In-memory analytical query processing on GPUs

Paul, Johns

2021

Paul, J. (2021). In-memory analytical query processing on GPUs. Doctoral thesis, Nanyang Technological University, Singapore. <https://hdl.handle.net/10356/146888>

<https://hdl.handle.net/10356/146888>

<https://doi.org/10.32657/10356/146888>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 23:19:33 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

**IN-MEMORY ANALYTICAL QUERY PROCESSING
ON GPUS**

JOHNS PAUL

School of Computer Science & Engineering

2021

IN-MEMORY ANALYTICAL QUERY PROCESSING ON GPUS

JOHNS PAUL

School of Computer Science & Engineering

A thesis submitted to the Nanyang Technological
University in partial fulfilment of the requirement for
the degree of
Doctor of Philosophy

2021

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

25-Sept-2020

.....

Date

.....

Johns Paul

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

2/10/20

.....

Date

.....

Assoc Prof. Chiew Tong Lau

Authorship Attribution Statement

This thesis contains material from 3 paper(s) published/submitted to the following peer-reviewed journal(s) where I was the first and/or corresponding author.

Chapter 4 is published as Johns Paul, Binghsheng He, Shengliang Lu and Chiew Tong Lau. Revisiting Hash Join on Graphics Processors: A Decade Later, 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), Macao, Macao, 2019, pp. 294-299, doi: <https://doi.org/10.1109/ICDEW.2019.00008>. An extended version of the manuscript is also published as Johns Paul, Binghsheng He, Shengliang Lu and Chiew Tong Lau. Revisiting hash join on graphics processors: a decade later. Distributed and Parallel Databases (2020). <https://doi.org/10.1007/s10619-019-07280-z>.

- I developed the techniques, conducted the evaluation and prepared the manuscript draft.
- Assoc. Prof Bingsheng He provided guidance for the project.
- Assoc. Prof Chiew Tong Lau reviewd the reserach to identify shortcomings.
- Shengliang Lu assisted with the evaluation.

Chapter 5 is submitted as Johns Paul, Bingsheng He, Shengliang Lu and Chiew Tong Lau. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In 2021 International Conference on Management of Data (SIGMOD).

- I developed the techniques, conducted the evaluation and prepared the manuscript draft.
- Assoc. Prof Bingsheng He provided guidance for the project.
- Assoc. Prof Chiew Tong Lau reviewed the research to identify shortcomings.
- Shengliang Lu assisted with the manuscript draft.

Chapter 6 is accepted as Johns Paul, Bingsheng He, Shengliang Lu and Chiew Tong Lau. Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs. In PVLDB 2020.

- I developed the techniques, conducted the evaluation and prepared the manuscript draft.
- Assoc. Prof Bingsheng He provided guidance for the project.
- Assoc. Prof Chiew Tong Lau reviewed the research to identify shortcomings.
- Shengliang Lu assisted with the evaluation.

25-Sept-2020

.....

Date

.....

Johns Paul

Acknowledgements

This thesis would not have been possible without the kind and generous assistance offered by many people to whom I owe a lot. While it is difficult to enumerate all the people I appreciate, I would like to list a few of them below to acknowledge their selfless help.

First and foremost, I would like to thank Assoc Prof. Bingsheng He, who was initially my supervisor and later co-supervisor during the past 5 years. His support and guidance have been critical to the progress I made in research during these years. His vision and enthusiasm have been a guiding light for my research throughout these years. He constantly encouraged me to challenge myself and not to give up during difficult times. I am incredibly grateful for having him as an advisor both in research and life.

I would also like to thank Assoc Prof Chiew Tong Lau, who was my supervisor at Nanyang Technological University during the final stages of my PhD. Without his guidance and support, I would not have been able to complete the final stages of my research.

I am also fortunate to have worked with Shengliang Lu at National University of Singapore. I would like to thank him for his hard work and assistance in conducting experiments and preparing the manuscript for numerous projects. He has also been a great friend during these years and constantly encouraged me to continue my research.

I am also lucky to be a member of the Xtra Computing Group, where I worked with Shuhao Zhang, Zeyi Wen, Zeke Wang and Jiong He. Their thoughtful advice and feedback have helped me significantly to improve my research skills. I am also grateful to Yong Liang Goh, Xiang Hui Nicholas Lim and Prof. See-Kiong Ng for their support during my time at Grab-AI lab at National University of Singapore.

Last but not least, I need to thank my friends and family for their enduring support throughout this long and stressful period.

Abstract

The high global memory bandwidth and the large number of parallel cores available in modern Graphics Processing Units (GPUs) make them ideal for high-performance Online Analytical Processing (OLAP) systems. However, it is challenging to design efficient high-performance query processing systems for GPUs due to the following reasons: 1) the rapid evolution of GPU hardware in the last decade, 2) the significant differences in the hardware architecture of GPUs when compared to CPUs, 3) the high overhead of moving the data between the CPU and GPU and 4) the small global memory size of a single GPU that necessitates the access of remote data over PCIe or NVLink interconnects when processing large data sets. In this thesis, we study existing query processing systems for GPUs and propose techniques to improve query execution efficiency on GPUs.

We begin by studying the performance of hash join, which is one of the most compute and memory intensive relational operator in OLAP systems. Specifically, we first revisit the major GPU hash join implementations in the past decade that were designed to execute on a single GPU. We then detail how these implementations take advantage of different GPU architecture features and conduct a comprehensive evaluation of their performance and cost-efficiency using different generations of GPUs. This helps shed light on the impact of different GPU architecture features on the performance of the hash join operation and identify the factors guiding the choice of these features. We further study how data characteristics like skew and match rate impact the performance of GPU hash join implementations. Novel techniques to improve the performance of the hash join operation when joining input relations with severe skew or high match rate were also proposed as part of this study. Our evaluation finds that the proposed techniques help avoid any performance degradation when joining skewed input relations and achieve up to 2.5x better performance when joining input relations with high match rate.

Next, we extend our study on the hash join operation to modern multi-GPU architectures. The recent scale-up of GPU hardware through the integration of multiple

GPUs into a single machine and the introduction of higher bandwidth interconnects like NVLink 2.0 has enabled new opportunities of relational query processing on multiple GPUs. However, due to the unique characteristics of GPUs and the interconnects, existing hash join implementations spend up to 66% of their execution time moving the data between the GPUs and achieve lower than 50% utilization of the newer high bandwidth interconnects. This leads to extremely poor scalability of hash join performance on multiple GPUs, which can be slower than the performance on a single GPU. In this thesis, we propose MG-Join, a scalable partitioned hash join implementation on multiple GPUs of a single machine. In order to effectively improve the bandwidth utilization, we develop a novel multi-hop routing for cross-GPU communication that adaptively chooses the efficient route for each data flow to minimize congestion. Our experiments on the DGX-1 machine show that MG-Join helps significantly reduce the communication overhead and achieves up to 97% utilization of the bisection bandwidth of the interconnects, resulting in significantly better scalability. Overall, MG-Join outperforms the state-of-the-art hash join implementations by up to 2.5x. MG-Join further helps improve the overall performance of TPC-H queries by up to 4.5x over multi-GPU version of an open-source commercial GPU database Omnisci.

Finally, we focus on optimizing the execution of entire queries on GPUs. Specifically, we look at improving the execution efficiency of just-in-time (JIT) compilation based query processing systems on GPUs. We observe that JIT compile-based query processing encounters *severe under-utilization of GPU hardware* due to divergent execution and degraded parallelism arising from resource contention. To address these issues, we propose a JIT compile-based query engine named *Pyper* to reduce divergent execution and improve hardware utilization during query execution on GPU. Specifically, Pyper has two new operators, *Shuffle* and *Segment*, for query plan transformation, which can be plugged into a physical query plan to reduce divergent execution and resolve resource contention, respectively. To determine the insertion points for these two operators, we present an analytical model that helps insert Shuffle and Segment operators into a query plan in a cost-based manner. Our experiments show that 1) the analytical analysis of divergent execution and resource contention improves the accuracy of the cost model and 2) Pyper on average can improve the performance of TPC-H and SSB queries by up to 5.01x and 3.06x when compared to Omnisci and Hawk, respectively.

Contents

Acknowledgements	ix
Abstract	xi
List of Figures	xvii
List of Tables	xxi
1 Introduction	1
1.1 Research Objectives	3
1.2 Research Contributions	4
1.3 Thesis Organization	5
2 Preliminaries	7
2.1 GPU Hardware Architecture	7
2.1.1 GPU Architecture Evolution	8
2.1.2 Multi-GPU Architecture	10
2.2 GPU Programming Model	12
2.2.1 Thread Execution Model	12
2.2.2 Memory Access Model	13
2.3 Query Processing on GPUs	14
3 Literature Survey	17
3.1 CPU Database Systems	17
3.2 Database Design on GPUs	20
3.2.1 Data Storage & Data Access	20
3.2.2 Operators Design	22
3.2.3 Concurrency Control	26
3.3 Query Processing on Single GPUs	27
3.3.1 Kernel Based Execution Systems	28
3.3.2 Pipelined Execution Systems	29
3.3.3 JIT Compilation Based Systems	29
3.4 Query Processing on Multiple Processors	30
3.4.1 Heterogeneous Architecture	30

3.4.2	Multi-GPU Architecture	31
4	Revisiting Hash Join on Graphics Processors: A Decade Later	33
4.1	Introduction	33
4.2	Implementation Details	35
4.3	Handling Data Skew & Match Rate	37
4.3.1	Data skew	37
4.3.2	Match Rate	40
4.4	Experimental Evaluation	43
4.4.1	Experimental Setup	43
4.4.2	Overall Comparison	44
4.4.3	Impact of GPU Architecture Features	45
4.4.3.1	Atomic and Shuffle Instructions	45
4.4.3.2	CUDA Streams	47
4.4.3.3	Unified Memory & Hardware Page Fault	47
4.4.3.4	High Bandwidth Memory	49
4.4.4	Handling Data Skew & Match Rate	51
4.4.5	Comparison Against CPU	52
4.5	Discussion	53
4.6	Summary	55
5	MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures	57
5.1	Introduction	57
5.2	MG-Join: Design Overview	61
5.2.1	Design Rationales	61
5.2.2	Design Overview	62
5.3	Data Distribution	65
5.3.1	Multi-Hop Data Transmission	66
5.3.2	Adaptive Routing	68
5.3.2.1	Need for adaptive routing.	69
5.3.2.2	Adaptive Routing in MG-Join	71
5.3.2.3	An Adaptive Routing Example	73
5.4	Experiments	74
5.4.1	Experimental Setup	74
5.4.2	Evaluating Data Distribution in Global Partitioning Phase	75
5.4.3	Evaluation of Join Performance	78
5.4.3.1	Factor Analysis	81
5.4.3.2	Evaluation on Other Hardware	82
5.4.4	TPC-H Query Evaluation	82
5.5	Summary	84
6	Improving Execution Efficiency of Compiled Systems on GPUs	85
6.1	Introduction	85

6.2	GPU Hardware & CUDA	88
6.3	Design of Pyper	89
6.3.1	Overview	89
6.3.2	Shuffle Operator	90
6.3.3	Segment Operator	94
6.4	Analytical Model	97
6.4.1	Pipeline Execution Cost	99
6.4.2	Estimation of γ_i	99
6.4.3	Estimation of Computation Cost ($\hat{C}_i(\gamma_i)$)	101
6.4.4	Estimation of Memory Cost ($\hat{M}_i(\gamma_i)$)	101
6.4.5	Example Cost Model Computation	103
6.5	Augmentation Optimizer	104
6.6	Experiments	107
6.6.1	Experimental Setup	107
6.6.2	Model Evaluation	109
6.6.3	Overall Comparison	112
6.6.4	Individual Operator Evaluation	114
6.6.5	Evaluation on Different GPU Architectures	120
6.6.6	Evaluation Against CPUs	120
6.7	Summary	121
7	Conclusion & Future Work	123
7.1	Contributions	123
7.2	Future Work	125
7.2.1	Query Execution on Multi-GPU Architectures	125
7.2.2	Optimizing for the New Bottleneck	126
7.2.3	GPUs in Edge Computing	126
	List of Publications	129
	Bibliography	131

List of Figures

2.1	GPU internal architecture.	8
2.2	Architecture design of DGX-1 machine from NVIDIA.	11
2.3	Steps involved in analytical query execution on GPUs.	14
4.1	Timeline of hash join implementations and GPU architecture features.	34
4.2	Execution timeline of PRBC and BSH.	37
4.3	Skew handling mechanism in BSH and BSH (w/ Skew Handling).	40
4.4	Performance of GPU hash join implementations on P100 GPU.	45
4.5	Occupancy rates achieved by WSH and BSH.	46
4.6	Normalized performance of BSH and PTH on K40c and P100.	46
4.7	Performance of BSH & PRBC on P100 GPU with CUDA Streams disabled.	48
4.8	Normalized performance of PRBC and UMJ implementations of the M60 and P100 GPU.	48
4.9	Performance comparison of P100 and Titan Xp for global memory write operation.	50
4.10	Speedup achieved by the Titan Xp GPU over the P100 GPU for the BSH implementation.	50
4.11	Performance comparison of BSH & BSH(w/ Skew Handling) under varying skew levels.	52
4.12	Performance comparison of BSH, BSH(w/ Baseline Estimation) & BSH(w/ Estimation).	52
4.13	Comparison against CPU.	53
5.1	Join performance and execution time break down of partitioned hash join on DGX-1 server from NVIDIA.	59
5.2	Architecture design of DGX-1.	60
5.3	Multi-hop data transmission example in MG-Join.	67
5.4	Impact of hardware configuration, data distribution and packet size on static routing policy (DGX-1)	70
5.5	Example routing for adaptive and static policies.	73
5.6	Data Transfer throughput comparison of multi-hop routing in MG-Join against direct routing in DPRJ.	76
5.7	Data Transfer throughput comparison of adaptive routing in MG-Join against static routing policies.	77

5.8	Interconnect hardware utilization of bisection bandwidth in MG-Join and DPRJ.	78
5.9	Normalized performance of different routing policies with data skew when executing over 8 GPUs of the DGX-1 machine.	78
5.10	Overall performance comparison of MG-Join against DPRJ and UMJ.	79
5.11	Execution time breakdown for MG-Join and DPRJ with varying number of GPUs.	80
5.12	Performance evaluation of MG-Join, DPRJ and UMJ for varying input size on DGX-1.	80
5.13	A factor analysis for MG-Join. Changes are added left to right and are cumulative.	81
5.14	Overall performance comparison of MG-Join against DPRJ and UMJ on the DGX-Station.	82
5.15	TPC-H query evaluation of MG-Join, Omnisci CPU and Omnisci GPU for a scale factor of 250.	83
6.1	Execution time breakdown of TPC-H and SSB queries demonstrating the impact of divergent execution and resource contention among threads.	87
6.2	System Architecture of Pyper	90
6.3	Warp executions with and without Shuffle operators	91
6.4	Impact of Segment operators on query plan and resource requirements	96
6.5	Variation of sequential memory access cost with γ_i for V100 GPU	103
6.6	Example cost estimation by the analytical model ($\downarrow$ indicates a decrease in execution cost and $\uparrow$ indicates an increase in the overhead resulting from the insertion of the Shuffle or Segment operator.)	104
6.7	An example for augmentation optimization	107
6.8	Relative error in the execution cost estimation by Pyper and GPL for TPC-H and SSB queries on V100	109
6.9	Normalized measured and estimated execution cost of different candidate solutions of Q6 on V100	111
6.10	Overall performance evaluation of Pyper for TPC-H queries on the V100 GPU.	111
6.11	Overall performance evaluation of Pyper for SSB queries on the V100 GPU.	112
6.12	Time breakdown of Pyper and the Hawk-M on V100	112
6.13	Normalized (to Hawk-M) warp execution and occupancy rate for TPC-H and SSB queries on V100	114
6.14	Performance benefit of Shuffle and Segment operators for TPC-H queries on V100.	115
6.15	Impact of Grid Shuffle on P_1 for varying levels of thread execution efficiencies on V100.	116
6.16	The query plan augmentation process of Q6 (notations can be found in Table 6.1)	117

6.17 The query plan augmentation process of Q1 (notations can be found in Table 6.1)	118
6.18 Impact of selectivity on the benefit of Shuffle and Segment operator insertions into P_1 on V100	119
6.19 Performance evaluation on P100 and Titan Xp	119
6.20 Performance comparison of Pyper on V100 GPU against Omnisci CPU on Intel Xeon E5-2698 v4	120

List of Tables

1.1	Hardware comparison between GPUs and CPUs.	1
2.1	Overview of GPU architecture features across hardware generations.	9
4.1	Hash join implementations evaluated in this study.	35
4.2	Hardware used for experiments.	43
6.1	Notations in the cost model	98
6.2	Hardware used for experiments.	107
6.3	Resource requirements of Hawk-M and Pyper on V100	116

Chapter 1

Introduction

The explosive growth in the amount of data that needs to be analyzed [1] and the slow down of CPU performance growth over the last decade due to the breakdown of Moore’s law [2] has pushed specialized accelerators like GPUs to the forefront of high performance in-memory data analytics. While CPU performance growth has slowed down, GPU hardware has achieved over 45x increase in core count and over 23x increase in global memory bandwidth since 2008. Hence, GPUs have evolved as the hardware of choice for high performance compute applications like Online Analytical Processing (OLAP).

	Hardware Feature	GPU (NVIDIA Quadro V100)	CPU (Xeon Gold 6230R)
Compute	Core Count	5,120	26
	Base Clock (GHz)	1.1	4.0
	Boost Clock (GHz)	1.62	4.0
Memory	Cache Size (MB)	6	35.75
	Memory Type	HBM 2.0	DDR4-2933
	Memory Size (GB)	32	1024
	Memory Bandwidth (GB/s)	870	140
Other	Hardware Generation	Volta	Cascade Lake
	Price ¹ (USD)	8,000	2,500
	TDP (W)	300	150

TABLE 1.1: Hardware comparison between GPUs and CPUs.

To demonstrate the attractiveness of GPU hardware for high performance data analytics, we compare the key hardware capabilities of a modern CPU against a

modern GPU in Table 1.1. As shown in the table, a typical GPU offers over 6x higher global memory bandwidth and close to 200x more cores than CPUs. This use of higher bandwidth global memory and their massively parallel architecture design is ideal for OLAP systems that require the parallel processing of a larger number of data entries. Due to these reasons, we have witnessed significant efforts in the past decade to enable the use of GPUs in high performance data analytics operations [3–6].

Despite its numerous hardware advantages, designing and optimizing high performance database systems for GPUs is challenging due to the following reasons. First, due to their significant hardware differences compared to CPUs, database systems need to be re-designed from the ground up to ensure efficient use of GPU hardware. Second, the GPU hardware architecture has been evolving rapidly over the last decade. This, combined with the lack of in-depth studies on the impact of different GPU architecture features, makes it difficult to design database systems that take advantage of the right GPU architecture features. Third, the small global memory size of a single GPU often necessitates data access from remote CPUs or GPUs when processing large data sets. This makes it challenging to design efficient scalable GPU database systems due to the significantly higher cost of remote data accesses.

Numerous studies in the past decade have proposed GPU database systems [3, 7–9] that try to address the above inefficiencies. These studies focus on optimizing the performance of GPU database systems at two key levels: operator level and query level.

Operator level optimizations. Operator level optimizations attempt to improve the performance of individual database operators on single or multiple GPUs. In this regard, significant effort has been put into optimizing compute and memory intensive operators like join [10–12], aggregation [13–16] and sort [17–20]. A variant of the join operation, *partitioned hash join*, has especially received significant attention from the research community [10, 12, 21].

Existing studies on operator level optimizations focus on improving the execution efficiency of GPU database operators through more efficient parallel execution of GPU threads or by taking advantage of newer GPU architecture features. Despite the significant efforts from the research community, existing database operator

implementations are often inefficient on GPUs due to their inability to make efficient use of the modern GPU hardware. For example, existing GPU hash join implementations fail to make efficient use of GPU hardware when joining skewed input relations on a single GPU or when joining large input relations on modern multi-GPU architectures.

Query level optimizations. Query level optimizations aim to improve the overall query execution performance on GPUs. Earlier studies focused on improving the overall query execution performance through data layout optimization [9, 22], better data placement [11, 23–25] and data compression [26–29]. More recent studies have focused on improving the query execution performance by minimizing the cost of moving the intermediate data between the relational operators/GPU kernels. To achieve this, techniques such as efficient pipelining of relational operators [4, 30], dynamic fusion of GPU kernels [31] and just-in-time (JIT) code generation [3, 5, 6] have been explored.

Despite these efforts, modern GPU database systems still suffer from inefficiencies like severe resource contention [31] and high degree of workload divergence when executing analytical queries [6]. These inefficiencies are often rooted at the unique hardware characteristics of modern GPUs. Hence more research and development of novel execution strategies are necessary to address these inefficiencies.

1.1 Research Objectives

The objective of this thesis is to study and improve the performance of in-memory analytical query execution on modern GPU hardware. We aim to conduct in-depth analysis of the factors affecting the performance of in-memory GPU OLAP systems at both the operator and query level. Based on the insights gained from this study, we aim to develop novel techniques to improve the execution efficiency of analytical queries on GPUs. At the operator level, we aim to improve the performance of the hash join operator on a single GPU as well as modern multi-GPU architectures. At the query level, we aim to improve the overall execution efficiency code generation based systems on modern GPU hardware.

1.2 Research Contributions

The major research contributions of this work are as follows:

- **Revisiting Hash Join on Graphics Processors: A Decade Later.**

This work has been published in a conference [32] and an extended version has been published in a journal [33]. In this work, we study the impact of the GPU architecture features on the performance of the hash join operation and identify the factors guiding the choice of different features. Our analysis finds that 1) efficient use of native shared memory atomic instructions can help improve the occupancy rate of GPU kernels by up to 3.6x, 2) overlapping data transfer with the help of CUDA streams can improve performance by up to 2x and 3) the use of the new unified memory feature without pre-fetching of data can lead to up to 50% lower PCIe data transfer efficiency. We also analyze how data characteristics like skew and high match rate impact the performance of GPU hash join operation and propose techniques to improve the performance analytical systems when joining such data sets. The proposed techniques help avoid any performance degradation when joining data sets with high skew and as a result, achieve up to 1.27x performance improvement when joining data sets with a high match rate.

- **MG-Join: A Scalable Join for Massively Parallel Multi-GPU Hardware.**

This work has been accepted for publication in a conference. We develop MG-Join, a partitioned hash join implementation for modern multi-GPU architectures that makes use of an adaptive multi-hop routing policy. Numerous communication frameworks have already been proposed in the literature to enable efficient communication across the GPUs in multi-GPU setups. Groute [34], Gluon [35] and Lux [36] were proposed for handling workload distribution and communication across GPUs within a single machine for graph applications. Further, NCCL [37] is a communication framework from NVIDIA and is designed to allow efficient communication between GPUs. However, all these implementations adopt static routing policies which are highly inefficient on modern multi-GPU hardware, as demonstrated in the comparison with multi-hop and adaptive routing in our experimental studies. The proposed adaptive multi-hop data routing policy helps minimize congestion across shared interconnect links and ensures more efficient use of

GPU hardware and interconnect links. Overall, our experiments find that 1) MG-Join ensures up to 97% utilization of the GPU interconnect links, 2) achieves up to 2.5x performance improvement over existing multi-GPU hash join implementations and 3) helps improve the overall performance of TPC-H queries by up to 4.5x over multi-GPU version of Omnisci [7].

- **Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs.** This work has been submitted to a conference. We propose *Pyper*, a JIT compilation-based query processing system to resolve GPUs' divergence and resource contention issues. LLVM based systems like Voodoo [38], HetExchange [5], HAPE [39] and OpenCL based HorseQC [40] and Hawk [3] were proposed to generate optimized executable code for heterogeneous systems containing both CPUs and GPUs. However, these systems fuses together all the operators in a pipeline into a single kernel, leading to divergence and resource contention issues. Hence, Pyper introduces two new operators for query plan transformation, *Shuffle* and *Segment*, to improve GPU hardware utilization during query execution. Specifically, Shuffle is applied to a pipeline to reduce the execution divergence. Segment is used to split a pipeline that has overwhelming resource usage into several pipelines with lower resource usage but much higher thread parallelism. The benefits of the two operators come with overheads. Thus, we develop a cost model to guide the optimal use of Shuffle and Segment operators. Our cost model is capable of estimating the impact of divergence and resource contention, which have been overlooked in the previous studies [28, 41]. Our experiments show that Pyper is able to improve the performance of TPC-H and SSB queries on average by 1.60x and 1.52x, compared to Hawk [3]. Pyper further achieves 5.01x and 2.55x performance improvement over Omnisci [7] for TPC-H and SSB queries respectively.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, we present the background on GPU hardware and programming model along with a brief overview of the steps involved in executing analytical queries on GPUs. In Chapter 3, we review the related work on GPU database systems, at both the operator

and query levels. Then, in Chapter 4, we study the impact of different GPU architecture features on performance of the hash join operation on single GPUs and propose novel techniques to efficiently join data sets with high skew and match rate. In Chapter 5, we propose MG-Join, a partitioned hash join implementation for modern multi-GPU hardware that uses an adaptive multi-hop routing policy to ensure efficient use of both GPU and interconnect hardware. In Chapter 6, we propose Pyper to address the execution efficiency issues of existing JIT compilation based systems, by introducing two new operators (Shuffle and Segment) and an analytical model to determine the usage of the two operators in a cost-efficient manner. Finally, in Chapter 7, critical conclusions are drawn based on the research presented in this thesis and future research directions are discussed.

Chapter 2

Preliminaries

In this chapter, we present the background on GPU hardware and query processing on GPUs. There are two key vendors in the GPU hardware landscape: NVIDIA [42] and AMD [43]. While both vendors adopt similar overall hardware design for their data center GPUs, the terms used for similar technologies often differ significantly based on the vendor. NVIDIA being the leading hardware vendor for high-performance workloads, this thesis primarily uses the terms from NVIDIA.

2.1 GPU Hardware Architecture

Graphics Processing Units or GPUs were initially designed to accelerate graphics rendering. More recently, they have evolved as a powerful accelerator for parallel processing of analytical workloads. Each GPU consists of processing cores, registers, multiple cache levels and a global memory unit (the equivalent of main memory for CPUs). However, GPUs differ from CPUs in terms of the design of each one of these components.

