Towards Large-Scale and Practical Data Analytics on GPUs

by

Bobbi Winema Yogatama

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: 08/13/2025

The dissertation is approved by the following members of the Final Oral Committee:
Xiangyao Yu, Assistant Professor, Computer Sciences
AnHai Doan, Professor, Computer Sciences
Ming Liu, Assistant Professor, Computer Sciences
Tsung-Wei Huang, Associate Professor, Electrical Engineering

For my family

Acknowledgments

I have been fortunate to receive the support of many people throughout my PhD.

First and foremost, I would like to thank my advisor, Xiangyao Yu. Xiangyao taught me how to be a better researcher and a better person in life. His guidance has been invaluable, and he has always offered thoughtful advice in every situation. I could not have asked for a better advisor than him.

I am also grateful to the other thesis committee members—AnHai Doan, Ming Liu, and Tsung-Wei Huang—for their invaluable feedback and support throughout the dissertation process.

Aside from my thesis commitees, I would like to thank my research collaborators: Weiwei Gong, Anil Shanbhag, Joshua Patterson, Gregory Kimball, Adrian Cockcroft, Yinan Li, Bailu Ding, Brandon Miller, Yunsong Wang, Graham Markall, Jacob Hemstad, Spyros Blanas, Kaushik Rajan, Ravi Ramamurthy, Wes McKinney, Yu Teng, and many more. Their insights and perspectives have enriched my research in ways I could not have achieved alone.

I am especially grateful to the Sirius project team– Yifei Yang, Kevin Kristensen, Devesh Sarda, and Abigale Kim. They are the most talented group of people I have ever worked with. Sirius would not have been possible without their collaboration.

I've also been lucky to have incredible friends who shaped my PhD experience—-Sujay Yadalam, Konstantinos Kanellis, Jihye Choi, Vinay Banakar, Dyah Adila, and Nicholas Robert—who made my PhD journey ten times more enjoyable.

A special thank-you goes to my girlfriend, Heather Hellenga, who has been the greatest source of joy in my life.

Finally, and most importantly, I thank my parents, Melisa Chrisyanti and Teddy Yogatama, and my brothers, Willie Yogatama and Dani Yogatama. They have supported me unconditionally and deserve all the credit for any success I have achieved.

Contents

List of Tables

List of Figures

Abstract

The era of GPU-powered data analytics has arrived!

This dissertation argues that recent hardware advancements—such as larger GPU memory, faster interconnects and I/O, and decreasing costs—have laid a strong foundation for scalable and practical GPU-accelerated data analytics. Despite these advancements, however, GPU databases have yet to see widespread adoption in both academia and industry, primarily due to three key challenges: (1) supporting larger-than-GPU memory workload, (2) difficulty supporting complex operators on GPUs, and (3) high switching costs from CPU-based to GPU-based databases.

This dissertation addresses these challenges with the goal of driving mainstream adoption of GPU databases. First, we improve performance for workloads that exceed GPU memory through three key system-level optimizations: data compression, heterogeneous CPU–GPU execution, and multi-GPU databases. We introduce a GPU-optimized framework for lightweight data decompression that achieves up to 2.2× speedup over state-of-the-art methods. We then present *Mordred*, a heterogeneous CPU–GPU database system with optimized data placement and query execution strategies, outperforming existing systems by up to 11×. Finally, we extend *Mordred* to a multi-GPU setting with *Lancelot*, demonstrating up to 12× performance gains over other multi-GPU DBMSes.

Next, we address the challenge of supporting complex operators on GPUs, specifically focusing on user-defined aggregate functions (UDAFs). Our framework accelerates UDAF execution by up to 8000× compared to existing approaches and as a result, has been deployed into production as part of the NVIDIA cuDF v23.02.

Lastly, to reduce the engineering and migration costs of switching from CPU to GPU databases, we introduce *Sirius*, an open-source GPU-native SQL engine that can serve as a drop-in accelerator for a variety of existing data systems. By leveraging composable data systems, *Sirius* can replace other CPU engines without migrating the

data or changing the user-facing interface. Sirius achieves 8.2× speedup as a drop-in accelerator for DuckDB (single-node) and up to 12.5× as a drop-in accelerator for Apache Doris (distributed).

Chapter 1

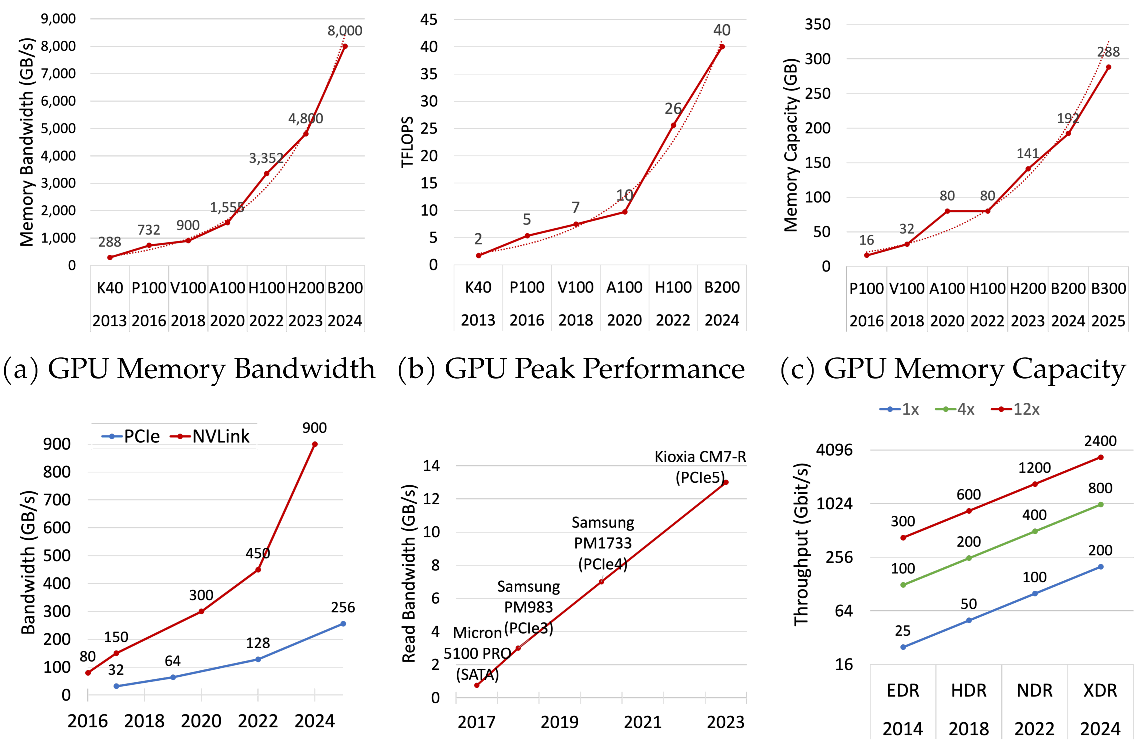
Introduction

The performance of a SQL engine is ultimately driven by the capabilities of the underlying hardware—especially compute throughput and memory bandwidth. Modern GPUs are rapidly outpacing CPUs on both fronts and have become the default hardware choice for data- and compute-intensive applications such as machine learning. Table 1 compares recent CPU and GPU architectures, highlighting the contrast in core count and memory bandwidth. Given the inherently parallel nature of relational data analytics, GPUs are a perfect fit for future analytical databases.

Table 1.1: Comparison of CPU and GPU Instances

	Amazon c6a.metal (AMD EPYC CPU)	GH200 (NVIDIA GPU)
Core Count	192 (vCPUs)	14,000+ (CUDA cores)
Memory BW	~400 GB/s	3,000 GB/s (HBM)
Memory Size	384 GB	96 GB (HBM)
Rental Cost	$7.344/h (AWS)	$3.2/h (Lambda Labs)

In this thesis, we argue that the hardware foundations are finally in place today to enable scalable GPU data analytics. For example, GPU memory capacity doubles almost every generation, starting from Volta (32 GB), to Ampere (80 GB), Hopper (192 GB), and most recently Blackwell (288 GB). Better interconnects such as `PCIe Gen6`, `NVLink-C2C` [22], and `GPUDirect` [14] significantly reduce the data movement overhead between GPUs and other system components. This allows a GPU to process data beyond on-device memory, enabling TBs of analytics even on a single GPU and more with distribution. Meanwhile, GPUs are increasingly affordable and accessible,



(a) GPU Memory Bandwidth (b) GPU Peak Performance (c) GPU Memory Capacity

(d) Interconnect Bandwidth (e) Storage Bandwidth (f) Network Bandwidth

Figure 1.1: Recent hardware trends

especially for older generations which are already sufficient for data analytics workloads. In the next section, we will explore in greater detail the catalysts behind the GPU era in data analytics.

1.1 GPU for Data Analytics: Why Now?

We argue that the GPU era for data analytics is finally here. In this section, we discuss the key drivers behind the rise of GPU-accelerated data analytics by examining the recent GPU hardware trends.

Higher Throughput and Computational Power. GPU memory bandwidth is an order of magnitude higher than that of CPUs. Modern GPUs can reach up to 8 TB/s and is doubling every two years, as shown in Figure 1.1a. This is especially important for data analytics, where workloads are throughput-heavy and often memory-bound. In addition, the highly parallel nature of data analytics makes it well-suited to fully leverage GPU's massive computational power. While CPU performance has stagnated in recent years due to the slowing of Moore's Law, GPU performance continues to scale, as shown in Figure 1.1b.

Bigger Memory at Faster Speed. The capacity of GPU device memory has tripled in the past 3 years, as shown in Figure 1.1c. While the largest GPU memory was merely 16 GB ten years ago, a modern NVIDIA B300 Ultra or AMD MI350X has 288 GB device memory, which is sufficient for small to medium-sized analytics workloads.

PCIe, the interconnect between GPU and CPU, has traditionally been a severe performance bottleneck due to its relatively low bandwidth. However, PCIe bandwidth has been doubling every two years with PCIe Gen6 offering 128 GB/s, which is comparable to CPU memory bandwidth. The `NVLink-C2C` [22] technology takes this even further, enabling 900 GB/s unidirectional bandwidth between CPU and GPU – see Figure 1.1d for more details. In a GH200 superchip, the Hopper GPU can access main memory at more than 400 GB/s, which is more than the maximum bandwidth between the CPU and main memory. These faster interconnects allow a GPU analytics engine to support data larger than the GPU device memory, enabling high-speed access to terabytes of main memory and beyond.

Faster Network and Storage. Besides faster memory speed, modern GPUs also have increasingly better support for network and storage, allowing a GPU engine to go beyond in-memory data processing. Technologies like `GPUDirectStorage` and `GPUDirectRDMA` allow GPUs to access data directly from storage or network with minimal CPU involvement at very high speed. Similar to interconnect technology, storage and network bandwidth have also grown rapidly in recent years and are expected to continue improving – see Figure 1.1e and Figure 1.1f. Moreover, the recent S3 over RDMA technique further enables high-speed object store for GPUs, which can achieve storage access speed at 200 GB/s [7]. With significantly higher computational power, GPUs are better positioned than CPUs to fully utilize these high-bandwidth resources.

Declining GPU Cost. While GPUs of the latest generation remain in high demand with high cost, older generations are becoming increasingly accessible in the cloud at substantially lower cost. For instance, the GH200 instance shown in Table 1.1 has an on-demand price of only $3.2/hour in Lambda Labs, which is significantly cheaper than many high-end CPU instances. The on-demand H100 prices in many GPU cloud providers has seen significant price drop from $8/hour in March 2023 to around $3/hour in 2025 [20, 34]. Even in major cloud provider such as AWS, in 2025 alone, the A100, H100, and H200 on-demand price have dropped by 33%, 44%, and 25% respectively [5].

1.2 Barrier in Adopting GPUs for Data Analytics

Despite their great potential, however, GPU-based SQL engines have not yet seen mainstream adoption. Prior efforts, both academic and commercial, have been constrained by the following three key challenges and thereby remain niche solutions.

1.2.1 Challenge I – Supporting Data Larger-than GPU Memory

Modern GPUs offer impressive computational throughput, but their memory capacity remains limited compared to CPU-based systems. Even the most advanced GPU, such as NVIDIA's Blackwell, supports up to 288 GB of device memory—far less than the multiple terabytes available in high-end CPU servers.

To address this limitation, existing GPU databases[15, 16, 9, 11, 39] stream data to the GPU on demand via the CPU-GPU interconnect when input data does not fit in the GPU memory. The bandwidth of these interconnects, however, is significantly lower than that of on-device memory. For example, PCIe Gen6 offers a peak bidirectional bandwidth of 128GB/s, whereas the NVIDIA B200 GPU provides up to 8TB/s of high memory bandwidth—a gap of nearly two orders of magnitude. Although newer interconnects such as NVLink C2C (900 GB/s) mitigate some of these limitations, the cost of offloading data between host and device memory remains substantially higher than accessing data from the on-device memory. As a result, GPU database performance degrades significantly when the working set exceeds available device memory [133, 90, 15, 11], making it difficult to scale analytics workloads beyond a certain point.

1.2.2 Challenge II – Supporting Complex Operators on GPUs

While GPUs excel at executing simple, data-parallel operations at massive scale, they often struggle with more complex or irregular computations. Some key examples are user-defined functions (UDFs) and user-defined aggregate functions (UDAFs), which pose significant challenges for GPU acceleration. First, their logic is typically defined at runtime, making it difficult to implement efficient GPU kernels ahead of time. Second, UDFs and UDAFs often introduce control-flow divergence and irregular memory access patterns, which are fundamentally at odds with the SIMT-style execution model of GPUs. As a result, existing GPU SQL engines often either (1) fallback to CPU execution, which results in no performance benefit, or (2) attempt to run these operators on GPUs inefficiently. For instance, executing a UDAF in cuDF—the NVIDIA GPU-accelerated data processing library—can be up to 2.5× slower than executing the same UDAF using single-threaded CPU execution in Pandas [129].

1.2.3 Challenge III – High Switching Cost from CPU to GPU DBMS

Even when hardware limitations and performance concerns are addressed, the practical adoption of GPU DBMSes remains hindered by high switching costs. First, there is substantial engineering cost required to build a SQL engine optimized for GPU architectures. This includes implementing core relational operators, managing memory hierarchies, and designing query execution engine that fully exploit the GPU's massive parallelism. These challenges are compounded by the fundamental architectural differences between CPUs and GPUs, and the limited availability of engineers with deep GPU programming expertise.

Second, transitioning from mature CPU-based systems to GPU-native engines introduces significant migration and compatibility hurdles. Many organizations rely on complex infrastructure, interface, tooling, and workflows tightly coupled with their existing DBMS. Differences in user interfaces or query APIs often require rewriting client applications, while data migration between CPU and GPU DBMSes may involve converting storage formats or moving large datasets. These friction points create resistance to adoption and have slowed the broader shift toward GPU-accelerated analytics.

1.3 Large-Scale and Practical Data Analytics on GPUs

This dissertation aims to address the three key challenges outlined in Section 1.2 to drive the adoption of GPU-accelerated data analytics. The first part of this thesis (**Part I**) addresses the GPU memory capacity limitation (**Challenge I**) by introducing key system-level optimizations that significantly improve performance when

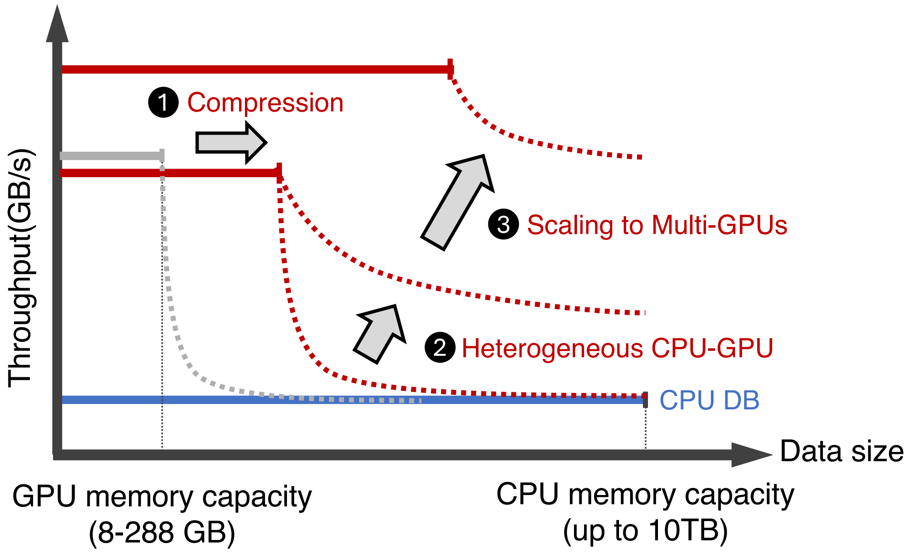


Figure 1.2: Towards Large-Scale Data Analytics on GPUs

workloads exceed available GPU memory. The second part of this thesis (**Part II**) tackles the remaining challenges: implementing complex operators efficiently on GPUs (**Challenge II**) and reducing the switching cost from CPU-based to GPU-based DBMSes (**Challenge III**), both of which are essential for practical adoption of GPU databases.

1.3.1 Towards Large-Scale Data Analytics on GPUs

To enable large-scale data analytics on GPUs, this thesis contributes three key ideas to address **Challenge I**: data compression, heterogeneous CPU-GPU DBMSes, and multi-GPU DBMSes, with their impact illustrated in Figure 1.2.

Data Compression. By compressing data, we can alleviate the GPU memory limitation by (1) fitting in a larger working set within the limited device memory and (2) reducing data transfer overhead across the CPU-GPU interconnect by transmitting compressed data instead. Step ❶ in Figure 1.2 illustrates the benefit of compression. Despite these advantages, the existing GPU data compression solutions often suffer from poor decompression performance - sometimes requiring more time to decompress data than to execute the query itself [68]. To address this, we develop the *tile-based decompression* framework [109] that enables efficient decompression of cascaded compression schemes in a single pass. Furthermore, we introduce a set of massively parallel, GPU-optimized bit unpacking formats—GPU-FOR, GPU-DFOR, and GPU-RFOR— [109] designed for high-throughput decompression on modern GPU hardware. Our schemes can achieve similar compression rates to the best existing scheme while being 2.2× faster in decompression speed and 2.6× faster in total query runtime.

Heterogeneous CPU-GPU DBMS. The next step to enable even larger workload is by leveraging both CPUs and GPUs during query execution. When data no longer fits in GPU memory, heterogeneous CPU-GPU DBMSes allows us to avoid excessive data transfer over the interconnect by executing portions of the query on the CPU. Step ❷ in Figure 1.2 illustrates the benefit of this approach. We improve existing heterogeneous GPU and CPU databases by contributing in two aspects: (1) data placement and (2) heterogeneous query execution strategy. We introduce *semantic-aware fine-grained caching policy* [131] which takes into account query semantics, data correlation, and query frequency when determining data placement between CPU and GPU. We also introduce *segment-level query execution*, a heterogeneous query execution strategy which can coordinate query execution in both devices at a fine granularity [131]. We integrate both solutions in *Mordred* [131] – our heterogeneous CPU-GPU analytical engine – which can outperform existing heterogeneous CPU-GPU databases by 11×.

Scaling to Multiple GPUs. Finally, to scale the workload further, we can extend to multi-GPU environment, gaining access to both greater aggregate memory and computational power. Step ❸ in Figure 1.2 illustrates the benefit of scaling to multiple GPUs. To enable this, we enhance both the data placement and heterogeneous query execution strategy in *Mordred* – our CPU-GPU analytical engine – to target multiple GPUs. To improve data placement, we introduce the *cache-aware replication policy* [128] which takes into account the cost of shuffle when replicating data across GPUs and could coordinate both caching and replication decisions for the best performance. To improve query execution, we extend the *segment-level query execution* strategy [131] with distributed query processing techniques to support multiple GPUs [128]. We integrate these optimizations into *Lancelot* [128] – our hybrid CPU and multi-GPU analytics engine – which achieves 12× speedup over existing multi-GPU DBMSes.

1.3.2 Towards Practical Data Analytics on GPUs

To enable practical data analytics on GPUs, this thesis contributes two key ideas: addressing **Challenge II** by supporting user-defined aggregate functions (UDAFs) on GPUs, and **Challenge III** by delivering a drop-in GPU-accelerated SQL engine.

Accelerating UDAF on GPUs. One of the most challenging operations to accelerate in GPU databases is the user-defined aggregate function (UDAF), which allows users to define custom aggregation logic beyond standard operations such as SUM(), MAX(), or AVG(). We present the first framework to accelerate UDAF execution on GPUs using a combination of a *block-wide execution model* and *just-in-time* (*JIT*) *compilation* [129]. First, we optimize the UDAF execution by mapping each thread-block to operate on each group using block-wide functions and pipeline the whole UDAF execution in a single kernel. Second, we develop a Numba-based JIT compilation framework to compile the UDAF kernel at runtime following the block-wide execution model. Our framework can speedup the UDAF execution by 3600× against Pandas and 8000× against the existing approach on GPUs. **Our framework has also been fully integrated and released in NVIDIA RAPIDS cuDFv23.02.**

A Drop-In GPU-Native SQL Engine. To minimize the switching cost from legacy systems to modern GPU databases, we introduce *Sirius* [130], an open-source GPU-accelerated SQL engine that provides drop-in acceleration across diverse data systems. *Sirius* integrates seamlessly with existing systems via the standard *Substrait* query representation—allowing it to replace CPU-based execution engines without requiring any changes to the data or the user-facing interface. Thanks to its modular design, *Sirius* already supports GPU acceleration for two data systems—DuckDB[105] and Apache Doris[3]—with more integrations planned in the future. Sirius can accelerate DuckDB by 8.2× in a single node setting at the same hardware rental cost, and accelerate Apache Doris by up to 12.5× in a distributed setting. Sirius has gained substantial traction from the community and made it on the front page of Hacker News [30].

1.4 Organization

The rest of this thesis is organized as follows. **Chapter 2** provides a relevant background and related work. The thesis is then divided into two parts.

Part I: Towards Large-Scale Data Analytics on GPUs. This part focuses scaling GPU-accelerated query processing by solving existing limitations in data compression, heterogeneous CPU-GPU execution, and multi-GPU DBMS.

1. **Chapter 3** presents a GPU-efficient decompression framework combined with GPU-optimized bit-packing schemes that achieve competitive compression ratios with significantly lower performance overhead.

2. **Chapter 4** introduces *Mordred*, a heterogeneous CPU–GPU DBMS with optimized data placement and query execution strategies.

3. **Chapter 5** scales out *Mordred* to multiple GPUs by introducing *Lancelot*, our heterogeneous CPU and Multi-GPU DBMS.

Part II: Towards Practical Data Analytics on GPUs. This part focuses on enhancing the practicality and ease-of-adoption of GPU databases.

1. **Chapter 6** focuses on optimizing user-defined aggregate functions (UDAFs) on GPUs, achieving over 8000× speedup compared to existing approaches.

2. **Chapter 7** introduces *Sirius*, a GPU-native SQL engine designed as a drop-in accelerator for existing CPU-based systems (e.g., DuckDB, Doris) without requiring changes to the user facing interface.

Finally, we summarize and discuss future directions in **Chapter 8**.

The work in this dissertation is adapted from a range of publications (SIGMOD'22 [109], VLDB'22 [131], DaMoN'23 [64], VLDB'24 [128], arXiv'25 [130]) and has inspired a number of extensions in the literature [42, 121, 46, 98, 83].

Research in this dissertation has also made impact in the community and industry. The decompression framework presented in Chapter 3 has been adopted by NVIDIA's `nvCOMP` library [19]. The techniques for accelerating user-defined aggregate functions (UDAFs) on GPUs, described in Chapter 6, have been integrated and released into

NVIDIA RAPIDS `cuDF` (v23.02) [9]. Our work on *Sirius* (Chapter 7) has also inspired several optimizations in NVIDIA's `cuCollections` [8] and `libcudf` [9] libraries.

Furthermore, prior to *Sirius*, there was no open-source GPU database for researchers and practitioners to implement and test their ideas. *Sirius* removes this barrier by providing an open-source platform on which the research community can build, integrate, and validate their ideas, ultimately pushing the boundaries of GPU-accelerated data analytics.

Chapter 2

Background

To further understand the opportunities and challenges in GPU-accelerated databases, and the contributions made by this thesis, it is essential to first review the fundamentals of GPU architecture and how existing works leverage GPUs for query execution. This chapter provides the necessary background for the rest of the thesis. We begin with an overview of GPU memory hierarchy and execution model, which largely determine the performance characteristics of database operators on GPUs. We then survey prior work on query execution in GPU-accelerated databases, highlighting the different system architectures and trade-offs for each design.

2.1 GPU Architecture

Performance of database operations on GPU is bound by the memory subsystem (either shared or global memory) [133]. In order to characterize the performance of different algorithms on the GPU, it is, thus, critical to properly understand its memory hierarchy.

Figure 2.1 shows a simplified hierarchy of a modern GPU. The lowest and largest memory in the hierarchy is the *global memory*. A modern GPU can have global memory capacity of up to 288 GB with memory bandwidth of up to 8000 GBps. Each GPU has a number of compute units called *Streaming Multiprocessors* (*SMs*). Each SM has a number of cores and a fixed set of registers. Each SM also has a *shared memory* (SMEM) which serves as a scratchpad that is controlled by the programmer and can be accessed by all the cores in the SM. Accesses to global memory from a SM are

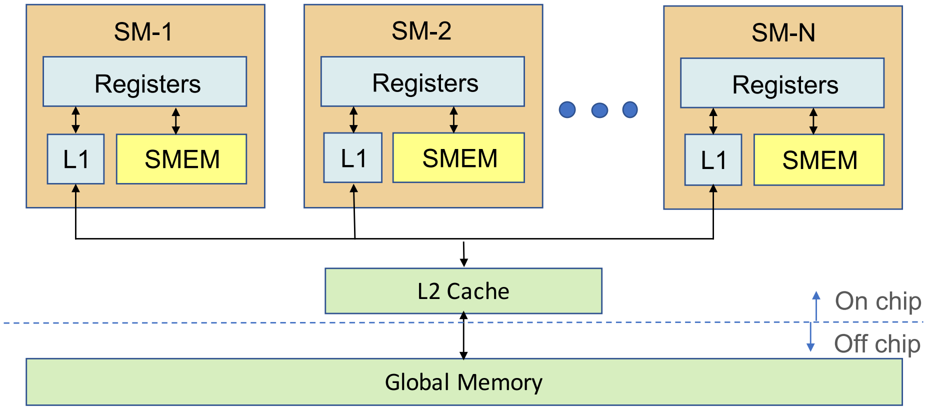


Figure 2.1: GPU Architecture

cached in the L2 cache (L2 cache is shared across all SMs) and optionally also in the L1 cache (L1 cache is local to each SM).

Processing on the GPU is done by a large number of threads organized into thread blocks (each run by one SM). Thread block size can vary from 32 to 1024 threads. Thread blocks are further divided into groups of threads called warps (usually consisting of 32 threads). The threads of a warp execute in a *Single Instruction Multiple Threads* (*SIMT*) model, where each thread executes the same instruction stream on different data. The device groups global memory loads and stores from threads in a single warp such that multiple loads/stores to the same cache line are combined into a single request. Maximum bandwidth can be achieved when a warp's accesses to global memory target neighboring locations.

The programming model allows users to explicitly allocate global memory and shared memory. Shared memory has an order of magnitude higher bandwidth than global memory but has much smaller capacity (a few MB vs. multiple GB). Finally, registers are the fastest layer of the memory hierarchy. If a thread block needs more registers than available, register values spill over to global memory.

2.2 Query Execution on GPUs

A flurry of recent work [70, 92, 82, 133, 110] and a number of commercial systems (e.g, HeavyDB [15], Kinetica [16], and BlazingSQL [6]) use GPU(s) to accelerate query performance. These works can be broadly categorized into three categories.

2.2.1 GPU as the Primary Execution Engine

The first category [15, 16, 110, 6] uses one or more GPUs to store all or significant fraction of the working set directly in GPU memory and aim to deliver interactive query response time; the main constraint is the limited GPU memory capacity. There has been a number of research projects and commercial systems [110, 15, 16, 6, 28, 32] that treat GPU as the primary execution engine. For example, Crystal [110] is a library of CUDA device functions that can execute SPJA analytic queries fully on GPUs. The NVIDIA RAPIDS cuDF [9] is a GPU-accelerated dataframe library that serves as a drop-in replacement for pandas. These systems, however, will either force query execution on the CPU or trigger an out of memory execution error when the data does not fit in the GPU. To allow more data to fit in GPU memory, existing works have also attempted to use multiple GPUs [32, 15, 16] for query execution. The benefit of a multi-GPU DBMS is the increased total GPU memory capacity. These GPUs are connected via NVLink, a modern interconnect with higher bandwidth to mitigate the inter-GPU bottleneck.

2.2.2 GPU as a Coprocessor

A major constraint of treating GPU as the primary execution engine is the limited size of GPU memory, which is often smaller than the size of the data set. One existing solution is to treat GPU as an accelerator in a coprocessor model. In these systems, the data mainly resides on CPU and will be transferred to GPU on demand during query execution. Some previous works in this category focused on accelerating individual database operations such as selection [112], join [101, 106, 79, 80, 86, 111, 107, 125], and sorting [113, 75] in one or more GPUs. To accelerate a single database operation, it is required to transfer the data from CPU to GPU and transfer the result back to the CPU main memory.

Several full-fledged GPU-as-co-processor engines have also been developed in the past [133, 90, 70, 123]. YDB [133] and HippogriffDB [90] stream the data from CPU memory and execute one operator at a time. To reduce the overhead of data transfer, they support compression over input data. Commercial systems such as HeavyDB [15] adopt GPU as a primary execution engine and switch to coprocessor model when the data no longer fits in GPU memory. Systems belonging to this category do not suffer from limited GPU memory capacity since the data mainly resides in CPU. However, the main limitation is the amount of data moving over PCIe, which can easily become a performance bottleneck since PCIe has very limited bandwidth.

2.2.3 Heterogeneous CPU-GPU DBMS

To minimize data transfer while supporting datasets larger than GPU memory, existing works have attempted to use both CPU and GPU for query execution [78, 82, 50, 135, 62, 87, 94, 81, 96, 131, 136]. By partially executing the query on CPU, excessive data transfer can be reduced. For example, CoGaDB [50] and Ocelot [82] cache hot columns in the GPU memory and use a cost-based optimizer [57, 51, 58, 59] to assign operators to either CPU or GPU. HetExchange [62, 63] introduced a query execution framework to encapsulate heterogeneous parallelism in hybrid CPU/GPU system through redesigning the classical *Exchange* operator.

Systems belonging to this category also do not suffer from limited GPU memory capacity. However, it can still suffer from performance bottlenecks due to frequent data transfers between CPUs and GPUs, as well as increased complexity in orchestrating heterogeneous query execution across both processors.

Part I

Towards Large-Scale Data Analytics on GPUs

Chapter 3

Data Compression

One key constraint in enabling large-scale data analytics on GPUs is the GPU memory capacity. Currently, GPUs have at most 288 GB of memory which is used both to cache the working set and as scratch memory for query execution. Therefore, to accommodate working set larger than GPU memory capacity, streaming data from CPUs to GPUs is necessary [92, 132]. However, this approach will incur performance penalty due to slower communication across PCIe or NVLink.

Data compression can play a critical role to address this constraint by achieving two goals: (1) Fit in more working set in a single GPU memory and (2) Reduce the data transfer time across interconnect by transferring the compressed data instead. Furthermore, data compression is also applicable beyond the database field as link bandwidth is often the bottleneck in many applications that use GPUs such as machine learning and image processing.

The current GPU data compression solutions, however, suffers from two major limitations:

1) Cascading Decompression

GPU-based systems [68, 90] have looked at frame-of-reference (FOR) [74, 139], delta coding (DELTA) [89], dictionary compression (DICT) [40, 139], run-length encoding (RLE) [40], and null suppression (NS) [40]. To minimize the size of the compressed data, existing systems cascade multiple compression schemes such that the output of one scheme is the input to another scheme. The database engine decompresses one layer at a time. Such a cascading decompression strategy leads to suboptimal performance as multiple GPU kernels are launched and each requires reading from and writing data back to the global memory. This causes high memory traffic which may lead to much worse performance compared to a design that does not use any compression. In this work, we improve decompression performance by treating a *thread block* as the basic decompression unit, that decodes one block of encoded entries. This way, we are able to cache a block of data in on-chip caches and inline multiple decoding steps into a single kernel, resulting in a single pass over the data. Moreover, we can even inline the decompression with query execution, resulting in a single round-trip to the global memory for the whole decompression step and query execution.

We call this *tile-based decompression model* which is inspired by Crystal [110]. *Tile-based decompression* allows us to decode at close to memory bandwidth speed, resulting in a very low decompression overhead over the previous compression work for GPU. It also eliminates the need for sophisticated compression planners used by past works [68, 90], since instead of balancing the trade-off between decompression time and compression ratio, we can simply choose the scheme with the best compression ratio — all schemes achieve similar performance.

2) Efficient Decompression of Bit-Packed Schemes

Besides the organization of different compression schemes, the decompression speed of individual schemes is also limited in GPU today. Prior works on CPU compression [66] have shown that bit-aligned packing compression schemes manage to achieve better compression ratio compared to byte- and word-aligned packing. Supporting efficient bit-level packing in GPU is challenging due to the SIMT programming model and the relatively limited instruction set to perform bit-level alignment, which is not a problem in CPU compression schemes due to more powerful CPU instructions.

In this work, we design optimized bit-packing based compression schemes and their optimization techniques in the context of GPU. Specifically, we introduce three new bit-packing based compression schemes: `GPU-FOR` does bit-packing in conjunction with Frame-Of-Reference (FOR) and work well with uniform data and can handle skew; `GPU-DFOR` uses delta encoding with bit-packing and FOR, targeting sorted or semi-sorted data; `GPU-RFOR` uses RLE encoding with bit-packing and FOR, targeting data with high average run length. These schemes are designed to offer improved compression ratios while still being able to decode data in parallel across thousands of threads at close to memory bandwidth speeds. Overall, our compression schemes can be decoded 2.2× faster than previous implementations.

We integrate the tile-based decompression model and GPU-FOR, GPU-DFOR, and GPU-RFOR into the Crystal framework [110], an open-source highly optimized GPU data analytics engine. We encapsulate decompression into a device function that enables programmers to change a kernel operating on an uncompressed array to a compressed column with a single line of code. Our compression schemes target integer, decimal, and dictionary-encoded strings and support all query operations that run on these data types. In data analytics workloads, prior works [47, 95, 110] and commercial systems [15] typically apply dictionary encoding on top of string columns to encode them to integers.

Contributions. This work makes the following contributions:

- We introduce *tile-based decompression*, a decompression strategy that allows us to decode the data in a single pass at close to memory bandwidth speed and inline with query execution.
- We present three optimized bit-packing based compression schemes (GPU-FOR, GPU-DFOR, and GPU-RFOR) that can be used to store data compactly on the GPU.
- We present an integration of the decompression routines into the Crystal framework and demonstrate ease of use.
- We present an evaluation[1] on multiple synthetic benchmarks and on the Star Schema Benchmark (SSB). On SSB, our schemes can achieve similar compression rates to the best state-of-the-art compression schemes in GPU (i.e., nvCOMP) while being 2.2× and 2.6× faster in decompression speed and query running time.

Organization. The rest of the chapter is organized as follows: related work and background are discussed in Section 3.1.1. We introduce the *tile-based decompression model* in Section 3. We present the data format and the unpacking implementation on the GPU for GPU-FOR, GPU-DFOR, and GPU-RFOR in Section 3.3, 3.4, and 3.5 respectively. Section 3.6 discusses the integration into Crystal. Section 3.7 discusses the usage and choice of compression scheme, parameter, and other relevant discussion Section 3.8 evaluates the performance and compression ratio of our approach against other schemes on GPU. Finally, we conclude in Section 3.9.

[1] The source code is available at `https://github.com/anilshanbhag/gpu-compression`

Acknowledgement. The design and implementation of GPU-FOR (Section 3.3) and GPU-DFOR (Section 3.4) were contributed by Anil Shanbhag, whose work I gratefully acknowledge.

3.1 Background

3.1.1 Compression Schemes

Lossless compression techniques have been heavily exploited in modern column-store databases for efficient query processing and can be categorized into two buckets: *lightweight* and *heavyweight*. Lightweight algorithms are mainly used in in-memory column stores while heavyweight algorithms like Huffman [84] and Lempel Ziv [137] (together with lightweight techniques) are used in disk-based column stores. In this work we focus on lightweight techniques. We show later in Section 3.8 that most of the compression gains can be achieved with just lightweight techniques.

There are five basic lightweight techniques to compress a sequence of values: frame-of-reference (FOR) [74, 139], delta coding (DELTA) [89], dictionary compression (DICT) [40, 139], run-length encoding (RLE) [40], and null suppression (NS) [40].

FOR represents each value in a sequence as a difference to a given reference value. FOR is applied to a block of integers and the reference value chosen is usually the minimum value to make all values positive. FOR is good when the block of integers have similar values.

DELTA represents each value as a difference to its predecessor value. DELTA is good when the array is sorted or semi-sorted.

DICT replaces each value by its unique key in the dictionary. DICT is effective for columns with low cardinality.

RLE replaces uninterrupted sequences of occurrences of the same values (called runs) by the value and length of the sequence. Hence a sequence of values is replaced by a sequence of pairs (value, length).

NS removes leading zeros from an integer's bit representation. NS is useful when a column contains many small integers.

FOR, DELTA, DICT, and RLE work at the logical level where a sequence of values is compressed into another sequence. NS addresses the physical level of bits with the basic idea of removing leading zeros in the bit representation of small integers. There are many different NS techniques proposed which can broadly be categorized as (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned. Bit-aligned NS algorithms compress an integer to a minimal number of bits; byte-aligned NS compress an integer with a minimal number of bytes; word-aligned NS encodes as many integers as possible into 32/64-bit words. The NS algorithms also differ in their data layout. We distinguish between *horizontal layout* [120, 91] and *vertical layout* [69, 91, 89]. In the horizontal layout, the compressed representation of subsequent values is situated in subsequent memory locations. In the vertical layout, each subsequent value is stored in a separate memory word in a striping fashion. Cascading multiple compression schemes together has been done on both CPU and GPU to achieve a better compression ratio [68, 90, 89, 19], where the input of one scheme is the output of another scheme.

CPU-based compression. Wang et al. [118] presented a survey of bitmap and inverted list compression in the area of database and information retrieval. Among these schemes, VB [65, 67] is a variant of NS algorithm with variable byte-aligned packing. PFOR and its variants [126, 138] encode a block of integers such that the majority of the integers can be encoded into b-bits and store the rest of the integers at the end. Simple-N and its variants [44, 43, 134] are word-aligned compression methods with 4 status bits to represent N combinations of bitwidth and 28 data bits used to store the data. All these schemes are typically a NS algorithm cascaded with Delta/RLE since they run on sorted datasets. Li and Patel [91] proposed Bitweaving which includes Horizontal Bit-Parallel (HBP) and Vertical Bit-Parallel (VBP) storage layouts. HBP achieves better decompression performance but VBP achieves better compression rates. ByteSlice [69] improves the performance of VBP by striping in bytes instead of bits but at the cost of larger storage footprint.

Across various compression schemes, the best performing ones are `SIMD-scan` [120, 104] which uses bit-aligned packing with a horizontal data layout and `SIMD-BP128` [89] (and its variants) which use bit-aligned packing with a vertical data layout. `SIMD-scan` stores column values in a tightly bit-packed horizontal layout, ignoring any byte boundaries. `SIMD-BP128` processes data in blocks of 128 integers at a time and stores these integers in a vertical layout using the number of bits required for the largest of them. Figure 3.1 illustrates the vertical layout where the first four integers Int1, Int2, Int3, Int4 start out in four different 32-bit words. Int5 is immediately adjacent to Int1,

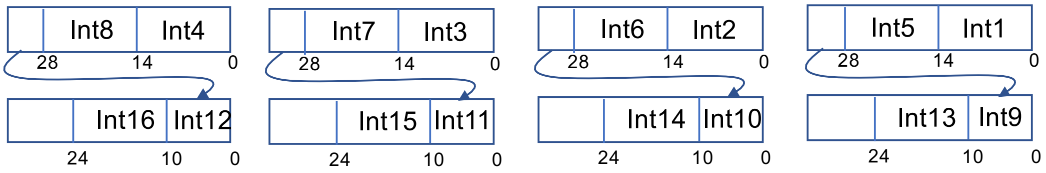


Figure 3.1: Bit-packing with vertical data layout

Int6 is adjacent to Int2, etc.

GPU-based compression. Fang et al. [68] and HippogriffDB [90] support the cascading of five basic lightweight techniques. They evaluated two byte-aligned schemes: NSF and NSV. NSF encodes every element in the input array with a fixed number of bytes whereas NSV allows each value to be encoded in a variable length of bytes. To generate the compression schemes, these designs use a compression planner to generate a plan for each column. Based on the column properties (i.e., sorted/unsorted, average run length, number of distinct values, etc.), the planner will generate a plan with the best compression ratio. During decompression, these designs treated each compression scheme as independent layers. Therefore, during decompression of cascaded scheme, different kernels will be called in succession to decode one compression layer at a time (see Figure 4 (left)). This leads to suboptimal decompression performance.

Mallia et al. [93] introduced two new NS algorithms (`GPU-BP` and `GPU-VByte`). `GPU-BP` encodes the data in a horizontal layout similar to `SIMD-Scan`. `GPU-VByte` decodes the input array with a variable length of bytes similar to NSV in [68]. This work, however, does not support cascading compression schemes which limits its compression rate. nvCOMP [19] is a generalized CUDA-based compression library that supports cascaded compression schemes consisting of the five basic lightweight techniques. Their bit-packing scheme does not saturate memory bandwidth. Further, nvCOMP does not support the end-to-end pipelining of multiple decompression steps with query execution. This leads to suboptimal performance during query execution. We will evaluate the performance of all the previous GPU compression schemes in Section 3.8.4.

3.1.2 Tile-based Execution Model

A recent work, Crystal [110], presented a library of CUDA device functions that can be composed together to execute analytic queries on GPUs. Crystal adopts the idea of *tile-based execution model*, which instead of viewing each thread as an independent execution unit, views a thread block as the basic execution unit with each thread block processing a tile of entries at a time. Compared to a single thread, a single thread block can hold a significantly larger group of elements collectively in shared memory. This group of elements is called a *tile*. The key advantage of this model is that after a tile is loaded into shared memory, subsequent passes over the tile will be read directly from shared memory, avoiding multiple round-trips to the global memory. As a result, Crystal could execute analytic queries close to memory bandwidth speed.

