous queue between the producer and the consumer. The producer inserts its results into the queue, while the consumer removes them and processes them. Both the producer and the consumer are not aware of the queue, and they interface with the Exchange operator in the same way as with any other operator. As both the producer and the consumer can execute in parallel on different processors, Exchange enables vertical parallelism. In addition, the Exchange controls the degree of parallelism of consumers and producers by spawning multiple instances of them and routing intermediary results between the different instances, introducing this way horizontal parallelism. Producers' results are either routed based on a policy to exactly one consumer or broadcasted to all of them. Lastly, introducing Exchange operators on both sides of a join creates bushy parallelism: each input of the join runs concurrently.

JIT compilation and exposing parallelism. Albeit Exchange makes it possible to parallelize sequential, single-threaded operator implementations without any code changes; it has certain drawbacks that limit its applicability as the mechanism of choice for parallelizing query execution in the modern in-memory data processing context. Interpreted query execution penalizes performance as the *next()* function is called for every tuple, resulting in frequent branch misprediction and poor code locality [46, 72].

State-of-the-art in-memory analytical engines avoid such interpretation overhead by eschewing interpreted execution in favor of JIT compilation [45, 46, 72]. JIT-based in-memory database engines split the query plan into non-blocking pipelines and use a compilation framework to translate a sequence of operators into straight-line code that loops over data one tuple at a time. Thus, these systems enable register-pipelining: a collection of non-blocking operators can be applied in one shot to a tuple stored in CPU registers. CPU-based JIT compilation techniques also do not use the traditional Exchange-based parallelism where operators, other than Exchange, are essentially sequential in nature. Instead, the typical approach, as exemplified by HyPer's morsel-driven parallelism [51], compiles and generates parallelization-

aware operators using atomic instructions in generated code for synchronizing access to shared data structures. Such code can then be executed in a task-parallel manner across multiple CPUs by using a thread pool.

CPU parallelism in heterogeneous servers. Unfortunately, the aforementioned approaches for parallelism can not be used in modern heterogeneous servers to parallelize queries across CPUs and GPUs. The traditional Exchange was not designed to work in heterogeneous parallel processing environments: pipelining across different processors requires Exchange's asynchronous queues of the operator to be placed such that they can be efficiently accessed by all processors. Morsel-based JIT-based Morsel parallelization requires system-wide, efficient cache coherence as it relies on atomic operations for synchronizing access to producer–consumer queues, or other shared data structures like joins' hash tables during the build phase. While these assumptions hold in homogeneous multicore CPU servers, in the general case, they are invalidated in heterogeneous servers with CPUs and GPUs due to a lack of global cache coherence.

Furthermore, unlike CPUs, executing an operator on the GPU requires moving the input data to GPU memory, launching a kernel to process the input, and potentially moving out the output data. As moving data is an expensive operation, it is important to move enough data so that the benefit gained from processing data on the GPU outweighs the data movement cost. Similarly, since kernel launches are expensive and slow, it is also important to minimize the number of kernel launches. Unlike a CPU-based JIT compiler, which has to generate executable code for just one processor, a GPU-based JIT compiler should generate both kernels that are executed on the GPU, and CPU code that invokes these GPU kernels. Thus, modern servers with heterogeneous CPU–GPU parallelism require rethinking traditional query execution and compilation strategies.

2.3 Parallel Query Execution on GPUs

Operator-at-a-time execution. The inability of the commodity CPU to achieve unconditional scalability has led to numerous research and industrial efforts that utilize GPU co-processors for the acceleration of analytical database workloads [16, 35, 37, 44, 73, 76, 102]. Most GPU-powered DBMS operate as follows: The DBMS expresses the query plan as a sequence of (micro-)operators [16, 35, 37, 44, 102], and then translates each operator into a *kernel* – a data-parallel function. The DBMS then executes the kernels, one after the other, on a GPU, fully materializing intermediate results in order to provide them as input to the next kernel.

Initially, such "operator-at-a-time" GPU DBMS [16, 37] required every kernel to have its input available at operator invocation time, and thus complicated the overlap of (GPU) computation and (CPU-to-GPU) data transfer. Subsequent systems thus introduce the following optimizations. First, they overlap data transfer with computation to mask the data transfer cost as much as possible. For example, GPUDB [102] uses *Unified Virtual Addressing (UVA)*, an NVIDIA CUDA feature that allows a GPU to directly access memory of the CPU side, while

Rui and Tu [89] use CUDA memory copies and CUDA streams for a similar purpose. Stehle et al. [96] propose CPU-GPU co-processing to accelerate sorting tasks; their approach parallelizes the production of sorted runs and interleaves it with data transfers to and from the GPU. Sioulas et al. [94] propose a CPU-GPU co-processing radix join that uses the high CPU DRAM bandwidth to apply an initial partitioning before transferring the inputs to the GPU for the join. This partitioning step allows bigger-than-GPU-memory tables to be broken down to co-partitions that fit in GPU memory and thus perform the join with a single pass over the PCIe. Second, modern GPU DBMS have followed the MonetDB/X100 [15] paradigm to reduce materialization overheads [72] between GPU kernel invocations; every kernel operates over a subset (i.e., a vector) of the input and produces a vector as its output. Intermediate vectors fit in GPU memory for the next kernel to read, and the DBMS avoids unnecessary data transfers of intermediate results to the CPU. Still, result materialization – even when amortized using cache-resident vectors between kernel invocations – is wasteful in terms of memory bandwidth [27]; the GPU DBMS has to flush GPU registers and shared memory between kernel invocations, thus hurting locality. In addition, the vector-at-a-time paradigm requires multiple passes, further wasting (GPU) memory bandwidth.

Pipelined GPU execution. An alternative to vector-at-a-time processing is performing as much work as possible over data that already resides in GPU registers / shared memory. Such *pipelined* query execution typically reduces the number of kernels per query plan. GPL [73] pipelines operators by having each one of them running on a separate kernel and having the kernels communicate and transfer data through *OpenCL 2.0 pipes* [31]. HAWK [19] is a query compiler that generates OpenCL which can execute on a variety of parallel processors, such as CPUs and GPUs, but on only one of these platforms at a time. HorseCQ [27] departs from the use of data-parallel algorithms for operations such as reductions, and instead implements pipelined versions of said algorithms using GPU atomic instructions. Kernel Weaver [101] is a compiler that automatically tries to fuse multiple relational operations together into a single kernel, in order to i) reduce data movement and ii) enable additional compiler optimizations over the fused operators. Finally, MapD [2] ports the paradigm of CPU-based query compilation [63] in the context of GPU DBMS. MapD uses the LLVM compiler infrastructure to generate the code for its kernels just in time; the kernels contain code which is specialized for the current query, and try to minimize the amount of intermediate results per query.

GPU engines on heterogeneous servers. Most GPU-powered DBMS adopt one point in the design spectrum and make one or more of the following simplifying assumptions: First, they rely on the input dataset being GPU-resident or copartitioned to avoid the PCIe transfer overhead for input and intermediate data [2]. Second, they support query execution on a single GPU instead of multiple ones [35]. Third, their mechanisms for parallelizing queries are strictly GPU-tailored. This leaves a substantial amount of CPU-based processing capacity underutilized when used on heterogeneous servers, and misses out on potential co-processing opportunities where a query can be parallelized across CPUs and GPUs simultaneously. The few engines that support executing queries on both CPUs and GPUs [44] rely on wasteful full materialization. Finally, in HAPE [24] we envision specializing device-oblivious operators to

each device to achieve efficient, heterogeneity-aware execution, via code generation. HAPE assumes an abstraction that encapsulates inter-device execution and provides only an abstract system blueprint. In HetExchange we materialize these abstractions. We discuss both in the following sections.

2.4 Hardware-conscious Operators

While hardware-oblivious algorithms simplify the optimization process and the execution over heterogeneous hardware, tuning algorithms for the underlying hardware can produce significant performance benefits. For modern CPUs, most previous studies take three architectural characteristics into account: cache hierarchy, TLBs and SIMD instructions. These dimensions are analyzed in conjunction with the available memory bandwidth and latency.

Prior work has introduced hardware-conscious variants of several operators. including scan-like operators, sort-based operations and index scans [39, 77, 105]. As a heavyweight operator, the join has been studied and tuned extensively for modern CPUs, resulting in multiple variants of the radix hash-join [8, 9, 14, 90, 92]. Specifically, Shatdal et al. [92] proposed a cache-conscious variant that introduces a partitioning step. The two input tables are co-partitioned such that for each partition pair, the hash table fits in cache. Then, all hash-table accesses during the probing phase are in cache, and cache misses are averted. Boncz et al. [14] observed that for the high number of output partitions, the performance is impacted by TLB misses. As a solution, they advocate the use of multiple partitioning passes, each producing a smaller number of partitions, reducing TLB misses at the expense of extra passes over the input. Schuh et al. [90] argue that the common denominator is that these works try to minimize the effects of random memory accesses by minimizing cache and TLB misses. Still, Blanas et al. [13] argue in favor of hardware-oblivious hash-joins as they require less parameter tuning and can outperform hardware-conscious implementations in some scenarios.

In contrast to CPUs, modern GPUs have a significantly different microarchitecture, including all three aforementioned characteristics. First of all, GPUs depart from the linear memory hierarchy of CPUs and adopt a fatter cache hierarchy (Figure 2.1), with a hardware-managed L1-like cache, called *shared memory*, which is a software-managed scratchpad, and other more specialized caches, like a constant cache. In addition, GPUs target different workloads and thus size their caches and TLBs differently to CPUs. Karnagel et al. [43] experimentally showed that GPU TLBs have 2MB pages to support the high number of threads and pack more addressable space per TLB entry. Finally, in the GPU SIMT model, each GPU thread has an independent register file, but, in contrast with the SIMD model, thread divergence is handled in hardware. As for CPUs, hardware-conscious algorithms that consider the GPU hardware improve performance. Karnagel et al. [43] take into consideration the TLBs to improve hash-based group-by operations, while partitioned hash-join [42, 89] implementations use shared memory to store histograms and per-partition hash-tables.

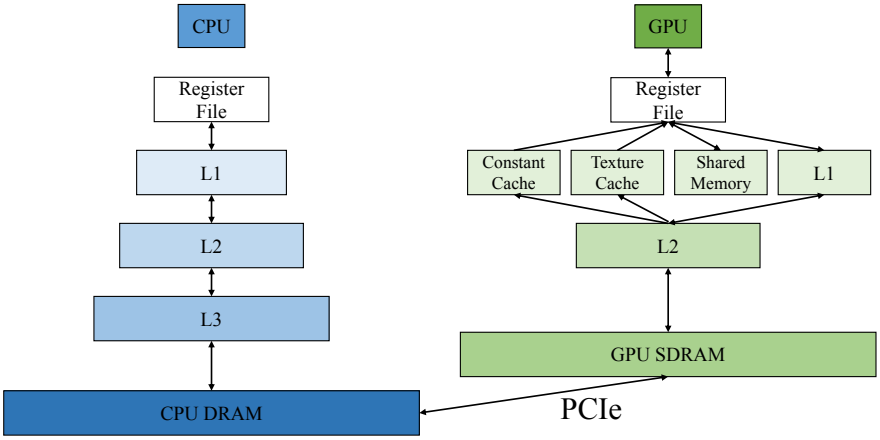


Figure 2.1: CPU and GPU data cache hierarchy.

A limiting factor for GPU algorithms is GPU memory size. Prior work makes simplifying assumptions about the types of workloads handled; [89] only addresses the case that at least one of the tables fits in GPU memory. Kaldewey et al. [42] use Unified Virtual Addressing (UVA) to join arbitrarily large data by accessing data over the interconnect. Still, interconnect bandwidth is an order of magnitude slower than GPU memory bandwidth, greatly impacting multi-pass algorithms such as radix joins.

Inter-device co-processing can reduce unnecessary interconnect traffic. Stehle and Jacobsen [96] present an efficient sorting algorithm that consists of two steps: generating sorted runs in GPU and merging them in CPU. Merging in the CPU side allows for a single pass, per direction, over the scarcest resource, the interconnect. Sioulas et al. [94] exploited the CPU memory bandwidth to partition the inputs of a partition-based hash-join before sending them to the GPU. The initial partitioning breaks down big relations into partitions that fit in the GPU memory, while its small fan-out allows for high throughput on the CPU side. On the GPU side, they further partition the inputs to fit the final partitions in the scratchpad and minimize the effect of random accesses.

3 Overview: Design Space of a Hybrid CPU-GPU Execution Engine

Existing analytical engines have been designed and optimized for specific compute units, usually the CPU. Yet, as the hardware becomes more diverse, the well-known trade-off between portability across microarchitectures (hardware-obliviousness) and performance (hardware-consciousness) becomes more acute. Analytical engines for heterogeneous hardware need to: i) execute efficiently on multiple types of devices, which requires taking into consideration both the programming model of each device as well as its microarchitecture, ii) be portable and modular to allow the reuse of subcomponents between heterogeneous devices without introducing overheads in the critical path of execution, and iii) orchestrate execution across multiple heterogeneous devices, despite the possibly limited interoperability across the devices. To achieve that, we decompose the design space into three independent axes to achieve portability and independence while optimizing for each dimension.

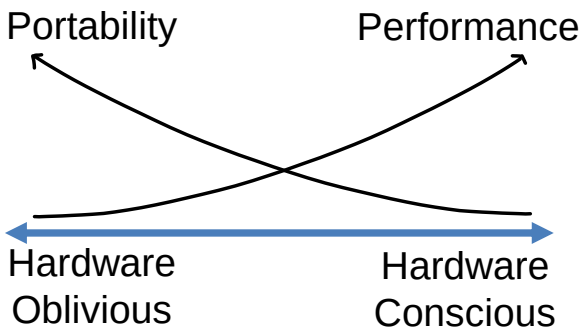


Figure 3.1: Conceptual trade-off between portability and performance.

Decoupling the design space. Our proposal embraces heterogeneity by separating analytical query execution into two logical layers: intra- and inter-device execution. This separation aligns the optimization requirements with the execution concerns. Optimizing based on a specific microarchitectural feature can improve performance but also reduce the optimized code's applicability to architectures with the corresponding feature. On the one hand, tuning intra-device execution can improve efficiency; however, it provides little gain for inter-device operations. On the other hand, microarchitectural obliviousness improves the scalability

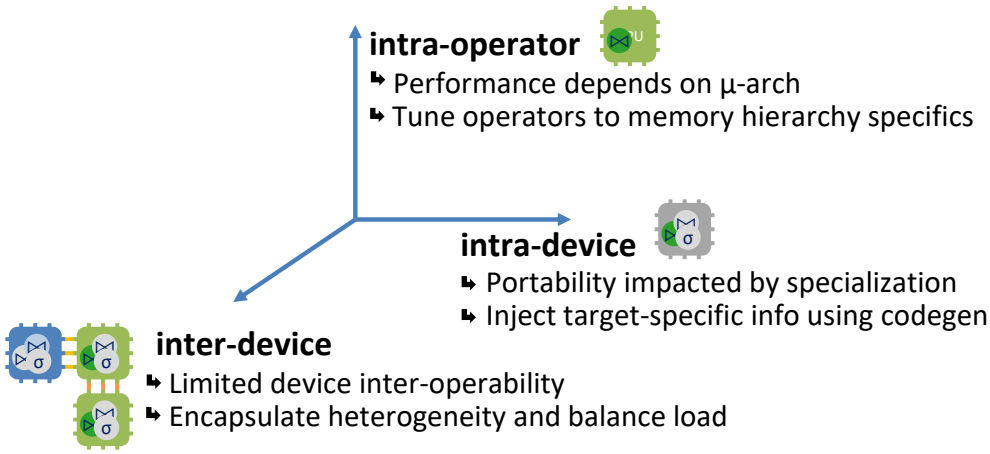


Figure 3.2: Design axes for CPU-GPU OLAP engines.

of inter-device execution by enabling it to target more devices; however, it penalizes intra-device efficiency. By decoupling intra- and inter-device execution, we can treat each layer independently – aligning our design with the hardware to get the best of both worlds. As a result, the intra-device layer handles each device separately and in isolation to enable localized hardware awareness. The inter-device layer virtualizes devices into a unified generic model, embracing hardware obliviousness to facilitate portability.

Intra-device execution still has to consider the microarchitectural differences, but the above separation allows reducing the problem to optimizing query execution in each device separately instead of concurrently optimizing for all devices. To achieve portability across devices, we further decompose intra-device execution into two subproblems: intra-operator execution and inter-operator. The former focuses on optimizing the relational operators to each microarchitecture, while the latter optimizes for portability across devices by modularizing the intra-operator solutions and reducing overheads introduced by this modularization.

Inter-device execution is concerned with execution at the granularity of devices. It focuses on inter-device pipelining and concurrent execution based on the system topology, NUMA nodes, and each device's current load. Device types are hidden from inter-device components, except for the components that are responsible for synchronizing between heterogeneous devices. In addition, by encapsulating the traits of heterogeneous execution in operators, we allow the query optimizer to argue about data and computation transfers. Lastly, to load balance between devices, inter-device execution can ignore most of the microarchitectural characteristics of the heterogeneous devices and, instead, rely on macro-characteristics of query execution, such as task throughput.

Efficient and scalable inter-device execution has three requirements by itself: First, analytical engines need to parallelize execution across the heterogeneous devices of a single server. Second, to avoid the capacity limitations of a single server, analytical engines for heterogeneous hardware need to scale to multiple machines and orchestrate devices across the server boundaries without introducing wasteful data transfers. Third, data movement across devices is a key factor in offloading to the most appropriate devices; however, it has significant bandwidth requirements, requiring careful consideration of the transfer policies. To address all three requirements, we start with presenting a parallelization framework (Chapter 4) for single-server execution that we then extend to multiple servers (Chapter 7), and independently optimize its data transfers (Chapter 6).

Intra-device execution is concerned with increasing the efficiency of pipelines running in each device as well as the portability of the operators between different devices (Section 5.1). To allow re-using relational operators across devices, the operators are modularized with respect to their memory mappings, how they use the memory hierarchy of each type of device, and common primitives, such as how to perform thread synchronization or atomic updates. Memory mappings depend on both the operators and the target device, while the common primitives depend on the device.

While this modularization increases the portability of operators, it introduces overheads due to the portability in the critical path and requires support for virtual function calls. To avoid these overheads, we employ just-in-time code generation that injects, through the memory mappings and the primitives, device specific-knowledge to portable operators. The generated code is specialized to the target device, and the modularization cost is reduced.

Intra-operator execution. Traditionally, hardware-conscious operators take into consideration the microarchitecture of the targeted compute units, and thus they can not be reused across different devices (Section 5.2). We argue that many hardware-conscious operators can be decomposed into an algorithmic part that focuses on how to mitigate a specific overhead (e.g., partitioning in radix-based join to mitigate overheads due to random accesses) and a resource-oriented part that assigns specific parts of the memory hierarchy to specific data structures (e.g., using L1 for storing intermediate partitions). The algorithmic part can be reused across devices, while the memory mappings are device-specific. We take advantage of this differentiation to increase the portability of hardware-conscious operators.

4 Inter-device: Encapsulating Heterogeneous CPU-GPU Parallelism

This chapter presents *HetExchange*–a framework that encapsulates the heterogeneous parallelism in modern servers to enable analytical query execution across multiple CPUs and GPUs. Similar to traditional Exchange [30], HetExchange encapsulates parallelism and provides a uniform interface to connect producers and consumers in a pipelined plan together with the memory infrastructure. However, unlike traditional Exchange, which dealt only with homogeneous parallelism across CPUs, HetExchange encapsulates heterogeneous parallelism across CPUs and GPUs. Additionally, unlike Exchange, which connects individual operators in an interpreted execution environment, HetExchange connects subpipelines in a JIT-compiled execution environment. Thus, HetExchange provides a framework that can be used by JIT-compiled engines to parallelize sequential, single-threaded code on multiple CPUs and single-GPU kernels across multiple GPUs, or even a single query plan across both CPUs and GPUs in a coprocessing fashion. In doing so, HetExchange shares the benefits of the two popular parallelization techniques without the disadvantages. By encapsulating heterogeneity, HetExchange proposes a single abstraction that can be used to encapsulate heterogeneous parallelism without making assumptions about hardware characteristics, like the availability of globally cache-coherent shared memory.

Contributions. The contributions of this work are the following:

- We introduce HetExchange–a novel parallel query execution framework that encapsulates heterogeneous parallelism and enables query plan deployment across i) CPUs, ii) GPUs, and iii) a mix of CPUs and GPUs.

- We integrate HetExchange in Proteus and evaluate our prototype against state-of-the-art CPU and GPU engines; showing up to 1.5x and 5x speed-up, respectively, when restricted to the same compute units, while, by using all the available units, we achieve up to 5.1x and 11.4x speed-up and linear scalability.

Flow	Scope	Trait	Operators
Control	Delegation	Heterogeneous Parallelism	device crossing
	Routing	Homogeneous Parallelism	router
Data	Transfer	Data Locality	mem-move
	Granularity	Execution Granularity	pack

Table 4.1: Execution traits in a heterogeneous server

4.1 The HetExchange Framework

Fully utilizing the devices in heterogeneous servers requires exploiting both intra-device data parallelism offered by GPUs, inter-core task-parallelism offered by CPUs, and cross-device heterogeneous parallelism across multiple CPUs and GPUs. In addition, the engine must use the fast node-local memory available in CPUs and GPUs while working around the limitation of non-cache-coherent shared memory.

HetExchange redesigns the classical Exchange operator to parallelize pipelines on multicore CPUs, multiple GPUs, and across CPUs and GPUs. In a heterogeneous parallel query execution engine, execution has to be routed between different devices. Traditionally, analytical engines use the Exchange operator to perform such *control flow* routing between consumers and producers running on CPUs. On heterogeneous platforms, producers and consumers are not guaranteed to be of the same nature: they may be CPU cores, GPUs, or a mix of CPUs and GPUs. To enable heterogeneous control flow transfers, HetExchange uses two control flow operators: *device crossing* and *router* operators. In addition to control flow, an Exchange operator should also deal with *data flow*, to ensure that data is transferred between producers and consumers in a pipelined fashion. HetExchange enables cross-device data flow transfers via two operators, namely, *mem-move* and *pack*.

There is a one-to-one match between the proposed operators and traits that characterize part of query execution on a heterogeneous server, as the operators are converters of the traits. The device-crossing operator is responsible for pipelining/delegating work between different types of devices, while the router is responsible for load-balancing, scheduling based on affinities, and parallelizing across different streams of execution. The mem-move operator is responsible for bringing data locally to compute, if necessary, and the pack operator is responsible for converting between different execution granularities (i.e., block-at-a-time to tuple-at-a-time). In addition, by encapsulating the four traits, the proposed operators allow a traditional query optimizer to argue about them. To introduce the operators, we will use a running example of a scan-filter-aggregate query, in which we want to execute part of it on GPUs and part of it on CPUs (green and blue nodes, respectively).

Figure 4.1: Step by step introduction of HetExchange operators.

4.2 Control Flow Operators

HetExchange decomposes control transfers into two types: transitions between exactly one producer and one consumer of different types, and transitions between an arbitrary number of homogeneous producers and consumers. HetExchange uses two operators, a device-crossing and a router, to handle each type of transfer. This separation provides a modular division of labor across the two control flow operators. Transitions between arbitrary numbers of heterogeneous units involve a combination of the two operators.

Device-crossing operators enable pipelining across heterogeneous hardware. Except from these operators, all other operators are oblivious to hardware heterogeneity and always execute on a single device. More specifically, HetExchange uses two device-crossing operators for CPU-GPU co-processing, called *cpu2gpu* and *gpu2cpu*. Cpu2gpu copies the CPU context to the GPU and transfers control flow by launching a GPU kernel, while gpu2cpu transfers the GPU context to the CPU and starts a CPU task. In contrast with launching a GPU compute kernel from the CPU, GPU programming frameworks do not support launching CPU tasks in the middle of the execution, which prevents fully pipelined execution across devices. HetExchange implements this functionality by breaking the gpu2cpu operator into two parts, each one running on a different device. These parts communicate using an asynchronous queue. When a GPU kernel is ready to send a task to the CPU, the gpu2cpu operator inserts the task into the queue. On the CPU side, the second part of the operator receives and executes it.

Router operators encapsulate parallelism across multiple processors. As in the classical Exchange, for vertical parallelism, router operates as an asynchronous queue between a producer and a consumer. For horizontal parallelism, it instantiates multiple instances of the consumers and asynchronously routes tasks between consumers and producers. In contrast

to the traditional Exchange, a parallel query plan consisting of routers is a directed acyclic graph: the router may have multiple parents, each of them targeting different devices. Each router's parent and child are instantiated multiple times to achieve the necessary degree of parallelism in each device type. The router implements various routing policies: hash-based routing for use in hash joins, round-robin/range routing for partitioning inputs to multiple consumers, and union routing for merging inputs from multiple producers.

In contrast with the classical Exchange, router only operates on the control plane. A task refers to the target input data via a block handle. The router transfers the block handle from the producer to the consumer but not the actual data. When needed, the data flow operators handle the block creation and its transfer, as described in Section 4.3. This division of labor between the router and data-flow operators enables the router to connect producers and consumers without making assumptions about data location or accessibility.

While the router avoids data transfers by operating with block handles, in some cases, the router itself needs access to the tuple values. For example, a hash-based routing policy uses the hash value of input tuples to determine their consumers. Due to the heterogeneity of memory access in CPU–GPU servers, such data might not be directly accessible by the router, as can be the case for a router running on the CPU that attempts to access a GPU resident block of data. Thus, performing routing would force the router to either transfer the data to evaluate the routing policy, or operate on multiple device types to run locally with respect to the target block of data. None of the solutions is modular, as they would duplicate data movement in the router and data flow operators.

HetExchange uses an approach tailored to heterogeneous servers to handle such cases. Instead of having the router access tuples for determining policies, HetExchange pushes the policy mechanism down to the data flow operators (described in Section 4.3) that have access to the data. For example, in order to use a hash-based routing policy over blocks of tuples, we require each block to have only tuples with the same hash value. This is achieved by enforcing the data flow operator that produces these blocks to maintain this invariant during the creation of each block. Each block handle provided by the data flow operator is then forwarded to the router operator with the corresponding hash value. Thus, the hash-based routing policy can decide without having access to individual tuples.

Another difference between the router and classical Exchange is that the router does not perform broadcasts. Efficiently executing a broadcast depends on both the memory topology and the initial location of the data. For example, it may be possible to just share data between the targets or use some multi-cast capability of the interconnect. In addition, broadcasts are inherently data flow operations, as they duplicate data flow inside the plan. On the contrary, assigning the different flows to different execution streams is a control flow operation. Thus, broadcast, in the sense of data duplication, is left to the mem-move operator, described in Section 4.3. For this case, the mem-move operator produces as output one block handle per

broadcast target and a value, the target id. Then the router routes the block based on the target id, without caring about how the data were actually broadcasted. For the router, this is similar to a hash-based policy.

Encapsulating heterogeneous parallelism example. Combining the router with the device crossing operators creates all the necessary control flow manipulations to enable all three types of parallelism across multiple heterogeneous compute units. Device crossing operators are placed between heterogeneous producers and consumers to move execution across device types. Routers are placed at strategic points before device crossing operators to parallelize query plans. We use a reduction over the results of an equijoin of two filtered tables as a running example to illustrate how control-flow operators work. Figure 4.1(a) depicts a physical plan for such a query, generated for sequential CPU-only execution.

In the running example, placing three device crossing operators is enough to move the execution of the hash-join to the GPU. An example of such a plan is shown in Figure 4.1(b). A cpu2gpu operator is placed on the left to kick-start the execution of the left-hand scan and filtering pipeline on the GPU. This cpu2gpu operator transfers execution from the CPU to the GPU and as a result, feeds the hash-join build phase on the GPU. The scan and filter operations for the probe table are executed on the CPU in this example. As the filter selects some tuples, it forwards them to the cpu2gpu operator above it which then transfers it to the probe phase of the GPU join. Similarly, the cpu2gpu above the hash-join transfers the execution back to the CPU side for the final reduction.

Figure 4.1(c) extends 4.1(b) with router operators. The example uses five routers in order to parallelize the hash-join over all the CPUs and GPUs. In the left-hand side, the segmenter will split the input file into small block-shaped partitions, that are treated as normal blocks. Partitions' block handles will be propagated to the router, which instantiates the scan-filter-gpu2cpu consumer multiple times and routes partitions to consumers, while load-balancing.

Each of the GPU scans will read the partitions which are propagated to it by the router, via the cpu2gpu operator. The filter performs predicate evaluation and propagates passing tuples to its corresponding gpu2cpu operator, which in turn forwards them to the router. This router unions the results from the GPU – which are then filtered and distributed to its consumers. This router has two parents in the plan, one of them to execute the hash-join on GPUs and the other to execute it on CPUs. Each parent is instantiated multiple times, for example, the first one as many times as the number of available GPUs, and the second as many times as the number of available CPU cores. As the results are routed between the consumers, the join ends up running in a mix of CPUs and GPUs. After the joins, a local reduction happens in each device and the output of each local reduction is sent to the union-like router which gathers all of them into a single thread to produce a final global aggregation.

4.3 Data Flow Operators

As a heterogeneous server usually has multiple memory nodes, query execution has to deal with the accessibility of each operator to its input. For example, depending on the exact hardware of the server, GPU memory is not directly accessible through CPU load/store instructions and, in some cases, might not be accessible by other GPUs. While the control flow operators enable parallel and pipelined execution across multiple heterogeneous devices, none of them actually considers whether the input data are accessible by consumers.