Instead of a small number of complex cores available on the CPUs, GPUs use a large number of relatively simpler cores (referred to as CUDA cores in NVIDIA GPUs) that work in a SIMD fashion to process large amounts of data. Multiple CUDA cores (16, 32, or 64) are grouped into what is referred to as a *streaming multiprocessor* (SM) as shown in Figure 2.1. Since groups of 16 or 32 CUDA cores within an SM execute in a SIMD fashion, they share resources like instruction

FIGURE 2.1: GPU internal architecture.

cache, instruction buffer, warp scheduler, instruction dispatch unit, registers file, L1 data cache and shared memory. GPUs also have an L2 data cache and a global memory that are shared among all the SMs.

2.1.1 GPU Architecture Evolution

Being relatively new hardware compared to CPUs, GPUs architecture is evolving at a rapid pace. Over the last decade and a half, NVIDIA itself has introduced six different GPU architectures: 1) Tesla (2006), 2) Fermi (2009), 4) Kepler (2012), 4) Maxwell (2014), 5) Pascal (2016) and 6) Volta (2018). While the high-level design of these different GPU architectures have remained similar to Figure 2.1, each one of these generations has introduced one or more novel architecture features that have helped push the adoption of GPUs in high-performance analytical applications.

The architecture features that are key to query processing systems were designed to improve 1) inter-thread communication, 2) GPU resource utilization, or 3) efficiency of data movement. In the remainder of this section, we detail how novel GPU architecture features introduced in the past 15 years to help improve the query execution performance on GPUs. Table 2.1 summarizes the availability of these architectural features across different generations of GPUs.

	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta
CUDA Streams	×	✓	✓	✓	✓	✓
Shuffle Operation	×	×	✓	✓	✓	✓
Dynamic Parallelism	×	×	✓	✓	✓	✓
Unified Memory	×	×	✓	✓	✓	✓
Shared Memory Atomics	×	×	×	✓	✓	✓
Hardware Page Fault	×	×	×	×	✓	✓
High Bandwidth Memory	×	×	×	×	✓	✓

TABLE 2.1: Overview of GPU architecture features across hardware generations.

Inter-Thread Communication. NVIDIA introduced native shared memory atomic operations and shuffle operation for more efficient inter-thread communication. The atomic operations allow the threads within a thread block to modify shared data structures without conflicts. In contrast, the shuffle operation, which has significantly lower overhead than atomic instructions, allows the threads within the same warp to access each other’s local memory (registers). Shuffle and shared memory atomic operations make it possible for threads to collaboratively process the input data that is assigned to them (e.g., for joins, aggregations, etc.).

GPU Resource Utilization. To improve resource utilization, NVIDIA introduced support for multiple CUDA Streams and Dynamic Parallelism. Support for multiple CUDA streams makes it possible to simultaneously assign operations to both execution and DMA engines on GPUs. Dynamic parallelism allows kernels to launch child kernels, making it possible to alter the number of threads allocated for an operation dynamically.

Efficient Data Movement. The unified memory feature allows the GPU driver to move the data between CPU and GPU without explicit memory copy instructions. Further, hardware support for page faults allows more efficient and fine-grained movement of data across the PCIe bus. The page fault system manages a global page table that is updated every time a data entry is moved from CPU to GPU or vice versa.

To improve the efficiency of data movement within the GPU, modern GPUs have started using High Bandwidth Memory (HBM), which is located on the same physical package as the GPU die. The HBM memory uses a wider memory bus that operates at a significantly lower clock rate than traditional GDDR RAM. The wider

bus helps HBM to outperform GDDR memory, especially for coalesced global memory accesses. However, the lower clock rate and the resulting higher latency of HBM can negatively impact the performance of random global memory accesses.

2.1.2 Multi-GPU Architecture

While a single GPU itself is capable of accelerating modern analytical systems, its limited global memory size combined with the low bandwidth of CPU-GPU interconnects has held back their adoption in systems designed to process large scale data sets that do not fit within the GPU global memory [39].

GPU hardware vendors have been trying to address this limitation in the following ways. First, through the integration of a much larger number of GPUs into a single machine and providing a unified memory interface. This allows GPU applications like hash join to access a much larger high-bandwidth global memory pool. Second, GPU vendors have been introducing new high bandwidth and low latency interconnects such as NVLink 2.0. This allows applications executing on a GPU to access data more efficiently from remote GPUs [44].

In Figure 2.2, we present the hardware architecture of the DGX-1 machine [45], which is one of the modern multi-GPU system introduced by NVIDIA. As shown in the figure, modern multi-GPU architectures consists of multiple GPUs interconnected over a combination of PCIe 3.0, NVLink 2.0 and QPI interconnects. Each one of these interconnects has different access capabilities, efficiency, bandwidth and latency characteristics, as detailed below.

PCIe. Peripheral Component Interconnect or PCIe is a bi-directional serial communication bus that can be used to connect GPUs to either CPUs or other GPUs. A single PCIe bus consists of multiple bi-directional lanes with each Generation 3 PCIe lane capable of achieving a peak theoretical unidirectional bandwidth of 1 GB/s. Modern GPUs often have a single PCIe interface consisting of 16 PCIe lanes, thus achieving a net unidirectional bandwidth of 16 GB/s. Note, multiple GPUs could share the same PCIe bus (using a PCIe bridge), in which case the bandwidth of the bus will be divided among the GPUs during trying to access the bus simultaneously. For example, the Xeon E5-2698v4 CPUs in the DGX-1 server from NVIDIA only have 40 PCIe lanes, shared among up to 4 GPUs (64 lanes

FIGURE 2.2: Architecture design of DGX-1 machine from NVIDIA.

required for full simultaneous access). Finally, due to their lower bandwidth and efficiency compared to GPU global memory and other modern interconnects like NVLink [46], PCIe buses are often the bottleneck in modern multi-GPU architectures [44].

NVLink. The NVLink interconnect is based on the High-Speed Signaling Interconnect (NVHS)[47] and supports CPU-GPU and GPU-GPU communication. Note, currently only IBM CPUs support NVLink [48] for CPU-GPU communication. Unlike PCIe, NVLink is an exclusive P2P interconnect and cannot be shared by multiple GPUs. Further, NVLink is bidirectional and each link consists of 2 sub-links (one in each direction). Version 2 of the NVLink interconnect offers a bandwidth of up to 25 GB/s per link. Modern GPUs from NVIDIA supports 4-8 NVLink links per GPU, allowing each GPU to be directly connected to up to 8 GPUs/CPUs. Further, multiple NVLink links can also be used to interconnect the same CPU-GPU/GPU-GPU pair. Overall, NVLink offers significantly higher bandwidth and communication efficiency than PCIe. Despite this, the peak bandwidth offered by NVLink is more than an order of magnitude lower than GPU global memory bandwidth.

QPI. Intel’s Quick Path Interconnect or QPI is a P2P interconnect and is widely used for communication between processors in multi-socket/NUMA hardware. Similar to NVLink and PCIe, QPI is a bi-directional interconnect with each link supporting up to 25.6 GB/s bandwidth. Depending on the hardware generation, each processor could have multiple QPI links connecting it to other processors in the system. Even though QPI interconnects are not directly connected to GPUs, they

are required for moving data between GPUs that are connected to different CPU sockets using PCIe/NVLink in a multi-socket server like the DGX-1 from NVIDIA. Note, when GPUs connected to different CPU sockets need to communicate, the data cannot be transferred directly. Instead, the data is first moved from the source GPU to the main memory of the associated CPU socket (over PCIe or NVLink). Then, the data is moved to the CPU socket of the destination GPU over QPI and finally moved to the destination GPU over NVLink/PCIe. This process is referred to as *staging*. Note, staged data transfers are tiled and pipelined by the GPU driver to maximize efficiency. Despite this, staging does negatively impact the efficiency of data transfers and staged data transfers often fail to make efficient use of the interconnect hardware.

2.2 GPU Programming Model

A program executed by the GPU is called a *kernel*. A kernel can be launched for execution either by a CPU thread or by another GPU kernel. Kernel configuration parameters like the total number of parallel threads executing the kernel and its resource requirements need to be specified during kernel launch and cannot be modified dynamically. Further, each kernel is executed as a grid of thread blocks, which can be broken down into warps (groups of 32 threads).

2.2.1 Thread Execution Model

Each GPU thread block executing a kernel is assigned to a single SM in hardware. Typically, the instruction dispatch unit in the SM (Figure 2.1) issues a single instruction during each GPU execution cycle. All threads within a warp then execute this instruction in parallel in a SIMD fashion i.e., each thread within the warp executes the same instruction on different data items.

This SIMD execution approach means that an entire warp of threads can only execute a single instruction during an execution cycle i.e. only threads that can execute the same instruction will be activated during each cycle and the resources allocated to the other threads would be wasted. When different threads within the same warp follow different code paths, it is referred to as *divergent execution*.

Divergence among GPU threads can be a severe performance issue on modern GPU hardware.

Now, all the warps/thread blocks assigned to the same SM shares the hardware registers and shared memory within the SM. Hence, the number of warps/thread blocks that can execute simultaneously on a SM is limited by their resource requirements. This means that new warps/thread blocks can only be scheduled for execution on a SM if there are enough free hardware resources available. When sufficient resources are not available the GPU scheduler will fail to sustain sufficient parallel active threads necessary to ensure efficient GPU hardware use.

While the shared memory and registers in hardware are shared among all the threads executing on an SM, each thread can only access the register allocated to its warp and the shared memory allocated to its thread block. This means that the threads within each warp can access each others registers but cannot access the registers allocated to other warps within the SM. Further, the threads within each thread block can access a common shared memory buffer, while threads across different thread blocks cannot access each others shared memory buffers.

2.2.2 Memory Access Model

To enable fast data access by a large number of parallel threads, the GPU memory subsystem is designed to efficiently group memory instructions from the threads under execution. This grouping is implemented at two key levels: at the cache level and the global memory level.

At the cache level, memory accesses from each thread warp are grouped into a single 128-byte instruction. Each 128-byte instruction accesses 128 contiguous bytes in the GPU cache, allowing all the threads within the warp to simultaneously access the memory subsystem. Now, if threads within a warp require non-coalesced memory access (access to non-contiguous locations), multiple 128-byte instructions need to be issued by the warp.

When the data cannot be found in the GPU cache, the 128-byte instruction is broken down into 32-byte global memory instructions. Further, multiple 32-byte global memory instructions are grouped and issued within a single GPU global memory cycle, depending on the width of the GPUs global memory bus.

FIGURE 2.3: Steps involved in analytical query execution on GPUs.

Similar to the case of CPU hardware, accessing GPU global memory data is orders of magnitude slower than accessing data from GPU registers or cache. Hence, once a global memory instruction is issued by a warp, the threads are swapped out immediately and a new warp is brought in for execution. This aggressive context switching between warps allow GPU hardware to hide the overhead of global memory accesses more efficiently. Hence, the larger the number of active warps per SM (up to the hardware limit), the better is the GPUs ability to minimize the overhead of global memory accesses.

2.3 Query Processing on GPUs

As shown in Figure 2.3, processing an analytical query on GPUs involves the following three key steps: 1) parsing the high-level query and generating an intermediate representation, 2) optimizing the intermediate representation to generate an optimal execution plan and 3) execution of the optimized plan on the GPU hardware. In the remainder of this section, we briefly describe the working of each one of these stages.

Query Parsing. The first stage in processing an analytical query on GPU is the query parsing, carried out by the parser (Figure 2.3). The parser works by first breaking down the pieces of a high-level query statement into tokens that can be processed by other routines. The parser then performs a syntax check through which it ensures that all the tokens that are part of the query statement are valid. After this, the parser performs a semantic check by comparing the sequence of input tokens against a system catalog. Once the parser completes the syntax and

semantic checks, it generates an intermediate logical tree and passes it to the query optimizer.

Query Optimization. The query optimization stage in Figure 2.3 is responsible for optimizing the intermediate logical tree generated by the parser. The optimization process is carried out in the following two phases.

In the first phase, the optimizer generates a number of possible logical plans from the intermediate logical tree, each representing a possible way to execute the query. The generated logical plans consist of logical operators that can be executed to produce the desired results. The optimizer then estimates the cost of each logical plan in terms of the amount of work that needs to be performed for each plan and chooses the logical plan with the lowest cost. Note, the cost of each logical plan is estimated by taking the database schema into account.

In the next phase, the optimizer generates a physical plan from the chosen logical plan. This is done by mapping the logical operators in the logical plan to physical operator implementations supported by the execution engine. Each logical operator could have multiple physical implementations and hence the optimizer tries to choose the physical operator that minimizes the query execution cost. For example, the join operator could be executed using nested loop join, sort merge join or hash join implementations.

The optimizer determines the optimal physical implementation taking into consideration the resource requirements of the physical implementation and its overall execution cost. These metrics are computed based on the database schema and the characteristics of the underlying GPU hardware. Finally, the optimizer generates the optimal physical plan by choosing a physical operator corresponding to each logical operator such that the overall execution cost of the query is minimized. The generated physical plan is then sent to the execution engine.

Query Execution. In this stage, the physical plan generated by the optimizer is executed by the execution engine to produce the intended results. In addition to executing the individual physical operators, the execution engine is also responsible for managing the necessary compute and memory resources. This involves 1) allocating compute resources as well as memory buffers on CPUs and GPUs, 2) scheduling the execution of different physical operators on the GPU hardware based on the resource availability and 3) releasing the resources allocated to the

physical operators as well as the query itself once their execution is completed. The execution engine executes each physical operator in the plan by invoking the GPU kernels associated with the operator. Finally, the results generated by the query are presented to the user or passed to another process, depending on the design of the system.

Chapter 3

Literature Survey

Making efficient use of specialized hardware accelerators like GPU is key to improving the performance of database systems in the coming years. In this chapter, we review the related work on optimizing database systems for GPUs. While the focus of this thesis is GPU database systems, it is important to understand the evolution of database systems on CPUs since many design features from CPU database systems have been adopted by GPU database systems. Hence, we briefly review the literature on CPU database systems in Section 3.1. This includes systems optimized for modern multi-/many- core CPU architectures and distributed CPU clusters. Then in Sections 3.2 - 3.4, we perform a comprehensive review of the literature on optimizing query processing systems for single and multi-GPU architectures.

3.1 CPU Database Systems

Database systems for CPUs have existed as early as the 1980s. In the early days, due to the limited amount of Random Access Memory (RAM) available in hardware, database systems like MySQL [49], Oracle [50], PostgreSQL[51] and IBM DB2 [52] were optimized for accessing data from the disk. Disk access being a major bottleneck, database systems at the time relied on aggressive compression[53–56] and indexing[57–60] to minimize the associated overhead.

However, the explosive growth in the size of the available CPU main memory (RAM) in the early 2000's led to a complete re-thinking of the design of high-performance database systems. Insights such as "Memory is the new disk, disk is the new tape" from Jim Gray [61] helped database systems pivot to an in-memory design where a significant part, if not the entirety, of the data fits in CPU main memory.

After the pivot towards In-Memory Database Systems (IMDBS), the literature on CPU database systems focused on the following aspects:

Indexing. Despite access from main memory being significantly faster than disk accesses, efficient indexing is necessary to ensure optimal performance since the compute capability of modern CPUs often outperforms the capabilities of the main memory [62]. However, indexes had to be re-designed to focus on memory and cache utilization instead of I/O utilization, which was the case for on-disk databases. Several modern IMDBS systems make use of hash-based indexes [63–65] to ensure efficient memory and cache utilization. However, due to their lack of support for range queries, numerous tree-based indexes have also been proposed for IMDBS[66–73].

Data Layout. Earlier database systems often adopted a row-oriented data layout where all the data within the same row is stored in contiguous memory regions [49, 51]. Later, MonetDB [74–77] and C-Store [78] pioneered column-oriented data layout, which helped achieve orders of magnitude performance improvement over row-oriented systems due to more efficient use of CPU cache[79, 80] and compression techniques[81] for read-oriented analytical workloads. However, column-store systems encounter higher penalty for data updates and during execution (due to invisible joins). Hence, hybrid storage systems have also been proposed in the literature [82–84].

Parallelism. For CPU database systems, the literature covers three levels of parallelism: 1) data parallelism, 2) scale-up parallelism and 3) scale-out parallelism. Data parallel systems work by executing the same operation on multiple data items during the same cycle in a Single Instruction Multiple Data (SIMD) fashion[85–89]. Scale-up systems take advantage of the multi-core/multi-socket design of modern CPUs by partitioning the input data among different threads/processes[90–93]. Finally, scale-out systems try to make efficient use of

distributed CPU hardware through efficient data partitioning and minimization of cross-partition communication[94–98].

Execution Model. IMDBS systems for CPUs have evolved through the following three execution models:

- Earlier systems like MonetDB [74, 75] adopt a *volcano-/iterator-style* execution model where the tuples are pushed between operators using the *next()* function calls. In this model, each operator reads an input tuple, processes it and then pushes it to the next operator. Since traditional volcano-style systems implemented the *next()* function call at the granularity of individual tuples, they suffered from unnecessary overhead and inability to make efficient use of data locality.
- Later, Vectorwise [99] pioneered the *vectorized execution model*. Vectorized systems differ from traditional volcano-style systems by processing input tuples in batches, thus amortizing the cost of the *next()* function calls over a batch of tuples. Since the development of Vectorwise, systems like DB2 BLU [92], Quickstep [100], columnar SQL Server [101] and others [102, 103] have adopted the vectorized execution model.
- More recently, Hyper pioneered the *Just-In-Time (JIT) compile based execution model*, which was later adopted by other systems like Apache Spark [104], Peleton [105], HetExchange [5] and others [106–113]. These systems adopt a pull-based execution model for moving the data between the operators and works by generating low-level representations like LLVM IR for each query at runtime. Compared to vectorized systems, compiled systems achieve better code locality, more efficient use of hardware resources and avoid unnecessary materialization of data at operator boundaries [114]. Citing the difficulty in using low-level representations like LLVM IR, high-level language-based systems like LegoBase [115], DBLAB [116], and LB2 [117] were proposed. Systems like Weld [118] also proposed the use of custom intermediate representations (IR).

3.2 Database Design on GPUs

As discussed in Section 2, GPUs are integrated into modern hardware as highly parallel co-processors. Hence, optimizing the storage as well as access of stored data (Section 3.2.1), operator design (Section 3.2.2) and concurrency control (Section 3.2.3) are key to designing an efficient query processing engine for GPUs.

3.2.1 Data Storage & Data Access

Data Layout. As in the case of CPU database systems, the layout of the underlying data can significantly impact the performance of GPU database systems. Ghodsnia [22] studied the impact of row and column store layout on the performance of GPU database systems. The study found that column store layouts are uniquely suitable for GPUs due to its ability to 1) ensure more coalesced data access, 2) achieve better compression ratios and 3) minimize the amount of data that needs to be moved to the GPUs. Later studies [11, 119] arrived at a similar conclusion and hence modern GPU database systems [4, 9] have almost exclusively adopted column store layout.

Data Placement. Due to the small size of the global memory on a single GPU, the entire data set that needs to be processed often does not fit within the GPU. Hence, the placement of data is key to optimizing the performance of GPU database engines [120]. Schaa et al. [121] and Owens et al. [122] observed that on earlier generations of GPUs, the PCIe interconnect often becomes the bottleneck when the entire working set cannot fit within the GPU. Following similar observations, earlier studies proposed redesigning of GPU database systems to minimize PCIe data transfer [123–125]. StoreGPU [126] proposed the use of pinned, non-pageable host memory to minimize the impact of data transfer overhead on the overall execution. Gelado et al. [23] introduced an implementation that adopts lazy updates to move data to/from GPU global memory. Becchi et al. [24] presented an implementation that makes use of a model to avoid data migrations that encounter high overhead. Govindaraju et al. [25] designed a sort algorithm that specifically limits PCIe data transfers. Despite this, Bakkum et al. [127] demonstrated that GPU database systems could encounter up to 1.4x performance degradation when data is located on CPUs. Volk et al. [128] recognized that the data transfer cost of each data entry

needs to be amortized over multiple database operations for GPUs to become a viable alternative for high-performance query processing.

Based on these observations, GPU vendors like NVIDIA introduced the UVA feature, allowing GPU kernels to access data directly from the CPU main memory. Following this, Kaldewey et al. [11] designed a system that takes advantage of the UVA feature. Later, with the introduction of separate streams for data transfer and computation by GPU vendors, database systems adopted overlapping of data transfer and computation to minimize the impact of the data transfer overhead on the overall query execution [12, 129]. More recently, there have also been attempts to take advantage of the newly introduced unified memory feature to achieve more fine-grained overlapping of data transfer and computation [12, 44].

The recent development of high throughput, low latency Non-Volatile Memory Express (NVMe) Solid State Drives (SSDs) has made it possible for GPU database systems to directly access data from these devices using GPUDirect[130], without having to buffer the data on the CPU main memory. Zhang et al. [130] proposed a non-volatile memory management unit (NVMMU) that reduces the data transfer overhead by directly moving the data from SSDs to GPUs. HippogriffDB [131] adopted a highly compressed placement of data in NVMe SSDs to design a highly scalable GPU database system that trades off GPU cycles for the improved I/O bandwidth.

Recent years have also seen a scale-up of GPU enabled hardware by integrating up to 16 GPUs within a single machine [45, 132, 133]. While a single GPU often offers too little global memory for large scale database systems, a modern scaled-up GPU hardware offers up to 512 GB of collective GPU global memory. This, coupled with the introduction of high bandwidth, low latency interconnects such as NVLink 2.0 [134], has made it possible to design database systems where all frequently used data is stored in the GPU global memory [8, 44].

Data Compression. The high cost of data transfer to/from GPUs combined with the relatively small size of the global memory in a single GPU has made data compression essential to minimizing the data transfer cost between GPUs and other hardware devices [135]. Fang et al. [135] proposed several lightweight compression schemes for GPUs. These lightweight compression schemes often allowed vectorized processing of uncompressed data [136–138], making them highly efficient on GPUs.

While combining different lightweight schemes was known to be undesirable on CPUs, the high memory bandwidth and computation capability of GPUs made this combination of different compression schemes viable [139]. Hence, Przymus et al. [26, 27] proposed lightweight compression schemes and a planner to determine the optimal compression scheme dynamically based on incoming data.

Numerous works have also focused on efficient decompression of data since not all data can be processed without decompression. Funasaka et al. [29] presented a lossless data compression scheme that allows the decompression phase to be highly parallelized on GPUs. Rozenberg et al. [140] developed an open-sourced library of lightweight decompression algorithms for GPUs. Finally, Jiong et al. [28] proposed CPU-assisted decompression to improve query execution performance on coupled CPU-GPU architectures.

Indexing. Data indexing is key to ensuring efficient data access in database systems [141, 142] and has been demonstrated to be highly efficient for columnar databases [78, 143]. GPUQP [135], one of the earliest database systems for GPUs, supported both tree and hash based indexes. Volk et al. [128] proposed an index based speculative query processing engine for GPUs. Bohm et al. [144] proposed a novel GPU oriented index for high-performance density-based clustering. Gosnik et al. [145] proposed a parallel index structure that utilizes Data Parallel Bin-based Index Strategy (DP-BIS) on GPUs. The same study along with others [146] also explored the use of compression for bitmap indexes.

Citing the high maintenance cost of bitmap indexes, Kim et al. [147] proposed a fast architecture sensitive layout for tree-based GPU indexes. The study adopted a unique layout for binary trees to optimize data access. Finally, Kaldewey et al. [148] proposed a SIMD-based traversal of GPU tree indexes to further improve the performance of index tree traversal on GPUs.

3.2.2 Operators Design

Designing efficient parallel implementations of the relational operators is key to ensuring efficient query execution on GPUs. In the remainder of this section, we discuss the related work on key operators in modern GPU databases.

Selection/Filter

The selection or the filter operator takes as input a set of tuples and then outputs only those tuples that satisfy a specified criteria. Earlier systems like GPUQP [135] and OmniDB [149] executed the filter operator by first applying the filter criteria and then gathering the tuples that satisfy the criteria into an output buffer. Kernel Weaver [31] proposed a design where the final gather operation is not performed when the filter kernel is fused with other operator kernels. Later, GPL [4] adopted a similar pipelined design where the gather stage is skipped.

Now, most early GPU filter implementations [9, 44], made use of branching *if* statements while checking the filter criteria. More recently, compiled systems like Voodoo [38] and Hawk [3] proposed the use of software predication instead of branching *if* statements to avoid the cost of branch mispredictions on GPUs. However, the exact benefit of predication over branching depends on various factors, including the hardware configuration and the selectivity of the filter operator [44].

Due to the difficulty in processing variable-length strings on GPUs, most GPU database systems either only support filter operations on integer data or encode strings into integers for efficient execution of the filter operator [9, 30, 150–152]. There have been numerous studies [153–156] on the cache and memory efficiency of string matching algorithms on GPUs. Sitaridi et al. [157] proposed the splitting of string matching into multiple steps to minimize divergence. More recently, DogQC [6] detailed the high cost of importing data using dictionary encoding for strings and instead proposed the use of techniques like push-down parallelism and lane refill to minimize the divergence encountered by GPU threads when processing variable-length string data.

Join

The relational join operator is responsible for joining two input relations based on a predefined criteria. Three major join implementations for GPUs have been studied in the literature: simple hash join, partitioned hash join and sort-merge join. Due to the high cost of GPU global memory accesses, the simple hash join implementation is costly for data sets that do not fit within the GPU shared memory or L2 cache [41, 44, 158, 159]. Numerous studies [160, 161] have also concluded that, the

partitioned hash join is more efficient than sort-merge join on modern hardware, when processing large data sets. Due to these reasons, the partitioned hash join is the most popular and widely adopted join implementation on GPUs.

Hash join on CPUs has been studied exhaustively for decades [62, 160–166] and the design of most partitioned hash join implementations on GPUs is based on these earlier CPU implementations. Partitioned hash join on GPUs work by first partitioning both input relations based on the same key into small partitions that can fit within the GPU shared memory and then joining the co-partitions (partitions with the same identifier from each input relation).

The first partitioned hash join implementation for GPUs was proposed by He et al. [21] in 2008. However, the implementation does not take advantage of many modern GPU architecture features. In the same year, NVIDIA GPUs started widely supporting atomic operations in global memory. Following this, Pirk et al. [167] tried to accelerate foreign-key joins by executing random table lookups on GPU global memory. After the introduction of the UVA feature by NVIDIA, Kaldewey et al. [11] designed a system that takes advantage of UVA and lock-update-release based atomic instructions to share the histogram among multiple threads.