3.2 Tile-based Decompression

Cascading compression schemes achieve better compression ratio than individual compression schemes [68, 90]. However, decompressing a cascaded scheme can degrade performance as intensive decompression overburdens the GPU. The reason behind the bad performance was that past work decoded one layer of compression at a time and thus required multiple passes to decode the compressed data and execute the query. Figure 3.2 (left) illustrates this approach for decompression of table data consisting of delta encoding, frame of reference, and fixed length byte-aligned packing (`Delta+FOR+NSF`). To fully decode the column and execute the query, these systems have to (1) do a first pass to unpack the byte-packed data (decompress `NSF`), (2) do a second pass to add the data with the base reference (decompress `FOR`), (3) do a third pass to delta decode the data (decompress `Delta`), and finally (4) launch the query kernel on top of the the fully decoded data. Step (2) and (3) can potentially be merged together into a single step — adding the reference while unpacking the byte-packed data. Prior works [68, 90], however, separated the kernels to support more flexible compression plans. If the query requires multiple encoded columns, steps (1)–(3) will have to be repeated for each column. In this model, decompression is expensive since the intermediate data is read and written to the global memory after every kernel pass, incurring significant memory traffic. We will refer to this model as the *cascading decompression model*.

In this work, we introduce the *tile-based decompression model* which is inspired by *tile-based execution model* [110]. In the tile-based decompression model, each *thread block* collectively loads a block of encoded data into shared memory, which is called a tile. Next, multiple steps of decompression are applied on the tile directly in shared memory, avoiding multiple passes to the global memory. Figure 3.2 (right) shows

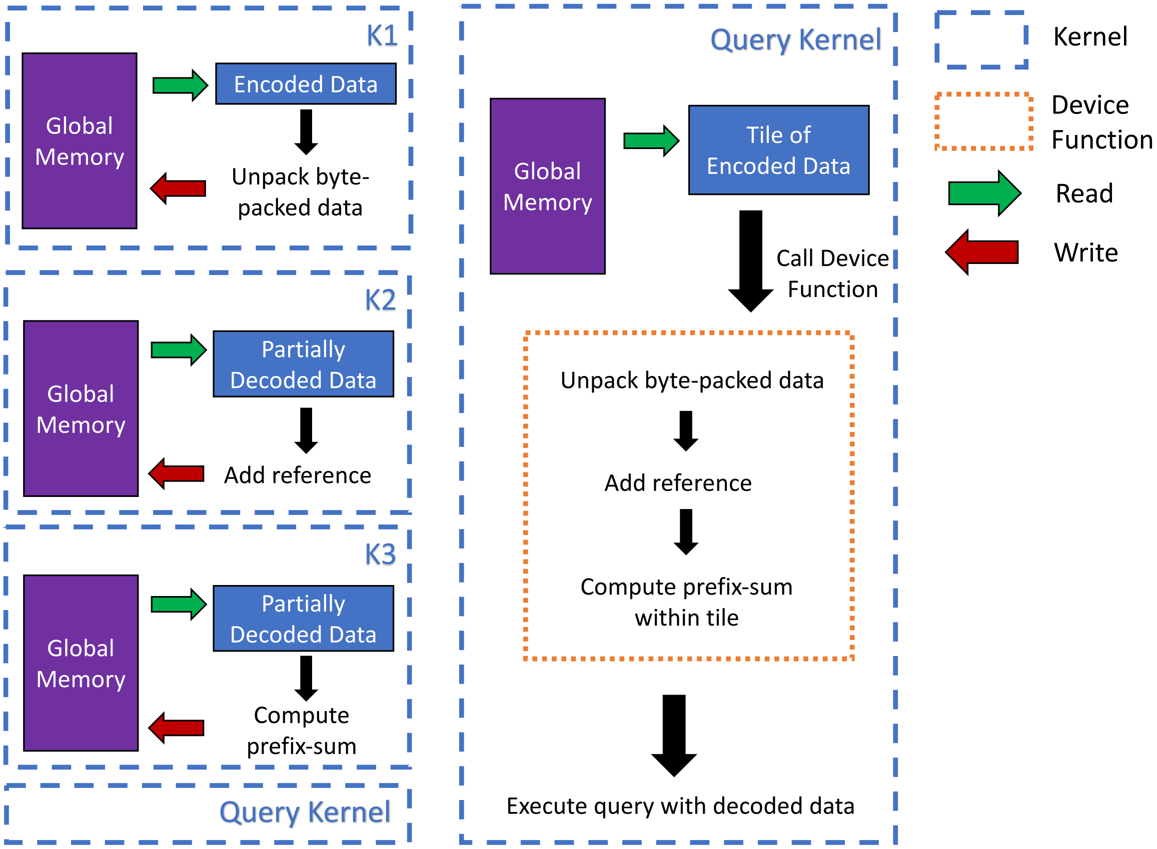


Figure 3.2: Decoding cascaded compression scheme (Delta+FOR+NSF) using *cascaded decompression* (left) and using *tile-based decompression* (right)

how to decompress the cascaded scheme `Delta+FOR+NSF` with *tile-based decompression*. The three decompression steps can now be encapsulated into a single function which can be called from inside the query kernel during execution. This would enable us to decompress cascaded compression schemes in a single pass over the column. Further, this decompression can be inlined with query execution. Compared to *cascading decompression*, the intermediate data transfer of *tile-based decompression* is X times less, where X is the depth of the compression layers.

Applying *tile-based decompression* requires each compression scheme in the cascade to have the following two properties:

Property 1: Tile-granularity data format
In tile-based model, the data needs to be partitioned and encoded in the granularity of tiles where each tile fits in shared memory. This allows us to decode each tile independently.

Property 2: Tile-based decompression routine

During decompression, we want to read the encoded data from global memory only once. To achieve this for cascaded schemes, we should be able to express the decompression routine as a function that takes a tile in shared memory as input and outputs a tile to shared memory.

In the next three sections, we will show that FOR, Delta, RLE and bit-aligned NS obey the above two properties. We will introduce three new bit-packing based compression schemes (GPU-FOR, GPU-DFOR, and GPU-RFOR) and discuss how they can be integrated with the *tile-based decompression model*. We will also discuss the integration with Crystal system in Section 3.6.

3.3 Fast Bit Unpacking

While Section 3.2 has demonstrated that *tile-based decompression* could solve the cascading decompression problem in GPU. However, individual compression schemes such as bit-packing have not been sufficiently optimized on GPU. Addressing this issue will be the main focus of the next three sections.

In this section, we describe the GPU-FOR[2] compression format, which uses bit-packing in conjunction with Frame-of-Reference (FOR) to store data compactly on the GPU and the fast bit unpacking routine used to decompress it efficiently on the GPU. GPU-FOR can be used to efficiently compress attributes of type integer, decimal, or dictionary-encoded string (i.e. sequence of integers) in a column store. At runtime, the executor decompresses data and runs the query on the decompressed data. Hence, optimizing the performance of decompression is critical for analytic workloads. In the rest of the section, we describe first the bit-packing representation we use and then the kernel implementation on the GPU. We present a series of optimizations that allow us to decode bit-packed data while saturating memory bandwidth.

3.3.1 Data Format

Bit-packing is a process of encoding small integers in $[0, 2^b)$ using b bits; b can be arbitrary and not just 8, 16, 32, or 64. Each number is written using a string of length b. Bit strings of fixed size b are concatenated together into a single bit string, which can span several 32-bit words. If some integer is too small to use b bits, it is padded

[2] GPU-FOR was designed and implemented by Anil Shanbhag – a collaborator in the project

with zeros. Compressing 32-bit integers to b bits achieves a compression ratio of 32/b, which can be significant.

Choosing a common bit size b for an entire array would mean that the occurrence of a single large value would increase the number of bits needed to encode the values. Hence, bit-packing is generally used in conjunction with FOR encoding. In GPU-FOR, the array of values is partitioned into *blocks* of 128 integers, which is the tile that will be processed by a thread block. The range of values in the block is first found and then all the values are written in reference to the minimum value. For example, if the values in a block are integers in the range [100,130], then using a reference of 100, we can store them using 5 bits $(\log_2(130 + 1 - 100))$. Each block is further divided into sequences of 32 integers called *miniblocks*. For each miniblock, we choose a bit-width based on the maximum number of bits needed to encode the largest value.

The choice of block size and miniblock size is to ensure they align on a 32-bit boundaries (shared memory access granularity). Having 32 integers in one miniblock means that for any bitwidth, the miniblock always ends on 32-bit boundary. This would allow us to use 32-bit arithmetic while decoding and make shared memory accesses efficient. The bitwidth for a miniblock can then be stored in 1 byte. Since we want to align along 32 bit boundaries, we group 4 miniblocks into a single block and store the 4 bitwidths at the start of the block using a single integer. Hence, each block now consists of 128 integers.

The bit-packed array needs to be decoded in parallel across a large number of threads. For this, we store the start index of the blocks in a separate array called block starts. Finally we store the metadata associated with the encoding: block size (i.e., the number of integers within each block), miniblock count (i.e., number of miniblocks per block), and the total count (i.e., total number of integers in the data array) in the header. Figure 3.3 shows a schematic of the format we use to store data.

Figure 3.4 shows an example of encoding 16 integers into a block with 2 miniblocks. The minimum value in the block (i.e., 99) is used as the reference. We calculate the difference of the block values from the reference. Each miniblock contains 8 integers. We see that the first miniblock needs 2-bits per block while the second miniblock needs 4-bits per block. We encode each miniblock with their respective bitsizes and store the reference and bitwidths at the start of the block.

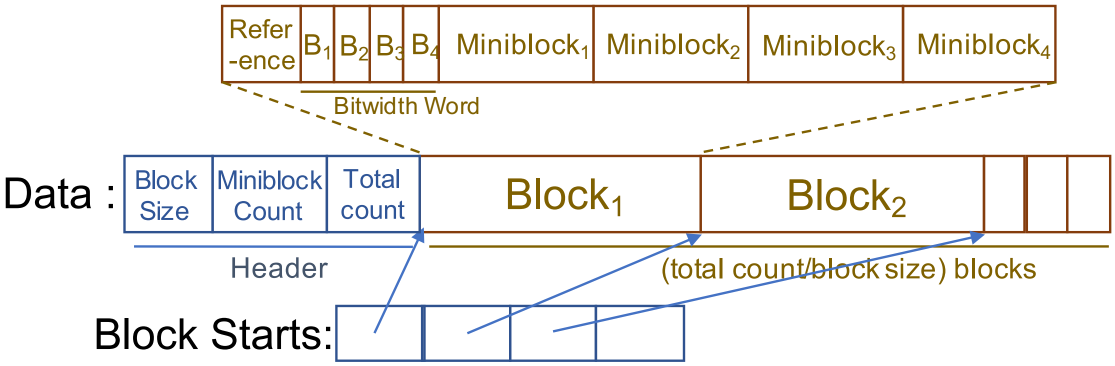


Figure 3.3: GPU-FOR Data Format

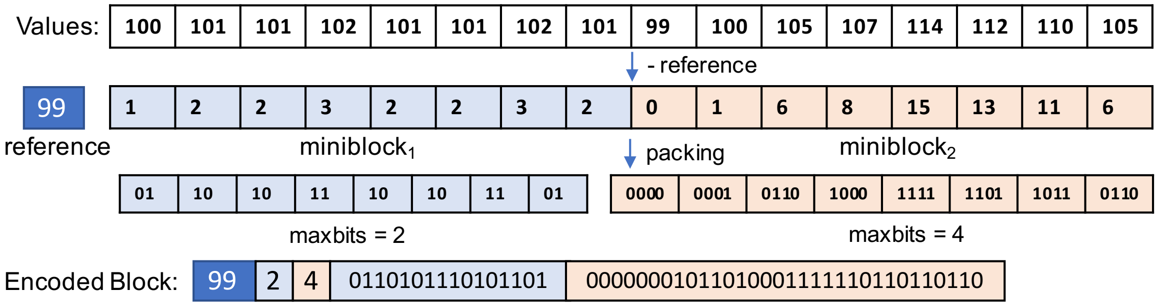


Figure 3.4: Example encoding with GPU-FOR

3.3.2 Implementation

In this section, we describe a number of optimizations at the implementation level that we applied to achieve decompression at close to GPU memory bandwidth speed. To give an impression of the importance of each optimization, we end every subsection with the time taken to decode a compressed dataset of 500 million integer values drawn from a uniform distribution $U(0, 2^{16})$. The details of the experimental setup can be found in Section 3.8.1.

Base Algorithm: Algorithm 1 shows the pseudocode that would run in parallel on each thread (n threads are allocated for n-element dataset). Each thread block (of size 128 threads) is assigned to decode a block (of 128 elements) with each thread decoding one element in the block based on its index. Each thread starts by reading the block start pointer of the block to find where in the data array the block starts (line 1–2). Each thread then reads in the `bitwidth_word`, and uses it to compute the offset of its miniblock in the data array (`miniblock_offset`) (lines 7–10). In computing the miniblock offset, we use the fact that if entries in a miniblock are encoded with b bits, then the miniblock occupies $4b$ bytes (since there are 32 entries per miniblock). Next,

Algorithm 1: Fast Bit Unpacking on GPU — The following code runs on each of the 128 threads within a thread block in parallel.

```
   Input  : int[] block_starts; int[] data; int block_id;
            int thread_id
   Output : int item
1  int block_start = block_starts[block_id];
2  uint * data_block = &data[block_start];
3  int reference = data_block[0];
4  uint miniblock_id = thread_id/32;
5  uint index_into_miniblock = thread_id & (32 - 1);
6  uint bitwidth_word = data_block[1];
7  uint miniblock_offset = 0;
8  for j = 0; j < miniblock_id; j++ do
9      miniblock_offset += (bitwidth_word & 255);
10     bitwidth_word ≫= 8;
11 uint bitwidth = bitwidth_word & 255;
12 uint start_bitindex = (bitwidth * index_into_miniblock);
13 uint header_offset = 2;
14 uint start_intindex = header_offset + miniblock_offset + start_bitindex/32;
15 uint64 element_block = data_block[start_intindex] | (((uint64)data_block[start_intindex +
     1]) ≪ 32);
16 start_bitindex = start_bitindex & (32-1);
17 uint element = (element_block & (((1≪bitwidth) - 1) ≪ start_bitindex)) ≫ start_bitindex;
18 item = reference + element;
```

we compute the offset in bits within the miniblock (line 12). Since the entries are bit-packed, they are not byte-aligned and can span byte boundaries. Using starting bit index, we calculate the starting integer index (`start_intindex`) of the entry (lines 13-14). We then load an 8-byte block starting at `start_intindex` (`element_block`) (line 15). This block contains the entire element, we use bitshift arithmetic to extract the entry (lines 16–17). Finally, we add `reference` and return the result. The result resides in a register and is used subsequently during query execution. In Section 3.6, we describe in greater detail how the algorithm is used during query execution.

To optimize performance, the compressed block can be loaded entirely into shared memory at the start of the operation. All subsequent requests are then directed to shared memory, which offers an order of magnitude higher bandwidth than global memory, thereby improving performance. Furthermore, to reduce irregular memory access, each thread block can process D data blocks instead of just one. Our evaluation shows that choosing D = 4 significantly increases the global memory coalescing and achieves the best performance. Hence, we use $D = 4$ for the rest of the chapter.

3.4 Fast Delta Decoding

Delta encoding (also called differential encoding) is a common approach used (typically in conjunction with other techniques) to compress sorted or partially-sorted integer/decimal arrays. Instead of storing the original array of integers $(x_1, x_2, x_3...)$, delta encoding keeps only the difference between successive integers together with the initial integer $(x_1, \delta_2 = x_1 - x_1, \delta_3 = x_3 - x_2, ..)$. Since the differences are typically much smaller than the original integers, delta encoding allows for more efficient compression. In this section, we describe the `GPU-DFOR`[3] coding scheme that uses delta encoding in conjunction with bit-packing and frame of reference to achieve good compression ratios and can be decoded efficiently.

The sequential form of delta encoding requires just one subtraction per value $(\delta_i = x_i - x_{i-1})$. During decoding, we require one addition per value $(x_i = \delta_i + x_{i-1})$. For an array A of k elements, the prefix sum p_A is a k-element array where $p_A[j] = \sum_{i=0}^{j-1} A_j = p_A[j-1] + A_j$. Hence, the process of delta decoding is equivalent to computing the prefix sum. Efficient parallel prefix sum routines have been proposed [77] that could be used to decode delta encoded data on the GPU. A simple approach to delta encode + bit-pack the data would be to do it as two separate steps: first compute the deltas for the entire array and then bit-pack the deltas. The decoding would then be a two-step process: the first pass would use the bit unpacking routine described in Section 3.3.2 to decode the deltas and write it to global memory; the second pass would use the prefix sum routine to decode the data. This is the approach used by past work [68, 90] and is inefficient as it requires multiple passes over the data. Later in this section we describe how we can combine the delta decoding step and the bit unpacking step into a single pass.

3.4.1 Data Format

Delta encoding an entire array as x_0, δ_1, δ_2... makes it hard to parallelize as decoding the n^{th} block requires the $(n-1)^{th}$ block to have been decoded already. To enable parallel decoding, we build our data format on `GPU-FOR` encoding scheme (Section 3.3.1) by

[3] `GPU-DFOR` was designed and implemented by Anil Shanbhag – a collaborator in the project

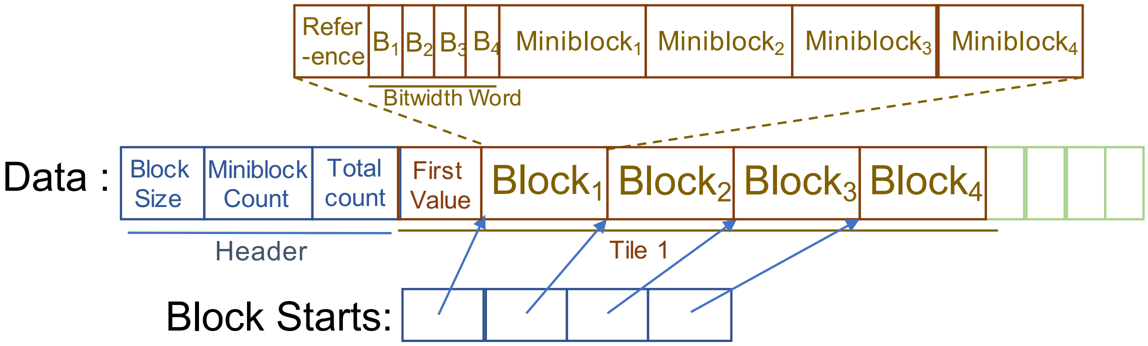


Figure 3.5: GPU-DFOR Data Format

partitioning the array into sets of D blocks where each block contains 128 integers and delta encoding each set of D blocks independently (where D is the number of blocks processed per thread block), as shown in Figure 3.5. Encoding x integers generates $x - 1$ deltas. During encoding, we pad the deltas with 0 to ensure every block has 128 entries. We store the first value separately before every D^{th} block, with start pointers still pointing to the start of each block.

GPU-DFOR compresses better than GPU-FOR on sorted and semi-sorted arrays, e.g., sorted primary keys or posting lists in search workloads. Consider a dataset of $n = 500$ million integers with entries from 1 to n sorted. This dataset when compressed using GPU-DFOR uses 1.8 bits per integer vs 7.8 bits per integer used by GPU-FOR. Using it for unsorted data could lead to worse compression ratios compared to simply bit-packing the data. For example, for an array of integers drawn uniform randomly from $[0, 32)$, the deltas will be in the domain $[-31, 31]$. In this case, GPU-DFOR will likely require more bits per integer than GPU-FOR.

3.4.2 Implementation

With the data format described above, each tile of D blocks can be decoded independently. During decoding, we first start by loading the D block segments into shared memory and use the fast bit unpacking routine (described in Section 3.3.2) to decode the deltas.

After bit unpacking the deltas, we have D deltas per thread. The output data entries can be calculated using prefix sum over the deltas of all threads within the thread block. We can use block-wide prefix sum to compute the prefix sum over the deltas based on the work-efficient prefix sum algorithm proposed by Blelloch et al. [45]. For an array of n integers, the algorithm is able to compute the prefix sum of the array in parallel using $\Theta(\log n)$ steps. Interested reader can refer to [45, 110] for more details.

Although prefix sum has been used widely in libraries like Thrust [35], doing prefix sum over an entire array is expensive and involves multiple passes over data. A key observation we make is that since we do delta coding in each tile (set of D blocks) separately, prefix sum can happen entirely within a thread block in shared memory. This also allows us to fuse bit unpacking and delta decoding steps into a single kernel which allows our implementation to perform decompression in a single pass over the data blocks in global memory compared to multiple passes required by previous works [68, 90]. The bit unpacking and delta decoding share the same shared memory buffer. In our implementation, we reuse an existing block-wide prefix sum implementation from Crystal. The total resource requirement of the kernel is D 4-byte entries in shared memory and D registers to store the output per thread.

3.5 Fast Run Length Decoding

Run length encoding (RLE) is a common approach used to compress a data sequence with consecutive runs of values. For example, given an original array of integers $(x, x, x, x, y, y, z, z, z)$, run length encoding stores the original array as two separate arrays: the values (x, y, z) and the run length of the values $(4, 2, 3)$. For data set with high average run length, RLE can greatly reduce the total memory footprint. In this section, we describe the `GPU-RFOR` coding scheme that uses RLE in conjunction with bit-packing and frame of reference to achieve good compression ratios and efficient decompression time.

A simple approach to RLE + bit-pack the data contains two steps: first generate the run length and values arrays and then bit-pack both arrays. To decode it, we would use the bit unpacking routine described in Section 3.3.2 to decode the run length and values array. After that, we decode RLE by using the decompression routines as described in [68]. This includes making 4 passes two the global memory in which the two are inclusive prefix sum. Interested readers can refer to [68] for a more in-depth explanation. Later in this section we describe how we can combine the decompression steps into a single pass.

To enable parallel decoding, we once again build our data format on the `GPU-FOR`

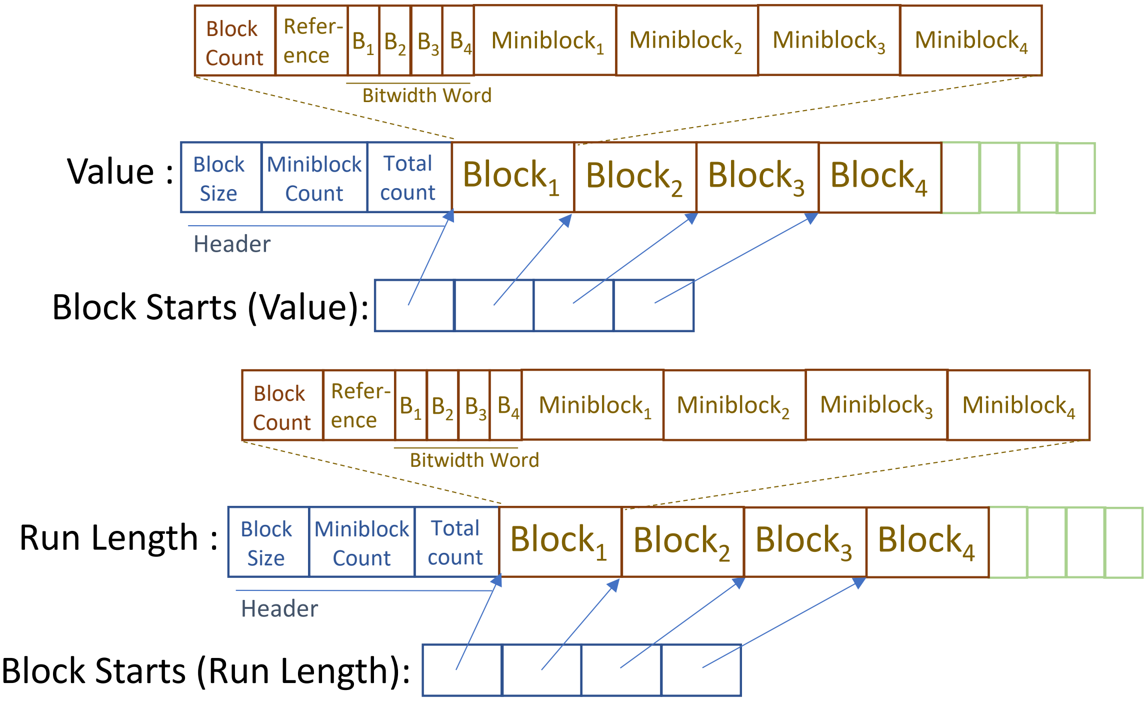

Block Count
Refer-ence
B_1 B_2 B_3 B_4
Miniblock_1 Miniblock_2 Miniblock_3 Miniblock_4
Bitwidth Word
Value :
Block Size
Miniblock Count
Total count
Block_1 Block_2 Block_3 Block_4
Header
Block Starts (Value):
Block Count
Refer-ence
B_1 B_2 B_3 B_4
Miniblock_1 Miniblock_2 Miniblock_3 Miniblock_4
Bitwidth Word
Run Length :
Block Size
Miniblock Count
Total count
Block_1 Block_2 Block_3 Block_4
Header
Block Starts (Run Length):

Figure 3.6: GPU-RFOR Data Format

encoding scheme (Section 3.3.1) by partitioning the array into blocks of 512 integers and applying RLE to each block independently to create the value array and the run length array. We then apply FOR on top of both arrays separately. In addition to reference and bitwidths (see Section 3.3.1), we store extra metadata of the run length-/values count at the beginning of each block. The two compressed representations of run length and values arrays are stored separately.

With the data format described above, each block can be decoded independently. During decoding, we first start by loading one block of compressed run length and one block of compressed values into shared memory and use the fast bit unpacking routine (described in Section 3.3.2) to obtain the uncompressed run length and values array for the corresponding block. After bit unpacking both columns, the output data block entries can be calculated using the four steps described in [68] for all the threads within the thread block. These steps include 2 scatter and 2 inclusive prefix sum operations. We use the same block-wide prefix sum as the one described in Section 3.4.

Since each block can be decoded independently, the four decoding steps of RLE can be served completely by the shared memory. This allows us to fuse bit unpacking and run length decoding steps into a single kernel which allows our implementation to perform decompression in a single pass over the data blocks in global memory. GPU-RFOR, however, requires twice more resources than GPU-DFOR since there are two input columns (value and run length) resulting in twice more blocks to be loaded into the shared memory.

3.6 Database Integration

Given the efficient massively parallel bit-unpacking implementations described in the previous sections, we were naturally interested in its usability in a full system. As a proof of concept, we implemented the decompression routines as CUDA device functions[4] and show how they can be used with an existing GPU analytical engine. In particular, we chose Crystal [110], an open-source GPU analytics framework developed recently.

Crystal is based on the idea of a *tile-based execution model*. Previous work [110] has shown that SQL query operators and analytical queries implemented with Crystal can saturate memory bandwidth and thereby deliver an-order-of-magnitude speedup compared to CPU-based implementations. We have implemented the decompression routines for GPU-FOR, GPU-DFOR, and GPU-RFOR as CUDA device functions LoadBitPack, LoadDBitPack, and LoadRBitPack respectively. These functions can be used in queries implemented in Crystal easily and can be used more broadly in any CUDA kernel as well. To integrate them into Crystal, the only required changes are to replace the load routines (BlockLoad) in Crystal with LoadBitPack. Therefore, the user can run the query on compressed data by just changing a single line of code. Readers interested in Crystal can refer to [110].

One key drawback of bit-packed data is that it lacks random access. Accessing any element requires loading the entire data block. As a result, when selections or joins filter data entries, we still have to read the entire column. We will show in Section 3.7 why this would not lead to material impact on performance.

Since the routines LoadBitPack, LoadDBitPack, and LoadRBitPack are ordinary device functions, they can be used directly in user's CUDA code in conjunction with other GPU frameworks like Thrust and Cub [35], and they can also be called directly from NVVM (a compiler internal representation based on LLVM IR designed to represent GPU compute kernels) [23].

[4] Device functions are functions that can be called from kernels on the GPU

3.7 Discussion

In this section, we discuss certain key aspects that we haven't covered with respect to usage and choice of compression method.

Choice of Compression Scheme: The rule-of-thumb when choosing a compression scheme is to use the one that has the lowest storage footprint for each column. Therefore, data distribution plays a very important role in such cases. In general, `GPU-DFOR` is suitable for sorted or semi-sorted columns with high number of distinct values. `GPU-RFOR` is suitable for columns which have a low number of distinct values or columns with a high average run length. Other columns which do not have such properties are typically suited for `GPU-FOR`. Since each column can be decoded with different schemes, we will refer to this hybrid scheme as `GPU-*` for the remainder of the dissertation. We will show how `GPU-*` performs when we measure the end-to-end system performance in Section 3.8.

Hyperparameter Tuning: The number of blocks processed per thread block D is the only hyperparameter in the schemes we propose. As discussed in 3.3.2, we will choose $D = 4$ in our evaluation. As GPUs improve, it is likely they will have more shared memory and registers per thread, thereby allowing us to use higher values of D during query processing.

Compression Speed: In data analytics workload, data compression is generally a one-time activity that happens on the CPU side. However, in the event of updates, the data need to be recompressed and transferred again to the GPU memory to replace the old data. We measure the time required to compress the data to simulate such cases. Compressing 250 million entries of random dataset using `GPU-FOR` and `GPU-DFOR` on a 6 cores CPU machine takes approximately 1.2s and 1.3s respectively. `GPU-RFOR` is not suitable for this distribution and therefore takes longer to compress (2.2s).

Out-of-core Dataset: In the event that the compressed dataset does not fit in the GPU memory, the GPU as a co-processor model can be leveraged. In such case, the compressed data will be transferred from CPU to GPU before running each query. Compression still brings benefits since it reduces the total amount of data being transferred through the limited PCIe interconnect. We show the benefit of compression for out-of-core dataset in Section 3.8.5.

Random Access Performance: We measure and compare the performance of our compression schemes against uncompressed data under random access patterns. To simulate random access behavior, we generate a predicate bitvector to filter random 250 million data entries and sweep the selectivity from 0 to 1. For our compressed schemes (GPU-FOR, GPU-DFOR, and GPU-RFOR), we achieve good performance when $\sigma < 1/\mathtt{TILE\_SIZE}$ since we can skip loading the entire compressed tile. When $\sigma > 1/\mathtt{TILE\_SIZE}$, however, we will have to load the entire compressed tile from the global memory and decompress it before applying the bitvector. This result in the constant performance of 2.1 ms for GPU-FOR and GPU-DFOR.

For uncompressed data, since the granularity of access in GPU global memory is 128B, when the selectivity is above ($\sigma > 1/32$), we would have to read the entire cache line for every random access which results in reading the whole dataset. This results in the constant performance of 2.5 ms. Our performance is better since the reduction in data size reduces the total data read which often compensates for the loss of efficiency in the case of a selective filter. This shows that random access does not have a material impact on performance in a compressed setting.

3.8 Evaluation

In this section, we evaluate the performance characteristics of the different compression schemes (GPU-FOR, GPU-DFOR, and GPU-RFOR) to understand (1) impact of applying *tile-based decompression*, (2) impact of data distribution, (3) performance within SQL queries, and (4) performance of our encoding schemes compared to the existing schemes from previous works.

The rest of the section is organized as follows: we discuss the experiment setup in Section 3.8.1. In Section 3.8.2 we evaluate the performance of different encoding schemes on synthetic dataset with varying bitwidths and illustrate the impact of tile-based decompression on our encoding schemes. In Section 3.8.3, we evaluate the impact of data distributions to different encoding schemes. In Section 3.8.4, we evaluate the end to end systems performance against previous works using the Star Schema Benchmark. Finally, in Section 3.8.5, we discuss the case when GPU is used as a coprocessor.

3.8.1 Setup

Hardware configuration: For the experiments, we use a virtual machine instance that has an Nvidia V100 GPU which is connected to Intel Xeon Platinum 8167M CPU via PCIe3. The Nvidia V100 GPU has 16 GB of HBM2 memory. The global memory read/write bandwidth is 880 GBps. The Intel Xeon Platinum CPU has 6 virtual cores and 180 GB DRAM. The bidirectional PCIe transfer bandwidth is 12.8 GBps. The system is running on Ubuntu 18.04 and the GPU instance uses CUDA 11.2.
Measurement: In our evaluation except for Section 3.8.5, we ensure that data is already loaded into the GPU memory before experiments start. We run each experiment 3 times and report the average measured execution time.

3.8.2 Performance with Varying Bitwidths

In this section, we test the performance of our proposed compression algorithms (`GPU-FOR`, `GPU-DFOR`, and `GPU-RFOR`) against existing encoding schemes on workloads with varying bitwidths. We will compare the performance against the following schemes:

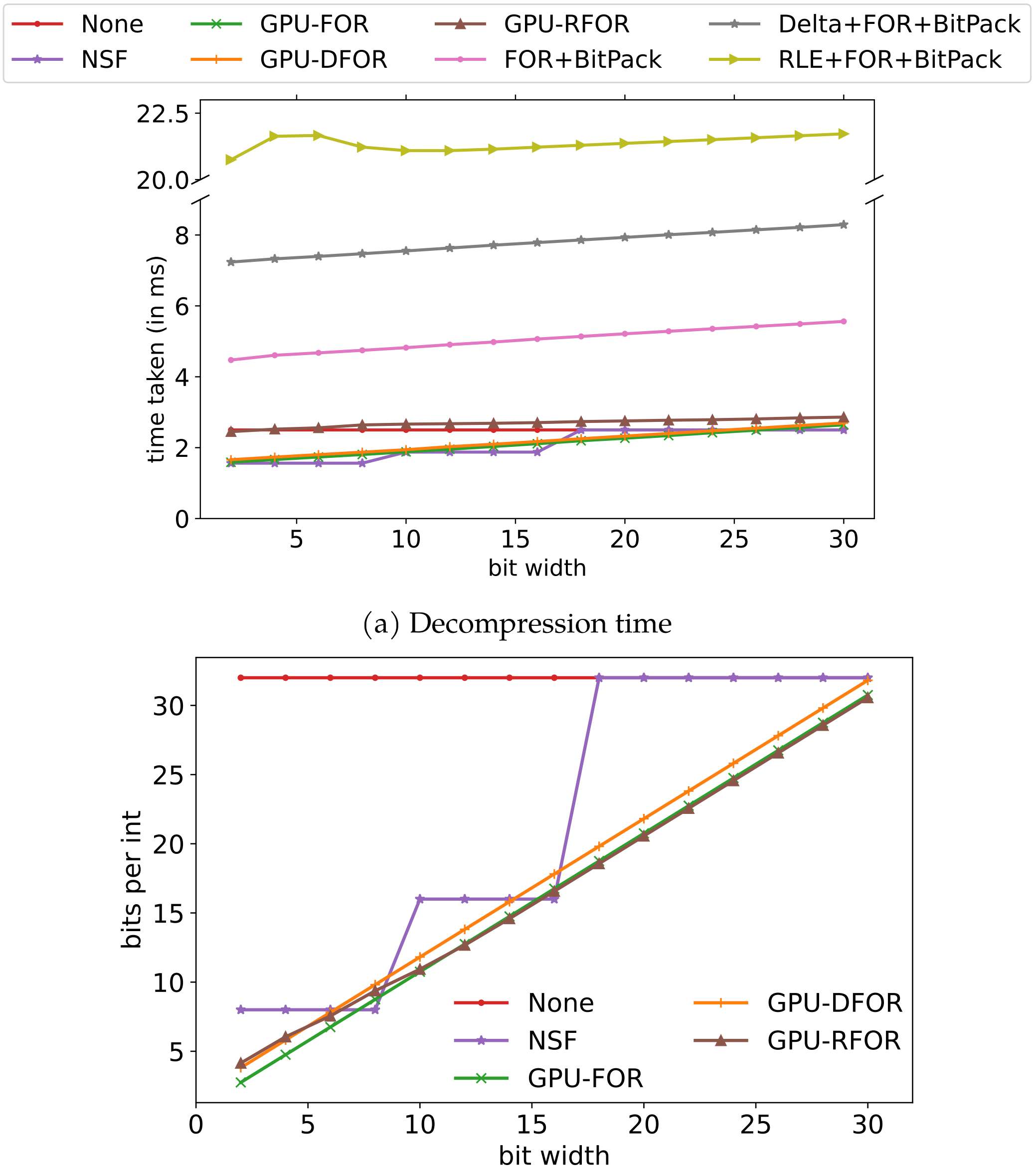
- `None`: Data is stored as 4-byte integers uncompressed.

- `NSF`: Null suppression with fixed length encoding. The entire array is encoded as 1, 2, or 4 byte entries depending on the maximum number of bits needed for any integer in the column.

- `FOR+BitPack`: Using `GPU-FOR` encoding scheme but adopting *cascading decompression* routine. Hence, it will decode the schemes in two kernel passes — the first pass will decode bit-packing and the second pass will decode frame of reference.

- `Delta+FOR+BitPack`: Using `GPU-DFOR` encoding scheme but adopting *cascading decompression* routine. Hence, it will decode the schemes in three kernel passes — the first two passes to decode `FOR+BitPack` followed by a prefix sum to decode Delta.

- `RLE+FOR+BitPack`: Using `GPU-RFOR` encoding scheme but adopting *cascading decompression* routine. It decodes the schemes in eight kernel passes — the first



(a) Decompression time

(b) Compression rate

Figure 3.7: Performance with Varying Bitwidths

four passes to decode `FOR+BitPack` for both the run length and values columns and the last four passes to decode RLE following the steps described in [68].

For this comparison, we use a synthetic dataset with varying bitwidth. We generate 15 unsorted datasets each with 250 million entries, such that all data elements in the i-th dataset have exactly i effective bits, i.e., the value range is $[0, 2^i)$ for $i = 2,4,..,30$. Within each range, the values are uniformly distributed.

Figure 3.7a shows the decompression time of these eight schemes for varying bitwidth. In this figure, we are (1) reading the compressed data from the global memory, (2) decoding it, and (3) writing back the uncompressed data to the global memory. Note that (3) is constant across bit widths while (1) will incur higher global memory operation (read) as the bitwidth increases which would translate directly to slightly increasing decompression time for all schemes. In all measurements, GPU-FOR, GPU-DFOR, and GPU-RFOR perform significantly better compared to their correspondence without *tile-based decompression*. GPU-FOR achieves 2.6× better performance compared to FOR+BitPack. GPU-DFOR achieves 4× better performance compared to Delta+FOR+BitPack. GPU-RFOR achieves 8× better performance compared to RLE+FOR+BitPack. This shows the benefit of applying *tile-based decompression* on our schemes.

The performance of NSF is a staircase pattern where the runtime is based on whether the entry size is 1, 2, or 4 byte. Both None and NSF saturate memory bandwidth. The performance of GPU-FOR is slightly worse than NSF with the worst case gap of 15% achieved at bitwidth 7. The gap is due to slightly larger data size and irregular access pattern associated with accessing the block_starts array used to find the block offsets in the data array. The decompression performance of GPU-DFOR is comparable but slightly worse to GPU-FOR as it has a slightly larger storage footprint. The decompression performance of GPU-RFOR is worse than GPU-FOR and GPU-DFOR. Apart from the fact that the random dataset is not suitable for this compression scheme, decoding GPU-RFOR also requires twice as much resource in terms of GPU registers and shared memory capacity and operations compared to GPU-DFOR.

Figure 3.7b shows the compression rate of five compression algorithms. The bit-packing schemes achieve the finest possible granularity and thus can perfectly adapt to any bitwidth. Consequently, the compression rate is a linear function of the bitwidth. The overhead for GPU-FOR is 0.75 bit per int (1 block start word + 1 reference word + 1 bitwidth word per block of 128 integer entries). The overhead for GPU-DFOR is 0.81 bit per int (0.75 + 1 first value word per $D = 4$ blocks). As the data is not sorted, the deltas vary in the range $[0, 2^i)$ and require one additional bit; our experiments in Section 3.8.3 show the benefit of GPU-DFOR on sorted data. Since the dataset generates random values, GPU-RFOR is not suitable for this dataset. Nevertheless, the compression rate is still a linear function of the bitwidth since the bit-packing scheme is applied on top of the RLE columns. The overhead for GPU-RFOR

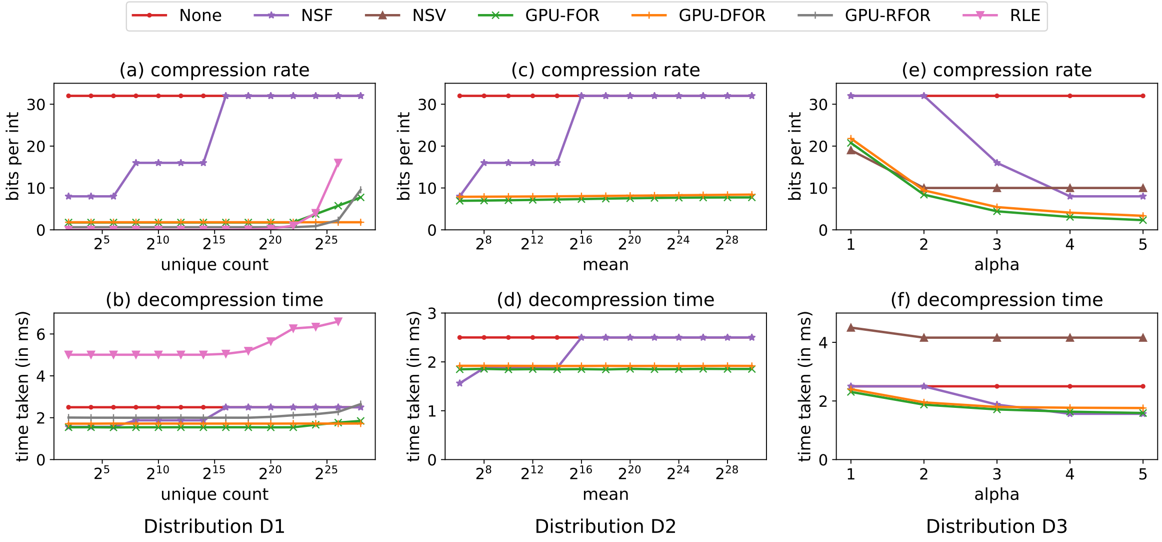


Figure 3.8: Comparison of compression schemes on different data distributions

is slightly less than GPU-FOR since this scheme uses 512 integer entries per block, resulting in less compression metadata (the block start word and the reference word is stored every 512 entries).