Mem-move operator. The mem-move operator is responsible for moving data between node-local memory of producers and consumers. It receives a block handle from its child that contains information about the sources and targets for each data block that it must move. Using this information, the mem-move ensures that data have been transfered and are accessible before the data consumer is executed.

Mem-move encapsulates the logic to drive the transfers over the interconnects as well as to take decisions based on the topology and the initial location of the data. In case the data are already local to the consumer, it only forwards the block handle, without doing any data transfers. In situations where a CPU producer must be connected to a GPU consumer, or vice versa, it is responsible for launching the necessary DMA transfers over the PCIe to move data from CPU host memory to GPU device memory. As the mem-move abstracts away memory heterogeneity issues, all other operators can be data-location agnostic. Thus, other operators do not have to be programmed to perform explicit data transfers or data accessibility checks. Based on the information mem-move has regarding the data flow from the query plan, it automatically prefetches data to consumer's local memory before the consumer accesses them.

Memory transfers happen asynchronously to computation. Mem-move internally consists of two parts, one that resides on the producer and one that resides on the consumer. When the producer's part of mem-move receives a block handle from the producer, it schedules the transfer and returns back to the producer, to allow it to generate the next block. The consumer part of mem-move waits for transfers to complete. When a transfer completes, it pushes the block to the consumer. As a result, both the consumer and the producer execute asynchronously with respect to the memory transfer. Mem-move is also responsible for multi-casting. For certain operations, like a broadcast-based hash-join, it is common that copies of the same chunk of data should be sent to multiple consumers. Multi-casting is essentially a special case of data transfer and multiple interconnects support it. Thus, mem-move bears the responsibility of broadcasting and implementations can potentially exploit the capabilities of the underlying interconnects to do it efficiently.

Pack/unpack operators. Moving data is expensive and is often the bottleneck in GPU query processing. HetExchange amortizes the data transfer cost by executing transfers at block granularity instead of tuple granularity. However, block-at-time execution of operators on

the GPU is suboptimal due to materialization overhead compared to fusing operators into a few kernels using JIT compilation, and having each GPU thread perform tuple-at-a-time execution with register pipelining [27].

HetExchange uses the *pack* operators to encapsulate the difference between block-at-a-time data movement and tuple-at-a-time execution. The two basic operators of this set are *pack* and *unpack*. The pack operator groups tuples into a block and flushes it to the next operator whenever it fills up. The unpack operator takes a block of tuples as input and feeds them one tuple at a time to the next operator. HetExchange also uses the pack operator to create blocks with interesting properties. When used to pack/unpack data for a consumer that is a GPU operator, these operators ensure that the grouping of tuples enables different GPU threads to read data in a coalesced manner. When used to pack/unpack data for a hash join, the pack operator generates blocks whose tuples have the same hash value by maintaining one block per hash value, that is flushed to the next operator whenever it's full. As all the tuples in a block have the same hash value, consumer operators, like the router, can operate over the whole block, without accessing individual tuples.

Encapsulating heterogeneous memory access example. We extend the running example shown in Figure 4.1(c) by placing mem-move operators in order to move the data to the point of their consumption. Figure 4.1(d) shows a plan that is distributing the data based on their hash values for the join. In the left-hand side of the plan, a mem-move is placed after the router responsible for distributing the input segments. As input segments are pushed from the segmenter to the router and routed to the different GPUs, the mem-move after the router will make sure that the data are accessible by the target GPU. For example, if a block is routed to a GPU but residents on another one or on the CPU, mem-move will transfer it to this GPU. If it is already on the destination node, it will propagate the block handle without transferring data.

Figure 4.1(e) extends 4.1(d) by adding pack/unpack operators. Notably, the scan operators of Figure 4.1(d) are replaced by unpack operators in Figure 4.1(d) to highlight the fact that each unpack operator processes multiple blocks of input. In addition, as the data shuffling between the filtering and join phases is in blocks, unpack operators are placed in each device to translate between blocks and tuples. For the same reason, both filters are followed by packing operators.

The query plan uses two hash-packs to hash-partition the inputs of the join. Each time the hash-pack outputs a block, it also outputs the hash value of the block elements. In the left-hand side of the plan, the hash-pack pushes block handles and the hash-value to the gpu2cpu operator, which propagates both of them to its CPU side and then to the router, which routes blocks based on the hash-value.

In this specific plan, all the consumers start with a mem-move. Thus, when a mem-move receives a block handle, it transfers the block data to the target device, if necessary. Then, mem-move forwards the handle to the cpu2gpu operator. Cpu2gpu will launch a kernel to consume this block, which will start by distributing and scanning the block to the different GPU threads using the unpack.

4.4 Evaluation

Experimental Setup. We implement HetExchange in Proteus [45] and compare it against state-of-the art commercial analytical engines DBMS C and DBMS G for CPU and for GPU execution. DBMS C is a columnar database that uses SIMD vector-at-a-time execution, similar to MonetDB/X100 [15], and supports multi-CPU execution. DBMS G uses JIT code generation, operates over columnar data and supports multi-GPU execution. We use various configurations of Proteus (i.e., CPU-only, GPU-only, and hybrid execution) to showcase its versatility and its ability to execute queries efficiently regardless of where data is originally located – i.e., the CPU or the GPU memory. We warm up each system by executing multiple queries before the measurements. The experiments run on a two Intel Xeon E5-2650L v3 CPU machine, running at 1.8GHz with 12 physical cores per socket. The server has 256GB of DRAM occupying 8 out of the 12 memory channels, with 128GB of DRAM local to each CPU socket. Each CPU socket has one NVIDIA GeForce GTX 1080 GPU attached via a dedicated PCIe 3.0 x16 connection and each GPU has 8GB of local memory. We measure a maximum bandwidth of ~12GBps on each interconnect, on an idle server.

We compare the behavior of Proteus against DBMS C and DBMS G, using the SSB decision support benchmark, and scaling it up to factor 1000. We examine scenarios in which data is either CPU- or GPU-resident, as well as configurations of Proteus that allow it to use i) only GPU(s), ii) only CPUs, or iii) both. We use the Star Schema Benchmark [71] to compare three configurations of Proteus against DBMS G and DBMS C. Proteus GPU and DBMS G use the two GPUs available, Proteus CPU and DBMS C use the two CPU sockets and Proteus Hybrid uses both the GPUs and the CPU sockets. For Proteus Hybrid, we select plans that parallelize all the relational operators across all the available CPUs and GPUs. While it is possible to pin parts of the plan to specific processors, we leave optimizer-driven plan generation with different parts of a plan running on different processor sets as future work.

Methodology. We use scale factor SF=1000 for SSB [71], which generates ~600GB of data. For all the queries the working set exceeds the aggregate device memory of the two GPUs. So, both Proteus GPU and DBMS G transfer the working set from CPU to GPU memory during query execution. Thus, their throughput is upper bounded by the PCIe bandwidth (~24GBps), shown as a dotted line in Figure 4.2. Proteus Hybrid load balances the work between GPUs and CPUs and thus transfers only part of the dataset to the GPUs. While Proteus supports datasets that are partially preloaded in GPU memory, for this experiment we disable this functionality to simulate worst-case transfer times. Figure 4.2 plots the results.

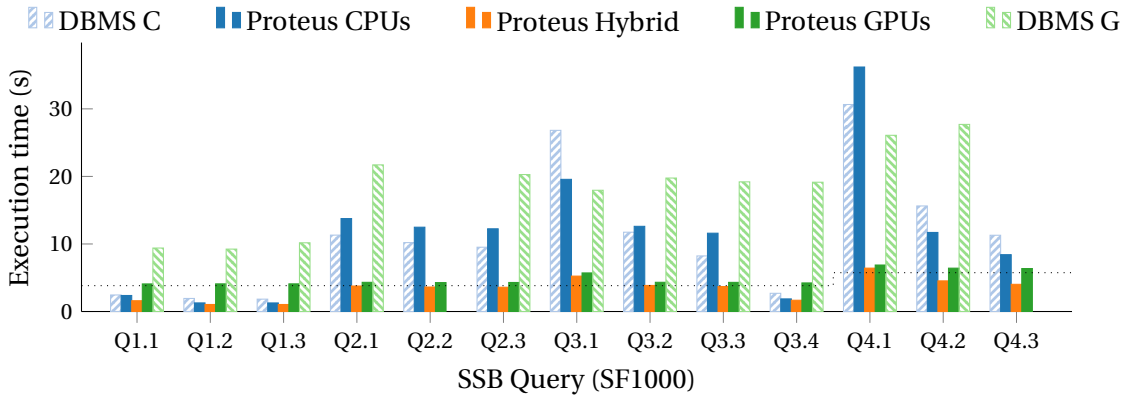


Figure 4.2: SSB with non-GPU-fitting working sets that are pre-loaded in CPU memory for all systems.

Proteus GPU achieves ~21GBps CPU-to-GPU bandwidth for all the queries except Q3.1, efficiently utilizing the interconnects. In Q3.1, the increased selectivity of the first joins of the query increases the number of probes in the next joins and random accesses become a bottleneck, reducing the overall throughput to 16GBps. In addition, HetExchange successfully pipelines transfers and execution and, combined with the efficient generated code, manages to completely overlap them.

In contrast, DBMS G does not reach the interconnect's throughput, as it is not optimized for non-GPU resident datasets and places the dataset into pageable memory. This limits the achievable transfer bandwidth to less than half of the available one and limits overlapping. As a solution, DBMS G proposes to use enough GPUs to fit the working set in GPU memory. For SF1000 and GPUs like the ones used in the experiment, this translates to 9-15 more GPUs. For Q2.2, DBMS G reverts to CPU-only execution and takes more than 1 hour to complete, while for Q4.3 it fails to perform a cardinality estimation that is required to execute the query, due to insufficient GPU memory.

The two CPU-only systems achieve similar performance, and their trends follow the ones of SF100. In contrast with the previous experiment, the GPU systems are bounded by the data transfers. Thus the CPU systems outperform the GPU ones, whenever they can achieve higher throughputs than the interconnects. For SSB, both Proteus CPU and DBMS C only manage to overcome the 24GBps mark for Q1.1-Q1.3 and Q3.4, thus in most queries Proteus GPU prevails. The dimensional table joined in the single join of queries Q1.1-Q1.3 is small enough to fit in the caches of the CPU and thus the CPU systems achieve a throughput of 38-72GBps, or 1.5x-3x the available bandwidth for the two GPUs to access the CPU-resident datasets. Similarly, the very high selectivity of Q3.4 allows both DBMS C and Proteus CPU to exceed the 24GBps landmark and thus run faster than their GPU counterparts.

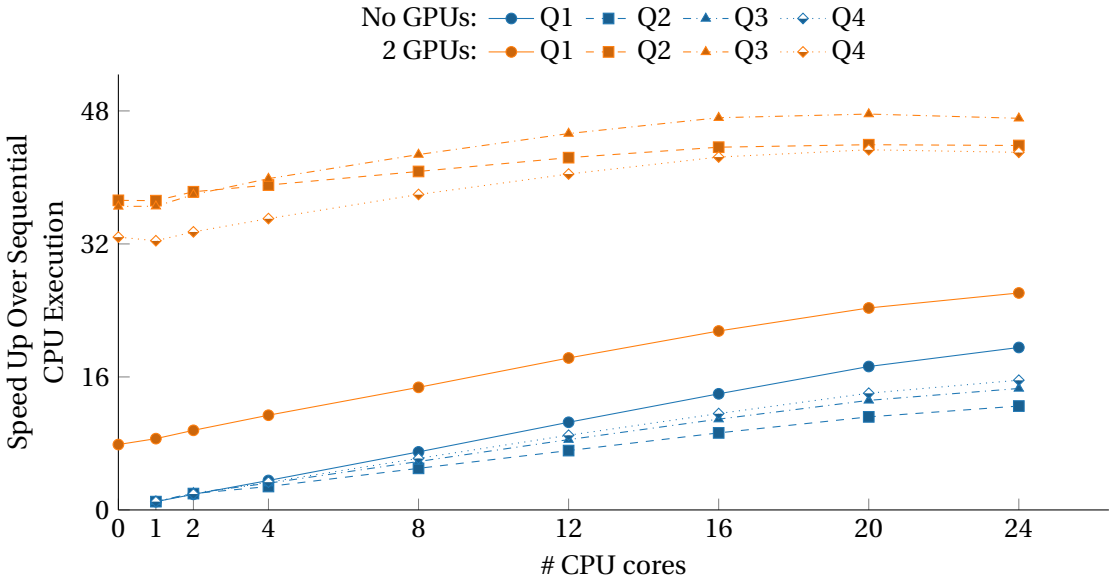


Figure 4.3: Scalability of Proteus on SSB SF=1000.

HetExchange allows Proteus Hybrid to parallelize its execution across all the CPUs and GPUs of the system and benefit in each case from the most appropriate compute units. When Proteus CPU and Proteus GPU have a significant performance difference, Proteus Hybrid's execution times are close to the fastest one, as most of the load will be directed to the fastest compute units. The highest speed-ups for Proteus Hybrid are achieved when Proteus CPU and Proteus GPU have similar performance, as in Q4.3. In these cases HetExchange balances the load evenly between CPUs and GPUs. In contrast with Proteus GPU, part of the load is served by CPUs and thus Proteus Hybrid is not bounded by the transfer time.

In addition, we measure the throughput of the three configurations of Proteus as the size of the working set over the execution time. On average, Proteus Hybrid throughput is 88.5% of the sum of the throughputs of Proteus CPU and Proteus GPU, showing that HetExchange successfully manages to distribute and balance work between the heterogeneous compute units.

Summary. HetExchange allows efficient use of the interconnects while the JIT compilation allows efficient code to be generated for each device. Even for working sets that do not fit in GPU memory, when Proteus is restricted to specific types of devices its performance is comparable or better than state-of-the-art DBMS specialized for these devices. In addition, by using all CPUs and GPUs, Proteus Hybrid outperforms both DBMS in all the queries of SSB SF1000. Specifically, Proteus Hybrid achieves 1.5-5.1x and 3.4-11.4x speed-up against the CPU-based and GPU-based homogeneous DBMS respectively, and up to 5.6x and 3.9x against its own CPU- and GPU-restricted configurations.

4.4.1 Scalability

Methodology. For each SF=1000 SSB query group and the three configurations of Proteus, Figure 4.3 plots the speed up in total query execution time compared to single threaded execution of the same query group. For all the measurements, we interleave the CPU cores between the two sockets and on the x-axis we report the degree of parallelism on the main part of the query.

The CPU-only configurations have almost linear scalability up to approximately 20 CPU threads and a very limited interference when reaching the number of physical cores, due to lightweight threads like the segmenter at the bottom of the plan. Group 1 has the best scalability for the CPU-only configurations with an average coefficient of 87.5% per CPU core, due to its simplicity and the small cache-friendly size of its join's build side. The worst scalability is achieved by query group 2, with a coefficient of 65% per CPU core, due to the high selectivity of its joins. Groups 3 and 4 achieve a coefficient of 74% and 77%, respectively.

Enabling GPUs improves Proteus' performance. Query group 1 exhibits the smallest relative improvements, as the GPUs provide a relatively limited support on its effective utilization of the CPU resources and its high throughput. Two GPUs provide a speed up similar to 8-10 CPU cores. Query groups 2-4, have a higher relative performance improvement from adding two GPUs, equivalent to adding 3.5-5 extra CPU sockets. In groups 2-4, the joins achieve a CPU throughput smaller than the PCIe bandwidth and therefore these queries benefit more than group 1 from additional GPUs.

Summary. HetExchange improves performance across all query groups almost linearly as the number of CPU cores assisting the GPUs are increased, up to approximately 16 cores. For groups 2-4, the benefit of adding more than 16 threads is offset by the interference they cause to threads that handle memory transfers and kernel launches. Using CPU cores is more efficient in query group 1, and therefore this group's performance continues to scale.

4.4.2 Microbenchmarking

Methodology. In the rest of this section, we micro-benchmark Proteus to evaluate the efficiency of HetExchange. Our evaluation uses two queries: i) a sum over a column and ii) a count of the results of a non-partitioned 1:N join. The first query is bandwidth intensive and thus CPU-friendly, as the GPU is behind the much-slower-than-memory-bus PCIe. The second query is GPU-friendly, as the random accesses impact the CPU side more than the GPU side. We use single-column inputs for the queries to stress out HetExchange overheads. For all the cases, the dataset is loaded and evenly distributed to the sockets. Non-HetExchange GPU Proteus overlaps transfers and computations using UVA, as in [102].

Scale-up. The first microbenchmark measures HetExchange's execution time for the two queries and plots the results for different combinations of CPU and GPU degrees of parallelism in Figure 4.4. For the sum query and the probing side of the join query we use a single column

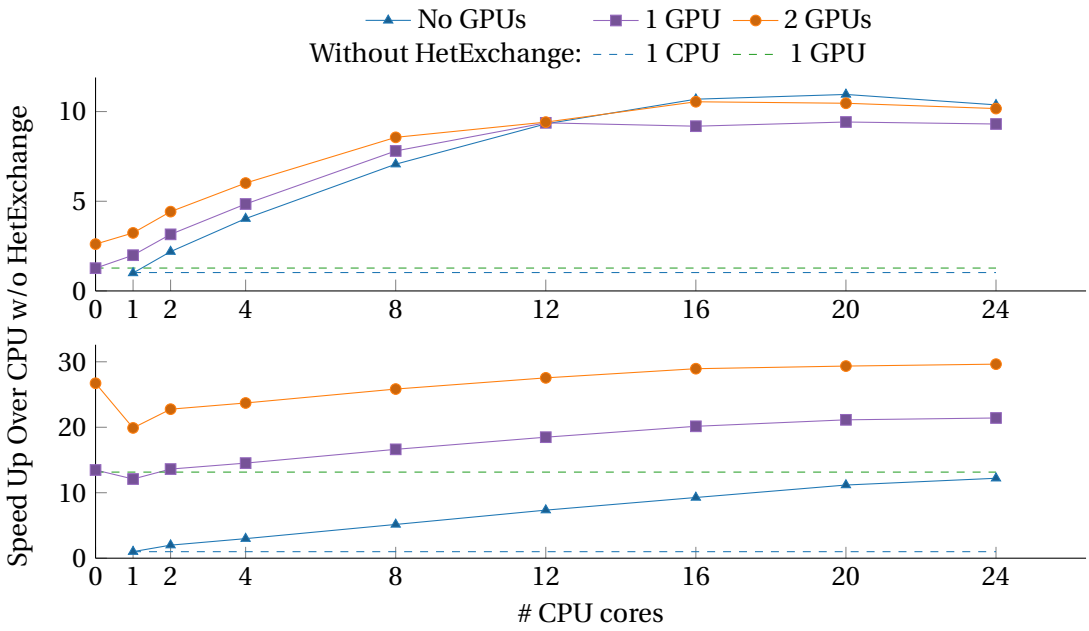


Figure 4.4: Proteus scalability. Top: sum, Bottom: join.

of 23GB, while the build side of the join uses a 7.7MB column. We repeat the experiment and measure the execution time of Proteus without the HetExchange operators, using only its JIT infrastructure and executing on a CPU and a GPU. The results are plotted presented using dashed lines that extend to all the degrees of parallelism to emphasize the functionality provided by the proposed operators: without them, Proteus does not scale up.

Single CPU and single GPU Proteus exhibits a very small overhead for using the operators. For the sum query, the HetExchange augmented Proteus scales almost linearly up until approximately 16 cores. At more than 16 cores, it operates with an input throughput of 89.7GBps which is very close to the maximum theoretical memory bandwidth we obtain from the machine (90.6GBps), given that only 66% of the memory slots are occupied. Adding two GPUs increases the throughput by 19GBps, which slowly diminishes as the number of CPU cores increases, due to exhausting the input memory bandwidth, yielding the same peak performance when the Proteus is trying to use the whole server. When using only one GPU, the peak throughput is smaller, as the routing policy schedules some blocks residing on the remote-to-GPU socket to the GPU and thus causes interference to the intermediate socket.

In the join query, the performance is bottlenecked by random accesses. Thus, each additional GPU provides a significant speed-up for HetExchange, due to the high memory bandwidth provided by each GPU. Further, HetExchange continues to scale even after 16 cores. Adding a single CPU core to the GPU-only configuration causes a performance drop as the GPUs wait for the CPU hash-join's building phase – time which is not replenished by the added performance of a single core. Adding cores eventually pays back, especially in the single-GPU Hybrid mode.

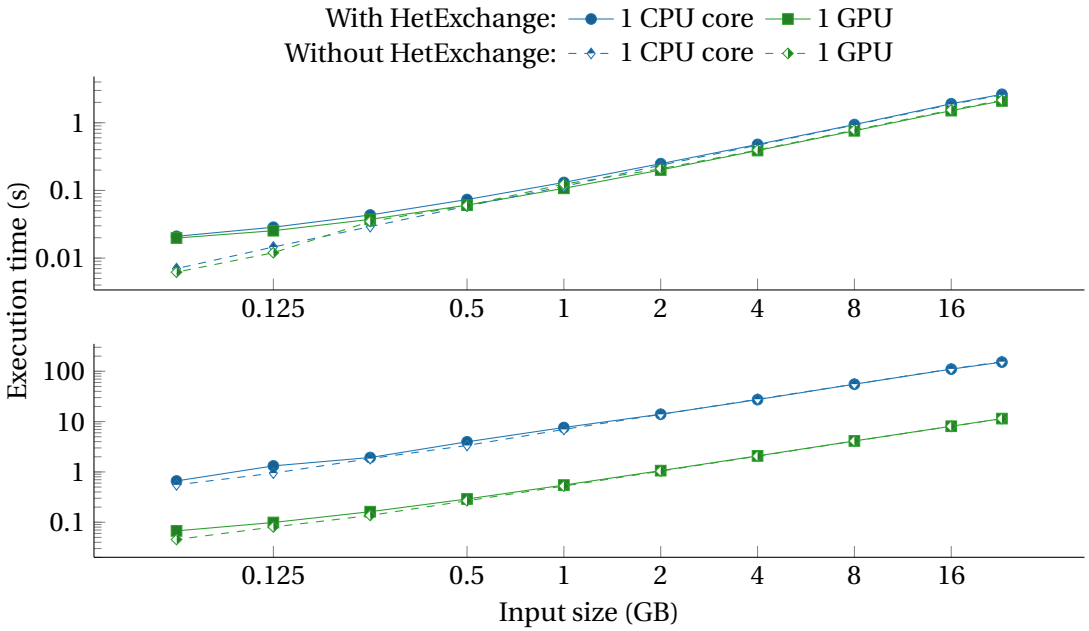


Figure 4.5: HetExchange for DOP=1. Top: sum, Bottom: join.

Size-up. For the same queries, we repeat the experiment and zoom in on the overheads of HetExchange in sequential execution to stress the framework even more. Figure 4.5 plots the execution times for Proteus with and without HetExchange, for varying input sizes. For the HetExchange-enabled configuration, we force the optimizer to add all the HetExchange operators, despite that normally it would avoid routers for sequential execution. We restrict the router's degree of parallelism to 1, to match the sequential execution mode of bare Proteus. For the join query we keep the build table size fixed to 7.7MB as in the previous microbenchmark.

In both queries, the performance is almost identical (at most 10% relative difference) for input sizes more than 512MB, as the overheads of the operators are amortized due to their block-at-a-time nature. For input sizes of 512MB and below, the difference is increased by up to 50% in the case of the summation query on the GPU and input size of only 64MB. For these small input sizes, the high throughput of the generated code makes our current implementation of a router's initialization and thread pinning (that take ~10ms) to become a significant overhead. For such small inputs, an optimizer would normally opt for disabling parallelization. HetExchange's modular operator-level design allows the optimizer to do that by removing the routers, which yields identical performance between the two Proteus flavors even for small inputs.

Summary. HetExchange allows Proteus to scale up and use the available hardware resources and our microbenchmarks show that it adds only a minimal overhead, visible only for small input sizes.

4.5 Conclusion

This chapter presents HetExchange, a framework that encapsulates the main characteristics of a modern multi-CPU, multi-GPU server and allows parallelization of query plans over all the CPUs and GPUs of the system. The rest of the operators remain agnostic to the heterogeneity of the server. HetExchange also enables the encoding of the heterogeneity into the plan and describes a mechanism to enforce the policies about query parallelization on the available hardware, resulting in an infrastructure that is flexible enough to either be directed to selected compute units or left unrestricted to use all of them.

5 Intra-device: Just-In-Time Hardware-conscious Pipelines

While HetExchange allows efficient parallelization across accelerators and decouples the query execution models of different devices, the operators and execution models for intra-device execution still have to be specialized for the different devices. To improve on that, we combine HetExchange a just-in-time code generation infrastructure to allow using modular, device-independent, operators that are specialized, just-in-time, to the target devices. In Section 5.1, we discuss how HetExchange generates specialized code for each pipeline, Section 5.2 discusses how operators differ across devices and how the per-pipeline code generation enabled by HetExchange allows operators to specialize themselves to each device. We describe the integration of the proposed solutions and HetExchange to Proteus in Section 5.3. We conclude an experimental comparison of our proposal with state-of-the-art alternatives in Section 5.4.

5.1 Inter-operator: Generating Heterogeneous Pipelines

This section discusses the lifetime of a query in a HetExchange augmented JIT DBMS and uses an aggregation over a filtered table as a running example. Figure 5.1 depicts the different stages in the lifetime of the query. When a query is submitted, it is first converted into a physical plan that is agnostic to the heterogeneity and parallelism of the server (Figure 5.1(a)). The physical plan is then augmented with the HetExchange operators, described in the previous section, to produce a heterogeneity-aware plan (Figure 5.1(b)): the resulting plan dictates which parts of the plan will be (partially or fully) offloaded, when will the data be moved, or materialized, and any differences between the physical GPU and CPU plan. As a result, the resulting plan parallelizes the query over the mix of CPUs and GPUs available on the system. Then, through JIT compilation the DBMS produces machine code specialized for the server's devices as described in Section 5.1. When invoked, the generated code controls the number of instances of its different parts to efficiently utilize the server.

During code generation the query plan is split into pipelines, and specialized code is generated for each pipeline. Operators that force materialization of intermediate results are typically called *pipeline breakers* [63], and produce code that i) materializes results emitted by the

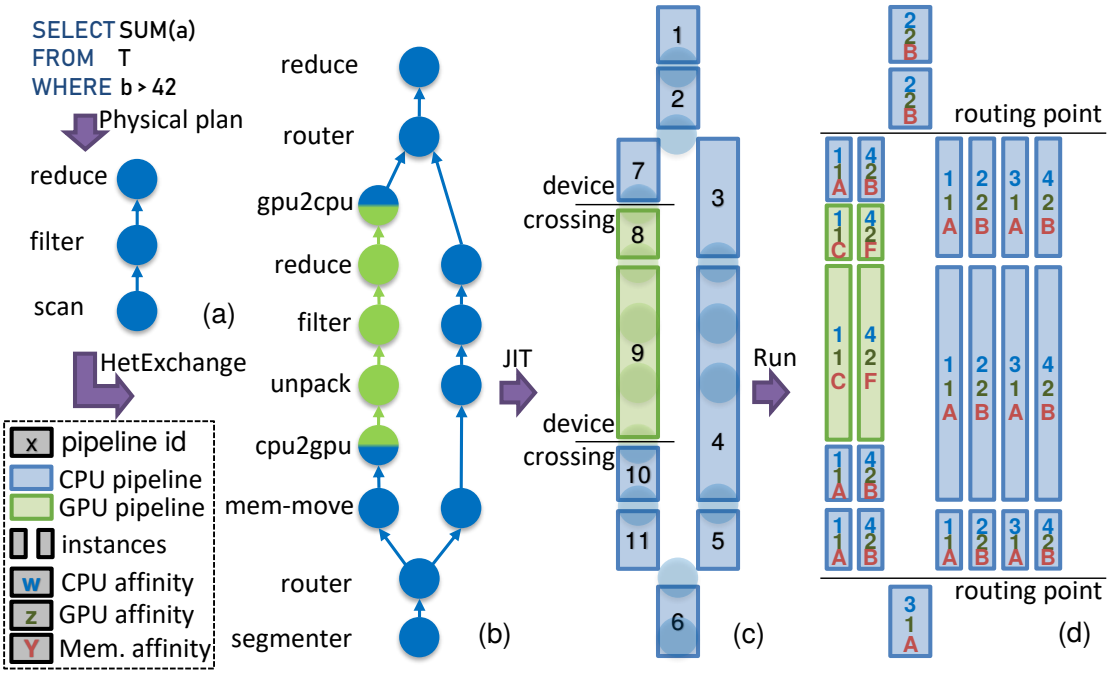


Figure 5.1: Pipelines and affinities of a hybrid plan.

pipeline before the pipeline breaking point (marshalling), and ii) triggers result iteration in the pipeline after the breaking point (demarshalling). As HetExchange operators are handling execution over multiple devices and memories, they are inherently pipeline breakers: they have to materialize the results out of the registers into memory to offload tasks and data across devices. Still, they emit output in batches, without having to first process the entire input. Specifically, in contrast with fully-blocking materialization points that wait for their full input before proceeding to the next phase, HetExchange operators operate in a streaming mode. That is, any materialization happens in small batches and, as soon as a batch is filled up, it is immediately pushed to the next operator without waiting for all the batches to be produced. Overall, this reduces the space required for intermediary results and allows dynamic orchestration of the tasks across the devices.