In 2015, Rui et al. [10] revisited the original hash join implementation and proposed improvements like the use of shuffle instructions to speed up the prefix sum operation on the histogram data. Later, Rui et al. [129] proposed a hash join implementation that overlaps the data transfer of the second input relation with the partitioning of the first input relation (using CUDA Streams). The implementation also proposed the use of the Dynamic Parallelism feature to join skewed input relations. Finally, Sioulas et al. [12] proposed an implementation that partitions the data using bucket chains, thus avoiding the need to build histograms. The same study also found that using the unified memory feature for moving the data between the CPU and GPU is highly inefficient.

Sort

Despite their higher complexity ($O(N \log^2 N)$), earlier sort implementations for GPUs [168–170] were based on bitonic sorting algorithms [171]. This is because bitonic sort implementation often requires less inter-thread communication and

can easily be mapped to SIMD architectures [172]. Later, GPU hardware improvements made other comparison sorts with lower algorithmic complexity ($O(N \log N)$) viable on GPUs. Leischner et al. [173] proposed a sample sort implementation for GPUs, capable of outperforming bitonic sort implementations at the time. Other implementations for GPUs based on merge sort [17–19] were also proposed around the same time.

Another comparison sort implementation is Quicksort and was first implemented on GPU by Sengupta et al. [174]. The implementation was later improved by Cederman et al. [175]. Dynamic Parallel Quicksort [176] by NVIDIA uses dynamic parallelism to execute Quicksort efficiently. Manca et al. [177] improved upon the implementation by Cederman et al. [175] and it outperforms most other implementations [20, 176].

While comparison sorts can be better mapped to SIMD architectures, other implementations like radix sort have lower algorithmic complexities [178]. Radix sort implementation for example, has significantly fewer data access operations [179]. Huang et al. [180] studied the impact of different algorithmic parameters on the performance of radix sort on different GPU architectures. An efficient radix sort algorithm for GPUs was proposed by Merrill et al. [20] and was later adopted by the NVIDIA thrust library [181].

Aggregation

One of the early implementations of the aggregate operator on GPUs was proposed by Govindaraju et al. [182]. The lack of hardware support for atomic operations on GPUs at the time made it challenging to design efficient aggregate operator for GPUs [183]. However, due to its massively parallel architecture, GPU implementations still managed to outperform its CPU counterparts. In 2010, Lauer et al. [13] proposed a fast multidimensional aggregation implementation based on the then recently introduced CUDA framework by NVIDIA.

Kernel Weaver [31] proposed fusing operators like aggregation with other operators to minimize the cost of reading the input data. Later, Karnagel et al. [14] performed a comprehensive comparison of hash vs. sort based group-by-aggregate operation in databases and found that the hash-based implementation performs significantly

better for a relatively small number of groups. Tome et al. [15] also studied group-by-aggregation on GPUs and found that while the support for atomic instructions in newer GPU hardware makes it easier to implement aggregate operator on GPUs, the high cost of atomic instructions in the global memory is often detrimental to these implementations. Gumber et al. [16] proposed an adaptive in-register aggregation with high group locality.

3.2.3 Concurrency Control

Due to the massively parallel design of GPU hardware, database implementations need to carefully manage concurrency at three key levels: data level, operator level and query level. Concurrency at the data level involves processing multiple tuples/data entries in parallel by the same operator. This has been discussed in detail in the previous section (Section 3.2.2). Hence, we focus on the operator and query level concurrency in this section.

Operator Level Concurrency refers to the simultaneous execution of multiple operators on the GPU. Traditional GPU hardware only supported the execution of a single kernel at a time. This, combined with the fine-grained design of older GPU database systems [9, 135, 184] meant that only a single operator was active on the GPU at a given time.

Later, both NVIDIA and AMD introduced support for concurrent kernel execution on GPUs, opening up the possibility of operator level concurrency. Following this, numerous multi-kernel optimization techniques were proposed to improve the utilization of GPUs [185–187]. In one of these studies, Ravi et al. [188] proposed kernel consolidation to enable space and time sharing on GPUs. However, these implementations were highly inefficient when each kernel has sufficient thread blocks to ensure high GPU occupancy. To address this, Kernelet [189] proposed slicing of kernels to create more time sharing opportunities and developed a performance model to guide the scheduling decision.

Following these studies, Paul et al. [4] demonstrated that a single relational operator is often unable to efficiently use all the hardware resources available on a single GPU. Taking advantage of the support for concurrent kernel execution, GPL improved the resource utilization of GPU hardware when processing relational

queries. However, the implementation did suffer from high synchronization overhead between operators running concurrently on the GPU hardware[190]. More recently, Dai et al. [191] proposed techniques to balance memory accesses of concurrent kernels by limiting the number of active memory requests from individual kernels.

The new generation of compiled database systems for GPUs have helped minimize the synchronization overhead encountered by pipelined systems [3, 5, 6]. This is because multiple operators are co-located within the same kernel for compiled systems, allowing them to synchronize between operators using lower overhead thread-block level synchronization intrinsics for operators within the same pipeline. Further, costly synchronization using GPU global memory is only required when encountering blocking relational operators.

Query Level Concurrency While it is possible to improve GPU hardware utilization by executing multiple operators concurrently, the achievable level of concurrency depends on the specific type of query being executed. Hence the literature has also explored the concurrent execution of different queries on GPUs. Kato et al.[192] proposed TimeGraph, a real-time GPU scheduler, to minimize performance interference for important GPU workloads in multi-user environments. Further, Wang et al.[193] proposed executing the kernels from different applications within a single GPU context to minimize the associated overhead.

On the database side of things, GPURT [194] was developed as a high-performance transnational engine that batches multiple queries into the same kernel. Wang et al. [195] demonstrated the low utilization of GPUs when executing only a single query for certain workloads. They proposed concurrent query execution and developed software support to address resource contention when sharing GPU resources. More recently, G-PICS [196] was developed as a framework for concurrent processing of a large number of spatial queries on GPUs. Finally, Yan et al. [197] proposed Harmonia, a high throughput B^+ -tree for concurrent query execution on GPUs.

3.3 Query Processing on Single GPUs

The research efforts into the design of GPU database systems can broadly be classified into three generations: 1) kernel based execution systems, 2) pipelined

execution systems and 3) JIT compilation based systems. In the remainder of this section, we briefly cover the details of major GPU database systems in each generation.

3.3.1 Kernel Based Execution Systems

Kernel based execution (KBE) systems [9, 11, 28, 135, 159, 198] process queries on GPUs by invoking the kernels corresponding to each physical operator in the query plan one after the other. Once invoked, each kernel is provided exclusive access to the GPU hardware. This is because these systems work under the assumption that a single GPU kernel can make efficient use of all the GPU hardware resources [4, 9]. Further, since a single kernel in KBE systems corresponds to a single physical operator in the query plan, the data is moved between the operators using GPU global memory. The producer operator generates the intermediate results that need to be pushed to the next operator and writes it to the GPU global memory. The consumer operator then reads the intermediate data from the GPU global memory.

While earlier generations of KBE systems [21, 198] were designed exclusively for the GPU hardware at the time, later implementations focused on the support for execution on a variety of parallel hardware devices. OmniDB [149] proposed a kernel-adaptor based design that uses extensible kernels based on an abstract parallel architectures model. To further enable portability, RedFox [184] proposed an Internal Representation (IR) of operators that make it possible to integrate machine-specific optimizers and operator implementations. More recently, Ocelot [9] proposed a parallel database engine based on hardware-oblivious operators that are compiled to executable machine code at runtime.

Paul et al. [4] demonstrated that, despite the effectiveness of KBE systems, they encounter high communication overhead and resource under-utilization on modern GPU hardware. This is due to 1) the use of GPU global memory for moving the data between the operators and 2) the exclusive use of the entire GPU hardware by a single kernel at any given time.

3.3.2 Pipelined Execution Systems

Pipelined execution systems were designed to address 1) the high overhead of intermediate data materialization and 2) the severe under-utilization of GPU hardware encountered by KBE systems [4]. To address these inefficiencies, pipelined execution systems take advantage of the concurrent kernel execution feature of modern GPUs. By concurrently executing multiple GPU kernels (corresponding the same or different relational operators) and moving the data between them using GPU L2 cache, pipelined systems are able to make more efficient use of GPU hardware and minimize the overhead of intermediate data materialization.

The exploration of pipelined execution on GPUs began with the proposal of a fine-grained pipeline based implementation for hash join on coupled CPU-GPU architectures by Jiong et al. [41]. This was based on similar studies on CPUs [199, 200]. Later, Wu et al. [31] proposed the fusion of GPU kernels of one or more relational operators (within the same pipeline) to minimize the communication overhead between the kernels. Paul et al. proposed GPL [4], the first fully pipelined database system for GPUs. GPL took advantage of the support for concurrent kernel execution and efficient data communication (*channel*) on modern GPUs. The study further developed an analytical model to guide the generation of the optimal pipelined query plan. More recently, Funke et al. [30] proposed solutions to efficiently execute all processing steps of a pipeline in a single pass over the data.

Despite their ability to reduce the cost of intermediate data materialization and improve GPU hardware utilization, pipelined systems suffer from the following inefficiencies. First, due to the small size of GPUs L2 cache, pipelined systems such as GPL require frequent synchronization between kernels executing concurrently, adding significant overhead to query execution. Second, they require a large number of GPU kernel invocations to enable tiled processing on the input data set, which adds unnecessary overhead to query execution.

3.3.3 JIT Compilation Based Systems

JIT compilation based systems (hereafter referred to as *compiled systems*) [3, 5, 39, 40] have evolved as the state-of-the-art query execution systems on GPUs. These systems work by generating the executable code for each query plan at runtime. For

this, compiled systems first break down each query plan into individual pipelines at blocking operators [201]. They then generate a single kernel for the entire pipeline of relational operators [3]. This helps compiled systems to reduce unnecessary materialization of intermediate data at operator/kernel boundaries without additional synchronizations and reduce the instruction count within GPU kernels.

LLVM based database systems like Voodoo [184], HetExchange [5], HAPE [39], OmniSci (formerly known as MapD) and OpenCL based HorseQC [40] and Hawk [3] were proposed to generate optimized executable code for heterogeneous systems containing both CPUs and GPUs. Both Voodoo [184] and Hawk [3] generates code for execute on a variety of parallel architectures. HetExchange [5] developed an abstraction for device-oblivious operators and exploited the heterogeneous parallelism of modern servers with multiple CPUs/GPUs. HorseQC [40] extended the operator-at-a-time approach by fusing multiple operators for better bandwidth utilization. An industrial startup OmniSci [202] developed its code generation following Hyper [201, 203]. More recently, DogQC [6] proposed techniques to address divergence in query execution on GPUs.

3.4 Query Processing on Multiple Processors

In this section, we focus on the literature on query processing systems designed for execution on a heterogeneous (hybrid of CPU and GPU) and multi-GPU architectures.

3.4.1 Heterogeneous Architecture

In its early days, GPUs were often used as co-processors [204] to accelerate a subset of relational operators instead of entire queries. This execution paradigm, known as Heterogeneous Query Processing, aims for efficient processing of queries on heterogeneous architectures by leveraging the best features of both GPUs and CPUs.

Earlier systems designed for such heterogeneous hardware often focused on efficient load balancing between CPUs and GPUs [205]. To achieve efficient load balancing between different hardware devices, the query optimizer was designed to identify

the sub plans or subset of operators of each query that is most suited for execution on GPUs [135, 206, 207]. Due to the data transfer between CPUs and GPUs being a major bottleneck in heterogeneous systems, database systems often chose operators for execution on GPU in a way that minimizes the data transfer between CPUs and GPUs[208]. Systems were also designed to further minimize the cost of data transfer by overlapping the execution of a GPU operator with a data transfer operation [119, 159]. This overlapping helps keep all hardware components busy and make more efficient use of the available hardware resources.

To further enable more closer integration between CPUs and GPUs, GPU vendors like AMD [43] designed hardware where the CPU and GPU are fabricated on the same chip [209]. These devices enable the GPUs to directly access the CPU main memory as well as the low level cache enabling a more fine grained integration between the two. Zhang et al.[210] proposed DIDO, a key-value store system for coupled CPU-GPU architectures that adopts dynamic pipelining. Jiong et al. [41] studied the execution of hash joins on such coupled CPU-GPU architecture and proposed a cost model to automatically guide decisions on workload distribution between the devices. Techniques like prefetching with the help of CPUs to minimize cache misses and decompression with the help of CPUs to improve query execution performance were also explored in the literature on coupled CPU-GPU hardware [28].

3.4.2 Multi-GPU Architecture

Traditionally, database systems tried to benefit from multi-GPU architectures by simply partitioning the workload among the GPUs in a shared nothing fashion [5, 7]. This was done to avoid the need for constant cross-GPU communication over the PCIe bus, which due to its low bandwidth could often become the bottleneck when processing data on multi-GPU architectures [46]. However, recent introduction of higher bandwidth interconnects like NVLink 2.0 [134] and PCIe 4.0 [211] has now made it possible to achieve higher bandwidth, lower latency cross GPU communication [44].

To enable more widespread adoption of such multi-GPU architectures, NVIDIA designed NCCL [37], a communication framework to that allows efficient circular communication between GPUs. Numerous communication frameworks have

already been proposed in the literature to enable efficient communication across the GPUs in multi-GPU architectures. Groute [34], Gluon [35] and Lux [36] were proposed for handling communication across GPUs within a single machine.

Recent studies on modern multi-GPU architectures have often focused on accelerating a single relational operator by allowing multiple GPUs to collaboratively process the input data set. Lutz et al. [44] studied the performance of simple hash join operation and well as selection-aggregation queries on such hardware. Guo et al. [212] proposed a hash join implementation for distributed RDMA clusters with GPUs. The implementation relies on CUDA communication APIs (which makes use of the direct routes between GPUs) for data transfer between the GPUs within the same node.

NVIDIA Rapids [213] is an open-source suite of data processing and machine learning libraries developed by NVIDIA that enables GPU-acceleration for data science workflows on multiple GPUs. RAPIDS was developed for integration into Python based data science workflows. While the underlying layer of RAPIDS is built on CUDA and provides incredible performance to data science workloads, it is not ideal for the high performance database systems which are optimized from the ground up (including data storage and memory management layer) for high bandwidth and low latency. Hence, RAPIDS is also not as efficient as the other database systems and techniques discussed in this thesis.

Chapter 4

Revisiting Hash Join on Graphics Processors: A Decade Later

4.1 Introduction

The high level of parallelism and the high memory bandwidth provided by GPUs make them attractive hardware accelerators for Online Analytical Processing [184, 214–217]. Over the last decade, a tremendous amount of research effort has been put into accelerating relational operators like hash join using GPUs [10–12, 21, 28, 31, 41, 129, 218, 219]. During this time, GPU hardware has made significant improvements through the introduction of many new architecture features.

Hence, we need to revisit GPU hash join implementations in conjunction with the GPU architecture improvements, as shown in the timeline in Figure 4.1. The timeline clearly shows the following trend: every 1-2 years, vendors like NVIDIA introduce new architecture features, and 2-3 years down the line, we see new GPU hash join implementations that outperform previous implementations by taking advantage of these new features. However, existing literature lacks a systematic revisit of GPU hash join implementations from this perspective. Further, given the trend in Figure 4.1, understanding the factors guiding the choice of different features is critical to future research and system designs. Hence, we revisit the hash

This work has been published in 2019 IEEE International Conference on Data Engineering Workshops (ICDEW). An extended version of this work has also been published in 2020 Distributed Parallel Databases.

FIGURE 4.1: Timeline of hash join implementations and GPU architecture features.

join implementations in the last decade and study them using different generations of GPU hardware.

GPUs are designed to process large amounts of data in parallel with minimal inter-thread communication. Hence, GPUs are often inefficient when processing unbalanced/skewed input relations. Other GPU hardware limitations like limited global memory size make it difficult to join data sets with high match rate due to possible overflow of output buffers. Still, there is a lack of studies on the impact of such workloads on the overall performance of hash join implementations. To address this, we study the impact of data characteristics like skew and match rate on the performance of hash join implementations and propose techniques to improve their performance for such data sets.

To summarize, the major contributions of this chapter are as follows. First, we revisit the major hash join implementations in the last decade and detail how these implementations take advantage of different GPU architecture features. Second, we study the impact of these architecture features on the performance of the hash join operation and identify the factors guiding the choice of different features. For this, we conduct in-depth experiments using three different generations of GPUs released over the last decade. Third, we study how data characteristics like skew and high match rate impact the performance of GPU hash join operation and propose techniques to improve the performance when joining such data sets. Fourth, we perform an in-depth comparison of the performance and cost-efficiency of GPU hash join implementations against the state-of-the-art CPU implementation.

Join	Paper	Description	Key Architecture Features
NOP	[158]	Hardware oblivious non-partitioned hash join.	CUDA Streams
PTH	[21]	Basic parallel radix hash join that partitions the data by building a per-thread histogram.	CUDA Streams
WSH	[32]	Same as PTH except the histogram is shared within the same warp using the shuffle instruction.	CUDA Streams Shuffle Instruction
BSH	[129]	Same as PTH except the histogram is shared within the same thread block using atomic instructions.	CUDA Streams Shared Memory Atomics Dynamic Parallelism
PRBC	[12]	Parallel radix hash join using bucket chaining.	CUDA Streams Shared Memory Atomics
UMJ	[12]	Same as PRBC except using the unified memory feature to move data between the CPU and GPU.	Unified Memory Hardware Page Fault Shared Memory Atomics
UMJ-PF	[32]	Same as UMJ except supports pre-fetching of data to the GPU.	Unified Memory Data Pre-fetching Hardware Page Fault Shared Memory Atomics

TABLE 4.1: Hash join implementations evaluated in this study.

4.2 Implementation Details

In this section, we introduce the representative hash join implementations used in this study. To conduct a comprehensive study of hash join operation on GPUs, we choose 7 different GPU hash join implementations. Table 4.1 provides a summary of these implementations along with the key architecture features used by each one.

NOP. This is the traditional non-partitioned hash join implementation [158] that adopts a hardware-oblivious design consisting of a build phase and a probe phase. The build phase generates a hash table using global memory atomic operations and the probe phase then probes the hash table using the second input relation. However, this implementation fails to take advantage of GPUs L2 cache and shared memory when the size of the hash table is larger than the L2 cache or shared memory. Hence, it is inefficient for large data sizes.

PTH, WSH & BSH. Implementations like PTH [21], WSH [32], and BSH [129] recursively partitions both input relations by building histograms in the GPU shared memory. This helps generate small co-partition pairs that fit within the

shared memory for the final join operation. In PTH, each individual thread maintains its own histogram data, while WSH and BSH are capable of sharing the histogram among multiple threads (up to 32 and 1024 threads respectively) using shuffle and atomic instructions.

Now, the shared memory requirements of the threads executing on an SM cannot exceed the shared memory available in hardware. Further, each partition requires a unique entry in the histogram data structure that is stored in the shared memory. Hence, the maximum number of partitions that can be generated in a single pass (P_{max}) by PTH, WSH, and BSH is given by Equation 4.1. Here S_{max} is the maximum amount of shared memory available to the threads allocated to the same SM in bytes, $\#T_{act}$ is the number of threads executing simultaneously per SM , $\#T_H$ is the number of threads sharing the same histogram (i.e., a unique histogram data structure is required for each group of $\#T_H$ threads), and $\widehat{H}_{sz}$ is the unit size of a histogram entry in bytes. The value of $\#T_H$ is 1, 32, and 1024 for PTH, WSH, and BSH implementations, respectively. Further, Equation 4.1 clearly shows that implementations with lower levels of shared histogram access (i.e. lower $\#T_H$) will have lower occupancy rates (i.e. lower T_{act}) for the same number of partitions (P_{max}).

$$P_{max} = \frac{S_{max} * \#T_H}{\widehat{H}_{sz} * \#T_{act}} \quad (4.1)$$

Once both relations are partitioned, the co-partitions are joined using a nested loop join. Further, to take advantage of separate DMA and execution engines, WSH and BSH simultaneously execute PCIe data transfer and unrelated kernels.

PRBC. The partitioning stage in PRBC [12] does not require the generation of a histogram. It instead makes use of bucket chaining, where empty buckets are allocated to partitions as needed. This allows PRBC to more efficiently overlap computation and data transfer when compared to BSH (Figure 4.2). This is because the partition kernel in PRBC offers sufficient computation to overlap the partitioning of each relation with the input data transfer of the same relation, which is not the case for the histogram build operation in BSH. Finally, PRBC joins each pair of co-partitions using nested loop join or NOP.

FIGURE 4.2: Execution timeline of PRBC and BSH.

UMJ & UMJ-PF. Unlike previous implementations, UMJ [12] and UMJ-PF [32] delegate the responsibility of data movement to the GPU driver using the unified memory feature. In UMJ, when a thread tries to access data that has not been moved to the GPU, the driver migrates the necessary page to the GPU using the page fault mechanism. Finally, in addition to using the unified memory feature, UMJ-PF enables prefetching of data (using *cudaMemPrefetchAsync* instruction) so that the data is available on the GPU before it is required by the kernel threads, thus avoiding the need for page table updates during kernel execution.

4.3 Handling Data Skew & Match Rate

In this section, we present novel techniques to more efficiently join input relations with high skew and high match rates. Note, in this study we study the impact of the skew in the distribution of the data values within a single relation. Now, the techniques proposed in this section can be used to improve the performance of all of the implementations in Section 4.2. We evaluate these techniques using the BSH implementation (Section 4.4) for simplicity.

4.3.1 Data skew

Problem: Skewed input relations can lead to severe workload imbalance as data is unevenly distributed across partitions. Existing implementations either dynamically launch more threads when skewed partitions are detected (BSH) or breakdown

Algorithm 1: Algorithm for generating the mapping between thread blocks and partitions, for efficient joining of skewed input relations.

Input : H_R, H_S : Histograms of the input relations

P_{max} : Total number of partitions

B_{max} : Total number of thread blocks

B_{size} : Size of each thread block

Output: M_p : The mapping between the partition ids and the thread blocks

```

1 Function GenerateMapping( $H_R, H_S, P_{max}, B_{max}, B_{size}, M_p$ )
2   /*Initialize arrays  $W$  and  $B$  of size  $P_{max}$  with all elements set to 0*/
3    $j \leftarrow 0$ ;
4   for  $i$  from 0 to  $P_{max}$  do
5      $W[j] \leftarrow W[j] + Work(H_R[i], H_S[i]);$ 
6     if  $W[j] > B_{size} - T$  then
7        $j \leftarrow i + 1$ ;
8    $W_{sum} \leftarrow \sum_0^{P_{max}} W[i];$ 
9   for  $i$  from 1 to  $P_{max} + 1$  do
10     $B[i] \leftarrow B[i - 1] + \frac{W[i-1] \times B_{max}}{W_{sum}};$ 
11    for  $i$  from 1 to  $P_{max} + 1$  do
12      if  $B[i] == B[i - 1]$  and  $Work(H_R[i - 1], H_S[i - 1]) > 0$  then
13         $M_p[i] \leftarrow M_p[i - 1];$ 
14      else
15        for  $j$  from  $B[i - 1]$  to  $B[i]$  do
16           $M_p[i].append(j);$ 

```

partitions into small buckets and distribute the buckets among thread blocks in a round-robin fashion (PRBC). However, launching additional kernels at runtime has significant overhead and breaking down partitions among multiple thread blocks is inefficient for uniform data sets [12].

Solution: To address the above inefficiencies, we adopt a lightweight approach that dynamically re-distributes the thread blocks among the partitions based on their size. For this, we analyze the histogram or bucket headers and then generate a mapping between the thread blocks and the partitions. Overall, our implementation assigns thread blocks to each pair of co-partitions based on the amount of work required for joining the co-partitions with respect to the total amount of work that needs to be done for joining the two input relations. as shown in Algorithm 1.

Algorithm 1 first computes the amount of work to be done for joining each pair of

co-partitions (Lines 3 - 7). Note, the amount of work for joining the co-partitions depends on the specific implementation (e.g. nested loop join and hash join will have different cost) and is calculated by the *Work* function. Further, W and B in Algorithm 1 are intermediate arrays that hold the amount of work and the number of blocks assigned to a particular partition. Now, if a specific co-partition does not have enough work necessary for a single block of thread (Line 6, where T is a pre-determined threshold), the work will be allocated to the thread block processing the previous co-partition(s). Note, only workloads from consecutive co-partitions are combined together as it is possible that the final join operation would be completed in multiple stages, to efficiently handle high match rate input relations (Section 4.3.2). Next, the number of thread blocks that needs to be allocated to each pair of co-partitions is determined based on the computed $W[i]$ values (Lines 9 - 10). Finally, a mapping between each partition i and the set of thread blocks allocated to it is generated by the code in Lines 11 - 16.

Figure 4.3 shows an example where thread blocks (T_0 - T_{10}) are allocated to a skewed relation by BSH and BSH (w/ Skew Handling). For the sake of simplicity, we assume only one relation has skewness in this example. Note, BSH (w/ Skew Handling) refers to an implementation developed by integrating Algorithm 1 into the BSH implementation. This means that unlike BSH which launches additional kernels for joining skewed input relations, BSH (w/ Skew Handling) dynamically re-distributes the thread blocks among the partitions based on their size (or level of skewness). One of the relations used in this example has five partitions ($P_0 - P_4$), containing 20, 40, 0, 80, and 60 tuples each. Further, for the sake of simplicity, we assume that the partitions of the other relation do not have any skew. We make the following observations from the example shown in Figure 4.3. First, BSH launches three additional kernels to join the skewed relation, while BSH (w/ Skew Handling) requires only a single kernel launch. Second, BSH allocates thread blocks to even empty partitions like P_2 , while BSH (w/ Skew Handling) re-distributes the thread blocks and avoids such inefficiencies.

FIGURE 4.3: Skew handling mechanism in BSH and BSH (w/ Skew Handling).