Overall, this experiment shows that (1) *tile-based decompression* can significantly improve the performance compared to *cascaded decompression model*, and (2) our bit-packing schemes manage to achieve very good compression rates with almost no overhead on decompression speed.

3.8.3 Performance Across Data Distributions

In this section, we test the robustness of various compression schemes by evaluating their performance using various data distributions. In addition to the schemes we compare against in the previous section, we also compare to the performance of two more compression schemes:

- RLE: Represents runs of the same value as a pair: (value, run-length). Values and run lengths are stored in two separate columns.

- NSV: Represents each value with a variable number of bytes (1,2,3 or 4). In a separate array it maintains the number of bytes used using 2 bits per value. This scheme is good for handling skew.

In this experiment, we generate synthetic workloads with the size of 250 million entries with the following three different distributions.

- D1: a sorted array where we vary the number of unique values from 4 to 2^{28}. Typically a table is sorted based on one column, which D1 is designed to resemble.

- D2: a normal distribution with a standard deviation of 20 and mean varying from 64 to 2^{30}.

- D3: a Zipfian distribution with the exponent *alpha* characterizing the distribution varying from 1 to 5 (1 is least skewed, 5 is most skewed). D3 resembles dictionary encodings of tweets or text corpora where distribution of words follows Zipf's law. For this distribution, we also compare against NSV.

We will not evaluate RLE/GPU-RFOR on D2, D3 as they are not suitable for these data distributions.

The results for D1 can be found in Fig. 3.8 (a-b). The bit-aligned algorithms GPU-FOR, GPU-DFOR, and GPU-RFOR achieve better compression ratios compared to None and NSF due to use of FOR. As the number of unique values increases beyond 2^{22}, the block of 128 integers is likely to have different values. As the dataset is sorted, GPU-DFOR can encode such cases with fewer bits compared to GPU-FOR and GPU-RFOR. In the extreme case, when the number of unique values equals 2^{28} i.e., each value is unique and the array is sorted, GPU-DFOR encodes the data with just 1.8 bits per int vs 7.8 bits per int used by GPU-FOR and 8 bits per int used by GPU-RFOR. The performance of GPU-DFOR (Fig. 3.8 (b)) is bounded by shared memory bandwidth which results in a performance gap in comparison to GPU-FOR.

GPU-RFOR and RLE achieve a very good compression ratio when the number of distinct values is less than 2^{22}. Beyond that, the compression ratio of these two schemes worsened with GPU-RFOR still better than RLE due to the use of FOR. GPU-RFOR is also 2.5× faster than RLE due to the use of tile-based decompression (Fig. 3.8(b)). Decompressing RLE is a 4-step process as explained in Section 3.5, which even after optimizations is similar to GPU-FOR making 4 passes over an array of size n. Across the entire range, we can see that GPU-RFOR is a better choice when the number of distinct values is low while GPU-DFOR is a better choice when the number of distinct values is high in a sorted dataset.

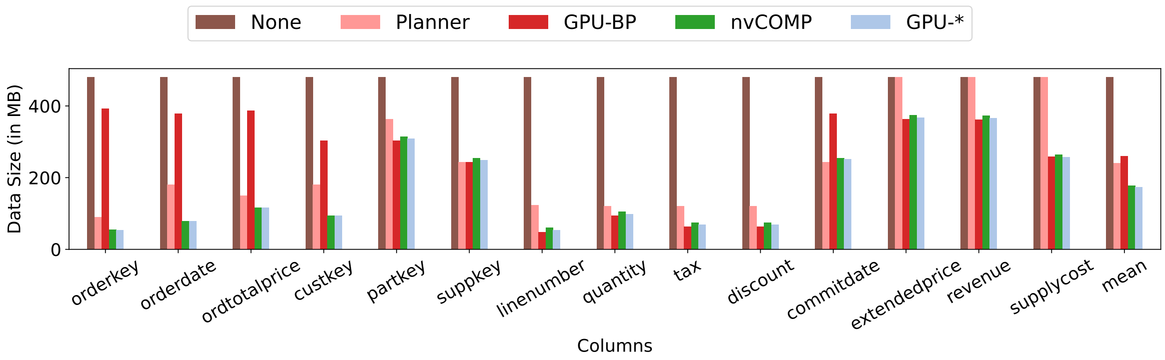


Figure 3.9: Compression Waterfall for Star Schema Benchmark columns

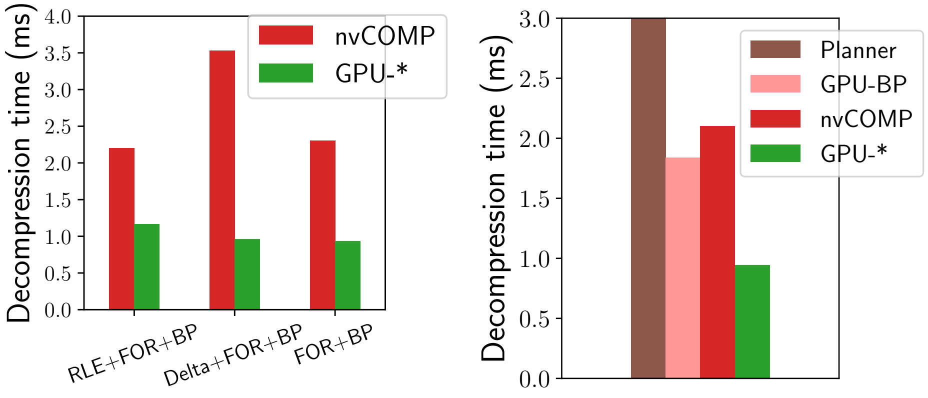
For D2 (Fig. 3.8 (c-d)), we can make the same general observations. When using GPU-FOR/GPU-DFOR, each block's entries generally lie within 3 standard deviations of the mean and occasional occurrence of a value outside this range does not move the compression rate significantly. For mean greater than 2^{16}, the bit-aligned schemes achieve 3× reduction in storage footprint compared to the other schemes and showcases the use of FOR.

For D3 (Fig. 3.8 (e-f)), we see that the bit-aligned schemes can adapt to change in skew and achieve both better compression rate and lower runtime compared to NSF and NSV. NSV is better at adapting to skew compared to NSF, however its performance is significantly worse compared to all the other schemes. Decoding NSF suffers from the same issues that affect RLE, it requires multiple steps that lead to multiple reads and writes, the decoding can't be inline with query execution and it requires buffer space for intermediates.

Overall, this experiment shows that compared to existing schemes, our bit-packing schemes are robust under various data distributions.

3.8.4 Performance on SSB

For the full query evaluation, we use the Star Schema Benchmark (SSB) [97] which has been widely used in various data analytics research studies [70, 90, 119, 133]. SSB is a simplified version of the more popular TPC-H benchmark. It has one fact table *lineorder* and four dimension tables *date, supplier, customer,* and *part* which are organized as a star schema. There are a total of 13 queries in the benchmark, divided into 4 query flights. Query flight 1 (q1.1, q1.2, and q1.3) are selection queries whereas query flights 2, 3, and 4 are join queries. In our experiments we run the benchmark



(a) `GPU-*` vs `nvCOMP`

(b) `GPU-*` vs Existing systems

Figure 3.10: Average Decompression Performance across SSB columns

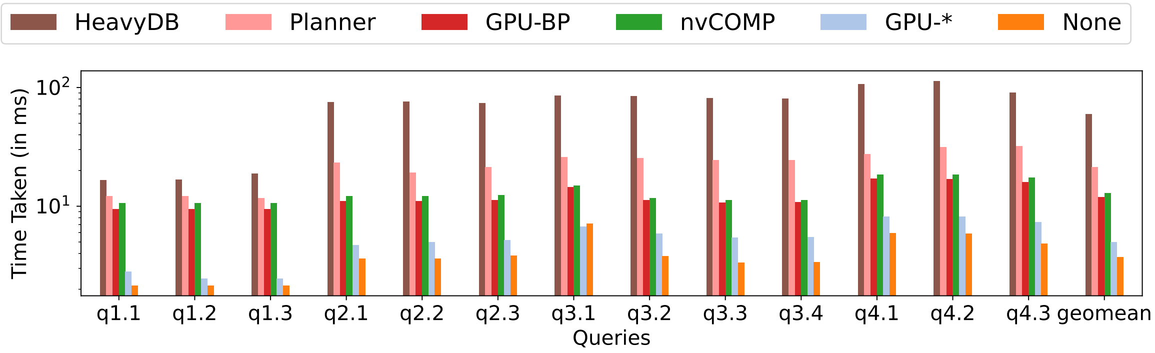


Figure 3.11: Performance on Star Schema Benchmark Queries

with a scale factor of 20 which will generate a fact table with 120 million tuples. To enable efficient query execution in GPU, we dictionary encode the string columns into integers prior to data loading and the queries run directly on dictionary-encoded values. The total dataset size is around 13GB.

In this experiment, we compare the performance of our compression schemes with four existing systems:

- **`Planner`:** We use the compression planner from [68] to generate compression plans for all the column. Note that this work does not support bit-packing-based schemes. For decompression, this work uses *cascading decompression* as explained in Section 3.2.

- **`GPU-BP`:** Mallia et al. [93] introduced two bit-packing schemes: `GPU-BP` and `GPU-VByte`. We compare against `GPU-BP` in this work since it has superior compression ratio and decompression performance. This scheme, however, only consists of a single compression layer without the use of FOR, Delta, or RLE. It also lacks optimization techniques which we introduced in Section 3.3.2.

- **nvCOMP:** We compare against the cascaded compression scheme in nvCOMP. Despite having various schemes such as bit-packing, delta, and RLE, nvCOMP does not support end-to-end pipelining of multiple decompression steps with query execution.
- **HeavyDB:** A commercial GPU database system. HeavyDB only uses dictionary encoding to encode the string column and therefore has the same compression scheme as None.

Figure 3.9 shows the column sizes after the compression for each system. GPU-* is the hybrid scheme that chooses between GPU-FOR, GPU-DFOR, and GPU-RFOR as explained in Section 3.7. Each column in SSB has different properties. For example, lo_orderkey is a sorted column with high average run length (similar to D1 in Section 3.8.3); lo_orderdate, lo_ordtotalprice, and lo_custkey are unsorted but also has high average run length. On average, GPU-* manages to reduce the total memory footprint by 2.8× compared to no compression (None).

Comparing to GPU-BP, GPU-* achieves 50% better compression rates. This is because GPU-BP only supports a single bit-packing layer without frame-of-reference. Therefore, GPU-BP has poor performance for columns with RLE pattern such as lo_orderkey, lo_orderdate, lo_ordtotalprice, and lo_custkey. GPU-BP also performs poorly on date columns such as lo_commitdate due to absence of frame-of-reference in GPU-BP.

Comparing to Planner, GPU-* achieves 40% better compression rates and outperforms Planner across all columns. Planner has poor performance on columns with large random integers such as lo_extendedprice, lo_revenue, and lo_supplycost. These columns can only be compressed using bit-packing schemes which are not supported by Planner. For columns with certain distribution (lo_orderkey, lo_orderdate, lo_ordtotalprice, and lo_custkey), Planner performs well as it is distribution-aware. GPU-*, however, still outperforms Planner on every column.

nvCOMP and GPU-* achieves similar compression rates across all columns while GPU-* is slightly superior by 2%. This is due to the fact that both of these schemes support the same set of compression schemes. It utilizes RLE based scheme for columns with consecutive runs of values and FOR based scheme for columns with large random integers. The 2% gain for GPU-* comes from our more compact data format which requires less metadata and therefore could store the compressed data more efficiently. Despite supporting the same compression schemes, we will show in the next experiment how GPU-* is superior in decompression performance and query running time compared to nvCOMP.

Figure 3.10a shows a detailed one-on-one decompression time comparison between different schemes in nvCOMP and GPU-*. It can be seen from this figure that despite supporting the same set of compression schemes, GPU-* outperform nvCOMP in each scheme. GPU-FOR outperform nvCOMP(FOR+BitPacking) by 2.4×, GPU-DFOR outperform nvCOMP (Delta+FOR+BitPacking) by 3.5×, and GPU-RFOR outperform nvCOMP(RLE +FOR+BitPacking) by 2×.

Figure 3.10b shows the geomean of decompression performance across all columns for each system. For nvCOMP and GPU-*, we choose the scheme which results in the best performance thereby resulting in a smaller performance gap compared to Figure 3.10a. Overall, GPU-* outperforms Planner, GPU-BP, and nvCOMP by 5.5×, 2×, and 2.2× respectively. The performance gap is due to the use of tile-based decompression and our optimization techniques presented in Section 3.3.2.

Figure 3.11 shows the end-to-end query performance across all systems. For the runtime comparison, we compare the performance of Crystal without any encoding (None) against Crystal with the decompression routines of GPU-* and nvCOMP. We also compare our system with Planner, GPU-BP, and HeavyDB.

Comparing GPU-* with None, None is 1.35× faster. This is to be expected since there are overhead of decompressing the columns, especially the lo_orderdate and lo_custkey columns which utilizes GPU-RFOR. These two columns are used in almost every query in SSB. Comparing GPU-* with HeavyDB, GPU-* is 12× faster. This result is consistent with result from prior works [110]. This is because the query execution engine in HeavyDB does not support the tile-based execution model. Comparing GPU-* with other compression schemes (Planner, GPU-BP, and nvCOMP), GPU-* outperform these schemes by 4×, 2.4×, and 2.6× respectively. Not only decompressing individual columns by GPU-* is faster, all these schemes cannot decompress the columns inline with the query execution. Therefore, these schemes will have to decompress each individual column one by one before executing each query, which results in a much higher total query runtime.

Overall, our experiment shows that GPU-* can achieve similar or better compres-

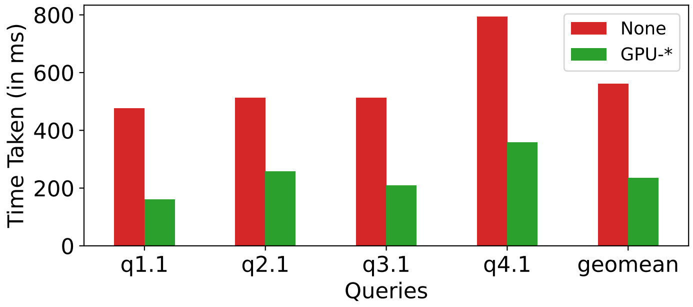


Figure 3.12: Benefit of data compression in GPU as coprocessor model

sion rates to the best state-of-the-art compression schemes in GPU (i.e., `nvCOMP`) while being 2.2× and 2.6× faster in decompression speed and query running time. For GPU-based DBMS systems, using `GPU-*` results in significantly lower storage footprint with minimal performance degradation.

3.8.5 GPU as a Coprocessor

Many systems use GPU strictly as a coprocessor [70, 133, 82, 92]. These systems move data from CPU to GPU across an interconnect like PCIe or NVLink when processing every query. The compression ratio of each compression scheme would play a big role as runtime is bound by the time taken to ship data over the interconnect (transfer time). We have shown that `GPU-FOR`/`GPU-DFOR`/`GPU-RFOR` achieve the best compression rate across a variety of data distributions and using them would reduce the amount of data moving across the slow interconnect thereby reducing transfer time. To evaluate this, we ran one SSB query from each flight (q1.1, q2.1, q3.1, q4.1) with data initially stored on the CPU. The encoded data will then be shipped to GPU over 12GB/s bidirectional PCIe. GPU will then decode the data and execute the query. In this experiment, we will compare the query performance of `GPU-*` against uncompressed data (`None`). Figure 3.12 shows that after applying compression, the query runtime is 2.3× faster. This shows that our compression scheme is useful when the working set is sharded across CPU-GPU or potentially multiple GPUs.

3.9 Conclusion

This work advances the state-of-the-art for GPU data compression by introducing (1) *tile-based decompression* which allows a database to decompress a cascade of compression schemes in a single pass and (2) GPU-FOR, GPU-DFOR, and GPU-RFOR bit-packing based compression schemes and efficient massively parallel bit unpacking routines for them. Our evaluation shows that our schemes can achieve similar compression rates to the best state-of-the-art compression schemes in GPU while being 2.2× and 2.6× faster in decompression speed and query running time.

Chapter 4

Heterogeneous CPU-GPU DBMS

Even with compressed data, GPU memory capacity may still be insufficient for large-scale analytics workloads. Existing solutions address this limitation using two different approaches. The first approach is to *transfer data to GPU on demand* through PCIe when a query accesses data that is not in GPU. This solution is straightforward and has been adopted in both commercial systems [15] and research projects [90, 123, 133]. However, the main downside is the potentially significant data traffic over PCIe, which can become a new performance bottleneck [110]. Although the interconnect bandwidth will increase through new hardware technologies like NVLink [21], it will likely remain much lower than GPU memory bandwidth in the foreseeable future.

To mitigate the limited GPU memory capacity and PCIe bandwidth, a second approach is to *leverage both CPU and GPU for heterogeneous query processing* [50, 62]. Such a design can fully exploit the computational power of both devices and avoid excessive data transfer over PCIe by running certain sub-queries in CPU. In this work, we aim to improve existing heterogeneous GPU and CPU databases by optimizing both data placement policies and heterogeneous query execution. We develop a heterogeneous analytical engine called ***Mordred***, which innovates mainly in the following two aspects:

Data Placement. Similar to prior work [50], Mordred models data placement as a caching problem where a subset of data is cached in GPU memory and the CPU maintains a copy of the entire database. Different from prior work, however, Mordred manages caching at a fine granularity (i.e., sub-column) and uses a novel semantic-aware cache replacement policy. The new policy considers various aspects of the workload such as the query semantics, data correlation, query frequency, etc.

Mordred uses a lightweight cost-based performance model to estimate the benefit of caching, guiding the decision of replacement. For example, Mordred prioritizes the caching of segments that are part of joins over filters. The optimized cache replacement policy can lead to 3× speedup compared to the best prior baseline we compared against.

Heterogeneous Query Execution. Caching data at a fine granularity in GPU presents new challenges during query execution. Ideally, a heterogeneous query executor should exploit data in both devices and coordinate query execution at a fine granularity, which prior systems could not achieve. In Mordred, we support fine-grained heterogeneous execution by introducing *segment-level query plan* — Mordred allows different segments of a column to execute different query plans depending on the segments' location. To further improve performance, Mordred also adopts various optimization techniques (including late materialization, operator pipelining, segment skipping, and lightweight memory allocation) to reduce the traffic through PCIe or device memory.

Contributions. This work makes the following contributions:

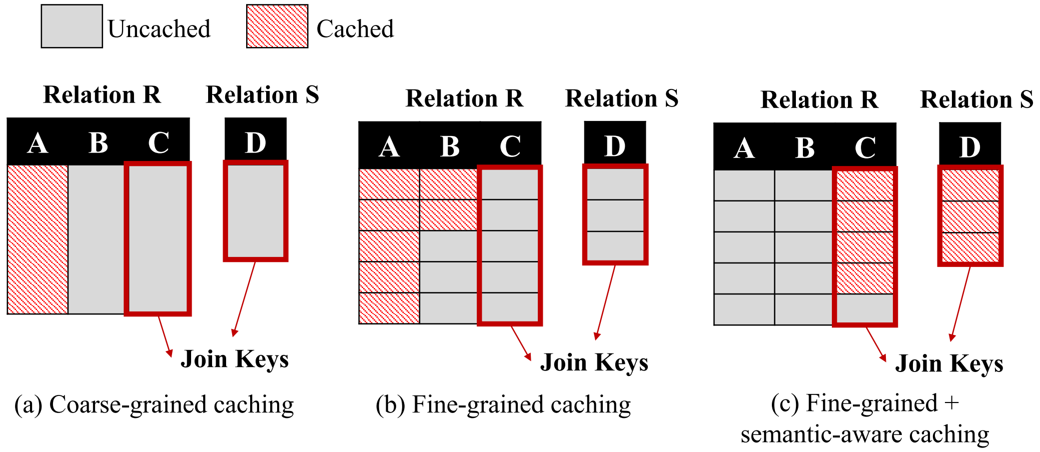
- We develop a semantic-aware fine-grained caching policy for heterogeneous CPU and GPU databases. The policy takes into account query semantics, data correlation, and query frequency to determine data placement.

- We develop a fine-grained heterogeneous CPU-GPU execution engine which converts a query plan into a segment-level query plan to be executed in parallel on both CPU and GPU.

- We build Mordred, a heterogeneous CPU-GPU analytics engine based on fine-grained semantic-aware caching and heterogeneous query execution; Mordred includes various performance optimizations. The source code will be made publicly available.

- We conduct a detailed evaluation and demonstrate that *semantic-aware caching* can outperform the best traditional caching policy by 3×. Mordred can outperform the best-prior GPU DBMSs by an order of magnitude.

Organization. The rest of the chapter is organized as follows: Section 4.1 describes the fine-grained semantic-aware caching policy. Section 4.2 presents the heterogeneous CPU-GPU query execution engine and its optimizations. Section 4.3 describes



(a) Coarse-grained caching (b) Fine-grained caching (c) Fine-grained + semantic-aware caching

Figure 4.1: Illustration of Cache-aware Replication Policy.

Mordred's system architecture and some implementation details. Section 4.4 evaluates the performance of Mordred. Section 4.5 discusses related work and Section 4.6 concludes the chapter.

4.1 Data Placement

Mordred treats data placement as a caching problem following previous work [50] — the complete data set resides in CPU memory and a mirrored subset of data is cached in GPU. Compared to the alternative design that partitions data into disjoint sets across CPU and GPU, maintaining a copy of all the data in CPU allows for more flexible query scheduling — the CPU can process a query if the GPU is occupied with other tasks. It also allows the CPU to reconstruct results to reduce PCIe traffic, as will be discussed in Section 4.2.

A key design decision with this model is to decide which data to cache in GPU memory. Previous work [50, 82] explored caching at column granularity using least-recently used (LRU) and least-frequently used (LFU) replacement policies. We observe that such a design cannot optimally capture the benefit of GPU acceleration. In Section 4.1.1, we argue (1) how a sub-column fine-grained policy can improve caching efficiency and (2) how semantic-aware replacement leads to better performance. Then in Section 4.1.2 we explain the proposed fine-grained semantic-aware caching policy in detail.

4.1.1 Motivation

4.1.1.1 A Case for Fine-Grained Caching

Caching data in column granularity has several problems. First, it is prone to fragmentation where a portion of the cache is empty due to the lack of space to fit in another column. This prevents us from making use of the full capacity of GPU memory. Second, column granularity caching cannot capture access skewness within a column. In real workloads, data accesses are often nonuniform, e.g., recent data is more frequently accessed than older data. In such cases, hotter data within the same column should be prioritized in caching, which cannot be captured in column-granularity caching.

Figure 4.1(a) and 4.1(b) illustrate the difference between coarse-grained (i.e., column granularity) and fine-grained (i.e., sub-column granularity) caching. Each column is split into *segments* with the same size and we assume a cache size of seven segments. We use the term *segment* to refer to sub-column for the rest of the dissertation. Column-granularity caching (Figure 4.1(a)) caches a single column of table R and leaves the remaining two cache slots empty since no column can fit in. In contrast, fine-grained caching can utilize all seven cache slots. Fine-grained caching does introduce new complexity in query execution; we will describe Mordred's solution in Section 4.2.

4.1.1.2 A Case for Semantic-Aware Caching

One way to implement fine-grained caching is to use LRU or LFU on segments rather than columns. However, such semantic-agnostic replacement policies may not be able to accurately identify data that benefits the most from GPU acceleration. For example, join operators are computationally complex and comprise significant execution time of queries. Therefore, data that participates in joins should be cached with higher priority. Such semantic-awareness is not exploited in conventional LRU and LFU.

Furthermore, semantic-aware caching should consider the correlation between multiple columns when deciding what data to cache. In particular, some operators can execute on GPU only if all the involved columns are cached simultaneously. One example is a join operator, where both join keys should be cached. Other examples include columns that are involved in the same filtering predicate or the same aggregation function.

Figure 4.1(b) and 4.1(c) illustrate the difference between naive fine-grained caching and semantic-aware fine-grained caching. In this example, the DBMS identifies that caching segments in the join columns leads to higher speedup over other segments.

4.1.2 Semantic-Aware Fine-Grained Caching

This section describes the proposed semantic-aware fine-grained caching policy. We extend conventional LFU with *weighted frequency counters*, where the weight reflects the potential benefits of caching the segment and is derived using a cost model. The cost model captures (1) the relative speedup of caching a segment and (2) the correlation among segments from different columns. Section 4.1.2.1 describes the general caching framework and Section 4.1.2.2 describes the cost model for the weighted frequency counters.

4.1.2.1 Cache Replacement Policy

The cache replacement policy is based on conventional LFU where each data segment is associated with a frequency counter, which is incremented when the segment is accessed by a query. The cache stores segments with the largest frequency counters. Different from the baseline LFU that increments the counter always by 1 for each access, the replacement policy in Mordred increments it with a weight that is calculated based on semantic information.

Algorithm 2 demonstrates the semantic-aware weight update algorithm in Mordred. We call *UpdateWeightedFreqCounter*() after accessing each segment S to update its weight. The algorithm first estimates the query runtime $RT_{uncached}$ if segment S is not cached in GPU (line 2). The runtime estimation is based on the current content in GPU (i.e., *cached_segments*) with S removed. The function *estimateQueryRuntime*() uses a simple model to predict runtime with the assumption that the CPU/GPU memory and PCIe bandwidth are the performance bottleneck; we will describe details of the cost model in Section 4.1.2.2. Then, the algorithm estimates the query runtime RT_{cached} if S and *all segments correlated with S* are cached in GPU (line 3), besides the currently cached segments.

We use the difference between $RT_{uncached}$ and RT_{cached} to represent the *weight* (line 4). The weight is added to the weighted frequency counter of S (line 5). Furthermore,

Algorithm 2: Update the *weighted frequency counter* for segment *S*

```
1  UpdateWeightedFreqCounter(segment $S$)
2      # estimate query runtime when $S$ is not cached.
3      $RT_{uncached}$ = estimateQueryRuntime($cached\_segments \setminus S$)
4      # estimate query runtime when $S$ and segments correlated with $S$ are cached.
5      $RT_{cached}$ = estimateQueryRuntime($cached\_segments \cup S \cup correlated\_segments$)
6      $weight = RT_{uncached} - RT_{cached}$
7      $S.weighted\_freq\_counter$ += $weight$
8      for $C$ in $correlated\_segments$ do
9          # evenly distribute weight to all segments correlated with $S$
10         $C.weighted\_freq\_counter$ += $weight$ / $|correlated\_segments|$
```

it is also evenly distributed to all the correlated segments of S (line 6–7) through dividing the weight by $|correlated\_segments|$. This means more correlated segments leads to smaller weight assigned to each, which bounds the total weight incremented by accessing a segment.

The precise definition of *correlated segments* for segment S depends on the operator being performed. So far, Mordred considers three operators: selection, join, and group-by aggregation.

For selection, we consider segments correlated if they are (1) involved in a predicate where the attributes cannot be separated by logic "and" or "or" and (2) correspond to the same set of rows. In such a case, all the segments in this correlation set will be incremented by the same weight.

For hash join, we make a key observation that GPU acceleration is effective only if at least one column (i.e., the build column) is completely cached. Therefore, we consider the build and probe relations differently. Specifically, we perform Algorithm 2 on every segment S in the probe column (i.e., typically the larger one) and consider all segments in the build column as correlated.

For group-by aggregation, a correlation exists between the aggregation column and the grouping key column. We consider two cases. First, if aggregation and grouping columns are all in the same table, we perform Algorithm 2 on every segment S in the aggregation column and consider segments in the same set of rows as correlated. Second, if aggregation and grouping columns are in different tables (i.e., they are joined together), we perform Algorithm 2 for every segment S in the aggregation column and consider all segments in the grouping column as correlated.

4.1.2.2 Cost Models

We now explain how the *estimateQueryRuntime*() function in Algorithm 2 works. In particular, we use the cost model presented in Crystal [110] to estimate the execution time of subqueries in both CPU and GPU. The cost model assumes that the queries can saturate the memory bandwidth and derives the execution time from memory traffic. The accuracy of the model has been verified in Crystal on simple operators. We extend the model to support more complex queries and to support PCIe. While the model may not be as precise in this more complex environment, we find the accuracy to be acceptable for the purposes of cache replacement policy.

In particular, we model filtering cost as follows:

$$\textit{filter runtime} = \frac{\mathsf{size(int)} \times N}{B_r} + \frac{\mathsf{size(int)} \times N \times \sigma}{B_w}$$

where N is the cardinality of input segments, σ is the predicate selectivity, B_r and B_w are read and write memory bandwidth, respectively. The first term of the equation is the time taken to scan the input column and the second term is the time taken to write the matched entries to the output array. This and the following equations assume relations of integers.

For hash join, the probe runtime is modeled as follows:

$$\textit{probe runtime} = \frac{\mathsf{size(int)} \times N}{B_r} + (1-\pi)\frac{N \times C}{B_r} + \frac{\mathsf{size(int)} \times N \times \sigma}{B_w}$$

where the extra parameter C is the cache line size, π is the probability the accessed cache line is in the last level cache. The first term is the time taken to scan the probe relation from memory, the second term is the time taken for probing the hash table, and the third term is the time taken to write the matched entries to the output array. Other database operations such as hash aggregation and sorting can be modeled in a similar fashion.

In heterogeneous query execution, there is an extra cost for materialization, merging, and data transfer between CPU and GPU. We model the cost of data transfer as follows:

$$\textit{data transfer time} = \frac{\mathsf{size(int)} \times N}{B_{pci}}$$

where B_{pci} is the interconnect bandwidth. This equation represents the time taken to transfer *N* integers over PCIe.

Materialization is the process of reconstructing tuples from intermediate results expressed as row IDs (more details in Section 4.2.2.1). We model the materialization cost as follows:

$$\textit{materialization time} = \frac{\mathsf{size}(\mathsf{int}) \times N}{B_r} + \frac{N \times C}{B_r}$$

where the first term is the time taken to scan the row IDs from the intermediate results and the second term is the time taken to reconstruct the tuples from the row IDs.

Finally, the following equation models the cost of merging the final results from CPU and GPU:

$$\textit{merging time} = \frac{2 \times \mathsf{size}(\mathsf{int}) \times N}{B_r} + \frac{\mathsf{size}(\mathsf{int}) \times N}{B_w}$$

where the first term is the time taken to scan the two columns to be merged and the second term is the time taken to write the results.

These equations are used as building blocks to estimate the query runtime in the *estimateQueryRuntime*() function. The cost model is lightweight such that its evaluation incurs negligible overhead.

4.1.2.3 Example of Semantic-Aware Caching

```
Qx: SELECT R.B, SUM(R.A) FROM R,S
    WHERE R.C = S.D

Qy: SELECT R.A AND R.B FROM R
```

We present an example to illustrate how semantic-aware caching works. Consider the schema in Figure 4.1 and performing replacement after executing *Qx* and *Qy*. Assume a cache size of seven segments.

Coarse-grained LFU will cache either column R.A or R.B since both columns are used by both queries and the cache has space for only one column, as illustrated in Figure 4.1(a). In contrast, fine-grain LFU (Figure 4.1(b)) can fill in the two empty cache slots with two segments from R.B. Different from both LFU policies, the semantic-aware policy caches segments that maximize performance using the cost model in Section 4.1.2.2. Therefore, even though R.C and S.D are used only by *Qx*, our policy assigns higher weight for segments in both columns since the GPU offers higher speedup for join over filtering operations. The caching decision is shown in Figure 4.1(c).

4.2 Query Execution

A new challenge introduced by the fine-grained caching policy is the extra complexity of query execution. It is possible that only a subset of data required by an operator exists in GPU memory so that the entire operator cannot directly run on GPU. Ideally, a heterogeneous query executor should fully exploit the data in GPU and coordinate query execution across the two devices at segment granularity instead of column granularity. While some existing systems [15] have adopted fine-grain caching, they still execute the entire query in GPU and transfer the uncached data to GPU during query execution, leaving the available CPU cores unutilized.

In Mordred, we exploit both intra-device parallelism within CPU/GPU and inter-device parallelism across CPU/GPU. Our heterogeneous query executor targets the following three goals:

Goal 1: Minimize inter-device data transfer. Currently, the interconnect bandwidth (i.e., PCIe) between CPU and GPU is very limited. Transferring too much data through this interconnect could throttle system performance.

Goal 2: Minimize CPU/GPU memory traffic. Data analytics applications are memory-bound. Reducing the data traffic to both CPU and GPU memory leads to more efficient query execution.

Goal 3: Fully exploit parallelism in both CPU and GPU. Ideally, we want to utilize all the available computational power in both multicore CPU and GPU during query execution.

4.2.1 Segment-Level Query Execution

A line of previous research [57, 51, 58, 59, 52, 87] has studied the problem of heterogeneous query execution in column granularity. In this section, we discuss how we address heterogeneous query execution in segment granularity in hybrid CPU and GPU systems.

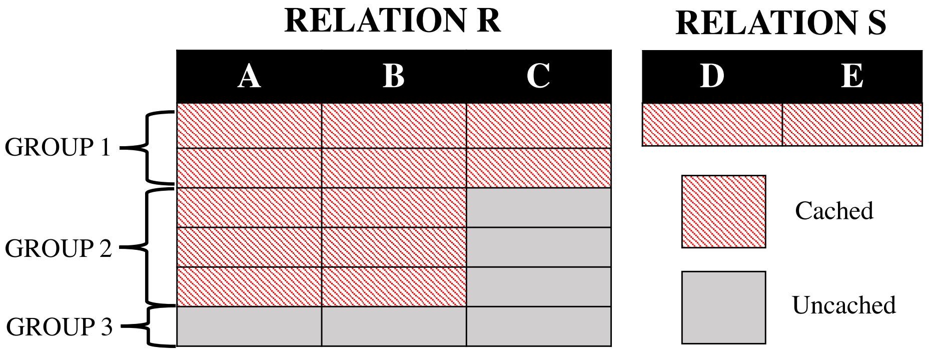


Figure 4.2: Example of Segment Grouping.

4.2.1.1 Operator Placement

Previous works have discussed operator placement strategies for CPU-GPU systems. In particular, Breß et al. [52] proposed a *data-driven operator placement* heuristic, where an operator is pushed to where the input columns reside. The operator is executed in GPU only if *all* the input columns are cached in GPU, otherwise, the operator is executed in CPU. This heuristic has been shown to outperform cost-based optimizer while being more lightweight.

In Mordred, we adopt the *data-driven operator placement* heuristic but apply it at segment granularity — instead of executing an operator in the device where the entire input *columns* reside, Mordred executes portions of the operator in the device where all the input *segments* reside. This means a single operator can be split to run in both CPU and GPU depending on the location of input segments. We call the resulting plan a *segment-level query plan*.

In Mordred, every operator can be split between CPU and GPU. For filter, each partition can be filtered independently in either CPU or GPU. For hash join, we require the build column to be entirely cached and split the probe operator across the two devices. For group-by, if the corresponding grouping and aggregation segments are cached, we perform group-by on them in GPU and send the results back to CPU.

4.2.1.2 Segment-Level Query Plan

Given a particular data placement determined by the caching policy, Mordred puts

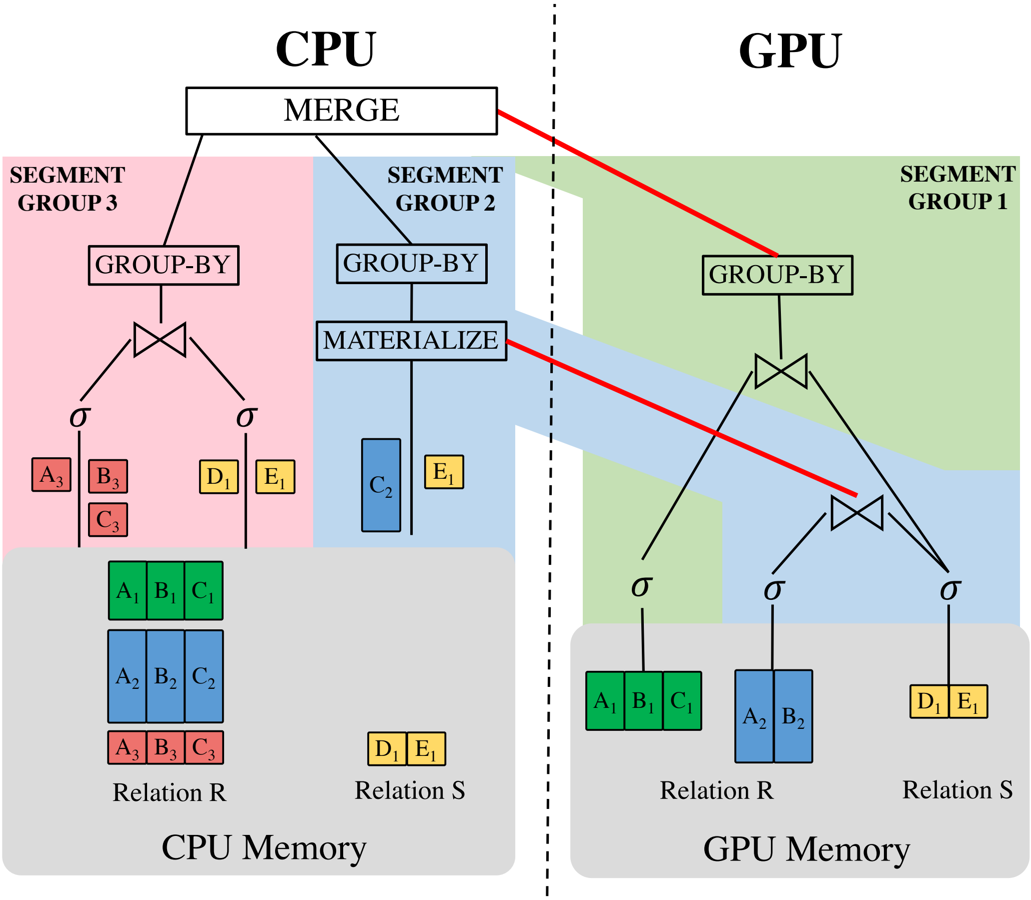


Figure 4.3: Example of Segment-level Query Plan.

input segments into groups and executes them in parallel. In particular, Mordred applies the *data-driven operator placement* heuristic to determine the execution plan for each segment and puts segments with the same plan into the same *segment group*. The query optimizer and executor work in segment-granularity.

Figure 4.2 demonstrates one example of segment grouping where relation R is partially cached and relation S is fully cached in GPU. One obvious grouping strategy is to split relation R into three segment groups — Group 1 comprises the first two rows of segments, Group 2 comprises 3rd to 5th rows of segments, and Group 3 comprises the last row of segment. Note that the grouping strategy may differ depending on the query plan. For example, if a query accesses only columns A and B in relation R, then Groups 1 and 2 above can be merged into a single large group. This is because both groups have identical execution plan for such a query.

After the grouping phase, each segment group is executed in parallel according to its execution plan. Finally, after all segment groups finish execution, all results will be sent back to CPU and be merged. The merge operation is lightweight. It simply combines the aggregation results from different segment groups together. In Section 4.4.3, we show that merging contributes to $<$ 2.5% of total query execution time in our workload. When the query result is very large, however, merging could potentially be a bottleneck. Such a case is partially captured in the cost model (Section 4.1.2.2), which will choose not to cache the data and run the query in CPU. A more general solution to the problem requires a full-fledged heterogeneous query optimizer, which we defer to future work.

4.2.1.3 Example of Query Execution

```
Q0: SELECT S.D, SUM(R.C) FROM R,S
    WHERE R.B = S.D AND R.A > 10 AND S.E > 20
    GROUP BY S.E
```

We present an example to illustrate how segment-level query execution works using the query Q0 shown above with the table schema and cache layout in Figure 4.2. In this example, R is the probe relation and S is the build relation. Since the query accesses all the three columns in relation R, we split R into three segment groups.

Figure 4.3 shows a physical query plan generated by Mordred. Activities in CPU and GPU are shown in the left and right halves of the figure, respectively. The red lines crossing the device boundary indicate the transfer of intermediate results across PCIe. The sub-query plans for individual segment group are shaded with different background colors. Below, we explain the sub-query plan for each segment group individually.

Segment Group 1: All segments in this group are fully cached and thus the entire sub-query is executed in GPU. After the group-by operator, the results are sent back to CPU.

Segment Group 2: Segments in this group are partially cached. Specifically, segments used for filter (A_2, E_1) and join (B_2, D_1) are in GPU but segments used for group-by (C_2) reside in CPU. The join can still be performed in GPU; in fact, the hash table on relation S can be reused for segment group 2 and segment group 1.

To minimize the data transfer across PCIe, the result of the join is expressed using row ID pairs between the two tables, i.e., $(\text{RowID}_R, \text{RowID}_S)$, to indicate the rows that join. Using the row ID pairs, the CPU will *materialize* the join results by reading the values from C_2 and E_1. The materialized results are sent to the group-by operator and the output is merged with other sub-queries' results.

Segment Group 3: In this group, no segment from relation R is cached in GPU. Therefore, join cannot be performed in GPU. The entire sub-query is executed in CPU with the final output merged with the other sub-queries' results.

4.2.2 Other Performance Optimizations

Memory and PCIe traffic is typically the performance bottleneck in data analytics applications [110]. In this section, we describe other optimization techniques in Mordred to further reduce data traffic.

4.2.2.1 Late materialization

With heterogeneous query execution, there are often cases that require the transfer of intermediate results of sub-queries from one device to another. Transferring the whole intermediate relation across PCIe could be expensive. Previous CPU-based columnar databases used the *late materialization* strategy to reduce data transfer [41], where the intermediate relation can be expressed in the form of row IDs. The receiver side can then reconstruct the tuples. We implement this technique in Mordred to significantly reduce the amount of data transfer.

Another benefit of late materialization is that we can execute an operator in GPU with the minimum number of input columns. For example, joining two relations in GPU requires only the two join key columns to be present in GPU. The rest of the attributes can be reconstructed in CPU using late materialization.

4.2.2.2 Operator Pipelining

Previous work [110, 62] has developed an optimization technique for GPU query execution by pipelining operators into a single kernel. Operator pipelining can avoid storing intermediate results in memory after each operator, and only materialize the results at the end of each pipeline. Crystal [110] has shown that pipelining operators in GPU query execution can further improve the performance by storing the intermediate result from each operator in the shared memory, which can reduce the total round trips to the global memory. We apply this optimization to Mordred by always pipelining consecutive operators on the same device whenever possible.