The code generation phase outputs a set of pipelines, where each pipeline is the result of fusing operators between pipeline breakers into tight segments of code. Pipelines corresponding to the leaves of the query plan trigger the entire generated code; every other pipeline is invoked as a result of invoking these ones.

Traditionally, in a JIT DBMS engine, operators are code generation modules that expose two functions [45, 63]: *produce()* and *consume()*. *Produce()* is called recursively by every operator in top-down fashion (i.e., starting from the root of the query plan): every operator asks its children to *produce* their result tuples. *Consume()* is called recursively by every operator in bottom-up fashion: every operator asks its parent to consume the tuples just pushed to it, essentially triggering the parent to generate its implementation.

Device Provider Methods		
allocStateVar	get/releaseBuffer	#threadsInWorker
freeStateVar	malloc/free	threadIdInWorker
storeStateVar	convertToMachineCode	loadMachineCode
loadStateVar	workerScopedAtomic<T, Op>	

Table 5.1: Functions overloaded in device providers, per device.

In the running example, when the router at the bottom of the plan is about to generate code, it will call the produce() method of the segmenter, so that the latter generates its code first. The segmenter is a leaf operator, so it will proceed with code generation without further produce() calls. Instead, the segmenter will generate code similar to lines 1–3 of Listing 5.1: The segmenter's generated code comprises a nesting of two loops, which gather the list of memory segments of relation T, and break them into blocks. Then, the segmenter will call the router's consume() method, triggering the router to produce its physical implementation. The router will then produce its implementation (lines 4–8) to evaluate the policy on each block, and, based on the result, it will send the block handle to a specific consumer that is either an instance of pipeline 5 or 11.

JIT on multiple devices: The missing pieces. Directly mapping traditional JIT techniques to the case of heterogeneous servers would require having multiple implementations of the same high-level operators, with each implementation targeting a different device. Such a design causes increased programming and maintenance efforts. For example, a relational reduce operator would require a different implementation and code generation procedure per device, thus hindering the extensibility of such an architecture. In addition, in order to achieve inter-device task parallelism, the JIT infrastructure has to be able to handle transitions between different device type targets; otherwise, the generated code will target only one device type.

JIT on multiple devices with HetExchange. HetExchange simplifies multi-device code generation in three steps: First, it decomposes the query plan into multiple parts, each of which is specific to a device type. Second, the aforementioned device crossing operators of HetExchange also encapsulate the transitions between compilation targets. Finally, HetExchange redesigns the *produce()* and *consume()* methods of each operator to enable them to generate code that is device-specific, yet not specializing their implementation to a device, by parameterizing each method with a device-specific *provider*.

Device providers. Even if a JIT DBMS generates code for a single device, it should ideally rely on a collection of utility functions as building blocks for its implementation. These utility functions should handle operations such as the following: i) locating a pipeline's state, such as pointers to data structures, ii) acquiring/releasing device memory, iii) acquiring/releasing locks, and performing atomic operations, and iv) retrieving device-specific characteristics,

such as the grid and block size used by a GPU kernel. Further, isolating them from the main operator code eases the operator portability as the implementation of these utilities differs across target architectures.

HetExchange groups the collection of all the utility functions into a device-independent interface, and offers a collection of *device providers* implementing said interface; a CPU- and a GPU-specific provider at the moment. Device crossing operators are the ones specifying which device provider every pipeline should use; each pipeline's operators then use the provider given to them when appropriate. Thus, if a pipeline targets, for example, a GPU device, the methods of the pipeline's operators will make calls to a *GPU provider* to generate GPU-specific stubs. The same pipeline could generate code for a CPU with no changes other than being instantiated with a different provider as input. The overall implementation of the produce() and consume() methods per operator will thus remain agnostic to the device properties.

Aside from their other responsibilities, the device providers also guide the final steps of the compilation in order to optimize the generated code and produce machine code for the target device. Upon completing the code generation of a pipeline, it is optimized, compiled down to machine code and loaded into the running instance of the DBMS. The device provider of each pipeline is responsible for specifying how each of these steps is achieved.

JIT code for heterogeneous servers example. As already described in the running example of Figure 5.1(c), the segmenter and the producer part of the bottom router will be fused into pipeline 6 which sends block handles to pipelines 5 and 11. Both of these pipelines wait for handles from the router, as part of the code generated by the consumer part of the router. Then, mem-move will generate code that checks for each received handle if the block is on the target memory node. If it isn't, the generated code requests a new block on that node and spawns an asynchronous DMA transfer to copy the data to it. In any case, mem-move propagates to the next pipeline a block handle that is on the local-to-the-consumer memory node together with information about which transfer the consumer should wait for, if any. In the beginning of pipelines 10 and 4, the two mem-moves inject code to receive these handles and wait for the transfer to complete (lines 22–27 of listing 5.1). Then, pipeline 4 will unpack the block, check the filter and update the accumulator, based on the code generated by the unpack, filter and consumer part of reduce respectively. Pipeline 10 will schedule a GPU kernel of pipeline 9 with the received block as argument, due to the code generated by the producer part of the cpu2gpu operator.

Listing 5.1 shows, in pseudocode, a simplified version of the generated code for pipeline 9 of the running example. The four participating operators are fused into a simple GPU kernel that scans each block, evaluates the filtering predicate and increments the accumulator accordingly. The consumer part of the cpu2gpu operator specifies the arguments of the pipeline and the unpack generates the scanning. For each tuple, the code generated by the filter and the reduce in lines 34–39 is executed.

```
1  def pipeline6()
2    for each segment in file                                          // segmenter
3      for each block in segment                                       // unpack
4        /* Use a routing policy to decide the target                 // router
5         * (based on block metadata and queue sizes)
6         * and send to corresponding consumer: */
7        c ← evaluate policy on block
8        send handle of block to consumer c
9
10 def pipeline11()
11   for each received block handle b                                  // router
12     /* If the block is not on the target device, move it.          // mem-move
13      * Either way, push it to the following instance of the
14      * consumer instance ("inst"). */
15     if b not on destination
16       d ← get block handle on destination
17       schedule DMA copy from b to d
18       send d to consumer (pipeline10) instance
19     else
20       send b to consumer (pipeline10) instance
21
22 def pipeline10()
23   /* Wait for blocks from the producer (pipeline 11)               // mem-move
24    * and for each block wait for the corresponding
25    * transfer, if any, to complete: */
26   for each received block handle b
27     wait DMA transfer for b to finish
28     schedule pipeline9(b) for GPU execution                         // cpu2gpu
29
30 def pipeline9(data_block[N], state)
31   local_acc ← 0                                                     // reduce
32   for i=threadIdInWorker to N-1 with step #threadsInWorker          // unpack
33     t ← data_block[i]
34     if t.a > 42                                                     // filter
35       local_acc ← local_acc + t.b                                   // reduce
36   /* Hierarchically reduce across worker threads: */                // reduce
37   nh_acc ← neighborhood_reduce(local_acc)
38   if thread neighborhood leader
39     atomic_add(state.acc, nh_acc)
```

Listing 5.1: Pseudo-code for pipelines 6 and 9-11.

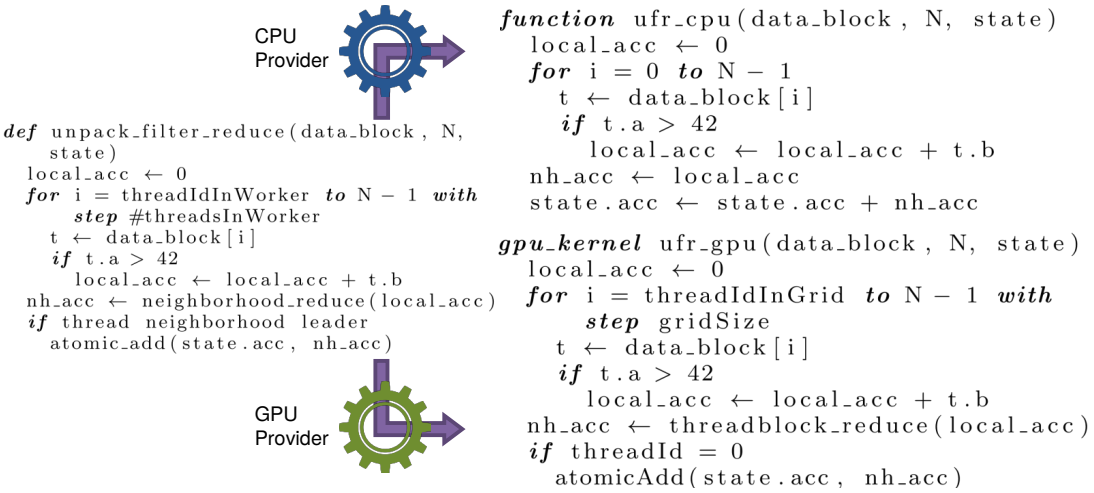


Figure 5.2: Providers specialize code to the target device type.

Pipeline 8 will read the final result of the aggregation and insert it in the queue of the gpu2cpu operator. On the consumer side, the gpu2cpu operator generates code to wait for input in the queue and when values are written, it reads them and propagates them to the router, which will send them to its single consumer, the single instance of pipeline 2. Similarly pipeline 3 reads the result of the CPU reduction and sends it to the same instance of pipeline 2 via the router's queues. Pipeline 2 waits for the partial aggregations to arrive via the router and accumulates them. Pipeline 1 will read the final aggregation which is the query result.

In the running example, pipeline 9 is associated with the GPU provider, as it targets GPU execution. The provider will translate threadIdInWorker into the thread id and $\#threadsInWorker$ to the number of GPU threads in the kernel. When memory is allocated for the $global\_acc$ accumulator in the state, the provider will generate a call to the GPU memory allocator in order to allocate the state in GPU memory. In addition, the neighborhood considered by $neighborhood\_reduce$ will be a GPU thread-block and the worker-scoped atomic add will be translated into the corresponding GPU atomic instruction. Lastly, the provider will optimize the pipeline after its generation, then compile it down to machine code for the GPU and load it into the GPUs. Pipelines 9 and 8 target GPU execution and thus are associated with the GPU provider. All other pipelines are associated with the CPU provider.

Figure 5.2 shows an example of a pipeline that results in different code depending on the provider it uses. The pipeline depicted on the left-hand side of the figure is provider-agnostic and generic enough to be specialized for a CPU or a GPU. The code loops through thread workers with increments of a given step, it evaluates a filtering condition and increases the value of a thread-local variable when the condition is successful. Once the loop completes, the operator accumulates the thread-local variables into per-warp variables, and then the leader of each warp updates a worker-scoped accumulator atomically. If HetExchange was not using device-specific providers, pipelines 4 and 9 would result in similar code, which is suboptimal for CPU execution, because it is overly complex. Instead, through the use of

CPU and GPU providers, HetExchange specializes code to the target device, while keeping the operator "blueprints" the same for both devices: For example, the $threadIdInWoker$ will be set to 0 for the CPU provider, while it will be set to the GPU grid-wide thread id for the GPU provider. Similarly, $\#threadsInWorker$ will be set to 1 for the CPU provider and to $gridSize$ for the other one. More importantly, as there is a single thread in the CPU case, the worker-scoped atomic and the neighborhood-local reduction will be optimized out.

5.1.1 Controlling Parallelism and Affinity

In a HetExchange-augmented DBMS, the router controls the horizontal degree of parallelism for the query plan operators above it. At code generation time, depending on its policy (e.g., hash- or round-robin-based) and the intended degree of parallelism, the router is responsible i) for producing multiple pipelines on its consumer side, and ii) for triggering code generation for these pipelines. An additional source of complexity is that while the classical Exchange has one parent and one child operator which are instantiated multiple times, the router has multiple parents and children to parallelize the rest of the plan to a mix of compute units.

Given that the pipeline instances to be generated are almost identical, it would be inefficient to trigger code generation from scratch for every one of them. Thus, the router generates a parameterizable version of the pipeline in question *per device* (instead of per thread), and then initializes multiple instances from this "pipeline template" (i.e., performs state creation for each one).

As only the router controls parallelism, it is also responsible for pinning pipelines to specific devices, based on pluggable policies. When a router instantiates its consumers, it locks them to specific devices. For the policies to be able to control pipelines not attached to a router (e.g., pipelines 9 & 4 of the running example), HetExchange forces pipelines to inherit both the degree of parallelism and the affinity of their instantiator. Assigning both a CPU and GPU affinity to all pipelines, but using only the appropriate one, allows routers to control the affinity of pipelines even after multiple device crossings (e.g. the bottom router controls the affinity of pipeline 7; the information is not lost by the device crossings).

In its current form, the router specifies operator affinity, degree of parallelism and routing policy statically at query time. Future work involves making such decisions dynamically and integrating existing work in this direction such as dynamic schedulers [100] and opportunistic task stealing between different pipelines [51].

Parallelism and affinity example. As in the running example of Figure 5.1(d) pipeline 6 is a leaf pipeline, it runs single-threaded. The bottom router injects code in pipeline 6 to instantiate pipelines 11 and 5, two and four times respectively. In addition, the router will pin the first instance of pipeline 11 to CPU core 1 and GPU 1, and the second instance to core 4 and GPU 2. Each instance of pipeline 11 will create an instance of pipeline 10 and the latter will copy its

instantiator's affinity. Similarly, pipelines 7-11 will have two instances with the corresponding instances pinned to the same compute units. The GPU affinities will be considered only for pipelines 8 & 9, while all other pipelines use the CPU ones.

5.1.2 Memory Management and Data Transfers

During query execution, memory is used either to store operator state, like hash-join's hash table, or to stage blocks of intermediate results before transferring them across devices. HetExchange distinguishes between the two and has a different manager for each of them. State memory is served by memory managers, while staging memory is served by block managers. Both memory and block managers are organized as a set of independent, local components – one per memory node. Requests by the pipelines are always served by their closest (appropriate) manager.

While memory managers only manage local memory, block managers frequently handle data operations that involve remote devices. Also, block managers need to be thread-safe, yet existing synchronization primitives are very expensive due to the absence of global, cache-coherent shared memory. HetExchange tackles these challenges in the following ways: Firstly, at system initialization time, the block managers pre-allocate memory (block) arenas, to avoid memory allocation costs at query execution time. Secondly, to circumvent the absence of coherence, HetExchange allows only local devices to acquire blocks from a block manager and opts for device-local synchronization primitives. To serve requests for remote blocks, managers acquire blocks by launching small tasks to the remote node. As this can become costly, HetExchange accelerates the common cases by i) having each local block manager maintain a cache of acquired blocks per remote manager, and ii) batching requests for block acquisition and release from remote nodes.

5.2 Intra-operator: Operator Portability

The proposed design allows decomposing execution into execution in homogeneous sub-systems and the optimizer to opt for hardware-conscious operators tuned for the specific target device alongside the range of supported hardware-oblivious operators. As discussed in Section 2.4, this brings the potential for significant performance benefits over generic hardware-oblivious operators.

Using target-specific hardware-conscious operators has the potential to boost performance. Prior work optimizes data movement and access patterns with respect to the device's caches [14, 15, 95, 96], including TLBs, and their characteristics. Other works consider properties and functionalities of processing units such as the instruction level parallelism (ILP), branch predictors, SIMD instructions for CPUs, as well as warp-wide execution and shuffles in GPUs. Operator implementations need to exploit properties of the underlying hardware and explore the available opportunities within the design space to achieve high performance.

Common design, different specialization. Despite the microarchitectural differences, the exploration of hardware-conscious operator designs is similar across different devices. When optimizing operators for each device, the challenges are the same (eg. avoiding random access overheads) and thus similar algorithmic solutions can be applied to a range of device types. The hardware-conscious radix join [94] is an indicative case: independently of CPU or GPU execution, random accesses are the main bottleneck of a non-partitioned hash-join, as they waste memory bandwidth due to over-fetching. In both CPUs and GPUs, similar algorithmic approaches can mitigate the problem by, for example, partitioning the input to fit the per-partition hash-tables in a memory (cache) with a higher bandwidth. On the CPU side, the partitioning fanout is restricted by the TLB size while, on the GPU side, it is restricted by the size of the cache used for write offsets and store consolidation. In both cases, the end result is a multi-pass partitioned hash-join.

In GPUs, Sioulas et al. [94] avoid random accesses to L1 that waste bandwidth due to over-fetching by using the scratchpad instead. More specifically, Sioulas et al. load the smaller partition to the scratchpad, build the hash-table using atomic operations and probe with the tuples of the corresponding partition. The scratchpad is organized into banks and is capable of serving a different word from each bank per warp-wide request, independently of its location in the bank. Thus, the scratchpad only penalizes accesses to the same bank, but does not waste bandwidth by over-fetching. The scracthpad has a similar size to L1, and thus careful tuning is needed to fit the output partitions in the scratchpad. In the CPU case, the partitioning is tuned to reduce the TLB misses and improve the cache locality of the output. Similarly, in the GPU case, tuning aims at reducing the sparsity of stores but the fanout is restricted by the memory available for consolidating the stores, which also happens using the scratchpad.

While the CPU and GPU hardware-conscious radix joins are tuned for the specific memory hierarchy of each device, the skeleton of the algorithm remains the same for both CPUs and GPUs. Thus, the design of hardware-conscious operators has two components: the algorithmic skeleton and the hardware-specific fine-grained building blocks that change between different device types, such as caching the hash table in the scratchpad. This allows re-using the algorithms across devices and separating hardware-consciousness from device-consciousness: algorithms may be capable of solving different hardware-specific device-invariant problems (e.g., random accesses through multiple partitioning steps), but the exact mappings to the hardware may differ per device (e.g., fanout based on TLB versus scratchpad capacity).

5.3 System

We integrate HetExchange and its system architecture to Proteus [45], an analytical query engine that utilizes LLVM-based code generation. Proteus originally generated CPU-specific and single-threaded code. Therefore, we extended Proteus's infrastructure to allow GPU-specific code generation, by introducing code generation components for single-GPU opera-

tors. Enabling Proteus to operate over multiple CPUs and GPUs requires i) extending its code generation infrastructure to produce code for parallel execution, and ii) coupling Proteus with HetExchange non-intrusively.

LLVM is capable of compiling code for multiple architectures by using a different back-end for each target, like the x86_64 back-end that is used by Proteus for code generation targeting Intel CPUs. In addition, LLVM has back-ends for both NVIDIA and AMD GPUs. For the evaluation of HetExchange we use the NVPTX back-end to generate code for NVIDIA GPUs. While our methods are applicable to AMD GPUs, we leave the implementation as future work.

In Proteus, the providers use LLVM's code generation interface for the low-level code generation, such as load and store operations, while for the high-level functionality, like state manipulation and memory allocations, they are emitting the relevant code. Generated code is optimized using LLVM. The CPU provider uses LLVM to compile the IR down to machine code and loads it in the running instance, while the GPU provider uses LLVM to compile the IR to PTX [70], an assembly language for NVIDIA GPUs, and the CUDA driver API to compile PTX to machine code.

Similarly to Figure 5.1, upon receiving a query, the extended Proteus parses and optimizes it in order to produce a single-threaded CPU-only physical plan, like the one in Figure 4.1a. This plan is then extended with the HetExchange operators to a heterogeneity-aware plan like the one in Figure 4.1e. The heterogeneity-aware plan describes which devices will be used in each part of the generated code. Then, based on the heterogeneity-aware plan, Proteus generates code for the query and start executing it. In our implementation, part of the query optimization is handled by Apache Calcite [11]. We opted for this three step query optimization process (logical → physical → heterogeneity-aware plan) as a proof of concept, but integrating the two last steps into a single one is also possible. Selecting between this two options is a trade-off between plan optimality and query optimization times. While producing heterogeneity-aware plans is a topic worth as much research as enforcing them, we leave it as future work and for this evaluation we heuristically add the HetExchange operators. For this work, we opted for the three step process, as between these two options, it creates the smallest overhead to the query optimizer.

5.4 Evaluation

Methodology. To evaluate the proposal of this chapter, we focus on the single-device-type performance. Further, to avoid distorting the experimental results with the cost of data transfers, we pre-load the data to the devices under evaluation, e.g., for GPU execution we preload the data in GPU memory. We use the same experimental setup as in Section 4.4. To evaluate the code generation and pipeline specialization, we evaluate query execution on data that are already resident to the device memory. Specifically, for each SSB SF100 query,

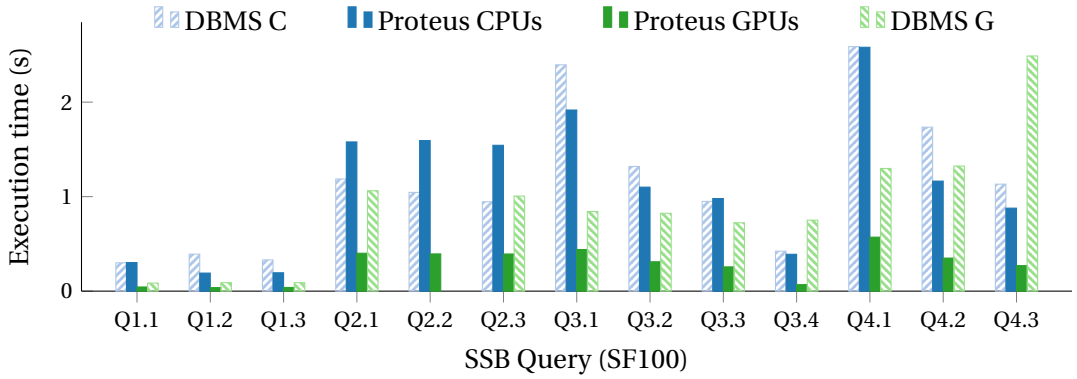


Figure 5.3: SSB with GPU-fitting working sets. Data in GPU memory for GPU systems.

Proteus GPU and DBMS G preload the necessary columns in the aggregate device memory of the two GPUs (16GB). DBMS C and Proteus CPU configurations operate over columnar data that are pre-loaded in CPU memory.

Proteus GPU randomly partitions each table between the two GPUs. We profiled DBMS G and noticed an absence of cross-GPU PCIe traffic during query execution; therefore, DBMS G either performs co-partitioning of the fact and the dimension tables, or broadcasts (dimension) tables to both GPUs a priori. For all queries, the optimizer of Proteus opts for broadcast-hash-join-based plans; HetExchange broadcasts the dimension table columns involved in joins to both GPUs. DBMS G opts for a star-join-specific join implementation: It conceptually treats each dimension table as a dense array $dimtable[]$, where the value $dimtable[key_i]$ corresponds to the tuple whose key column value is key_i. DBMS G performs the (star) join by iterating over the fact table and fetching the corresponding values from the dimension tables/arrays via array index lookup.

Figure 5.3 depicts results for SF100; HetExchange enables Proteus to seamlessly parallelize its execution across computational units. Q1.1 - Q1.3 are the simplest SSB queries; they perform a single join of the fact table with the dates table. Proteus GPU and DBMS G outperform the CPU-based systems, because the GPU devices offer high memory bandwidth (320GB/s) and number of hardware threads. Proteus GPU utilizes the resources of the GPU devices more efficiently and thus outperforms DBMS G. Specifically, every thread block that DBMS G triggers on the GPU devices allocates double the number of GPU registers than Proteus GPU. Thus, DBMS G launches fewer simultaneous execution units and underutilizes the large number of available GPU hardware threads.

Q2.1 - Q2.3 increase the number of joins between the fact table and dimension tables to three; the effect of hardware underutilization becomes more visible for DBMS G, thus its difference from Proteus GPU increases, and its performance resembles that of DBMS C. DBMS G fails to execute Q2.2's string inequalities.

Q3.1 - Q3.4 also have three joins, with each consecutive query being more selective than the previous; Proteus GPU is consistently faster. For Q3.1 and Q3.2, Proteus CPU is faster than DBMS C because the operators of DBMS C have to either materialize a result vector or a bitmap vector, whereas Proteus CPU operates as much as possible over CPU-register-based values to avoid materialization costs. Q3.3 and Q3.4 are more selective, therefore the gap between Proteus CPU and DBMS C becomes minimal. In addition, although the star-join implementation of DBMS G turns joins into inexpensive array lookups, DBMS G also opts to apply filtering predicates *after* the completion of the star-join, so that the dimension tables resemble sorted, dense arrays at join time, and the star-join turns into a sequence of array index lookups. Thus, DBMS G's benefit from selective filtering predicates is minimal.

Q4.1 - Q4.3 increase the number of joins to four, with each consecutive query being more selective, and are the most challenging part of SSB. All systems except DBMS G benefit from queries being more selective. Proteus configurations outperform their CPU/GPU counterparts due to the minimal generated code that comprises every query pipeline that Proteus executes and the better utilization of GPU hardware resources.

Summary. HetExchange enables Proteus to parallelize queries across multiple CPUs and GPUs and operate over different initial data placements, in the same infrastructure, without loss of generality or performance. Proteus is comparable or outperforms state-of-the-art DBMS that target CPUs or GPUs. When the working set fits in the aggregate GPU memory, Proteus achieves up to 2x and 10.8x versus CPU- and GPU-based alternatives, respectively.

5.5 Conclusion

Designing HetExchange and incorporating it into our system required considering a number of challenges related to i) encapsulation of parallelism, ii) encapsulation of hardware heterogeneity and iii) choice of execution model for analytical queries; tackling these challenges led us to a number of observations that can be useful as guidelines to database system architects.

Separation of concerns. The design space for a system that can execute queries over both CPUs and GPUs is significantly wide. Picking and changing the degree (and type) of parallelism, transferring data between processors, and handling arbitrary data placement across processors' memories, are a few of the concerns to be resolved. HetExchange deals with this design space explosion by enforcing a clear separation of concerns: Explicit operators deal with orthogonal issues such as cross-device transfers, parallelism encapsulation, and memory affinity. Such compartmentalization allows for a generic and extensible system; extending HetExchange to another type of processor in the future would be non-trivial with a monolithic design, while the current design only requires an extra device provider and two device crossing operators.

Vectorization vs. compilation. Despite being coupled with a JIT-compiled architecture in this work, HetExchange can enable execution over heterogeneous processors for any type of query execution engine, be it interpreted or compiled. Still, implementing a real-world system required considering a type of execution engine to pick. Given the performance benefits they bring in analytical query processing, our main considerations were vectorized [15] and pipelined, compiled engines [46]. If HetExchange targeted CPU processors exclusively, vectorized execution would have been a great fit as well, as there are families of operations for which it can even outperform compiled execution [47, 60, 95]. In addition, implementing a vectorized engine is more straightforward than a JIT-compiled one. However, vector-at-a-time execution can be wasteful in the context of GPU processing; the materialization overhead it entails becomes more pronounced when (cache) memory is scarce: GPUs have lower per-thread cache capacity compared to CPUs. Also, relying on code generation infrastructure allows the resulting system to have a single, unified code base of pipelined operators instead of a CPU- and a GPU-family of vectorized ones. Lastly, our design is compatible with the work of Menon et al. [60] which introduces SIMD vectorization in CPU JIT engines.

The compiler (sometimes) knows better. Writing code to be executed on a GPU can be a very subtle process [18, 22, 27, 42]. Conventional knowledge has it that a developer needs to explicitly reason about numerous low-level details, such as, among others, i) the organization of GPU threads in thread blocks, and of thread blocks in grids, ii) thread divergence within a thread warp, and iii) avoiding atomic operations. When this source of complexity is coupled with the complexity of implementing a code-generating engine, the end result can be very burdening to a developer. During the coupling of HetExchange with Proteus, we observed that the compiler has become significantly better in optimizing code that has not been meticulously fine-tuned to the device-specific "magic numbers" required for thread block size, etc., to the degree that a lot of the conventional GPU coding wisdom [22] has become obsolete for modern GPUs. Thus, choosing to offload a part of the GPU code optimization to the compiler reduces developer effort and focuses on the bigger system picture instead of micro-optimizations.

6 Efficient Interconnect Utilization

Fast analytics are essential for generating timely business intelligence. In the previous chapters, we showed that hybrid CPU-GPU processing significantly accelerates analytical query processing. However, as datasets increase in size, the limited GPU memory capacity forces in-memory analytical engines to store the data in CPU memory. As a result, offloading tasks to the GPU also requires accessing the corresponding data over the interconnect – limiting the potential acceleration despite potentially idle GPU resources.

To overcome the interconnect limitation, multiple approaches have been proposed in the literature. First, some GPU analytical engines rely on hardware scaling: either they suggest a sufficient number of GPUs to cache the datasets in GPU memory [2], or they use integrated GPUs that reside on the same package as the CPU and have access to memory at full CPU memory bandwidth [36]. However, increasing the number of GPUs comes at a significant monetary cost compared to using the existing CPU memory, and integrated GPUs are generally weaker, resulting in lower analytical performance [44, 87]. Second, prior work has used data compression to increase the number of tuples kept in GPU memory [44] and reduce the data transfer volume by streaming compressed pages and decompressing them on the GPU side [88]. However, such approaches rely on the data distribution to reduce the data volume instead of the actual query – potentially missing additional optimization opportunities for selective queries. Third, Pirk et al. [75] propose fixed, data- and query-agnostic compression methods that send and store approximate data representations in the GPU. These approximate methods allow the GPU to produce an initial query result set that is then refined by the CPU. Such approaches have the potential to avoid the interconnect; however, the proposed methods rely on the approximate structures fitting in the GPU memory and the refinement step to introduce minimal CPU overhead. Lastly, Yuan et al. [102] rely on UVA and the interconnect to access the corresponding data, causing significant data overfetching and relying on the GPU kernels being able to hide the corresponding data stalls. Overall, existing techniques rely either on the data distribution enabling sufficient compression or hardware primitives providing sufficient bandwidth.