4.3.2 Match Rate

Problem: Relations with a high match rate can cause output buffer overflows when joining co-partitions. The existing solution is to use an additional pass to accurately predict the number of output tuples that will be generated by each pair of co-partitions. This information is then used to join only a sub-set of the partitions during each probe kernel invocation. Note, the set/group of partitions for each invocation is determined as the largest set of partitions whose output can fit within the global output buffer. However, this additional pass adds unnecessary overhead [129]. Further, simply determining the set of partitions by estimating the worst-case output size (from partition size) can lead to the joining of only a small set of partitions during each invocation, thus resulting in inefficient use of GPU cores.

Solution: To address the above problem, we adopt an approach that identifies the set of partitions for each invocation of the probe kernel through dynamic estimation of the number of output tuples that will be generated by each pair of co-partitions. This approach helps achieve higher occupancy rates without encountering output buffer overflows. Finally, we minimize the overhead of recovering from an incorrect estimation by detecting possible future overflows and terminating execution. Algorithm 2 presents the pseudo-code for our probe kernel that joins a subset of the

Algorithm 2: Modified probe kernel of BSH (w/ Estimation) for efficient handling of high match rate input relations.

Input : R, S : Input relations

P_b : Bit-string representing the status of each partition (whether it has been processed or not).

O_{max} : Max output buffer size

Output: O : Output buffer

```

17 Function Probe( $R, S, P_b, O, O_{max}$ )
18   if  $P_b == 0$  then
19      $X \leftarrow \frac{\sum_i^P S[i]}{O_{max}};$ 
20   /*Allocate temporary output buffer  $O_L$  in local/shared memory*/
21   /*Compute  $P$  using Equation 4.2*/
22    $index \leftarrow FirstPartitionToJoin(P_b);$ 
23    $count \leftarrow P;$ 
24    $ProcessedPartitionCount \leftarrow 0;$ 
25   while  $count > 0$  do
26     if  $P_b[index]$  then
27        $i \leftarrow i + 1;$ 
28       continue;
29     if  $O_{max} - O.size() > B_{act} \times \widehat{O}_b$  then
30       /*Probe smaller partition of co-partition pair  $i$  using the larger one
31       and materialize the output in  $O_L$  */
32        $pos \leftarrow atomicAdd(O.size(), O_L.size());$ 
33       if  $pos + O_L.size() \leq O_{max}$  then
34         /*Write  $O_L$  to  $O$  from index  $pos$  to index  $(pos + O_L.size())$ */
35          $atomicAdd(ProcessedPartitionCount, 1);$ 
36          $P_b[index] \leftarrow 1;$ 
37          $\widehat{O}_b \leftarrow Avg(\widehat{O}_b, O_L.size(), ProcessedPartitionCount);$ 
38        $count \leftarrow count - 1;$ 
39     if  $ProcessedPartitionCount == P$  then
40        $X \leftarrow X + 1;$ 
41     else
42        $X \leftarrow \frac{X \times ProcessedPartitionCount}{P};$ 

```

partitions during each invocation. Note that for the sake of simplicity, Algorithm 2 does not contain the skew handling logic discussed in Section 4.3.1.

Estimating Output Size. As mentioned before, BSH (w/ Estimation) avoids the need for the additional pass by estimating the number of partitions that could be joined by each *probe* kernel (P) without causing an output buffer overflow. As

denoted in Line 21, we estimate the value of P using Equation 4.2, where X is an integer (≥ 1), O_{max} is the maximum number of output tuples that can fit within the GPU global memory, and R^j (S^j) is the j^{th} partition of R (S). The initial value of X is estimated assuming an average selectivity of 1 (Line 19). BSH (w/ Estimation) then updates the value of X after every *probe* kernel invocation as shown in Lines 38 - 41, where P is the number of partitions assigned to the probe kernel and P' is the number of partitions joined by the kernel before an output buffer overflow was encountered.

$$\frac{\sum_{j=1}^P |R^j| * |S^j|}{X} \leq O_{max} \quad (4.2)$$

Recovering from Incorrect Estimation. Over-estimation of the value of P can lead to output buffer overflows. Further, when an overflow is encountered in existing implementations, the output buffer will have to be discarded, and the same partitions will have to be joined again. To avoid this, BSH (w/ Estimation) ensures that the output tuples of each pair of partitions are written into the output buffer only if there is enough free space for all the tuples generated by them (Lines 32 and 33). Finally, the output tuples of the remaining partitions are discarded, and these partitions will be assigned to the next *probe* kernel.

To further minimize the overhead, BSH (w/ Estimation) minimizes the number of partitions whose output needs to be discarded. BSH (w/ Estimation) achieves this by preventing new thread blocks from starting execution if the system detects a possible output buffer overflow before all currently active thread blocks could finish execution (Line 29). Provided the GPU kernel is not pre-empted, the GPU scheduler ensures that the thread blocks that are scheduled for execution are only removed from the SM once they complete processing of all the input tuples allocated to them. However, the GPU scheduler provides no guarantee of execution order across thread blocks that are executing concurrently. Hence, when each new wave of thread block is scheduled for execution, Line 29 ensures that there are enough free slots (O_{free}) in the output buffer to accommodate all the output tuples that will be generated by the thread blocks. Note here, $\widehat{O}_b$ is the average number of output tuples generated by each thread block (in the current kernel invocation) and B_{act} is the maximum number of thread blocks that can execute simultaneously on the GPU hardware (determined based on the resource requirements of the GPU

kernels). This approach helps minimize the unnecessary computations even in the initial invocation of the kernel where the selectivity is assumed to be 1.

4.4 Experimental Evaluation

	P100	Titan Xp	M60	K40c	Xeon E5 2640 v4
Generation	Pascal	Pascal	Maxwell	Kepler	Broadwell
Core Count	56 x 64	48 x 64	16 x 128	15 x 192	2 x 10
Clock Rate (GHz)	1.328	1.4	0.899	0.745	2.4
Memory Size (GB)	12	12	8	12	256
Memory Bandwidth (GB/s)	549	547.7	160	288	68
Memory Type	HBM2.0	GDDR5	GDDR5	GDDR5	4-Channel DDR4
L3 Cache / Shared Memory per SM	64 KB	64 KB	96 KB	48 KB	25 MB
Price ¹ (USD)	4259	2184	1029	1999	2 x 900

TABLE 4.2: Hardware used for experiments.

4.4.1 Experimental Setup

Hardware. To compare the overall performance of different GPU hash join implementations, we use the P100 GPU from NVIDIA. Further, to study the impact of different GPU architecture features on the performance of GPU hash join implementations, we use three other GPUs from NVIDIA: K40c, M60, and Titan Xp. Together, these GPUs cover three different generations of GPUs manufactured by NVIDIA. All GPUs are connected to the CPU via x16 PCIe 3.0 interface. We use a dual-socket CPU configuration with two Xeon E5-640 v4 from Intel for comparing GPU hash join implementations with its CPU counterparts. A summary of the hardware devices used in our experiments can be found in Table 6.2. Finally, we use CUDA 9.0 and gcc 4.8.5 with full compiler optimization (-O3) to compile the code and use the *NVProfile* tool to collect performance metrics.

¹<https://www.amazon.com>

Input Data. Following previous studies [10, 12, 129, 166, 218], we use a synthetic data set consisting of two relations R and S with a tuple size of 8 bytes (4-byte *key* and 4-byte *id*). The *key* values are generated as sequential integers and then randomly shuffled. We further set $|R| = |S|$ for all experiments and vary the total input size ($|R| + |S|$) from 2M to 256M tuples.

Experimental Outline. In Section 4.4.2, we present the overall performance comparison of the seven representative GPU hash join implementations, using the P100 GPU. Then, in Section 4.4.3 we evaluate the impact of using different GPU architecture features. We then evaluate the efficiency of the techniques proposed to handle data sets with high skew and high match rate in Section 4.4.4. Finally, we compare the GPU hash join implementations against the state-of-the-art CPU based partitioned hash join implementation [163] (referred to as *PRO*) in Section 4.4.5. Note, PRO is a one-pass parallel radix join with software-managed buffer (buffered writes from local to main memory) and non-temporal streaming (by-passing cache and directly writing to DRAM). For all our experiments, we present end-to-end throughput, assuming that the database initially resides in the CPU main memory. Note that, following previous studies [11, 12, 166], we define throughput as the number of input tuples processed per unit time.

4.4.2 Overall Comparison

To get a comprehensive understanding of the performance of hash join implementations, we present the throughput achieved by all seven implementations (Section 4.2) on the latest P100 GPU from NVIDIA, for data sizes from 2M to 256M in Figure 4.4. We now make the following observations from the results in Figure 4.4.

First, among the histogram-based implementations (BSH, WSH, and PTH), WSH achieves the best performance for small data sizes, while BSH outperforms both PTH and WSH by up to 3.9x for large data sizes (Section 4.4.3.1). Second, the PRBC implementation that adopts a bucket chaining scheme outperforms the best histogram-based implementation (BSH) by up to 1.36x and achieves over 5.3x performance improvement over the 2008 implementation (Section 4.4.3.2). Third, the use of unified memory for data movement (UMJ) leads to significant performance degradation. However, using a combination of unified memory and prefetching (UMJ-PF) helps to achieve performance similar to the PRBC implementation that

FIGURE 4.4: Performance of GPU hash join implementations on P100 GPU.

makes use of explicit memory copy instructions (Section 4.4.3.3). We dive deeper into each one of these observations in the next section.

4.4.3 Impact of GPU Architecture Features

In this section, we detail the reasons behind the observation made in Section 4.4.2 from the perspective of GPU architecture changes.

4.4.3.1 Atomic and Shuffle Instructions

To understand how the choice of Atomic (BSH), Shuffle (WSH), or simple arithmetic instructions (PTH) for histogram updates lead to significant performance difference, we need to look back at Equation 4.1. For the P100 GPU S_{max} is 64K and the value of $\widehat{H}_{sz}$ is 4 (integer data) for all three implementations. Based on this, to generate 1024 partitions in a single pass, PTH needs to set the number of active threads per SM ($\#T_{act}$) to 16; while WSH and BSH can achieve $\#T_{act}$ values of 512 and 2048 respectively. Note, the maximum $\#T_{act}$ value supported by the P100 GPU is 2048. Hence, the PTH implementation fails to utilize the GPU hardware efficiently and achieves significantly worse performance than WSH and BSH, in spite of its use of lower overhead instructions for histogram updates.

WSH, on the other hand, is able to achieve reasonable utilization of GPU hardware compared to BSH for small data sets. To demonstrate this, we present the

Data Size	2M	256M
WSH	90%	24.7%
BSH	90%	89%

FIGURE 4.5: Occupancy rates achieved by WSH and BSH.

FIGURE 4.6: Normalized performance of BSH and PTH on K40c and P100.

occupancy rates (the number of threads executing on the GPU hardware simultaneously as a percentage of the maximum number of threads that can execute simultaneously on the given hardware) achieved by WSH and BSH for data sizes of 2M and 256M in Figure 4.5. The results show that WSH is able to achieve the same occupancy rate as BSH for small data sizes. This, combined with the use of lower-cost shuffle instruction, helps WSH to achieve better performance for such data sets. However, as the input size increases (which requires larger partition counts), the BSH implementation is able to achieve significantly higher occupancy rates than WSH implementation (based on Equation 4.1).

Finally, to understand the benefit of the hardware support for shared memory atomic operations, we present the normalized performance improvement of the BSH implementations over the PTH implementation on the K40 and P100 GPUs (Figure 4.6). To limit the impact of other factors, we use thread configurations that allow BSH and PTH implementations to achieve similar occupancy rates on both hardware. The results in Figure 4.6 clearly show that the performance improvement of the BSH implementation over the PTH implementation is significantly higher on the P100 GPU compared to the K40c GPU. This is because the K40c GPU belonging to the Kepler generation lacks hardware support for shared memory atomic instructions. Hence, the BSH implementation encounters significantly

higher overhead per histogram update on the K40c GPU, when compared to the P100 GPU.

4.4.3.2 CUDA Streams

As shown in Section 4.2, both BSH and PRBC overlap data transfer and computation using multiple CUDA streams. However, the PRBC implementation is able to make more efficient use of CUDA streams and thus achieve significantly better overlapping data transfer and computation (Figures 4.2b and 4.2a). In fact, our analysis shows that the BSH implementation spends close to 25% - 30% of its execution time on input data transfers that cannot be overlapped with any partitioning stage kernel execution. The PRBC implementation on the other hand is able to break input relations into partitions as small as 8 MB (required to ensure maximum PCIe efficiency) and hence spends less than 1% of its execution time on data transfers that need to be completed before the partitioning stage kernels can begin execution. Note, during the second stage when the co-partitions are joined, both the BSH and PRBC break down the execution into multiple phases (each generating a small block of output), thus making it possible to overlap the output data transfers with the kernels joining the co-partitions.

To further demonstrate this, we present the throughput achieved by PRBC and BSH when the use of multiple CUDA streams is disabled in Figure 4.7. The results clearly show that when the use of multiple CUDA streams is disabled, BSH is able to achieve performance close to PRBC. Overall, the above observation means that, for small GPU resident data sets, both the BSH and PRBC implementations are capable of achieving similar performance despite their use of different partitioning techniques. However, for large CPU resident data sets, PRBC implementation achieves better overall performance.

4.4.3.3 Unified Memory & Hardware Page Fault

When using the unified memory feature on modern GPUs that support hardware page faults, the data is moved to the GPU global memory only when it is accessed by one of the threads (at the granularity of a page). Hence, it requires the updating of page table data during kernel execution. Further, since the page table is shared

FIGURE 4.7: Performance of BSH & PRBC on P100 GPU with CUDA Streams disabled.

FIGURE 4.8: Normalized performance of PRBC and UMJ implementations of the M60 and P100 GPU.

across all the GPU threads, page table updates lead to severe synchronization overhead and thread stalling/blocking. To minimize this overhead, the GPU driver groups together page table updates from multiple threads. However, it is not possible to group together the requests from all active threads. In fact, the UMJ implementation encounters over 6K unique page table updates for a data size of 256M tuples (collected from NVProfile). This is why the use of the unified memory feature leads to significant performance degradation in UMJ (Figure 4.4).

To further understand the impact of hardware page faults on the overall performance of the GPU hash join implementations, we present the normalized performance of the PRBC and UMJ implementations on the M60 and P100 GPUs (Figure 4.8). Note, we use the M60 GPU for this comparison as it supports unified memory feature, but lacks hardware support for page faults. Hence, the entire data will be transferred over to the GPU global memory by the GPU driver before the

kernel begins executions i.e., the UMJ implementation on the M60 GPU achieves no overlap of data transfer and computations. Finally, unlike the K40c GPU, the M60 has hardware support for atomic instructions, thus avoiding any inefficiencies associated with histogram updates.

Despite the lack of overlap in data transfer and computation, the results in Figure 4.8 clearly show that the performance improvement of the PRBC implementation over the UMJ implementation is lower on the M60 GPU, compared to the P100 GPU. This is because, due to the large number of threads getting blocked or stalled during page table updates, the UMJ implementation achieves less than 50% PCIe bus utilization when executing on the P100 GPU. While on the M60 GPU, the UMJ implementation is able to achieve over 95% PCIe bus efficiency as data is transferred in bulk and no page table data need to be updated during kernel execution. Note, the observation made here regarding the negative impact of hardware support for page table updates is specific to GPU hash join operation.

Finally, despite its use of unified memory, UMJ-PF is able to achieve performance similar to the PRBC implementation, even on the P100 GPU (Figure 4.4). This is because UMJ-PF avoids page table updates during kernel execution through prefetching of data (i.e., no page table updates during thread execution). Further, UMJ-PF is able to hide the cost of prefetching by overlapping this operation with the execution of unrelated threads. Note that, the performance of UMJ-PF is still lower than PRBC due to additional work that needs to be done by UMJ-PF to update the page tables during data prefetching.

4.4.3.4 High Bandwidth Memory

As mentioned before, the High Bandwidth Memory makes use of a wider memory bus operating at a significantly lower clock rate than traditional GDDR RAM. For example, the P100 GPU from NVIDIA uses HBM with a 3,072-bit bus operating at 715 MHz, and the Titan Xp GPU from the same Pascal generation uses a GDDR RAM with a 384-bit memory bus operating at 3 GHz.

Now, to understand the impact of HBM, we run a micro-benchmark containing two kernels that spawn 1,024 thread blocks (with 1,024 threads) each, on the Titan Xp and the P100 GPU. The threads in one kernel write to random global memory locations while the threads in the other kernel write to sequential global memory

FIGURE 4.9: Performance comparison of P100 and Titan Xp for global memory write operation.

FIGURE 4.10: Speedup achieved by the Titan Xp GPU over the P100 GPU for the BSH implementation.

locations (20 times each). Note that, in spite of their use of different memory technologies, both these GPUs have similar theoretical peak global memory bandwidths of around 550 GB/s (Table 6.2). However, the results in Figure 4.9 clearly show that the P100 GPU performs better than the Titan Xp GPU for sequential writes, while the P100 GPU performs significantly worse than the Titan Xp for random memory writes. This is because, while the memory hardware in the P100 GPU is able to make efficient use of the wide HBM bus during sequential writes, it fails to efficiently group together the memory instructions during random memory writes, leading poor utilization of the memory bus.

To further prove the above point, we present the speedup achieved by the P100 GPU over the Titan Xp GPU for the partitioning stage (dominated by random writes) and the matching stage (dominated by sequential writes) in Figure 4.10. The results show that for the majority of the cases, the P100 performs better for

the matching stage, while the Titan Xp outperforms the P100 for the partitioning stage in spite of its lower GPU core count and slightly lower peak global memory bandwidth (Table 6.2). This is because the significantly better performance of the GDDR hardware in the Titan Xp during random memory writes more than compensates for its lower GPU core count and slightly lower peak global memory bandwidth. However, during the matching stage, the P100 GPU is able to make efficient use of its wide memory bus, thus outperforming the Titan Xp GPU.

4.4.4 Handling Data Skew & Match Rate

To demonstrate the benefits of our skew handling technique, we present the throughput achieved by BSH and BSH(w/ Skew Handling) when joining a skewed data set (256M tuples) that follows the zipf distribution (z values from 0 to 1). Note that BSH addresses data skew by launching additional threads at runtime (using Dynamic Parallelism), while BSH(w/ Skew Handling) dynamically re-distributes threads blocks (Section 4.2). The results show that BSH encounters over 12% performance degradation as the skew increases, while BSH(w/ Skew Handling) shows absolutely no performance degradation even for very high z factor values. Note that BSH(w/ Skew Handling) performs better than BSH even for small z factor values due to the higher overhead of detecting skews in the BSH implementation.

To demonstrate the efficiency of the technique proposed to efficiently join input relations with high match rate, we present the throughput achieved by BSH, BSH(w/ Baseline Estimation) and BSH(w/ Estimation) when the match rate is increased from 100% to 800% (input size is 256M). As discussed before, BSH groups together partitions using an additional pass; while BSH(w/ Baseline Estimation) and BSH(w/ Estimation) group together partitions by estimating the possible output size. When compared to BSH(w/ Estimation), BSH(w/ Baseline Estimation) makes the worst-case estimation. The results show that BSH(w/ Estimation) outperforms BSH and BSH(w/ Baseline Estimation) by up to 1.27x and 2.5x, respectively. This is because, BSH(w/ Estimation) does not require an additional pass compared to BSH and makes better estimations than BSH(w/ Baseline Estimation), which helps generate larger groups. Note that the throughput achieved by all the implementations drop significantly as the match rate increases, due to the higher cost of materializing output data.

FIGURE 4.11: Performance comparison of BSH & BSH(w/ Skew Handling) under varying skew levels.

FIGURE 4.12: Performance comparison of BSH, BSH(w/ Baseline Estimation) & BSH(w/ Estimation).

Finally, since the techniques proposed in this study for efficient joining of input relations with high skew and match rate mostly only impact the performance of the probe stage, we expect similar or better performance gains when the techniques are implemented on the PRBC implementation. This is because, the probe stage implementation of both BSH and PRBC are relatively the same and PRBC outperforms the BSH implementation due to its more efficient overlapping of data transfer and computation during the partitioning stage (as detailed in Section 4.4.3.2).

4.4.5 Comparison Against CPU

Figures 4.13a and 4.13b show the overall throughput and cost efficiency of the PRBC and PRO implementations when executed on the P100 GPU and a dual-socket CPU machine using two Xeon E5-2640 v4 (Table 6.2), respectively. Note,

(A) Throughput

(B) Cost Efficiency

FIGURE 4.13: Comparison against CPU.

here the *Performance/Dollar* is calculated as the ratio of the throughput and the cost of the GPU hardware. The results clearly show that due to the significant hardware improvements in the last decade, GPUs are now capable of achieving significantly higher performance than CPUs (up to 2.3x) while also providing similar cost efficiency. Hence, GPUs are an ideal hardware choice when joining input relations that can fit within the GPU global memory.

4.5 Discussion

In this section, we first highlight the following major experimental findings from Section 4.4 .

- Modern GPUs support varying levels of inter-thread communication: and the choice of the right scheme depends on the workload as well as the hardware being used.
- The computational capability of GPUs has improved significantly over the years, while the PCIe bus has shown very minimal throughput improvement. Further, new faster interconnects like NVLink are currently not supported by CPU vendors like Intel and AMD. Hence, the movement of data between CPU and GPU is a major bottleneck and systems designers should favor implementations that can achieve maximum overlap of data transfer and computation.
- Through data prefetching, unified memory based implementations can achieve performance similar to implementation using explicit/manual memory copy while also minimizing the complexity of memory management.
- The demand for higher memory bandwidth (especially in machine learning applications), has lead to GPU vendors adopting the HBM memory over GDDR RAM. However, the lower clock rate of HBMs can lead to performance degradation, especially during the partitioning stage in hash join which requires large number of random memory accesses.
- Due to the significant improvements to GPU hardware over the last decade, GPUs are now able to achieve significantly higher performance than CPUs, while also ensuring similar cost efficiency. Hence, GPUs are an ideal choice for joining input relations that fit within the GPU global memory.

Further, as echoed throughout the study, careful adoption/tuning of GPU architecture features is essential to achieve high performance hash join execution on GPUs. For example, numerous parameters like the input tile size (for data transfer and computation overlap), partition count, pre-fetching size, output buffer size etc. need to be tuned to ensure efficient performance on GPUs. Further, the optimal values of these parameters often depend on the GPU hardware and the input data distribution. However, existing implementations still mostly adopt an ad-hoc approach to tuning these parameters. Hence, a study on designing an adaptive implementation that can self-tune these parameters could be an interesting future study.

Finally, the implementations in this study focus on making efficient use of the architecture features of modern NVIDIA GPUs. However, with OpenCL 2.0 and the latest Navi generation of GPUs, the specific architecture features used in this study are also available on GPUs from AMD. For example, command queues can be utilized to overlap data transfer and computation; atomic instructions can be used to allow multiple work items (thread equivalent in OpenCL) to collectively update shared data structures like the histograms; and shuffle / group reduce instructions can be used by the WSH implementation for updating the histogram. Hence, all the implementations used in this study can also be extended to execute on AMD GPUs. And barring any significant variation in the relative performance of different architecture features, we expect the observations made in this study to be valid on AMD GPUs as well.

4.6 Summary

Over the last decade, significant research effort has been put into studying and improving the performance of hash join operation on GPUs. During this time, GPU hash join implementations have managed to achieve tremendous performance improvements by taking advantage of new GPU architecture features. However, existing literature still lacks a systematic revisit of the hash join implementations from the perspective of GPU architecture changes.

To address this, I revisit the major GPU hash join implementations in the last decade and detail how they take advantage of different GPU architecture features. Our study finds that, by efficiently taking advantage of GPU architecture features like CUDA Streams and shared memory atomic operations, PRBC outperforms all other existing GPU hash join implementations. The study also sheds light on the impact of different architecture features on the the hash join operation and has identified a number of factors guiding the choice of different GPU architecture features. Finally, the techniques proposed in this study help avoid any performance degradation when joining data sets with high skew and achieves up to 1.27x performance improvement when joining data sets with high match rate.

Chapter 5

MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures

5.1 Introduction

The use of higher bandwidth global memory and the availability of larger number of parallel cores have made GPUs the ideal accelerator for in-memory relational query processing [10, 12, 32]. Typically, a single GPU offers 10x higher global memory bandwidth and 100x more cores than a CPU. However, the limited global memory size of GPUs combined with the low bandwidth of CPU-GPU and GPU-GPU interconnects has held back their adoptions for large data sets that do not fit within the global memory of a single GPU [39]. In recent years, we have witnessed the rise of emerging multi-GPU architectures in a single machine. First, the integration of multiple GPUs (up to 20) into a single machine means that it is possible to have hundreds of gigabytes of GPU memory. For example, a machine with 16 GPUs each containing 32 GB of global memory offers a total of 512 GB GPU memory, which is sufficient for many online analytical workloads. Second, the introduction of new high-bandwidth and low-latency interconnects such as NVLink 2.0 significantly resolves the communication overhead among GPUs.

This work has been submitted to SIGMOD 2021.

The scale-up multi-GPU architecture has enabled new opportunities of relational query processing on multiple GPUs. Overall, the multi-GPU architecture has paved the way for GPUs becoming primary computing devices for relational query processing on large data sets. Hence, in this study we focus on scenarios where the GPU is being used as a primary computing device and the input data is distributed over the GPUs. In spite of many previous studies on query processing on a single GPU [9, 10, 21, 28, 41, 158, 159, 167], there has been little work on studying databases, especially hash joins on these multi-GPU architectures.

To demonstrate this, we study the performance of hash join, whose input relations are equally distributed to multiple GPUs. Specifically, we study two hash join designs for multi-GPU architectures, including UMJ [32] and DPRJ [212]. UMJ makes use of the NVIDIA unified memory feature to move the data between the GPUs, and DPRJ is designed based on the current state-of-the-art join implementation available for distributed GPU architectures. We present the GPU cycles per tuple along with the breakdown for computation and data transfer when executing UMJ and DPRJ on the DGX-1 server from NVIDIA [45]. We use two input relations (with 8-byte tuples) and a join selectivity of 100% for this test. When we increase the number of GPUs, we increase the size of both input relations by keeping 512M¹ tuples of each relation on each GPU. We study the cross-GPU communication and computation time components for DPRJ from NVIDIA GPU profiler (denoted as “DataTransfer“ and “Computation“, respectively), and we were not able to get such information from profiler for UMJ. More experimental details can be found in Section 5.4.