4.2.2.3 Segment Skipping

Many queries in OLAP workload only access a portion of the data after predicate evaluation. For example, business intelligence queries often access only data in a specific period of time. Minmax pruning is a well-known technique to reduce the amount of data being scanned by the query [26]. The technique maintains per-segment minimum and the maximum values in the segment. During query execution, an entire segment can be skipped if the predicate cannot possibly be satisfied based on the min and max values.

Typically, minmax pruning is applied only to predicate evaluation. In Mordred, we extend it to filter during join operators as well in order to further reduce the amount of data that needs to be cached in GPU.

```
Q1: SELECT R.y FROM R
    WHERE  R.datekey = S.datekey AND S.year > 1997
```

We demonstrate this using an example query Q1 above. R is the probe relation and S is the build relation. In the build phase, Mordred maintains the min and max values of all keys (`S.datekey`) in the hash table. After the build phase is finished, we use these min-max values to skip segments in the probe relation. In particular, segments from R can be skipped by looking at the min-max of the hash table and the min-max of the segment in `R.datekey`. This optimization allows us to further reduce memory and PCIe traffic and identify the truly hot segments when performing cache replacement.

4.3 Systems Integration

This section describes the implementation of Mordred, a hybrid CPU-GPU DBMS with *fine-grained semantic-aware caching* and *segment-level query execution*. Figure 4.4

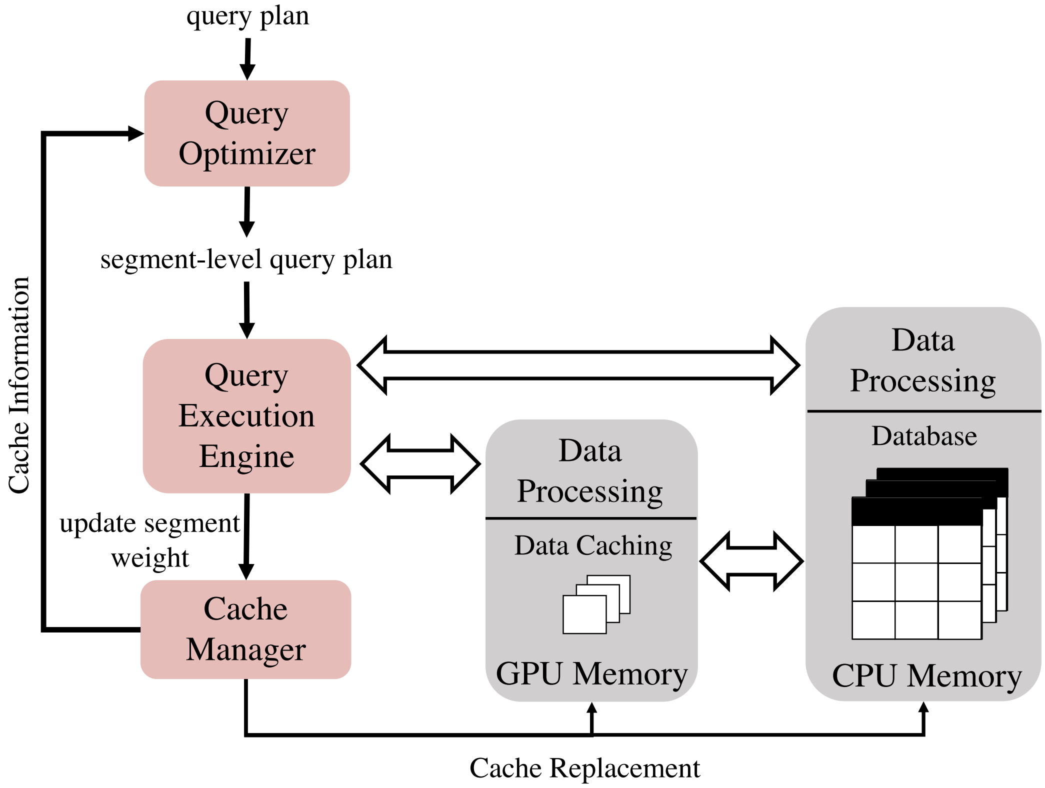


Figure 4.4: Mordred System Overview

shows the overview of Mordred, which consists of three main modules that will be described in the following subsections.

4.3.1 Cache Manager

The Cache Manager performs periodic data replacement in GPU memory based on the caching policy described in Section 4.1. The segment size is user defined and by default 1,048,576 records (2^{20}).

Mordred divides GPU memory into the following two regions.

Data Caching Region: The data caching region handles the caching of raw data in segment granularity. Data in this region is managed by the cache manager, using the *semantic-aware caching policy* described in Section 4.1.2.

Data Processing Region: The data processing region stores intermediate data (e.g., hash table, intermediate query result, etc.) during execution. Since frequent memory allocation in GPU is expensive, we develop a lightweight memory allocation strategy.

Specifically, when the cache manager is initialized, the data processing region is preallocated. The cache manager maintains a pointer to the starting address of the data processing region. Whenever a memory allocation request arrives, the cache manager returns the address of the pointer and advances it by the size of the allocated region. After a query is executed, we reset the pointer to the starting address of the data processing region. The same lightweight memory allocation strategy is also applied in CPU.

The cache manager also handles metadata management. Metadata involves statistics information of each segment (e.g., access timestamp and weighted frequency counters) and the metadata required during query execution (e.g., min-max of each segment, offset of cached segments, etc.). We use bitmap to mark if the segment is cached in GPU. We use a hash table to track the offset and the statistics of each segment (min-max, counter, etc). Finally, we use a free list to track available segments in GPU. The whole metadata management in Mordred is handled by the CPU.

4.3.2 Query Optimizer

The query optimizer module converts a query plan into a *segment-level query plan*. Currently, we take the query plan from Crystal [110], which is already highly optimized for GPU.

Taken the input query plan, Mordred performs segment grouping as described in Section 4.2.1 based on the data-driven operator placement heuristic. Meanwhile, Mordred also reorders the operators based on where they will be executed. For example, operators that will be executed on GPU will be clustered together. We do this to avoid *ping-pong effect* where the execution plan will go back and forth between CPU and GPU, causing excessive data transfer. Segment skipping as described in Section 4.2.2.3 is also partially performed in this stage by the query optimizer.

For simplicity, our optimization is currently heuristic-based and focuses on minimizing interconnect traffic. As interconnect bandwidth improves, a cost-based optimizer that takes into account the interconnect bandwidth might outperform our current design. We leave this to future work.

4.3.3 Query Execution Engine

The query execution engine executes segment-level query plan generated by the query optimizer. Specifically, the engine executes the sub-query plan for each segment group in parallel and merges the final results. Finally, the execution engine notifies the cache manager to update the weight of segments following Algorithm 2.

To launch operators in CPU, Mordred uses IntelTBB parallel programming library. For GPU kernel implementation, we use Crystal [110], a CUDA-based library for query execution. We add new functions to express *late-materialization* in Crystal.

During execution, each segment group is assigned to a dedicated CPU thread which holds information about the execution plan. These CPU threads start the execution of all segment groups in parallel. Each thread will launch either CPU kernels or GPU kernels to execute operators according to its corresponding execution plan. Since each CPU kernel will also be executed using multiple threads, Mordred leaves the thread assignment to the thread scheduler of the parallel programming framework (i.e., Intel TBB).

Currently, we implement a compound kernel for each pipeline to enable operator pipelining (cf. Section 4.2.2.2). We leave automatic code generation of pipelined kernels as part of our future work.

4.4 Evaluation

In this section, we report the performance of Mordred. The section will answer the following key questions:

- How does fine-grain semantic-aware caching perform compared to traditional caching policies?

- How does segment-level query execution improve the performance of heterogeneous query processing?

- How much performance improvement is achieved through various optimizations (i.e., segment skipping, operator pipelining, and late materialization) implemented in Mordred?

- How does Mordred perform compared to existing heterogeneous CPU/GPU DBMSs?

4.4.1 Experimental Setup

Hardware configuration: We use a virtual machine instance in Oracle Cloud with NVIDIA V100 GPU connected to Intel Xeon Platinum 8167M CPU via PCIe3. The Nvidia V100 GPU has 16 GB of HBM2 memory with read/write bandwidth of 880 GBps. The Intel Xeon Platinum CPU has 24 virtual cores and 180 GB DRAM. The bidirectional PCIe bandwidth is 12.8 GBps. The system is running on Ubuntu 18.04 and the GPU instance runs CUDA 11.2.

Benchmark: For our experiments, we use the *Star Schema Benchmark* (SSB) [97] which has been widely used in various data analytics research studies [70, 90, 119, 133]. We use Scale Factor 40 (i.e., 25 GB data set) in all experiments unless otherwise stated. To enable efficient query execution in GPU, we dictionary encode the string columns into integers prior to data loading and manually rewrite the queries to directly reference the dictionary-encoded value. Therefore, we ensure that all column entries are 4-byte in value. We also sort the fact table by the `orderdate` column to enable the *segment skipping* optimization. In our evaluation, the entire data set is loaded to CPU memory before each experiment starts.

Measurement: Before each experiment, we first warm up the GPU memory by running 100 random queries from the SSB benchmark and then perform a replacement to populate the cache. We run 500 queries in each experiment unless otherwise stated and perform replacement after every 50 queries. We found the replacement cost to be negligible ($< 1.5\%$) and thus do not include it in our measurement. For each measurement, we repeat the experiment 3 times and report the average results.

4.4.2 Comparison with Other Caching Policies

This subsection evaluates the performance of *fine-grained semantic-aware caching policy*. Specifically, we will compare the performance of seven different caching policies:

- **LRU (Column):** This policy maintains the access timestamp for each column and caches columns with the latest timestamps. This policy is used by [82, 50].
- **LFU (Column):** This policy maintains a frequency counter for each column and cache columns with the largest counters. This policy is used by [50].
- **LRU-K (Column):** This policy maintains the backward K- distance for each column — the distance backward to the K^{th} most recent reference to the column.

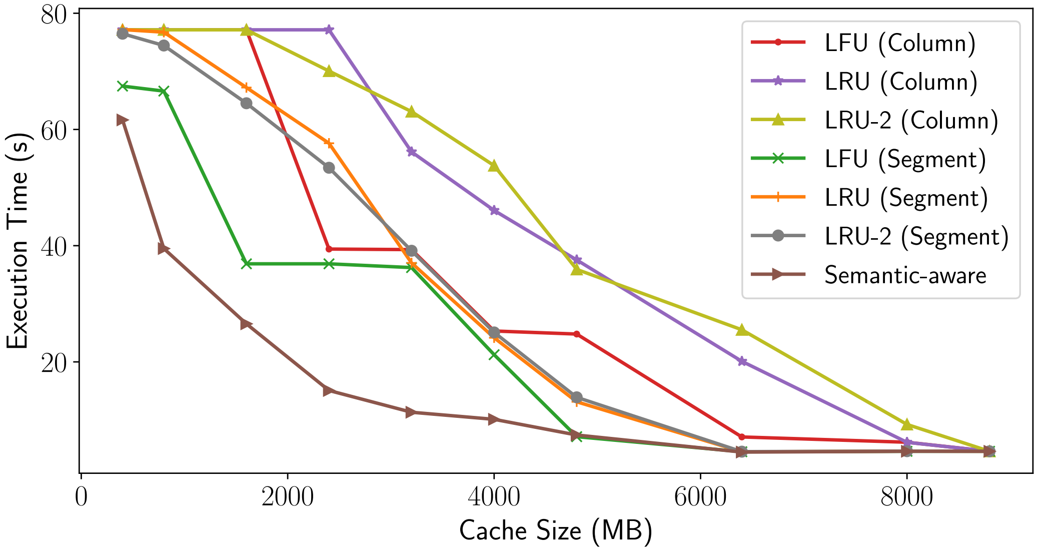


Figure 4.5: Execution Time of Various Caching Policies with Different Cache Size (Uniform distribution with $\theta = 0$)

Columns with the smallest backward K-distance will be cached. We use $K = 2$ in our experiments.

- **LRU (Segment):** Similar to LRU (Column) but timestamps are maintained for segments instead of columns.
- **LFU (Segment):** Similar to LFU (Column) but frequency counters are maintained for segments instead of columns.
- **LRU-K (Segment):** Similar to LRU-K (Column) but the backward K-distance are maintained for segments instead of columns.
- **Semantic-aware:** The segment-granularity semantic-aware caching policy described in Section 4.1.

4.4.2.1 Performance on Standard SSB

In this experiment, we sweep the GPU cache size and measure the performance of different caching policies when running SSB queries. We sweep the cache size from 400 MB to 8.8 GB; all columns that are accessed by queries fit in an 8.8 GB cache. The query access distribution is uniform following the default configuration.

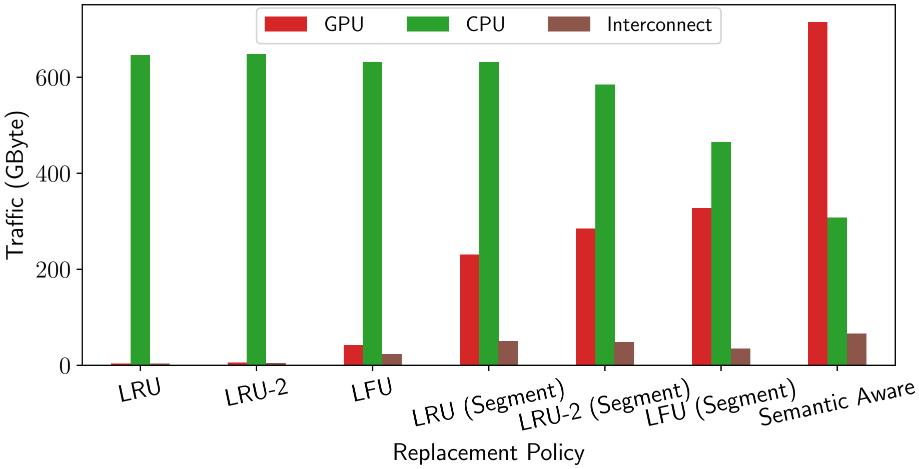


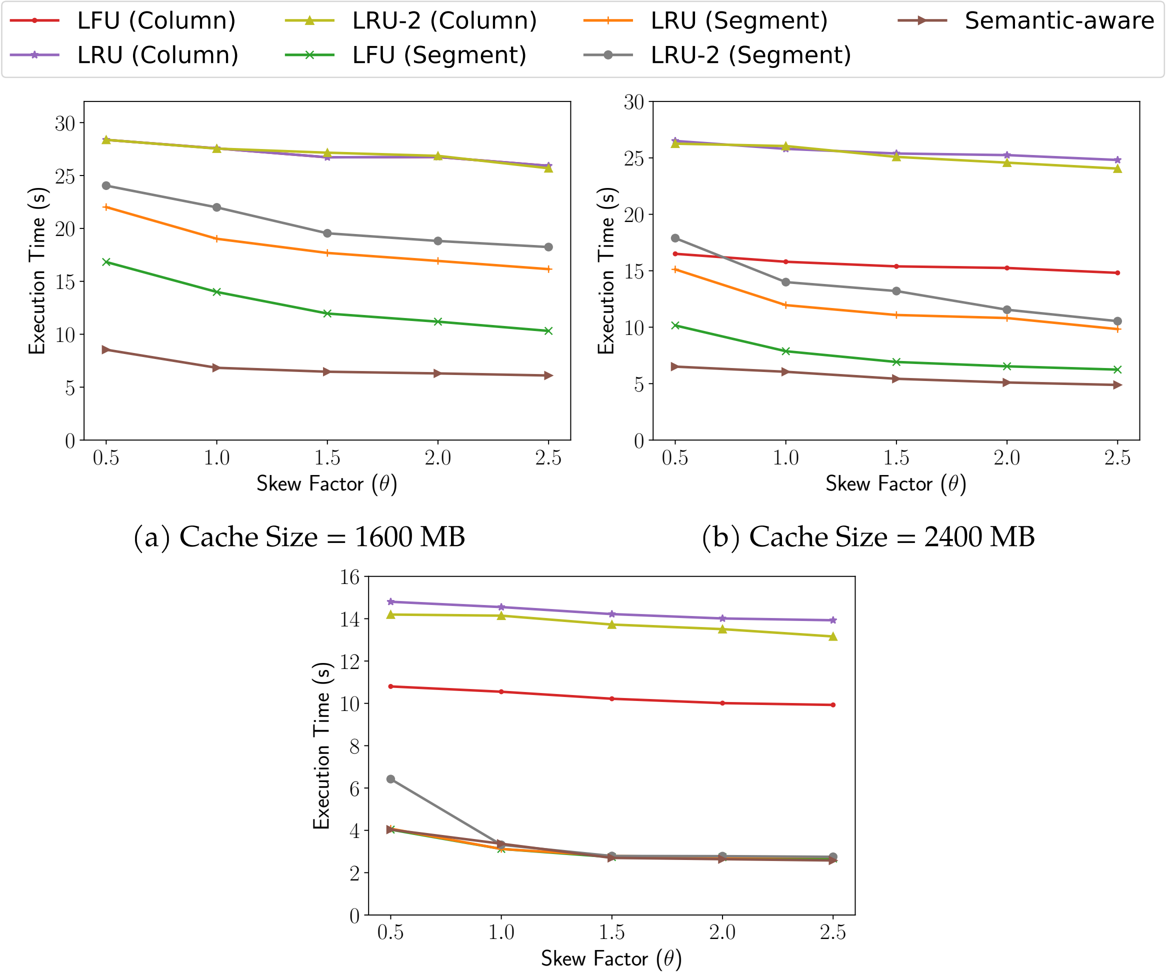
Figure 4.6: Memory Traffic Breakdown for Each Caching Policy

Figure 4.5 shows the result of our experiment. Overall, LFU-based schemes perform better than LRU-based schemes. LRU-2 performs similarly to LRU but slightly better for small cache sizes. For each policy, the fine-grained version always outperforms its coarse-grained counterpart. This is because the coarse-grained policy often suffers from fragmentation, causing a portion of the cache unused.

The semantic-aware caching policy outperforms all the other policies in all cache sizes, especially when the cache size is small (3–7× lower runtime). This is because the new policy can more accurately identify hot segments to cache leading to higher speedup.

Specifically, Figure 4.5 illustrates the limitation of conventional schemes. For example, in LFU (Segment) from the cache size of 1.8 GB to 3.6 GB, there is little performance improvement even though twice as much data has been cached. A closer investigation reveals that the newly cached data comes from the `lo_revenue` column which is used only for group-by. Since the join is performed in CPU, the heuristics decides to execute group-by also on CPU to avoid excessive PCIe traffic, rendering the cache ineffective. This results in performance stall from 1.8 GB to 3.6 GB. The semantic-aware caching policy, in contrast, is aware of the query semantics and thus does not suffer from this performance stall.

4.4.2.2 Memory Traffic Breakdown



(a) Cache Size = 1600 MB

(b) Cache Size = 2400 MB

(c) Cache Size = 4800 MB

Figure 4.7: Execution Time of Various Caching Policies with Varying Query Access Distribution

To gain deeper insight on each caching policy, we compare the data traffic going through the CPU memory, GPU memory, and interconnect (PCIe) for each policy. We use the same setup as Section 4.4.2.1 with a cache size of 1.6 GB (which can hold 20% of all accessed columns).

Figure 4.6 shows the traffic breakdown for different caching policies. The interconnect traffic remains low across all policies; this is partly because of the *data-driven operator placement* heuristic (see Section 4.2.1) where an operator is executed on GPU only if the input data is cached. Column-granularity caching shows a very low GPU memory traffic and high CPU memory traffic. For these policies, only a single column from the fact table fits in the cache resulting in most of the queries completely being executed on the CPU. The fine-grain version of each policy always has higher GPU memory traffic and lower CPU memory traffic.This shows that with segment granularity caching, more work can be offloaded to the GPU.

Across all policies, semantic-aware caching has the highest GPU memory traffic and lowest CPU memory traffic. This is because the policy caches only critical segments which would greatly reduce the total CPU memory traffic. Since GPU has a larger cache line (128B) compared to CPU (64B), the GPU memory traffic can be much higher especially when random reads/writes are involved. However, since GPU memory has the highest bandwidth (10× of CPU memory bandwidth), this still results in better performance. Overall, our traffic breakdown from this experiment is aligned with the query performance results from Section 4.4.2.1.

4.4.2.3 Varying Query Access Pattern

This experiment evaluates the performance of semantic-aware caching policy under varying query access distribution. To simulate nonuniform query access distribution, we incorporate skewness into the predicates of SSB queries. We pick the values following a Zipfian [76] distribution with a tunable skewness that is controlled by a parameter θ. Skewness is applied to the date predicate such that more recent data has a higher probability to be accessed by the query. A larger θ indicates higher skewness.

Figure 4.7 shows the execution time with three different cache sizes when we sweep the skew factor. When the cache size is small (Figure 4.7a), fine-grained caching is more sensitive to skewness than column-granularity caching. This is because for nonuniform query accesses, fine-grain policies can cache only the hot portion of the data. The higher the skew factor, the more accurate these policies can capture the hot portion. We can also see that LFU (Segment) performs better than LRU (Segment) and LRU-2 (Segment) in most cases. This shows that access frequency is better than access timestamp for capturing skewness in query distribution.

Across experiments in Figure 4.7, semantic-aware caching outperforms traditional caching policies. For large cache sizes (Figure 4.7c), the performance of LFU (Segment), LRU (Segment), and LRU-2 (Segment) are very close to semantic-aware caching. This is because the cache size is enough to fit in almost all the hot portion of the data. For smaller cache sizes (see Figure 4.7a), however, the performance gap is

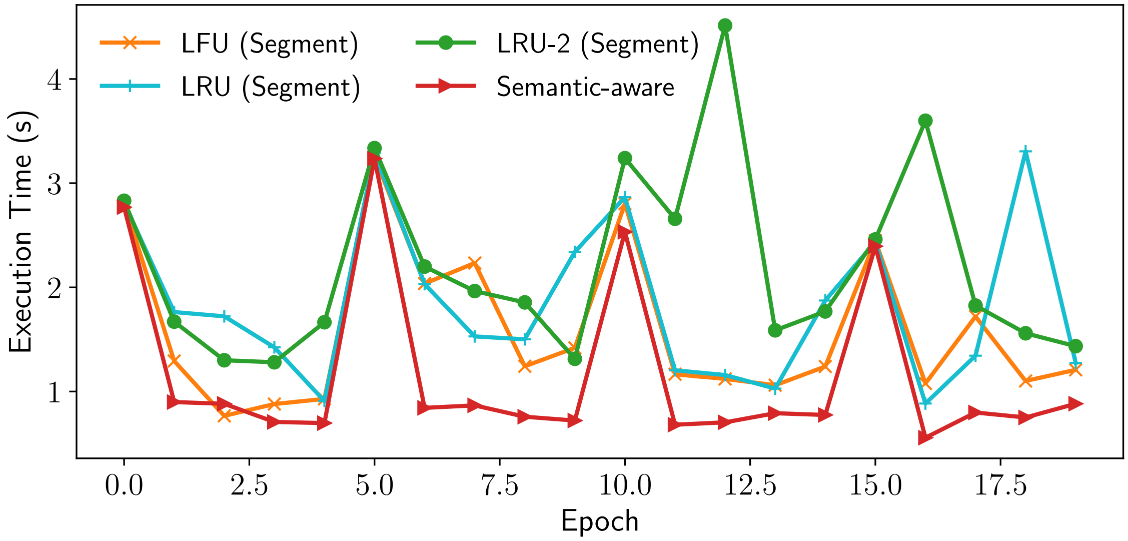


Figure 4.8: Execution Time of Various Caching Policies under Phase Changing Workload

bigger since not all the hot portion can fit in. In this case, semantic-aware caching can more accurately identify critical segments that provide the most benefit from GPU acceleration.

4.4.2.4 Performance on Phase-Changing Workload

This experiment evaluates different caching policies under a phase-changing workload. We incorporate skewness into the date predicate of SSB queries. We use a normal distribution with $\delta = 0.5$ years and a tunable mean controlled by parameter μ. After every 5 batches of queries (50 queries per batch), we shift the mean by 3 years. The results are shown in Figure 4.8.

For frequency-based policies (LFU and semantic-aware), we multiply the current weight by an aging factor (default is 0.5) when a new epoch starts. This would prioritize recent frequency information over history in the past. Our experiment shows that LFU-based policy performs better than LRU and LRU-2. LRU and LRU-2 suffer from random performance spikes since the cache content depends on the last query in the previous batch, which can be random. If the last query accessed data from the cold data region, replacement can significantly degrade performance. Overall, our experiment shows that our policy adapts quicker and outperforms other

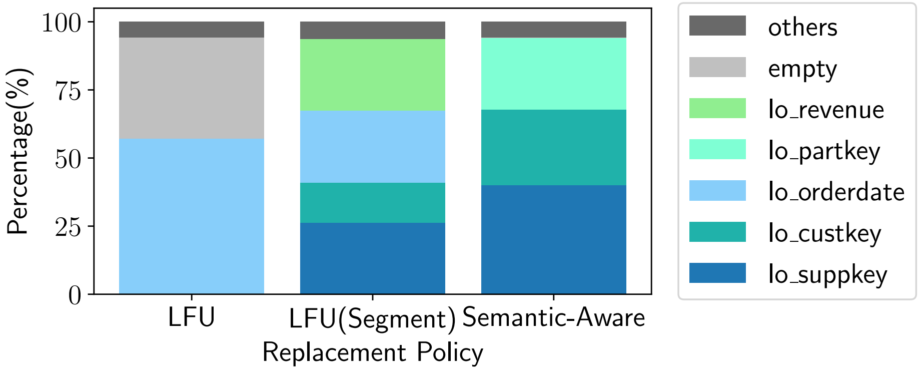


Figure 4.9: Cached Columns in Different Replacement Policies

schemes under a phase-changing workload.

4.4.2.5 Caching Statistics

In this experiment, we show the content of the cache for each caching policy. We use a cache size of 1.6 GB (which can hold 20% of all accessed columns). Figure 4.9 shows the result of our experiment.

LFU (Column) policy can only cache a single column in the fact table (`lo_orderdate`) and leaves 37% of the cache unused. Column `lo_orderdate` is a foreign key to the `d_datekey` column from the `DATE` relation. Caching this column enables us to perform join against the `DATE` relation in GPU. However, the `DATE` table is not as selective as the other dimension tables. This could result in suboptimal performance since transferring the join result to CPU will be expensive.

LFU (Segment) policy caches a more diverse set of columns compared to the LFU (Column). It does not suffer from fragmentation and tends to cache the hot portion of the data. A large chunk of the cache, however, is still for `lo_revenue` which is only used in `GROUP BY` expression. `GROUP BY` is often lightweight and therefore should not be prioritized over caching columns used for `JOIN`.

Our semantic-aware caching prioritizes caching the foreign keys from the fact table (`lo_suppkey`, `lo_custkey`, `lo_partkey`). This often enables us to perform join with the `SUPPLIER`, `CUSTOMER`, and `PART` relations in GPU. Joins involving these relations are very selective. Therefore, caching segments from these columns is very beneficial since the join output transferred from GPU is usually small. This also leaves CPU with a much more lightweight execution over a smaller relation in the later stage of query execution.

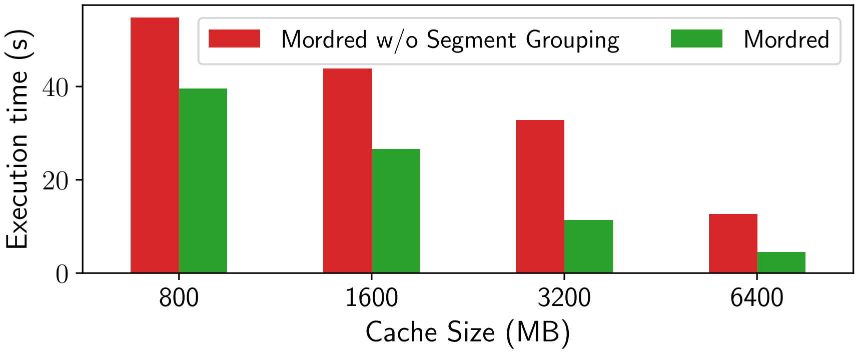


Figure 4.10: Impact of Segment Grouping in Mordred

4.4.3 Evaluating Segment-level Query Execution

One important optimization in segment-level query execution is the grouping phase prior to query execution (see Section 4.2.1). In this section, we will evaluate the performance difference when we enable vs. disable segment grouping. Figure 4.10 shows the results with different cache sizes.

Without segment grouping, the execution engine will launch kernels in the granularity of segments instead of segment groups. This leads to an excessive number of kernels launched, which leads to performance degradation.

With segment grouping, all the segments with the same execution plan are grouped and executed with only a single kernel launch. We see from Figure 4.10 that segment grouping can speed up query execution by up to 3×. The gain increases as the cache size gets larger as more work can be offloaded to the GPU. The segment grouping optimization is novel in Mordred and has not been adopted by existing approach (i.e., HetExchange [62]).

We also measure the breakdown of each phase in segment-level query execution (grouping, execution, and merging). Our experiment shows that for a small cache size (0.8 GB), Mordred spends only 0.3% of the time for grouping, 99.2% of the time for execution, and 0.5% of the time for merging. For a large cache size (6.4 GB), the execution is much faster, and therefore merging and grouping contribute to larger portions of the runtime; Mordred spends 4% of the time for grouping, 93.6% of the time for execution, and 2.4% of the time for merging. In either case, merging and grouping are not performance bottlenecks.

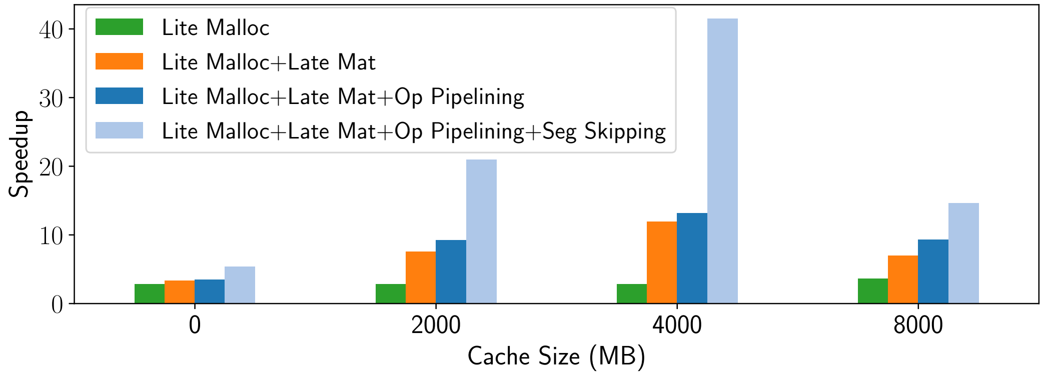


Figure 4.11: Performance Speedup after Each Optimization

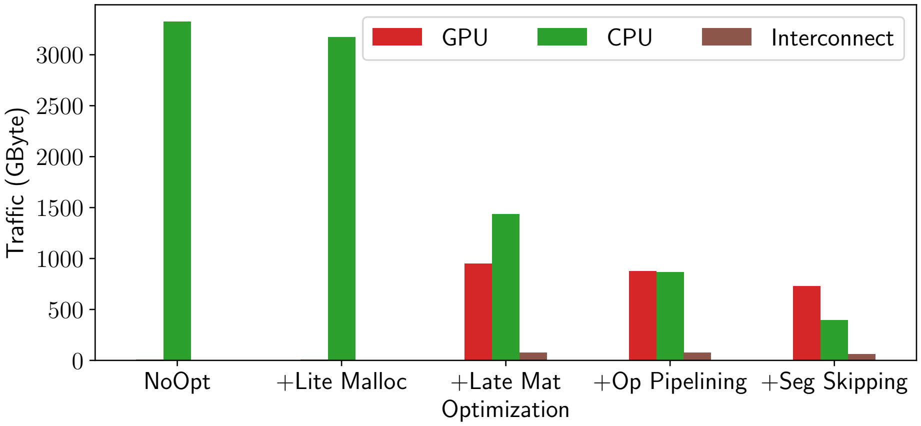


Figure 4.12: Memory Traffic after Each Optimization

4.4.4 Breakdown of Mordred Optimizations

4.4.4.1 Query Runtime Speedup

We now measure the speedup from each optimization applied to Mordred. We evaluate the performance gain from four optimizations: (1) lightweight memory allocation (Section 4.3.1), (2) late materialization (Section 4.2.2.1), (3) operator pipelining (Section 4.2.2.2), and (4) segment skipping (Section 4.2.2.3). We perform the measurement for 4 different cache sizes as shown in Figure 4.11. The query will run completely in CPU when the cache size is 0 and will run completely in GPU when the cache size is 8 GB. For 2GB and 4GB cache sizes, Mordred caches data following the semantic-aware policy.

Across all cache sizes, lightweight memory allocation (`Lite Malloc`) consistently improves performance by around 3.3×, which is due to eliminating the expensive memory allocation operations.

Late Materialization (`Late Mat`) provides performance speedup by up to 3×. Without late materialization, operators are forced to be executed on the CPU since we need to cache the whole relation to execute an operator on the GPU. With late materialization, however, we can offload some operators to GPU or divide the operators between CPU and GPU which will provide significant speedup.

Operator pipelining (`Op Pipelining`) provides an extra speedup by up to 1.3×. Operator pipelining reduces the memory traffic during query execution Since our GPU implementation with Crystal is close to saturating the memory bandwidth, the benefit from operator pipelining is more significant in GPU.

Finally, segment skipping (`Seg Skipping`) provides another 1.6–3× speedup. Segment skipping reduces the amount of data being processed by the query. In the original SSB queries, segment skipping manages to skip 48% of the data across all the queries. The speedup is more significant when the data is partially cached in GPU. This is because it can more accurately identify data that benefits the most from GPU caching. Specifically, when segment skipping is not applied, a full column scan is often required during query execution. This causes the weight of all segments in the column to be incremented following Algorithm 2. When segment skipping is applied, however, the skipped segment will not get its weight incremented. This improves the accuracy of our semantic-aware caching and improves the overall the query performance.

4.4.4.2 Memory Traffic Breakdown

To reveal deeper insight of the performance optimizations, we show the memory traffic breakdown after every optimization is applied. Figure 4.12 shows the breakdown with 1.6 GB cache size (which can hold 20% of all accessed columns).

After lightweight memory allocation, the traffic does not really affected since it just eliminates memory allocation operations. After late materialization, we can offload more work to the GPU. This reduces the CPU memory traffic and increases the interconnect and the GPU memory traffic. After operator pipelining, both CPU traffic and GPU traffic are reduced even further since the intermediate results are not materialized. The reduction in CPU traffic is more significant since the cache size

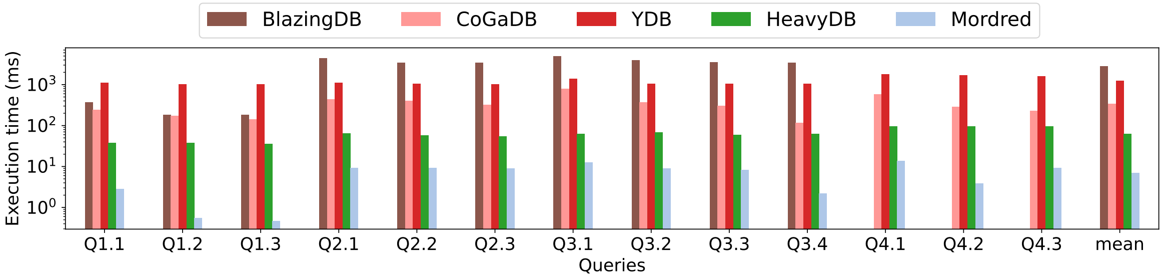


Figure 4.13: SSB Query Performance of Different CPU/GPU DBMS (Data fits in GPU)

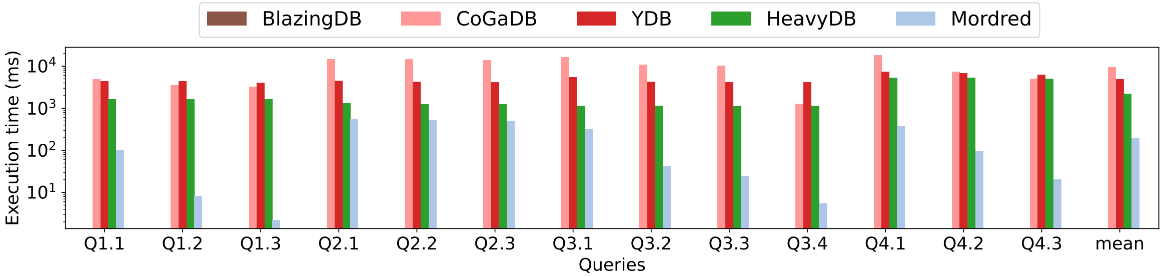


Figure 4.14: SSB Query Performance of Different CPU/GPU DBMS (Data does not fit in GPU)

is small and therefore most of the operators are executed in CPU. Finally, segment skipping reduces the CPU traffic and GPU traffic by reducing the total amount of data being processed by the queries.

To summarize, in this experiment we see how various optimizations in Mordred could: (1) lower the CPU traffic by offloading more work to GPU and (2) lower the total traffic by reducing the data being processed and pipelining operations.

4.4.5 Comparison with Other CPU/GPU DBMS

This subsection reports the end-to-end performance evaluation of four existing CPU/GPU DBMSs:

- **CoGaDB:** CoGaDB [50] is a prototype of column-store CPU-GPU DBMS. CoGaDB uses column-granularity LRU and LFU based replacement policy and utilizes a learning-based optimizer called Hype [57].

- **HeavyDB:** HeavyDB [15] is a commercial GPU DBMS. HeavyDB treats GPU as the primary execution engine. When the data does not fit in GPU, HeavyDB will divide the query plan into multiple stages and execute each stage on GPU one step at a time. To reduce the amount of data transfer, HeavyDB caches the data in GPU using the LRU policy.

- **BlazingDB:** BlazingDB [6] is a commercial GPU DBMS. BlazingDB uses the RAPIDS library [28] as its execution engine.
- **YDB:** YDB [133] is a prototype of column-store GPU DBMS. When the data does not fit, the input data will be transferred to GPU following the coprocessor model.
- **Mordred:** Mordred is our prototype of Hybrid CPU-GPU DBMS which utilizes fine-grain semantic-aware caching policy and segment-level query execution.

We run two sets of experiments: (1) when data fits in GPU and (2) when data does not fit in GPU. For Mordred and CoGaDB, we use 8 GB cache size. For HeavyDB, BlazingSQL, and YDB, we let the system control the GPU memory. When the data does not fit in GPU, we enable the coprocessor mode in HeavyDB and YDB. We use a scale factor of 40 for when the data fits in GPU and a scale factor 160 (4× the cache size) for when the data does not fit in GPU.

4.4.5.1 Data Fits in GPU

Figure 4.13 shows the query performance when the data fits in GPU. For Q4.1-Q4.3, BlazingDB suffers an out-of-memory exception error and not shown in the figure. Compared to BlazingDB, YDB, CoGaDB, and HeavyDB, Mordred is around 400×, 175×, 48×, and 9× faster, respectively. These systems do not have the tile-based execution model in Crystal and thus do not utilize the GPU memory bandwidth as efficiently as Mordred. They also do not have all the performance optimizations we implement in Mordred, such as operator pipelining, segment-grouping, and segment skipping.

Across the 13 queries, Mordred's performance is significantly better compared to other systems especially in Q1.2, Q1.3, and Q3.4. These queries only access data in the range of a week or a month and benefit significantly from the segment skipping optimization.

4.4.5.2 Data Does Not Fit in GPU

Figure 4.14 shows the query performance when the data does not fit in GPU. Since BlazingDB requires the dataset to fit in the GPU, we do not include it in this experiment.

Compared to CoGaDB, Mordred is around 48× faster. Apart from the lack of optimizations described in Section 4.4.5.1, our semantic-aware caching policy will be superior compared to the column-granularity LFU/LRU policy used by CoGaDB.

Compared to HeavyDB and YDB, Mordred is around 11× and 25× faster. This is because when the data does not fit in GPU, HeavyDB and YDB will switch to coprocessor mode and stream the data on demand from CPU memory during query execution. This results in suboptimal performance due to excessive data transfer.

Across the 13 queries, Mordred's performance gain in Q3.1-Q4.3 is more significant than Q2.1-Q2.3. This is because our semantic-aware policy chooses to prioritize caching more segments from Q3.1-Q4.3 which results in performance difference between the query sets. This behavior would not be apparent in other systems which do not adopt semantic-aware policy.

4.5 Related Work

There have been a few previous systems that attempted to leverage both CPU and GPU for query execution [78, 82, 50, 135, 62, 87, 94, 81, 63].

GDB [78] was the earliest effort in this direction. GDB can execute an operator on both CPU and GPU by partitioning the input data prior to execution (e.g. partitioned hash join). OmniDB [136] improved GDB with kernel adapter design so that it could target different hardware architectures efficiently. Unlike Mordred, these systems do not handle data placement between CPU and GPU.

He et al. [81] discussed heterogeneous query execution that specifically targets an integrated CPU-GPU architecture in a single chip (e.g AMD APU). In this architecture, PCIe is no longer a bottleneck but the GPU is less powerful and has a much lower memory bandwidth than the ones in a discrete architecture.

DB2 BLU [94] showed how to use GPU and CPU cores for faster processing in IBM DB2 database. This work uses a heuristic to decide where to execute an operator based on its runtime features (e.g. input size, number of groups, etc). This work, however, only supports limited number of operators (group-by, aggregation, and sort) and does not address data placement between CPU and GPU.

Ocelot [82, 54] is a hardware-oblivious database engine which integrates GPU backend to the in-memory column-store MonetDB [48, 47]. Ocelot uses OpenCL [24] runtime to enable hardware-agnostic operator implementation. For its data placement policy, Ocelot caches the most recently used columns in GPU memory.

CoGaDB [50] is a main-memory DBMS with built-in GPU accelerator. CoGaDB utilizes a framework called Hype [57, 51, 58, 59, 49, 53, 60, 56], which uses a learning-based cost model to assign operators on either CPU or GPU. The latest work of CoGaDB [52] introduced *data-driven operator placement* heuristic as an alternative to Hype (see Section 4.2.1.1). Experiments showed that this heuristic manages to outperform the learning-based optimizer [52]. Mordred also adopts the same heuristic as CoGaDB. Similar to Ocelot, CoGaDB uses column-granularity LRU or LFU policy to cache data in GPU. We compared the performance of Mordred against CoGaDB in Section 4.4.5.