In this work, we propose Laconic, a set of data access methods that reduce the overhead of data fetching during data transfers for GPU-accelerated analytics. Laconic exploits the query selectiveness and the fine-grained in-memory data accesses. First, we evaluate the impact of the interconnect and GPU technology in offloading data analytic tasks to the GPUs, and model the throughput overhead of fine-grained data accesses. Based on expression selectivities, we eagerly prefetch columns used in expressions that provide limited filtering, reducing data stalls associated with over-the-interconnect data fetches. Then, we identify the curse of the first column: GPUs have to pull in at least one entire column to benefit from highly selective conditions, which limits the performance benefits of selectively pulling data into the GPUs even for highly selective queries. Lastly, we propose a CPU-GPU push-based access method that reduces data transfers over the interconnect and avoids the curse of the first column by pushing lightweight prefiltering operations to the CPU. Overall, Laconic optimizes the data transfers over the interconnects based on the query selectivity and reduces the data transfer cost required to benefit from GPU acceleration.

In summary, Laconic makes the following **contributions**:

- We show that GPU-accelerated analytics are bottlenecked by interconnect transfers and model the overhead of lazy pull-based data transfers.
- We optimize data transfers by exploiting over-the-interconnect fine-granularity data accesses combined with near-data prefiltering techniques to accelerate even very selective queries.
- We integrate Laconic into Proteus and show that GPU acceleration benefits even very selective queries, with Laconic achieving up to 13x and 4.6x speed-up versus GPU-accelerated query processing that uses eager data transfers and lazy direct memory accesses, respectively.

Overall, Laconic reduces over-the-interconnect data transfers – one of the main performance bottlenecks for CPU-GPU analytical query processing. As a result, Laconic increases the spectrum of analytical queries that benefit from accelerator-level parallelism.

6.1 The Role of the Interconnect in CPU-GPU Analytics

In-GPU Query Processing: Capacity-Limited Acceleration. The data-parallel nature of analytics and the high-bandwidth GPU memory make them a great match. However, the GPU memory is optimized for iterative workloads that exhibit data locality. Thus, hardware vendors prioritize high bandwidth over high capacity when sizing the on-package GPU memory. As a result, GPU memory is often on-package, faster than CPU memory, but also much smaller – typically less than 100 GB. As a consequence, in-GPU query processing on GPU-resident data is capacity limited, and for bigger workloads GPU-accelerated engines have to stream input data from the CPU memory.

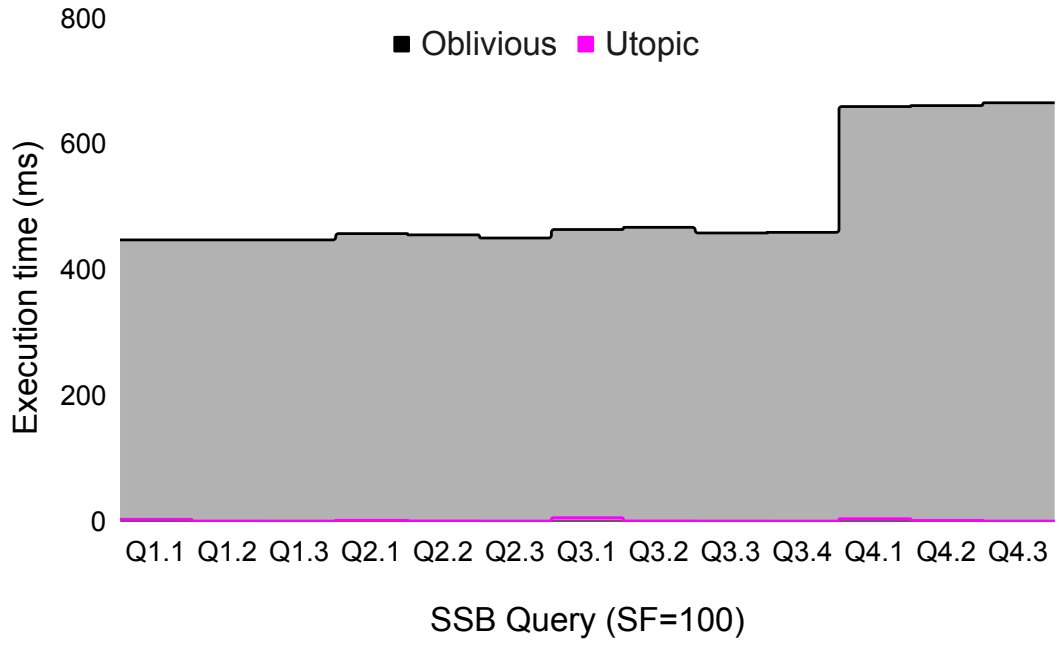


Figure 6.1: In-GPU query execution using an eager (Oblivious) data transfer method, versus transferring only the qualifying tuples

GPU-accelerated Analytics: Processing on the Other Side. Still, the high bandwidth of the GPU memory combined with the big register files and the high number of on-the-fly memory requests is a great fit even for random-access-heavy operations. For example, sequences of non-partitioned hash joins can stream tuples from the CPU side and spawn multiple probes in the local GPU memory. Given the ratio of interconnect-to-GPU-memory bandwidth, most GPUs can sustain at least a few consecutive joins before becoming GPU-memory bound. However, given the available local memory bandwidth, the analytical GPU power is often highly underutilized as the interconnect limits the data bandwidth available for offloading data to GPUs.

Interconnect: The Narrow Waist. With the interconnect bandwidth being a known constraint of existing GPU systems, many new solutions and standards have been proposed. The most common standard for CPU-to-GPU communication is PCIe, which provides 12.8 GBps CPU-to-GPU data bandwidth at PCIe 3.0 x16. NVIDIA also has its own specialized interconnect, NVLink, which is generally available only for GPU-to-GPU communications. Still, some CPUs, like IBM Power 8 & 9, and the upcoming NVIDIA Grace CPU provide support for CPU-to-GPU communication over NVLink. In version 2.0, NVLink provided up to 75 GBps CPU-to-GPU bandwidth. While high-bandwidth interconnects improve the data transfer bandwidth, they are still an order of magnitude slower than the GPU memory and almost 2x slower than the CPU memory. As a result, to efficiently use the GPUs, we must improve the input-data streaming efficiency.

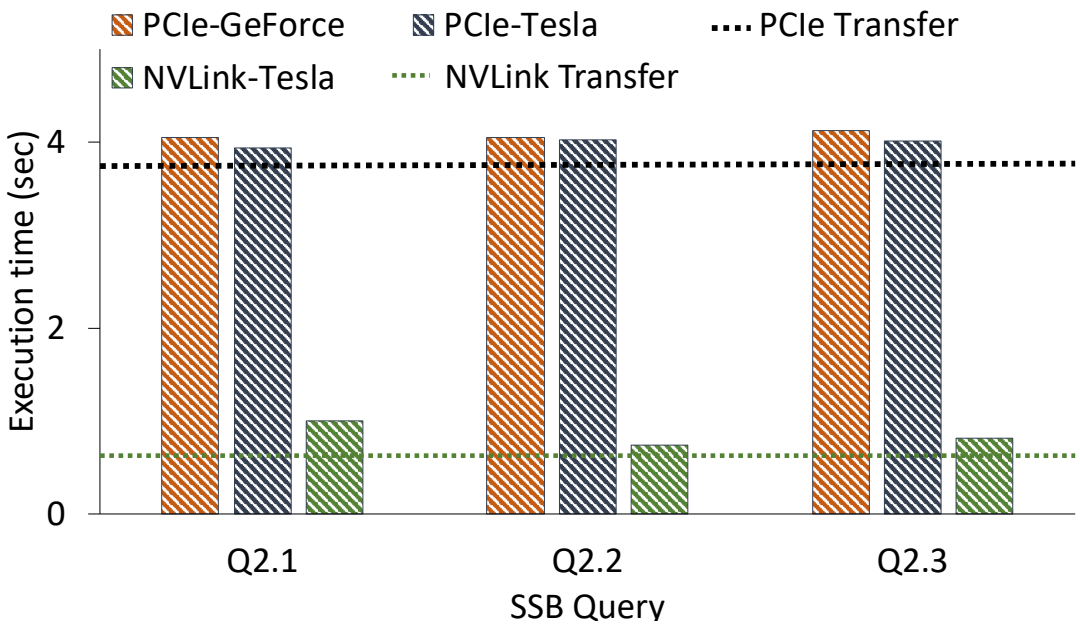


Figure 6.2: GPU query execution on CPU-resident SSB (SF=1000) across different GPU architectures & interconnects

The ideal transfer. To access the CPU data, GPU-accelerated engines typically transfer the input, in chunks, to the GPU for the actual processing, to avoid excessive, over-the-interconnect latencies in the critical path of execution and guarantee that the accessed memory during GPU execution is accessible [23]. In Figure 6.1, we plot, in grey, the query execution time for each SSB query (scale factor 100) when using the PT server of our experimetanl setup (Section 6.7). Despite the diverse selectivity of these queries, the execution time is relatively stable, especially when considering that the last three queries only see an increase in execution time because they transfer two more columns. In a utopic scenario, the analytical engine would only transfer the tuples that pass through all the conditions. We plot the time for transferring only those tuples in pink to make it visible. In the rest of this work, we discuss: (i) how the different interconnects and GPUs affect these results, and (ii) how different approaches for accessing the input data, including direct memory accesses over PCIe, and CPU-GPU coprocessing, enable Laconic to reduce the execution time significantly.

Data transfers and GPU (under-)utilization. GPU-accelerated analytical engines access the CPU-resident data via the interconnect, making it a central component and a critical factor for the performance profile of the engine: First, the interconnect bandwidth imposes a hard limit on analytical throughput because the engine must transfer the to-be-scanned columns to the GPU for any analytical processing. Second, interconnect utilization consumes memory bandwidth and serves as a resource-sharing metric between the CPU and GPU parts of the analytical engine.

Figure 6.2 presents the impact of the execution time for executing queries Q2.1, Q2.2, and Q2.3, which have the same query template and decreasing selectivities from Q2.1 to Q2.3, across server-grade GPUs with PCIe and NVLink as well as consumer-grade GPUs with PCIe. The dashed lines represent the time required for transferring the working set of the queries (9.2 GB) to the GPUs using different interconnect technologies, specifically PCIe (black) and NVLink (green). Based on this experiment, we observe the following properties hold for GPU-accelerated data management systems:

- **GPU Underutilization**. GPU-accelerated query processing is interconnect-bound. Figure 6.2 shows that the execution time for all queries is at most 11% higher than the transfer time that the interconnect requires for the given configuration. For PCIe-based configurations, upgrading the GPUs from consumer-grade GTX 1080 (320 GB/s) to server-grade V100 (900 GB/s) brings a marginal performance improvement of $0.5-2.7\%$. By contrast, upgrading the interconnect from PCIe to NVLink in a V100-based configuration yields a speedup of $3.94-5.42$. The interconnect bandwidth dictates the query execution time, whereas the processing capabilities of the GPU are underutilized, especially for slower interconnects. Recent work shows that even expensive queries are interconnect-bound when PCIe is used [23, 24, 94].
- **Selectivity-Insensitive Performance**. The size of the working set determines the execution time of the query. Figure 6.2 shows that the execution time of the query is roughly constant, even though the selectivity is decreasing. The observation is counter-intuitive: The expectation is that queries with more selective filters have a shorter time to result. While this expectation holds for CPU-based analytical systems which access CPU memory directly at a fine granularity, it does not hold for GPU-accelerated systems that eagerly transfer the whole working set of the query over the interconnect, one vector at a time.

These two properties of GPU-accelerated OLAP engines are undesirable. It indicates that the analytical engine fails to fully take advantage of the hardware and workload characteristics in the execution environment.

6.2 Laconic: Minimizing Data Transfers for CPU-GPU Analytics

We propose Laconic, a novel data fetching approach designed to optimize GPUs' effective input data access bandwidth for CPU-resident datasets. Laconic minimizes unnecessary data transfers over the interconnect to enable efficient GPU acceleration for analytics. Specifically, Laconic: (i) combines word-level data accesses and page transfers to lazily pull only the necessary data over the interconnect without introducing unnecessary data stalls, (ii) uses the coprocessing and the near-data nature of CPUs to further reduce over-the-interconnect data transfers, even for the first column, and (iii) packs to-be-transferred data to avoid overfetching.

Sparse GPU data pulling. In data-centric pipelined execution, operator sequences and expression conjunctions in filters and operator sequences are evaluated in a pipelined manner for each tuple, attributes that participate in expressions later in those sequences are less likely to be needed. For example, consider a filter followed by a join and then a projection. For each tuple, attributes partitipating in this sequence are needed with a probability that reduces as the attribute's first-seen position appears later into this sequence. Laconic uses this observation to judiciously select between fine-granularity pulling of input data to the GPU and proactive coarse-grained data pushing.

Prefiltering on the CPU. While the interconnect restricts the bandwidth available for accessing CPU-resident input data from the GPU, the CPU can access them at full memory bandwidth. Laconic pushes lightweight prefiltering operations to the CPU to reduce the data input. As a result, Laconic overcomes the traditional requirements of pull-based approaches that need to send at least the first column over the interconnect.

Densification. While sparseness on which input elements are required for the different steps of the query evaluation has the potential to reduce the data fetched to the GPU, it also causes overfetching: direct memory accesses fetch full interconnect lines and thus bring unnecessary data. Laconic selectively packs the input elements after the prefiltering steps to reduce overfetching and improve the interconnect bandwidth utilization.

6.3 Fine-granularity Accesses Without Regret

Eager transfers. Although the cost of transferring data is high, GPU-accelerated DBMS eagerly transfer data to GPUs before processing. Prefetching blocks of data has multiple advantages. First, it guarantees to the subsequent operations that the data are in an accessible memory, as depending on the hardware configuration, a GPU may not have direct access for reading data in any other memory location in the system. Second, the access pattern over the interconnect is sequential, and thus, it allows full utilization of the available bandwidth. Third, kernels access data stored in GPU memory and benefit from the high memory bandwidth and low memory latency as a result. Fourth, the copy of the data in the GPU memory can be used multiple times within the lifetime of a query. However, despite the benefits of asynchronously prefetching data to the GPU, the approach tends to over-fetch by transferring data that is not required. Query performance becomes insensitive to selectivity.

Lazy transfers. Modern GPUs have a unified address space and allow on-demand access to data in the CPU memory. The mechanism constitutes a lazy way of transferring data by pulling into the GPU and can bring significant benefits to GPU query processing. Primarily, it reduces the amount of transferred data for highly selective queries by leveraging the higher granularity of interconnect accesses. The kernels load columns accessed after the evaluation of predicates only partially. Furthermore, the memory footprint of query processing decreases because intermediate buffers are unnecessary. Finally, lazy accesses allow optimizations such as compression and invisible joins.

SemiLazy transfers. We extend our system to support a hybrid transfer strategy that enforces a transfer policy at the column level. We route, transfer, and process data in blocks, which are logical horizontal partitions. A data transfer operator copies the eagerly transferred columns of the block, or any columns inaccessible from the GPU, to GPU memory and forwards a pinned main memory pointer for the lazily transferred columns. Then, during execution, the GPU transparently reads the former from the GPU memory and the latter over the interconnect using the UVA. With the proposed design, we achieve more flexible reconstruction policies that deliver better performance than both pure eager and pure lazy transfer strategies because we optimize for each column judiciously, without adding extra complexity to the query engine design.

Summary and data transfers outlook. Still, even with these fine-granularity transfers, significant bandwidth may be used for transferring tuples that will be filtered out. Consider as an example the case of SSB Q3.4. While the query has an overall selectivity of 0.00000076 over the lineorder fact table [71], the engine has to transfer the first column to the GPU and do the first join probe to reduce the number of fetches for the rest of the columns. This means that even completely ignoring the cost of bringing the rest of the columns into the GPUs, the engine has to fetch 25% of the query input – more than six orders of magnitude more tuples than bringing only the qualifying ones.

The rest of this chapter, (i) mathematically models the expected gains from the above pull-based approaches and quantifies the impact of the "curse of the first column", (ii) describes how Laconic uses the near-data positioning of the CPU to prefilter the transferred data using bloom filters, and (iii) concludes with an approach that combines prefiltering lazy transfers to minimize the overfetching cost.

6.4 Modeling Pull-based Accesses

The presented pull-based methods, Lazy and SemiLazy, reduce the input size by relying on fine granularity accesses and query selectivity. However, often hardware components are optimized for workloads with temporal and spatial locality. Specifically, the over-the-interconnect access granularity, the cache lines, and memory management highly affect the performance of the two pull-based approaches. In the rest of this section, we model the performance of the proposed pull-based approaches.

Both approaches fully scan the first column to reduce the transferred volume of data. Then, they apply the first selective condition, e.g., a filter or a join-probe, before moving to the following condition. For simplicity, we will assume that the filter/joins are applied in a most-to-less selective sequence, although the discussion holds even when the query optimizer picks a different, e.g., more cache-friendly, operation order.

Moving to the next condition does not imply materialization: the data-parallel GPU design, combined with the high number of active contexts, allows the GPU to maintain multiple tuples in the – big relative to the CPU – GPU register files. Thus, all the above operations happen in a pipelined manner through operator fusion.

Then, using the result of the first evaluated condition, the GPU has to fetch the attribute required for the next condition. For example, consider the fact table A with foreign keys b1 and b2 referencing tables B1 and B2, respectively. First, attribute b1 will be requested by the GPU, representing the first column access, and, as a result, it will bring into the GPU, in chunks, the full first column. Depending on whether a Lazy or Semilazy approach is used, the engine will fetch the first column through direct accesses or an eager data transfer. Then, for the tuples that had a match in the join, the GPU will request attribute b2.

Ideally, the request for the second attribute would be as expensive as reading exactly the corresponding memory bytes from memory and nothing more. However, there are multiple hardware components between the GPU threads requesting the corresponding data and the actual CPU-resident data that affect this process. Specifically, the GPU thread is going to schedule the load instructions, which will, in turn, pass through the in-GPU cache hierarchy, then into the PCIe ports and links, to reach the CPU socket and then the CPU memory. GPUs like the NVIDIA V100 Tesla GPU of our experiments have multiple (32 B) sectors per L1 cache-line (128 B). As per NVIDIA, cache "management" [69] happens in 128 B. Then PCIe operations pass through the L2, and each read PCIe transaction has a 16 B header.

As lazily fetching the second attribute requires passing through all those layers, it's subject to overfetching and communication overheads. While the attribute may be only a few bytes, many of these hardware components operate at a different, potentially higher granularity (e.g., 32B sectors), causing extra data to be fetched. Similarly, communicating over the PCIe requires the header information, adding additional overhead. Furthermore, the GPU cache combines multiple load instructions into the minimal number of fetches when communicating between the cores and the cache with the L2. As a result, this propagates to the PCIe controlling mechanism, and thus consecutive reads are combined into a single one.

Overall, each requested data has an overhead similar to a traditional overfetch: requesting a specific byte brings a potentially bigger, fixed-size memory range, while two consecutive accesses to the same region will result in a single request, independently of whether the requests originate from the same or adjacent threads. Specifically, assuming an F bytes fetching granularity, requesting an attribute of size S bytes will result in fetching F bytes, while requesting multiple S-sized attributes that reside on the same F-sized region will also result in a single F-sized fetch.

Pessimistically assuming no temporal correlation between consecutive tuples and their first condition qualification, then if k tuples reside on each F-sized region and the first condition selectivity is s, then the probability of bringing an F-sized region from the CPU memory to the

GPU is:

$$p_{F-fetched} = 1 - (1 - s)^k. \tag{6.1}$$

Thus, in the running example, $P_{b2} = p_{F-fetched} N_{b2}$ (bytes) will be (over)fetched for column $b2$ as the GPU evaluates the query, where N_{b2} is the overall size of column $b2$ (in bytes). While the size F, to the best of our knowledge, is not directly documented by NVIDIA, we observe that our GPU uses $F = 128B$. This value aligns with the assumption that the request happens at cache-management-size granularity, and the read optimistically requests more bytes than actually needed to minimize the PCIe header overhead for the common case of sequential data fetches. Furthermore, the 128 B aligned reads do not cross TLB boundaries and thus create no invalid page faults.

The same pattern holds for consecutive lazy data fetches. Specifically, the effective data fetch size for column b_i is given by

$$P_{b_i} = \left(1 - \left(1 - \prod_{j=0}^{i-1} s_j\right)^{k_i}\right) N_{b_i}, \tag{6.2}$$

where s_j is the selectivity over the j-th fetched attribute (and $s_j = 1$ if that attribute is to be combined with another attribute before a condition) and $k_i = {}^{F_i}/_{S_i}$. Overall, $\sum P_{b_i}$ bytes will be transferred over the interconnect, resulting in a GPU execution time that is the maximum of the in-GPU time if all the data were GPU-reside and the time needed to transfer the $\sum P_{b_i}$ bytes over the interconnect.

Note that P_{b_0} is always the entire column: to reduce the data fetched over the interconnect, both pull-based approaches rely on accessing the first column. As a result, pull-based approaches have the "curse of the first column": even if they filter everything out after accessing the first column, e.g., because of a query with no qualifying tuples, counter-intuitively, they still have to transfer an entire column over the interconnect to benefit from GPU acceleration.

6.5 Heterogeneity to Overcome the Curse of the First Column

To overcome the curse of the first column and benefit from GPU acceleration, we use the hybrid CPU-GPU processing capabilities. Specifically, we exploit the near-data position of the CPU to push down pre-filtering operations: while the CPU's low memory bandwidth sets it into a disadvantageous position compared to GPUs, the CPU is not restricted by the interconnect in terms of input data access for data that are bigger than the GPU-memory capacity.

To reduce the over-the-interconnect data transfers, we push filters to the CPU side and introduce small bloom filters to filter data before the join. For the filter evaluation, the CPU can access data at memory bandwidth. Similarly, for the bloom filters, while the CPU may not be able to sustain its memory bandwidth in terms of throughput, small bloom filters can take advantage of the CPU cache and the local DRAM random access throughput.

As a result of the pre-filtering, tuples are filtered early, and only the ones that pass the prefiltering stages are transferred over the interconnect. Effectively, the accesses to the columns that participate only in the prefiltering are pushed to the CPU-side – before the interconnect. More importantly, P_{b_0}, the access to the first column, is executed on the CPU, releasing Laconic from the curse of the first column. In modeling terms, the filters that are pushed into the CPU use the CPU overfetching size instead of the over-the-interconnect overfetching parameter F. Furthermore, while for over-the-interconnect overfetching the time was driven by the interconnect bandwidth, in the CPU overfetching case, the corresponding bandwidth is the CPU memory bandwidth.

In contrast to CPU-only approaches, there is more cache memory available to keep the bloom filter cache-resident and performant. Traditional CPU-only approaches still share the caches and the random access bandwidth with the potential bloom filters and the rest of the operations like the join. Instead, Laconic only runs the first filters in the CPU. The rest are offloaded to the GPUs, so random accesses that would otherwise happen inside the CPU are now offloaded to the GPU, leaving the rest of the CPU memory bandwidth available for the bloom filters. Similarly, while joins highly benefit from in-cache hash tables and traditionally compete with the bloom filters for the cache, Laconic pushes those hash table probes to the GPU, releasing cache memory pressure.

Overall, pushing the first filters to the CPU side as lightweight (bloom) filters enables the GPU to overcome the limitation of having to transfer a full column. Still, there is significant overfetching over the interconnect even with the bloom filter.

6.6 Pushing The Overfetch Away

To reduce the last part of the redundant overfetching, Laconic provides two alternatives to the execution engine: either use the lazy access method to fetch the rest of the data or offload the input repacking to the CPU. For the latter, instead of lazily accessing all columns from the GPU, we combine the idea of SemiLazy with filter push downs to pre-filter and pack data before offoading the rest of the query to the GPU.

Specifically, we convert the (bloom) filter into an unpack-filter-pack operation: it scans the input, filters it using the (bloom) filter and then materializes its output into packed blocks. These blocks are then transferred to the GPU for the random-access-heavy operations, e.g., joins. Despite the CPU scanning and writing more data, the accesses are predictable, and,

depending on the amount of the repacked columns, the corresponding overhead is lower than doing the equivalent operation over the interconnect as the CPU has more input data bandwidth.

Lastly, as the bloom filter reduces the cost of performing the corresponding join, the in-GPU execution is also impacted. First, we rewrite the in-GPU plan to push the join(s) corresponding to the push-down operations later into the plan. Second, as the corresponding join will be of a lower cost, the GPU execution is further accelerated – a benefit of using the accelerator-level parallelism of the machine.

6.7 Evaluation

In this section we evaluate the impact of the interconnect for GPU-accelerated analytics. Section 6.7.1 evaluates the impact of different interconnects and GPU architectures. Section 6.7.2 compares the different Laconic's access methods and the accuracy of our modeling technique, showing how Laconic minimizes wasteful data transfers.

Hardware. To study the effects across consumer- and server-grade GPUs connected over relatively slow and fast interconnects, we executed our experiments on three different servers. The first and the second server have 2 x 12-core Intel Xeon Gold 5118 CPU (Skylake) clocked at 2.30 GHz with HyperThreads, that is, 48 logical threads and a total of 376 GB of DRAM. The first server has 2 x NVIDIA GeForce 1080 GPUs (consumer-grade), whereas the second one has 2 x NVIDIA Tesla V100 GPUs (server-grade) over PCIe 3 in both cases. The third server is based on IBM POWER9 equipped with 2x16-core SMT-4 CPUs clocked at 2.6 GHz, a total of 128 logical threads, and 512 GB of DRAM. Like the second server, it hosts NVIDIA Tesla V100 GPUs; however, it has four of them, and they are connected to CPUs over NVLink 2.0 interconnect with three links per GPU. For symmetry in all the experiments, we use 2 GPUs, one local to each CPU socket, even for the IBM server. In the rest of the section, we will refer to these servers as `PG` for the first one, `PT` for the second one and `NT` for the third one. We use the first letter, `P` and `N`, to signify the interconnect standard (PCIe/NVLink) and the second, `G` and `T`, for the GPU architecture (GeForce/Tesla). Currently, there is no consumer-grade GPU with support for NVLink.

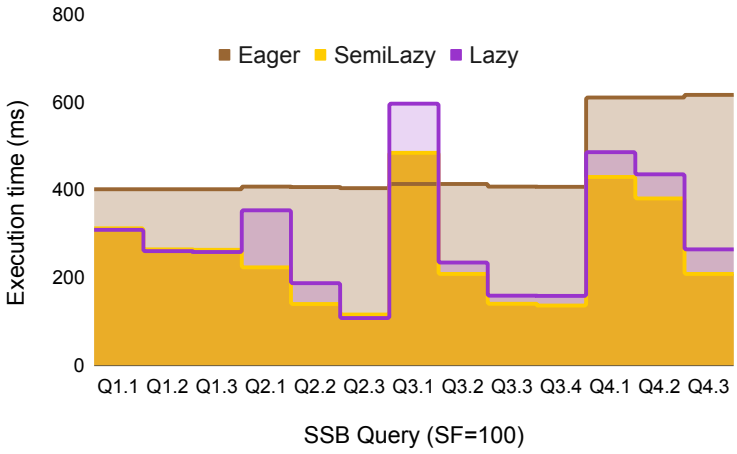
We use the three hardware configurations to show how the evolution of GPU architectures and interconnects affect the performance of the access methods. By comparing PG with PT, where we change only the GPU architecture, we observe how the increased GPU capabilities affect performance. Then, by comparing PT with NT, we keep the same GPUs, but we change the interconnect from PCIe to NVLink, thereby evaluating the effect of the interconnect on the query performance. Migration from Intel to POWER CPU architecture was necessary since NVLink is available as a CPU-GPU interconnect only on IBM POWER 8 & 9 servers, to the best of our knowledge.

Software. We implement our methods in our in-house code-generation-based CPU-GPU database engine, Proteus [23, 24]. Proteus uses the HetExchange framework for parallelizing query execution across heterogeneous devices. Lazy transfers are enabled by extending the mem-move operations to avoid transfers when the input is already in memory accessible by the target device (for the columns that should be lazily accessed). Furthermore, we hint to the compilation layer that data accesses to the various attributes should happen as late as possible so that there are no unnecessary data fetches. We rely on the per-thread software-based loop-interleaving and the GPU hardware to hide the increased latency introduced by the over-the-interconnect load operations. For all the experiments of this section, we use 2-GPUs, one per NUMA node, and force all main join operations to a GPU-only execution mode to isolate the benefits and challenges introduced by having to transfer CPU-resident data over the interconnect to benefit from GPU-acceleration. We evaluate our methods using the Star Schema Benchmark (SSB) [71] with a scale factor 100, so that we can compare against in-memory execution on GPU-resident data.