The results of this study are shown in Figure 5.1. Both UMJ and DPRJ encounter poor scalability when the number of GPUs is increased from one to eight. In fact, the DPRJ implementation only achieves a 2.13x improvement when the number of GPUs increases from one to eight. This is due to the severe increase in the communication overhead with the increasing number of GPUs, which accounts for up to 66% of the total execution time. Our detailed study further shows that it achieves lower than 50% utilization of the bisection bandwidth of the interconnects (details in Section 5.4). UMJ on eight GPUs is even slower than on a single GPU. *Overall, the results clearly demonstrate that existing partitioned hash join implementations*

¹M = 1,048,576

FIGURE 5.1: Join performance and execution time break down of partitioned hash join on DGX-1 server from NVIDIA.

encounter high communication overhead and achieve poor scalability when executed on modern multi-GPU architectures.

To understand the reason behind this inefficiency, we study the architecture design of the DGX-1 machine in Figure 5.2, and make the following observations. First, GPUs in the DGX-1 machine are connected over a combination of PCIe 3.0, NVLink 2.0 and QPI interconnects, resulting in a highly heterogeneous interconnect network design. This highly heterogeneous interconnect design leads to the data movement between certain pairs of GPUs becoming a bottleneck when performing operations like hash join on GPUs. Second, slower PCIe links are shared across multiple GPUs causing severe congestion when multiple GPUs try to gain simultaneous accesses to the same PCIe interconnect hardware. For example, when joining the input relations over 8 GPUs the PCIe 3.0 links between the GPUs with a bandwidth of 16 GB/s becomes the bottleneck.

It is a non-trivial task to reduce the communication cost of hash joins on such multi-GPU architectures. The communication overhead encountered by the hash join implementations on modern multi-GPU architectures depends on many factors like 1) hardware configuration, 2) input data distribution and 3) the data transfer characteristics (e.g., size of data flows). Determining the optimal data transfer schedule and route for each data flow given a set of constraints can be reduced to

FIGURE 5.2: Architecture design of DGX-1.

the classic multi-commodity flow problem, which cannot be solved in a reasonable amount of time, even for eight GPUs [220–222].

In this study, we propose MG-Join, a scalable partitioned hash join implementation that is optimized for modern multi-GPU architectures. To ensure efficient distribution of communication across the interconnect links, MG-Join adopts a multi-hop routing that is capable of routing data flows over intermediate GPU nodes. Further, through the use of an adaptive policy that takes into consideration the hardware configuration, the input data distribution and the characteristics of different data flows, MG-Join is able to dynamically re-route data flows to minimize the congestion.

To summarize, the major contributions of this work are as follows:

- We argue that multi-GPU architectures enable new opportunities for scale GPU-based query processing on hundreds of gigabytes of GPU memory. However, we study the performance of partitioned hash joins on multi-GPU architectures, and demonstrate that the cross-GPU communication overhead can severely hinder the scalability.
- We develop MG-Join, a scalable partitioned hash join implementation for modern multi-GPU architectures that makes use of the proposed adaptive multi-hop routing policy. The adaptive multi-hop data routing policy helps minimize congestion across shared interconnect links and ensures more efficient use of GPU hardware and interconnect links.

- We further conduct in-depth experiments to demonstrate the efficiency of MG-Join in joining large data sets. Our experiments find that 1) MG-Join achieves up to 97% utilization of the bisection bandwidth of the GPU interconnect links, and 2) MG-Join achieves up to 2.5x performance improvement over existing multi-GPU hash join implementations [32, 212]. Moreover, MG-Join helps improve the overall performance of TPC-H queries by up to 4.5x over multi-GPU version of an open-source commercial GPU database, Omnisci [7].

The rest of this chapter is organized as follows. In Section 5.2, we present the design overview of MG-Join. We present the details of the novel data distribution step in MG-Join in Section 5.3. Finally, we present the experiments in Section 5.4.

5.2 MG-Join: Design Overview

In this section, we present the design overview of MG-Join, a novel partitioned hash join implementation designed for modern multi-GPU architectures. MG-Join is designed for joining large input relations that are distributed across multiple GPUs in the same machine.

5.2.1 Design Rationales

We have the following rationales when designing MG-Join for modern multi-GPU architectures.

Rationale 1: To ensure efficient data transfer between GPUs, we make use of an adaptive and multi-hop data routing policy that is capable of dynamically re-routing each data flow over the most efficient route between each source and destination GPU pair. The decision is adaptive to each data flow to reduce the congestion that happens at runtime.

Rationale 2: MG-Join breaks down data flows of cross-GPU communication into fine-grained packets of a cost-effective size. On the one hand, a packet can be transmitted as soon as it is generated. This is to maximize the overlap between computation and data transfer. On the other hand, the fine-grained packet size

can reduce the pressure of memory usage on the GPU in our proposed multi-hop data transfer.

Rationale 3: We decide to compute a histogram at the beginning of the join to gain the data distribution associated with partitioning information. On the one hand, the histogram computation is rather lightweight, in comparison with cross-GPU communication. Paul et al. [32] has demonstrated that the histogram based partitioning is capable of achieving similar performance to the histogram-free approach [12] when the data is already available on the GPU. On the other hand, histograms are useful in the optimization of multiple phases in our proposed join design.

Rationale 4: Last but not the least, we avoid reinventing the wheel and reuse/adapt state-of-the-art implementations of GPU accelerations in hash joins whenever appropriate. We carefully select the right optimization and design for MG-Join, given that a lot of previous studies have been conducted on GPU-based hash joins [9, 10, 21, 28, 41, 158, 159, 167]. For example, we choose the histogram-free local join [12] on each GPU to maximize the overlap between communication and computation, while we choose to generate the histogram at the beginning of MG-Join.

5.2.2 Design Overview

In the MG-Join implementation, joining two large input relations, R and S , distributed over multiple GPUs involves the following four phases: *Histogram Generation*, *Global Partitioning Phase*, *Local Partitioning* and *Probe*.

Histogram Generation: During the histogram generation phase, each GPU analyzes the input data set that is allocated to it and builds a histogram based on its data. Each entry in the histogram data structure represents the number of tuples in a specific partition and is later used for partitioning the input data among the GPUs.

Although the relations could be partitioned among the GPUs without generating a histogram [12], we choose to generate a histogram according to Rationale 3. Histograms have two major benefits on reducing the communication cost. First, the generation of a histogram before the actual partitioning makes it possible to

overlap the compute-intensive partition assignment and the memory-intensive data partitioning during the global partitioning phase. Second, the histogram serves as a simplified bloom filter that avoids unnecessary data transfers during the global partitioning phase.

For generating the histogram, we use the same implementation as Rui et. al. [10] where the histogram is built using the GPU shared memory. This is done to ensure that all the threads have fast access to the histogram in the shared memory. This also means that the maximum number of partitions (P_{max} , also equal to the number of entries in histogram) is limited by the size of the GPU shared memory per SM (M_s), the size of each histogram entry ($\widehat{H}_s$) and the number of thread blocks executing on each SM of the GPU (T_b) as shown by Equation 5.1.

$$P_{max} = \lfloor \frac{M_s}{\widehat{H}_s \times T_b} \rfloor \quad (5.1)$$

In contrast with the previous studies [9, 10, 28, 41], MG-Join generates the largest number of partitions that can be generated based on Equation 5.1. This decision comes with two benefits. First, it helps achieve a more efficient balancing of workload when processing skewed input relations. Second, it helps reduce the amount of work that needs to be done during the local partitioning stage. For example, in a V100 GPU equipped with 64 KB shared memory and requiring at least two thread blocks per SM to ensure efficient use of GPU resources, the histogram kernel can maintain 4,096 4-byte histogram entries in the GPU shared memory.

Global Partitioning Phase: The global partitioning phase partitions the input relations to ensure that each input tuple is assigned to one or more GPUs and none of the GPUs requires the access of tuples from a remote GPU during the local partitioning and probe phases. This phase involves three steps: 1) the initial partitioning of the input relations on each GPU, 2) assigning the partitions of each GPU to one or more GPUs, and 3) the distribution of the partitioned data based on the partition assignment. In the following, we present more details for 1) and 2), and leave 3) to Section 5.3.

In Step 1), for partitioning the input data based on the generated local histogram, we use the implementation proposed by Rui et al. [10]. Overall the implementation works as follows. Each thread executing the partitioning kernel reads a unique

input tuple, determines the index for the tuple after partitioning based on the histogram and then writes the tuple into the local buffer at the determined index. After partitioning, all the tuples belonging to the same partition will be co-located in the local GPU global memory. We refer readers to the original study [10] for a better understanding of our partitioning implementation.

In Step 2), for the problem of partition assignment, multiple previous studies have explored the optimal solution for the similar problem in distributed CPU nodes [223–225]. In MG-Join, we adopt the migration and selective broadcast based approach previously proposed by Polychroniou et al. [223]. For each partition, the implementation works by first migrating the tuples of one of the input relations to a subset of the GPUs and then selectively broadcasting the tuples of the other relation to all the nodes containing the tuples of the other relation. We refer the readers to the original implementation [223] as well as the proof of cost optimality for the migration and selective broadcast based system.

Specifically, we have the following three major modifications over [223]. Overall, the modified design takes advantage of the massively parallel architecture of GPUs and quickly computes the network optimal partition assignment.

First, we overlap the partition assignment of Step 2) with the partition generation kernel of Step 1) to completely hide its overhead.

Second, in MG-Join, each GPU computes the assignment for all the partitions in parallel based on the histogram. Within each GPU, the partition assignment computation is further parallelized by allowing different warps (groups of 32 threads) to determine the assignment for different partitions in parallel. The assignment for each partition is determined by estimating the benefit of all possible migrations of each relation and then choosing to migrate the relation that achieves the maximum benefit from migration.

Third, as in the original implementation [223], the benefit of migrating the tuples of a relation from a source GPU to a destination GPU is computed as the difference in the cost of broadcasting all the tuples of the other relation to the source GPU and the cost of migrating the tuples from the source GPU to the destination GPU [223]. In MG-Join, the cost of moving a tuple from one GPU to another is computed as the cost of moving the tuple over the lowest transmission cost path between the GPUs when there is no congestion.

Local Partitioning: In this phase, the partitions generated during the global partitioning phase are further partitioned. As detailed in previous studies [10, 12], joining co-partitions on GPUs achieves the peak performance when at least one of the co-partitions fit within the GPU shared memory. Thus, multiple partitioning passes may be required to generate very fine-grained partitions that can fit within the GPU shared memory. Specifically, the data on each GPU received during the data distribution phase is recursively partitioned until the individual partitions are small enough to fit within the GPU shared memory (unless both relations are heavily skewed). To leverage the skew handling techniques, we make use of the partitioning approach proposed by Panagiotis et al. [12].

One remark on choosing the approach by Panagiotis et al. [12] is that this approach does not require the generation of a histogram (unlike that we generate a histogram before the global partitioning phase) and can hence help achieve more overlap between the data distribution and the local partitioning phase. This demonstrates Rationale 4 that we need to carefully revisit existing GPU-based hash join techniques and choose the suitable ones. Specifically, we start the execution of the data distribution and local partitioning phases together. The local partitioning kernel first begins partitioning the locally available tuples (i.e., tuples that were assigned to the same GPU), followed by the packets arriving from remote GPUs. Since the local partitioning phase does not require a histogram, the packets from the remote GPUs can be processed as soon as they arrive at the destination.

Probe: During the probe phase, the local co-partitions with the same partition id from both R and S relations are joined together. The co-partitions can be joined using nested loop join or hash join (where a hash table is built in the shared memory). Existing literature [12] has demonstrated that both implementations achieve similar performance for most partition sizes. Hence in MG-Join, we simply use the nested loop variant.

5.3 Data Distribution

We present the design and implementation details of the data distribution mechanism of the global partitioning phase in MG-Join. In the data distribution step of global partitioning, the cross-GPU communications generate data shuffling flows

among almost all GPU pairs. The design is rooted at Rationales 1 and 2, presented in Section 5.2.1. Each data flow is transferred at the granularity of packets, whose route will be determined by the proposed multi-hop adaptive routing policy. We first present the design and implementation of an efficient multi-hop routing, followed by the adaptive routing policy for determining the links in the multi-hop routes.

5.3.1 Multi-Hop Data Transmission

Existing multi-GPU communications (e.g., [34, 226]) adopt single-hop routing that simply chooses the direct route with highest bandwidth (a route that does not require the data to pass through any intermediate GPU). Note, in this study, we consider data transfers that are staged in the CPU main memory (when the source and destination GPUs are connected to different CPU sockets) as direct transfers since there is no intermediate GPU involved in the data transfer.

As shown in Introduction, the single-hop routing strategy adopted by existing multi-GPU join implementations can often lead to severe congestion in the interconnect links in modern multi-GPU architectures. Thus, in this thesis, we propose to use multi-hop transmission for multi-GPU architectures, which offers the flexibility of avoiding congestion and choosing more efficient routes. For each data flow from the source GPU to the destination GPU, data are sent in packets. To multi-hop routing, each packet contains a header with the packet id (4 byte), packet size (4 byte) and a vector of GPU ids (1 byte per entry) representing the route that needs to be taken by the packet. In our experiments, we set the packet size to be 2MB so that it can fully utilize the link bandwidth. The header only adds a negligible overhead per packet.

Challenges: It is challenging to perform efficient multi-hop data transmission on multi-GPU architectures due to the lack of dedicated routing devices and the limited amount of global memory available on individual GPUs. Due to these hardware limitations, each GPU needs to allocate a certain amount of its global memory for routing data packets. The larger the amount of GPU global memory allocated for data routing, the smaller the input data set that can be processed by the GPU. Hence, an efficient multi-hop implementation should be designed to reduce the costs in 1) synchronization across multiple data flows and 2) dynamic

FIGURE 5.3: Multi-hop data transmission example in MG-Join.

and efficient memory allocation on GPUs. In the remainder of this section, we detail how MG-Join implements multi-hop routing.

Figure 5.3 shows an example of multi-hop routing in MG-Join when routing two data flows ($F_{0→2}$ from GPU 0 to GPU 2 and $F_{0→3}$ from GPU 0 to GPU 3) on a machine with four GPUs. Now, $F_{0→2}$ and $F_{0→3}$ take routes $0 → 1 → 2$ and $0 → 1 → 3$, respectively. As shown in the figure, MG-Join adopts a push based routing policy where a source GPU along the route pushes the packet to the next destination GPU. Each GPU has two separate modules attached to it as shown in Figure 5.3: a sender and a receiver. The sender module is responsible for sending out the data packets that have been stored locally; while the receiver module is responsible for unpacking the received packets destined for that GPU (if it is the final destination of the route). Further, both sender and receiver units at each GPU together enables forwarding packets destined for other GPUs as shown in Figure 5.3.

Implementation details: At each GPU, we maintain multiple outgoing queues (one for each neighbouring GPU) from which the sender module can pick up packets for transmission. Now, each sender picks up packets from one of the queues in a weighted round-robin fashion. The weight for each queue is set as the number of waiting packets in the queue normalized to the total number of packets waiting to be sent out at the GPU. Further, once a queue is chosen, the sender picks up a batch of packets that follow the same route to the destination and then begins the transmission of these packets. For a batch, all the packets within the same flow can be transmitted out and in quick succession at each sender, ensuring a

pipelined transmission of the packets. We experimentally determine the batch size for balancing the pipelined transmission as well as bandwidth utilization. We set the batch size to be eight in our implementation.

In addition to pushing the packet to the routing buffer at the next GPU, the sender module also appends the pointer to each packet in the incoming queue of the receiver GPU. The receiver module at the GPU that receives the packets will then pick up the pointer from the incoming queue and then either unpacks the data and passes it to the local partitioning phase or appends the pointer to the corresponding outgoing queue if the packet is not destined for the same GPU. The queues only contain the pointers to the packet instead of the actual packet data to avoid redundant copies of the actual data between the queues in the same GPU.

We develop effective buffer management for packet routing. To minimize the synchronization cost, the routing buffer at each GPU is implemented as a collection of buffers, one for each neighboring GPU. Each buffer is implemented as a circular buffer. For the example in Figure 5.3, the routing buffer at GPU 1 will be shared by both $F_{0 \rightarrow 2}$ and $F_{0 \rightarrow 3}$. Overall, this design allows MG-Join to 1) reduce the overall size of the routing buffer since the same buffer can be shared among multiple data flows and 2) minimize the synchronization overhead since a data flow only needs to synchronize its buffer access with other data flows arising from the same GPU (i.e no cross GPU synchronizations are required).

To avoid high memory management cost of managing the circular buffers accessible by multiple GPUs (sender and receiver), the status of the circular buffer is synchronized between the GPU only when the sender has no more free slots to push the data packets to the receiver. When this happens, the sender checks with receiver to determine the number of slots that have been freed by the receiver.

5.3.2 Adaptive Routing

In the previous section, we explored how multi-hop routing can give more flexible routing choices to the data distribution step of the global partitioning phase. However, there can be many possible routes between each pair of GPUs in modern multi-GPU architectures (Figure 5.2) and the overall cost of the data distribution can vary significantly based on the choice of routes for data transmission between

each pair of GPUs. Further, the choice of a specific route between a pair of GPUs can impact the choice of routes at other GPUs due to the limited number of DMA engines available per GPU hardware as well as the current data traffic load of each link. In the following, we first study a number of common static routing policies and identify their inefficiency for the hash join in multi-GPU architectures. This motivates us to design a fine-grained adaptive routing policy.

5.3.2.1 Need for adaptive routing.

Determining the global optimal route/schedule for the flows can be reduced to the classic multi-commodity flow problem [220–222]. Hence, it cannot be solved in a reasonable amount of time or scales to more GPUs. This cost can be even more significant for high-performance hardware like multi-GPU architectures. Thus, we need light-weight approaches to determine the route. There have been lightweight static heuristics based on simple metrics like peak bandwidth, latency or hop count. A routing policy using the bandwidth metric will always choose the shortest route with the highest bandwidth to the destination. Further, routing policies using latency and hop count always choose the route with the lowest latency and the smallest number of hops, respectively.

We study those three commonly used routing metrics under different hardware configurations, data distribution and packet size. In Figure 5.4a, we study how the different hardware configurations impact the cost of the data distribution step of the global partitioning phase. Note, the results are based on an equi-join of 1B^2 tuples ($|R|$ and $|S|$ has 512M tuples each). The x-axis lists the identifiers of all the GPUs participating in the MG-Join. In Figure 5.4b, we study the impact of different packet sizes and skew (data distribution) on the choice of the routing metric. Note, for this test we use the same input data set and the four GPUs ($\{0, 3, 4, 7\}$). $X(Z)$ in the x-axis represents XKB packet size and a Zipf factor of Z for the distribution of the input data.

Overall, those results show that the best static routing policy for routing the data can vary depending on hardware configurations, data distribution and packet size. Generally, bandwidth-based policy performs well for large data flows when executed on a reasonable number of GPUs (4-6). However, the efficiency of long routes

²B = 1024M

FIGURE 5.4: Impact of hardware configuration, data distribution and packet size on static routing policy (DGX-1)

begins to fall. Hop count based routing policy avoids long routes. However, the high heterogeneity in the interconnect network design can lead to the hop count based policy picking highly inefficient routes. In contrast, the latency based policy is efficient for small packet sizes and tend to achieve the best performance for many reasonable sized configurations. However, it tends to fall back to highly inefficient PCIe interconnects for configurations with a large number of GPUs resulting in high overhead.

To summarize, none of the static policies is a clear winner for different cases. Hence, an adaptive framework needs to be developed to ensure efficient use of interconnect

hardware in modern multi-GPU architectures.

5.3.2.2 Adaptive Routing in MG-Join

The key design goals of the adaptive routing framework in MG-Join are to dynamically route data flows to avoid the congestion in a subset of interconnect links and to improve the bandwidth utilization. We develop a lightweight metric to adaptive routing for transferring each packet in MG-Join. Specifically, we propose an adaptive routing metric (**ARM**) for determining the routes. ARM in MG-Join is computed based on a combination of dynamic (network delay and packet size) and static parameters (latency and bandwidth). Further, MG-Join performs the routing at the granularity of individual packets, making it possible to dynamically re-route data flows quickly to minimize the effects of congestion in the network. For routing each packet, the source node picks up the route with the least congestion (the route with the lowest dynamic delay).

We define the ARM values for a packet P over route R to be the total cost of moving the packet over the route. It is computed as a combination of two components: 1) a dynamic delay (D_R) resulting from congestion at the links and 2) the transmission cost of a pipelined transfer of data from source to destination (T_R), as shown in Equation 5.2.

$$ARM(R, P) = T_R + D_R \quad (5.2)$$

Transmission cost T_R for a packet P (denoting its size $||P||$) when transmitted over a route R can be computed based on static factors (assuming there is no congestion), as shown in Equation 5.3. Here, $B_E(||P||, R)$ is the maximum effective bandwidth achievable by a packet of size $||P||$ over the route R . In the initialization, we profile each type of link for different packet sizes and generate a function of B_E for different types of links. As multi-hop routing in MG-Join is designed to ensure pipelined transmission of packets, the effective bandwidth of the packet will be determined by the slowest link along the route which will be the transmission bottleneck for the entire data flow.

$$T_R = \frac{||P||}{B_E(||P||, R)} \quad (5.3)$$

The dynamic delay of a packet of data over route R (D_R) is computed as the sum of the queuing delay (Q_i) and the latency (L_i) of packet transmission over every single link constituting the route R , as shown in Equation 5.4. To enable quick computation of the dynamic delay of any route, each GPU locally maintains a table containing the Q_i and L_i values of every link in the interconnect network. The latency of data transmission of a link (L_i) is a physical characteristic of the link and does not change dynamically; while the queuing delay of a link (Q_i) changes with the level of congestion in the network. When the system begins execution, the Q_i values of the links are initialized based on the histogram data, by assuming a uniform arrival rate of packets to be forwarded. Later, whenever the Q_i value is updated due to the link transmission load, the GPU linked to it would broadcast the change in the queuing delay to every other GPU in the system. From the estimation, we can see that the dynamic delay value helps indicate congestion. Hence, any significant increase in the queuing delay serves as an indicator of network congestion.

$$D_R = \sum_{i=1}^{|R|-1} Q_i + L_i \quad (5.4)$$

Implementation details. To minimize the synchronization costs across GPUs, the route for each packet of data is determined at the source node where the packet was generated and packets will not be changed at intermediate nodes. This helps avoid the need for additional synchronization to ensure in-order packet arrival and to avoid circular routes. Based on our experiment, data packets on average require only 2-3 intermediate hops before reaching the destination.

As mentioned in Section 5.3.1, to reduce the routing overhead, packets from the same data flow are sent in a small batch with the same route. That means, in our implementation, adaptive routing is implemented at the basis of small batches. Specifically, for each batch, we calculate the ARM values for all the possible routes from the source GPU to the destination GPU with the constraint of at most 3 intermediate hops, and choose the route with the smallest ARM value as the route for that batch of packets from the source GPU to the destination GPU. Given the route, the source GPU generates the header of the packet, especially for the vector of GPU ids representing the route.

FIGURE 5.5: Example routing for adaptive and static policies.

5.3.2.3 An Adaptive Routing Example

To better understand how the adaptive routing policy in MG-Join helps determine efficient routes, we present an example in Figure 5.5. As shown in the figure, we use a configuration with four GPUs (Figure 5.5a) and we present the ARM metric computation for a data flow between GPU 1 and GPU 3 for all four routing policies (bandwidth, hop count, latency and adaptive) under the following two scenarios: 1) when the only active flow is a flow between GPU 1 and GPU 3 ($F_{1 \rightarrow 3}$), 2) when a new flow between GPU 2 and GPU 1 ($F_{2 \rightarrow 1}$) is activated simultaneously with $F_{1 \rightarrow 3}$.

Scenario 1. As shown in Figure 5.5b, when the only active flow is $F_{1 \rightarrow 3}$, the bandwidth and latency based policies use the high bandwidth route over GPU 2 for routing $F_{1 \rightarrow 3}$, while the hop count based policies make use of the less efficient direct route between GPUs 1 and 3. Now, since there is no congestion in the intermediate GPUs, the adaptive implementation chooses the same route as the bandwidth and latency based policy.

Scenario 2. When flow $F_{2 \rightarrow 1}$ is activated (not drawn in the figure), the ARM metric increases for the route 1-2-3 as shown in Figure 5.5c. This is due to the increased congestion at GPU 2 due to two simultaneously active flows. As shown in Figure 5.5c, when the increase in ARM is detected, adaptive implementation

updates the route for all new packets of $F_{2->1}$ and routes them over the GPU 0; while the bandwidth and latency based policies fail to detect this congestion and keep routing the data flow over the inefficient route.

To summarize, while the static policies are unable to update their routes for each flow depending on the level of congestion in the network, the adaptive routing policy in MG-Join adapts to dynamic network conditions.

5.4 Experiments

5.4.1 Experimental Setup

Hardware. We mainly use the DGX-1 server for our experiments, unless mentioned otherwise. The machine is equipped with 8 V100 GPUs, each containing 80 SMs operating at a boost clock rate of 1.53 GHz. Each GPU has 32 GB of HBM (High Bandwidth Memory), operating at a peak bandwidth of 900 GB/s. Its interconnects are shown in Figure 5.2. We further use a DGX-Station machine [133] which is equipped with 4 V100 GPUs to demonstrate the generality of the techniques proposed in this study. The DGX-station consists of 4 V100 GPUs interconnected to each other over NVLink and PCIe interconnects. For comparison against CPU, we use a dual socket machine with 512 GB of main memory and two Intel Xeon E5-2698 v4 CPU with 20 physical cores (40 logical cores with hyperthreading) running at 2.20GHz.

Workload. Following previous studies [10, 10, 12, 166, 218], we use a synthetic data set consisting of two relations R and S with a tuple size of 8 bytes (4-byte *key* and 4-byte *id*). The *key* values are generated as sequential integers and then randomly shuffled. We further set $|R| = |S|$ for all experiments and vary the total input size ($|R| + |S|$) from 512M to 4,096M tuples. We also evaluate the impact of data distribution based on a Zipf distribution.

Implementation details. We apply database compression to reduce the amount of cross-GPU traffic. First, as we use radix-based data partitioning, we use the first n bits of the keys in the global partitioning (n is determined by the hash key and the number of times the data is partitioned recursively during the global partitioning phase). Thus, during the global partitioning, we do not need to transfer the entire

key. Instead, the global partitioning phase partitions the data into groups based on the first n -bits of each key (the remaining bits of the key are not transferred in the global partitioning). Second, the tuple id values are compressed in a block basis (8KB as a block in our implementation). We first apply a delta operation on each tuple (based on the min value within a block) and then perform a null suppression encoding on the column to remove the zero-bits. Both these data compression approaches help MG-Join to achieve reasonable compression ratios (1.3x - 2x in our experiments) and thus reduce the overall overhead of the data distribution.