HERO [87] investigated adaptive work placement in heterogeneous computing. Operator placement decision typically depends on the processed and transferred data in terms of data cardinalities. This work proposed a placement optimization strategy that can be completely independent on cardinality estimation of the intermediate result. However, unlike Mordred, this work only focuses on placement optimization and does not address data placement and heterogeneous query execution. HERO is developed as an extensible virtual layer on top of YDB [133]. We compared our performance HERO's core execution engine (YDB) in Section 4.4.5.

Lutz et al. [92] discussed query execution across CPU and multiple GPUs through fast interconnect. This work, however, specifically targets CPU-GPU with NVLINK interconnect instead of PCIe, which is available only for IBM Processor in the current market. Most other processors (e.g., Intel and AMD) still use PCIe as the inter-device interconnect. Moreover, this work only focuses on hash join and does not support general queries like Mordred.

Finally, HetExchange [62, 63] introduced a query execution framework to encapsulate heterogeneous parallelism in hybrid CPU/GPU system through redesigning the classical *Exchange* operator. This framework is also integrated with just-in-time compilation engine to enable operator pipelining. HetExchange, however, does not address the data placement between CPU and GPU and lacks an optimizer component to generate the heterogeneous query plan based on the data location. Mordred addresses these issues through *segment-level query plan* which we described in Section 4.2. Moreover, HetExchange does not support segment-grouping which can provide speedup as shown in Section 4.4.3.

4.6 Conclusion

This work advances the state-of-the-art for heterogeneous CPU-GPU DBMS by contributing in two aspects: (1) data placement and (2) heterogeneous query execution. We introduce semantic-aware fine-grained caching policy which takes into account query semantics, data correlation, and query frequency when determining data placement between CPU and GPU. We also introduce a heterogeneous query executor which can fully exploit data in both devices and coordinate query execution at a fine granularity. We integrate both solutions in Mordred, our hybrid CPU-GPU analytical engine. Evaluation on the Star Schema Benchmark shows that our semantic-aware caching policy manages to outperform the best traditional caching policy by 3×. Mordred also manages to outperform existing GPU databases by an order of magnitude.

Chapter 5

Scaling to Multiple GPUs

In the previous chapter, we have discussed one effective strategy to address GPU memory capacity limitation is by leveraging CPU memory to support larger data sets and execute the query with both CPU and GPU (i.e., *heterogeneous CPU-GPU query processing*) [131, 82, 50, 62]. Such a design can exploit the parallelism of both CPU and GPU while minimizing the data transfer overhead between the two devices.

This solution, however, only applies to heterogeneous CPU-GPU DBMS with a single GPU [131, 82, 50, 96]. In contrast, multi-GPU systems provide significantly more aggregate GPU memory and computational power. They not only allow larger datasets to fit in GPU memory but also enable greater parallelism and performance gains when used effectively. In this work, we aim to address the unique challenge of scaling heterogeneous CPU-GPU DBMS to multiple GPUs.

One key challenge in this design space arises from the dual requirements of both *heterogeneity* (CPUs vs GPUs) and *scalability* (multiple GPUs). To address this challenge, we propose the *Unified Multi-GPU Abstraction*. This abstraction aims to simplify the design space by treating multiple GPUs as a single large monolithic GPU (shown in Figure 5.1). By doing this, we can decouple the design into a two-step process. The first step navigates the design space between CPUs and the Unified Multi-GPU, and the second step navigates the design space across multiple GPUs. Building on this abstraction, we develop **Lancelot**, a heterogeneous CPU and multi-GPU DBMS. Lancelot aims to scale the following critical design aspects in CPU-GPU DBMS to multiple GPUs:

Data placement. Leveraging both CPU and multi-GPU presents us with the opportunity to (1) cache the data collectively across multiple GPUs and (2) replicate data across GPUs to reduce communication. Both caching and replication can lead to better query performance since they can improve GPU utilization and minimize data transfer between devices. One unique challenge for data placement in this architecture is the coordination between caching and replication. These two goals may be in conflict, e.g., more data replicated across GPUs means less data can be cached in total in GPUs. To achieve the best overall performance, Lancelot introduces the *cache-aware replication policy*, a cost-based replication strategy that selectively replicates shuffle-intensive data to strike a balance between caching and replication.

Query execution. Another challenge in scaling a heterogeneous DBMS to multiple GPUs is to transform the heterogeneous query plan to leverage multi-GPU hardware. Previous work such as Mordred [131] introduces *segment-level query execution*, which allows different segments of a column to execute different query plans depending on whether the segments are cached in GPU. This approach, however, does not scale to multiple GPUs, because the number of subquery plans will blow up due to the fact that segments can be located across multiple GPUs. In Lancelot, we solve this challenge by extending the *segment-level query execution* with distributed query processing techniques to run efficiently on multiple GPUs.

Contributions. This work makes the following contributions:

- We develop the *unified multi-GPU abstraction*, which views multi-GPU devices as a single large monolithic GPU to reduce the design complexity in hybrid CPU and multi-GPU DBMS.

- We develop a *cache-aware replication policy* for heterogeneous CPU and multi-GPU DBMS. The policy takes into account the cost of shuffling when replicating data and could coordinate both caching and replication decisions for the best performance.

- We extend *segment-level query execution*, a fine-grained hybrid execution strategy to support query plans on multiple GPUs.

- We build Lancelot, a hybrid CPU and multi-GPU analytical engine that incorporates the proposed optimizations. Our detailed evaluation shows that *cache-aware replication* can lead to 2.5× speedup and Lancelot can outperform existing GPU DBMSes by at least 2× on SSB and 12× on TPC-H.

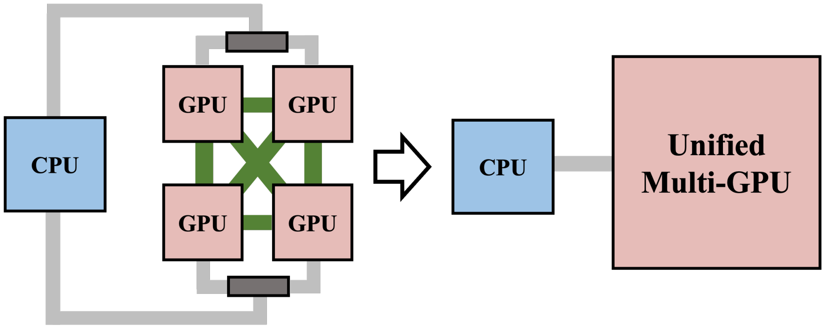


Figure 5.1: Illustration of Unified Multi-GPU Abstraction

Organization. The rest of the chapter is organized as follows: Section 5.1 describes the *unified multi-GPU abstraction*. Section 5.2 describes the *cache-aware replication policy*. Section 5.3 describes our hybrid CPU and multi-GPU query execution strategy and its optimizations. Section 5.4 describes Lancelot's implementation details. Section 5.5 evaluates the performance of Lancelot. Section 5.6 discusses related work and Section 5.7 concludes the chapter.

5.1 Unified Multi-GPU Abstraction

One key challenge in expanding a heterogeneous DBMS to multiple GPUs arises from the need to address both *heterogeneity* and *scalability*. In Section 5.1.1, we describe the challenges and discuss the limitations of previous work. Section 5.1.2 describes our proposed solution, *unified multi-GPU abstraction*, which reduces the design complexity by treating multiple GPUs as a single large GPU.

5.1.1 Challenge

In hybrid CPU and multi-GPU DBMS, one key challenge stems from the need to navigate both *heterogeneity* (CPU vs GPU) and *scalability* (across multiple GPUs) when making design decisions. Such requirements can significantly escalate the design complexity. Prior solutions focus on only one aspect but have not thoroughly explored the design space of data placement and query execution in hybrid CPU and multi-GPU DBMS. For example, HetExchange [62] provides a framework to express hybrid query plans but does not explore the design space of data placement and query execution in CPU and multi-GPU DBMS. HERO [87] explores the design space for operator placement but does not address the challenges in other database aspects, such as data placement and query execution.

5.1.2 Design Abstraction

To simplify the design space in hybrid CPU and Multi-GPU DBMS, we introduce the *Unified Multi-GPU Abstraction*. The main idea behind this abstraction is to view multiple GPUs as a unified, large monolithic GPU, rather than as individual discrete GPUs. Figure 5.1 illustrates this abstraction. With the unified multi-GPU abstraction, the system design is decomposed into two separate steps:

Step 1: Addressing heterogeneity. In this step, we treat multiple GPUs as a single GPU with larger aggregated memory and more processing power. For the case of data placement, the DBMS determines what data should be cached in the *unified Multi-GPU* with respect to the CPU, but does not concern data placement across the GPUs. By doing this, we can directly apply existing techniques that have been developed for heterogeneous CPU-GPU DBMS.

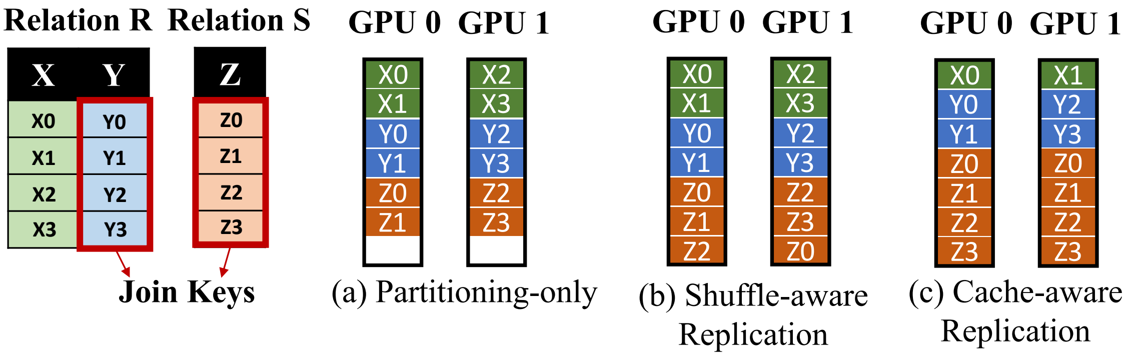
Step 2: Addressing scalability. In the second step, we zoom inside the *unified Multi-GPU* to address the challenges posed by multiple GPUs. In particular, the DBMS will decide how the cached data should be partitioned and/or replicated across the GPUs.

The *unified multi-GPU abstraction* is tailored for systems with homogeneous GPUs, which is common in data centers and the cloud. Moreover, modern GPUs often use fast interconnects like NVLink, capable of up to 900 GB/s [21], making inter-GPU data transfer significantly faster than CPU-to-GPU transfer.

In Lancelot, we use *the unified multi-GPU abstraction* as a guideline to explore the design space in hybrid CPU and multi-GPU DBMS. This abstraction is straightforward and does not require additional formulation. In the next two sections, we will discuss in detail the implementation of this abstraction for data placement (Section 5.2) and query execution (Section 5.3).

5.2 Data Placement

Intelligent data placement can lead to better query performance and memory efficiency in heterogeneous CPU-GPU DBMSes. Lancelot treats data placement as a caching problem following previous works [50, 131] — the complete data set resides



(a) Partitioning-only (b) Shuffle-aware Replication (c) Cache-aware Replication

Figure 5.2: Illustration of Cache-aware Replication Policy – Assuming a Host System with 2 GPUs.

in CPU memory and a mirrored subset of data is cached in GPUs. Lancelot borrows the *semantic-aware fine-grained caching policy* [131], which is the state-of-the-art caching policy in heterogeneous CPU-GPU DBMS.

To address data placement across multiple GPUs, Lancelot treats data placement as a replication problem inspired by previous works in distributed databases [116] — the data set can be either partitioned and distributed across multiple GPUs or replicated across multiple GPUs. Whether a piece of data should be replicated is determined by the benefit of replication (e.g., reduction of network transfer) and the associated cost (e.g., extra disk/memory space consumption).

A unique challenge in hybrid CPU and multi-GPU is the correlation between (1) the caching policy in the *unified Multi-GPU* and (2) the replication policy across GPUs. More data replication across GPUs means less data can be cached in total. To achieve the best overall performance, the caching and replication decisions must be holistically considered to balance their effects on performance.

Lancelot addresses this challenge by developing a *cache-aware replication policy*. The key insight behind the policy is to use a unified cost model to estimate the effects of caching and replication. In Section 5.2.1, we demonstrate how replication can lead to better performance. Then, we explain the proposed policy in Section 5.2.2.

5.2.1 Motivation

5.2.1.1 The benefit of shuffle-awareness

In a multi-GPU system, despite being interconnected with high-speed interfaces like NVLink, data transfer between GPUs can still be the bottleneck of query execution since NVLink bandwidth is lower than that of the GPU device memory. Existing multi-GPU DBMSes always partition the data across multi-GPUs (Figure 5.2a) which often results in significant portions of query execution spent on shuffling data across multiple GPUs [101, 15, 11]. Our experiments show that when running a co-partitioned join with 4 V100 GPUs connected by NVLink, around 50% of the total join runtime is consumed by partitioning and transferring data across GPUs.

By strategically replicating data, we can reduce or even eliminate the data shuffling overhead. Figures 5.2a and 5.2b illustrate the benefit of shuffle-aware data replication. Without data replication (Figure 5.2a), joining relations R and S requires the DBMS to either (1) broadcast column Z (which is a join key) or (2) co-partition R and S on columns Y and Z, respectively. Figure 5.2b shows both GPUs' cache content after *shuffle-aware data replication*. By replicating Z0 and Z2, we now only need to broadcast Z1 and Z3 to perform the join locally in each GPU. Compared to Figure 5.2a, the data transfer overhead during join is now halved. Our experiment running join with 4 V100 GPUs shows that replication can lead to 2× speedup.

5.2.1.2 The benefit of cache-aware replication

As depicted in Figure 5.2b, data replication will consume the available cache size in each GPU. As replication continues, we will reach the limit of GPU memory capacity, bringing us to a decision point where we can either (1) stop replication or (2) opt to continue replication at the cost of evicting some data back to the CPU.

Continuing replication at the expense of evicting cached data may result in worse query performance since it may lead to certain query operations executed on CPUs. Conversely, it may also lead to speedup, particularly if the evicted data are infrequently accessed (cold data) or involved in operations that can be executed efficiently on CPUs. This underlines the importance of integrating cache-awareness into the replication policy to effectively navigate the trade-off between caching and replication. Such property has not been exploited by existing works.

Figure 5.2c illustrates the benefit of *cache-aware* data replication. In this example, column X is only used for aggregation, which does not introduce much overhead when executed on CPUs. Therefore, the policy chooses to replicate data from column

Algorithm 3: Update the replication weight for Segment S

```
# estimate the cost of shuffling
Cost = estimateShuffleCost(shuffled_segments)
for S in shuffled_segments do
    # increment replication weight of segment S
    S.replication_weight += Cost / (|shuffled_segments| × |CARD|)
```

Algorithm 4: Decide between Caching and Replication

```
CacheOrReplicate(segment $S_x$, segment $S_y$):
if $S_x$.replication_weight > $S_y$.caching_weight then
    for gpu in NUM_GPU do
        Cache($S_x$, gpu) # Replicate $S_x$
else
    gpu = mapSegmentToGPU($S_y$) # Hash function to select gpu
    Cache($S_y$, gpu)# Cache $S_y$ in gpu
```

Z instead of caching data from column X. Doing this allows the join to be performed locally in each GPU at the cost of running aggregation on the CPU.

5.2.2 Cache-Aware Replication Policy

The key challenge of a hybrid CPU and multi-GPU DBMS is to find a holistic solution to handle both caching and replication decisions. Replication reduces data transfer between GPUs but also results in caching less data, hence executing more operators on the CPUs. In contrast, caching would ensure more operators to be executed on the GPUs, but may result in excessive data transfer across multiple GPUs. As both caching and replication would lead to performance speedup and space consumption in the GPU memory, we need to resolve the conflict between the two policies to achieve the best overall performance. The goal of our *cache-aware replication policy* is to: (1) selectively replicate data to reduce data transfer overhead, and (2) navigate the trade-off between caching and replication.

The key insight of our policy is to use a unified cost model to estimate the effect of both caching and replication. Our policy splits each column into equal-sized partitions, which we will refer to as *segment* for the rest of the dissertation. The policy will cache and replicate data on segment granularity. Each data segment will be assigned weights derived through cost models which will be used to navigate the trade-off between caching and replication. Section 5.2.2.1 describes the general replication framework and Section 5.2.2.2 describes the cost model used by the policy.

5.2.2.1 Replication Policy

Our *cache-aware replication policy* is inspired by the *semantic-aware caching policy* used in Mordred [131]. In Mordred, each segment will be assigned a weight that reflects the benefit of caching the segment in GPU. The weight is derived using the cost model from Crystal [110] and data will be cached starting from the segment with the highest weight. Lancelot uses the same weight-based mechanism for caching data in GPU. Furthermore, Lancelot generalizes the mechanism to support both *caching* and *replication*.

Algorithm 3 shows the shuffle-aware weight update in our replacement policy. We execute Algorithm 3 each time we shuffle or broadcast data during query execution. The algorithm first predicts the data shuffling cost using the function *estimateShuffleCost*(), which utilizes a simple cost model that will be described in detail in Section 5.2.2.2. Following this, for each segment that is involved in data shuffling or broadcast, we will increment the replication weight by the data shuffling cost divided by the total number of segments involved and normalized by the table cardinality.

Since both caching and replication policies in Lancelot collect per segment weight, each segment now possesses two weighted counters: (1) replication weight from the *cache-aware replication policy* and (2) caching weight from the *semantic-aware caching policy*. Both policies derive their weights from the same cost model [110], so that direct comparison between the two weights becomes feasible. Function *CacheOrReplicate*() outlined in Algorithm 4 can be used to determine whether to replicate Segment S_x or cache Segment S_y in order to occupy the available GPU cache space: if the replication weight of S_x is larger than the caching weight of S_y, we opt for replication of S_x; conversely, if the caching weight of S_y surpasses the replication weight of S_x, we cache S_y instead.

For example, in Figure 5.2, segments from column X have low caching weights since they are only used for aggregation, which does not impose much overhead when executed on CPUs. In contrast, segments from columns Y and Z have high caching and replication weights since joins are expensive and may result in excessive data transfer across the GPUs. Given that the replication weight of column Z is higher than the caching weight of column X, Algorithm 4 would decide to replicate Z rather than caching X to occupy the available GPU memory space.

Since our policy requires prior knowledge of query statistics, Lancelot gathers these statistics and calculates the weight after each run. Caching and replication are performed periodically every *n* queries, with *n* being a user-defined parameter set to 100 by default.

5.2.2.2 Cost Model

This subsection explains how the *estimateShuffleCost*() in Algorithm 3 works. In particular, we use the cost model presented in Crystal [110] and Mordred [131] to estimate the cost of shuffling or broadcasting the data across GPUs. These models operate under the assumption that queries can fully utilize memory bandwidth, thereby deriving execution time from memory traffic. The accuracy of the model has been verified in Crystal on simple operators. Mordred extends the model to support hybrid CPU-GPU execution and PCIe, demonstrating acceptable accuracy for the purposes of caching policy. In this work, we extend the model even further to support operators used in multi-GPU query execution.

In particular, in distributed query execution with multiple GPUs, there is an extra cost for data transfer and data partitioning. We model the cost of data transfer as follows:

$$\mathit{data\_transfer} = \frac{\mathsf{size}(\mathsf{int}) \times N}{BW_{\mathsf{gpu2gpu}}}$$

Where N is the cardinality of input segments and BW_{gpu2gpu} is the interconnect bandwidth between GPUs. We model the cost of data partitioning as follows:

$$\mathit{data\_partitioning} = \frac{\mathsf{size}(\mathsf{int}) \times N}{B_r} + \frac{(1 - \pi) \times N \times C}{B_w}$$

Where B_r and B_w are GPU read and write memory bandwidth, C is the cache line size, and π is the probability the accessed cache line is in the last level cache. Using these equations as building blocks, we can express various communication collectives in *estimateShuffleCost*(). For example, *broadcast* and *all-to-all* can be expressed with a collection of *data_transfer* between GPUs as follow:

$$\textit{broadcast} = \sum_{i=1}^{nGPUs} \frac{size(int) \times N}{BW_{gpu2gpu}}$$

$$\textit{all-to-all} = \sum_{i=1}^{nGPUs} \sum_{j=1}^{nPartition} \frac{size(int) \times N_{ij}}{BW_{gpu2gpu}}$$

Similarly, data shuffling can be expressed as *data_partitioning* on each GPU followed by an *all-to-all communication*.

Since our cost model takes into account the hardware properties (e.g. interconnect bandwidth, GPU memory bandwidth, etc.), our policy can adapt to various hardware and interconnect speeds.

5.3 Query Execution

Scaling heterogeneous query execution to multiple GPUs introduces new challenges and opportunities. First of all, since caching and replication are performed in segment granularity, we need to maintain fine-grain coordination during query execution across the CPU and multiple GPUs. Previous work has attempted to address such challenge with *segment-level query execution* — different segments of a column will execute different subquery plans depending on the segments' location — but does not extend the solution for multiple GPUs. In Lancelot, we leverage the *unified multi-GPU abstraction* to extend *segment-level query execution* to multiple GPUs.

Moreover, as discussed in Section 5.2.1.1, multi-GPU query execution can introduce significant data transfer overhead. In Lancelot, we further minimize this overhead by applying several optimizations from distributed query processing to multi-GPU query execution. Section 5.3.1 describes the general query execution framework in Lancelot and Section 5.3.2 describes various optimizations in Lancelot to speedup the query performance.

5.3.1 Scaling Query Execution to Multiple GPUs

By leveraging the *unified multi-GPU abstraction*, we can decouple query execution into two steps: (1) query execution between CPU and the *Unified Multi-GPU*, and (2) query execution across multiple GPUs. We will discuss how Lancelot addresses each

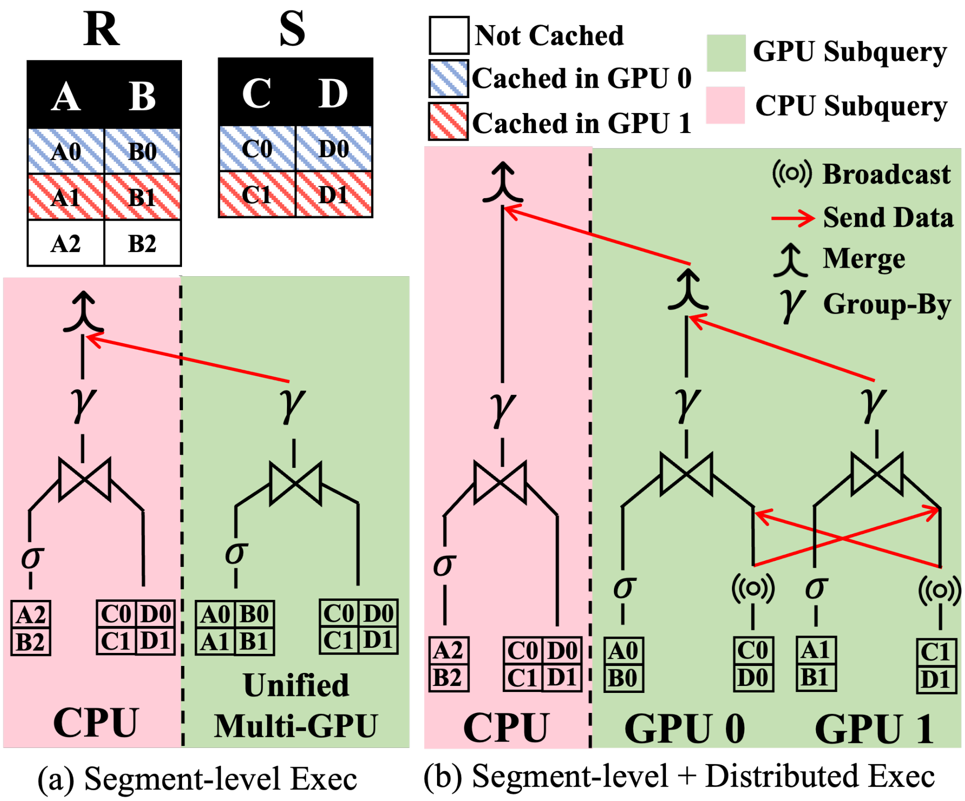


Figure 5.3: Example of Query Execution in Lancelot

step individually in Section 5.3.1.1 and Section 5.3.1.2. Then, we show an example of query execution in Lancelot in Section 5.3.1.3.

5.3.1.1 Query Execution between CPU and the Unified Multi-GPU

One unique challenge in query execution involving both CPU and multiple GPUs lies in the communication between the two device types. Lancelot covers 4 different data transfer scenarios:

Transferring different partitions from CPU to GPUs: In this scenario, data is partitioned on the CPU before each partition is sent to different GPUs. Lancelot will utilize all available PCIe lanes to transfer the data from the CPU to each GPU efficiently.

Broadcasting data from CPU to GPUs: For NVLink-connected GPUs, Lancelot will send the data to one GPU, which then broadcasts the data to all the other GPUs. Broadcasting data from the GPU is typically faster due to the high-speed NVLink interconnect. For PCIe-connected GPUs, Lancelot will utilize all available PCIe lanes to broadcast the data from the CPU to each GPU.

Transferring groupby/aggregation results from GPUs to CPU: Since merging results in GPU is faster than CPU, Lancelot will first merge the groupby result in one GPU, before sending it to the CPU.

Transferring join/filter results from GPUs to CPU: In this case, Lancelot will use all the available PCIe lanes to directly transfer the different partitions from each GPU to the CPU.

To coordinate fine-grain query execution across CPUs and GPUs, Lancelot adopts the *segment-level query plan* [131]. *Segment-level query plan* allows different segments of a column to execute different query plans depending on the segments' location. Lancelot uses a simple heuristic to push an operation to where the segments reside to minimize the data transfer. Figure 5.3a shows an example of a segment-level query plan. To support multi-GPU query execution, Lancelot will simply expand each GPU subquery plan to a *distributed query plan* (Figure 5.3b). Doing this will prevent the heuristic from generating an excessive number of subquery plans.

5.3.1.2 Query Execution across Multiple-GPUs

Lancelot executes a multi-GPU query plan similar to how a distributed query engine works. For distributed join, Lancelot supports both broadcast and co-partitioning join. For filtering, Lancelot executes the operator locally in each GPU. For group-by and aggregation, Lancelot first executes the operator locally in each GPU, followed by merging the results across all the GPUs.

Lancelot provides three multi-GPU communication routines to support distributed query plans as follows:

Broadcast: This routine is used to send data from one GPU to all the other GPUs. Common use cases include broadcast join and broadcasting intermediate results from the CPU.

Gather: This routine is used to collect data from all the GPUs to one GPU. Mainly used for merging group-by results across GPUs.

All-to-all: This routine is used in co-partitioned join to exchange data between every pair of GPUs, allowing a GPU to send and receive its partition from every other GPU.

Lancelot implements these routines by leveraging NCCL [17], a topology-aware multi-GPU communication library that has been optimized to achieve high throughput on multi-GPU platforms.

5.3.1.3 Example of Query Execution

```
Q0: SELECT S.D, SUM(R.A) FROM R,S
    WHERE R.B = S.C AND R.A > 10 GROUP BY S.D
```

Figure 5.3 illustrates an example of multi-GPU query execution in Lancelot when executing Q0. In this example, R is the probe relation and S is the build relation. In Figure 5.3a, we can see how the query plan is divided into subquery plans following the *segment-level query plan* based on where the data is originally located. Specifically, the query is divided into 2 subqueries:

CPU Subquery: This subquery operates on uncached data ($A2$, B2) and executes filter, join, and group-by on CPUs.

GPUs Subquery: This subquery operates on cached data ($A0$, B0, $A1$, B1) and executes filter, join, and group-by on GPUs.

Lancelot will launch both subqueries in parallel to utilize all the available computation power of CPUs and multiple GPUs.

Figure 5.3b shows the generated query plan after Lancelot expands the **GPU Subquery** into a *distributed query plan* for 2 GPUs. For simplicity, we will use broadcast join as our join strategy, although other approaches such as co-partitioning join can also be applied. To perform the broadcast join, segments C0, C1, D0, and D1 will be broadcasted such that each GPU has the complete copy of relation S. Doing this will allow all remaining operations (filter, join, group-by, and aggregation) to be executed locally on each GPU. Finally, after each GPU computes its final result, results from GPU 0 and GPU 1 are merged in GPU 0 before being transmitted to the CPUs for the final aggregation with the **CPU subquery**.

5.3.2 Reducing Data Transfer Overhead

In this section, we describe optimizations we adopt in Lancelot to further reduce the data transfer overhead across GPUs. We will discuss three optimizations: late materialization (Section 5.3.2.1), adaptive join (Section 5.3.2.2), and join reordering (Section 5.3.2.3).

5.3.2.1 Multi-GPU Late Materialization

During query execution, transferring intermediate relations can often be expensive. For example, a co-partitioned join operation will partition either relation into multiple smaller partitions, and transfer each partition to the corresponding GPUs. Previous CPU-based columnar databases used the late materialization strategy to reduce data transfer by expressing intermediate relations in the form of row IDs. Lancelot employs a late materialization strategy specifically tailored for multi-GPU query execution.

During the partitioning phase in co-partitioning join, Lancelot will reconstruct only the join key columns used by the query. The remaining columns in the relation will be expressed in the form of row IDs. For subsequent query operations, Lancelot reconstructs the accessed columns using row IDs lazily, by leveraging the *Unified Virtual Addressing* (UVA). UVA is a GPU-specific technology that allows GPU kernels to directly access peer GPUs' memory in byte granularity through the device interconnect (e.g. NVLink).

For queries with highly selective joins, this reconstruction cost can be negligible, reducing the overall data transfer overhead. For queries with non-selective join, however, reconstruction can introduce non-trivial overhead due to random accesses to the peer GPUs' memory. Hence, to decide whether to do late materialization, Lancelot leverages a cost model to estimate the reconstruction overhead as follows:

$$\mathit{reconstruction} = \frac{N \times C}{BW_{gpu2gpu}}$$

Where N is the number of tuples to reconstruct and C is the cache line size. Based on this estimation, Lancelot will do late materialization if the reconstruction cost is cheaper than the cost of broadcasting or shuffling the input data (obtained from Section 5.2.2.2).

5.3.2.2 Adaptive Join Strategy

Lancelot supports two types of distributed joins: broadcast join and co-partitioning join. Since both joins have different costs, choosing the right join strategy could lead to performance speedup. For example, when joining with small relations, broadcast join is often more efficient compared to co-partitioning join, since the broadcasting small relations incurs little overhead. To decide between the two strategies, Lancelot once again leverages the cost model in Section 5.2.2.2 to estimate the cost of both joins and choose the join strategy with a lower cost. While the model might not be precise, we find the accuracy to be acceptable for this purpose.

5.3.2.3 Join reordering

We can further reduce the data transfer during query execution by reordering the joins based on their join strategy. In particular, if we have an n-way joins with a left-deep join tree query plan, Lancelot will reorder the co-partitioning joins to the end of the join pipeline. Executing co-partitioning join at the start of the join pipeline can be expensive since we would need to shuffle the full input relations that are relatively large. By pushing the joins to the end of the pipeline, we can speed up the co-partitioning joins as the sizes of the input probe relations would often have been reduced by the selectivity of the prior joins in the pipeline.

5.4 Systems Integration

This section describes the implementation of Lancelot, our hybrid CPU and multi-GPU DBMS. Figure 5.4 illustrates the architecture of Lancelot, which consists of three main modules described in the following subsections: Buffer Manager (Section 5.4.1), Query Optimizer (Section 5.4.2), and Query Execution Engine (Section 5.4.3).

5.4.1 Buffer Manager

The Buffer Manager is a CPU component that manages data placement on the CPU and across multiple GPUs in Lancelot. It divides each GPU memory into two regions:
Data Caching: This is a region that stores the cached or replicated data in segment granularity. It performs periodic data placement by leveraging *semantic-aware caching*

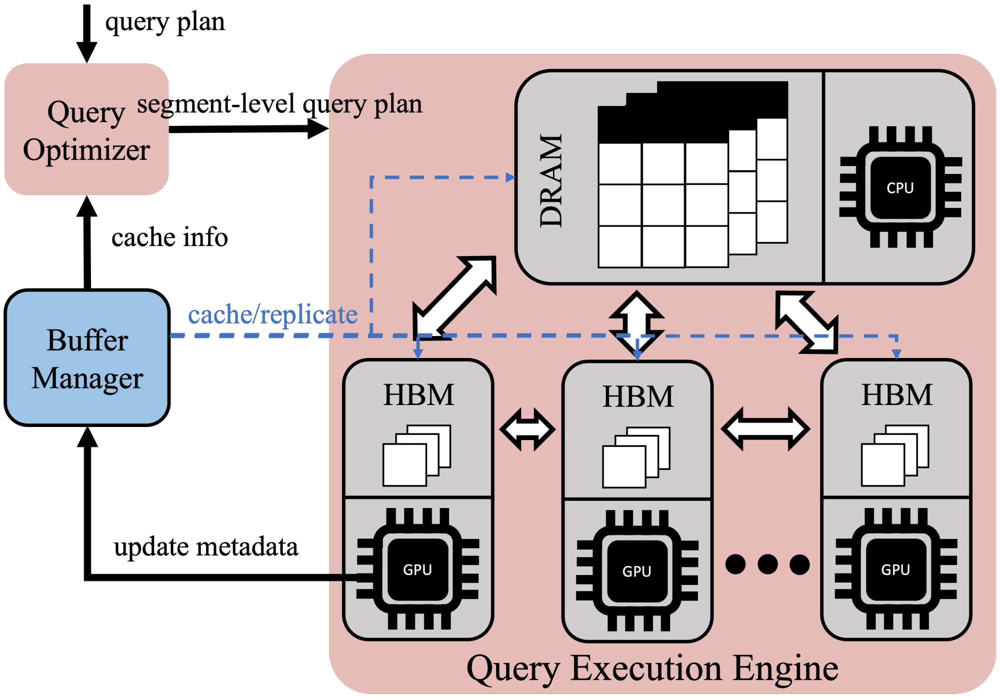


Figure 5.4: Lancelot System Architecture

and *cache-aware replication policy* discussed in Section 5.2. The segment size is user defined and set to 4MB by default.

Data Processing: This is a region which stores intermediate results during query execution (i.e., hash tables, intermediate results, etc.).

Both the data caching and the processing region are pre-allocated with a fixed size during system initialization to avoid frequent dynamic memory allocation in the GPUs.

The Buffer Manager also handles the metadata management in Lancelot. The two main data structures used by our buffer manager are an array of free lists (one free list per GPU) and a 2-level hash table. We use the free list in each GPU to track the available slots in their respective caches. We use the 2-level hash table to store the location of each segment. The first level maps each segment to the corresponding GPU, and the second level maps to an address in the GPU memory. Although using a single-level hash table is also possible, we opt for 2-level hash table due to its simplicity, especially since metadata management has a negligible impact on query performance. The Buffer Manager also stores the segment statistics in Lancelot, such as the replication weight, caching weight, and the min-max of each segment.

The metadata of Lancelot resides in the CPU memory. When a GPU kernel requires some metadata, the Buffer Manager will send the necessary metadata to the GPUs prior to launching the kernel.

5.4.2 Query Optimizer

The query optimizer converts a query plan to the segment-level query plan shown in Figure 5.3(b). Our original query plan is taken from Crystal [110] which is already optimized for GPUs.

To decide which operator should be executed on GPUs, our optimizer leverages the *data-driven operator placement*[52], where we execute operators on GPUs only when the input data is cached in at least one of the GPUs. Subsequently, to construct the multi-GPUs query plan, the optimizer will replace the join with either broadcast join or co-partitioning join and apply the optimizations in Section 5.3.2.2 and Section 5.3.2.3. For each group-by and aggregation, the optimizer will insert a merge operator following the group-by.

5.4.3 Query Execution Engine

The Query Execution Engine in Lancelot executes the query plan generated by the query optimizer across CPU and multiple GPUs. In particular, similar to Mordred, it executes each subquery plan in parallel and merges the results at the end.

CPU execution in Lancelot is implemented with Intel TBB. GPU execution in Lancelot is implemented with CUDA, using Crystal library [110]. Lancelot extends Crystal to support multi-GPU query execution, such as distributed join and merge operations. Communication across multiple GPUs is implemented using NCCL [17] and the `cudaMemcpy()` primitives in CUDA. To launch a multi-GPU kernel, Lancelot switches the device context to each GPU using `cudaSetDevice()` API and launch the kernel asynchronously in each GPU by leveraging the `cudaStream_t` primitives.

5.5 Evaluation

In this section, we evaluate the performance of Lancelot. The section will answer the following key questions:

- How does Lancelot scale as we increase the number of GPUs?

- How does the *cache-aware replication policy* perform compared to other data placement policies?
- How much performance improvement is achieved through various optimizations introduced in Section 5.3.2?
- How does Lancelot perform compared to other GPU DBMSs?

5.5.1 Experimental Setup

Hardware configuration: We ran our experiment with three different compute instances:

- **GPU3.8 instance (OCI):** This instance features 8 NVIDIA V100 GPUs connected through NVLink 2.0 with up to 300 GB/s bidirectional bandwidth per GPU. Each V100 GPU has 16 GB of HBM2 memory with a read/write bandwidth of 880 GBps. These GPUs are connected to 52-Cores Intel® Xeon® E5-2698 via PCIe3 with 12.8 GB/s bidirectional bandwidth.
- **GPU4.8 instance (OCI):** This instance features 8 NVIDIA A100 GPUs connected through NVLink 3.0 with up to 600 GB/s bidirectional bandwidth per GPU. Each A100 GPU has 40 GB of HBM3 memory with a read/write bandwidth of 1550 GB/s. These GPUs are connected to 64-Cores AMD EPYC 7542 (Rome) CPUs via PCIe4 with 25.6 GB/s bidirectional bandwidth.
- **g6.48xlarge (AWS):** This GPU instance features 8 NVIDIA L4 GPUs paired with 96-Cores AMD EPYC 7R13 (Milan) CPUs. Each L4 GPU has 24 GB of GDDR5 memory with a read/write bandwidth of 320 GB/s. The GPUs and CPUs are connected via PCIe4 with 25.6 GB/s bidirectional bandwidth. Since there is no NVLink, GPU-to-GPU communication also occurs through PCIe.

Benchmark: Most of our experiments use the *Star Schema Benchmark* (SSB) [97]. The SSB dataset has five tables with one fact table and four dimension tables. Since the dimension tables in SSB are very small (< 0.5% of the fact table), we increase the size of the dimension table to demonstrate more interesting scenarios when comparing different data placement policies. The supplier, customer, and part tables are now 5%, 15%, and 25% of the lineorder table size respectively. In Section 5.5.5, we will also

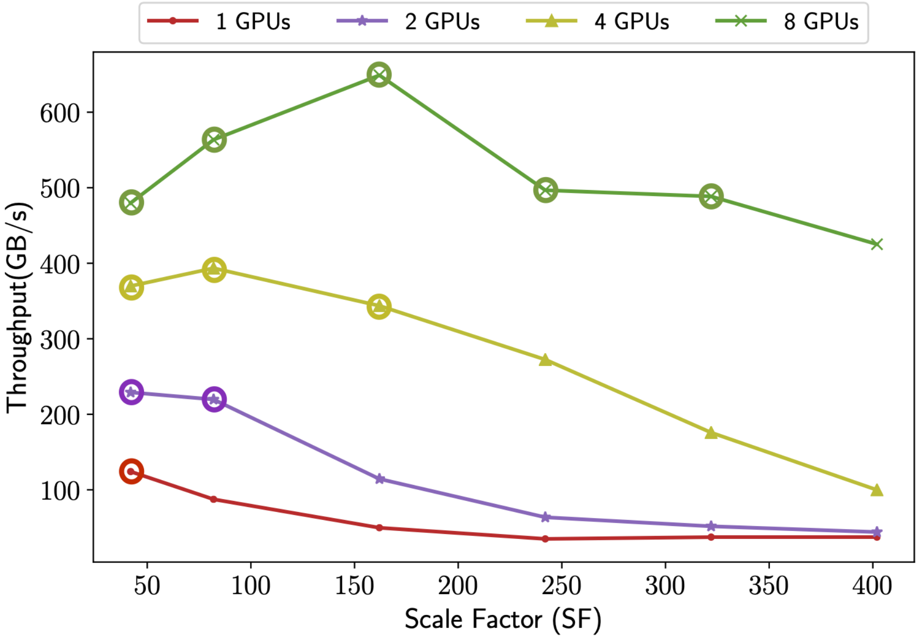


Figure 5.5: Performance Throughput of Lancelot with Different Scale Factors and Number of GPUs

run the experiments with a subset of the TPC-H benchmark. We follow the method described in [117] to select a representative subset of queries.

To enable efficient query execution in GPU, we store the data in columnar stores and dictionary encode the string columns into integers prior to data loading. Therefore, we ensure that all column entries are 4-byte in value. In our evaluation, the entire data set is loaded to CPU memory before each experiment starts.

Measurement: Unless otherwise stated, we dedicate 50% of each GPU memory for data caching, and the other half for data processing. Before each experiment, we first warm up the GPU memory by running 100 random queries and then perform the caching and replication to populate the GPU memory. For each experiment, we will run 100 random queries and measure the query execution time.

5.5.2 Scalability Evaluation

This subsection evaluates how Lancelot can scale to multiple GPUs. In this experiment, we use a GPU3.8 instance and sweep the scale factor from 40 to 400, measuring throughput when running 100 random SSB queries. $SF = 40$ translates to a total of 7.5GB of data that are accessed by all the SSB queries (fits in a single GPU cache). The data size of other scale factors (80–400) is a multiple of $SF = 40$. In this experiment, the throughput is calculated as the total amount of input data scanned by the queries divided by the total query execution time. Figure 5.5 shows the result of the experiment when running with 1, 2, 4, and 8 GPUs. *The circles around the data points indicate that the data still fits in the aggregated GPU memory*.

Overall, as the data size increases on the x axis, the throughput decreases especially when the data can no longer be replicated (more data transfer overhead) or no longer fits in the aggregated GPU memory (queries are partially executed on the CPU). For example, when running with 1, 2, and 4 GPUs, data exceeding $\mathsf{SF} = 40$, $\mathsf{SF} = 80$, and $\mathsf{SF} = 160$ respectively, no longer fits in the GPU memory, leading to diminished performance. The degradation, however, can be kept minimum due to our intelligent data placement policy.