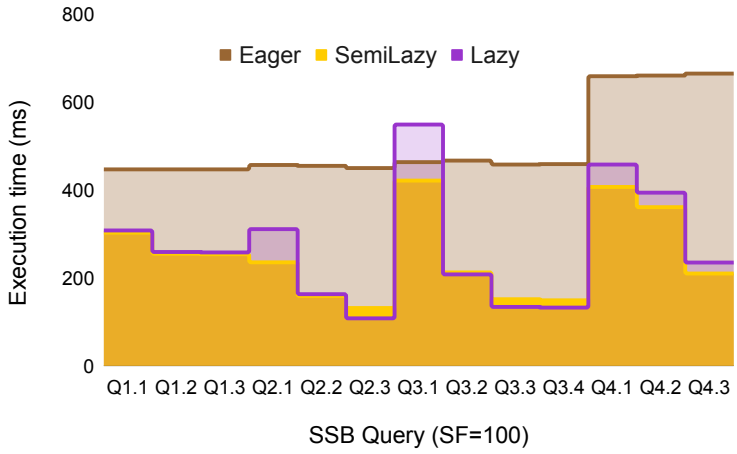
6.7.1 Pull-based Data Accesses

Methodology. Figure 6.3 evaluates the performance of different data transfer techniques across two axes: the interconnect characteristics and the query selectivity. First, we show how the interconnect bandwidth affects query execution and then evaluate the performance gain achieved by lazily accessing the input. Each query fetches the required data over the interconnect, and any transfer sharing or caching is disabled to simulate the case of reading fresh data from the CPU. *Eager* prefetches all the data to the GPU memory. In the *lazy* method, all data are accessed directly from the GPU threads during execution, without an explicit bulk copy. As a result, during kernel execution, GPU threads experience a higher latency during load instructions for input data compared to the eager method, where data are accessed from the local GPU memory. To reduce requests to remote data and the actual transferred volume, all read requests are pushed as high into the query plan as possible in the generated code. For the *SemiLazy* method, we prefetch the firstly accessed column of the fact table for each query and access the rest of the working set using the lazy approach. For query groups 2-4, the first column is the foreign key used in the first join, while for query group 1, the first column that filters the fact table before the first join, we prefetch the two columns that are used in the filter predicates.

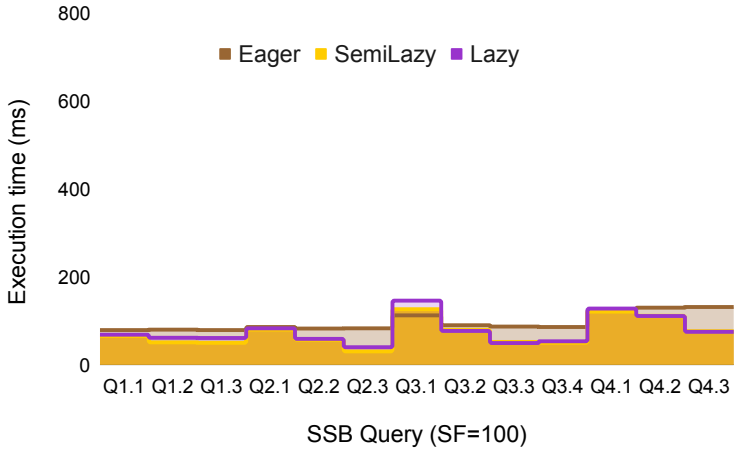
Eager access methods achieve throughput (working set size over execution time) very close to the interconnect bandwidth for all three hardware configurations. The only exception is Q3.1 on NT, which has the lowest selectivity and, combined with the multiple joins stressing the GPU caches, it causes high memory stalls. The bandwidth of the PCIe interconnect is low enough that these stalls have a minimal impact on the execution time, as they are hidden by the transfer time. In contrast, the ~5x higher bandwidth of NVLink 2 makes transfer times slower than kernel execution times, causing the stalls to become the new bottleneck. For all other queries, kernel execution is overlapped and hidden by data transfers.



(a) PG (consumer-grade GPU & PCIe 3)



(b) PT (server-grade GPU & PCIe 3)



(c) NT (server-grade GPU & NVLink 2.0)

Figure 6.3: Performance of the different access methods on SSB (SF=100) on the three different configurations.

Lazy access methods allow for reducing the transferred volume. SSB queries are highly selective, and thus performance improves in almost all queries, except for query 3.1. As Q3.1 is compute-heavy, its performance degrades due to the increased latency that materializes as memory stalls. Inside each query group, the benefit compared with the Eager method increases as queries become more selective.

SemiLazy improves upon Lazy by reducing the main penalty for lazy access methods. While laziness reduces the transferred data, it does so at the cost of higher memory latency. Nevertheless, some columns are accessed almost entirely. Prefetching those columns decreases the overhead of laziness at the expense of fetching extra tuples. The higher effect of SemiLazy is on the low-end GPU, where compute resources are limited compared to the server-grade GPUs. Additionally, while Lazy improves mostly the performance of very selective queries, SemiLazy access methods improve queries that are less selective and have multiple dependent operations, like Q2.1. The smaller latency of NVLink, compared to PCIe, reduces the impact of this method.

Summary. Eager methods reduce memory stalls during execution and allow more opportunities for caching data transferred to the GPUs. However, eager transfers unnecessarily move data and penalize query execution on selective queries. On the other hand, Lazy access methods reduce the execution time of highly selective queries by avoiding unnecessary transfers at the expense of memory stalls and data dependencies which are combined with partially transferred columns that impede column reusability. SemiLazy adapts between the two to achieve the best of both worlds by using a different method for each column: if a query touches most of a column, it will be prefetched eagerly, while columns that are only accessed depending on evaluated conditions are lazily accessed. This results in SemiLazy adapting between the two methods and improving the performance of queries that are selective but also compute-intensive.

6.7.2 Cooperative Data Accesses

Modeling Pull-based Accesses. Figure 6.4 plots the execution time of SemiLazy for the different SSB queries against the execution time predicted by the proposed model. The prediction closely follows the actual execution time. However, its error is constantly one-sided due to two reasons. First, the proposed model assumes that no access to the same overfetched line happens twice. In practice, this only holds when threads do not diverge, or they diverge and soon thereafter reconverge, causing part of the misprediction. Second, the proposed model assumes no start-up cost and that the actual processing is not affected by the data access. However, over-the-interconnect data fetches introduce additional latency that by itself can cause a slight slowdown and thus affect the prediction of the model.

Prefiltering: Repacking versus Lazy Reconstruction. Figure 6.5 shows the execution time for the SSB queries for the push-based approach. We plot two variants of this push-based approach: combining the pushdowns with i) eager repacking (solid bars) and ii) lazy data

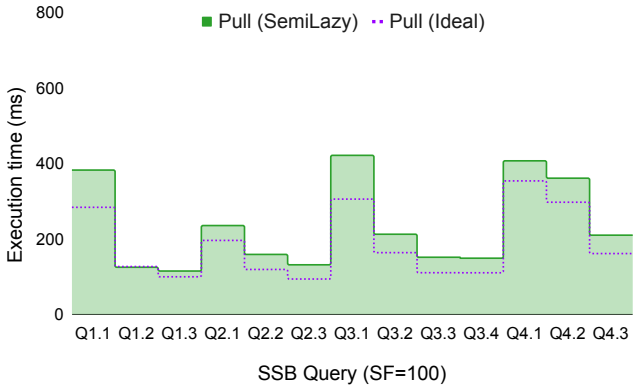


Figure 6.4: Comparison of SemiLazy to the interconnect modeling ("ideal")

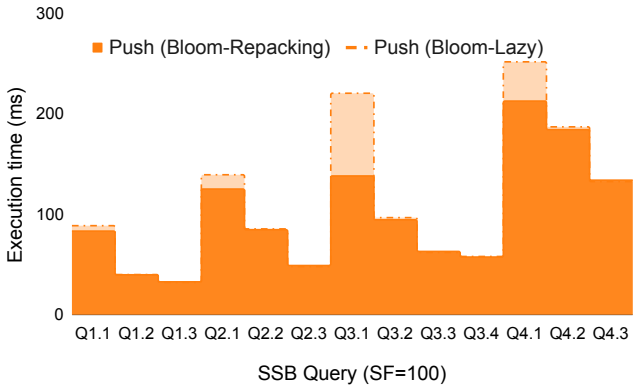


Figure 6.5: Bloom filter push-down with and without repacking

fetching (dashed outline bars). We push fact table filters, when applicable, and a bloom filter for the first join to the CPU for the pre-filtering. We use a single-hashing bloom filter to reduce the CPU overhead, and we manually select the filter size that provides the best overall execution time for each configuration. For most queries, the high selectivity of the SSB queries makes the difference minimal. However, for queries that are less selective on the first condition, like the first query of each group, eagerly repacking has a substantial impact. Specifically, by eagerly repacking after the first column, Q3.1 sees a 1.6x speedup, as 1) the overall query selectivity is distributed across the first two joins (20% selectivity each), while the third one drops very few tuples (85.7% selectivity), 2) the selectivity of the first condition is not significant to avoid the majority of the F-sized line accesses: for $F = 128B$ and $4B$ items, there is only 0.1% chance to skip an F-sized line for the second column, without repacking. In contrast, for the same query, repacking allows for sending only 20% of each transmitted column over the interconnect.

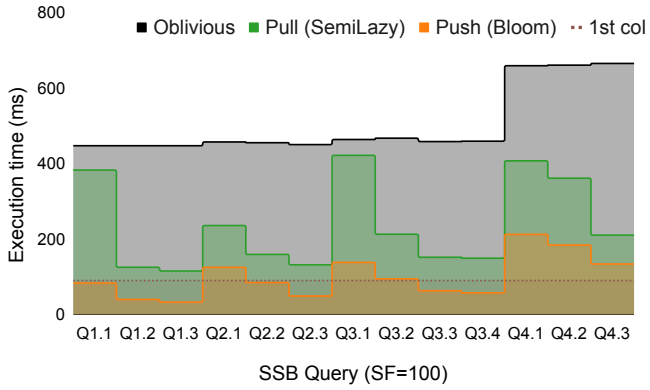


Figure 6.6: Push- vs Pull-based data accesses on SSB SF=100

Push- versus Pull-based Access Methods. Figure 6.6 compares the push- and pull-based methods against the data-oblivious eager transfers and the time required to eagerly transfer the first column. Both the push- and pull-based approaches outperform the eager data transfer as they exploit the query selectivity. However, the pull-based approach is 1) slower than the push-based alternative as it does not benefit from the pre-interconnect data reduction, and 2) the performance of the pull-based approach cannot exceed the "1st col" line, as it needs to fetch the first data column to select which elements to fetch for the second one. In contrast, the push-based approach exploits the high selectivity of the SSB queries to outperform even sending a single column over the interconnect for $^{7}/_{13}$ queries.

6.8 Conclusion

This chapter shows that the interconnect bandwidth significantly limits analytical GPU performance. Further, it shows that using fine-granularity data fetches over the interconnect reduce the data volume for selective queries. Still, there is overhead caused by data overfetching, which we model and build Laconic to overcome it. Laconic uses the CPU to prefilter and pack data before sending them to the GPU for the actual processing – essentially treating the CPU as a lightweight near-data processor. We show that prefiltering CPU-resident data before sending them to the GPUs results in up to 4.6x speed-up versus the GPU pulling the data during query execution – showcasing the importance of workload-aware data transfer methods over workload-oblivious ones.

7 Inter-server: Rack-Scale Analytics through Accelerator-Level Parallelism

Recent advances in hardware accelerators and networking technologies have rendered servers with high-bandwidth networks and Graphics Processing Units (GPUs) mainstream. With an order of magnitude higher internal memory bandwidth than CPUs [41], GPUs provide increased analytical throughput, even in the presence of random accesses [43, 58, 83, 94]. As a result, analytical engines can benefit from offloading complex operations to GPUs, even at the expense of sequential data transfers over the relatively slow PCIe bus [18, 23, 35]. Similarly, high-bandwidth networks coupled with Remote Direct Memory Access (RDMA) offer fast, low-latency data transfers across servers with minimal CPU overhead, allowing CPU-based analytics to expand their memory capacity and scale across multiple servers [5, 55, 85]. However, despite GPU and networking advances, existing analytical engines fall short in efficiently combining accelerators at rack scale.

Traditionally, analytical engines relied on the multi-core CPUs and the memory bandwidth available from multiple machines by combining scale-up and rack-scale execution. The standard deployment setup considers that the CPU is the sole compute unit type in every machine and, hence, the CPU is responsible for all operations on the data. As a result, state-of-the-art CPU-only rack-scale engines optimize the communication channels using RDMA [12, 56] while relying on the symmetry of CPU memories to handle intra-server NUMA effects through work-stealing [84].

However, GPUs question the CPU-only deployments as they (i) provide additional processing power and memory bandwidth, introducing non-uniformity inside the servers, and (ii) are placed behind slow interconnects, further amplifying NUMA effects. Scale-up multi-CPU-multi-GPU deployments rely on data locality and timely, globally available, relative device throughput information to maximize the interconnect utilization [18, 23, 24, 58, 83]. As a result, CPUs are no longer in the critical path of every data processing operation, and the bottleneck shifts to the CPU-GPU interconnect.

Rack-scale multi-CPU-multi-GPU deployments bring new challenges compared to both scale-out CPU-only and scale-up multi-CPU-multi-GPU ones. First, the asymmetry of the memory hierarchy in scale-up deployments is further amplified from the network connection, questioning the fundamental assumption of the symmetric memory hierarchy in scale-out CPU-only approaches. Second, the relative device throughput information derived and leveraged in scale-up deployments cannot be aggregated timely and with low overhead at rack scale. Given the above challenges, existing works (i) cannot fully benefit from GPU acceleration due to the implicit assumption of homogeneous intra-server resources [12, 56, 84] or are bound to single-server execution by relying on a shared global accelerator load view [23]; (ii) focus on GPU-only execution and do not consider the high memory capacity and data locality that CPUs can offer [28, 53]; (iii) statically shuffle the data through the CPU [54], causing interconnect and CPU memory bandwidth interference; (iv) split processing based on tasks [97] executed in coarse-grain stages on either CPU-only or GPU-only mode, resulting in high sensitivity to workload, execution time predictions, data locality, and skew.

This chapter proposes RuSH, an analytical engine that enables scalable analytics through CPU-GPU rack-scale acceleration. RuSH employs a dataflow execution model to overcome the inherent processing skew and scalability challenges raised by hardware heterogeneity. It decomposes the parallelization operations to enable load balancing while holding data transfers until the target device is known, thereby avoiding unnecessary data transfers. Overall, RuSH enables analytical engines to exploit the heterogeneous CPU-GPU hardware of modern rack-scale deployments.

In summary, RuSH makes the following **contributions**:

- We identify that the inherent processing skew and reduced processor visibility in multi-server CPU-GPU deployments result in poor scalability of analytical engines in rack-scale deployments (Section 7.1).
- We propose RuSH, a novel design (Section 7.2) that overcomes the scalability limitations and achieves fast execution, effective orchestration, and transparent resource management of CPU-GPU racks (Section 7.3).
- We describe the key challenges of orchestrating the CPU-GPU hardware in a multi-server setup (Section 7.4), and we propose composable orchestration (Section 7.5) and network management mechanisms (Section 7.6).
- We integrate RuSH in our in-house DBMS engine (Section 7.7), and we demonstrate scaling over InfiniBand, where CPU-GPU systems achieve up to 1.8x speedup against CPU-only counterparts by exploiting the direct GPU-to-network and fast intra-server interconnects (Section 7.8).

Overall, our design enables analytical engines to avoid excessive data transfers and static offloading decisions that cause wasteful hardware utilization. Instead, our proposal efficiently exploits the available CPU-GPU hardware in rack-scale RDMA-enabled configurations by orchestrating the device and data transfers based on data locality, query, and relative device throughput.

7.1 Scalability of CPU-GPU Analytics

The scalability of GPU-accelerated engines is addressed either in scale-up or scale-out deployments. GPU-accelerated scale-up engines go beyond the limitations of traditional, CPU-only architectures, but they are limited to single-server setups, and specifically: 1) they rely on shared memory, 2) they require a global view of the device utilization for load-balancing, or 3) their buffer management is static, forbidding inter-server load-balance, or dynamic but requires expensive, low-latency coordination that is prohibitive across servers. On the other hand, scale-out engines underutilize the available hardware, and specifically: 1) they use only a single type of processor per server, 2) they decouple inter- from intra-server execution at the expense of intermediary data transfers that are wasteful for internally heterogeneous servers, or 3) they are unable to load balance across the multitude of devices.

Analytics on modern CPUs. CPUs have been the main processing unit for analytical engines. To exploit their modern capabilities, analytical engines have long employed techniques like code generation [63], vectorization [15, 77], parallelization [25, 30, 51], prefetching [21, 78, 79], and combinations thereof [32, 60, 93]. Still, while state-of-the-art CPU analytical engines are highly optimized for the modern CPU features, the server hardware landscape evolves towards heterogeneous CPUs [26, 34], CPUs made of chiplets [62], and hardware accelerators [38, 65].

GPU-accelerated scale-up analytics. In addition, recent work has explored the benefits of running analytics on GPUs to exploit their high parallelism [18, 23, 43, 61, 76, 91, 99, 102]. However, whether the CPUs or GPUs are more appropriate for each query depends on multiple factors, including the initial data placement, the system topology, and the query itself. As a result, recent work proposes partitioning algorithmic steps across the devices [58, 83, 94, 96] as well as orchestrating the heterogeneous devices for whole-query CPU-GPU coprocessing [7, 23]. The latter inspects the processing throughput of the different devices to route tasks based on the relative device processing throughputs. However, such approaches require: 1) global view of device utilization to effectively route the tasks despite the processor heterogeneity, 2) centralized resource managers to satisfy the resource requirements and exchanges of the different devices, thus limiting them to single-server configurations.

Scale-out analytical engines. To overcome the scalability limitations of a single server, analytical engines have scaled out to multiple servers by introducing parallelization meta-operators into the query plan [30]. However, the advent of high-bandwidth networks introduced new challenges and opportunities [5, 12, 98, 106]. Liu et al. [56] propose a data shuffling operator that encapsulates the complexities of managing the low-level queues and buffer management

required for achieving the bandwidth offered by the hardware. Rödiger et al. [84] decompose inter-server and intra-server parallelization through a two stage-parallelization strategy: intra-server, they use morsel-driven parallelism to exploit the multi-core CPUs but, inter-server, they use a traditional exchange-based [30] parallelization strategy. While the proposed two-stage parallelization strategy reduces the communication overheads and the required number of active connections, it is prone to execution skew as remote servers have reduced observability of the load of remote cores and sockets. Specifically, they distribute incoming data equally across the homogeneous CPU sockets in a round-robin, skew-oblivious, manner and any execution skew is resolved through inter-socket data accesses over the UPI/QPI during the intra-server execution load balancing. However, with GPUs, 1) the processor heterogeneity introduces inherent execution skew as different devices operate at different rates, 2) given the high interconnect bandwidth in existing servers, accessing data during rebalancing results in significant memory bandwidth interference – one of the scarcest resources during analytical query processing. As a result, existing analytical engine designs cannot efficiently use the available accelerator-level parallelism offered by GPUs.

Summary. While analytical engines can efficiently combine scale-up and scale-out approaches to use internally-homogeneous servers, GPUs accelerators break the fundamental assumption of processing uniformity inside each server. Exploiting the analytical processing capabilities of GPUs can provide significant acceleration opportunities; however, existing analytical engines have to select between efficiently using a single server's CPU-GPU hardware or scaling to multiple servers but being prone to processing skew.

7.2 RuSH: Rack-Scale Hybrid Analytics

We propose RuSH, a novel in-memory analytical engine design that achieves efficient query execution over heterogeneous CPU-GPU hardware without compromising scalability. RuSH uses multiple GPUs to achieve high analytical processing throughput and CPUs to allow fast near-data processing of in-memory data. To avoid wasted processing throughput, RuSH: (i) orchestrates execution to offload tasks to devices based on the overall throughput gains, (ii) increases the offloading opportunities using the high-bandwidth networking, and (iii) decouples path and data binding from task scheduling to allow scalable orchestration despite the heterogeneity and limited memory resources.

Fast analytics. To achieve fast analytics, RuSH considers three key dimensions that affect the overall response time: processing throughput, data access throughput, and scheduling flexibility. By combining CPUs, GPUs, and high-bandwidth NICs, we innovate across these three dimensions to improve the overall performance.

Orchestration. RuSH coordinates all the involved devices and optimizes data transfers across them to reach the maximum performance of the processing resources, despite the data path complexity. RuSH represents execution as a data flow to build an execution graph that

allows load awareness and composable load-balancings. It combines dataflow execution with a decomposed version of the parallelization operations to allow multi-stage scheduling decisions as well as direct data transfers to the final data consumer.

Resource management. The increased NUMA effects of CPU-GPU hardware and the limited memory capacity of GPUs amplify the memory pressure and scalability issues of rack-scale buffer management. RuSH improves the scalability of the buffer management through tight integration to the decomposed parallelization operators. As a result, RuSH minimizes the remote buffer allocations without compromising the affinity of the allocated buffers or creating scheduling artifacts due to suboptimal allocations.

7.3 GPUs and RDMA in Modern Racks

This section describes the usefulness of the CPU, GPU, and RDMA technologies in rack-scale analytics. Further, it explains why their combination provides a unique set of tools that we leverage to achieve scalable CPU-GPU analytics.

7.3.1 Performance Bottlenecks in Analytics

The data-intensive and parallel nature of analytics make their performance susceptible to three system parameters.

Requirement #1: Analytical processing throughput. Optimizing the throughput of operations like joins, aggregations, and sorting has gained a lot of focus [9, 10, 13, 28, 42, 43, 48, 58, 78, 89, 90, 94, 96] as their irregular access patterns highly affect the overall query response time. Such methods optimize memory accesses to reduce unnecessary data stalls and off-chip memory accesses using the caches and memory-level parallelism. Still, the hardware diversity results in a significant performance difference depending on the target device.

Requirement #2: Data access throughput. Besides quickly accessing hash tables and other intermediary data structures, processors also need to sustain a corresponding bandwidth to access the input data – otherwise, waiting for the input data wastes the quick to intermediary ones. The disaggregated nature of memory resources in multi-server and multi-device (CPUs/GPUs) setups increases the importance of input memory bandwidth. In traditional in-memory databases, the input data are placed near the CPUs, the only processing unit. However, with faster but lower capacity tiers and devices (i.e., GPUs), different devices observe different input access bandwidths.

Requirement #3: Scheduling flexibility and offloading. Both input and processing imbalances can cause underutilization and skew for static work assignments. The imbalances are exacerbated by the heterogeneity of processing resources and memory disaggregation while modeling the interplay of multiple, heterogeneous devices can be challenging [58].

For analytical engines to achieve efficient execution, they need to handle imbalances by flexibly scheduling the analytical processing across the available resources, considering the corresponding offloading overheads.

The three pitfalls of one-processor-rules-them-all analytics. While CPUs have long powered analytical engines, three hardware trends necessitate the transition to hardware-accelerated engines designed to efficiently and cooperatively combine CPUs with GPUs and RDMA: First, CPUs have a low ratio of random-access-throughput to interconnect-bandwidth: As we increase the number of random accesses performed in CPU memory per cache line read sequentially over the interconnect, the limited CPU memory bandwidth quickly makes the random accesses the bottleneck, despite the random versus sequential access pattern. Thus, CPUs are often unable to catch up with the incoming data rate and results in poor scalability, as additional machines cause unnecessary shuffling [84]. Second, CPU-only engines assume that sending data to a remote processing unit has a performance penalty. However, access bandwidths to remote and local devices have greatly improved, flattening the hierarchy. Therefore, the distance between the processing unit and the data has shrunk, and the computational capacity has increased; thus, the cost of what was considered expensive shuffles is significantly reduced. Third, CPU-only designs assume uniform processing throughput per CPU, which does not hold in the presence of hardware accelerators that exhibit significantly different performance profiles than CPUs. On the other hand, GPU-only engines prefer to store data in the GPU memory to exploit its high bandwidth. However, the GPU memory has a much smaller capacity than the CPU memory, limiting such approaches' applicability. As a result, there is no single solution matching every possible case. Instead, data locality needs to be considered to achieve efficient analytics, essentially rendering the CPU a near-data processor. The rest of this section analyzes the benefits of different accelerators and how they address the above challenges.

7.3.2 Optimizing for the Performance Bottlenecks

Modern CPUs, GPUs and RDMA-enabled NICs provide a unique arsenal for reducing the impact of the above challenges on analytical query response times; however, each comes with its limitations.

GPUs: high analytical throughput. With approximately an order of magnitude higher memory bandwidth than CPUs and a design optimized for data-parallel applications, server-grade GPUs outperform CPUs for most analytical experiments as long as the data fits in the GPU memory [23, 102]. However, in the general case, data do not fit in the GPU memory. Then, the data have to be stored in the CPU memory, providing fast access by the CPUs, but GPUs access them over the interconnect [83]. As a result, whether the CPU or the GPU is faster is query specific and depends on the relative performance of the two as well as the interconnect bandwidth [23, 58, 73, 83].

CPUs: the new near-data processor. Traditionally, CPUs have been the main processing unit for analytical engines. In the presence of GPUs and RDMA-enabled NICs, their main benefits for data-parallel analytical tasks stem from their position near DRAM, as they have fast and fine-grained access to the in-memory storage. On the other hand, while PCIe attached devices can still access CPU memory at subpage granularity, their bandwidth is limited by the interconnect. As a result, in the presence of hardware accelerators, CPUs are turning into powerful near-data processors for in-memory analytical engines.

NICs: fast and flexible interconnectivity. Traditionally, the CPU was placed in the center of query execution, and all incoming data passed through its memory, requiring at least one extra transfer to offload any incoming data to the accelerators. Instead, RDMA not only enables zero-copy and CPU-free data transfers but also allows direct transfers between PCIe devices like GPUs. Thus, incoming data can be written to either a PCIe-enabled accelerator or to the CPU. Furthermore, the same interconnect technology used for CPU-to-GPU connections is also used for GPU-to-NIC and CPU-to-NIC, thus ending up with the same communication bandwidth across the devices, enabling new load-balancing and offloading opportunities.

GPUs-NICs-CPUs: a complementary set. While device heterogeneity increases the design spectrum and requires careful consideration of the role of every device in query execution, each of the devices comes with its own benefits: GPUs provide high analytical throughput, NICs allow remote devices to be accessed with bandwidths comparable to local devices, and CPUs provide fast access to local, in-memory data. Furthermore, combining all the devices provides constructive interference: hybrid execution can combine the best of each device to accelerate analytical query processing by load balancing based on the query and data placement at hand.

7.4 Coordination in CPU-GPU Racks

This section describes the challenges that prevent existing engines from exploiting the advantages of both CPUs and GPUs without the scalability limitations of operating on a single server.

Challenge #1: Processing skew is the rule, not the exception, under heterogeneity. Different devices have different execution characteristics and response times. Furthermore, their relative performance depends not only on the hardware characteristics but also on the task-at-hand and runtime factors such as the current data placement [23, 44]. Thus, to achieve efficient device utilization, aka minimal idle device time, tasks need to be distributed across the devices unevenly, in such a way that minimizes the overall response time. To effectively route tasks across the heterogeneous resources, scale-up engines rely either on cost estimation [44] or on routing operations having a global view of the device load [23, 49, 58]. However, exploiting the accelerator-level parallelism requires multi-server analytics to handle intra-server heterogeneity both when processing local tasks and during offloading to remote servers.

Challenge #2: Intra- and inter-server execution orthogonality causes CPU interference, under heterogeneity. Load rebalancing operations required for decoupling inter- and intra-server execution become prohibitive in the presence of heterogeneity. In the case of internally homogeneous servers, assigning tasks in a round-robin manner across the server sockets inside each server is sufficient to make intra-server rebalancing the exception, allowing improved scalability by delaying intra-server load-balancing into a second independent step [84]. Specifically, given the low occurrence of rebalancing operations, Rödiger et al. [84] show that this two-step process is overall beneficial as the scalability benefits outweigh the overall little rebalancing cost. However, as heterogeneous hardware creates intra-server processing imbalances, rebalancing operations become commonplace, resulting in round-robin solutions requiring frequent data transfers across the interconnects to compensate for the multi-step task handling.

Rebalancing operations are expensive as they use interconnect bandwidth, which is a scarce resource. The advent of high bandwidth networks amplifies their performance penalty: with PCIe 4.0, up to five 16-lane PCIe slots per CPU socket, and each NIC/GPU saturating a PCIe 4.0 x16 slot, simply staging data through an AMD Epyc 7413's CPU memory can take more than 60% of the available CPU memory bandwidth, effectively starving data-intensive CPU tasks, and that is using each PCIe link just in a single direction.

Challenge #3: Increased network buffers pressure due to diverse paths and limited memory, under heterogeneity. Each data transfer across two machines requires at least two buffers to be active for the duration of the data transfer: the source buffer in the sender and the target buffer in the receiver. Concurrent transfers require multiple buffers to be active on each side simultaneously. Furthermore, even if a transfer finishes, the buffers can not be released back to the engine until the corresponding notification arrives, which can be delayed, e.g., due to notification batching.

In internally homogeneous servers, the NUMA node of the target buffer was of little importance, as rebalancing was less frequent and the relative penalty small. As a result, existing approaches were sharing the buffer pool of each server across multiple connections, assigning a limited number of buffers to handle a big group of incoming data streams – the routing of those buffers to the corresponding processors was without any data movement. Device heterogeneity, however, i) amplifies the NUMA effects due to the interconnects and the variety in memory bandwidth across CPU and GPU memories, ii) requires different incoming rates for each input and target device stream to operate optimally, as even when the server-inbound data are transmitted at line rate, inside the server, they should be distributed across the devices based on their relative performance, and iii) low memory capacity devices (GPUs) have a low number of GPU-resident network buffers, despite the high-bandwidth network needing only a few seconds to write to their full memory. Thus, data shuffling must carefully manage the limited network buffers per device, without exhausting the device memory and while supporting a variety of offloading rates.