Experimental Outline. Our experimental evaluation of MG-Join is organized as follows. All experiments are performed on the DGX-1 server, except Section 5.4.3.2.

- In Section 5.4.2, we evaluate the individual impact of multi-hop data transmission and the adaptive routing policy used in MG-Join.
- In Section 5.4.3, we compare the overall performance of MG-Join against UMJ [32] and DPRJ [212]. For communication across GPUs within the same node, DPRJ makes use of CUDA communication APIs that chooses the direct routes between GPUs for data transfer.
- In Section 5.4.4 we evaluate the performance benefit of adopting MG-Join when executing TPC-H queries on large data sets by comparing its performance gains over Omnisci [7] running on CPU and GPU. Although there have been other query processing systems on GPUs such as DogQC [6] and Ocelot [9], they do not support multi-GPU architectures and thus cannot support the scale factor tested in our study. To the best of our knowledge, Omnisci is the state-of-the-art system capable of executing on both CPUs and modern multi-GPU systems. We use all the six queries in TPC-H which do not contain sub-queries (Q3, Q5, Q10, Q12, Q14 and Q19) and have at least one join operation. The detailed SQL clauses can be found in their benchmark websites [227].

5.4.2 Evaluating Data Distribution in Global Partitioning Phase

In this section, we focus on the data distribution step of the global partitioning phase of MG-Join. We evaluate the impact of varying the number of GPUs, while

FIGURE 5.6: Data Transfer throughput comparison of multi-hop routing in MG-Join against direct routing in DPRJ.

keeping each GPU allocated with 512M tuples from each input relation.

Impact of multi-hop data routing: Figure 5.6 shows the total data transfer throughput achieved during the data distribution step for the multi-hop routing used in MG-Join and direct route (used in DRPJ) when the number of GPUs participating in the join operation is increased from 2 to 8. Note, the total data transfer throughput is computed based on the total amount of data transferred between the GPUs and total execution time of the data distribution stage. Direct routing achieves almost the same bandwidth as MG-Join for a very small number of GPUs. This is because for a small set of GPUs, direct routing is already very efficient. However, as we scale the number of GPUs, the multi-hop routing policy achieves higher throughput (by up to 2.35x).

Impact of Adaptive Routing: We evaluate the efficiency of our adaptive routing policy as opposed to other static metric based routing policies. The results of this comparison in Figure 5.7 clearly show that, for a small set of GPUs, MG-Join achieves the same performance as other static metrics. This is because due to the small number of well-connected GPUs, all metrics end up choosing the same route. However, as the number of GPUs participating in the join operation increases, the adaptive routing approach begins to outperform static routing policies based on bandwidth, hop count and latency by up to 5.37x, 3.45x and 2.64x respectively. When a larger number of GPUs are involved, the adaptive routing policy is able

FIGURE 5.7: Data Transfer throughput comparison of adaptive routing in MG-Join against static routing policies.

to dynamically re-route the data flows, making more efficient use of available link bandwidth.

To further demonstrate the efficiency of MG-Join in taking advantage of the GPU and interconnect hardware, we present the average utilization of the bisection bandwidth of the interconnect network hardware achieved by DPRJ and MG-Join in Figure 5.8 for configurations with 4, 6 and 8 GPUs. Note, the utilization is computed based on a bisection bandwidth of each configuration. The results clearly show that, interconnect utilization of the DPRJ implementation drops with the number of GPUs to as low as 30%. In contrast, MG-Join ensures high utilization of GPU interconnect hardware, especially for configurations with a large number of GPUs. Overall, MG-Join achieves 97% utilization for eight GPUs. MG-Join achieves relatively low utilization when the number of GPUs is small. This is because, the small number of possible routes limits MG-Join’s ability to re-route data flows when congestion is detected in a subset of the links.

To demonstrate the ability of MG-Join to adapt to different distributions of the input data, we compare the normalized performance of the adaptive routing policy against the three static routing policies (bandwidth, hop count and latency) when input tuples are distributed based on a Zipf distribution among the 8 GPUs in the DGX-1 machine. The result is shown in Figure 5.9 for Z values varying from 0.0 to 1.0. The adaptive routing policy encounters the least degradation in performance with increase in the input data distribution skew, whereas other static routing

FIGURE 5.8: Interconnect hardware utilization of bisection bandwidth in MG-Join and DPRJ.

FIGURE 5.9: Normalized performance of different routing policies with data skew when executing over 8 GPUs of the DGX-1 machine.

policies encounter up to 3x degradation in performance. This demonstrates the ability of adaptive routing to dynamically re-route data flows and allocate the most efficient route regardless of data distributions.

5.4.3 Evaluation of Join Performance

In this section, we perform the overall comparison on the entire join performance of MG-Join against DPRJ and UMJ. Figure 5.10 presents the overall throughput of MG-Join against UMJ and DPRJ. The input tuples are distributed randomly across the GPUs with each GPU allocated with 512M tuples from each input relation.

FIGURE 5.10: Overall performance comparison of MG-Join against DPRJ and UMJ.

Following previous studies [11, 12, 32], we define throughput as the number of input tuples processed per unit time. We now make the following observations.

First, the UMJ implementation is highly inefficient due to its use of the unified memory feature, which encounters high overhead of handling page faults resulting from page tables getting locked by GPU threads [32]. Further, the effect of the page tables getting locked gets worse with the increasing number of GPUs, as larger number of GPU threads attempt to access the page table simultaneously. As a result, the performance of UMJ on multiple GPUs (from 5 to 8) is even worse than that of a single GPU.

Second, MG-Join outperforms DPRJ by up to 2.5x due to 1) network optimal partition assignment, 2) efficient adaptive routing during data distribution and 3) better overlapping of the compute and data distribution by allowing data routing at the granularity of individual packets. We will present the study on the impact of individual techniques later.

Third, while both DPRJ and UMJ achieve poor scaling with the GPU hardware resources, MG-Join achieves close to linear scaling with GPU hardware resources, demonstrating very good scalability on modern multi-GPU architectures.

To further understand the performance comparison, we compare the execution time breakdown of both DPRJ and MG-Join implementations with the number of GPUs varied in Figure 5.11. Note, in the figure $X(P)$ and $X(M)$ represents the execution on X GPUs for DPRJ and MG-Join, respectively. The DPRJ spends up to 72%

FIGURE 5.11: Execution time breakdown for MG-Join and DPRJ with varying number of GPUs.

FIGURE 5.12: Performance evaluation of MG-Join, DPRJ and UMJ for varying input size on DGX-1.

of its execution time moving the data between GPUs, while MG-Join spends only up to 35% of its execution time in cross-GPU data transfers. Further, for larger configurations like the one with 8 GPUs, MG-Join spends less than 20% of its execution time moving the data between the GPUs.

We present the performance of UMJ, DPRJ and MG-Join when the total input size of two relations is varied from 512M to 4B tuples on all the 8 GPUs in DGX-1 (Figure 5.12). Similar to the previous cases, MG-Join outperforms both UMJ and DPRJ for all input sizes. Overall, MG-Join achieves 10.2x and 3.6x better performance than UMJ and DPRJ implementations, respectively.

FIGURE 5.13: A factor analysis for MG-Join. Changes are added left to right and are cumulative.

In the remainder of this section, we present more detailed studies on the join performance.

5.4.3.1 Factor Analysis

To understand the individual benefits of both multi-hop and adaptive routing, we show a factor analysis in Figure 5.13. Baseline is the same as MG-Join, except that it routes the data over the highest bandwidth direct route between each pair of GPU. “+ Multi-Hop” adds multi-hop routing to the baseline, and “+ Adaptive” further enables adaptive routing (thus the same as the complete MG-Join). We have the following observations. First, multi-hop routing can help improve the performance of the hash join implementation with an optimal partition assignment by up to 1.69x. This is because 1) data flows gets routed through more efficient multi-hop routes instead of simply the direct route and 2) the packet based design of cross-GPU communication helps fine-grained and more effective overlap between the computation and communication.

Second, the use of adaptive routing policy further improves the performance by an additional 1.40x by minimizing the impact of congestion in the GPU interconnect through data compression and dynamic routing of data packets.

FIGURE 5.14: Overall performance comparison of MG-Join against DPRJ and UMJ on the DGX-Station.

5.4.3.2 Evaluation on Other Hardware

To demonstrate the ability of MG-Join to efficiently join relations on different multi-GPU architectures, we present the study on the DGX-Station machine [133]. The results in Figure 5.14 show that MG-Join is able to outperform UMJ and DPRJ by up to 2.4x and 1.37x, respectively. The slightly lower performance gain of MG-Join on DGX-Station when compared to DGX-1 is because DGX-Station has only 4 GPUs and more homogeneous interconnect network design, whereas DGX-1 has eight GPUs and more heterogeneous links.

5.4.4 TPC-H Query Evaluation

We evaluate the impact of MG-Join using six TPC-H queries to demonstrate the ability of MG-Join in improving the overall performance of query execution on modern multi-GPU architectures. For this, we implement GPU versions of 6 TPC-H queries that make use of MG-Join and compare them against CPU and GPU versions of Omnisci. When executing on multiple GPUs, OmniSci adopts a shared-nothing architecture between GPUs, i.e., each GPU processes its own local slice of data. Even though our MG-Join based implementation is capable of processing data sets with much larger sizes, we use a scale factor of 250 for this test due to limitations in Omnisci which prevents it from processing larger data sets on GPUs.

FIGURE 5.15: TPC-H query evaluation of MG-Join, Omnisci CPU and Omnisci GPU for a scale factor of 250.

The results of this comparison in Figure 5.15 show that the MG-Join implementation still outperforms DPRJ, and also outperforms both the CPU and GPU versions of Omnisci by 25x and 4.5x, respectively. Note, we are only able to present the results for Q14 and Q19 for Omnisci GPU since it fails to execute the other queries on the multi-GPU system for a scale factor of 250 (missing values are denoted by *NA*). We make further observations from this result.

First, MG-Join in this study helps GPU database systems to make efficient use of the GPU and interconnect hardware in modern multi-GPU systems.

Second, modern multi-GPU systems are able to significantly outperform the CPU version (with two sockets here) for high performance analytical workloads. Due to the explosive growth in deep learning workloads, these multi-GPU servers are becoming increasingly popular. NVIDIA alone has released 5 different multi-GPU server configurations for such workloads. Further, these devices are also becoming popular as cloud instances, making them even more accessible and cheaper. At the time of this writing, Amazon ec2 instance with 8 GPUs has an hourly cost of as low as \$8.39, while a CPU instance (in US West) with 48 physical cores cost \$7.776 per hour [228]. We acknowledge that the comparison between CPU and GPU is not extensive in this study and an interesting future work can be comparing the CPU systems with more sockets.

5.5 Summary

The rise of multi-GPU architectures with faster interconnects offer new opportunities of relational query processing on hundreds of gigabytes of GPU memory. However, we find that due to the unique characteristics of GPUs and the interconnect hardware, existing hash join designs demonstrate very bad scalability on multi-GPU architectures, which spend up to 66% of its execution time in cross-GPU communications and achieve lower than 50% GPU-GPU interconnect bandwidth utilization. To address this, we propose MG-Join, a scalable partitioned hash join implementation for modern multi-GPU hardware. MG-Join makes use of an adaptive multi-hop data distribution policy that helps ensure efficient use of GPU hardware and interconnect links and minimizes congestion across interconnect links. Our experiments show that MG-Join achieves up to 97% utilization of the interconnect bandwidth on DGX-1 with eight GPUs, and outperforms existing partitioned hash join implementations by up to 2.5x when joining large input data sets distributed over multiple GPUs. MG-Join further helps improve the performance of TPC-H queries by up to 4.5x over OmniSCi.

Chapter 6

Improving Execution Efficiency of Compiled Systems on GPUs

6.1 Introduction

In recent years, we have witnessed a significant effort in improving the performance of Online Analytical Processing (OLAP) systems on graphics processing units (GPUs) [8, 9, 184]. More recently, JIT compilation-based systems (hereafter referred to as *compiled systems*) [3, 5, 39, 40] have evolved as the state-of-the-art query execution systems on GPUs. The JIT compilation has the major advantage to reduce unnecessary materialization of intermediate data at operator/kernel boundaries within query pipelines and to reduce the instruction count within GPU kernels. On the other hand, many efforts have focused on improving the memory efficiency since memory stalls play an important role in query processing performance on GPUs [9, 11, 28, 135, 159, 229]. Given all these optimizations, it is time to revisit the query execution on GPUs and identify the opportunities to improve query processing performance on GPUs further.

Taking a closer look at existing compiled systems [3, 5, 39, 40], we find that they work by first breaking down the query plan associated with each query into pipelines and then generating a single kernel for each pipeline of relational operators. We refer to this compilation strategy as a *monolithic* approach, since it generates a monolithic kernel for each pipeline. While this monolithic approach eliminates

This work has been accepted for publication in PVLDB 2020.

the amount of intermediate data materialization, we have identified two major performance issues.

First, the generated kernels in the monolithic approach often have a large number of nested branch instructions per kernel. Hence, the thread warps executing the kernels generated by the monolithic approach can have a high degree of divergence, resulting in poor utilization of a large number of GPU cores.

Second, the high resource requirements of the generated kernels lead to low parallelism on GPU. Overall, the monolithic approach generates kernels containing multiple relational operators, which require even more registers and shared memory resources. That means the monolithic approach exaggerates the problem of resource contention on the GPUs since threads can only be scheduled for execution on GPUs if there are enough free hardware resources available. Therefore, the monolithic approach also leads to degraded parallelism on GPUs and cannot fully utilize the large number of parallel cores on the GPU.

To demonstrate the impact of the two execution efficiency issues, we present the execution time break down of the code generated from the monolithic approach for TPC-H [227] (queries without subquery) and SSB [230] in Figure 6.1. We divide the execution time into three components: 1) estimated overhead of divergence, 2) estimated overhead of resource contention, and 3) others. The overhead of divergence and resource contention were estimated based on the achieved execution efficiency of the warps and GPU hardware occupancy rate, both of which can be obtained using profiling tools from GPU vendors. The results of the breakdown clearly demonstrate that divergent execution and resource contention among the threads have a significant impact on the overall performance of both TPC-H and SSB queries. Overall these inefficiencies can contribute to more than 50% of the total execution time. Therefore, divergence and resource contention have now become the most significant bottleneck for query processing on GPUs.

In this chapter, we propose *Pyper*, a JIT compilation-based query processing system to resolve GPUs' divergence and resource contention issues. Pyper introduces two new operators for query plan transformation, *Shuffle* and *Segment*, to improve GPU hardware utilization during query execution. Specifically, *Shuffle* is applied to a pipeline to reduce the execution divergence. *Segment* is used to split a pipeline that has overwhelming resource usage into several pipelines with lower resource

FIGURE 6.1: Execution time breakdown of TPC-H and SSB queries demonstrating the impact of divergent execution and resource contention among threads.

usage but much higher thread parallelism. The benefits of the two operators come with overheads. Thus, we develop a cost model to guide the optimal use of Shuffle and Segment operators. The challenge is that we need to estimate the impact of divergence and resource contention, which have been overlooked in the previous studies [28, 41].

The major contributions of this chapter are as follows.

- We study the efficiency of current JIT compilation-based query processing approaches and identify their severe under-utilization of GPU hardware due to divergent execution and degraded parallelism arising from resource contention.
- We propose Pyper to address the execution efficiency issues of existing JIT compilation-based systems, by introducing two new operators (Shuffle and Segment) and an analytical model to determine the usage of the two operators in a cost-efficient manner.
- We conduct in-depth experiments demonstrating the benefit of using the Shuffle and Segment operators as well as the accuracy of our analytical model. Our experiments show that Pyper is able to improve the performance of TPC-H and SSB queries on average by 1.60x and 1.52x, compared to Hawk [3]. Pyper further achieves 5.01x and 2.55x performance improvement over Omiscsi [7] for TPC-H and SSB queries respectively.

The rest of this chapter is organized as follows. In Section 6.2, we present the GPU hardware’s background and the related work on JIT query compilation. We then present the architecture design of Pyper in Section 6.3. We present the details of the analytical model and query optimizer in Sections 6.4 and 6.5, respectively. Finally, we present the experiments in Section 6.6.

6.2 GPU Hardware & CUDA

A single GPU consists of multiple *streaming multiprocessors* (SMs), each of which consists of multiple CUDA cores. All CUDA cores in an SM share resources like the registers and shared memory among each other. Due to this sharing, the workload scheduler in GPU only schedules new threads for execution on an SM, if there are enough registers and shared memory resources available. GPUs also have an L2 cache and a global memory that are shared among all the SMs. Each thread block is given exclusive access to shared memory, i.e., data in shared memory cannot be shared across different thread blocks.

Warp-at-a-time execution model: In the CUDA programming model, a program executed by the GPU is known as a kernel. A kernel is executed as a grid of thread blocks, which can further be divided into *warps* (groups of 32 threads). Each thread block is assigned to a single SM, and the CUDA cores inside each SM executes the threads in a SIMD fashion, at the granularity of a single warp. Hence, during any given execution cycle, an entire warp of threads can only execute a single instruction. That means, during each instruction issue cycle, only threads that can execute the same instruction will be activated, and the resources allocated to the other threads would be wasted. Thus, branch divergence can be a serious performance issue on GPU executions.

Resource constraints per thread: For the efficiency of GPU executions, registers, and shared memory are the key resources that constrain the number of parallel thread executions. Warps can only be scheduled for execution on GPUs if there are enough free hardware resources available.

6.3 Design of Pyper

To resolve the performance issues of divergent execution and resource contention in the monolithic approach, we propose Pyper, a JIT compilation-based query processing system designed for GPUs. In the following, we first give a system overview of Pyper and present the detailed design and implementation of Shuffle and Segment operators.

6.3.1 Overview

Figure 6.2 provides an overview of the architecture design of Pyper. As shown in the figure, Pyper is developed by augmenting a physical query plan generated from existing compiled systems [39, 113]. Specifically, we have the following featured components.

- Two plan transformation operators, Shuffle (Section 6.3.2) and Segment (Section 6.3.3). They can be inserted into a physical query plan, to help reduce the divergent execution and resource contention among threads. We designed these operators such that they can be inserted into any query pipeline, without modifying the implementations of the other physical, relational operators. Moreover, we develop efficient designs in order to reduce their runtime overheads.
- An analytical model (Section 6.4), which is capable of estimating the execution cost of a pipeline of relation operators. Our model has non-trivial extensions over the existing GPU analytical model [4] by developing a cost analysis of divergent executions and resource contention.
- An augmentation optimizer (Section 6.5), which uses the analytical model along with a branch and bound based optimization algorithm to identify the optimal insertion points of Shuffle and Segment operators into the physical query plan.

Putting it together, Pyper works as follows. In this study, we use the pipeline physical query plan generated by the previous study named GPL [4]. Pyper further optimizes each of the pipelines by considering the insertions of Shuffle and Segment.

FIGURE 6.2: System Architecture of Pyper

This is guided by the augmentation optimizer, which takes the estimations of the analytical model into consideration. The outcome is an augmented (or optimized) query plan. Then, the code generator generates the code corresponding to each pipeline in the augmented query plan. Finally, the generated code is compiled into hardware-dependent machine code using GPU backend (e.g., *libnvvm* or *nvcc* from NVIDIA).

6.3.2 Shuffle Operator

The Shuffle operator is an operator that can be inserted between two relational operators (producer and consumer). It buffers the tuples from divergent warps from the producer operator and consolidates the input tuples to the consumer operator in the pipeline so that the execution of the consumer operator has the reduced divergence among the threads within the same warp. Thus, this thesis focuses on the warp divergence brought by the divergent execution path of multiple operators in a pipeline. The divergence within an operator (mainly branch divergence) is not the main focus of this thesis. Specifically, we focus on the divergence when different threads are assigned different amounts of work. For example, the number of tuples selected for processing in the next operator after a filter predicate can vary across threads, resulting in divergence among the threads of executing the operators after the filter. The same applies to a series of hash probe operations where different

FIGURE 6.3: Warp executions with and without Shuffle operators

threads could end up generating different numbers of output tuples depending on the join selectivity.

How does Shuffle reduce divergent executions? To demonstrate the benefit of using the Shuffle operator, we present an example execution of the warps executing the pipeline P_1 (shown on the left of Figure 6.3) from the previous example query. We assume a warp size of 4 and the uniform selectivity of 50% per operator for this example for simplicity. We now look at the execution of the pipeline with and without inserting Shuffle operators.

Without Shuffle: Due to 50% selectivity of $\bowtie_{a=b}$ and $\bowtie_{c=d}$, only 50% and 25% threads in each warp will be allocated work in the hash probe of $\bowtie_{c=d}$ and the hash probe of $\bowtie_{f=g}$ respectively. Such workload imbalance leads to the poor overall utilization of the GPU hardware since only a small subset of the threads in each warp are allocated work.

With Shuffle: Inserting a Shuffle operator between each pair of relational operators in P_1 forces the workload to be concentrated over fewer warps executing at 100% efficiency, as shown in Figure 6.3. This helps make more efficient use of the GPU hardware since warps without any work can be quickly swapped out, allowing other warps to execute.

Lightweight Shuffle on the GPU. Now, re-balancing the workload in a lightweight manner is a non-trivial task. Consider a producer-consumer operator pair in the pipeline (denoting O_p and O_c as the producer and consumer operators, respectively). The Shuffle operator is inserted into this pipeline so that the output from

O_p (which is sparsely distributed across the threads) is consolidated into a smaller number of warps, thus reducing the divergent execution in O_c . A potential implementation of the Shuffle operator is to simply gather the outputs from O_p . While this implementation helps eliminate divergent execution, it has severe efficiency issues. This is because the implementation is blocking and thus causes unnecessary intermediate result materialization of all the tuples between O_p and O_c .

To overcome these inefficiencies, we develop a lightweight approach that takes advantage of GPU hardware features to implement Shuffle. We have the following design rationales. First, our implementation completely avoids blocking across thread blocks, thus avoiding costly synchronizations using GPU global memory. Second, warps that do not encounter high levels of divergence are prevented from participating in the re-assignment of the workload, thus minimizing the synchronization within a thread block and global memory accesses. Third, we make use of GPU specific low overhead intrinsics to implement the Shuffle operator. Listing 6.1 illustrates the implementation of the Shuffle operator, which consists of the following two stages: *voting* and *shuffling*.

Voting Stage: During the voting stage, each thread votes using the `--ballot` instruction to reach a consensus on whether there is workload divergence within the warp (Lines 6 – 9). The GPU voting intrinsics are very efficient, each executing within a single cycle. `--ballot(vote)` in Listing 1 generates a 32-bit integer where the i^{th} bit is set to 1 if the i^{th} thread has a valid input tuple assigned to it (i.e., $vote = 1$). The `--popc` instruction is another intrinsic to compute the number of bits in this value that has been set to 1. In summary, `--popc(--ballot(vote))` computes the number of threads that been assigned tuples for processing in each warp ($actThreadsW$). Finally, Lines 10 and 11 keep track of the number of threads that are assigned with tuples within an entire block using a variable in shared memory ($actThreadsB$).

Shuffling Stage: During the shuffling stage (Lines 13 – 26), the workload is redistributed based on the consensus reached in the voting stage. We can perform workload shuffling at different levels, which come with different benefits and overheads. Particularly, the shuffling across the entire kernel (named grid-level) achieves the most divergence reduction but has a high overhead. In contrast, the shuffling within a thread block (named block-level) achieves reasonable divergence reduction among the warps within a thread block but has relatively low overhead. We use grid-level

shuffling (Lines 14 – 17) or block-level shuffling (Lines 19 – 26), based on the level of divergence encountered by the threads.

- Grid-level shuffling requires moving tuples across multiple thread blocks and is costly due to the use of GPU global memory. The threshold for Grid-level shuffling (*GridT*) depends on the GPU hardware as well the downstream operator. This is because the Grid-level shuffling is beneficial only when the additional cost of global memory use is more than compensated by the reduced divergence in the downstream operators. Hence, during system initialization, Pyper profiles the GPU hardware and determines the benefit of Grid-level shuffling for the supported relational operators. Finally, to avoid deadlocks and blocking execution across thread blocks, each thread block executing the *GridShuffle* function checks a global buffer (*global_shuffle_mem*) and either 1) picks up tuples from the buffer and redistributes them across its threads or 2) writes the tuples held by its threads into the buffer if there are not insufficient tuples already in the buffer for redistribution. The last set of thread blocks that will be scheduled for execution only picks up tuples from the buffer to ensure correctness.
- Block-level shuffling is more frequently used by Pyper and helps address more moderate levels of divergence (which is often encountered in common analytical workloads). Based on our experiments, we consider a warp as requiring block-level shuffling if a warp (32 threads) achieves less than 75% occupancy on the GPU hardware (i.e., *BlockT* = 24). Overall, if more than 2 warps are requiring block-level shuffling within a thread block (i.e., *numWarp* > 2 as shown in Line 22), then the Shuffle operator invokes the *BlockShuffle*. BlockShuffle is implemented as an efficient filter process described in the previous study [198]. Note, we use variables in shared memory and registers to invoke and perform the block-level shuffling, making it highly efficient. The output of BlockShuffle is the number of valid input tuples for this thread block. After BlockShuffle, all valid tuples are stored consecutively, thus avoiding any divergence in the next/downstream operator.