Figure 5.5 also shows that increasing the number of GPUs does not always scale throughput linearly. For example, the throughputs of 2, 4, and 8 GPUs for $\mathsf{SF} = 40$ are only 1.85×, 3×, and 3.85× higher than a single GPU. This is because when running $\mathsf{SF} = 40$ on 4 and 8 GPUs, each GPU only processes a small amount of the data, leading to underutilization. In Section 5.5.5.1, we will show an experiment investigating the scalability when each GPU is fully utilized.

For 8 GPUs, the throughput increases up to $\mathsf{SF} = 160$ and then declines. At $\mathsf{SF} = 40$ and $\mathsf{SF} = 80$, using 8 GPUs for query execution results in GPU underutilization, which limits the throughput. As the data size increases, throughput improves, peaking at $\mathsf{SF} = 160$ when each GPU is fully utilized. At $\mathsf{SF} = 240$ and $\mathsf{SF} = 320$, Lancelot can no longer fully replicate the dimension tables, leading to increased data transfer and reduced throughput. Finally, at $\mathsf{SF} = 400$, the data no longer fits in GPU memory, further lowering throughput as queries will be partially executed on the CPU.

5.5.3 Comparisons between Different Data Placement Policies

This subsection evaluates the performance of the *cache-aware replication policy* against other data placement policies. In this experiment, all compared policies will use the *semantic-aware caching policy* as their caching policy. We will then compare the performance of three different replication policies:

- **Replication-Only:** This policy will replicate all the segments that are cached across all the GPUs.

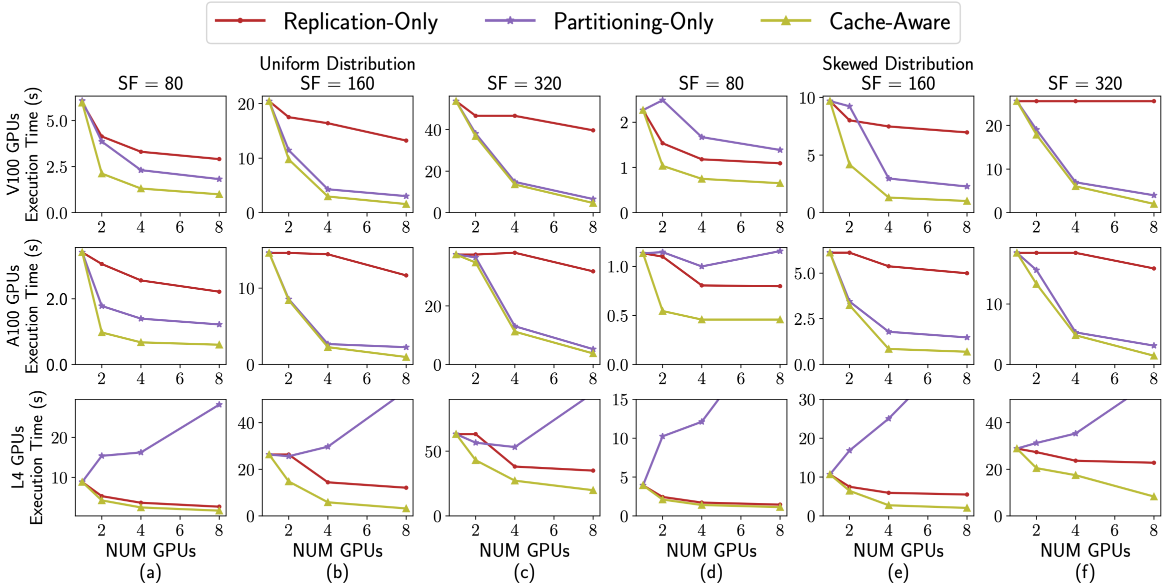


Figure 5.6: Execution Time of Different Data Placement Policies with Uniform (a–c) and Highly-Skewed (d–f) Distribution.

- **Partitioning-Only:** This policy will partition all the segments that are cached across all the GPUs. Segments are assigned to different GPUs in a round-robin fashion.

- **Cache-Aware Replication:** The *cache-aware replication policy* described in Section 5.2.

5.5.3.1 Performance on Standard SSB

In this experiment, we sweep the number of GPUs and measure the query execution time of different data placement policies when running 100 random SSB queries. We run each experiment on three different GPU instances: GPU3.8 (V100), GPU4.8 (A100), and g6.48xlarge (L4); For each configuration, we will use 8GB cache size per GPU. We sweep the number of GPUs from 1 to 8 and run the experiment with SF = 80, SF = 160, and SF = 320. Therefore, all columns that are accessed by queries fit in 2 GPUs for SF = 80, 4 GPUs for SF = 160, and 8 GPUs for SF = 320. The query access distribution is uniform following the default configuration.

Figures 5.6(a)–5.6(c) show the result of our experiment. For V100 GPUs (top) and A100 GPUs (center), *Partitioning-Only* performs better than *Replication-Only*.

On the other hand, for L4 GPUs (bottom), *Replication-Only* performs better than *Partitioning-Only*. *Cache-Aware* outperforms the other policies in all the cases.

Replication-Only fails to efficiently leverage the combined GPU capacity due to the necessity of replicating each segment across all GPUs. While this approach eliminates the data transfer overhead between GPUs, it also minimizes data caching within the GPU memory. Consequently, query execution primarily occurs on the CPU, leading to performance degradation.

Partitioning-Only can cache the most data in GPUs since none of the data will be replicated. As a result, it can execute a bigger portion of queries on GPUs compared to the other two policies. However, the performance may be suboptimal since fully partitioning data across the GPUs could incur significant data transfer overhead. This has been particularly evident with L4 GPUs (bottom). Since these GPUs do not feature NVLink, inter-GPU communication occurs through PCIe, causing significant performance degradation. On V100 and A100 GPUs, this policy performs better as transferring data is cheaper due to the fast NVLink interconnect.

Our *Cache-Aware replication* outperforms the other policies in all cases by up to 2.5×. Our policy will replicate segments that give the most replication benefit which leads to speedup compared to *Partitioning-Only*, but stops replication if it starts to hurt caching performance. Thus, it can better utilize the GPU memory capacity while minimizing the data transfer overhead across GPU devices.

5.5.3.2 Performance on Skewed Workload

This experiment evaluates the performance of *cache-aware replication policy* under highly skewed query access distribution. To simulate nonuniform distribution, we incorporate skewness into the date predicates of SSB queries such that more recent data has a higher probability of being accessed by the query. We pick the values following a Zipfian [76] distribution, resulting in 90% of the queries accessing the data from the last 3 years. Figures 5.6(d) - 5.6(f) show the results of the experiment.

Overall, *Cache-Aware Replication* outperforms the other policies across all scale factors and hardware configurations by up to 3×. *Replication-Only* performs better on the skewed workload than on the uniform workload. This is because caching only the hot data can already give us a decent performance speedup for highly skewed workloads, leaving more GPU memory capacity to benefit from replication. In fact,

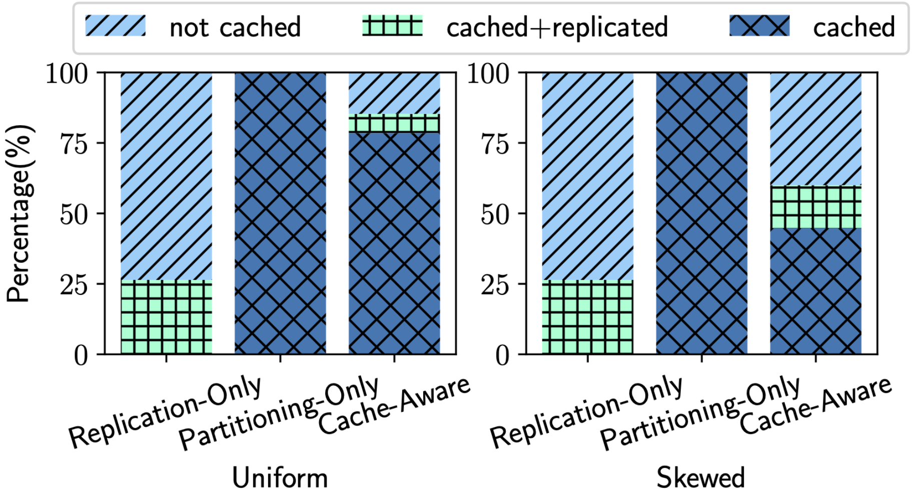


Figure 5.7: Memory Statistics of Various Data Placement Policies (SF = 160, 4 GPUs)

despite the faster NVLink interconnect on V100 (top) and A100 (center) GPUs, *Replication-Only* can outperform *Partitioning-Only* at smaller scale factors (SF=80 and SF=160).

We observe more speedup compared to the uniform distribution. This advantage stems from the policy's adept handling of skewed workloads, prioritizing the replication of segments from dimension tables over caching colder data from the fact table. This strategic decision yields more substantial speedup in the highly skewed workload. To better understand the decision making in our policy, we will dive deeper into the memory statistics in Section 5.5.3.3.

5.5.3.3 Memory Statistics

In this experiment, we show the memory statistics for each data placement policy. We use 4 GPUs and $SF = 160$ on GPU3.8 instance and show the memory statistics for both uniform and skewed query access patterns.e Figure 5.7 shows the result of our experiment. The y-axis in the Figure indicates the percentage of data (from the entire dataset) that are (1) cached, (2) cached and also replicated across all the GPUs, and (3) not cached in the aggregated GPU memory.

For *Replication-Only*, the data is either cached+replicated or not cached. This results in a total of only 25% of the GPU memory being used to cache data. Since most of the data is not cached, a large portion of query execution will be executed on

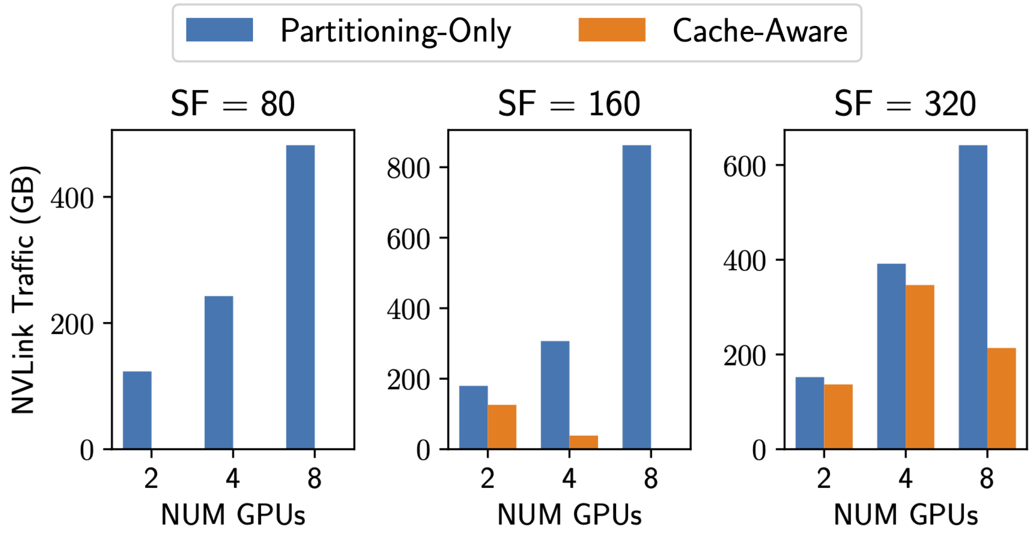


Figure 5.8: NVLink Traffic of Various Data Placement Policies

the CPU which results in significant performance overhead.

For *Partitioning-Only*, the data that are being accessed by the queries can be cached but not replicated across the GPUs. For NVLink-connected GPUs, this results in better performance compared to *Replication-Only* but can result in worse performance in PCIe-connected GPUs.

For *Cache-Aware Replication*, there is more variation of data that are being cached, being replicated, and not being cached. On uniform workload, a total of 84% of the data is being cached, and among those 84%, 8% are replicated across all the GPUs. Conversely, 16% of the data is not cached in the GPUs. Further investigation reveals our policy decides to replicate the smaller dimension tables at the cost of not caching less relevant columns. For example, segments from columns that are used in filters (lo_discount) and aggregation (lo_extendedprice) are not cached since those operations are not expensive on the CPUs (relatively to columns that participate in joins). On skewed workload, our policy decides to cache less data in total (60%) but replicate more data at the same time (17%). This is because, for skewed workload, the cold data (data from earlier years) are less relevant since it will be pruned by *min-max filtering*. Instead, the policy will prioritize replicating segments from the dimension tables to reduce the data transfer across GPUs.

5.5.3.4 NVLink Traffic

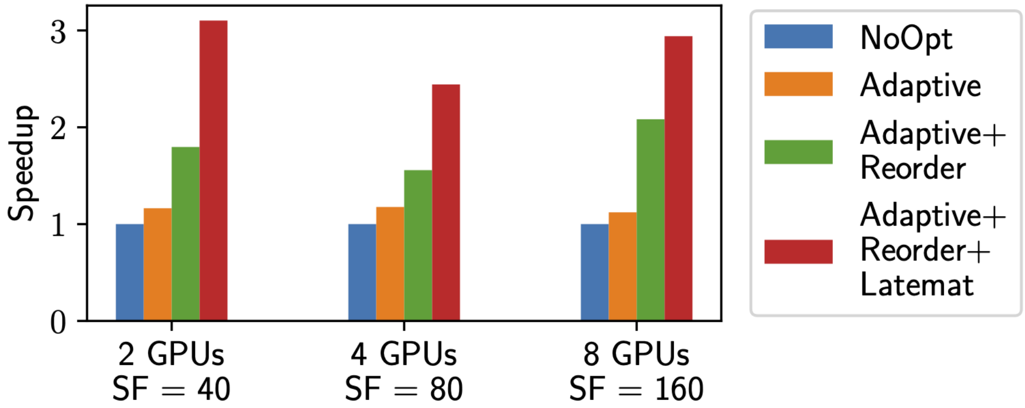


Figure 5.9: Speedup after Each Optimization in Lancelot

To gain deeper insight on each data placement policy, we compare the data traffic going through the NVLink interconnects for each policy. To obtain this metric, we simply aggregate the total *bytes* going across each NVLink channel in each GPU in both directions. Since *Replication-Only* will not incur any data traffic across GPUs, we do not include it in this experiment. We use the uniform data distribution used in Section 5.5.3.1 for this experiment.

Figure 5.8 shows the NVLink traffic when running on GPU3.8 instance. For SF = 80, *Cache-Aware Replication* incurs almost no NVLink traffic since it manages to fully replicate the dimension tables. For SF = 160, *Cache-Aware Replication* has higher traffic for 2 GPUs compared to 4 GPUs. Further investigation reveals that with 2 GPUs, the policy replicates only 2 dimension tables (supplier and date), while with 4 GPUs, it accommodates replication of 3 dimension tables (supplier, date, and customer) by evicting less-performance-critical segments in the fact table (e.g., filter and aggregation columns). A similar trend is observed for SF=320, which results in less traffic for 8 GPUs compared to 4 GPUs.

Overall, across all experiments, *Cache-Aware Replication* reduces the NVLink traffic compared to *Partitioning-Only*. This results in faster query runtime as shown in Figure 5.6.

5.5.4 Breakdown of Optimizations in Lancelot

5.5.4.1 Speedup after Each Optimization

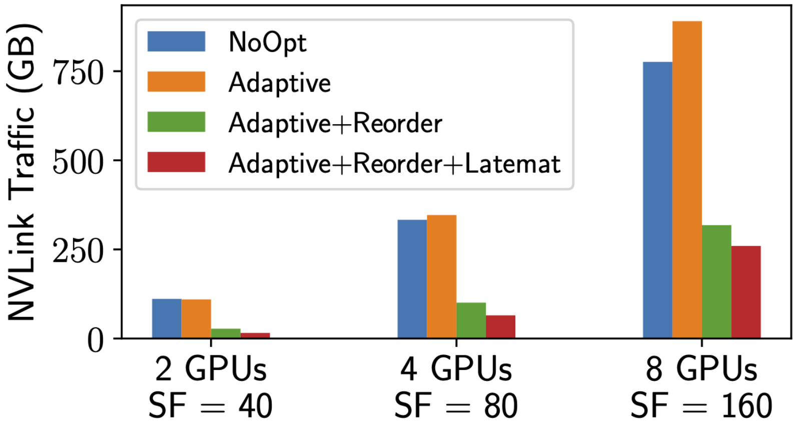


Figure 5.10: NVLink Traffic after Each Optimization in Lancelot

We now measure the speedup from each optimization applied to Lancelot. We evaluate the performance gain from three optimizations: (1) *late materialization* (Section 5.3.2.1) (2) *adaptive join strategy* (Section 5.3.2.2), and (3) *join reordering* (Section 5.3.2.3). We conduct the experiment on GPU3.8 instance and sweep the number of GPUs alongside the scale factor as shown in Figure 5.9. To demonstrate the best speedup, we choose a data size in which all the query processing will be done on GPUs. For this experiment, we reduce each GPU cache size from 8GB to 4GB since otherwise, `NoOpt` will suffer from insufficient memory.

When no optimization is applied (`NoOpt`), we default to always use the co-partitioned join as our distributed join strategy. We will join in the order of selectivity starting with the part, supplier, customer, and date table. Without late materialization, we always materialize the relation after each partitioning phase.

Across all scale factors, *adaptive join* (`Adaptive`) gives around 1.2× speedup. With this optimization, broadcast join will be used when joining with smaller tables, whereas co-partitioned join will be used when joining with larger tables.

By incorporating the *join reordering* (`Reorder`), we will reorder the join such that all the broadcast joins will occur first, followed by co-partitioning joins. Doing this can significantly reduce the data transfer between GPUs during co-partitioned joins. We observe up to 2× speedup after applying this optimization.

Finally, *late materialization* (`Latemat`) will give another 1.6× speedup. Our late materialization strategy will reconstruct only the join key after each partitioning phase, and utilize UVA to materialize the rest of the columns lazily during query execution.

5.5.4.2 NVLink Traffic

To reveal deeper insight, we will show the NVLink traffic after each optimization is applied. Using the same setup as Section 5.5.4.1, Figure 5.10 shows the result of this experiment.

After applying the adaptive distributed join strategy (`Adaptive`), we observe higher NVLink traffic for 4 and 8 GPUs. This is because switching from co-partitioned join to broadcast join can potentially increase the amount of data transferred across the GPUs. However, this does not necessarily translate to slower query execution time since the partitioning phase in co-partitioned join can also introduce non-trivial overhead and is not reflected by the NVLink traffic.

Reordering joins (`Reorder`) significantly reduces the NVLink traffic by up to 2.5×. Moreover, the late materialization (`Latemat`) can further reduce the NVLink traffic by up to 20%. These traffic reductions will translate to the speedup we observe in Section 5.5.4.1.

5.5.5 Comparison with Other GPU DBMSes

This subsection reports the end-to-end performance evaluation of four existing CPU/GPU DBMSes:

- **DuckDB:** DuckDB [105] is an embedded CPU-based analytical database which supports columnar engine and parallel execution.

- **Dask-cuDF:** Dask-cuDF [11] is a multi-GPU extension of cuDF [9], a GPU-accelerated DataFrame library. Dask-cuDF caches the data in GPUs using the LRU policy.

- **HeavyDB:** HeavyDB [15] is a commercial multi-GPU DBMS. HeavyDB caches the data in GPUs using the LRU policy and simply partition the data across all the GPUs.

- **Mordred:** Mordred [131] is a hybrid CPU-GPU DBMS which utilizes the *semantic-aware caching* and the *segment-level query execution*. Mordred, however, only supports a single GPU.

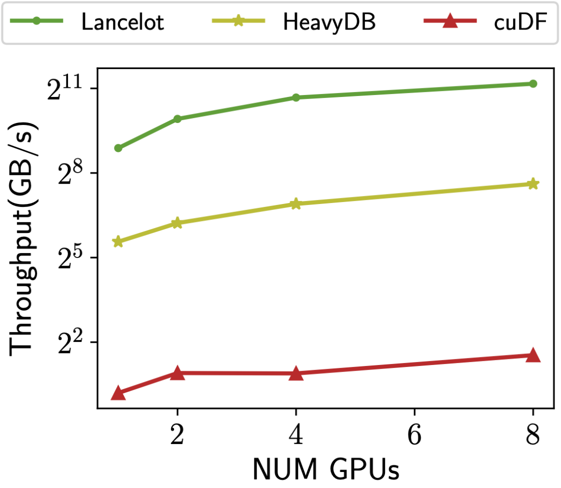


Figure 5.11: Scalability Comparisons

- **Lancelot:** Lancelot is our prototype of Hybrid CPU and Multi-GPU DBMS explained in Section 5.4.

In this Section, we will run four sets of experiments: (1) scalability comparison (Section 5.5.5.1), (2) query performance on SSB (Section 5.5.5.2), (3) query performance on TPC-H (Section 5.5.5.2), and (4) query performance on various hardware configurations (Section 5.5.5.3). Throughout the experiment, for `Lancelot` and `Mordred`, we allocate half of the GPU memory as the cache size for each GPU. For `HeavyDB` and `Dask-cuDF` we let the system control the GPU memory. For this experiment, instead of scaling up the dimension table size as described in Section 5.5.1, we conducted the experiment using the original dimension table sizes in SSB and TPC-H.

5.5.5.1 Scalability Comparison

Figure 5.11 shows the scalability comparison between Lancelot and existing GPU DBMSes when running SSB on GPU3.8 instance (V100 GPUs). To avoid GPU underutilization, we conducted a weak scaling experiment — we increase the number of GPUs as the data size increases. Hence, we will use 1 GPU for $\mathsf{SF} = 40$, 2 GPUs for $\mathsf{SF} = 80$, 4 GPUs for $\mathsf{SF} = 160$, and 8 GPUs for $\mathsf{SF} = 320$.

The result shows that Lancelot outperforms both HeavyDB and Dask-cuDF in terms of throughput and scalability. The performance difference between Lancelot

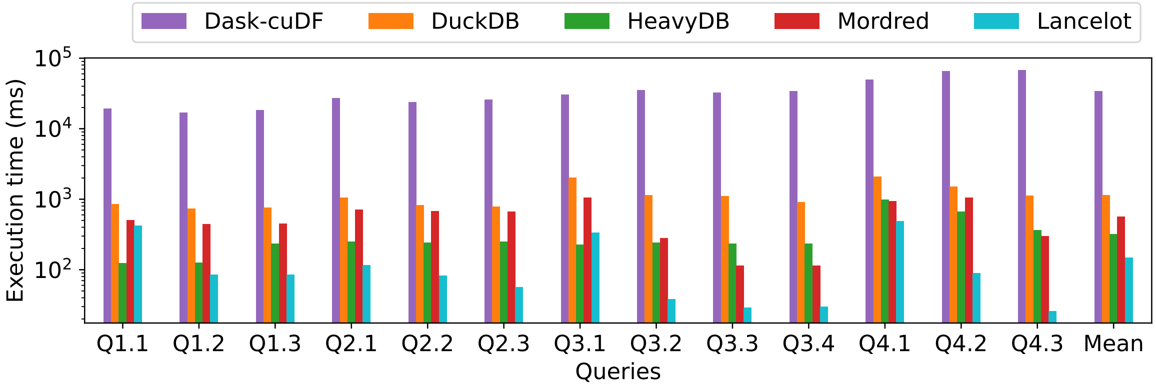


Figure 5.12: SSB Performance of Different GPU DBMS

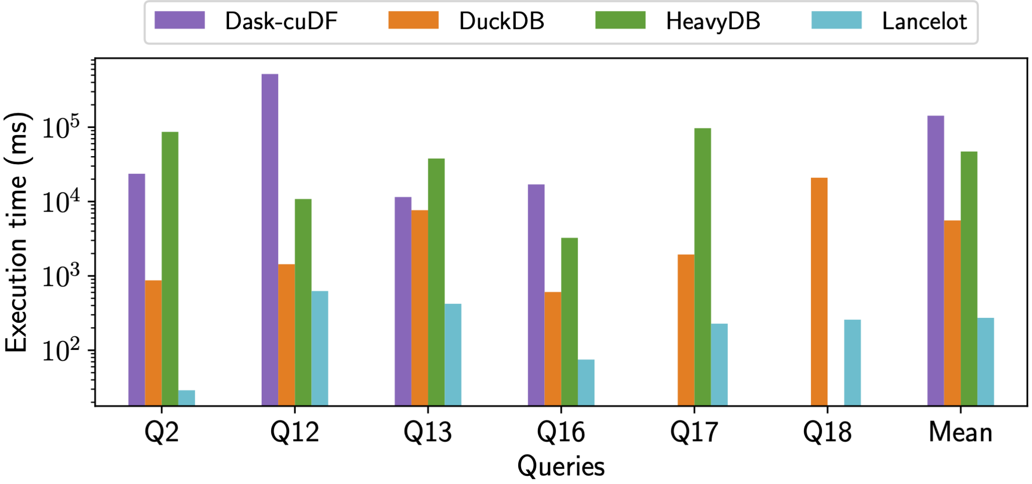


Figure 5.13: TPC-H Performance of Different GPU DBMS

and other GPU DBMSes boils down to several factors: (1) Lancelot is built on top of Crystal library which has shown to perform better compared to other GPU DBMSes [61]. (2) Lancelot adopts an intelligent data placement strategy discussed in Section 5.2 which is not employed by other GPU DBMSes. (3) Lancelot adopts various GPU database optimizations listed in Section 5.3.2 which are not adopted by the existing GPU DBMSes.

Figure 5.11 also shows that Lancelot throughput is not fully linear when increasing the number of GPUs. This is because as the data size increases, the hash table sizes also increase, reducing join throughput due to higher random access overhead. Since SSB queries involve multiple joins, this will have a significant impact on the query performance.

5.5.5.2 End-to-end Performance Comparison on SSB and TPC-H

Figure 5.12 and Figure 5.13 shows the query performance when running SSB and TPC-H respectively using 4 V100 GPUs on GPU3.8 instance. In this experiment, we use SF=320 which translates to about 50% of the total input data cached in the GPU memory.

Compared to `DuckDB`, `Lancelot` is 8× faster on SSB and 20× faster on TPC-H. This is due to the performance gap between CPUs and GPUs. Compared to `Mordred`, `Lancelot` is around 4× faster on SSB. This is because `Lancelot` can provide a larger cache size compared to `Mordred` by utilizing 4 GPUs. We do not compare to `Mordred` on TPC-H since it does not support TPC-H queries.

Compared to `Dask-cuDF`, `Lancelot` is 232× faster on SSB and 522× faster on TPC-H. `Dask-cuDF` suffers from out of memory execution which forces the intermediate result to spill to the CPU memory. This will result in back and forth data transfer between CPU and GPUs which degrades performance. On TPC-H, Q17 and Q18 failed since these engines cannot allocate enough memory space for the intermediate results.

Compared to `HeavyDB`, `Lancelot` is around 2× faster on SSB. `HeavyDB` is faster for Q1.1 but significantly slower for the other queries. This is because `Lancelot` decides to cache columns used in the other query sets (Q2.x, Q3.x, and Q4.x) since Q1.x are mostly just scan queries, which are not expensive when executed on the CPUs. On TPC-H, `Lancelot` is around 172× faster than `HeavyDB`. `HeavyDB` performance suffers heavily due to large intermediate results on complex TPC-H queries, resulting in excessive memory spilling and data transfer between CPUs and GPUs. `HeavyDB` also failed to execute Q18 due to out of memory execution.

5.5.5.3 End-to-end Performance on Various Hardware Platforms

Figure 5.14 show the query performance when running both SSB and TPC-H on three different instances: GPU3.8 (V100), GPU4.8 (A100), and g6.48xlarge (L4). Across all the instances, we use 4 GPUs and ran the experiment with `SF` = 320. This results in approximately 50% of the input data being cached in the V100 GPUs' memory, 75% in the L4 GPUs' memory, and 100% in the A100 GPUs' memory. The results for GPU3.8 instance are discussed in Section 5.5.5.2.

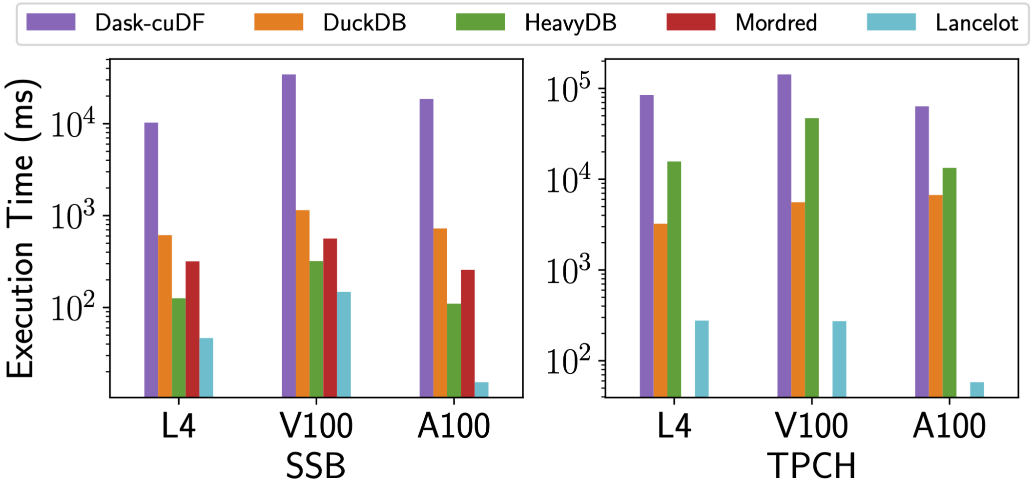


Figure 5.14: Performance on Different Compute Instances

Running on the g6.48xlarge instance, we see performance improvement across all systems compared to GPU3.8 instance. Even though L4 has lower memory bandwidth than V100 (330 GB/s vs 1TB/s), the larger GPU memory capacity (24 GB vs 16 GB) and significantly more CPU cores (192 cores vs 108 cores) made up for the difference. Despite slightly larger GPU memory capacity, query failures still occur for HeavyDB (Q18) and Dask-cuDF (Q17 and Q18). On this instance, Lancelot outperforms the the existing systems by at least 3× on SSB and 12× on TPC-H.

Running on the GPU4.8 instance, we observe performance improvement across all GPU DBMSes compared to g6.48xlarge and GPU3.8 instances since A100 has the largest GPU memory capacity (40 GB) and the highest memory bandwidth (1.5TB/s). Nevertheless, query failures still occur on Q18 for HeavyDB. DuckDB runs slower on this instance compared to g6.48xlarge instance due to fewer CPU cores (128 cores vs 192 cores), Overall, Lancelot still outperforms the existing systems by at least 7× on SSB and 115× on TPC-H.

5.6 Related Work

There have been several previous systems that attempted to leverage multiple GPUs for query execution.

A subset of existing works focused on accelerating individual database operators with multiple GPUs. Paul et al. [101] and Gao et al. [73] introduced a hash join implementation on multiple GPUs. Rui et al. [106] proposed a nested loop, sort-merge and hybrid joins implementation for multi-GPU. Maltenberger and Ilic et al. [114] showcased the advantage of multi-GPU sorting with fast interconnects. All these operator implementations are orthogonal to our work and can be incorporated to Lancelot.

There have also been commercial systems which support multiple GPUs. HeavyDB is an open-source, commercial GPU-accelerated DBMS. HeavyDB caches the most recently used data and partitions it across multiple GPUs. HeavyDB does not have a heterogeneous query execution strategy. When executing data larger than aggregated GPU memory, it will either (1) fallback to query execution on CPUs or (2) execute the query in multiple stages on GPUs.

PG-Storm is a GPU-accelerated PostgreSQL extension. Its core feature is GPUDirect-SQL, which enables reading data directly from NVMe to the GPUs. We do not compare against PG-Storm since the multi-GPU feature is not open-sourced. PG-Storm does not have data placement or hybrid query execution strategy.

cuDF is a GPU-accelerated DataFrame Library from NVIDIA. cuDF can support multiple-GPUs by leveraging Dask [11], hence the name Dask-cuDF. Similar to PG-Storm, Dask-cuDF does not have data placement and a hybrid query execution strategy.

BlazingSQL [6] is a GPU accelerated SQL engine built on top of the RAPIDS [28] ecosystem. Both cuDF and BlazingSQL[6] use the same internal execution engine but provide different API (pandas vs SQL). We compare against Dask-cuDF since it remains actively developed, unlike BlazingSQL, which has been inactive since 2021.

HetExchange [62, 63] is a query execution framework which encapsulates heterogeneous parallelism in CPUs and GPUs through redesigning the classical *Exchange* operator. It supports just-in-time code generation and hybrid CPU and multi-GPU query execution. HetExchange, however, does not address the data placement between CPU and multi-GPU systems and lacks an optimizer component to generate the physical query plan based on the data location.

Finally, HERO [87] proposed an operator placement strategy for CPU and multi-GPU systems that can be completely independent on cardinality estimation of the intermediate result. However, unlike Lancelot, this work only focuses on operator placement and does not address the challenges in other aspects of database design such as data placement and query execution.

5.7 Conclusion

This work advances the state-of-the-art of GPU DBMS by showing how to leverage both (1) hybrid CPU-GPU, and (2) multiple GPUs DBMS at the same time. To simplify the design space, we introduce the *unified multi-GPU abstraction* which will treat multi-GPU as a single large GPU. We then optimize hybrid CPU and multi-GPU DBMS in two aspects: (1) data placement and (2) query execution. For data placement, we introduce the *cache-aware replication policy* that takes into account the cost of shuffle when replicating data and could coordinate both caching and replication decisions for best performance. For query execution, we extend the existing CPU-GPU query execution strategy with distributed query processing techniques to support multiple GPUs. We then integrate both solutions in Lancelot, our hybrid CPU and multi-GPU DBMS. Our evaluation shows that our *cache-aware replication policy* outperforms other policies by up to 2.5× and Lancelot outperforms existing GPU DBMSes by at least 2× on SSB and 12× on TPC-H.

Part II

Towards Practical Data Analytics on GPUs

Chapter 6

User-Defined Aggregate Functions (UDAF) on GPUs

One of the most challenging operations to accelerate in GPU databases is the user-defined aggregate function (UDAF) [36, 38, 37, 64]. A UDAF is a user-defined routine that processes multiple input rows and returns a per-group aggregated result. It extends the set of built-in aggregate functions (e.g., SUM(), AVG(), MAX(), etc.), allowing users to define custom aggregation logic. Because the function is user-defined and often opaque to the system, it is especially difficult to parallelize efficiently on architectures like GPUs. In this chapter, we present a framework to accelerate UDAFs on GPUs to make GPU databases more practical and expressive. This work also lays the foundation for supporting general user-defined functions (UDFs) on GPUs in the future.

Our work builds on RAPIDS cuDF [9], an open-source library developed by NVIDIA to accelerate dataframe operation on GPUs. cuDF can serve as a drop-in replacement for Pandas, making it easy for users to switch between the two libraries. Previous results have shown that porting join and groupby from Pandas to cuDF can lead to around 138× speedup [10]. cuDF popularity has increased massively over the years and has reached more than 100k downloads per month in 2021 [88]. In cuDF and Pandas, UDAF is exposed to the user via the `groupby.apply()` API (see Listing 6.1).

In cuDF v22.12 and earlier, the UDAF execution suffers from a major performance limitation. Specifically, the UDAF execution in cuDF is slow for workloads with large group counts due to an implementation that iteratively launches a CUDA kernel for each group. We refer to this existing approach as `cuDF(Iterative)`. Our experiment shows that `cuDF(Iterative)` is slower than single-threaded CPU execution in Pandas for inputs with $> 10\text{k}$ groups.

In this work, we aim to tackle this issue by introducing a UDF execution framework which innovates in the following aspects:

Block-wide execution model. The previous implementation in cuDF resulted in suboptimal performance for a large number of groups due to an excessive number of kernel launches. *Block-wide execution model* [110, 1] allows us to tackle this issue by mapping each threadblock on a GPU to operate on a separate group and having each threadblock collectively call the CUDA device block-wide function to execute different operators in the UDAF. This approach will allow us to pipeline the whole UDAF execution in a single kernel, parallelize the execution of multiple groups on a GPU, and scale the performance for large number of groups.

JIT Compilation. Since UDAF execution is user-defined, the nature of the kernel would not be known prior to execution. This raises a need for just-in-time compilation [99] to generate the kernel at runtime. However, existing Python-based runtime compilers [18, 27, 2] do not recognize dataframe-like objects on GPUs. Therefore, in this work, we develop a Numba-based JIT compilation framework which can take dataframe-like objects as inputs and map each threadblock to each group in a dataframe following *the block-wide execution model* as it translates the UDF. Even though this approach has a tradeoff in JIT compilation time, the overhead is amortized by generating more efficient kernels. We also introduce a mechanism to reuse the compiled kernel to reduce the overhead. We will refer to this approach as `cuDF(JIT)`.

Both `cuDF(Iterative)` and `cuDF(JIT)` have their own advantages. In Section 6.3.2, we will show how `cuDF(Iterative)` is typically good for small group counts and `cuDF(JIT)` is good for large group counts. Therefore, to get the best performance out of both approaches, we add logic in our framework to let cuDF automatically dispatch between the two methods based on the group counts. We will refer to this approach as `cuDF(Auto)`. *As of now, our framework has been fully integrated and released in cuDF version 23.02.*

Contributions.. This work makes the following contributions:

- We showed how to accelerate UDAF execution on GPUs using *block-wide execution model*, which enables us to pipeline the whole UDAF execution into a single kernel.

- We introduced a Numba-based JIT compilation framework (`cuDF(JIT)`), which recognizes dataframe-like object and enable us to compile the UDAF kernel at runtime.
- We introduced `cuDF(Auto)` engine which can automatically dispatch different UDAF execution strategies based on the total number of groups.
- We conducted a detailed evaluation and showed how our framework can speedup the UDAF execution by 3600× against Pandas and 8000× against the existing approach in cuDF.

Organization. The rest of the chapter is organized as follows: We discuss the background and related work in Section 6.1. Section 6.2 describes the optimizations we introduced to accelerate the UDAF execution on GPUs. Section 6.3 evaluates the performance of our approach and Section 6.4 concludes the chapter.

6.1 Background and Related Work

6.1.1 Just-in-time (JIT) Compilation on GPUs

Just-in-time (JIT) compilation [99] is a way of executing computer code that involves compilation during execution of a program rather than before execution. Previous works have attempted to use JIT compilation for query execution on GPUs. [124, 71, 72, 100, 62, 55].

KernelWeaver [124] and HorseQC [71] show how we can use JIT compilation to generate the pipelined kernel by fusing multiple database operators. DogQC [72] improves HorseQC for both string processing (expansion divergence) and selective operation (filter divergence). Hawk [55] introduces a framework which could generate a device-aware code for both CPU and GPU. HetExchange [62] uses JIT compilation in its framework to encapsulate heterogeneous parallelism in hybrid CPU/GPU system. Finally, Pyper [100] further improves DogQC by introducing (1) block-level shuffle operator to handle data divergence and (2) segment operator to break the pipelined kernel into two smaller pipelines. However, none of these works support JIT compilation for user-defined function (UDF) on GPUs which will be the main focus of this work.

```
def midrange(df):
   return (df["val"].max() - df["val"].min())/2
out = df.groupby(["key"]).apply(midrange)
```

Listing 6.1: Example of UDAF API in Python

6.1.2 RAPIDS cuDF Library

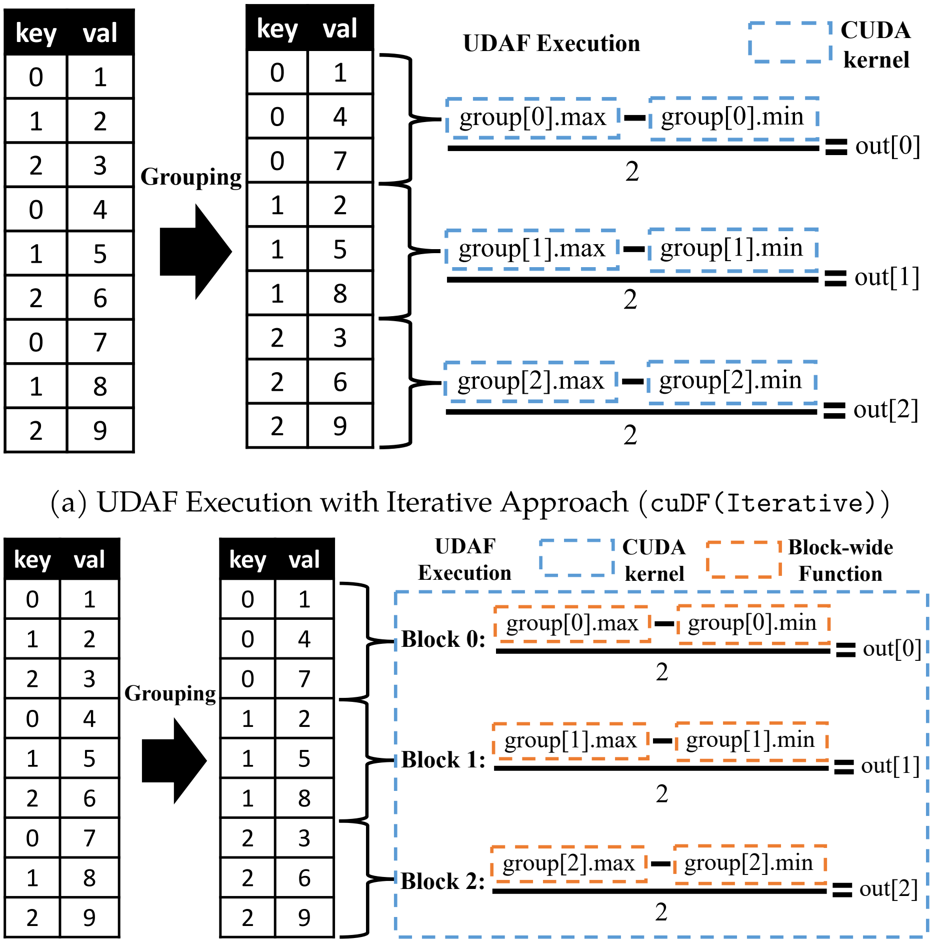
cuDF [9] is a GPU DataFrame library developed by NVIDIA for loading, processing, and manipulating data. It provides a Pandas-like API such that data scientists can accelerate their workflow without prior knowledge of CUDA programming. Previous results have shown that porting join and groupby from Pandas to cuDF can lead to around 138× speedup [10]. cuDF popularity has increased massively over the years and has reached more than 100k downloads per month in 2021 [88]. In this work, we will discuss how we can significantly accelerate one of the highly-used dataframe operation in cuDF (UDAF execution) through a series of optimizations.