Summary. GPUs offer both additional analytical processing power to CPUs but also processing skew across the devices and diverse capabilities that cause different queries to exhibit different acceleration benefits. As a result, efficient and scalable analytical engines must i) efficiently load balance tasks across the devices, despite the server boundaries, ii) minimize unnecessary interference caused by the orchestration mechanisms, and iii) efficiently manage the limited system buffers, despite the diversification of the buffer types.

7.5 Composable CPU-GPU Orchestation

RuSH overcomes the communication and coordination challenges imposed by heterogeneity by representing query execution as a dataflow execution graph, combined with a separation of concerns across the OLAP components. Specifically, RuSH is based on three key observations: i) bulk data transfers have a sequential access pattern that, based on the current interconnect and memory bandwidths, often challenges the status-quo of preferring local data processing, ii) the bulk processing throughput dictates whether a local device is overloaded or not, and iii) using asynchronous bounded queues of pending tasks allows composable load-balancing.

7.5.1 Dataflow for Heterogeneous Processing

RuSH uses a dataflow execution model to enable load-balancing across the CPU-GPU device of multiple servers. Specifically, the query plan is treated as a dataflow graph [3, 20, 29, 40, 103, 104], composed of multiple pipelines that are connected to each other. Tasks flow through the pipelines: a task enters a pipeline and results in the pipeline producing zero, one or more output tasks, depending on the operations in the pipeline, that enter the next pipeline in the graph.

Each pipeline is instantiated multiple times, creating multiple flows. Each instance runs in a specific machine and processing unit. Routing points control the exchange of tasks across pipeline instances and, due to the flow-centric execution, they can differentiate between fast and slow paths, even if the stagnation happens in a later part of the flow. To support different execution sub plans for CPUs and GPUs, we allow the query execution plan to diverge from a traditional tree structure into a directed graph with potentially different paths for each compute target, similar to prior work [23, 76]. The paths may converge, even in the middle of the plan, and split again, for example, to reshuffle tasks across devices.

Each node pushes its output to the next one, one at a time, but the output can be a full rowset (block-at-a-time execution mode), similarly to vectorized execution [15]. However, operators may as well output single tuples through the same interface, as long as the next operator supports it. The query plan leaves are segmenters [23]: special nodes that output the available input chunks – essentially a set of rowsets. Relational operators, like join and aggregations, logically operate in a tuple-at-a-time mode; however, as we fuse consecutive operators during code generation, the result is tight loops between (un)packing [23, 60] and pipeline breaking

points [27, 63]. Except for very low-cardinality cases, the execution plans transmit a block-at-a-time near the device-crossing points to amortize communication overheads. Note, though, that passing a block across two operators does not imply a data movement: this is delayed until an actual mem-move operator [23]. Similarly, data access to the contents of a block is delayed until there is an unpack operation.

We instantiate each path multiple times, with each instance creating a logical execution flow, similarly to a pipeline. The number of instances depends on each path's degree of worker parallelism and uses routing points to merge and split control flows, similarly to HetExchange. Each flow passes through a specific sequence of devices; for example, a flow may be passing through CPU1 of server 3, followed by GPU0 of server 3, CPU1 of server 3, but it would not be possible for the same flow to pass conditionally from GPU0; instead, for conditional use of the GPU, a different flow would be created and a routing point before the flows would be responsible for selecting the corresponding path. This flow *linearity* enables RuSH to estimate the relative throughput of each flow. Thus in routing points, it can distinguish between logically-equivalent flows (flows that apply the same operations) but physically distinct ones (flows that use different device sequences) that exhibit different performance.

Asynchronous operations for overlapping and parallelization. To exploit the available parallelism and hide system latencies, all operations that cross-device or server boundaries operate asynchronously: when a routing point receives a task from one of its input flows, it will route it to one of its outputs and immediately return control to its input flow, without waiting for the task to be picked up by the output flow that will consume it. As a result, multiple output tasks of a routing point may be pending as well as run in parallel: the routing point receives the input, places it into a queue, and returns, while another thread of the routing point is responsible for handling the rest of the tasks' work. Furthermore, this asynchronicity allows RuSH to hide latency that would otherwise be incurred for waiting for tasks to be sent over the network, scheduled into devices (e.g., GPU kernel launch), and completed.

Bounded queues for composable load-balancing. RuSH bounds asynchronicity to enable composable load-balancing across the servers and despite the hardware heterogeneity or reduced load visibility across servers. Specifically, by having infinite queues, RuSH would invalidate the flow linearity benefits as every asynchronous point would look like an infinite throughput flow, resulting in no visible difference between fast and slow paths. To avoid that, RuSH uses bounded queues in all asynchronous points: the operation is asynchronous as long as the number of pending tasks does not exceed a specific threshold – if it does, then the operation stalls until a free slot is created. This way, the input flow is frozen, and the processor can be used for another flow if any. Even more importantly, if a flow stalls, its input flow will eventually also have to stop sending tasks into the stalled flow, as the queue in between will fill up. Similarly, if all devices inside a server become overwhelmed, all the flows will stall and the queues towards that server will also fill up, resulting in a composable load-balancing beyond the server boundaries: if a server has to slow down due to the load, this information is implicitly captured by the number of pending tasks in the queues towards that server.

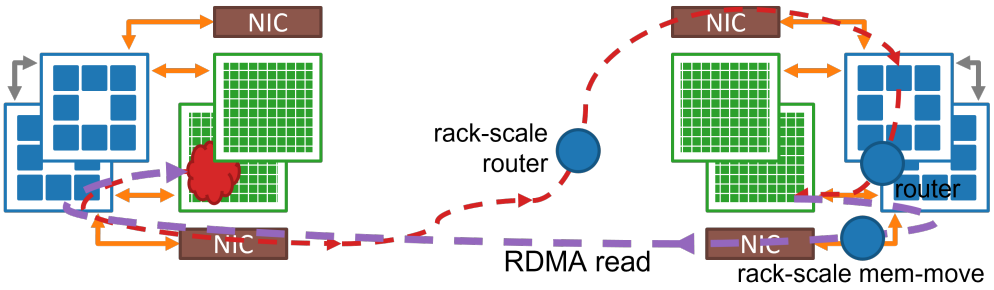


Figure 7.1: The operators orchestrating execution of two servers. Cloud: a data chunk generated on one of the GPUs. Red & purple arrows: a control message and a data fetch.

Implicit and explicit resource sharing. Each server hosts multiple devices; thus, multiple flows share the same resources, such as an inbound NIC connection. This can result in the processors behind the NIC having available cycles, the network being saturated, and still, none of the ingress flows to the server to seem congested as they underutilize the bandwidth, creating a scheduling issue. Our design aligns the queues with the hardware resources to propagate backpressure not only for explicitly used resources but also implicitly shared ones like interconnects. Specifically, with servers pulling their input, the queue of pending tasks for the operations pulling the data will fill up and notify its input that it is too busy to handle tasks, pushing backpressure below the inter-server routing.

7.5.2 Delayed Data Transfers

Decoupling data- and control-flow. For efficient execution, data should be offloaded to a remote device only when they are going to be processed faster than in the local device. Such a decision requires knowing: (i) whether the data-local device has enough processing capacity to consume them and, (ii) which one is the target device. The former depends on hardware characteristics, data and operations to be performed, while the latter depends on the load of remote devices and the interconnects in-between.

A traditional approach would keep track of the load of each device in the system and perform global optimization to decide where to offload the task. Global optimization requires every server to report the data chunks that it consumes. However, as the internal bandwidth of each server can reach up to terabytes per second, communication on the data chunk granularity quickly introduces unnecessary monitoring and synchronization: instead of globally optimizing based on how exactly each server consumes its tasks, we only need to know if a server has the processing capacity to accept tasks from another server and, e.g., details on how it processed its own tasks are unnecessary. Thus, we split the decision-making into local decisions with the observation that devices behind an interconnect can be abstracted as a single device for the scope of deciding whether to offload the task – enabling RuSH to reduce

the monitoring requirements. Specifically, contrary to a global optimizer, RuSH 1) only sends peer-to-peer messages, 2) only for offloaded tasks, and 3) it sends no inter-node notifications for tasks consumed in their initial node.

We enable local decisions while avoiding excessive transfers by decoupling the control-flow from the data-flow to allow delayed data fetching. Specifically, we create a three step-decision process that aligns each decision with the hardware boundaries: First, a *rack-scale router* operator decides whether a data chunk should be consumed on the local machine or whether it should be sent to a remote one, similar to an Exchange [30] operator. Then, the rack-scale router on the target server redirects the task to a local *router* [23] that selects a local compute unit for consuming the task. If the task requires data transfers, the router on the target server redirects the task to a *rack-scale mem-move* operator that fetches data directly from the source device to the consuming one. Finally, while load-based task-offloading is an interesting case, the rack-scale router has various policies to decide the offload target, including locality-based policies to prioritize local processing and hash-based policies for partition-based shuffling.

The rack-scale router keeps track of how many tasks it has offloaded to remote servers as well as to the local machine. Further, it collects completion notifications from remote rack-scale routers about tasks it offloaded. These notifications are used for load-balancing: when deciding where to offload a task, it avoids machines that have too many uncompleted tasks. Thus, machines with fewer uncompleted tasks represent faster machines with respect to the current server, regardless of whether they are faster because the rest are busier or have fewer processing resources. Essentially, the amount of offloaded but uncompleted tasks represents the throughput that a given machine can achieve by offloading tasks since instead of measuring throughput in terms of line bandwidth, and it encapsulates the load of work and how appropriate the remote server is for the offloaded tasks. Finally, each rack-scale router receives tasks from remote instances, and hence it is also responsible for notifying them as soon as it pushes a task to the next operator.

In contrast to the Exchange operator [30], the separation into multiple steps allows fine-grained decisions, with each part of the decision-taking place on a different server, allowing access to server-local information about interconnects and device utilization. Furthermore, while HetExchange [23] follows a similar control- and data-flow separation, our multi-stage decision process breaks the dependency of the HetExchange's router operator to a global, inter-server state. Finally, while Rödiger et al. [84] also separate intra-node and inter-node routing decisions, they rely on the processor uniformity inside each server to prepare to receive buffers in a round-robin manner across the NUMA nodes and use work-stealing to handle intra-server load imbalance. Instead, RuSH uses the router to decide the next device/NUMA node receiving the data based on runtime information.

For example, Figure 7.1 shows two servers, each with two NICs (brown boxes) and two GPUs (green boxes). A plan that uses both servers would contain a rack-scale router to move the control flow across the two servers, followed by a router to orchestrate the execution internally

to each server and a rack-scale mem-move to transfer data across the servers. Each of the two router operators would be instantiated once per server, while the rack-scale mem-move would be instantiated, in each server, as many times as the parallelism dictated by the router before it. The rack-scale router is depicted in the middle of the figure, as the two instances would be interconnected and exchange task descriptions. Assume now that the left-hand side server decides to offload a task to the second server, and the data corresponding to that task are loaded on the GPU memory (red cloud). First, it will send a message to the rack-scale router instance on the other server, with the description of the task and the references to the corresponding data chunks, but without moving the data chunks themselves. The instance on the right-hand side server will receive the task and push it to the local router. If the router is too busy with pending tasks, it will block waiting for the server to complete enough, incurring back-pressure to the rack-scale router, which would eventually propagate to the remote server's router. As soon as the router gets a free slot, it will receive the task from the rack-scale router, and the latter will notify its counterpart on the other side that the task is completed. Then, the router will inspect the intra-server load and decide on a device to offload the task, pushing it to the corresponding mem-move. If the mem-move is busy, it will stall the router, generating the aforementioned back-pressure towards the remote server. Otherwise, it will trigger an asynchronous RDMA read to fetch the data directly from the left-hand side GPU to the right-hand side one. As soon as RDMA reads complete, mem-move will push the task to the next operator, possibly a CPU-to-GPU operator, to transfer execution to the GPU and consume the task.

Data movement. In contrast to traditional architectures that try to minimize data movements, the memory and interconnect ratios generate data flows from the main, in-memory, CPU-resident, storage to other devices in the system. With the inter-server interconnects achieving similar bandwidth as intra-server ones, GPU-accelerated analytical engines are often called to efficiently transfer the data across servers so that queries can use remote devices as if they are local.

To encapsulate the data transfers across servers, we use the rack-scale mem-move operator. The rack-scale mem-move is responsible for overlapping data transfers as well as picking the most appropriate data path to transfer the data. Specifically, when a rack-scale mem-move receives a task that requires data chunks from a remote server, the operator decides which NICs to use to fetch the data based on the system topology: The router dictates the target device, and the mem-move finds the closest, in terms of interconnect bandwidth, NIC that has access to the target device memory. Then, it spawns the transfer and registers itself for a notification completion, as described in Section 7.6. While waiting for the notification, rack-scale mem-move continuous to the next input entry to allow multiple on-the-fly transfers and thus achieve better network bandwidth utilization. For load-balancing purposes, however, it limits the number of pending notifications for each instance, thus creating back-pressure to the previous operators and causing work to shift towards alternative paths, if the operator becomes overloaded. As soon as a completeness notification arrives, rack-scale mem-move pushes the now local data chunks to the next operator.

Data pulling and task pushing. Operations that pull data perform read operations: they allocate a local buffer and pull the remote data. Then they are free to release the source staging area. Operations that push data are seen in two variants: the data broadcasting operation, which multicasts the source buffer to multiple other machines and thus it has to allocate remote staging areas, and the task offloading operations that communicate offloading requests to remote machines through writes to a predetermined remote ring-buffer and thus only require remote memory allocation during the opening of a pipeline.

7.5.3 Buffer Allocations and Message Directionality

The high-bandwidth networking combined with the limited memory of accelerators means that staging areas are a scarce resource. With 48 GB GPU memory and ~12 GB/s of network bandwidth available to each GPU, the GPU memory can only withstand 3 seconds of ingress traffic without consuming it. Considering that under the general case the same memory is used for holding hash tables and other data structures, with a 10% allocation of the memory space to staging, the ingress time is already down to 0.3 seconds. A naive implementation that partitions this region across the connected servers would drop this number down to 100 ms even with a cluster of 4 machines. If, on the other hand, we partition these buffers across the devices of the cluster, with four devices per node, we get 25 ms. Meaning that careful staging area management is needed to hide irregularities in ingress traffic and avoid idle time on the critical system resources.

Data-flow through one-sided RDMA operations to control target buffer. Network management and maintaining multiple on-the-fly operations introduce overhead to both the receiver and the sender side. To reduce the overhead, rack-scale mem-move relies on a one-sided RDMA read operation to bypass the remote CPU and directly read from the remote CPU/GPU memory to the local one. Furthermore, while big data chunks and fully materializing stages simplify communication, they harden load-balancing and impose memory capacity restrictions which are amplified by the scarce memory on the accelerator. Thus, RuSH relies on data chunks that are small enough to allow load-balancing and require only partial materialization by the stages before the data exchange, but also big enough to amortize the communication overheads. In our implementation, we inject partially materializing operators that generate rowsets in a columnar format (containing only the required columns), in chunks of approximately a huge page (although the exact size depends on the relative size of the attributes). Thus, each task arriving at the rack-scale mem-move is a set of MB-sized chunks, over which the rack-scale mem-move amortizes the network management cost.

The task arriving at a rack-scale mem-move contains a data handle and a source server for each of them. When the rack-scale mem-move sees that a block is on a remote server, it spawns an RDMA read operation and puts a handle that keeps track of the operation into the task descriptions. The task description is then placed into the queue that resides between the producer and consumer part of the rack-scale mem-move. The producer is then allowed to

return to its caller to generate the next task. The consumer pulls tasks from the queue and waits for their completion. If when the producer tried to place a task into the queue and the queue is full, then it stalls to create backpressure for the linearity described above.

Regardless of the overlapping degree, however, architectural choices in current hardware may limit the bandwidth available for some paths. To overcome such limitations, rack-scale mem-move provides two alternative operation modes for transfers to/from GPUs: (i) direct transfer mode that transfers data directly to the target device and, (ii) hybrid transfer mode that uses intermediary hops to recover lost bandwidth due to the GPU-direct accesses. The direct transfer mode minimizes the interference to CPU memory, and it's the optimal one if the NIC-to-GPU communication can use the full bandwidth. Nevertheless, the hardware [68], like some CPU IOH, limits the available bandwidth, in which case rack-scale mem-move reverts into hybrid transfer mode. In hybrid transfer mode, RuSH takes advantage of the throughput-oriented execution nature of analytical tasks: it sends part of the traffic through the CPU memory, introducing an intermediary staging area and additional latency, albeit achieving the NIC-to-GPU transfers at the bandwidth of NIC-to-CPU ones by overlapping the CPU-to-GPU transfer step with the next NIC-to-CPU data transfer.

While one-to-one data transfers are handled as late as possible to improve load balancing, broadcast operations, e.g., for a broadcast-based join, know a-priori the target machines. Thus we improve the network usage by performing broadcasts eagerly. Specifically, we send the broadcasted data once to each machine and let a consecutive local operation broadcast them across the machine devices later on. While sending the data, the rack-scale broadcast invokes one-sided RDMA write operations to each machine to write the data without interrupting the remote CPUs. For all but GPU-only execution, the broadcast performs the RDMA write to the remote memory with the highest source-memory-to-network-to-memory bandwidth, which is the CPU memory in all our configurations. As the normal data transfers, the broadcast operation also performs the operations asynchronously to allow queuing consecutive operations and hiding the network latency.

Back to Figure 7.1, if the rack-scale mem-move supports up to two on-the-fly operations, then all routers will send the tasks to the same rack-scale mem-move. Therefore, there will be three tasks originating from the left-hand side server. The first two tasks will arrive at the rack-scale mem-move, which will trigger the RDMA reads to read the corresponding data chunks from the remote memory directly to the local GPU memory without the intervention of the remote CPU. If the third task reaches the rack-scale mem-move before any of the transfers are complete, then the mem-move will hold the tasks and wait for a transfer to complete, before returning to the router by signaling the oversubscription and applying back-pressure to the previous operators, essentially notifying them that they should preferably pick a different path. As soon as all RDMA reads for a task are complete, the mem-move will push the task to the next operator, replacing the remote data chunk handles with the corresponding local

ones. Furthermore, it will release the ownership of the remote data chunks back to the remote server, and it will continue with the chunks for the third task. If any of the chunks are already local, it will skip the remote data transfers, and it will do the local data transfers if any.

Control-flow through two-sided send/recv communication to avoid remote buffer allocations. RuSH minimizes the remote buffer allocations that are required for control-flow operations using two-sided communication. In contrast to data-intensive memory moves, task offloading by itself only transmits a short description containing the task & connection identifiers and the data chunk handles, but not the data itself. Furthermore, control messages are always be handled by the CPUs, and due to their small message size, receiving the message to a CPU socket different from the one that will process the offload requests has a negligible performance impact compared to the overheads of sending many small messages for buffer allocations and message notifications. RuSH exploits the insensitivity of the offload message affinity to offload message receival to the network manager: control-messages are sent and received using two-sided communication, with the network manager preparing a receive buffer for such messages – as a result, buffer allocation for receiving tasks is done locally on the receiving end.

7.6 Inter-server Infrastructure

To support the high-level operations described in Section 7.5, we build a network manager tailored to the GPU and InfiniBand requirements for high bandwidth data access. Specifically, both GPUs and InfiniBand NICs require memory to be registered and page-locked to allow efficient access and address translations across the different units. To reduce the registration overhead and allow efficient data transfers, despite the uncertainty of which data chunks will be transferred internally or outside the server, we co-design the networking infrastructure with an accelerator-oriented block manager that provides access to transfer-optimized memory.

When PCIe devices communicate between them or access CPU memory, they require: (i) that the memory is page-locked so that it is not, for example, swapped out without the knowledge of the accessing device and, (ii) in the case of a virtual address, they require the translation from the virtual to the physical memory for during access. To guarantee the first one as well as prepare any relevant accelerator-specific metadata, both GPUs and InfiniBand NICs either require pre-registering accessed memory or more recently, allow potentially more expensive, on-demand registration and paging. For example, NVIDIA CUDA allows allocating pre-registered memory or registering pre-allocated memory. Similarly, InfiniBand verbs provide an interface to register memory and retrieve an access key that remote devices use to access local memory.

The delayed decision making about data transfers conflicts with the requirement for placing to-be-transferred data without unnecessary copies into memory which is optimized for the transfers, since the information about whether and where the data will be transferred is unknown at (partial) materialization time. To overcome this limitation while avoiding costly

registrations in the critical path, we maintain a block pool per NUMA node (including GPU NUMA nodes) that serves as a memory pool of memory registered with both the GPUs and the NICs. Furthermore, for InfiniBand registrations, the corresponding keys are distributed to the participating OLAP engine instances. Maintaining this memory allows operations to write in any block allocated from this pool without any registration overhead to get memory that will allow full bandwidth transfers independently of whether they are inter- or intra-node.

Operations like the RDMA write performed during the broadcast require remote memory that is also pre-registered. While an operation could fetch such a handle, the network manager periodically prefetches blocks from remote nodes to serve operations with remote block handles without incurring a network round-trip. Specifically, for each remote NUMA node, the network manager maintains a pool of remote blocks that, from the perspective of the remote machine, appear allocated, but the local network manager can directly provide them to local operators requiring remote blocks. As our current implementation never requires a remote GPU block, we only prefetch CPU-resident blocks to reduce GPU memory pressure.

Finally, our data-flow design has multiple rack-scale routing operators active at the same time, requiring a varying number of connections per query and time step. To avoid the connection setup and managing overheads, we instantiate one connection per pair of NIC devices and virtualize multiple flows over the same connection. For example, if a plan has three rack-scale routers requiring communication between NIC A and B, all the communications would happen across the same InfiniBand connection, and the network manager will multiplex them to both batch notifications and avoid creating unnecessary short-lived connections.

7.7 System

We implement RuSH in Proteus, a code-generation-based DBMS. To integrate RuSH, we introduced the new operators, the networking manager and update the block management infrastructure to register the blocks with the GPUs and InfiniBand.

Code generation. Proteus uses LLVM to generate code: the operators generate LLVM IR, calling into CPU and GPU backends for low-level operations that are common for both device types but specialized differently for each of them. The final LLVM IR for both the CPU and the GPU is passed through the compiler passes of LLVM and then CPU code is directly compiled to binary and loaded for execution, while GPU code is converted to PTX and passed through the CUDA driver API for the final compilation steps, before loading it for execution. The generated operators call interface with the network manager by calling into C++ code. Specifically, RuSH's meta-operators generate thin interfaces that extract or invoke the offloaded tasks and then call into generic C++ code as there is little specialization for those – except for injecting the routing policies. Then, the meta-operators' C++ code calls into the managers to execute the actions described above.

Placement into the system architecture. RuSH optimized the runtime performance of the analytical engine. To that end, it adds the infrastructure for scalable analytics through the code generation modules for the new operators, the network manager. RuSH plugs into the block manager as an additional registration call to register the transfer-optimized blocks with the InfiniBand and GPU runtimes. The new operators are currently placed into the plan through simple heuristic plan shapers exactly above the segmenters and after aggregation steps; however, our design is compatible with existing approaches for placing the operators during query optimization. Lastly, RuSH installs a routine in the Proteus to listen to remote query invocations instead of waiting for commands from the default local query planner and the corresponding call on the arrival of a new query into a node.

7.8 Evaluation

Experimental setup. To evaluate RuSH, we integrate it in Proteus, an analytical engine that uses LLVM to generate code for the available CPUs and GPUs. The experiments use a four-machine cluster interconnected through InfiniBand EDR. Each machine has two Intel Xeon Gold 5118 CPUs @ 2.30GHz, with 12 physical cores per socket. Each CPU socket has an NVIDIA Tesla V100S GPU with 32 GBs of memory and a dual-port Mellanox MT27800 ConnectX-5 EDR InfiniBand NIC, both connected to the local CPU through a dedicated PCIe 3.0 x16 link, totaling two GPUs, four 4x EDR links per server and 376 GB CPU memory. While the NICs are dual-ports, the second port is built for redundancy as the maximum bandwidth each NIC can achieve is limited by its PCIe 3.0 x16 link to 12.5 GB/s, effectively the same as a single port switch. Furthermore, one of the machines uses NVIDIA Tesla V100 GPUs instead of V100S.

Methodology. Similar to prior work on GPU-accelerated analytics [18, 23, 102], we use the Star Schema Benchmark [71] (SSB) as well as scan-heavy aggregation queries to evaluate RuSH against CPU-only and GPU-only execution, both in data shuffling, potential acceleration, scalability as well as the adaptiveness of rack-scale hybrid execution across the different configurations. For all the SSB experiments, we use scale factor (SF) 1000, and 4-byte columns – generating a total of 24 GB of binary data per column. We do not employ any compression; all data are warmed-up in CPU memories before the first measurement and equally distributed, in a collocated manner, across the servers. All plans use broadcast-based joins, with shared hash-tables across the CPU cores, similar to morsel-driven parallelism [51]. Parallelization across the devices happens, inter-server, through the rack-scale router, and intra-server through simple routers. For the CPU- and GPU-only baselines, we use the same system and prune the unnecessary paths during the instantiation of the plan. Specifically, for CPU-only execution, we prune the device crossing operators and all routers consider only CPUs. Similarly, for GPU-only execution, we force all relational operators to execute on the GPUs, by adding a device crossing operator below the first relational operator. During code generation, we specialize each pipeline to its target device, similar to HetExchange [23] our code generation specializes each pipeline to the target device. While our implementation supports zero-copy caching of the transferred input data to the target machine, we disable it to force each query to

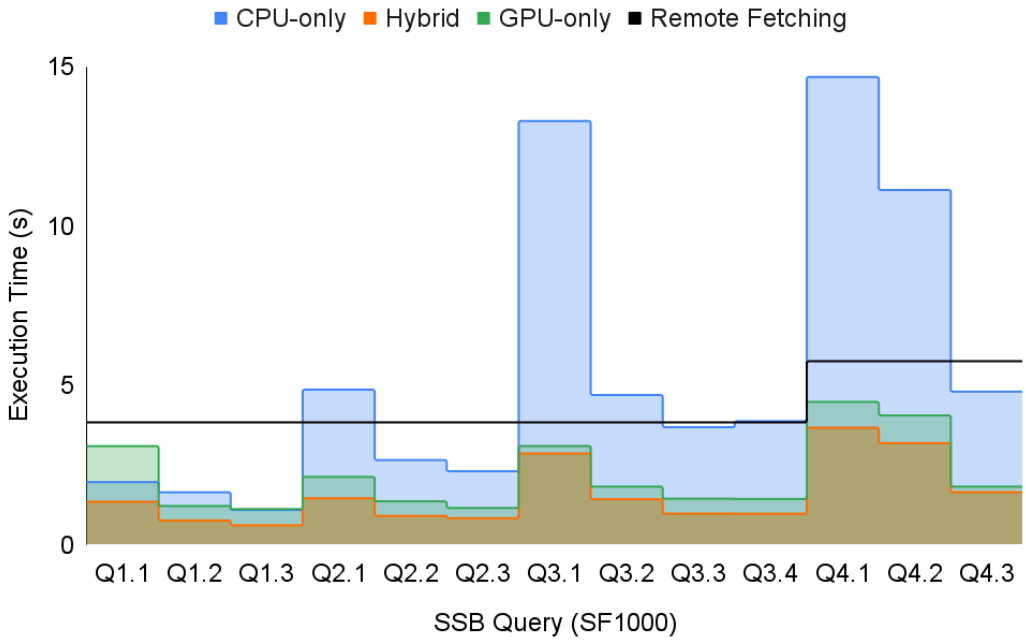


Figure 7.2: Execution time vs network bandwidth

transfer its data. We report the average execution time of prepared plans across five iterations. As a baseline, except if otherwise mentioned, we use HetExchange [23] augmented with lazy CPU-to-GPU transfers [83] and Morsel-driven [51] parallelism for the in-CPU workers of HetExchange.

7.8.1 Evaluating the CPUs-GPUs-NICs Interplay

GPUs. To illustrate the GPU potential, Figure 7.2 shows for how many queries each of the CPU, GPU, and Hybrid execution is faster than sending the data over the network (black line). For all configurations, we preloaded all data in the CPU memory. In all cases, the engine is NUMA-aware, and while it prefers local data consumption, it load-balances to send data out of an overloaded device, when applicable. The CPU is faster than the network only for SSB query group 1 and the very selective ones. For queries with lower selectivity and more join probes (accounting for the selectivity of the first joins that reduce the probes perform for subsequent joins), CPU-only execution takes more time than transferring the input data over the network. Instead, GPU and hybrid execution are faster than sending the data over the network.

NICs. Figure 7.3 shows the theoretical maximum access bandwidth achieved by different CPU, GPU, and Hybrid execution configurations, for data fetched from (i) CPU memory, (ii) GPU memory, (iii) data evenly distributed across the two memories, (iv) data placement that maximizes input bandwidth for each case, and (v) over remote data fetching over the network. For simplicity, the reported access bandwidths assume a standard configuration with

Data	Execution model (GB/s) CPU	Hybrid	GPU	Capacity
CPU	**100**	**100**	12.5	1-10 TB
GPU	12.5	**1000**	**1000**	16-80 GB
C+G (50%-50%)	22.2	**200**	24.7	<160 GB
C+G (Per-Config Opt)	100	**1100**	1000	varies
Remote	**12.5**	**12.5**	**12.5**	Inf

Figure 7.3: Bandwidths for various data placements and execution models, assuming PCIe 3 x16 and InfiniBand EDR 4x

one GPU and one NIC per CPU socket, assuming 1TB/s GPU memory bandwidth, 100GB/s CPU memory bandwidth, and 12.5GB/s per-device interconnect and networking bandwidth – similarly to the Intel Xeon Skylake CPU, NVIDIA V100S GPU, and NVIDIA Mellanox EDR NIC of our main cluster. All numbers are reported per CPU socket. For newer hardware generations, similar values (multiplied by a factor) hold at the moment: PCIe 4 has double the bandwidth of PCIe 3, NVIDIA A100 has double the GPU memory bandwidth, HDR InfiniBand doubles the EDR bandwidth, and Intel Xeon Ice Lake reaches almost 190GB/s.