Even though we have a very efficient implementation for Shuffle, this operation still leads to an increase in the kernel resource requirements. Hence, the insertion of a Shuffle will be used only when it is beneficial for the performance. To further,

```

1 Input tuple; // input generated by the previous operator
2 __shared__ Input shuffle_mem[ ]; // buffer for shuffling
3 __shared__ int numWarp = 0;
4 __shared__ int actThreadsB = 0;
5
6 /* Voting Stage */
7 int vote = 0
8 if (tuple != nil) vote = 1
9 int actThreadsW = __popc(__ballot(vote));
10 if (tidInWarp == 0) //tidInWarp: threadID in warp.
11     atomicAdd(actThreadsB, actThreadsW)
12
13 /* Shuffling Stage */
14 if (actThreadsB <= GridT) {
15     //tidInBlock: threadID in block.
16     shuffle_mem[tidInBlock] = tuple;
17     tuple = GridShuffle(shuffle_mem, global_shuffle_mem)
18 }
19 else if (actThreadsW <= BlockT && tidInWarp==0) {
20     atomicAdd(numWarp, 1);
21 }
22 if (numWarp > 2) {
23     shuffle_mem[tidInBlock] = tuple;
24     numValid = BlockShuffle(shuffle_mem);
25     if (tidInBlock < numValid)
26         tuple = shuffle_mem[tidInBlock];
27 }

```

LISTING 6.1: Simplified code for Shuffle operator

minimize the overhead, the code corresponding to grid-level shuffling (Lines 10 – 17) is only generated as part of the Shuffle operator when extremely low levels of execution efficiencies are encountered. We leave these decisions to the cost model and augmentation optimizer.

6.3.3 Segment Operator

The Segment operator is to split a pipeline into two smaller pipelines. It can be inserted into a long resource-constrained pipeline, split into two smaller pipelines with lower resource requirements. The monolithic approach aggressively fuses relational operators in the same pipeline and compiles it into a single kernel on the GPU. Such a kernel usually has high resource requirements, resulting in low occupancy rate of GPU and degraded parallelism, which makes it difficult for modern

GPU hardware to hide memory latency. We find that the Segment operator is desirable to split monolithic kernels and improve the GPU’s utilization.

In this thesis, we consider two major constrained resources on the GPU: registers and shared memory. Modern GPUs have limited sizes of shared memory and register files that need to be shared across a large number of threads. For example, the NVIDIA V100 GPU used in our experiments is configured with 64KB shared memory and 256 KB registers per SM, which needs to be shared among 64 active warps to ensure complete occupancy of each SM.

Consider P_1 in Figure 6.4, where the pipeline consists of two hash probe and one hash build operators. Assuming each operator needs to keep the hash table in the GPU shared memory, the thread block executing the kernel corresponding to the pipeline would require 33 KB of shared memory per thread block, as shown in Figure 6.4a. If this query plan runs NVIDIA V100 GPU, the pipeline can only achieve a maximum theoretical occupancy rate of 50%.

How does Segment improve the parallelism? To understand how the use of the Segment operator helps address the above inefficiency, we present Figure 6.4b, in which a Segment operator is inserted into the pipeline P_1 between two of its hash probe operators. The insertion of the Segment operator at this point splits P_1 into two smaller pipeline fragments: P'_1 containing hash probe of $\bowtie_{a=b}$ and P''_1 containing hash probes of $\bowtie_{c=d}$ and $\bowtie_{f=g}$. Such a split means that the resulting pipeline fragments P'_1 and P''_1 only require 16 KB and 17 KB of shared memory respectively, which enable P'_1 and P''_1 to achieve a theoretical occupancy rate of 100% individually. Hence, by inserting the Segment operator into the pipeline, we can generate kernels that achieve significantly better occupancy rate and in turn, better utilization of GPU hardware. In the following, we present a lightweight design for Segment.

Lightweight design for Segment. Overall, the Segment operator is implemented as a *blocking* operator with a write ($Segment(W)$) and a read ($Segment(R)$) stages, as shown in the example of Figure 6.4b. When generating code for the pipeline fragments of a pipeline split using the Segment operator, the code for the write stage is generated in the kernel corresponding to the upstream pipeline fragment, and the code for the read stage is generated in the kernel corresponding to downstream pipeline fragment.

FIGURE 6.4: Impact of Segment operators on query plan and resource requirements

The write stage of the Segment operator includes the instructions to collect all the tuples generated by the downstream operator and writes this data into the GPU global memory; while the read stage includes the instructions to reads the data written by the write stage from the global memory and passes this data to the upstream operator in the query plan. We optimize write with *write buffer* [32], which writes the output in batches rather than per tuple. For read, we take advantage of coalesced memory accesses to improve bandwidth utilization.

The usage of the Segment operator has runtime overhead due to the materialization of the data into GPU global memory between the write and read stages of the Segment operator. Hence, the Segment operator should only be used when it can improve the overall performance. This will be determined by the cost model in the next section.

The idea of the segment operator in Pyper is similar to previous studies [231–233], with the following major differences. First, the segment operator in Pyper is inserted to minimize resource contention in the generated code, whereas previous studies have different purposes, such as vectorization opportunities [232, 233]. Second, our segment implementation is designed specifically for GPUs and is more

efficient for GPU hardware due to its use of local buffering and coalesced writes. Third, comparison to previous studies, this study has an efficient cost model to determine the optimal insertion point (with the awareness of the resource contention of the generated code) of the Segment operator.

6.4 Analytical Model

There have been several studies on building cost models to estimate the execution time of a query plan on the GPU [28, 41]. However, those studies fail to effectively account for the impact of divergent execution and resource contention. As we demonstrated in Section 6.1, the overhead of divergent execution and resource contention can contribute to over 50% of the total execution time in the monolithic approach. Thus, this thesis extends the previous model to handle divergent execution and resource contention in JIT compilation-based query plans.

It is a non-trivial task to build a cost model that accounts for divergent execution and resource contention. This is because the impact of these factors depends on a variety of factors like characteristics of the GPU hardware, relational operators in the query plan and the input data. To resolve these complexities in building an analytical model, we have the following rationales.

First, instead of building an analytical model from scratch, we use the previous model [4] in estimating the cost of a pipeline execution, and then extend this model to estimate the impact of divergent execution and resource contention. To account for divergent execution, we need to model the number of active threads per warp in each operator, and then estimate the divergent execution for the entire pipeline. The major impact of resource contention is the degree of thread parallelism, which is a key factor for the performance in the execution unit and memory sub-system of the GPU.

Second, we treat the cost analysis for Shuffle and Segment operators the same as other relational operators. Hence, any query plan augmented by the insertions of Shuffle and Segment can be analyzed uniformly.

TABLE 6.1: Notations in the cost model

Name	Description	Sources
P	A pipeline in the query plan	Query Plan
$ P $	The number of operators in P	
O_i	i^{th} operator in P	
$\hat{P}$	The execution cost of P	Model Output
$\hat{O}_i$	The execution cost of O_i	
γ_i	Number of scheduled threads with input in O_i	
$\hat{C}_i(\gamma_i)$	Computation cost per tuple of O_i	
$\hat{M}_i(\gamma_i)$	Memory access cost per tuple of O_i	
T_i	Avg. active threads per warp of O_i	Program & Selectivity Analysis
$ E_i $	Number of <i>warp execution</i> of O_i	
T_{ij}	Number of threads with work in the j^{th} <i>warp execution</i> of O_i	
W_i	Number of warps of O_i that can execute simultaneously on GPU	
W_i^{sm}	Number of warps of O_i that can execute simultaneously per SM	
$ B_i^w $	Warps per thread block in O_i	
S_i	Shared mem. per thread block in O_i	
R_i	Registers per thread block in O_i	
$ M_i^s $	Number of shared mem. accesses in O_i	
$ M_i^g $	Number of global mem. accesses in O_i	
H_i	L2 cache hit rate in O_i	
$\hat{M}_{ij}^s(\gamma_i)$	Cost of j^{th} shared mem. access in O_i	Profiling
$\hat{M}_{ij}^{ddr}(\gamma_i)$	Cost of j^{th} global memory access in O_i	
$\hat{M}_{ij}^{l2}(\gamma_i)$	Cost of j^{th} L2 cache access in O_i	
W_{max}	Max. simultaneous warps per SM	Platform Input
B_{max}	Max. simultaneous blocks per SM	
R	Registers available per SM	
S	Shared mem. available per SM	
$\#SM$	Number of <i>SMs</i> in the GPU	

6.4.1 Pipeline Execution Cost

The previous study GPL [4] proposed a model to compute the execution cost of a pipeline of relational operators (P). Note, here the execution cost refers to the total execution time of the pipeline. The model in GPL computes the cost of processing a single tuple by a pipeline of operators ($P = \{O_1, O_2, \dots, O_{|P|-1}, O_{|P|}\}$) using Equation 6.1. Note, O_i can be Shuffle, Segment or relational operators. Here, $\widehat{C}_i$ and $\widehat{M}_i$ represent the computation and memory access cost of operator O_i for processing each tuple, respectively. λ_i represents the estimated number of input tuples for O_i from the query optimizer.

$$\widehat{P} = \sum_{i=1}^{|P|} (\widehat{C}_i + \widehat{M}_i) \times \lambda_i \quad (6.1)$$

GPL's model does not properly account for the divergent execution and resource contention in JIT compilation-based query plan. The total number of active threads issuing and executing instructions in parallel on the GPU (denoted as γ_i) can vary significantly based on the factors in divergence and resource usage. γ_i has a significant impact on query processing performance. Thus, we need to extend the estimation of $\widehat{C}_i$ and $\widehat{M}_i$ to account for the impact of γ_i . Specifically, we model the computation and memory access costs as functions of γ_i (denoted as $\widehat{C}_i(\gamma_i)$ and $\widehat{M}_i(\gamma_i)$, respectively). Thus, the estimated cost of P in our cost model is given in Equation 6.2.

$$\widehat{P} = \sum_{i=1}^{|P|} (\widehat{C}_i(\gamma_i) + \widehat{M}_i(\gamma_i)) \times \lambda_i \quad (6.2)$$

In the following, we detail how we estimate the values of γ_i , $\widehat{C}_i(\gamma_i)$ and $\widehat{M}_i(\gamma_i)$.

6.4.2 Estimation of γ_i

Due to the warp-at-a-time execution model of GPUs, we estimate γ_i to be the number of active warps multiplied by the average number of active threads per warp. That is, γ_i can be computed using Equation 6.3, where T_i represents the average number of active threads per warp (in O_i) and W_i denotes the number of active warps of O_i that can execute simultaneously on the GPU.

$$\gamma_i = T_i \times W_i \quad (6.3)$$

Calculation of T_i . T_i is an effective measure of divergence among the threads executing O_i . Specifically, lower T_i represents a higher degree of divergence among the threads. T_i depends on the selectivity of O_{i-1} in the pipeline. The idea is, if a tuple is eliminated by O_{i-1} , the thread in O_i and thereafter in the pipeline could have less work to handle. This effect has to account for all the ancestor operators in the pipeline. Assuming the selectivity of O_i to be e_i . In this case, the ratio of the active threads per warp is $\prod_{j=1}^{i-1} e_j$. We estimate T_i as the number of threads in a warp multiplied by $\prod_{j=1}^{i-1} e_j$.

Calculation of W_i . Since GPU SM hardware design and the resource distribution across the thread blocks of a kernel are uniform, the number of warps that can execute simultaneously on a given GPU hardware (W_i) can be computed as the product of W_i^{sm} and the number of SM in GPU hardware ($\#SM$), as shown in Equation 6.4.

$$W_i = \#SM \times W_i^{sm}. \quad (6.4)$$

For an operator O_i , the maximum number of warps per SM that can execute simultaneously on the GPU hardware (W_i^{sm}) depends on 1) the limit per SM set by the hardware and 2) the resource requirements of the kernel executing the operator.

- *Hardware Limit.* Due to the limited amount of resources available for managing thread contexts at the warp and thread block levels, NVIDIA CUDA sets hard limits on the number of warps and the number of thread blocks that can execute simultaneously on each SM of the GPU (denoted as W_{max} and B_{max} , respectively). Therefore, W_i^{sm} will be limited by Equation 6.5 where $|B_i^w|$ is the number of warps per thread block in O_i .

$$W_i^{sm} \leq \min(W_{max}, B_{max} \times |B_i^w|) \quad (6.5)$$

- *Resource Limit.* Now, each thread block executing O_i requires certain amount of register and shared memory resources (denoted as R_i and S_i , respectively) and can only be scheduled for execution when sufficient free resources of both types are available in hardware. Hence, the number of thread blocks that can be scheduled for simultaneous execution on an SM is limited by the register

(R) and shared memory resources (S) available at the hardware level in each SM. Hence, the value of W_i^{sm} is further limited by Equation 6.6.

$$W_i^{sm} \leq \min(\lfloor \frac{R}{R_i} \rfloor, \lfloor \frac{S}{S_i} \rfloor) \times |B_i^w| \quad (6.6)$$

To summarize, the number of warps of O_i that can execute simultaneously on each SM (W_i^{sm}) is the largest integer satisfying Equations 6.5 and 6.6.

6.4.3 Estimation of Computation Cost ($\widehat{C}_i(\gamma_i)$)

Modern GPU hardware is designed to allow GPU threads to issue and execute instructions in parallel. Hence, the computation cost of a relational operator ($\widehat{C}_i(\gamma_i)$) can be estimated as the execution cost of all the instructions that need to be executed in O_i (denoted as C_i) divided by the number of parallel active threads issuing instructions (γ_i), as denoted by Equation 6.7. C_i can be computed through offline profiling of the specific hardware and the operator code as done in GPL [4]. Overall, Equation 6.7 clearly shows that the lower the workload divergence and resource contention among threads (i.e., higher γ_i), the lower the computation cost of the generated kernel is.

$$\widehat{C}_i(\gamma_i) = \frac{C_i}{\gamma_i} \quad (6.7)$$

6.4.4 Estimation of Memory Cost ($\widehat{M}_i(\gamma_i)$)

The cost of a memory access on GPU can vary depending on the following factors: 1) the type of physical memory being accessed (shared or global memory), 2) the pattern of memory access (sequential, random, stride) and 3) the number of threads issuing the memory access request in parallel. Taking into consideration the type of physical memory being accessed, the memory access cost ($\widehat{M}_i(\gamma_i)$) per tuple can be computed as the sum of the memory access cost to the shared memory, L2 cache and global memory required to process a single input tuple (Equation 6.8). For an operator O_i , $|M_i^s|$ and $|M_i^g|$ represent the number of accesses to the shared memory and global memory (which could also be cached in the L2 cache), respectively.

To have a more accurate estimation, we consider the cost per memory instruction. This is possible because the instructions from each operator are known in JIT-based compilation system. Thus, $\widehat{M}_{ij}^s(\gamma_i)$, $\widehat{M}_{ij}^{l2}(\gamma_i)$ and $\widehat{M}_{ij}^{ddr}(\gamma_i)$ represent the cost of shared memory, L2 cache and global memory accesses cost of the j^{th} memory access in O_i as a function of γ_i . H_i is the average L2 cache hit rate in O_i and is obtained following the same approach as GPL [4].

$$\widehat{M}_i(\gamma_i) = \sum_{j=1}^{|M_i^s|} \widehat{M}_{ij}^s(\gamma_i) + \sum_{j=1}^{|M_i^g|} (H_i \times \widehat{M}_{ij}^{l2}(\gamma_i) + (1 - H_i) \times \widehat{M}_{ij}^{ddr}(\gamma_i)) \quad (6.8)$$

It is challenging and complicated to build analytical models for these metrics. In this thesis, we use a calibration-based approach for the calculation. For each memory instruction, each thread issues the memory access and those memory accesses form a certain access pattern (such as sequential, random and strided). Thus, we need to consider the factors that affect the cost of the memory access, including the size of the memory access, the access pattern, and the number of active threads. Particularly, in JIT-based compilation systems, since the supported operators are known in advance, we perform an offline profiling during the model initialization to derive their functions with respect to the above-mentioned factors. Particularly, we perform profiling on the GPU memory hardware to measure the time per memory access for varying the number of active threads on a given configuration of access pattern and memory access unit size. We study different configurations by considering all the combinations of memory unit sizes (e.g., 2 bytes, 4 bytes and 8 bytes) and the access pattern commonly used in databases (such as sequential, random and stride from 2 bytes, 4 bytes, ... to 32 bytes). Then, for each configuration, we then perform a regression analysis for the function which gives the memory cost by given the number of active threads as input. In our study, the entire calibration takes around 30 minutes when the system initializes (this calibration runs only once, and the calibration results can be reused afterwards for the same hardware). At runtime, we simply use these functions to quickly compute the cost of each memory access for each memory instruction.

As an example, Figure 6.5 shows the calibration data along with the fitted curve for sequential access to shared memory, L2 cache and the global memory of the V100 GPU (γ_i values from 40K to 160K). Their regression curves are also shown in the figure, with RMSE lower than 8%. For clarity of the figure, we scale up $\widehat{M}_i^s(\gamma_i)$

FIGURE 6.5: Variation of sequential memory access cost with γ_i for V100 GPU

by a factor of 10. The performance trends of all the three types of memory (shared memory, cache and global memory) clearly show that the performance needs to take γ_i into consideration. In experiments, the regression approach gives sufficient accuracy and contributes to the accuracy of our cost model.

6.4.5 Example Cost Model Computation

We use our running example pipeline P_1 to show the computation of values corresponding to divergence (T_i), resource contention (W_i), γ_i , number of processed tuples (λ_i), computation cost ($\hat{C}_i(\gamma_i) \times \lambda_i$) and memory access cost ($\hat{M}_i(\gamma_i) \times \lambda_i$) by our analytical model for the following three scenarios: 1) the original pipeline, 2) the pipeline after insertion of the two Shuffle operators (Figure 6.3) and 3) the pipeline after the insertion of a Segment operator (Figure 6.4). Note, all these values are computed for a demo kernel using Equations 6.1 - 6.8. The hardware is the V100 GPU used in the experiment.

Figure 6.6b shows the values of the parameters after two Shuffle operators are inserted into the original pipeline. The insertion of the Shuffle operator helps improve the T_i and subsequently γ_i values for both $\bowtie_{c=d}$ and $\bowtie_{f=g}$. As in our model estimation, this higher γ_i value represents an increase in the execution efficiency of the warps and a decrease in execution cost of the relational operators.

FIGURE 6.6: Example cost estimation by the analytical model (↓ indicates a decrease in execution cost and ↑ indicates an increase in the overhead resulting from the insertion of the Shuffle or Segment operator.)

Figure 6.6c shows the values of the parameters after a Segment operator is inserted into the pipeline. The insertion of the Segment operator breaks down the large pipeline helping the generated kernels to achieve higher W_i and subsequently γ_i values, as shown in our model estimation.

6.5 Augmentation Optimizer

With the analytical model in hand, we are able to estimate the cost of a query plan with arbitrary insertions of Shuffle and the Segment operators. However, we can potentially insert Shuffle and Segment operators after each operator into

the pipeline. The solution space can be large for complex queries. Also, different insertions can have very different performance, as observed in our experiments. Thus, we develop a GPU aware optimizer in Pyper to efficiently find the augmented query plan with the lowest execution time.

Specifically, we make use of a Branch & Bound (B&B) based technique [234] to solve this problem. In the remainder of this section, we detail how the B&B based technique is used to determine the optimal insertion point for the Shuffle and Segment operators.

Overview. The augmentation optimizer in Pyper applies the B&B based optimization algorithm to each pipeline in the query plan individually. For each pipeline, the algorithm starts with the original pipeline in the form of $P = \{O_1, \dots, O_n\}$ as the root, where n is the number of operators in the pipeline, and O_i is the producer operator for O_{i+1} ($1 \leq i < n$). Then, the augmentation optimizer systematically enumerates a tree, based on a bounding function. Each node in the tree represents an augmented query plan for the input pipeline, which is generated through an insertion of: 1) a single Shuffle operator, 2) a single Segment operator or 3) a pair of Shuffle and Segment operators between any two relational operators in its parent node. When a pair of Shuffle and Segment operator is inserted, the Shuffle operator is always inserted before the Segment operator as the write stage of the Segment operator ensures that the data is written into consecutive memory locations (i.e., the warps in the next operators will not encounter any divergence). Thus, there are a total of 4^{n-1} possible nodes in the solution space.

Bounding Function. To limit the size of the solution space and to reduce the overhead of determining the optimal solution, the B&B based algorithm in Pyper makes use of a bounding function based on the cost of the kernels being generated by each node. Specifically, the augmentation optimizer keeps track of node with the current lowest execution cost. If any node generates a kernel with a single relational operator and the kernel has an execution cost greater than the current optimal node, then the node is marked as sub-optimal and its child nodes are not explored. This is because the overall execution cost of the pipeline represented by this node and its child nodes will be higher than the current optimal solution.

Pruning Criteria. We use the following pruning criteria or heuristics to reduce the number of nodes that needs to be explored.

- Due to the limited amount of resource available on the GPU, NVIDIA CUDA sets hard limits on the resources that can be utilized at the warp and thread block level (Equation 6.5). Hence, any node that requires the generation of a kernel that violates these resource constraints is marked as invalid and its child nodes are not explored. The reason is, if these resource constraints are violated, there is no choice but to segment the pipeline. Those solutions would be explored in another part of the tree, and thus we mark this node as invalid to avoid redundancy.
- Further, we do not attempt to insert the Shuffle or Segment operators into nodes in which all the generated kernels can achieve 100% execution efficiency or theoretical occupancy rate. This is because the insertion of operators cannot further improve the execution efficiency or occupancy rate and just brings runtime overhead.

Example. We demonstrate a working example of our B&B based optimization algorithm in Figure 6.7. The example shows a simple pipeline consisting of three relational operators, O_1 , O_2 and O_3 . In initialization, the root node is represented by $P = \{O_1, O_2, O_3\}$. As branching, we generate 3 different child nodes by insertions of a single Shuffle, a single Segment and a pair of Shuffle and Segment (denoted Node 2, Node 3 and Node 4 in the figure, respectively) between operators O_1 and O_2 . Their resource usage is illustrated on the right. As shown in the table corresponding to Node 2, the insertion of the Shuffle operator leads to an increased resource requirements and thus the violation of the resource constraints. Node 4 is marked as sub-optimal along with all its nodes due to the high cost of materialization when a pair of Shuffle and Segment operators are inserted into the pipeline. Finally, the algorithm chooses Node 3 as the optimal solution for further exploration if it has child nodes.

Runtime overhead. The B&B procedure is very efficient in practice, thanks to the effective pruning and the efficient evaluation of the cost model. In our experiments, the augmentation optimizer is able to identify the optimal solution for each query with very low overhead (smaller than 1% of the query execution time).

FIGURE 6.7: An example for augmentation optimization

6.6 Experiments

6.6.1 Experimental Setup

Hardware. To demonstrate the efficiency of Pyper across different hardware generations, we use V100, P100, and Titan Xp GPUs from NVIDIA. All three GPUs are connected to the CPU via x16 PCIe 3.0 interface. The summary of the specification of all three GPUs is provided in Table 6.2. V100 and P100 have HBM memory; Titan Xp has GDDR5 memory. We mainly present the results on V100 that is a more recent GPU architecture and present the results on different GPU architectures in Section 6.6.6. For comparison against CPU, we use a dual socket machine with 512 GB of main memory and two Intel Xeon E5-2698 v4 CPU with 20 physical cores (40 logical cores with hyperthreading) running at 2.20GHz.

	V100	P100	Titan Xp
Core Count	80 x 64	56 x 64	48 x 64
Clock Rate (GHz)	1.53	1.328	1.4
Memory Size (GB)	32	16	12
Memory Bandwidth (GB/s)	900	549	547.7
Memory Type	HBM2.0	HBM2.0	GDDR5
Shared Memory per SM	64KB	64 KB	64 KB
Register File per SM	256 KB	256 KB	256 KB

TABLE 6.2: Hardware used for experiments.

Workload. For our experiments, we use the TPC-H [227] and Star Schema Benchmark (SSB) [230] data sets with a scale factor of 50 (50 GB in size). SSB is a simplified version TPC-H benchmark and has been widely used in previous studies on OLAP. We use the entire set of SSB queries for evaluation. Since our system front-end is based on GPL, we do not currently support the processing of sub-queries and leave this for future work. Hence, we used all the eight queries in TPC-H which do not contain sub-queries (Q1, Q3, Q5, Q6, Q10, Q12, Q14 and Q19). The detailed SQL clauses can be found in their benchmark websites [227, 230].

All our experiments are based on the entire data that was already loaded on to the GPU global memory during initialization. As the rise in GPU memory capacity, as well as the increased capacity of hosting multiple GPUs (up to 20) on a modern server, we have witnessed hundreds of gigabytes of GPU memory on a single server, where many analytical workloads can be processed completely on GPUs.

Experimental Outline. First, we show the impact of our estimation on divergence and resource contention, as well as the effectiveness of augmentation optimizer. In order to demonstrate that impact, we have integrated our cost model and augmentation optimizer into a previous pipelined query processing engine on the GPU named GPL [4].

Second, we evaluate the overall performance benefit of Pyper with other implementations on the GPU. Here, we omit the comparison with the systems on the CPU such as Hyper [201], since the previous studies [3] have shown that their GPU-based systems are faster than Hyper. Specifically, we conduct a comprehensive comparison using the following GPU-based systems: Ocelot [9], GPL [4], Hawk [3], Omnisci [7] and DogQC [6]. Ocelot is a kernel based execution system that materializes the intermediate data between operators in the GPU global memory, whereas GPL is a pipelined query processing system that makes use of GPU L2 cache for moving the data between the operators. All other systems are compiled systems. To the best of our knowledge, Hawk [3] is the state-of-the-art opensource system that supports code generation for GPUs. However, even Hawk only supports earlier generations of GPUs and cannot run on the V100 GPU. Hence, we modified Hawk [3] for evaluation (denoted as Hawk-M) so that it can run on V100. Omnisci [7] is considered to be the state-of-the-art open-sourced commercial compiled system for GPUs.

FIGURE 6.8: Relative error in the execution cost estimation by Pyper and GPL for TPC-H and SSB queries on V100

Third, in Section 6.6.4, we evaluate the individual techniques in terms of efficiency and effectiveness of the Shuffle and Segment operators in reducing the divergence and resource contention among threads, respectively.

Fourth, in Section 6.6.6, we evaluate the generality of the Shuffle and the Segment operators along with the proposed cost model across different GPU hardware.

6.6.2 Model Evaluation

Key Findings

Our analytical model is highly accurate, with less than 6.6% error on average for the tested queries. Without consideration of divergent execution and resource contention in compiled systems, existing models have a much higher error.