6.2 Accelerating UDAF on GPUs

In this section, we will talk about how we can accelerate UDAF execution on GPUs. Specifically, we will talk about the default UDAF execution strategy in cuDF in Section 6.2.1 (`cuDF(Iterative)`). Then we will introduce `cuDF(JIT)` engine which further accelerates UDAF execution via *block-wide execution model* (Section 6.2.2) and JIT compilation (Section 6.2.3). Finally, in Section 6.2.4, we introduce `cuDF(Auto)` which automatically switch UDAF execution strategies between `cuDF(Iterative)` and `cuDF(JIT)`.

6.2.1 User-Defined Aggregate Function in cuDF

One of the frequently-used operation in dataframe manipulation is user-defined aggregate functions (UDAF) [36, 38, 37, 64]. UDAFs are user-programmable routines that act on multiple rows at once and return per-group aggregated value as a result. UDAF works in a similar fashion as pre-defined aggregate function but with the freedom for users to define the routines. Therefore, UDAF can contain arbitrary logics such as conditional statements, loop, and complex mathematics operations (i.e. sine, cosine).



key	val
0	1
1	2
2	3
0	4
1	5
2	6
0	7
1	8
2	9

key	val
0	1
0	4
0	7
1	2
1	5
1	8
2	3
2	6
2	9

Grouping
UDAF Execution
CUDA kernel
group[0].max − group[0].min
2
= out[0]
group[1].max − group[1].min
2
= out[1]
group[2].max − group[2].min
2
= out[2]

(a) UDAF Execution with Iterative Approach (`cuDF(Iterative)`)

key	val
0	1
1	2
2	3
0	4
1	5
2	6
0	7
1	8
2	9

key	val
0	1
0	4
0	7
1	2
1	5
1	8
2	3
2	6
2	9

(b) UDAF Execution with Block-wide Execution Model (`cuDF(JIT)`)

Figure 6.1: Comparison Between Different UDAF Execution Strategies

cuDF can accelerate user-defined aggregate function (UDAF) with GPUs. Similar to Pandas, the usage of UDAF in cuDF is exposed via `groupby.apply()` API. Listing 6.1 shows how to calculate the *midrange* of each group via the `groupby.apply()` API in Python.

Figure 6.1a shows the default approach in cuDF to accelerate the UDAF on GPUs. First, we will perform a sort-based algorithm to group the data. After the data is sorted by group, we will iteratively loop over each group and launch a separate CUDA kernel for each operator in the UDAF to calculate the aggregation. In Figure 6.1a, for each group, we will launch two kernels to find the maximum and the minimum of the group. The two CUDA kernels will return a scalar value and the CPU will subtract the two scalar values as a regular Python operation. The same process will be repeated for each group, resulting in a total of six CUDA kernels to be launched sequentially. We refer to this approach as `cuDF(Iterative)`.

The advantage of this approach is it provides the flexibility to express even the most complicated UDAF and it performs well for a small number of groups. Nevertheless, this approach does not scale well as the group count increases. Specifically, since separate CUDA kernels will be launched for each operator in each group, an excessive number of kernels will be launched for a large number of groups. This would hurt the performance, making it slower than the single-threaded Pandas when the group count exceeds several thousand. In the next two sections, we will show how we can tackle this issue via the two optimizations we introduce in this chapter.

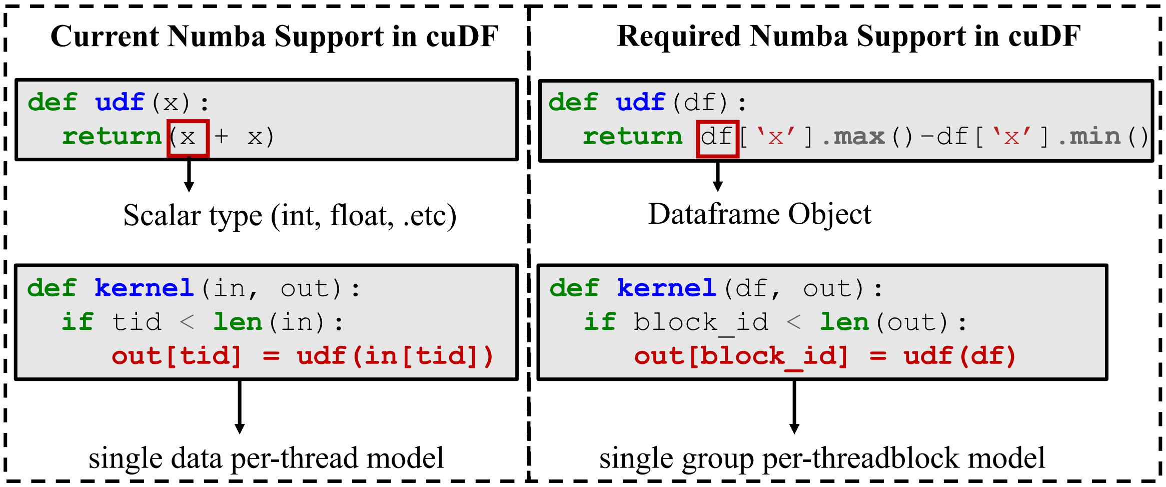
6.2.2 Block-wide Execution Model

Threadblock-level functions, or more commonly known as block-wide functions are CUDA device functions in which threads in the same threadblock collectively perform a certain operation (i.e. reduction, scan, sorting, etc.). A GPU execution model which adopts a threadblock-level function is usually called the *block-wide execution model*, where instead of viewing each thread as an independent execution unit, it views a thread block as the basic execution unit.

Block-wide function primitives were originally introduced in CUB [1] to perform block-level reduction, scan, sorting, etc. More recently, Crystal [110] presented a library of CUDA device block-wide functions to perform data analytics operations such as probe, filter, groupby, etc. The main idea of using *block-wide execution model* in Crystal is to pipeline multiple operators in a single kernel by storing the intermediate result in the shared memory to avoid multiple round-trips to the global memory.

In this work, we will borrow the idea of *block-wide execution model* from Crystal and CUB to accelerate the UDAF on GPUs. Instead of launching separate kernels for each operator in each group, we will now map each threadblock to each group and encapsulate the whole UDAF execution in a single kernel. Each threadblock will then collectively call the CUDA device block-wide function to execute different operators in the UDAF. We believe mapping each threadblock into each group is a reasonable approach since in most workloads, group sizes are comparable to the threadblock size (see Section 6.3.5 for further studies on what happens if they are not).

Figure 6.1b shows an example of how we execute the midrange UDAF with *block-wide execution* in cuDF. We will first sort the data based on the group. After each group is sorted, we will launch a single kernel with 3 threadblocks to execute the


Current Numba Support in cuDF
Required Numba Support in cuDF
Scalar type (int, float, .etc)
Dataframe Object
single data per-thread model
single group per-threadblock model

```
def udf(x):
  return (x + x)
```

```
def kernel(in, out):
  if tid < len(in):
     out[tid] = udf(in[tid])
```

```
def udf(df):
  return df['x'].max()-df['x'].min()
```

```
def kernel(df, out):
  if block_id < len(out):
     out[block_id] = udf(df)
```

Figure 6.2: The Gap Between the Current Numba Support and the Required Numba Support in cuDF.

Functions	Description
`count`	Count the number of elements in the group
`sum`	Calculate the sum the whole elements in the group
`min`	Find the minimum elements in the group
`max`	Find the maximum elements in the group
`mean`	Calculate the average of the group
`var`	Calculate the variance of the group
`std`	Calculate the standard deviation of the group
`idxmin`	Find the index of the smallest element in the group
`idxmax`	Find the index of the largest element in the group

Table 6.1: Currently Supported Functions in `cuDF(JIT)`

entire UDAF. Each threadblock will run in parallel and collectively call the respective block-wide function (i.e., `BlockMax` and `BlockMin`) to calculate the maximum and the minimum of each group.

There are several advantages to this approach compared to the `cuDF(Iterative)` approach (Section 6.2.1). First, we can pipeline the whole UDAF execution in a single kernel and therefore it will not suffer from excessive kernel launches. Second, we can parallelize the execution of multiple groups since each threadblock can run in parallel on GPUs. Third, this approach scales well with a large number of groups. Our experiment (Section 6.3.2) shows that `cuDF(JIT)` outperform `cuDF(Iterative)` for groups > 8.

6.2.3 Just-in-time Compilation with Numba

To accelerate UDAF with block-wide function, it is necessary to encapsulate the entire UDAF operation in a single kernel. Since it is user-defined, the nature of the kernel would not be known prior to execution. As it is not feasible to implement a custom kernel for each UDAF, this raises the need for just-in-time (JIT) compilation [99] to generate the kernel at runtime.

We use Numba [18] for our JIT compilation framework. Numba is a Python-based compilation framework that supports both multicore CPU and CUDA GPUs. Numba uses type information from the kernel arguments at runtime to analyze its way through a chunk of Python code and replace statements with equivalent GPU machine code. Numba exposes CUDA's SIMT programming model directly where kernels can be written to map data to threads in any fashion. In cuDF, Numba by default generates kernels which adopts "*single data per thread model*", and therefore each GPU thread will operate on a single element in the input array, execute the UDAF, and write the results to the output array.

For our case, however, this model would not be sufficient for two major reasons. First, the input to our UDAF will not be a primitive data type, but a data frame-like object which would not be recognized by Numba. Second, instead of using the "*single data per thread model*", our execution adopts a "*single threadblock per group model*" where each threadblock on a GPU will operate on a single group via block-wide level functions as shown in Figure 6.1b. This GPU execution model is by default not used by cuDF. Figure 6.2 shows the difference between the current Numba support in cuDF and the required support for our framework.

To address these issues, we implement Numba extensions to support (1) dataframe-like objects to represent a group of data and (2) mapping operations on dataframe objects to *block-wide execution model* where each threadblock will operate on each group. This feature is built on existing Numba extensions in cuDF that leverage the Numba APIs for adding new types and functions [12]. The properties of a dataframe object are an array of data and a set of functions which can operate on this data (`max()`, `min()`, `mean()`, etc.). The implementation of these functions will be linked to an externally implemented C++ block-wide functions library but can also reuse any existing block-wide functions library such as those in CUB [1]. Each device function implements a single operation, and Numba can map the operation to each group as it translates the UDAF to enable *block-wide execution model*.

To further extend our framework with new functions/operators, we can simply add new block-wide functions to this library and expose them to Numba. This would allow our framework to support aggregation functions and binary operations that are supported in Pandas as long as they can be expressed as block-wide functions. Our initial development includes 9 functions which are shown in table 6.1. For our future development, we plan on adding further sets of functions and operators to support more expressive UDAF. We refer to this approach as `cuDF(JIT)`.

`cuDF(JIT)` compilation overhead can be up to 0.3-0.4s depending on the complexity of the UDAF (see Section 6.3.4). Kernel execution on GPUs, on the other hand, is only in the range of ms. To eliminate this overhead, we add further optimization in `cuDF(JIT)` to cache and reuse an already compiled kernel. We will reuse the previously generated kernel if (1) the same UDAF has been previously executed, and (2) the input data types are the same as the input data types of the previously executed UDAF. As a result, we can reuse the compiled kernel for the same UDAF even for different input columns as long as they have the same input data types. Currently, our framework has been fully integrated and released in cuDF v23.02.

6.2.4 Choice between Iterative vs JIT

Currently, users can choose between the two engines for UDAF, `cuDF(Iterative)` and `cuDF(JIT)`, by passing the "`engine=cudf`" or "`engine=jit`" as the second argument to `groupby.apply()` API. Typically, `cuDF(Iterative)` is not suitable for large group counts since it will launch a separate GPU kernel for each operator in each group which could result in significant kernel launch overhead. On the other hand, the single threadblock per group model in `cuDF(JIT)` is not suitable for small group counts since launching a few threadblock could result in poor GPU utilization. Based on this observation, we provide a third option to let cuDF dynamically choose the engine based on the number of groups. By passing the "`engine=auto`" argument to `groupby.apply()` API, we will now run `cuDF(JIT)` only when $ngroups > N$ where N is a tunable parameter. We perform tuning by running the t-test formula with 10M rows of `uint64_t` data and find that setting $N = 8$ delivers the best performance overall. We refer to this approach as `cuDF(Auto)`.

6.3 Evaluation

In this section, we evaluate the performance of our proposed approach to understand (1) the impact of *block-wide execution model* and JIT Compilation, (2) the timing breakdown of different approaches, (3) the impact of caching the generated code, and (4) the impact of various group distributions.

The rest of the section is organized as follows: we discuss the experiment setup in Section 6.3.1. In Section 6.3.2 we evaluate the performance of different approaches on uniform group sizes. In Section 6.3.3, we evaluate the timing breakdown of different schemes. In Section 6.3.4, we show the speedup of caching the generated kernel. In Section 6.3.5 we evaluate the performance of different approaches on non-uniform group sizes.

6.3.1 Setup

Hardware configuration: For the experiments, we use an instance that has an NVIDIA V100 GPU which is connected to an Intel Xeon Platinum 8167M CPU via PCIe3 with 12.8GBps bidirectional bandwidth. The NVIDIA V100 GPU has 16 GB of HBM2 memory with 768 GBps read/write bandwidth. The Intel Xeon Platinum CPU has 12 virtual cores and 88 GB DRAM. The system is running on Ubuntu 20.04 and the GPU instance uses CUDA 11.5.

Benchmark: For our experiments, we generate a dataframe with random `uint64_t` values with 10 million rows. We will apply user-defined aggregate functions to run two widely-used statistics algorithms on each group:

$$\textit{Midrange} = \frac{max(x) - min(x)}{2} \qquad t - test\,\textit{formula} = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\delta_1^2}{N_1} + \frac{\delta_2^2}{N_2}}}$$

Measurement: For each experiment, we ensure that data is already loaded into the GPU memory before experiments start. This is because cuDF targets in-GPU memory workload and intends for data to remain resident on the device. Therefore, host-to-device data transfer will be excluded from the measurement. We run each experiment 10 times and report the average execution time.

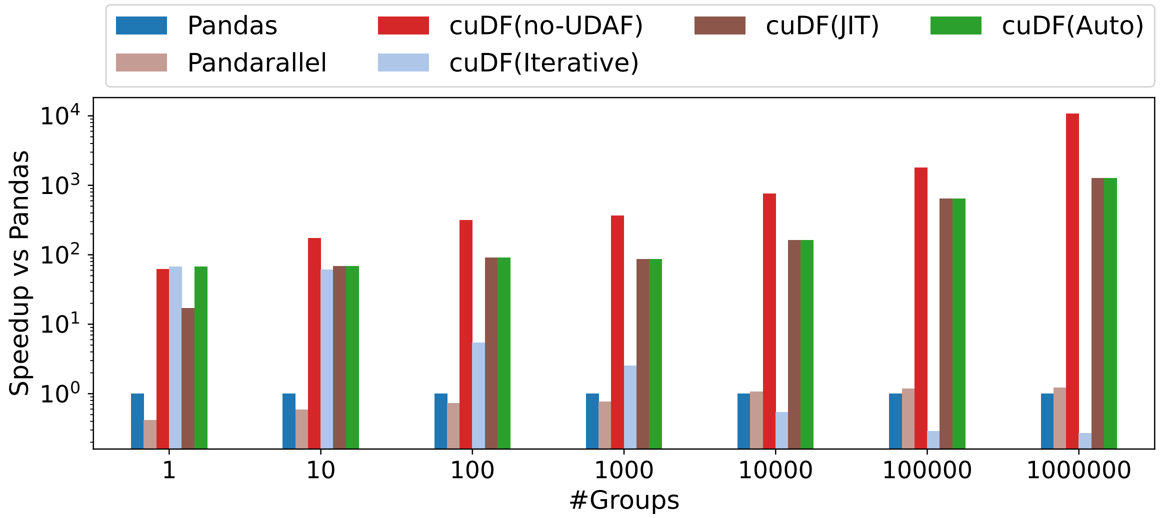


Figure 6.3: Midrange formula (uniform)

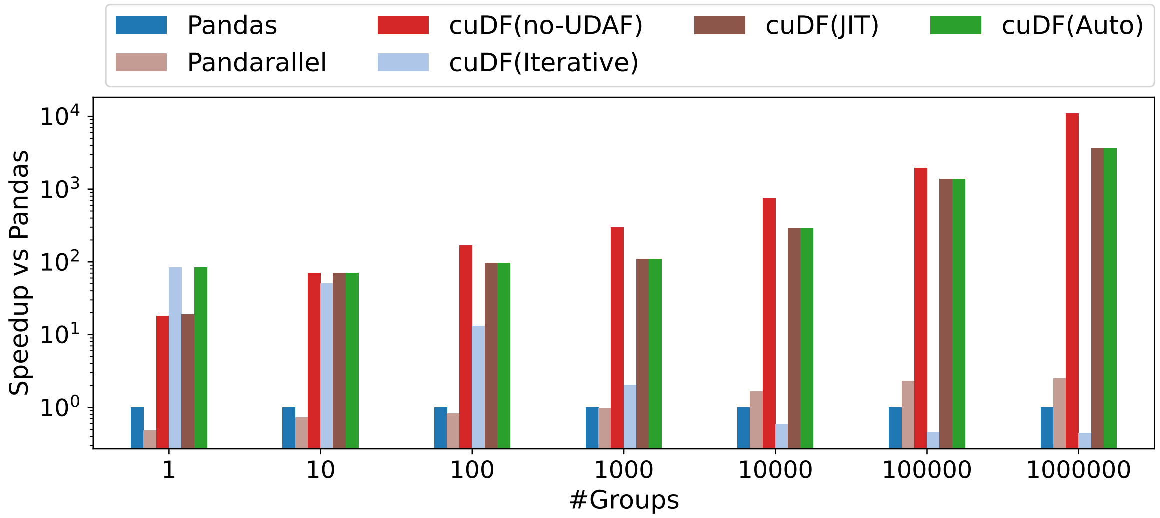


Figure 6.4: t-test formula (uniform)

6.3.2 Performance with Uniform Group Size

This subsection evaluates the performance of 6 different schemes with uniform group sizes:

- **`Pandas`:** Single-threaded Python Data Analysis Library.
- **`Pandarallel`:** Python library to parallelize Pandas operation on multicore

CPUs [25]. We can run this approach by running `groupby.parallel_apply()`.

- **`cuDF(Iterative)`:** Default approach for UDAF in cuDF explained in Section 6.2.1.
- **`cuDF(JIT)`:** Our proposed approach to accelerate UDAF with JIT compilation and *block-wide execution model*. The generated code is cached to eliminate compilation overhead in subsequent execution. We can run this approach by passing "`engine = jit`" as the second argument to `groupby.apply()`
- **`cuDF(Auto)`:** Our proposed approach to automatically switch between `cuDF(JIT)` and `cuDF(Iterative)` based on the number of groups. We can run this approach by passing "`engine = auto`" as the second argument to `groupby.apply()`
- **`cuDF(no-UDAF)`:** Calculate the result through manual aggregation followed by binary operation instead of using the UDAF API. This approach represents the best performance we would reasonably expect to compare against. However, this approach is not commonly used by the users as it is more natural and readable to write functions as row-wise (i.e. UDFs) rather than chaining vector ops.

For this comparison, we fixed the total dataset to 10M rows and sweep the total number of groups from 1 group (10M elements per group) to 1M groups (10 elements per group). Figure 6.3 and 6.4 show the speedup against Pandas for each approach.

Since Pandas is single-threaded, we also compare against the multi-core implementation of Pandas (Pandarallel [25]). Pandarallel could parallelize UDAF execution by assigning different groups on different CPU cores. Our experiment shows that Pandarallel starts performing better than Pandas when $ngroups > 1k$ (up to 2.5×). This is because when $ngroups < 1k$, Pandarallel does not have enough work to schedule for each CPU core to offset the coordination and synchronization overhead.

The performance of `cuDF(Iterative)` degrades significantly as we increase the total number of groups. For both formulas, this approach performs better than Pandas and Pandarallel for small group count but slower when $ngroups > 10k$. The reason behind this is this approach will launch an independent GPU kernel for each operation and for each group. This results in multiple round-trips through the GPU global memory and significant kernel launch overhead especially for a large number of groups.

`cuDF(no-UDAF)` has the best performance overall compared to the other approaches. In this approach, we do not have to pay the overhead of sorting the data since aggregation will be achieved through hash aggregation. This approach, however, requires users to calculate the results via manual aggregation and binary operation. This approach is not favoured by users since they would expect to execute the formula via the UDAF approach (`groupby.apply()`). Nevertheless, we recommend advanced users who are used to chaining vector ops to try this approach due to its overall performance.

`cuDF(JIT)` speedup improves significantly over Pandas as we increase the total number of groups. JIT compilation with *block-wide execution* enable us to execute multiple groups with a single kernel by mapping each threadblock to each group. This approach gets up to 1400× and 3600× speedup against Pandas and 1150× and 1440× speedup against Pandarallel for *midrange* and *t-test formula* respectively. Comparing against `cuDF(Iterative)`, `cuDF(JIT)` gets up to 5000× and 8000× speedup for large number of groups for both formulas respectively. For small number of groups (1 group and 10 groups), `cuDF(JIT)` does not perform as well since we only map a small number of threadblocks to operate on the whole datasets. This will cause GPU to be underutilized.

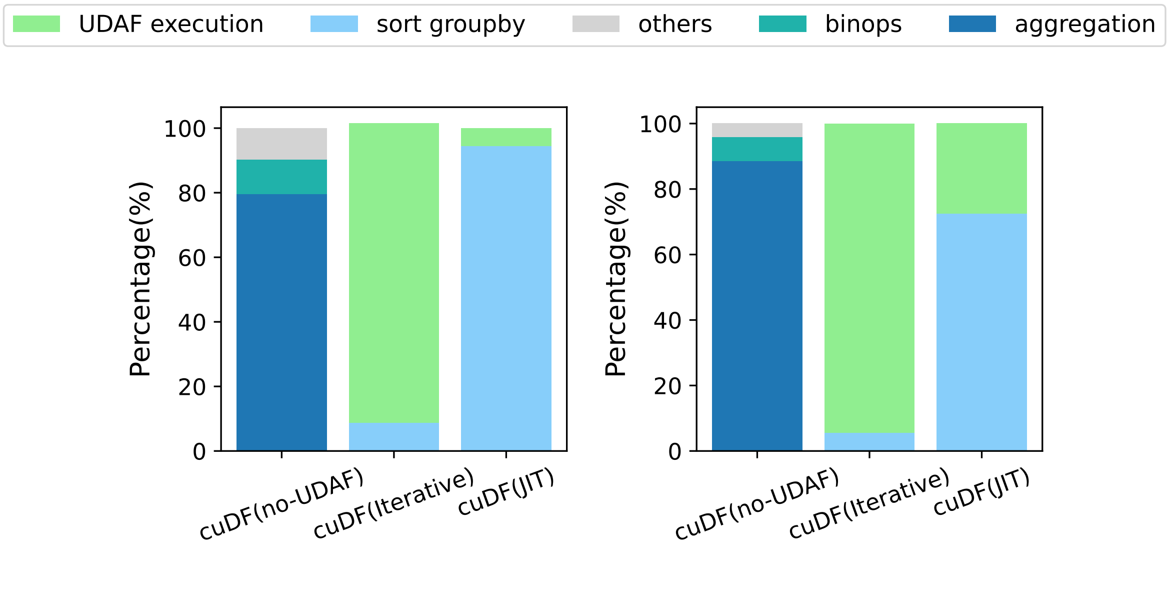
Comparing the results for *midrange* and *t-test formula* (Figure 6.3 and 6.4), `cuDF(JIT)` gained more speedup for the *t-test formula*. This is because the gain from pipelining operator into a single kernel would be more significant for more complicated UDAF.

Finally, `cuDF(Auto)` manages to get the best performance out of `cuDF(Iterative)` and `cuDF(JIT)` by applying JIT compilation and *block-wide execution* only for a large number of groups. Comparing against `cuDF(no-UDAF)`, this approach is still slower but with the ergonomic API that users expect from a Python UDAF.

6.3.3 Timing Breakdown

This subsection evaluates the timing breakdown between three designs: `cuDF(Iterative)`, `cuDF(JIT)`, and `cuDF(no-UDAF)`. We will use the same workload as Section 6.3.2 with a fixed number of groups (1k groups with 10k elements per group). Figure 6.5 shows the timing breakdown for both *midrange* and *t-test formula* respectively.

Comparing the three schemes, we can see that `cuDF(Iterative)` execution time is dominated by UDAF execution on the grouped data (92%). For `cuDF(JIT)`, the UDAF execution is no longer the bottleneck (27%) and the execution is now dominated



(a) Midrange formula

(b) t-test formula

Figure 6.5: Breakdown of Different Schemes ($n\mathrm{Groups} = 1000$)

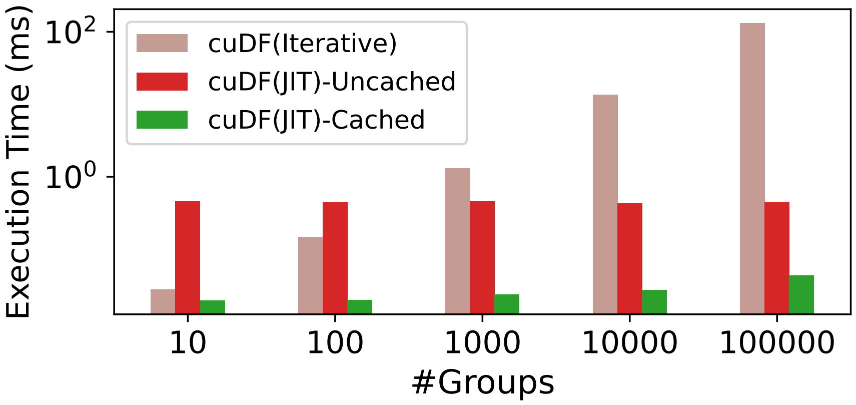


Figure 6.6: Performance Comparison Before and After Caching the Compiled Kernel.

by grouping and sorting the keys (73%). Our further experiment shows that for workload with pre-sorted group keys, the `cuDF(JIT)` approach can be 4-10× faster. This shows that the series of optimization that we introduced in this work manages to reduce the UDAF execution significantly. For `cuDF(no-UDAF)`, the execution time is dominated by aggregation (88.5%) and the rest is spent on reconstructing the UDAF via binary operation (11.5%). Unlike the other schemes, this approach does not require the keys to be sorted since the aggregated results are obtained via hash-based aggregation.

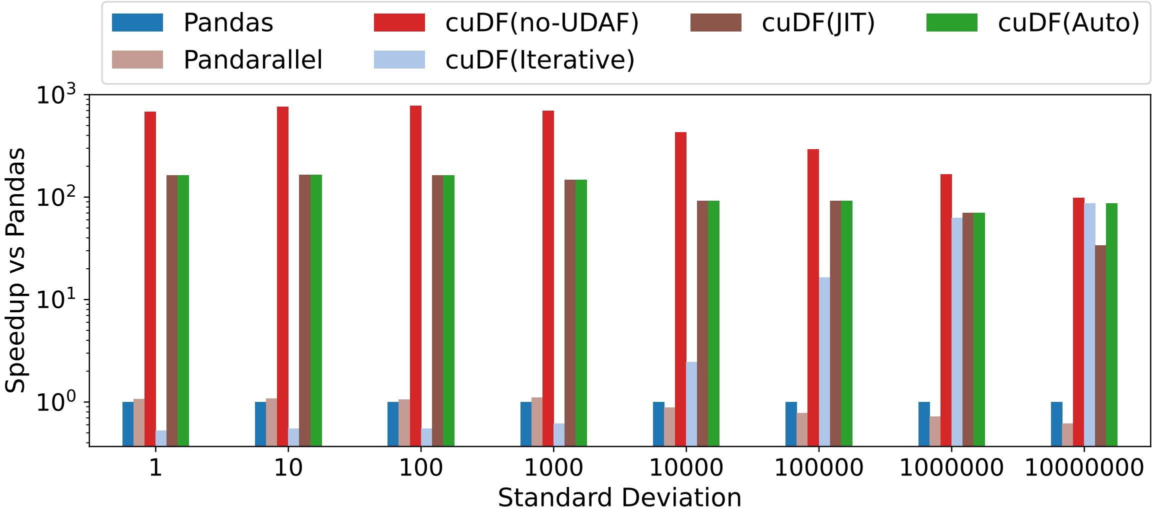


Figure 6.7: Midrange formula (non-uniform)

6.3.4 Impact of Caching the Generated Kernels

This subsection evaluates the impact of caching the generated kernel on `cuDF(JIT)` performance. We use the same workload as Section 6.3.2. Figure 6.6 shows the impact of caching the generated kernel for the *t-test formula*. Our experiment shows that caching the generated kernel could improve the performance by up to 22×. Caching the kernel could eliminate the compilation overhead which is much more expensive compared to the kernel execution on GPUs.

Figure 6.6 also shows the comparison between `cuDF(Iterative)` and `cuDF(JIT)` if we take into account the JIT compilation overhead. We can see that `cuDF(JIT)` still outperforms `cuDF(Iterative)` even when the kernel is uncached for group size larger than hundreds. This shows that the JIT compilation overhead in `cuDF(JIT)` is amortized by far more efficient kernel compared to `cuDF(Iterative)`. Furthermore, we would expect this overhead to be negligible during the execution of any production-scale (i.e. large/long-running) data.

6.3.5 Performance with Non-uniform Group Size

This subsection evaluates the performance of 6 different schemes under non-uniform group sizes. We will compare the same set of schemes as Section 6.3.2. For this comparison, we fix the dataset size to 10M rows and vary the group size following a

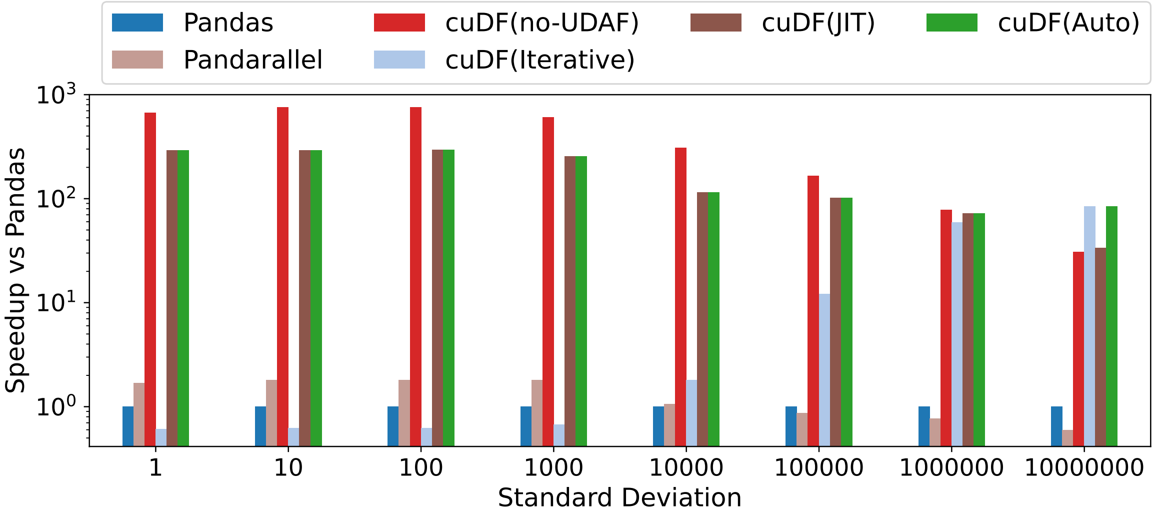


Figure 6.8: t-test formula (non-uniform)

normal distribution. We set the mean (μ) of the group size to 1000 and sweep the standard deviation (δ) from 1 to 10M. As a result, when the standard deviation is low, we will have a higher number of groups (around 10k) with fairly uniform sizes and as the standard deviation increases, we will get a smaller number of groups with imbalance sizes. Figure 6.7 and 6.8 show the speedup of each approach against Pandas and Pandarallel for both *midrange* and *t-test formula*.

`Pandarallel` performs better than Pandas on low standard deviation but slower on high standard deviation. The reason behind this is that our workload tends to have a higher number of group count (around 10k) for low standard deviation which results in better multi-core utilization.

`cuDF(Iterative)` performs worse on low standard deviation. This is because low standard deviation leads to a higher number of groups which will cause performance degradation for this approach (see Section 6.3.2 for more details). For high standard deviation, this approach performs well since the group count will be less (only 3 groups for $\delta = 10M$).

`cuDF(no-UDAF)` still has the best performance overall compared to other approaches. The performance of `cuDF(no-UDAF)` slightly degrades as the standard deviation increases since hash-based aggregation becomes less effective with highly diverse group sizes.

`cuDF(JIT)` performs better for low standard deviation. For high standard deviation, we will have a small number of groups with highly diverse groups sizes.

Since we map each threadblock to a different group, the load between each threadblock will be imbalanced. For example, one threadblock might operate on group with 1M elements but another threadblock might operate on group with 1 element. This phenomenon would not apply for low standard deviation as the size will be relatively constant all across the groups. As a result, `cuDF(JIT)` can get up to 53× speedup over `cuDF(Iterative)` for low standard deviation but is 2.5× slower than `cuDF(Iterative)` for high standard deviation.

Comparing the results between the *midrange* and *t-test formula* (Figure 6.7 and 6.8), `cuDF(JIT)` performance is slightly more sensitive to non-uniform group size for the *t-test formula*. This is because *t-test formula* involves more data read and operation which causes a more significant imbalance workload between threadblocks.

Finally, `cuDF(Auto)` manages to get the best performance out of `cuDF(JIT)` and `cuDF(Iterative)`. It performs well across standard deviation since it selectively applies JIT compilation only when the standard deviation is low (large group count).

6.4 Conclusion

This work advances the state-of-the-art data analytics on GPUs by introducing (1) *block-wide execution model* (Section 6.2.2) and (2) *JIT compilation* (Section 6.2.3) to accelerate UDAF execution on GPUs. *Block-wide execution model* allows us to pipeline the entire UDAF execution in a single kernel by mapping each threadblock to operate on a separate group. JIT compilation allows us to generate the GPU machine code at runtime based on the UDAF while adopting the *block-wide execution model*. Our evaluation shows that our scheme manages to gain up to 3600× speedup against UDAF execution in Pandas and 8000× speedup against the existing approach on GPUs for a large number of groups. Our framework has been fully integrated and released in NVIDIA RAPIDS cuDF version 23.02.

Chapter 7

Sirius: A Drop-In GPU-Native SQL Engine

Another major challenge in adopting GPU databases lies in the migration cost from legacy CPU-based systems to GPU databases as well as the engineering cost required to build and optimized GPU SQL engine. These challenges have slowed the adoption of GPU databases in practice. In this chapter, we introduce *Sirius* [31], an open-source GPU SQL engine designed to address these challenges by delivering drop-in acceleration across diverse data systems.

Sirius is enabled by the rise of *composable data systems* [103], where different system components in a database are treated as modular building blocks and connected through open standards such as Substrait [33], Apache Arrow, and Apache Parquet. Rather than implementing a standalone database from scratch, Sirius embraces this trend by reusing existing components—such as SQL parsers and optimizers from host databases—while accelerating query execution on GPUs via the Substrait [33] intermediate query plan representation. This design enables Sirius to operate as an accelerator for a wide range of data systems. For instance, plugging Sirius into DuckDB requires no modification to the DuckDB codebase or the user-facing interface: instead of executing queries on the CPU, DuckDB simply hands off the Substrait plan to Sirius for GPU-native execution.

Internally, Sirius is built on top of matured GPU libraries such as `libcudf` [9], `RMM` [29], and `NCCL` [17], reusing optimized implementations of core relational operators like joins, filters, aggregations, and data shuffle. This modular approach minimizes engineering effort while allowing Sirius to deliver end-to-end GPU acceleration. Importantly, Sirius's design allows developers to easily switch between default implementations in `libcudf` and custom CUDA kernels, providing flexibility for performance tuning and experimentation. Our detailed evaluation on TPC-H benchmark shows that Sirius can deliver 8.2× speedup as a drop-in accelerator for DuckDB [105] (single-node) and up to 12.5× speedup as a drop-in accelerator for Apache Doris [3] (distributed).

Organizations. The rest of the chapter is organized as follows: We discuss the recent hardware and software trend in Section 7.1. Section 7.2 describes Sirius' design, implementation, and future roadmap. Section 7.3 evaluates the performance of Sirius and Section 7.4 concludes the chapter.

Acknowledgment. Sirius integration with Apache Doris and the exchange service layer described in Section 7.2.2.4 were implemented by Yifei Yang, whose work I gratefully acknowledge.

7.1 Motivation

As discussed in Section 1.1, GPU hardware is advancing rapidly along multiple dimensions. On the memory capacity front, GPU capacity has nearly doubled with each new generation—from 32 GB in Volta, to 80 GB in Ampere, 192 GB in Hopper, and most recently, 288 GB in Blackwell. In addition, modern interconnects such as PCIe Gen6, NVLink-C2C [22], and GPU Direct [14] have significantly reduced data transfer overhead between the GPU and other system components. These improvements enable GPUs to efficiently process data beyond on-device memory, making it possible to run terabyte-scale analytics on a single GPU, and even larger workloads in distributed settings. At the same time, GPUs are becoming increasingly affordable and accessible—particularly older-generation models, which are already well-suited for many data analytics workloads.

Despite the positive hardware trajectory, adopting GPU-accelerated database systems remains challenging due to the high engineering cost of developing a GPU-optimized software stack from scratch, as well as the switching cost associated with migrating from the legacy database systems. Fortunately, recent developments in the software ecosystem have significantly lowered these barriers, driven by two key trends.

Composable Data Systems. Modern open-source data systems are embracing composability [103] (e.g. Velox [102]), where different system components can be independently developed and interoperate in a data system. A new database can reuse existing components such as the language frontend, query optimizer, or execution engine instead of reinventing the wheel. A crucial component in these systems is Substrait [33], a unified IR for physical and logical representations of query plans. The trend of composable data systems significantly lowers the barrier to adopt a GPU database—only the execution engine needs to be developed from the ground up tailored for GPU but other system components could be reused.

Maturing GPU Libraries. Open-source GPU libraries optimized for data processing have emerged and are rapidly reaching maturity in recent years. For example, libcudf [9] provides GPU-efficient implementation for relational operators such as joins, aggregations, and sorting. RMM [29] complements this by offering a high-performance GPU memory allocator, serving as a foundation for GPU-centric buffer managers. On the distributed side, the NCCL [17] provides high-speed communication between GPU devices over various interconnect, such as PCIe, NVLink, and Ethernet/RDMA.

Motivated by the hardware and software trends discussed above, we designed Sirius to be both GPU-native and composable. Sirius builds on open-source GPU data processing libraries such as `libcudf` and `RMM`, and adopts universal query plan formats like Substrait to integrate seamlessly with existing database systems.

7.2 Sirius Design and Architecture

Motivated by the hardware and software trends discussed above, we have designed Sirius, the first open-source GPU-native SQL engine that provides drop-in acceleration across a variety of existing data systems. In this section, we will discuss the design overview, system architecture, and the future roadmap of Sirius.

7.2.1 Design Overview

Sirius was designed with the following two key design principles:

GPU-Native Execution. Recent hardware trends show that the overhead of data movement between GPUs and other system components—especially CPUs—is diminishing. As the GPU memory barrier is breaking down and GPU performance

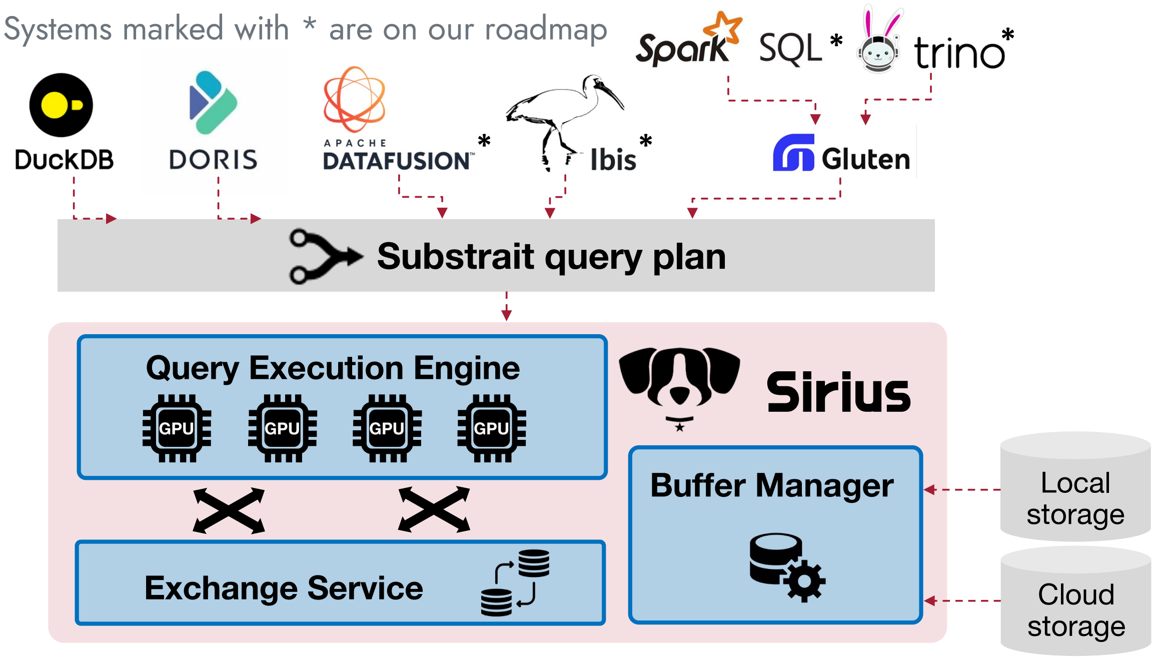


Figure 7.1: Sirius Architecture

continues improving, GPU-only query execution is becoming increasingly competitive with hybrid CPU–GPU query execution.