Capacity-wise, CPU memory can hold up to a few TBs of data. In contrast, existing GPUs can host a few 10s of GBs, with NVIDIA A100 and H100 GPUs providing configurations with 80 GB device memory. The evenly partitioned data configurations are capped to 160 GB – double the GPU memory, by definition. While such a configuration improves the access bandwidth for both GPU-only and Hybrid execution compared to CPU-resident datasets, it is still slower than accessing GPU-resident data for Hybrid execution: the CPU consumes only 10% of its data by the time the GPU is done with its local data. Then, the execution waits for the CPU to finish and even pulling data to the GPUs will not improve the bandwidth as the memory bus is the bottleneck. Instead, tuning the percentage of the data distribution based on the execution model (fourth line) provides the highest access bandwidth for all three execution methods: CPU-only execution benefits from CPU-resident data, GPU-only execution from GPU-resident data, and hybrid execution from distributing the data reversely proportional to the access bandwidth of each device to its local memory. However, each of these configurations provides a different capacity: CPU-only and GPU-only are capped by the CPU and GPU memory respectively, while Hybrid is capped by the GPU memory plus the 10%, as the CPU has approximately an order of magnitude less bandwidth to its local memory than the GPU to its local memory.

Over-the-network data fetches are performed with almost the same bandwidth for both CPU and GPU accesses. In addition, for device-local data, the GPU outperforms the CPU by almost an order of magnitude additional bandwidth, while hybrid execution outperforms both by combining the different, independent, memory bandwidths. Essentially, from a throughput perspective, in a well-balanced, in terms of accelerators, modern server, it is better

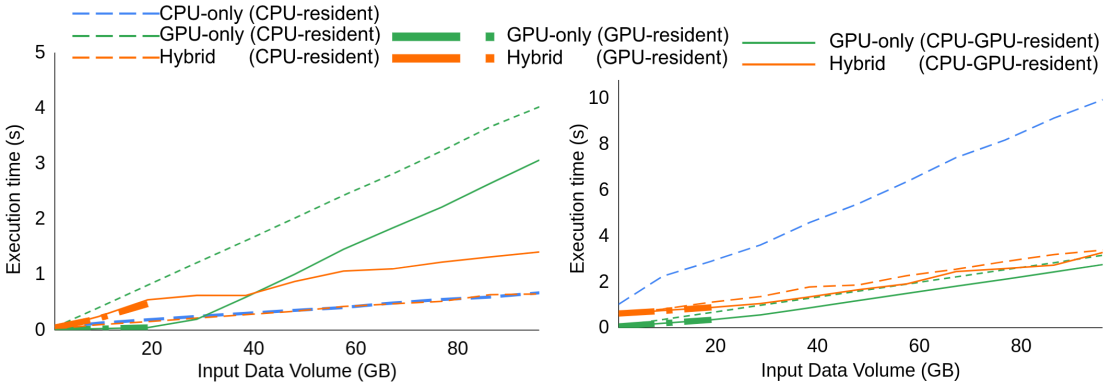


Figure 7.4: Various execution methods and placements: i) an aggregation across 4 columns (left), ii) a 3-join query (right).

for incoming remote data to reach the GPU that has the high memory bandwidth, than the slower-memory CPU which only serves as a higher capacity memory tier for large intermediate results.

CPUs. Figure 7.4 shows how the query complexity and data placement affect the effectiveness of CPU-only and GPU-only configurations. Specifically, we run a scan-and-sum query over 4 SSB columns (Figure 7.4.i) and a 3-join query (SSB Q3.1, Figure 7.4.ii), for three data placement configurations: CPU-resident data (dashed lines), GPU-resident data (thick dash-dotted lines) and CPU-GPU-resident data (solid lines). For the latter, we split the dataset horizontally and place up to 24 GB of data into each GPU and the rest, if any, on CPU memory. CPU-resident data provide the highest access bandwidth, with only two exceptions: (i) the capacity-limited, GPU-resident data case and, (ii) the CPU-GPU-resident case where the benefit depends on the fraction of the data that fit in the GPU memory; naturally, as the dataset size increases, the GPU fraction becomes smaller, and the benefits over CPU-resident datasets diminish. While for the simple scan-and-sum query CPU-only execution has a consistent performance near the optimal regardless of the input size, with three joins the overhead of random accesses causes up to 2.4x slowdown versus hybrid execution.

7.8.2 Data Shuffling and Saturating the Network

Methodology. In Figure 7.5, we use the Star Schema Benchmark and force each server to shuffle its local data to a specific server, to evaluate: i) the potential of each of the execution modes to saturate the network bandwidth for shuffling-heavy queries, ii) whether any execution mode is slower than the network bandwidth and, iii) the potential of RuSH to overlap data transfers with execution. Specifically, we distribute the input data equally across the servers, and provide a query plan that makes each server receive data from exactly one other server. For all queries, the plan contains a local pre-aggregation per (CPU core/GPU device) after the joins and the final result is collected at a single server for the final aggregation.

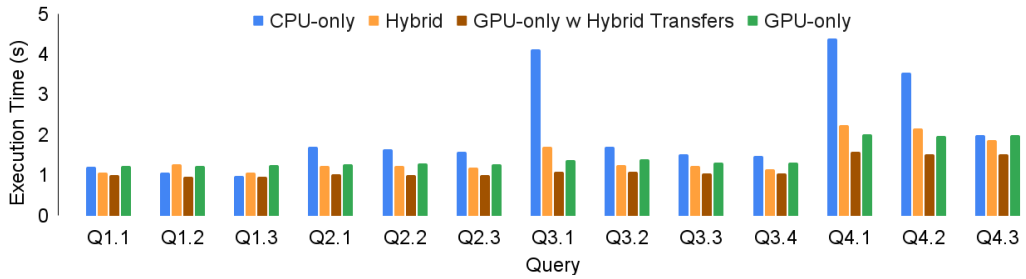


Figure 7.5: SSB (SF1000) with forced fact table shuffling to introduce network pressure.

The rack-scale router handles inter-server control transfers, and the rack-scale mem-moves handle the data transfers. As we force the rack-scale router to send its data to a specific node, load-balancing across servers is effectively disabled; nevertheless, inside each server, the local router load balances across the available devices: in the CPU-only and GPU-only execution modes, the router load-balancing across the CPU cores and the GPUs, respectively, while for Hybrid execution, the router has to load balance across both CPUs and GPUs. In all cases, the rack-scale mem-moves have to consider the NUMA locality (including whether the memory node is on a CPU or GPU) to transfer the data based on the intra-server load-balancing decisions. Given that the working set for each of the Q1.1-Q3.4 is 96 GB and for Q4.1-Q4.3, it is 144 GB, we have 8 EDR links (2 links per server). As we force remote, eager data transfers, given the optimal transfer rate of ~12.5 GB/s×8=100 GB/s, the minimal execution time is the time to transfer the data over the line. That is, 0.96 s and 1.44 s for queries of groups 1-3 and 4, respectively.

CPU-only execution achieves near-line-speed execution times for most of the queries, but not all, as shown by the blue bars of Figure 7.5. For queries Q3.1, Q4.1 and Q4.2, CPU-only execution is bottlenecked by the random accesses performed by the join: while there is enough network bandwidth to send/receive data, the join causes the CPU to have a throughput lower than the incoming data throughput when probing the hash-tables with tuples from the fact table. In contrast, their more selective counterparts (Q3.2-4 and Q4.3) join the same tables but due to the higher selectivity of the first joins they prune random accesses required by the latest joins reducing the memory bus load. Essentially, these three queries would benefit from additional CPU memory bandwidth, but keeping the aggregated CPU memory bandwidth constant and adding additional network links would provide little benefit. For the rest of the queries, CPU-only execution saturates the network bandwidth, becoming bottlenecked by incoming data transfer rates. There is, however, a small deviation from the line-speed rates in multiple queries, including queries of groups 3 and 4. This deviation is caused by the pipelines used to build the hash-tables for the joins: while the input data for the hash-tables are orders of magnitude smaller than the big fact-table, when scaling to 4 servers, the building time becomes non-negligible.

GPU-only execution (green) has a consistent behavior with respect to the network bandwidth across the SSB SF1000 queries; nevertheless, it's still far from optimal: CPU-to-GPU transfers do not always saturate the EDR 4x links due to IOH limitations. Overall, in queries Q1.1-Q3.4, GPU-only execution achieves a throughput (input data over execution time) of 8.7-9.6 GB/s per EDR link, similar to the Linux RDMA perftest utility. Q4.1-Q4.3 sees a lower throughput (6 GB/s) due to the increased overhead of build phases and broadcasts happening between the GPUs for the hash-table allocation and creation.

To overcome the bandwidth limitation imposed by the IOH of our hardware, we allow GPU-only execution with Hybrid data transfers: instead of using GPU-direct transfers, the rack-scale mem-move stages data through the CPU memory. This allows the full network bandwidth that the NIC-CPU connection allows, with GPU's reliability in overlapping computations with network transfers – resulting in the best of both worlds.

Lastly, with Hybrid execution, RuSH is responsible for not only transferring the data across the nodes and load-balancing between homogeneous resources but also selecting the most appropriate execution units (rate of tasks consumed by CPUs versus GPUs) on each server. As in all the queries of Figure 7.5, executing the query at network line speed is the optimal execution time, Hybrid would have the optimal performance if it achieves the same execution time as GPU-only with Hybrid transfers. In all the queries, Hybrid achieves execution times similar to GPU-only execution; nevertheless, it sees some small performance penalties due to i) having to build hash-tables on the CPU-side, which is slower than building them only on the GPU-side, causing most of the Hybrid queries to the aforementioned penalty and, ii) an additional overhead in queries where the CPU-only execution is significantly slower than GPU-only. When CPU-only execution is significantly slower than GPU-only execution, then tasks scheduled to the CPU can cause long tails in execution times, hardening the load-balancing task of the routers. Nevertheless, even for queries, like Q3.1, Q4.1, and Q4.2, where the CPU is significantly slower, Hybrid execution sees only slightly slower execution than GPU-only, despite the additional load-balancing complexity it handles.

7.8.3 Locality-aware Rack-scale Execution

Methodology. In Figure 7.6, we use the Star Schema Benchmark and, this time, allow RuSH to load balance based on the workload and data locality, to evaluate: i) the performance of each execution model on a rack-scale environment, ii) the orchestration decisions and, iii) the capabilities of the load-balancing and direct transfers employed by Hybrid execution to appropriately combine the different device types. Similar to the previous experiment, we pre-load the data into CPU memory before the first query execution. The optimal execution for the current setup, however, given that the data are equidistributed and uniform, is to execute each data chunk locally and only load balance between the heterogeneous devices.

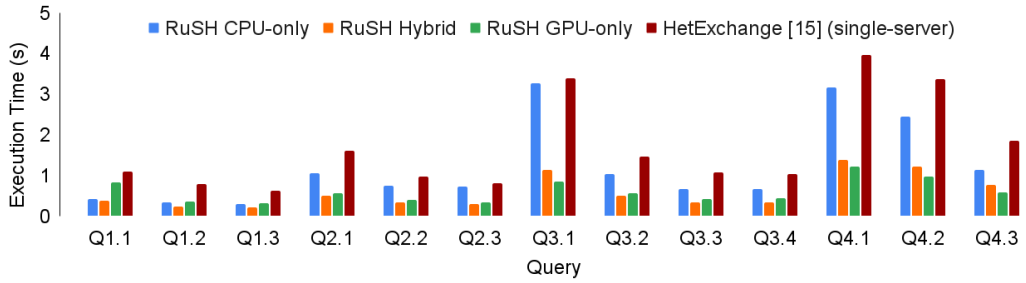


Figure 7.6: SSB (SF1000) with CPU-resident data.

CPU-only execution has a similar behavior as the previous experiments, with queries that were not saturating the bandwidth seeing a small speed-up compared to the data-shuffling case: ~25 GB/s per CPU socket that were previously used to send and receive data across the servers are now available to the CPU to speed up the joining operation for queries Q3.1, Q4.1, and Q4.2 that were starving for additional memory bandwidth. GPU-only execution sees a similar pattern: the queries that were previously bottlenecked by the incoming data fetches can now use fine granularity accesses to fetch their data over the PCIe interconnect, from CPU memory, overcoming the 12.5 GB/s throughput when selectiveness increases.

Hybrid execution achieves a similar execution time to the best of CPU-only and GPU-only execution and even better performance when CPU-only and GPU-only execution have similar throughput, e.g., Q1.2, Q1.3, Q3.3, Q3.4, as it allows combining their throughputs. Similar to the shuffling case, the high selectivity of the SSB queries combined with the high-throughput execution, however, leaves little room for load-balancing mistakes, penalizing Hybrid execution in queries like Q3.1 and Q4.* where sending a task to the slower execution unit can create tail in execution.

7.8.4 Impact of Different Features

Methodology. RuSH combines a variety of optimizations to achieve efficient execution on multi-server CPU-GPU platforms. To showcase the impact of each optimization, we incrementally enable the different performance-related features and plot in Figure 7.7 the speed-up over the single-server CPU-only execution for SSB Q3.1. We use two configurations: i) a single-server setup to showcase how RuSH's implementation combines multiple ideas from prior-art for single-server CPU-GPU query execution and set the baseline for multi-server execution, and ii) a multi-server setup where we artificially create skew across the machines by placing 90% of the data in one machine and equidistributing the rest of the data to the remaining three machines.

Inside a single-server, RuSH uses ideas from HetExchange [23] to parallelize execution across CPUs and GPUs, providing 2.3x speed-up by enabling the execution engine to exploit the available GPU processing capabilities. Combined with lazy transfers [83, 102], we observe a 3.8x speed-up as GPU-acceleration benefits from the selectivity of the first joins to reduce the

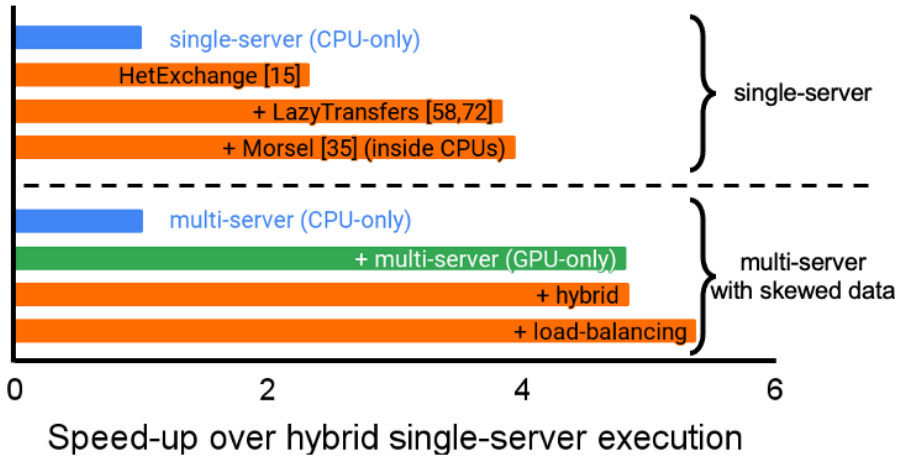


Figure 7.7: Breakdown of the impact of different features

data transferred to the GPU. HetExchange's execution model has each CPU worker be a single thread and parallelization across workers happens through the router. In contrast, RuSH follows a Morsel-like [51] approach for CPU workers: (CPU) join instances collaboratively build and share the hashtable to reduce the execution time for the build pipelines.

For the multi-server configuration, Figure 7.7 uses a configuration with skewed data (90% of the fact table resides on one machine). As a result, a "multi-server CPU-only" configuration with a policy that consumes all (fact-table) data on the server where they initially reside sees almost no speed-up compared to single-server execution. Specifically, for this query, any acceleration on query execution is offset by the overhead of running the network management, (de)initializing the communications paths, and gathering data/synchronizing at the end of the query. In contrast, replacing CPUs with GPUs provides significant speed-up, similarly to the single-server case – however, as one server has 10% less data than in the single-server case, multi-server GPU execution is faster than the single-server case. Furthermore, in contrast with the CPU-only multi-server configuration, the GPU-only configuration sees even less overhead from the network management, as network management runs on the CPU.

With the GPU having a high advantage over the CPU for Q3.1, the selected setup stresses hybrid execution: the diversity in processing resources across CPUs and GPUs makes routing tasks across the two, even inside each server of the multi-server setup, crucial in achieving efficient execution. RuSH's two-step offloading such task routing: instead of treating all devices equally, RuSH offloads first to a remote server and as a second step to a device. As a result, even when forcing server-local offloading, the '+hybrid' bar can achieve performance close to combining the throughput of the GPU and CPU probe pipelines – the overhead that hiders it from achieving a speed-up greater than the GPU-only configuration is time spent on the build pipelines. Specifically, while the GPU-only execution only builds the hash-table in

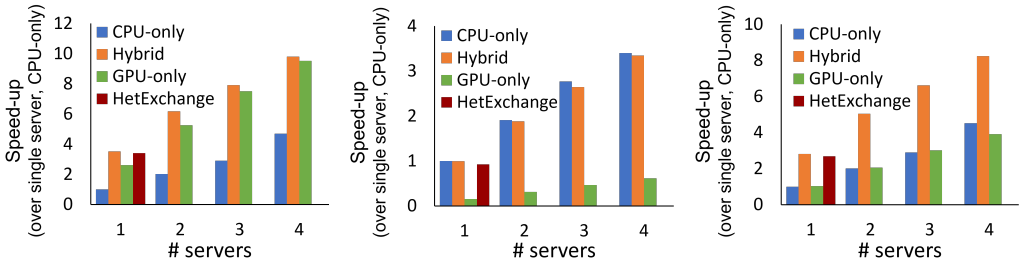
the GPUs, hybrid execution has to also construct the CPU-side hashtables, which becomes a (relatively to the speed-up) significant overhead as overall execution time decreases with more servers.

Lastly, the skewed data distribution across the servers creates a load imbalance. Combining RuSH's two-stage offloading with its delayed data transfers enables the overloaded server to push tasks to remote devices, resulting in the speedup shown by '+load-balancing'. The first offloading stage recognizes that the current server, while overloaded, it has an opportunity to push out data to another server. Specifically, the local GPUs pull data through the lazy accesses generating a specific throughput on the flows that pass through them. The local CPU also accesses the local data at a specific, albeit lower, throughput. However, the rackscale router above these two flows, sees that the flows towards the remote servers have available capacity that can be used and thus it starts assigning tasks to these flows whenever the local flows have more than a number of pending tasks. As a result, periodically, tasks are pushed out to remote servers. In terms of the hardware resources of the server with the 90% of the data, the PCIe bus to the local GPUs is used to selectively pull data, the bus to the local NICs is used to sequentially transfer data to remote servers, and the local CPUs use part of the remaining memory bus capacity for the local in-CPU processing. As there is only one NIC per CPU socket in our configuration and each NIC has the same PCIe bandwidth as a GPU, the best someone would expect for Q3.1 from load-balancing would be 2x – as much as using the NICs to reach out to an equal number of remote GPUs. However, this assumes eager CPU-to-GPU transfers while GPUs access CPU data in smaller granularity to benefit from the query selecivity [83, 102], through zero-copy lazy accesses. In contrast, network transfers require coarser accesses and thus benefit less from the query selectivity. As a result, the acceleration opportunity from carefully offloading to remote GPUs for the specific query is approximately equal to the selectivity of the first join (20%), assuming all time is spent on the probing pipeline. RuSH achieves 10% speedup, with the majority of the missing speed-up due to the non-scalability of the build-pipelines.

7.8.5 Adaptivity and Scalability

Methodology. To show the potential of RuSH, we evaluate the performance of Hybrid rack-scale execution. While SSB has multiple joins, combined with its high selectivity, it makes GPU-only execution near-optimal, misrepresenting CPU-only execution when GPU execution is performing lazy transfers [83] as the CPU execution is penalized for random access due to its low-memory bandwidth and GPU execution is accelerating PCIe transfers through fine-grained and lazy data fetching. Thus, we use a set of mixed SSB and scan-heavy queries.

To showcase the importance of CPU execution, we perform a scalability experiment and plot the speed-up over single-server CPU-only execution when varying the number of servers for three different cases: i) SSB queries, in Figure 7.8a, ii) scan-heavy query shown in Figure 7.8b, and iii) a mix of 50% SSB AND 50% scan-heavy queries, in Figure 7.8c. In the case of scan-



(a) SSB (SF1000) queries (b) Scan-heavy query (c) Mix of queries

Figure 7.8: Scalability on various workloads

heavy queries, CPU-only execution benefits from the local CPU-memory bandwidth while the GPU performance is limited as all data are remote to the GPUs, providing execution times significantly faster than GPU-only execution. In contrast, GPU-only execution benefits in the random-access heavy SSB queries due to the high internal memory bandwidth. In both cases, Hybrid achieves similar performance to the best of both, and when using a mixed workload, it overcomes the limitations of both approaches, providing consistently better execution times. Overall, all methods achieve near-linear scalability, showing that RuSH does not impose significant overheads and it efficiently parallelizes execution across servers and heterogeneous accelerators.

7.9 Related Work

Parallel query execution on CPUs. Traditionally there have been two common analytical query parallelization strategies: using and orchestrating operators, that are parallelism-agnostic, through the Volcano's *exchange* operator [30], or manually optimizing the parallel versions of the relational operators [4, 8, 13, 48, 81, 90]. Leis et al. [51] propose morsel-driven parallelism to consolidate the two approaches: parallel operator versions are used in conjunction with work-stealing and orchestration of operator pipelines.

RuSH is compatible with the above execution modes for intra-server CPU parallel execution. However, the aforementioned approaches are tailored for uniform CPU architectures and architectures that either observe little intra-server rebalancing penalty. In contrast, RuSH expands to GPU-accelerated architectures, uses a dataflow execution model to allow load-balancing across heterogeneous processors, and optimizes the resource management for the highly NUMA configurations of CPU-GPU servers by supporting late data binding and decentralized network buffer management.

Scale-up analytical query processing on GPUs. Similar to the CPU case, there has been a substantial effort in using GPUs to parallelize query execution [17, 87, 91], both in terms of offloading specific operators [43, 57–59, 61, 74, 86, 99], performing pre-query data placement based on query patterns [18], operator placement for column-at-a-time execution [44] and full query execution with Ocelot [37] providing a GPU backend for MonetDB, Voodoo [76] pro-

viding a vector algebra that allows running analytical queries to either CPUs or GPUs, through the same fronted, SiliconDB [25] aiming accelerators with limited functionality, HAPE [24] specializing operators to different devices and HetExchange [23] providing multi-CPU, multi-GPU hybrid query execution.

RuSH builds on top of the above GPU-accelerated engines, but 1) alleviates operators from the responsibility of managing multiple servers and CPUs, in addition to the GPUs, and 2) does not require globally accessible memory. Furthermore, while it builds on GPU-accelerated engines, it provides hybrid multi-CPU, multi-GPU execution, and load balances across the devices. RuSH uses a streaming model and thus it does not require full (intermediary) materializations. Lastly, RuSH builds on HetExchange and relies on variants of its operators for intra-server orchestration. Thought, RuSH also extends HetExchange to rack-scale, modifies its memory move operators and buffer management to support non-shared-memory architectures and limited inter-server observability.

Rack-scale query execution and RDMA. Volcano's exchange operator [30] allows parallel and distributed query execution across servers as its message-passing-friendly nature makes it appropriate for shared-nothing scale-out architectures. Binnig et al. [12] discuss how fast networks affect existing DBMS designs and opportunities for improvement. Liu et al. [55, 56] proposed using RDMA to alleviate the network stack overheads and propose a data shuffling operator. Furthermore, they proposed encapsulating the RDMA-awareness in a shuffling operator. However, Rödiger et al. [84] demonstrate that traditional exchange-based solutions do not scale in high-speed networks due to overheads caused by the network stack, poor intra-server load balancing, and high memory requirements. In addition, they use RDMA to reduce the number of intermediate copies and a hybrid parallelization technique that uses i) an exchange-based approach across servers, and ii) a Morsel-based approach inside each server. The inter-server exchange-based approach allows for shared-nothing architectures, while the Morsel-based approach, which allows data structure sharing across operator instances, requires cache-coherence and shared memory.

RuSH builds on Rödiger et al.'s two-stage parallelization strategy and on Liu et al.'s data shuffling operation by further decoupling the inter-server exchange operation to enable efficient and direct data transfers through late memory binding. Specifically, while Liu et al. [55] rely on multiple connections and Rödiger et al. [84] rely on round-robin assignment of the receiving buffers across NUMA nodes on the target servers, RuSH selects the target (CPU/GPU) NUMA nodes based on the current load. Thus, RuSH avoids rebalancing operations that would be frequent for a round-robin policy.

Distributed GPU execution. Gao and Sakharnykh [28] scale a GPU radix join to more than a thousand GPUs over InfiniBand. RuSH is complementary to the thousand-GPU-join as the thousand GPU joins shows the feasibility of large-scale GPU joins, while RuSH shows the

benefits of hybrid CPU-GPU execution in rack-scale analytics. RAPIDS [97] allows offloading Spark operations to GPUs; however, RuSH also allows dynamic load-balancing between CPUs and GPUs during query execution.

7.10 Conclusion

We show that analytical engines can efficiently scale to multiple CPU-GPU servers and utilize the available hardware accelerators to offer fast analytics despite heterogeneity-imposed scalability challenges. Using the combination of CPUs, GPUs, and high-bandwidth NICs, RuSH provides linear scalability over single-server hybrid CPU-GPU analytical engines. In addition, dataflow execution combined with delayed data transfers and bounded queues provide composable load-balancing despite the limited inter-server load visibility and that multiple accelerators share the same networking resources. Overall, we show that RuSH achieves up to 2.1x speedup over single-processing-unit rack-scale analytical engines and up to 8.2x over scale-up alternatives over a query sequence.

8 Conclusion and Outlook

This thesis investigates the impact of modern CPU-GPU server architectures on analytical query execution. Existing analytical engine designs target a specific set of processors, e.g., CPUs, despite the evolving hardware landscape introducing multiple accelerators into the server hardware. While such approaches can use hardware-conscious implementations to optimize execution for the target accelerator, they scale suboptimally with the hardware improvements: accelerator-specific optimizations cannot be ported to different hardware. Further, the engines that rely on hardware-oblivious approaches and generic implementations have an improved potential for utilizing hardware accelerators like GPUs through generic programming models. However, they generally waste hardware resources as advanced microarchitectural features are not used. In summary, existing monolithic analytical engine designs face a tradeoff between hardware efficiency and scalability to different accelerators. Thus, they are unable to utilize the full potential of the underlying hardware platform efficiently.

This thesis makes the case for heterogeneous-hardware engines: analytical engines that abolish static execution models tuned for single microarchitectures and instead encapsulate heterogeneity and decouple inter- from intra-device execution to achieve efficient execution on heterogeneous hardware. Through this decoupling, heterogeneous-hardware engines i) allow intra-device execution to be tuned independently for each target device, and ii) unify execution on heterogeneous devices to enable orchestration of the inter-device execution. During per-device tuning, heterogeneous-hardware engines use code generation to fuse generic operators with accelerator-specific primitives to enable operator specialization without the cost of modularity. In addition, heterogeneous-hardware engines enable composable execution across devices and servers by further decomposing inter-device execution to 1) model the devices based on their query-instance-specific throughput, 2) delay data transfers until the last moment to avoid unnecessary transfers, and 3) enable generic load-balancing and hierarchical orchestration. Lastly, to minimize the task offloading cost, heterogeneous-hardware engines use fine-granularity data accesses and coordinated prefiltering.

The ultimate goal of heterogeneous-hardware engines is to decouple the operators and orchestration from the underlying hardware and still provide the performance of an engine designed ground-up for the specific hardware. This chapter summarizes this thesis' contributions and discusses some ongoing efforts to expand the principles of heterogeneous-hardware engines beyond pure analytics, and into the full database management system for holistic accelerator use.

8.1 Heterogeneous Hardware: Efficient and Scalable Analytics

Our work optimizes the analytical engine for the modern CPU-GPU hardware through a three-step process: first, we decompose (Chapter 3) the analytical design space into three independent axes, inter-device, inter-operator, and intra-operators. Then, we achieve scalability by focusing on inter-device execution and virtualizing the hardware devices during orchestration. For efficiency, we use the decomposition to independently optimize for each device during inter- and intra-operator optimizations, while inter-device efficiency optimizations focus only on reducing data transfers.

Regarding scalability, we focus on the first axis, inter-device execution, and provide an encapsulation of the main heterogeneity-related hardware traits of a server (Chapter 4). This encapsulation enables heterogeneous-hardware engines to control the traits independently and optimize for each of them separately, through a meta-operator per trait. Then, we use a data-flow-like execution model to capture the performance of the different devices generically despite the asynchronous execution model. Further, the separation of the traits allows separate handling of the control and the data flow of our execution graph – a property that enables late and direct data transfers. We then extend the single-server concepts into rack-scale deployments (Chapter 7). We show how high-bandwidth RDMA-enabled networks impact GPU-accelerated analytics, including the amplification of the role of the CPU as the near-data processor in GPU-accelerated architectures, and extend our execution model to enable a hierarchical device orchestration capable of maintaining point-to-point data transfers without sacrificing orchestration scalability.

Regarding efficiency, we specialize execution in two steps: operator execution and data transfers. In the first step, we use our decomposition combined with the heterogeneity encapsulation to generate a different pipeline instance per target device (Chapter 5). Specifically, we enable operators to consider only one device at a time, and, during code generation, we fuse them and produce code specific to this device – optimizing over the inter-operator axis. Further, we i) identify key similarities of operators across CPUs and GPUs, and ii) provide device backends with simple primitives that are common across operators to increase the operator portability – optimizing over the intra-operator axis. Then, the generated code is instantiated across the devices and orchestrated through the operators described in the previous paragraph.

In the second step, we use the query selectivity, over-the-interconnect byte-addressability, and coprocessing capabilities to reduce the data transfer and overfetching across the interconnect – resulting in a reduced barrier to benefit from GPU acceleration (Chapter 6).

8.2 Looking Ahead: Heterogeneous Hardware beyond Analytics

Out-of-memory Analytics on High-bandwidth Storage. This thesis focuses on in-memory analytics, where input data are stored across the CPU and GPU memories. However, the available single-server in-memory space is currently limited to a few TBs of input data. After that point, to increase the input data capacity, the analytical engine either has to scale out, similarly to Chapter 7, or rely on out-of-memory data. Still, scaling out to increase the input capacity is wasteful in scenarios where the server's accelerators and interconnects are idle.

Directly attached NVMe arrays [33, 64] provide significant storage bandwidth when enough storage devices are used, reaching tens of GBps of input bandwidth for sequential data scans. Our proposed HetExchange framework (Chapter 4) is a natural fit for integrating GPU-accelerated analytics with PCIe storage devices, like NVMes. The mem-move operators can invoke direct NVMe-to-GPU or NVMe-to-CPU data transfers through GDS [52, 66], while the router operator would orchestrate execution. As a result, a HetExchange-augmented analytical engine could be coupled with NVMe storage arrays to provide CPU-GPU acceleration for out-of-memory workloads and instead use the CPU memory for latency-critical tasks, like transactions, or intermediary data structures, like hash tables.

Data Caching for GPU-accelerated Analytical Engines. NVMe arrays provide a straightforward way to satisfy GPU-accelerated analytics' input data bandwidth requirements for datasets that do not fit in memory. However, combining NVMe arrays and GPU acceleration also introduces significant caching challenges. First, NVMe arrays can provide similar bandwidth as inter-device communication and direct NVMe-to-GPU access – creating a new landscape where a storage device streams data to multiple, single-server but disaggregated processing units. Second, the page caching location provides significantly diverse cache hit gains: hits on GPU memory have an order of magnitude more data access bandwidth than a CPU memory hit. Still, there is considerably less available GPU memory capacity than CPU. Third, given the high storage bandwidth of NVMe arrays, some queries may be unable to sustain the available input bandwidth [64]; thus, caching the corresponding inputs into memory is wasteful.

The central concept underlying the orchestration methods proposed in Chapters 4 and 7 is the throughput-oriented view of the pipelines that allows modeling the relative device performance for load-balancing purposes. For analytics on disk-resident data, the same concept can be used to direct the data streams originating from the NVMe arrays to the CPU-GPU processing units. For data caching of disk-resident data, in HPCache [64], we extend the throughput-oriented system view to infer the benefit of different caching policies for CPU-only setup. Specifically, we cache column proportions by analyzing the in-memory pipeline throughput versus the available storage bandwidth. Then, we tune the cached

column proportions to minimize the overall execution time across a set of queries, relying on the pipeline throughputs. As each accelerator has a different performance profile, extending HPCache to CPU-GPU setups requires considering the alternative pipeline throughputs, device memory constraints, and the interconnect interference generated by shared GPUs and NVMes paths. With the limited GPU memory capacity, combining HPCache and GPU-accelerated analytics has the potential to increase the in-GPU caching benefits.

GPU-accelerated Analytics in DRAM-as-a-device Configurations. Over the past years, hardware vendors introduced multiple interconnects and refreshed past standards: PCIe reached version 6.0 in 5 years, after seven years in version 3; NVIDIA implemented multiple iterations of NVLink, and multiple vendors backed up CXL [1] as interconnect for "memory expansions and accelerators", that includes byte-addressable far-memory pools. Such far-memory setups and diverse interconnectivity standards will likely increase the memory capacity available on servers and push CPU architectures towards on-package High-Bandwidth Memory (HBM), similar to the upcoming NVIDIA Grace CPU [67].

While the discussion in this thesis revolves around the CPU memory being the high-capacity, byte-addressable memory, the proposed device-centric analytical engine designs make no actual distinction between CPU and GPU memory. In fact, the more the memories that support fine-granularity accesses, the more acute the impact of Lazy and SemiLazy accesses introduced in Chapter 6. Further, as the interconnect and memory bandwidths increase, for example, with the new NVLink and PCIe iterations, the higher the impact of direct point-to-point transfers and local routing discussed in Chapter 7, as unnecessary transfers and data staging cause higher interference.

Hybrid Transactional and Analytical Query Processing. This thesis focuses on analytical query processing and treats the input data as up-to-date snapshots. However, timely business intelligence requires incorporating fresh data into the analytical snapshots or accessing the transactional storage. Further, due to the task-parallel nature of transactional workloads, the transactional engine usually runs and generates the fresh data on the CPU side. Still, naively integrating both can result in significant resource interference [80].

For a symbiotic coexistence of the transactional engine that generated the data and the bandwidth-hungry analytical engine, Proteus currently relies on the Caldera [7] to drive the computational resources of the two engines, and a two-tiered multi-versioning storage [83] that maintains a two-versioned storage in the tier responsible for sharing snapshots across the transactional and analytical engine. This allows GPU-based analytical query processing to operate in a non-blocking way and with controlled interference between the two engines [83] by employing the pull-based data access methods of Chapter 6 for accessing the snapshot from the GPUs. However, as Laconic uses both CPU and GPU processing resources, it can increase the OLTP-OLAP interference. Extending the resource scheduling techniques of RDE [82] is a promising direction towards reducing interference by adapting the snapshot update mechanism based on the amount of fresh data and competing tasks.

Bibliography

[1] Compute Express Link. https://en.wikipedia.org/wiki/Compute_Express_Link.

[2] MapD. https://www.mapd.com/.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5 (10):1064–1075, 2012. doi: 10.14778/2336664.2336678. URL http://vldb.org/pvldb/vol5/p1064_martina-cezaraalbutiu_vldb2012.pdf.

[5] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. DPI: the data processing interface for modern networks. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL http://cidrdb.org/cidr2019/papers/p11-alonso-cidr19.pdf.

[6] Kamil Anikiej. Multi-core parallelization of vectorized query execution. Master's thesis, VU University, 2010.

[7] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. URL http://cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf.

[8] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, 2013. doi: 10.14778/2732219.2732227. URL http://www.vldb.org/pvldb/vol7/p85-balkesen.pdf.

[9] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 362–373. IEEE Computer Society, 2013. doi: 10.1109/ICDE.2013.6544839. URL https://doi.org/10.1109/ICDE.2013.6544839.

[10] Maximilian Bandle, Jana Giceva, and Thomas Neumann. To partition, or not to partition, that is the join question in a real system. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 168–180. ACM, 2021. doi: 10.1145/3448016.3452831. URL https://doi.org/10.1145/3448016.3452831.

[11] 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 Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 221–230. ACM, 2018. doi: 10.1145/3183713.3190662. URL https://doi.org/10.1145/3183713.3190662.

[12] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, 2016. doi: 10.14778/2904483.2904485. URL http://www.vldb.org/pvldb/vol9/p528-binnig.pdf.

[13] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48. ACM, 2011. doi: 10.1145/1989323.1989328. URL https://doi.org/10.1145/1989323.1989328.

[14] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65. Morgan Kaufmann, 1999. URL http://www.vldb.org/conf/1999/P5.pdf.

[15] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237. www.cidrdb.org, 2005. URL http://cidrdb.org/cidr2005/papers/P19.pdf.

[16] Sebastian Breß. The design and implementation of cogadb: A column-oriented gpu-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014. doi: 10.1007/s13222-014-0164-z. URL https://doi.org/10.1007/s13222-014-0164-z.

[17] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. Gpu-accelerated database systems: Survey and open challenges. *Trans. Large Scale Data Knowl. Centered Syst.*, 15:1–35, 2014. doi: 10.1007/978-3-662-45761-0_1. URL https://doi.org/10.1007/978-3-662-45761-0_1.

[18] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1891–1906. ACM, 2016. doi: 10.1145/2882903.2882936. URL https://doi.org/10.1145/2882903.2882936.

[19] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating custom code for efficient query execution on heterogeneous processors. *VLDB J.*, 27(6):797–822, 2018. doi: 10.1007/s00778-018-0512-y. URL https://doi.org/10.1007/s00778-018-0512-y.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015. URL http://sites.computer.org/debull/A15dec/p28.pdf.

[21] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 116–127. IEEE Computer Society, 2004. doi: 10.1109/ICDE.2004.1319989. URL https://doi.org/10.1109/ICDE.2004.1319989.

[22] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[23] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hetexchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.*, 12(5):544–556, 2019. doi: 10.14778/3303753.3303760. URL http://www.vldb.org/pvldb/vol12/p544-chrysogelos.pdf.

[24] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL http://cidrdb.org/cidr2019/papers/p127-chrysogelos-cidr19.pdf.

[25] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. A morsel-driven query execution engine for heterogeneous multi-cores. *Proc. VLDB Endow.*, 12(12):2218–2229, 2019. doi: 10.14778/3352063.3352137. URL http://www.vldb.org/pvldb/vol12/p2218-dursun.pdf.

[26] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012. doi: 10.1109/MM.2012.17. URL https://doi.org/10.1109/MM.2012.17.

[27] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1603–1618. ACM, 2018. doi: 10.1145/3183713.3183734. URL https://doi.org/10.1145/3183713.3183734.

[28] Hao Gao and Nikolai Sakharnykh. Scaling joins to a thousand gpus. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*, pages 55–64, 2021. URL http://www.adms-conf.org/2021-camera-ready/gao_adms21.pdf.

[29] Gábor E. Gévay, Tilmann Rabl, Sebastian Breß, Lorand Madai-Tahy, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. Efficient control flow in dataflow systems: When ease-of-use meets high performance. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 1428–1439. IEEE, 2021. doi: 10.1109/ICDE51399.2021.00127. URL https://doi.org/10.1109/ICDE51399.2021.00127.

[30] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, pages 102–111. ACM Press, 1990. doi: 10.1145/93597.98720. URL https://doi.org/10.1145/93597.98720.

[31] Khronos OpenCL Working Group. The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf.

[32] Tim Gubner and Peter A. Boncz. Charting the design space of query execution using VOILA. *Proc. VLDB Endow.*, 14(6):1067–1079, 2021. doi: 10.14778/3447689.3447709. URL http://www.vldb.org/pvldb/vol14/p1067-gubner.pdf.

[33] Gabriel Haas, Michael Haubenschild, and Viktor Leis. Exploiting directly-attached nvme arrays in DBMS. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf.

[34] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011. doi: 10.1109/MM.2011.77. URL https://doi.org/10.1109/MM.2011.77.

[35] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, 2009. doi: 10.1145/1620585.1620588. URL https://doi.org/10.1145/1620585.1620588.

[36] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache query co-processing on coupled CPU-GPU architectures. *Proc. VLDB Endow.*, 8(4):329–340, 2014. doi: 10.14778/2735496.2735497. URL http://www.vldb.org/pvldb/vol8/p329-he.pdf.

[37] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, 2013. doi: 10.14778/2536360.2536370. URL http://www.vldb.org/pvldb/vol6/p709-heimel.pdf.

[38] Mark D. Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *Commun. ACM*, 64 (12):36–38, 2021. doi: 10.1145/3460970. URL https://doi.org/10.1145/3460970.

[39] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *16th International Conference on Parallel Architectures and Compilation Techniques (PACT 2007), Brasov, Romania, September 15-19, 2007*, pages 189–198. IEEE Computer Society, 2007. doi: 10.1109/PACT.2007.12. URL http://doi.ieeecomputersociety.org/10.1109/PACT.2007.12.

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 59–72. ACM, 2007. doi: 10.1145/1272996.1273005. URL https://doi.org/10.1145/1272996.1273005.

[41] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826, 2018. URL http://arxiv.org/abs/1804.06826.

[42] Tim Kaldewey, Guy M. Lohman, René Müller, and Peter Benjamin Volk. GPU join processing revisited. In Shimin Chen and Stavros Harizopoulos, editors, *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, pages 55–62. ACM, 2012. doi: 10.1145/2236584.2236592. URL https://doi.org/10.1145/2236584.2236592.

[43] 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. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*, pages 6:1–6:10. ACM, 2017. doi: 10.1145/3076113.3076115. URL https://doi.org/10.1145/3076113.3076115.

[44] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *Proc. VLDB Endow.*, 10(7): 733–744, 2017. doi: 10.14778/3067421.3067423. URL http://www.vldb.org/pvldb/vol10/p733-karnagel.pdf.

[45] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.*, 9(12):972–983, 2016. doi: 10.14778/2994509.2994516. URL http://www.vldb.org/pvldb/vol9/p972-karpathiotakis.pdf.

[46] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011. doi: 10.1109/ICDE.2011.5767867. URL https://doi.org/10.1109/ICDE.2011.5767867.

[47] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, 2018. doi: 10.14778/3275366.3275370. URL http://www.vldb.org/pvldb/vol11/p2209-kersten.pdf.

[48] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, 2009. doi: 10.14778/1687553.1687564. URL http://www.vldb.org/pvldb/vol2/vldb09-257.pdf.

[49] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. SABER: window-based hybrid stream processing for heterogeneous architectures. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 555–569. ACM, 2016. doi: 10.1145/2882903.2882906. URL https://doi.org/10.1145/2882903.2882906.

[50] Christoph Lameter. An overview of non-uniform memory access. *Commun. ACM*, 56 (9):59–54, 2013. doi: 10.1145/2500468.2500477. URL https://doi.org/10.1145/2500468.2500477.

[51] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM, 2014. doi: 10.1145/2588555.2610507. URL https://doi.org/10.1145/2588555.2610507.

[52] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. *Proc. VLDB Endow.*, 9(14):1647–1658, 2016. doi: 10.14778/3007328.3007331. URL http://www.vldb.org/pvldb/vol9/p1647-li.pdf.

[53] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous CPU/GPU spark platform for machine learning algorithms. In *10th IEEE International Conference on Networking, Architecture and Storage, NAS 2015, Boston, MA, USA, August 6-7, 2015*, pages 347–348. IEEE Computer Society, 2015. doi: 10.1109/NAS.2015.7255222. URL https://doi.org/10.1109/NAS.2015.7255222.

[54] Zhifang Li, Mingcong Han, Shangwei Wu, and Chuliang Weng. Shadowvm: accelerating data plane for data analytics with bare metal cpus and gpus. In Jaejin Lee and Erez Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 147–160. ACM, 2021. doi: 10.1145/3437801.3441595. URL https://doi.org/10.1145/3437801.3441595.

[55] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 48–63. ACM, 2017. doi: 10.1145/3064176.3064202. URL https://doi.org/10.1145/3064176.3064202.

[56] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. *ACM Trans. Database Syst.*, 44(4):17:1–17:45, 2019. doi: 10.1145/3360900. URL https://doi.org/10.1145/3360900.

[57] Clemens Lutz, Sebastian Breß, Tilmann Rabl, Steffen Zeuch, and Volker Markl. Efficient k-means on gpus. In Wolfgang Lehner and Kenneth Salem, editors, *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*, pages 3:1–3:3. ACM, 2018. doi: 10.1145/3211922.3211925. URL https://doi.org/10.1145/3211922.3211925.

[58] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Pump up the volume: Processing large data on gpus with fast interconnects. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q.

Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1633–1649. ACM, 2020. doi: 10.1145/3318464.3389705. URL https://doi.org/10.1145/3318464.3389705.

[59] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Triton join: Efficiently scaling to a large join state on gpus with fast interconnects. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1017–1032. ACM, 2022. doi: 10.1145/3514221.3517911. URL https://doi.org/10.1145/3514221.3517911.

[60] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, 2017. doi: 10.14778/3151113.3151114. URL http://www.vldb.org/pvldb/vol11/p1-menon.pdf.

[61] Georgios Michas, Periklis Chrysogelos, Ioannis Mytilinis, and Anastasia Ailamaki. Hardware-conscious sliding window aggregation on gpus. In Danica Porobic and Spyros Blanas, editors, *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, pages 13:1–13:5. ACM, 2021. doi: 10.1145/3465998.3466014. URL https://doi.org/10.1145/3465998.3466014.

[62] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering chiplet technology and design for the AMD epyc™ and ryzen™ processor families : Industrial product. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*, pages 57–70. IEEE, 2021. doi: 10.1109/ISCA52012.2021.00014. URL https://doi.org/10.1109/ISCA52012.2021.00014.

[63] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011. doi: 10.14778/2002938.2002940. URL http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[64] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. Hpcache: Memory-efficient OLAP through proportional caching. In Spyros Blanas and Norman May, editors, *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*, pages 7:1–7:9. ACM, 2022. doi: 10.1145/3533737.3535100. URL https://doi.org/10.1145/3533737.3535100.

[65] Joel Nider and Alexandra (Sasha) Fedorova. The last CPU. In Sebastian Angel, Baris Kasikci, and Eddie Kohler, editors, *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, pages 1–8. ACM, 2021. doi: 10.1145/3458336.3465291. URL https://doi.org/10.1145/3458336.3465291.

[66] NVIDIA. GPUDirect Storage Documentation, . https://docs.nvidia.com/gpudirect-storage/index.html.

[67] NVIDIA. NVIDIA Grace CPU, . https://www.nvidia.com/en-us/data-center/grace-cpu/.

[68] NVIDIA. Benchmarking GPUDirect RDMA on Modern Server Platforms, . https://developer.nvidia.com/blog/benchmarking-gpudirect-rdma-on-modern-server-platforms/.

[69] NVIDIA. VOLTA Architecture and performance optimization, . https://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf.

[70] NVIDIA. Parallel Thread Execution ISA, . https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[71] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2009. doi: 10.1007/978-3-642-10424-4_17. URL https://doi.org/10.1007/978-3-642-10424-4_17.

[72] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block oriented processing of relational database operations in modern computer architectures. In Dimitrios Georgakopoulos and Alexander Buchmann, editors, *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 567–574. IEEE Computer Society, 2001. doi: 10.1109/ICDE.2001.914871. URL https://doi.org/10.1109/ICDE.2001.914871.

[73] Johns Paul, Jiong He, and Bingsheng He. GPL: A gpu-based pipelined query processing engine. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1935–1950. ACM, 2016. doi: 10.1145/2882903.2915224. URL https://doi.org/10.1145/2882903.2915224.

[74] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. Mg-join: A scalable join for massively parallel multi-gpu architectures. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1413–1425. ACM, 2021. doi: 10.1145/3448016.3457254. URL https://doi.org/10.1145/3448016.3457254.

[75] Holger Pirk, Stefan Manegold, and Martin L. Kersten. Waste not... efficient co-processing of relational data. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 508–519. IEEE Computer Society, 2014. doi: 10.1109/ICDE.2014.6816677. URL https://doi.org/10.1109/ICDE.2014.6816677.

[76] Holger Pirk, Oscar R. Moll, Matei Zaharia, and Sam Madden. Voodoo - A vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14): 1707–1718, 2016. doi: 10.14778/3007328.3007336. URL http://www.vldb.org/pvldb/vol9/p1707-pirk.pdf.

[77] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD vectorization for in-memory databases. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1493–1508. ACM, 2015. doi: 10.1145/2723372.2747645. URL https://doi.org/10.1145/2723372.2747645.

[78] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: A practical approach for robust index joins. *Proc. VLDB Endow.*, 11 (2):230–242, 2017. doi: 10.14778/3149193.3149202. URL http://www.vldb.org/pvldb/vol11/p230-psaropoulos.pdf.

[79] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *VLDB J.*, 28(4):451–471, 2019. doi: 10.1007/s00778-018-0533-6. URL https://doi.org/10.1007/s00778-018-0533-6.

[80] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. Scaling up mixed workloads: A battle of data freshness, flexibility, and scheduling. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking. Traditional to Big Data - 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1-5, 2014. Revised Selected Papers*, volume 8904 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2014. doi: 10.1007/978-3-319-15350-6_7. URL https://doi.org/10.1007/978-3-319-15350-6_7.

[81] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, 2013. doi: 10.14778/2536222.2536233. URL http://www.vldb.org/pvldb/vol6/p1080-barber.pdf.

[82] Aunn Raza, Periklis Chrysogelos, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. Adaptive HTAP through elastic resource scheduling. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceed-*

ings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, pages 2043–2054. ACM, 2020. doi: 10.1145/3318464.3389783. URL https://doi.org/10.1145/3318464.3389783.

[83] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. Gpu-accelerated data management under the test of time. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf.

[84] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4):228–239, 2015. doi: 10.14778/2856318.2856319. URL http://www.vldb.org/pvldb/vol9/p228-roediger.pdf.

[85] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1194–1205. IEEE Computer Society, 2016. doi: 10.1109/ICDE.2016.7498324. URL https://doi.org/10.1109/ICDE.2016.7498324.

[86] Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Performance analysis and automatic tuning of hash aggregation on gpus. In Thomas Neumann and Ken Salem, editors, *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, pages 8:1–8:11. ACM, 2019. doi: 10.1145/3329785.3329922. URL https://doi.org/10.1145/3329785.3329922.

[87] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous cpu/gpu systems. *ACM Comput. Surv.*, 55(1), jan 2022. ISSN 0360-0300. doi: 10.1145/3485126. URL https://doi.org/10.1145/3485126.

[88] Eyal Rozenberg and Peter A. Boncz. Faster across the pcie bus: a GPU library for lightweight decompression: including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*, pages 8:1–8:5. ACM, 2017. doi: 10.1145/3076113.3076122. URL https://doi.org/10.1145/3076113.3076122.

[89] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*, pages 17:1–17:12. ACM, 2017. doi: 10.1145/3085504.3085521. URL https://doi.org/10.1145/3085504.3085521.

[90] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of*

Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 1961–1976. ACM, 2016. doi: 10.1145/2882903.2882917. URL https://doi.org/10.1145/2882903.2882917.

[91] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1617–1632. ACM, 2020. doi: 10.1145/3318464.3380595. URL https://doi.org/10.1145/3318464.3380595.

[92] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 510–521. Morgan Kaufmann, 1994. URL http://www.vldb.org/conf/1994/P510.PDF.

[93] Panagiotis Sioulas and Anastasia Ailamaki. Vectorizing an in situ query engine. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2261–2262. ACM, 2016. doi: 10.1145/2882903.2914829. URL https://doi.org/10.1145/2882903.2914829.

[94] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 698–709. IEEE, 2019. doi: 10.1109/ICDE.2019.00068. URL https://doi.org/10.1109/ICDE.2019.00068.

[95] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. Vectorization vs. compilation in query execution. In Stavros Harizopoulos and Qiong Luo, editors, *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*, pages 33–40. ACM, 2011. doi: 10.1145/1995441.1995446. URL https://doi.org/10.1145/1995441.1995446.

[96] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 417–432. ACM, 2017. doi: 10.1145/3035918.3064043. URL https://doi.org/10.1145/3035918.3064043.

[97] RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018. URL https://rapids.ai.

[98] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. DFI: the data flow interface for high-speed networks. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1825–1837. ACM, 2021. doi: 10.1145/3448016.3452816. URL https://doi.org/10.1145/3448016.3452816.

[99] Diego G. Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A. Boncz. Optimizing group-by and aggregation using GPU-CPU co-processing. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*, pages 1–10, 2018. URL http://www.adms-conf.org/2018-camera-ready/tome_groupby.pdf.

[100] Li Wang, Minqi Zhou, Zhenjie Zhang, Yin Yang, Aoying Zhou, and Dina Bitton. Elastic pipelining in an in-memory database cluster. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1279–1294. ACM, 2016. doi: 10.1145/2882903.2882904. URL https://doi.org/10.1145/2882903.2882904.

[101] Haicheng Wu, Gregory Frederick Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pages 107–118. IEEE Computer Society, 2012. doi: 10.1109/MICRO.2012.19. URL https://doi.org/10.1109/MICRO.2012.19.

[102] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.*, 6(10):817–828, 2013. doi: 10.14778/2536206.2536210. URL http://www.vldb.org/pvldb/vol6/p817-yuan.pdf.

[103] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010. URL https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

[104] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[105] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 145–156. ACM, 2002. doi: 10.1145/564691.564709. URL https://doi.org/10.1145/564691.564709.

[106] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 741–758. ACM, 2019. doi: 10.1145/3299869.3300081. URL https://doi.org/10.1145/3299869.3300081.

Periklis Chrysogelos

Curriculum Vitae

Chemin des Triaudes 4A
1024 Ecublens VD
Switzerland
+41 78 681 48 12
periklis.chrysogelos@epfl.ch

Education

2016–present **Ph.D. student**, *School of Computer and Communication Sciences*, École Polytechnique Fédérale de Lausanne, Switzerland.
Tentative Thesis: **Efficient Analytical Query Processing on CPU-GPU Hardware Platforms**,
Advisor: Professor Anastasia Ailamaki

2011–2016 **Diploma in Electrical and Computer Engineering**, *School of Electrical and Computer Engineering*, Technical University of Crete, Greece.
Diploma Thesis: **Streaming High Performance Support Vector Methods**,
Advisor: Professor Minos Garofalakis,
Grade: 9.41/10.0, **Second** in class rank

Professional Experience

2016–present **Research Assistant**, DIAS, *École Polytechnique Fédérale de Lausanne*.

Jun-Sep 2020 **Research Intern**, Data and In-Memory Technologies, **Oracle**.

Honors and Awards

2019 **Appreciation for Exceptional Performance**, Award by the School of Computer and Communication Sciences of École Polytechnique Fédérale de Lausanne.

2019 **HDMS student award**.

2016 **EPFL EDIC Ph.D. Fellowship**, *for outstanding academic achievements*, Award by École Polytechnique Fédérale de Lausanne.

2016 **Award of Academic Excellence**, *for graduating in 2016 with the 2nd highest GPA from the School of Electrical and Computer Engineering of Technical University of Crete*, Award by Limmat Stiftung.

2011 **Finalist**, *23rd Hellenic Computing Olympiad*.

Publications

[1] H. Nicholson, P. Chrysogelos, and A. Ailamaki. HPCache: Memory-Efficient OLAP Through Proportional Caching. In *DaMoN 2022*.
`https://dl.acm.org/doi/10.1145/3533737.3535100`.

[2] A. Herlihy, P. Chrysogelos, and A. Ailamaki. Boosting Efficiency of External Pipelines by Blurring Application Boundaries. In *CIDR 2022*.
`http://cidrdb.org/cidr2022/papers/p81-herlihy.pdf`.

[3] G. Michas, P. Chrysogelos, I. Mytilinis, and A. Ailamaki. Hardware-Conscious Sliding Window Aggregation on GPUs. In *DaMoN 2021*.
`https://dl.acm.org/doi/10.1145/3465998.3466014`.

[4] A. Raza, P. Chrysogelos, A. C. Anadiotis, and A. Ailamaki. Adaptive HTAP through Elastic Resource Scheduling. In *SIGMOD 2020*.
`https://dl.acm.org/doi/10.1145/3318464.3389783`.

[5] A. Ailamaki, P. Chrysogelos, A. Deshpande, and T. Kraska. The SIGMOD 2019 Research Track Reviewing System. *SIGMOD Record*, 48(2):47–54, 2019. `https://sigmodrecord.org/2019/10/19/the-sigmod-2019-research-track-reviewing-system/`.

[6] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. In *PVLDB 2019*. https://dl.acm.org/doi/10.14778/3303753.3303760.

[7] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious Joins on GPUs. In *ICDE 2019*. https://ieeexplore.ieee.org/document/8731612.

[8] P. Chrysogelos, P. Sioulas, and A. Ailamaki. Hardware-conscious Query Processing in GPU-accelerated Analytical Engines. In *CIDR 2019*. http://cidrdb.org/cidr2019/papers/p127-chrysogelos-cidr19.pdf.

Community Service

2020, 2021 ADMS, *Program Committee*.

2021 ACM Computing Surveys, *Reviewer*.

Research Interests

Databases, Analytical Query Processing, Multi-CPU Multi-GPU systems.

Large Scale and High-Performance Computing, Distributed and Parallel Systems.

Programming Languages and Frameworks

Fluent **C++, CUDA**, *Java, C, Python*.

Familiar Assembly, LLVM IR, PTX, SQL, VHDL, Matlab/GNU Octave.

Frameworks **LLVM**, MLIR, Apache Calcite, MapReduce, Theano.

Others

2015 Quantitative score of 168, *GRE*.

2014 **Parallel engine for 2048 Game**, publicly available coursework (team of two).

2014 **Parallel** implementation of a **chess engine**.

Languages

2015 Score of 104, ***TOEFL*** *iBT*.

2009 Certificate of **Proficiency in English**, *University of Michigan*.

Native **Greek**.

Profiles on Coding Sites

CodeChef, *https://www.codechef.com/users/chapeiro*.

Github, *https://github.com/Chapeiro*.