Taking this into account, we design Sirius as a ***GPU-native engine***: it treats GPUs as the primary execution engine, aiming to run the entire query plan—from scan to result—on the GPU. Sirius differs from systems that retrofit GPU acceleration onto traditional CPU-optimized engines while maintaining backward compatibility [32, 13], or hybrid systems that split execution between CPU and GPU [131, 85, 128, 62]. Instead, it builds its execution model, buffer management, and exchange mechanisms entirely around full GPU residency. The CPU serves only as a fallback path when certain features are not supported on GPUs. This architecture can reduce the design complexity compared to a retrofitted or heterogeneous design, thereby improving portability and performance.

We believe GPU-native design is a promising direction. History shows that when a major infrastructure shift occurs—such as the rise of the cloud—systems built natively for the new paradigm (e.g., Snowflake) ultimately surpass retrofits of prior-generation architectures. Similarly, as GPUs emerge as a central computational platform for data analytics, GPU-native systems like Sirius have the potential to outperform hybrid or retrofitted approaches.

Drop-In Acceleration. Modern software stacks are gradually moving toward composable architectures [103], where components such as query optimizers, storage format, and query execution engine are decoupled. Sirius embraces this modular design to significantly reduce the development cost and the barrier of adoption.

By consuming a universal query plan format (e.g., Substrait) and connects it with existing GPU-accelerated libraries (e.g., libcudf), Sirius can serve as a ***drop-in accelerator*** across a variety of data systems with minimal or no modification to the host systems. When using Sirius, the users can benefit from GPU acceleration without having to modify their user interface or migrate to a different DBMS, thereby lowering the barrier of adoption.

7.2.2 System Architecture

In this Section, we will discuss the system architecture of Sirius as shown in Figure 7.1:

7.2.2.1 Host Database Layer

This layer includes the query parser and optimizer of external data systems that Sirius accelerates. These host systems generate either query plans in a standardized format such as Substrait which Sirius can consume. Currently, Sirius supports DuckDB and Apache Doris as external hosts, and more will be supported in the future.

For distributed query execution, Sirius leverages the host database's coordinator to manage the control plane. This includes identifying active nodes via heartbeat, scheduling plan fragments across nodes, determining partitioning schemes, and maintaining global metadata. For data exchange, however, it is handled by Sirius built-in exchange service layer explained in Section 7.2.2.4.

DuckDB supports Substrait and has an extension framework, allowing us to implement Sirius as an extension with *zero modification* to DuckDB's codebase. In contrast, Apache Doris does not have extension framework and cannot export Substrait plans. To accelerate Doris, we introduce minimal modifications to Doris codebase to convert its internal query plans into the Substrait format.

7.2.2.2 Query Execution Engine

The query execution engine in Sirius executes the Substrait plan generated by the optimizer. Sirius adopts a **pipeline execution model**, in which the query plan is divided into pipelines. Each task—at the granularity of a pipeline—is enqueued into a global task queue. Idle CPU threads pull tasks from this queue and execute the pipelines by invoking each operator. This model is widely-used in modern data systems such as DuckDB, Hyper, and Velox.

Within each pipeline, Sirius adopts the **push-based execution model**. The query executor acts as a coordinator, maintaining the state of query execution (e.g., operator's input and output). Instead of having operators *pull* data from their predecessors –as done in systems like Velox– the executor *pushes* data into operators. This design keeps the operators stateless, simplifying their implementation. This model is adopted by modern systems such as DuckDB.

Most physical operators in Sirius are implemented using the `libcudf` library, with a few exceptions for specialized functions such as predicate pushdown, and materialization. Thanks to its modular design, Sirius allows developers to easily switch the operator implementation between `libcudf` and custom CUDA kernels. Sirius also includes a graceful fallback mechanism to the host database systems in the case of an error or missing features in Sirius.

7.2.2.3 Buffer Manager

The buffer manager is responsible for managing device and host memory in Sirius. It divides the memory into two separate regions:

Data Caching. This region is used to store cached data in Sirius either in device memory or pinned host memory. To minimize the overhead of dynamic memory allocation during query execution, the caching region is pre-allocated in advance.

Data Processing. This region is in device memory which stores intermediate results during query execution (e.g., hash tables, intermediate results, etc.). Sirius uses the RMM [29] (RAPIDS Memory Manager) pool allocator to manage this region efficiently.

Moreover, the buffer manager is responsible for converting between different columnar formats used throughout the system: the internal Sirius columnar format, the `libcudf` columnar format, and the columnar format used by the host database. Both Sirius and libcudf derive their columnar format from Apache Arrow, which allows for zero-copy conversion via pointer passing. The only exception is when converting join results from `libcudf` to Sirius, as `libcudf` represents the row IDs using `int32_t`, while Sirius uses `uint64_t`. Conversion between Sirius and the host database format also requires deep copies, but this occurs only during the cold run (initial data load) and when returning results to the user.

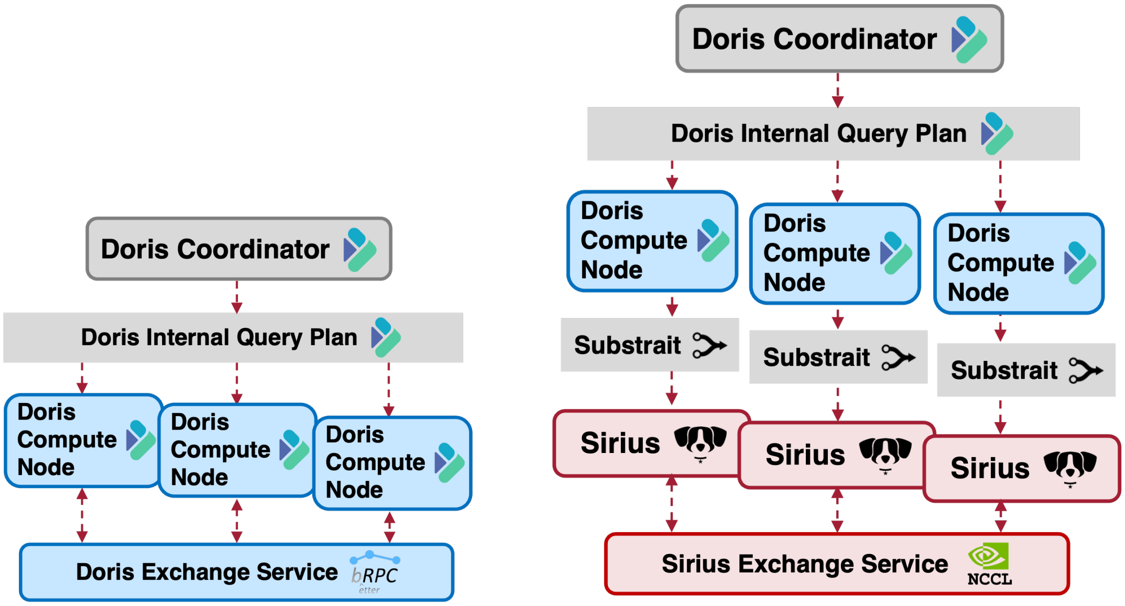
Currently, Sirius relies on the host database to read data from disk. Once data is read, the buffer manager automatically caches it into the pre-allocated caching region for future reuse. More discussion on disk support and spilling will be discussed in Section 7.2.4.

7.2.2.4 Exchange Service Layer[1]

The exchange service layer orchestrates distributed query execution across multiple nodes. In single-node deployments, this layer can be bypassed entirely. However, in multi-node settings, it plays a critical role in coordinating data exchange between nodes. In Sirius, exchange is modeled as dedicated physical operators. Sirius supports common exchange patterns—`broadcast`, `shuffle`, `merge`, and `multi-cast`—all implemented using NCCL primitives.

In distributed execution, the query plan is divided into multiple fragments, each of which is executed locally on a participating node. Data exchange occurs between fragments—after one fragment completes, its output is transmitted across the compute nodes and consumed as input by the subsequent fragments. Sirius internally maintains a runtime registry of exchanged intermediate data as temporary tables. These temporary tables are automatically deregistered once the corresponding fragments finish execution.

[1] The exchange service layer was implemented by Yifei Yang – a collaborator in the project



(a) Vanilla Doris

(b) Sirius-Accelerated Doris

Figure 7.2: Distributed Query Lifecycle in Doris vs Sirius.

7.2.3 Query Lifecycle in Sirius

In this section, we will describe the end-to-end query lifecycle when Sirius is used as a drop-in replacement for DuckDB (in single-node settings) and Apache Doris (in distributed environments).

Single-Node. Users begin by issuing SQL queries through the DuckDB CLI or Python API. DuckDB then parses and optimizes the query, producing a DuckDB-specific logical plan. In the vanilla DuckDB workflow, DuckDB-native execution engine will execute this plan and return the result to the user. When accelerated by Sirius, however, DuckDB delegates the execution to Sirius by passing the query plan to its GPU-accelerated execution engine (Section 7.2.2.2). Once completes, the buffer manager converts the result back to DuckDB's internal format and returns it to the user.

Distributed. Figure 7.2 illustrates the query lifecycle for vanilla Apache Doris (Figure 7.2(a)) and GPU-accelerated Doris powered by Sirius (Figure 7.2(b))[2]. In both cases, users submit SQL queries via the Doris SQL interface. The Doris coordinator generates a distributed query plan and dispatches plan fragments to Doris compute nodes. In standard Doris workflow, the compute nodes will execute the fragments

[2] The integration with Apache Doris was implemented by Yifei Yang – a collaborator in the project

locally, exchange intermediate data via Doris' native exchange service, and return the results to the coordinator.

When accelerated by Sirius, instead of executing the plan fragments directly, each compute node translates its assigned fragment into Substrait and forwards it to the local Sirius execution engine (Section 7.2.2.2). To exchange intermediate results across nodes, Sirius uses its exchange service layer described in Section 7.2.2.4. Once execution completes, Sirius returns the results to Doris compute node, which then forward the results to the coordinator.

7.2.4 Future Extensions

While our initial prototype targets the in-memory setting with limited feature sets, we plan to extend Sirius into a more complete and production-ready system.

Out-of-Core Execution. Currently, Sirius operates under the assumption that the input data fits in the caching region and the intermediate results fit in the processing region. To support larger-than-memory workloads, we plan to extend the buffer manager to support spilling to pinned memory and disk. We will also enhance the execution engine to support batch execution by partitioning input data into batches and pipelining them into the GPU [132, 62].

Full Distributed and Multi-GPU Support. The current distributed mode offers limited SQL coverage compared to its single-node counterpart. For instance, it does not support functions such as `avg`. Going forward, we plan to expand the SQL coverage for the distributed execution and introduce other key features such as fault tolerance. Additionally, we will extend Sirius to support multiple GPUs per node [128], enabling it to handle larger workload.

Expanding SQL Coverage. Sirius already supports most of the basic SQL operators and data types. We plan to broaden the coverage by adding support for more complex data types, such as LIST, and nested types. Additionally, we aim to implement advanced SQL operators, including ASOF joins, vector search, and UDFs.

Expanding Host Database Coverage. Sirius currently serves as a drop-in accelerator for both DuckDB and Apache Doris. Going forward, we plan to support other substrait-compatible systems, such as DataFusion, Ibis, and Gluten [4].

Performance Optimizations. Sirius is still in its early stages, and we see significant opportunities for performance improvement. Future optimizations include predicate transfer [127], efficient distributed shuffling, operator-specific enhancements [122], and lightweight compression techniques [109, 42] to mitigate GPU memory capacity limitations—particularly during batch/pipeline execution [46]. Additionally, we aim to optimize I/O paths for reading data from disk [115] and across the network using GPUDirect [14].

7.3 Evaluation

In this section, we will evaluate the performance of Sirius as a drop-in accelerator for DuckDB (single-node) and Doris (distributed).

7.3.1 Experiment Setup

Hardware configuration. We ran Sirius with two different setups:

- **NVIDIA GH200:** This machine features NVIDIA Grace CPU and Hopper GPU connected through NVLink C2C with 900 GB/s bidirectional bandwidth. The Hopper GPU has a 92 GB of HBM3 memory with a read/write bandwidth of 3 TB/s. The Grace CPU has 72 Neoverse Armv9 cores with 480 GB of LPDDR5X memory.

- **4 × NVIDIA A100:** This cluster consists of 4 identical nodes, where each node has one NVIDIA A100 GPU and 64-cores Intel Xeon Gold 6526Y CPUs. The A100 GPU has 40 GB of HBM3 memory with a read/write bandwidth of 1550 GB/s. The GPU and CPU are connected via PCIe4 with 25.6 GB/s bidirectional bandwidth. Every node support InfiniBand 4x NDR (Next-Generation Rate), providing network bandwidth of up to 400 Gbps.

Benchmark. We use the *TPC-H* benchmark in our experiment, which has been widely used both in academia and industry.

Measurement. We dedicate 50% of each GPU memory for data caching, and the other half for data processing. *The numbers reported are the hot runs when the data is already* ***cached in the GPU memory***. Throughout the experiment, we will evaluate the performance of the following query engines:

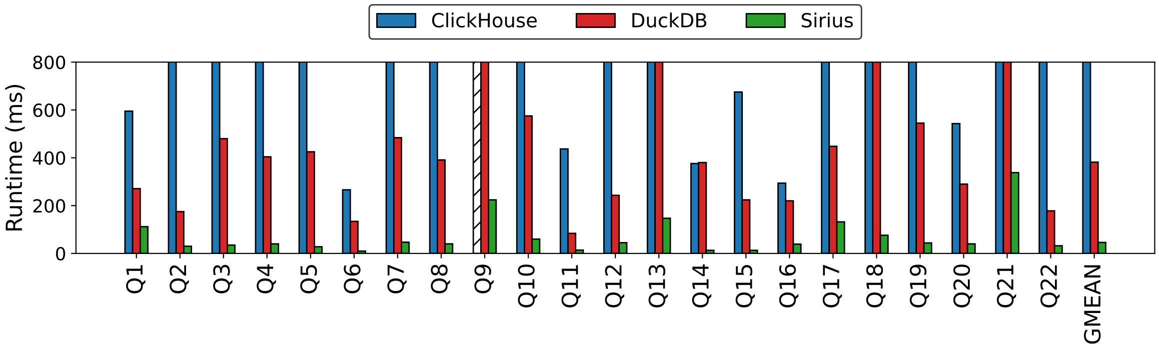


Figure 7.3: TPC-H End-to-end Query Performance on a Single Node.

- **DuckDB:** DuckDB [105] is a single node open-source embedded CPU-based analytical database.

- **ClickHouse:** ClickHouse [108] is a distributed open-source CPU-based online analytical database.

- **Apache Doris:** Doris [3] is an open source high-performance and real-time distributed CPU-based data warehouse.

- **Sirius:** Our GPU-native SQL engine that serves as drop-in accelerator for DuckDB (single node) and Apache Doris (distributed).

7.3.2 Single-Node Performance

In this Section, we evaluate Sirius performance running on a single node as a drop-in accelerator for DuckDB. We ran 22 TPC-H queries with a scale factor of 100 and compare the performance against DuckDB and ClickHouse. To ensure cost-normality, we ran DuckDB and ClickHouse on a CPU instance (m7i.16xlarge@AWS) that has the same hourly rental cost ($3.2/h) as the GPU instance (GH200@LambdaLabs) used to run Sirius.

End-to-end Performance Comparison. Figure 7.3 shows the end-to-end performance of various query engines on the TPC-H benchmark. Compared to DuckDB, Sirius achieves a 8.2× speedup with the same hardware rental cost. Sirius leverages DuckDB's optimized logical plans but replaces its backend with GPUs, demonstrating the significant performance advantage of GPU acceleration.

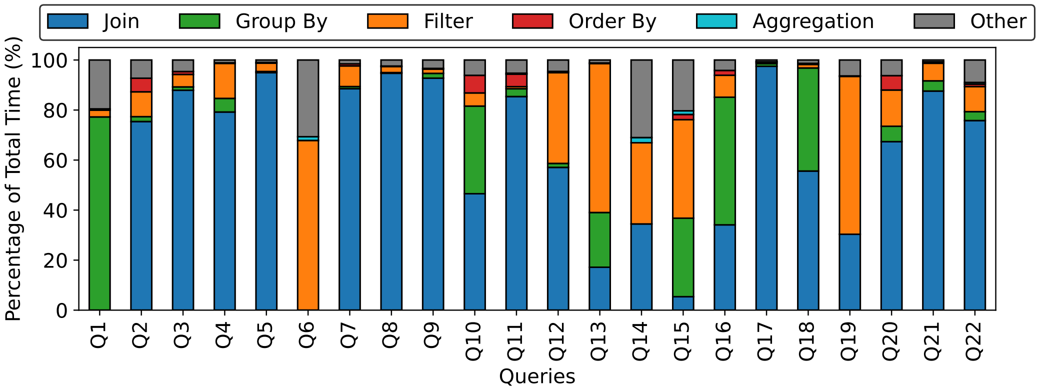


Figure 7.4: Performance Breakdown in Sirius

Since ClickHouse does not support correlated subqueries [108], we rewrite queries containing subquery correlation for compatibility. Compared to ClickHouse, Sirius is 60× faster with the same hardware rental cost. Q9 does not finish in ClickHouse and Q21 is not supported. We observed that ClickHouse is not optimized for join-heavy workloads, leading to suboptimal performance in many TPC-H queries (e.g., Q2, Q5, Q10, etc).

Performance Breakdown. Figure 7.4 presents a performance breakdown across different TPC-H queries in Sirius. The results indicate that join operations dominate query execution time in most cases, particularly in join-heavy queries (Q2–Q5, Q7–Q8, Q20–Q22).

Group-by also account for a substantial portion in several queries, notably for Q1, Q10, Q16, and Q18. This overhead is especially visible in queries involving group-by on string keys (Q10, Q18) or with a small number of distinct groups (Q1). For string keys, libcudf defaults to a sort-based group-by strategy, which is less performant than hash-based group-by. For group-by operations with a small number of groups, GPU suffers from memory contention.

Filter operation contributes to a majority of the query execution time in filter-heavy queries such as Q6 and Q19. They also contribute significantly in Q13, where a complex string-matching expression with very low selectivity is used. In contrast, aggregation and order-by operations do not dominate the end-to-end execution time for any of the TPC-H queries. This is because the input size for these operations is typically very small.

7.3.3 Distributed Performance

In this section, we ran a subset of TPC-H queries (i.e., Q1, Q3, and Q6) that are currently supported by the distributed mode of Sirius and compare the performance of Sirius against Doris and ClickHouse using a cluster of 4 × A100 GPUs.

Table 7.1: TPC-H End-to-End Query Performance in the Distributed Setting.

	Doris	ClickHouse	Sirius	Breakdown in Sirius		
				Compute	Exchange	Other
Q1	1193	393	**97**	33	3	61
Q3	838	12785	**341**	43	233	75
Q6	199	294	**84**	36	1	47

End-to-end Performance Comparison. Table 7.1 presents the performance comparison of different query engines in a distributed environment. Compared to Doris, Sirius achieves a speedup of 12.5×, 2.5×, and 2.4× in Q1, Q3, and Q6, respectively. Sirius leverages the exact same distributed query plan from Doris, but replaces its execution engine with GPUs, demonstrating the performance advantage of GPUs in distributed query execution.

Sirius outperforms ClickHouse by 4×, 37.5×, and 3.5× in Q1, Q3, and Q6, respectively. When joins are not involved (Q1, Q6), ClickHouse is able to outperform Doris. However, its performance degrades significantly when distributed joins are involved (i.e., Q3), resulting in substantial slowdown compared to Doris and Sirius.

Performance Breakdown. We further present a breakdown of Sirius' performance in Table 7.1, specifically, into the categories of local GPU computation, data exchange, and the rest portion during execution (e.g., query optimization, plan dispatching, etc.).

We noticed that GPU execution is not the primary performance bottleneck in the current implementation of distributed Sirius, indicating ample opportunities for further optimizations. For example, in Q1 and Q6, a significant portion of query execution time is spent in Doris' query optimizer and coordinator, which run on the CPU. This overhead does not scale with the data size and is expected to be less significant for larger dataset. In Q3, data exchange is the main bottleneck as the Doris' distributed query plan shuffles both the `orders` and `lineitem` tables. We have identified several opportunities to further improve the shuffle performance in Sirius but leave a deeper exploration to future work.

7.4 Conclusion

This work introduces Sirius, an open-source GPU-native SQL engine that provides drop-in acceleration across a variety of existing data systems. Sirius treats GPU as the primary engine and leverages libraries like libcudf for high-performance relational operators. It provides drop-in acceleration for existing databases by leveraging the standard Substrait query representation, replacing the CPU engine without changing the user-facing interface. Our evaluation on TPC-H benchmark shows that Sirius achieves 8.2× and up to 12.5× speedup when integrated with DuckDB (single-node) and Doris (distributed), respectively.

Chapter 8

Summary and Future Directions

The era of GPU-powered data analytics has arrived!

Over the past decade, the trajectory of GPU hardware has been nothing short of revolutionary. Increases in GPU peak performance, memory capacity, and memory bandwidth have far outpaced the CPU hardware advancement, which has stagnated due to the constraints of Moore's Law and Dennard scaling. At the same time, improvements in interconnect technologies are beginning to erode the fundamental bottleneck in data analytics: **data movement**. As these trends converge, the database landscape is shifting. In the foreseeable future, the bottleneck may no longer be I/O, but computation itself—an area where GPUs excel, with the throughput to saturate even the highest bandwidth.

Despite this potential, GPU databases still faced significant barriers to adoption. This thesis has identified three of the most critical: supporting data exceeding GPU memory capacity (**Challenge I**), supporting complex SQL operators on GPUs (**Challenge II**), and migrating from CPU to GPU-based DBMSes (**Challenge III**).

This dissertation has taken concrete steps toward addressing these challenges. To address **Challenge I**, we improved existing approaches to data compression (Chapter 3), CPU–GPU query execution (Chapter 4), and multi-GPU DBMS (Chapter 5) to enable large-scale data analytics on GPUs. To address **Challenge II**, we introduced a framework for executing UDAFs on GPUs which can laid a foundation for general UDF execution in the future (Chapter 6). To address **Challenge III**, we introduced *Sirius*, the first open-source GPU-native SQL engine designed to be a drop-in accelerator for existing databases (Chapter 7).

Looking ahead, we will explore future directions beyond this thesis to push even wider adoption of GPU databases.

8.1 Future Directions

Supporting Storage and S3. To fully overcome GPU memory capacity limitations, GPU databases must deliver speedups not only for in-memory data, but also for data residing on SSDs and S3. Technologies such as `GPUDirectStorage` and `GPUDirectRDMA` will be essential in this context. Achieving this requires more than simply scanning data from storage; it highlights the need for a GPU-native buffer management strategy to efficiently coordinate spilling, caching, and data movement to-and-from the GPU.
Distributed Query Execution on GPUs. We have demonstrated that Sirius can operate in distributed settings, but broader challenges—such as scalability, fault tolerance, and metadata management—remain open. There are also significant opportunities to optimize performance in distributed execution, for example by designing a GPU-native distributed runtime or by fully overlapping network transfers with GPU computation. Overcoming these challenges could enable GPU databases to efficiently power analytics at petabyte scale and beyond.
Complex Operators and Data Types. While we have demonstrated GPU execution of UDAFs, the broader challenge of supporting general UDFs remains unsolved. Several other operators are also still underexplored on GPUs, including `RECURSIVE SQL`, `ASOF JOIN`, `WINDOW FUNCTION`, and the integration between `SQL` and `VECTOR SEARCH`. Moreover, variable-length and nested data types—such as large strings, `LISTs`, `ARRAYs`, and `JSON`—can easily suffer from load-balancing issues on GPUs, presenting significant opportunities for further performance optimizations.

Until now, the absence of an open-source GPU database has made it difficult for researchers to build, validate, and compare new ideas. With the introduction of *Sirius*, that barrier has been removed. It is our hope that our work can serve as a catalyst—inviting researchers and practitioners alike to drive adoption, push the boundaries, and together kickstart the GPU era for data analytics!

references

[1] 2022. CUB Documentation. https://nvlabs.github.io/cub/.

[2] 2025. A Lightweight LLVM Python Binding for Writing JIT Compilers. https://pypi.org/project/llvmlite/.

[3] 2025. Apache Doris. https://doris.apache.org/.

[4] 2025. Apache Gluten. https://github.com/apache/incubator-gluten.

[5] 2025. AWS cuts costs for H100, H200, and A100 instances by up to 45%. https://nvidia.github.io/spark-rapids/.

[6] 2025. BlazingSQL. https://blazingsql.com.

[7] 2025. Cloudian Shatters AI Storage Barriers with Direct GPU-to-Object Storage Access. https://cloudian.com/blog/cloudian-delivers-groundbreaking-performance-with-nvidia-gpudirect-support.

[8] 2025. cuCollections. https://github.com/NVIDIA/cuCollections.

[9] 2025. cuDF. https://github.com/rapidsai/cudf.

[10] 2025. cuDF- GPU DataFrame Library. https://github.com/rapidsai/cudf.

[11] 2025. Dask-CUDA. https://docs.rapids.ai/api/dask-cuda/nightly/.

[12] 2025. Extending Numba. https://numba.readthedocs.io/en/latest/extending/index.html.

[13] 2025. Extending Velox - GPU Acceleration with cuDF. https://velox-lib.io/blog/extending-velox-with-cudf/.

[14] 2025. GPU Direct. https://developer.nvidia.com/gpudirect.

[15] 2025. HeavyAI. https://www.heavy.ai/.

[16] 2025. Kinetica. https://kinetica.com/.

[17] 2025. NCCL. https://developer.nvidia.com/nccl.

[18] 2025. Numba. https://numba.pydata.org/.

[19] 2025. NVComp. https://github.com/NVIDIA/nvcomp.

[20] 2025. NVIDIA H100 Price Guide 2025: Detailed Costs, Comparisons & Expert Insights. https://docs.jarvislabs.ai/blog/h100-price.

[21] 2025. NVLINK. https://www.nvidia.com/en-us/data-center/nvlink/.

[22] 2025. NVLink-C2C. https://www.nvidia.com/en-us/data-center/nvlink-c2c/.

[23] 2025. NVVM IR. https://docs.nvidia.com/cuda/nvvm-ir-spec/.

[24] 2025. Opencl. https://www.khronos.org/opencl/.

[25] 2025. Pandarallel. https://nalepae.github.io/pandarallel/.

[26] 2025. Parquet Encoding Format. https://github.com/apache/parquet-format/blob/master/Encodings.md.

[27] 2025. Pyjion - A drop-in JIT Compiler for Python 3.10. https://www.trypyjion.com/.

[28] 2025. RAPIDS. https://rapids.ai.

[29] 2025. Rapids Memory Manager. https://github.com/rapidsai/rmm.

[30] 2025. Sirius: A GPU-Native SQL Engine. https://news.ycombinator.com/item?id=44404876.

[31] 2025. Sirius: A GPU-Native SQL Engine. https://github.com/sirius-db/sirius.

[32] 2025. Spark RAPIDS. https://nvidia.github.io/spark-rapids/.

[33] 2025. Substrait. https://substrait.io/.

[34] 2025. The Missing Guide to the H100 GPU Market. `https://blog.lepton.ai/the-missing-guide-to-the-h100-gpu-market-91ebfed34516`.

[35] 2025. Thrust. `https://thrust.github.io/`.

[36] 2025. User Defined Aggregate Functions (UDAFs). `https://docs.oracle.com/cd/B10501_01/appdev.920/a96595/dci11agg.htm`.

[37] 2025. User Defined Aggregate Functions (UDAFs). `https://spark.apache.org/docs/latest/sql-ref-functions-udf-aggregate.html`.

[38] 2025. User Defined Aggregates. `https://www.postgresql.org/docs/current/xaggr.html`.

[39] 2025. Voltron Data. `https://voltrondata.com/`.

[40] Abadi, Daniel, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 acm sigmod international conference on management of data*, 671–682. ACM.

[41] Abadi, Daniel J., Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. 2007. Materialization strategies in a column-oriented dbms. In *2007 ieee 23rd international conference on data engineering*, 466–475.

[42] Afroozeh, Azim, Lotte Felius, and Peter Boncz. 2024. Accelerating gpu data processing using fastlanes compression. In *Proceedings of the 20th international workshop on data management on new hardware*. DaMoN '24, New York, NY, USA: Association for Computing Machinery.

[43] Anh, Vo Ngoc, and Alistair Moffat. 2004. Inverted index compression using word-aligned binary codes. *Information Retrieval* 8:151–166.

[44] ———. 2010. Index compression using 64-bit words. *Softw. Pract. Exper.* 40(2): 131–147.

[45] Blelloch, Guy E. 1989. Scans as primitive parallel operations. *IEEE Transactions on computers* 38(11):1526–1538.

[46] Boeschen, Nils, Tobias Ziegler, and Carsten Binnig. 2024. Golap: A gpu-in-data-path architecture for high-speed olap. *Proc. ACM Manag. Data* 2(6).

[47] Boncz, Peter A, Marcin Zukowski, and Niels Nes. 2005. Monetdb/x100: Hyper-pipelining query execution. In *Cidr*, vol. 5, 225–237.

[48] Boncz, Peter Alexander, et al. 2002. *Monet: A next-generation dbms kernel for query-intensive applications*. Universiteit van Amsterdam [Host].

[49] Breí, Sebastian, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. 2013. Efficient co-processor utilization in database query processing. *Inf. Syst.* 38(8):1084–1096.

[50] Breß, Sebastian. 2014. The design and implementation of cogadb: A column-oriented gpu-accelerated dbms. *Datenbank-Spektrum* 14:199–209.

[51] Breß, Sebastian, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. 2013. Efficient co-processor utilization in database query processing. *Inf. Syst.* 38:1084–1096.

[52] Breß, Sebastian, Henning Funke, and Jens Teubner. 2016. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 international conference on management of data*, 1891–1906. SIGMOD '16, New York, NY, USA: Association for Computing Machinery.

[53] Breß, Sebastian, Ingolf Geist, Eike Schallehn, Maik Mory, and Gunter Saake. 2012. A framework for cost based optimization of hybrid cpu/gpu query plans in database systems. *Control and Cybernetics* 41.

[54] Breß, Sebastian, Max Heimel, Michael Saecker, Bastian Köcher, Volker Markl, and Gunter Saake. 2014. Ocelot/hype: Optimized data processing on heterogeneous hardware. *Proc. VLDB Endow.* 7:1609–1612.

[55] Breß, Sebastian, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal* 27(6):797–822.

[56] Breß, Sebastian, Siba Mohammad, and Eike Schallehn. 2012. In *Self-tuning distribution of db-operations on hybrid cpu/gpu platforms*.

[57] Breß, Sebastian, and Gunter Saake. 2013. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.* 6(12):1398–1403.

[58] Breß, Sebastian, Norbert Siegmund, Max Heimel, Michael Saecker, Tobias Lauer, Ladjel Bellatreche, and Gunter Saake. 2014. Load-aware inter-co-processor parallelism in database query processing. *Data Knowl. Eng.* 93: 60–79.

[59] Breß, Sebastian, Felix Beier, Hannes Rauhe, Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. 2012. In *Automatic selection of processing units for coprocessing in databases*, vol. 7503, 57–70.

[60] Breß, Sebastian, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2013. In *An operator-stream-based scheduling engine for effective gpu coprocessing*, vol. 8133, 288–301.

[61] Cao, Jiashen, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. Gpu database systems characterization and optimization. *Proc. VLDB Endow.* 17(3):441–454.

[62] Chrysogelos, Periklis, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. *Proc. VLDB Endow.* 12(5):544–556.

[63] Chrysogelos, Periklis, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. 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.

[64] Cohen, Sara. 2006. User-defined aggregate functions: Bridging theory and practice. In *Proceedings of the 2006 acm sigmod international conference on management of data*, 49–60. SIGMOD '06, New York, NY, USA: Association for Computing Machinery.

[65] Cutting, D., and J. Pedersen. 1989. In *Optimization for dynamic inverted index maintenance*, 405–411. SIGIR '90, New York, NY, USA: Association for Computing Machinery.

[66] Damme, Patrick, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *Edbt*, 72–83.

[67] Dean, Jeffrey. 2009. Challenges in building large-scale information retrieval systems: invited talk. In *Wsdm '09: Proceedings of the second acm international conference on web search and data mining*, 1–1. New York, NY, USA.

[68] Fang, Wenbin, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proceedings of the VLDB Endowment* 3(1-2):670–680.

[69] Feng, Ziqiang, Eric Lo, Ben Kao, and Wenjian Xu. 2015. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 acm sigmod international conference on management of data*, 31–46.

[70] Funke, Henning, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 international conference on management of data*, 1603–1618. ACM.

[71] Funke, Henning, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 international conference on management of data*, 1603–1618. SIGMOD '18, New York, NY, USA: Association for Computing Machinery.

[72] Funke, Henning, and Jens Teubner. 2020. Data-parallel query processing on non-uniform data. *Proc. VLDB Endow.* 13(6):884–897.

[73] Gao, Hao. 2021. Scaling joins to a thousand gpus. In *Adms@vldb*.

[74] Goldstein, Jonathan, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th international conference on data engineering*, 370–379. IEEE.

[75] Govindaraju, Naga, et al. 2006. Gputerasort: high performance graphics coprocessor sorting for large database management. In *Sigmod*.

[76] Gray, Jim, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 acm sigmod international conference on management of data*, 243–252. SIGMOD '94, New York, NY, USA: Association for Computing Machinery.

[77] Harris, Mark, Shubhabrata Sengupta, and John D Owens. 2007. Parallel prefix sum (scan) with cuda. *GPU gems* 3(39):851–876.

[78] He, Bingsheng, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.* 34(4).

[79] He, Bingsheng, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Sigmod*.

[80] He, Jiong, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *PVLDB*.

[81] He, Jiong, Shuhao Zhang, and Bingsheng He. 2014. In-cache query co-processing on coupled cpu-gpu architectures. *Proc. VLDB Endow.* 8(4):329–340.

[82] Heimel, Max, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*.

[83] Huang, Zezhou, Krystian Sakowski, Hans Lehnert, Wei Cui, Carlo Curino, Matteo Interlandi, Marius Dumitru, and Rathijit Sen. 2025. Gpu acceleration of sql analytics on compressed data. `2506.10092`.

[84] Huffman, David A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40(9):1098–1101.

[85] Kabić, Marko, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A modular accelerated query engine for data analytics on heterogeneous systems. *Proc. ACM Manag. Data* 3(3).

[86] Kaldewey, Tim, Guy Lohman, Rene Mueller, and Peter Volk. 2012. Gpu join processing revisited. In *Damon*.

[87] Karnagel, Tomas, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive work placement for query processing on heterogeneous computing resources. *Proc. VLDB Endow.* 10(7):733–744.

[88] Kraus, Keith. 2021. The State of RAPIDS AI. GPU Technical Conference 2021.

[89] Lemire, Daniel, and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45(1):1–29.

[90] Li, Jing, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9(14):1647–1658.

[91] Li, Yinan, and Jignesh M Patel. 2013. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 acm sigmod international conference on management of data*, 289–300.

[92] Lutz, Clemens, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. In *Pump up the volume: Processing large data on gpus with fast interconnects*, 1633–1649. SIGMOD '20, New York, NY, USA: Association for Computing Machinery.

[93] Mallia, Antonio, Michał Siedlaczek, Torsten Suel, and Mohamed Zahran. 2019. In *Gpu-accelerated decoding of integer lists*, 2193–2196. CIKM '19, New York, NY, USA: Association for Computing Machinery.

[94] Meraji, Sina, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. 2016. Towards a hybrid design for fast query processing in db2 with blu acceleration using graphical processing units: A technology demonstration. In *Proceedings of the 2016 international conference on management of data*, 1951–1960. SIGMOD '16, New York, NY, USA: Association for Computing Machinery.

[95] Neumann, Thomas. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4(9):539–550.

[96] Nicholson, Hamish, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. Hetcache: Synergising nvme storage and gpu acceleration for memory-efficient analytics.

[97] O'Neil, Patrick, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology conference on performance evaluation and benchmarking*, 237–252. Springer.

[98] Pandey, Shweta, and Arkaprava Basu. 2025. H-rocks: Cpu-gpu accelerated heterogeneous rocksdb on persistent memory. *Proc. ACM Manag. Data* 3(1).

[99] Paul, Johns, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on gpus. *Proc. VLDB Endow.* 14(2):202–214.

[100] ———. 2020. Improving execution efficiency of just-in-time compilation based query processing on gpus. *Proc. VLDB Endow.* 14(2):202–214.

[101] Paul, Johns, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. *Mg-join: A scalable join for massively parallel multi-gpu architectures*, 1413–1425. New York, NY, USA: Association for Computing Machinery.

[102] Pedreira, Pedro, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proc. VLDB Endow.* 15(12):3372–3384.

[103] Pedreira, Pedro, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The composable data management system manifesto. *Proc. VLDB Endow.* 16(10): 2679–2685.

[104] Polychroniou, Orestis, and Kenneth A. Ross. 2015. Efficient lightweight compression alongside fast scans. In *Proceedings of the 11th international workshop on data management on new hardware*. DaMoN'15, New York, NY, USA: Association for Computing Machinery.

[105] Raasveldt, Mark, and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*, 1981–1984. SIGMOD '19, New York, NY, USA: Association for Computing Machinery.

[106] Rui, Ran, Hao Li, and Yi-Cheng Tu. 2020. Efficient join algorithms for large database tables in a multi-gpu environment. *Proc. VLDB Endow.* 14(4):708–720.

[107] Rui, Ran, and Yi-Cheng Tu. 2017. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th international conference on scientific and statistical database management*, 17. ACM.

[108] Schulze, Robert, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. Clickhouse - lightning fast analytics for everyone. *Proc. VLDB Endow.* 17(12):3731–3744.

[109] Shanbhag, Anil, Bobbi Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based lightweight integer compression in gpu. In *Proceedings of the 2022 acm sigmod international conference on management of data*.

[110] Shanbhag, Anil, Xiangyao Yu, and Samuel Madden. 2020. A study of the fundamental performance charecteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 international conference on management of data*. ACM.

[111] Sioulas, Panagiotis, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious hash-joins on gpus. Tech. Rep.

[112] Sitaridi, Evangelia A, and Kenneth A Ross. 2013. Optimizing select conditions on gpus. In *Proceedings of the ninth international workshop on data management on new hardware*, 4. ACM.

[113] Stehle, Elias, and Hans-Arno Jacobsen. 2017. A memory bandwidth-efficient hybrid radix sort on gpus. In *Sigmod*. ACM.

[114] Thostrup, Lasse, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed gpu joins on fast rdma-capable networks. *Proc. ACM Manag. Data* 1(1).

[115] Torp, Karl B., Simon A. F. Lund, and Pınar Tözün. 2025. In *Path to gpu-initiated i/o for data-intensive systems*. DaMoN '25, New York, NY, USA: Association for Computing Machinery.

[116] Traiger, Irving L., Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. 1982. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.* 7(3):323–342.

[117] Vandierendonck, H., and P. Trancoso. 2006. Building and validating a reduced tpc-h benchmark. In *14th ieee international symposium on modeling, analysis, and simulation*, 383–392.

[118] Wang, Jianguo, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 acm international conference on management of data*, 993–1008. SIGMOD '17, New York, NY, USA: Association for Computing Machinery.

[119] Wang, Kaibo, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent analytical query processing with gpus. *Proceedings of the VLDB Endowment* 7(11):1011–1022.

[120] Willhalm, Thomas, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment* 2(1): 385–394.

[121] Wu, Bowen, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. 2025. Terabyte-scale analytics in the blink of an eye. `2506.09226`.

[122] Wu, Bowen, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently processing joins and grouped aggregations on gpus. *Proc. ACM Manag. Data* 3(1).

[123] Wu, Haicheng, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th annual ieee/acm international symposium on microarchitecture*. IEEE.

[124] ———. 2012. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th annual ieee/acm international symposium on microarchitecture*, 107–118.

[125] Yabuta, Makoto, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational joins on gpus: A closer look. *IEEE Transactions on Parallel and Distributed Systems* 28(9):2663–2673.

[126] Yan, Hao, Shuai Ding, and Torsten Suel. 2009. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on world wide web*, 401–410. WWW '09, New York, NY, USA: Association for Computing Machinery.

[127] Yang, Yifei, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2023. Predicate transfer: Efficient pre-filtering on multi-join queries. `2307.15255`.

[128] Yogatama, Bobbi, Weiwei Gong, and Xiangyao Yu. 2024. Scaling your hybrid cpu-gpu dbms to multiple gpus. *Proc. VLDB Endow.* 17(13):4709–4722.

[129] Yogatama, Bobbi, Brandon Miller, Yunsong Wang, Graham Markall, Jacob Hemstad, Gregory Kimball, and Xiangyao Yu. 2023. Accelerating user-defined aggregate functions (udaf) with block-wide execution and jit compilation on gpus. In *Proceedings of the 19th international workshop on data management on new hardware*, 19–26. DaMoN '23, New York, NY, USA: Association for Computing Machinery.

[130] Yogatama, Bobbi, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, Weiwei Gong, and Xiangyao Yu. 2025. Rethinking analytical processing in the gpu era. `2508.04701`.

[131] Yogatama, Bobbi W, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous cpu-gpu dbms. *Proceedings of the VLDB Endowment* 15(11):2491–2503.

[132] Yuan, Yichao, Advait Iyer, Lin Ma, and Nishil Talati. 2025. Vortex: Overcoming memory capacity limitations in gpu-accelerated large-scale data analytics. `2502.09541`.

[133] Yuan, Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The yin and yang of processing data warehousing queries on gpu devices. *PVLDB*.

[134] Zhang, Jiangong, Xiaohui Long, and Torsten Suel. 2008. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on world wide web*, 387–396. WWW '08, New York, NY, USA: Association for Computing Machinery.

[135] Zhang, Kai, Feng Chen, Xiaoning Ding, Yin Huai, Rubao Lee, Tian Luo, Kaibo Wang, Yuan Yuan, and Xiaodong Zhang. 2015. Hetero-db: Next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. *Journal of Computer Science and Technology* 30.

[136] Zhang, Shuhao, Jiong He, Bingsheng He, and Mian Lu. 2013. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proc. VLDB Endow.* 6(12):1374–1377.

[137] Ziv, Jacob, and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23(3):337–343.

[138] Zukowski, M., S. Heman, N. Nes, and P. Boncz. 2006. Super-scalar ram-cpu cache compression. In *22nd international conference on data engineering (icde'06)*, 59–59.

[139] Zukowski, Marcin, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar ram-cpu cache compression. In *22nd international conference on data engineering (icde'06)*, 59–59. IEEE.