Estimation errors. To demonstrate our analytical model’s accuracy, we compare the *relative error* associated with the estimation of the query execution cost by Pyper and GPL for the tested queries in Figure 6.8. We define the relative error as $\frac{|\hat{Q}_m - \hat{Q}_e|}{\hat{Q}_m}$, where $\hat{Q}_m$ and $\hat{Q}_e$ represent the total measured and estimated (by the analytical model) execution time of query Q . Note, the relative error values of Pyper presented in Figure 6.8 are from the query plan obtained from our augmentation optimizer. We also show the geometric mean of all the results as “Mean“ in the

figure. We have studied the relative error of other plans including those generated during the augmentation optimization, and observed similar results to Figure 6.8. We present one representative result later in Figure 6.9, and omit the results for other queries due to space limits.

The results in Figure 6.8 clearly show that Pyper is able to achieve significantly higher accuracy when compared to GPL, for most queries. If we take a close look at the comparison, we can observe that Pyper can achieve much lower relative errors than GPL in some queries, or have similar relative errors to GPL in other queries. To understand this, we take TPC-H Q1 and Q6 as two representative queries [227]. 1) TPC-H Q6 represents the cases where Pyper achieves much lower relative errors than GPL. TPC-H Q6 is a query with a low-selectivity filter where many tuples are filtered out and the operators encounter a high degree of divergence. Pyper is able to address this divergent execution, whereas GPL is not. 2) TPC-H Q1 represents the cases where Pyper has similar relative errors to GPL. This is because TPC-H Q1 is an aggregation query with higher selectivity (most tuples are remained as results in the pipeline), i.e., the operators in Q1 encounter minimal divergence during execution. As we show later in Section 6.6.4, the improvement of the model accuracy can come from our improved estimation on resource contention.

We analyze the source of the relative error of our model. One source can be that the cost of shared memory and global memory instructions can vary significantly based on the number of memory bank conflicts and access patterns, making it difficult for Pyper to accurately estimate this cost. The other potential source of errors can be from multiple joins, where join intensive queries like Q5, Q10 and Q4.1 compared to the other queries. This is because the join queries lead to errors in estimating the cost of synchronization in accessing the hash table in parallel.

Effectiveness of query augmentation optimizer. We demonstrate the effectiveness of our query plan augmentation optimizer. We present the measured and estimated execution time of all candidate solutions (30 in total after pruning) explored by the augmentation optimizer for TPC-H Q6 in Figure 6.9. The execution time is normalized with respect to that of GPL. We observe similar results for other queries and omit the results.

The results in Figure 6.9 clearly show the following observations. First, there is significant performance difference between the different candidate solutions (the

FIGURE 6.9: Normalized measured and estimated execution cost of different candidate solutions of Q6 on V100

FIGURE 6.10: Overall performance evaluation of Pyper for TPC-H queries on the V100 GPU.

difference can be 2.75x for TPC-H Q6). Hence, it is very important to accurately identify the optimal insertion points of Shuffle and Segment operators into a query plan. In fact, an inaccurate estimation of the execution time leading to the selection of a non-optimal configuration could lead to severe degradation in query performance. Second, the augmentation optimizer in Pyper with the help of the analytical model is able to identify the optimal solution for Q6 (highlighted by the rectangle of Pyper in Figure 6.9), thus demonstrating its effectiveness. The augmentation optimization is quite lightweight due to the B&B approach, with the runtime overhead smaller than 1% of the query execution time.

FIGURE 6.11: Overall performance evaluation of Pyper for SSB queries on the V100 GPU.

FIGURE 6.12: Time breakdown of Pyper and the Hawk-M on V100

6.6.3 Overall Comparison

Key Findings

For TPC-H queries, on average the code generated by Pyper outperforms Ocelot, Omnisci, GPL, Hawk-M and DogQC by 34.73x, 5.01x, 3.06x, 1.60x and 1.40x, respectively. Pyper on average achieves 6.59x, 2.55x, 2.35x, 1.52x performance improvement over Ocelot, Omnisci, GPL and Hawk-M for SSB queries.

We compare the overall performance of Pyper against other systems using TPC-H and SSB queries in Figures 6.10 and 6.11, respectively. Compiled systems are generally much faster than non-compiled systems.

Omnisci is a commercial system with a more complete set of functionalities than other systems in the test. It is still slower than other tested systems, due to its low execution efficiency of the GPU kernels. Further, Omnisci also fails to efficiently optimize complex join queries like Q3, Q5 and Q10, as demonstrated by the significantly worse performance of Omnisci for these queries. Overall, Pyper outperform Omnisci by 5.01x and 2.55x for TPC-H and SSB queries on average.

As mentioned before, the efficient use of the Shuffle and Segment operators helps Pyper minimize the divergent execution and resource contention among threads. This combined with the availability of a cost model that avoids unnecessary insertions of the Shuffle and Segment operators help Pyper to outperform Hawk-M by 1.60x and 1.55x on average for TPC-H and SSB queries respectively.

Even though DogQC segments its pipeline and makes use of techniques to minimize workload divergence and resource contention, Pyper still manages to outperform DogQC by up to 1.4x on average (up to 3.9x on TPC-H queries), due to the following reasons. First, DogQC lacks a proper cost model that can help determine the optimal insertions of the Shuffle and Segment operators (the importance has been demonstrated by the results in Figure 6.9). Second, Pyper adopts more efficient implementations for the Shuffle and Segment operators. Note, we omit DogQC in Figure 6.11 since the open-source version of DogQC does not currently support SSB queries.

Time Breakdown. To get a deeper understanding of the reasons behind the performance improvement of Pyper, we present the execution time breakdown of Pyper along with Hawk-M in Figure 6.12. The breakdown involves the overhead of divergent execution, the overhead of resource contention, the cost of Shuffle and Segment (including data shuffling resulting from Shuffle and the cost of additional data materialization arising from Segment). For the clarify of figure, Qx(H) and Qx(P) represent the execution time breakdown of code generated corresponding to query Qx by Hawk-M and Pyper, respectively. We have the following observations.

FIGURE 6.13: Normalized (to Hawk-M) warp execution and occupancy rate for TPC-H and SSB queries on V100

First, due to the use of the Shuffle and Segment operators, Pyper is able to generate code that almost eliminates the overhead due to divergent execution and resource contention among threads.

Second, the use of the Shuffle and Segment operator adds very little runtime overhead associated with data shuffling and materialization (less than 12%). Thus, such small overhead is much smaller than the performance improvement by resolving divergent execution and resource contention among the threads.

6.6.4 Individual Operator Evaluation

Key Findings

Key Findings: The detailed profiling results show that, with Shuffle and Segment operators, Pyper achieves up to 2.20x higher warp execution efficiency and 2.30x higher occupancy rate on GPUs than Hawk-M.

For a more thorough comparison of individual operators, we present the execution time of Pyper (w/o Shuffle) and Pyper (w/o Segment), which are the same as Pyper except without using Shuffle and Segment, respectively. The results are shown in Figure 6.14. Both Shuffle and Segment individually helps improve the performance of query execution on GPUs. Further, the benefit of each operator individually varies depending on the query.

FIGURE 6.14: Performance benefit of Shuffle and Segment operators for TPC-H queries on V100.

In the following, for each operator, we present the detailed profiling results to demonstrate that Shuffle and Segment operators are able to eliminate most of the overhead from divergent execution and resource contention.

We use the following metrics collected using the NVProfile tool from NVIDIA: 1) *warp execution efficiency* which represents the average number of active threads per warp and helps account for the level of divergence among the GPU threads and 2) *achieved occupancy* which represents the maximum number of active warps per SM and helps account of the resource contention among the threads. Higher the warp execution efficiency and achieved occupancy indicate lower levels of divergence and resource contention among threads, respectively. We present the normalized warp execution efficiency and occupancy values of Pyper with respect to Hawk-M in Figure 6.13.

Impact of Shuffle Operator

As demonstrated by the normalized warp execution efficiency values in Figure 6.13, the use of the Shuffle operator helps improve the warp execution efficiency of almost all queries used in our study. Overall, the warps executing the code generated by Pyper achieves up to 2.20x higher warp execution efficiency compared to the Hawk-M implementation. Such significant improvement in the warp execution efficiency of the kernels means that threads executing the code generated by Pyper can make

Query	Shared Mem. / Warp (KB)		Register / Warp (KB)		Occupancy (%)	
	Hawk-M	Pyper	Hawk-M	Pyper	Hawk-M	Pyper
Q1	0.1	0.1	4.5	3.8	55.2	93.1
Q5	1.3	1.0	4.4	3.1	58.1	93.2
Q10	1.3	0.9	3.3	3.1	56.8	90.0
Q2.3	1.1	0.6	3.8	3.5	60.2	92.5
Q3.2	1.5	1.0	4.3	3.7	49.0	88.3
Q4.1	1.1	0.8	4.0	3.8	47.3	89.1
Q4.3	1.3	1.0	4.5	3.2	40.1	93.0

TABLE 6.3: Resource requirements of Hawk-M and Pyper on V100

FIGURE 6.15: Impact of Grid Shuffle on P_1 for varying levels of thread execution efficiencies on V100.

more efficient use of the GPU hardware by allowing a larger number of active threads in each warp.

Threshold value of Shuffle. As mentioned before, the threshold value of the Shuffle operator is set by finding the right balance between the overhead of the Shuffle operation and the benefit of the Shuffling (in terms of reduced divergence). We set it to 1.5% in our original submission as we found that the value works well for the set of hardware under consideration (V100). We have expanded our experiments to included different GPU hardware in our re-submitted version and hence the threshold for each hardware is determined during the initial profiling phase which is executed on each GPU during system startup. The results for varying this threshold value on V100 is shown in Figure 6.15.

FIGURE 6.16: The query plan augmentation process of Q6 (notations can be found in Table 6.1)

Finally, we study the use of the Shuffle operator to augment the query plan using TPC-H Q6 as example (Figure 6.16). As shown in the example figure, since filter1 and filter2 have very low selectivity (S_i shows the selectivity of operator O_i), severe divergent executions occur in filter2 and filter3. Thus, the augmentation optimizer decides to insert Shuffle after filter1 and filter2.

Impact of Segment Operator

As shown in Figure 6.13, Pyper is able to achieve up significantly higher occupancy rate when compared to Hawk-M for queries Q1, Q5, Q10, Q2.3, Q3.2, Q4.1 and Q4.3. This is because one or more kernels generated by the Hawk-M implementation for these queries encounter severe resource contention on GPUs due to its high resource requirement. Pyper breaks down those large pipelines in these queries using the segment operator, thus reducing the resource requirements of the kernels and improving the parallelism.

We present the resource requirements of the most expensive kernel generated by the Hawk-M implementation and Pyper for queries Q1, Q5, Q10, Q2.3, Q3.2, Q4.1 and Q4.3 in Table 6.3. The shared memory and register requirements of the queries in Table 6.3 clearly show that the kernels generated by Pyper has significantly lower resource requirements compared to Hawk-M. More importantly, the resource

FIGURE 6.17: The query plan augmentation process of Q1 (notations can be found in Table 6.1)

requirements in Pyper are low enough to ensure up to 64 simultaneous active warps per SM on modern GPU hardware (with 64 KB shared memory and 256 KB registers per SM). Hence, the kernels generated by Pyper is able to achieve up to 2.3x better occupancy rate on GPUs (Figure 6.13).

An example of using Segment to augmenting the query plan, we use TPC-H Q1 (Figure 6.17). As shown in the example figure, since the two aggregation operators have very high usage of shared memory (in the hash table), there is severe resource contention which leads to low thread parallelism. Thus, the augmentation optimizer decides to insert Segment between the two aggregation operators and splits the pipeline into two.

Impact of Intermediate Data

Difference sizes of intermediate data between the pipelines can impact the benefit of the Shuffle and Segment operator (Equation 1). For example, the size and distribution of intermediate data can impact 1) the amount of divergence, 2) the cost of workload re-distribution in Shuffle and 3) the cost of the Segment operator. Our operator implementations are agnostic to the size of the intermediate data since our cost model takes this into account (Equations 1 and 3) while determining the optimal insertion points for the Shuffle and segment operators.

FIGURE 6.18: Impact of selectivity on the benefit of Shuffle and Segment operator insertions into P_1 on V100

FIGURE 6.19: Performance evaluation on P100 and Titan Xp

We have studied performance improvement of our pipeline P_1 due to the insertions of the Shuffle and Segment operators when the selectivity of the first probe is varied from 100% to 25%. The results are shown in Figure 6.18. Note, the performance is normalized with respect to the implementation without any Shuffle or Segment operators. First, the performance benefit of the Shuffle operator increases with the level of divergence as demonstrated by the highest performance gain for a selectivity of 50%. Second, the performance benefit of the Segment operator increases with the decreasing selectivity. Regardless of the selectivity, the Segment operator helps improve the occupancy rate of kernels. But the amount of intermediate results decides the overhead of the Segment operator.

FIGURE 6.20: Performance comparison of Pyper on V100 GPU against Omnisci CPU on Intel Xeon E5-2698 v4 .

6.6.5 Evaluation on Different GPU Architectures

We have conducted experiments on two older generation of GPUs including P100 and Titan Xp. The results of TPC-H queries are shown in Figure 6.19. Overall, the results demonstrate the efficiency of Pyper and the generality of both our operators and the cost model for different generations of GPU hardware. The slightly lower performance gain of Pyper is mainly due to the reduced performance benefit of the Segment operator.

6.6.6 Evaluation Against CPUs

We compare the performance of Pyper running on the V100 against Omnisci CPU, which is the state-of-the-art commercial CPU database system, using 6 TPC-H queries. For this experiment, Omnisci CPU is running on an Intel Xeon E5-2698 v4 CPU with 40 logical cores. As shown by the results in Figure 6.20, due to the availability of large number of parallel cores, high memory bandwidth and efficient query execution on GPUs, Pyper outperforms Omnisci CPU by 14x on average.

6.7 Summary

JIT compilation-based code generation of queries has evolved as a powerful paradigm for query processing on GPU. In this study, I find that, thanks to advanced techniques such as database compression and JIT compilation, memory stalls are no longer the most significant bottleneck. Instead, existing compiled systems encounter severe overhead in divergent execution and resource contention because they adopt a monolithic approach for generating the code of query plan on the GPU. To improve the execution efficiency, I present *Pyper*, a compiled system designed for GPUs. Pyper introduces two new query plan transformation operators, *Shuffle* and *Segment*, which can be inserted into any query pipeline of the physical plan to reduce the divergent execution and resource contention. Further, Pyper makes use of an analytical cost model and an effective optimizer to determine the optimal use of the Shuffle and Segment operators. Our experiments show that Pyper generates code that on average is able to improve the performance of TPC-H queries by 34.73x, 5.01x, 3.06x, 1.60x and 1.40x when compared to Ocelot, Omnisci, GPL, Hawk and DogQC, respectively. Pyper on average achieves 6.59x, 2.55x, 2.35x, 1.52x performance improvement over Ocelot, Omnisci, GPL and Hawk for SSB queries.

Chapter 7

Conclusion & Future Work

In this chapter, we first highlight our novel contribution to *in-memory analytical query processing on GPUs*. We then outline some interesting research topics for the future.

7.1 Contributions

The main contributions of this thesis are as follows:

- **Revisiting Hash Join on Graphics Processors: A Decade Later.** In this study, we revisited the major GPU hash join implementations in the last decade and detailed how they take advantage of different GPU architecture features. Our study found that, by efficiently taking advantage of GPU architecture features like CUDA Streams and shared memory atomic operations, the PRBC implementation outperforms all other existing GPU hash join implementations for non-GPU resident data sets. While for data sets that reside within the GPU, implementations like BSH are capable of achieving similar performance to PRBC. The study also sheds light on the impact of different architecture features on the hash join operation and identified a number of factors guiding the choice of these features. Through an in-depth analysis of existing implementations, we find that 1) efficient use of native shared memory atomic instructions can help improve the occupancy rate of GPU kernels by up to 3.6x, 2) overlapping data transfer with the help

of CUDA streams can improve performance by up to 2x and 3) the use of the new unified memory feature without pre-fetching of data can lead to up to 50% lower PCIe data transfer efficiency. Finally, the techniques proposed in this study helped avoid performance degradation when joining data sets with high skew and achieved up to 1.27x performance improvement when joining data sets with a high match rate.

- MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architecture.** The rise of multi-GPU architectures with faster interconnects offer new opportunities of relational query processing on hundreds of gigabytes of GPU memory. However, we find that due to the unique characteristics of GPUs and the interconnect hardware, existing hash join designs demonstrate very bad scalability on multi-GPU architectures, which spend up to 66% of its execution time in cross-GPU communications and achieve lower than 50% GPU-GPU interconnect bandwidth utilization. To address this, we propose MG-Join, a scalable partitioned hash join implementation for modern multi-GPU hardware. MG-Join makes use of an adaptive multi-hop data distribution policy that helps ensure efficient use of GPU hardware and interconnect links and minimizes congestion across interconnect links. Our experiments show that MG-Join achieves up to 97% utilization of the interconnect bandwidth on DGX-1 with eight GPUs, and outperforms existing partitioned hash join implementations by up to 2.5x when joining large input data sets distributed over multiple GPUs. MG-Join further helps improve the performance of TPC-H queries by up to 4.5x over OmniSci.
- Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs.** JIT compilation-based code generation of queries has evolved as a powerful paradigm for query processing on GPU. In this study, we find that thanks to advanced techniques such as database compression and JIT compilation, memory stalls are no longer the most significant bottleneck. Instead, existing compiled systems encounter severe overhead in divergent execution and resource contention because they adopt a monolithic approach for generating the code of query plan on the GPU. To improve the execution efficiency, we developed Pyper, a compiled system designed for GPUs. Pyper introduced two new query plan transformation operators, Shuffle and Segment, which can be inserted into any query pipeline of the physical plan to reduce the divergent execution and resource contention.

Further, Pyper makes use of an analytical cost model and an effective optimizer to determine the optimal use of the Shuffle and Segment operators. Our experiments show that Pyper generates code that on average is able to improve the performance of TPC-H queries by 34.73x, 5.01x, 3.06x, 1.60x and 1.40x when compared to Ocelot, Omnisci, GPL, Hawk and DogQC, respectively. Pyper on average achieves 6.59x, 2.55x, 2.35x, 1.52x performance improvement over Ocelot, Omnisci, GPL and Hawk for SSB queries.

7.2 Future Work

As mentioned before, its high global memory bandwidth, massively parallel architecture and the recent availability of high bandwidth communication interfaces have pushed GPUs to the forefront of high performance compute applications like OLAP systems. While the techniques proposed in this thesis have helped improve the performance of GPU database systems, both at the operator and query level, many challenges remain to be addressed. In the remainder of this section, we list some of the attractive future research directions.

7.2.1 Query Execution on Multi-GPU Architectures

In recent years, GPU vendors have been trying to push the adoption of GPUs as primary computing devices through the integration of multiple GPUs into a single machine and the introduction of higher bandwidth interconnects like NVLink 2.0 [45, 132, 133]. In this thesis, we already explored the design of a novel partitioned hash join implementation for such multi-GPU architectures (Chapter 5). Further, Guo et al. [200] proposed a join implementation for distributed GPU clusters. More recently, Lutz et al. [44] proposed a non-partitioned hash join implementation for multi-GPU architectures with low latency CPU-GPU interconnects.

However, existing studies are limited and only aim to optimize the execution of a small set of operators on modern multi-GPU architectures; while optimizing the entire query execution pipeline on these architectures has still not been studied. Optimizing the overall execution of a query instead of individual operators, on a multi-GPU architecture is challenging due to the following reasons. First, to make

efficient use of all available GPU hardware resources, both placement of operators and data needs to be optimized, taking into consideration the entire query plan that needs to be executed. Second, novel fine-grained communication schemes/policies need to be developed for efficient collaborative execution of the query plan by different GPUs. Hence significant research effort is needed in studying and developing GPU database systems for modern multi-GPU architectures.

7.2.2 Optimizing for the New Bottleneck

When executing current GPU architectures, communication remains a key bottleneck for most relational operators like join, sort and aggregate [5, 12, 44]. However, a major focus of GPU hardware improvement in recent years has been the communication interfaces. Over the past decade alone, we have seen up to a 10x improvement in CPU-GPU and GPU-GPU interconnect bandwidth, from 16x PCIe 2.0 to 3x NVLink 2.0 interconnects. Further, PCIe 4.0 devices are already becoming available and NVLink 3.0 interconnect is set to become available in the near future. These interconnects offer theoretical data transfer bandwidths of 32 GB/s [211] and 50 GB/s [235], respectively; which is higher than the compute bandwidth achieved by most relational operators, including hash join [12, 32].

Hence, with the availability of these new hardware interconnects, the bottleneck is expected to shift from communication to computation. However, almost all existing GPU database systems are designed around the communication bottleneck. Hence, this hardware improvement would necessitate a complete redesign of many aspects of existing GPU database systems, both at the operator and at the query level. This redesign needs to take into account the performance characteristics of different relational operators as well as the underlying hardware and optimize each one for the specific bottleneck. Also, the expected reduction in the latency for these new interconnects along with the support for cache coherence across CPUs and GPUs [44] would further drive the redesign on new GPU database systems.

7.2.3 GPUs in Edge Computing

Due to their high energy requirements, the use of GPUs were often limited to high performance data centers where there is ample power available. However, due to

the explosive growth in the amount of data being collected, the increasing demand for low latency data processing and the limited growth in the network bandwidth it is becoming essential to efficiently process the data at the source where its being generated, without having to transmit it to a centralized data center. To enable this vendors like NVIDIA have been introducing GPU embedded edge devices like the EGX A100 [236]. Being on the edge of the network, these devices also provide additional benefits in terms of privacy, bandwidth efficiency and scalability.

However, due to the inherent design characteristics of these edge computing devices and the limited power available to them, it is not often possible to simply port over existing high performance applications to these devices. This is because existing application are often designed to operate on substantial computation resources and are not optimized for lower power devices. Hence GPU applications, including analytical systems will need to be re-designed from the ground up to enable efficient use of such hardware devices[237].

List of Publications

- **Johns Paul**, Bingsheng He, Shengliang Lu and Chiew Tong Lau. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. *Currently under review in SIGMOD 2021.*
- **Johns Paul**, Bingsheng He, Shengliang Lu and Chiew Tong Lau. Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs. *In Proceedings of the 2021 VLDB Endowment.*
- **Johns Paul**, Jieliang Ang, Tianyuan Fu, Bingsheng He, Shengliang Lu, Sien Yi Tan and Feng Cheng. TraV: Poet: an Interactive Spatial Query Processing System in Grab. *In 2020 International Conference on Advances in Geographic Information Systems (SIGSPATIAL).*
- Zhang, Shuhao, Feng Zhang, Yingjun Wu, Bingsheng He, and **Johns Paul**. Hardware-conscious stream processing: A survey. *In 2020 ACM SIGMOD Record 48.*
- Jieliang Ang, Tianyuan Fu, **Johns Paul**, Shuhao Zhang, Bingsheng He, Teddy Sison David Wenceslao, and Sien Yi Tan. **TraV**: An Interactive Exploration System for Massive Trajectory Data. *In 2019 IEEE Fifth International Conference on Multimedia Big Data (BigMM).*
- **Johns Paul**, Bingsheng He, Shengliang Lu and Chiew Tong Lau. Revisiting hash join on graphics processors: a decade later. *In 2020 Distributed Parallel Databases.*
- **Johns Paul**, Bingsheng He, Shengliang Lu and Chiew Tong Lau. Revisiting Hash Join on Graphics Processors: A Decade Later. *In 2019 IEEE International Conference on Data Engineering Workshops (ICDEW).*

- **Johns Paul**, Bingsheng He and Chiew Tong Lau. Query Processing on OpenCL-Based FPGAs: Challenges and Opportunities. *In 2018 IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.
- Zeke Wang, **Johns Paul**, Bingsheng He, and Wei Zhang. Multikernel Data Partitioning With Channel on OpenCL-Based FPGAs. *In 2017 IEEE Transactions on Very Large Scale Integration (VLSI)*.
- **Johns Paul**, Jiong He, and Bingsheng He. GPL: A GPU-based Pipelined Query Processing Engine. *In Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*.
- Zeke Wang, **Johns Paul**, Huiyan Cheah, Bingsheng He, and Wei Zhang. Relational query processing on OpenCL-based FPGAs. *In 2016 International Conference on Field Programmable Logic and Applications (FPL)*.
- Zeke Wang, Huiyan Cheah, **Johns Paul**, Bingsheng He, and Wei Zhang. Accelerating Database Query Processing on OpenCL-based FPGAs (Abstract Only). *In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

Bibliography

- [1] Big Data Market by Component (Solutions and Services), Function (Finance, Human resources, Marketing and Sales, Operations, and Others), Industry Vertical (IT & Telecom, BFSI, Retail, Healthcare, Government and Defense, Manufacturing, and Others), and Reg. URL <https://www.adroitmarketresearch.com/contacts/request-sample/1437>. 1
- [2] Continuing the Scaling of Digital Computing Post Moore’s Law:. URL <https://escholarship.org/uc/item/1rk5c69g>. 1
- [3] Sebastian BreB, Bastian Kocher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *The VLDB Journal*, 27(6):797–822, dec 2018. ISSN 1066-8888. doi: 10.1007/s00778-018-0512-y. URL <https://doi.org/10.1007/s00778-018-0512-y>. 2, 3, 5, 23, 27, 29, 30, 85, 87, 108
- [4] Johns Paul, Jiong He, and Bingsheng He. GPL: A GPU-based Pipelined Query Processing Engine. *Proceedings of the 2016 International Conference on Management of Data - SIGMOD ’16*, pages 1935–1950, 2016. ISSN 07308078. doi: 10.1145/2882903.2915224. URL <http://dl.acm.org/citation.cfm?doid=2882903.2915224>. 3, 20, 23, 26, 28, 29, 89, 97, 99, 101, 102, 108
- [5] Periklis Chrysogelos, Manos Karpapthiotakis, Raja Appuswamy, and Anastasia Ailamaki. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proceedings of the VLDB Endowment*, page 13, 2019. URL <http://infoscience.epfl.ch/record/262531>. 3, 5, 19, 27, 29, 30, 31, 85, 126

- [6] Henning Funke and Jens Teubner. Data-Parallel Query Processing on Non-Uniform Data. *Proc. VLDB Endow.*, 13(6):884–897, feb 2020. ISSN 2150-8097. doi: 10.14778/3380750.3380758. URL <https://doi.org/10.14778/3380750.3380758>. 2, 3, 23, 27, 30, 75, 108
- [7] Omnisci. Accelerated Analytics Platform. <https://www.omnisci.com/>, 2009. 2, 5, 31, 61, 75, 87, 108
- [8] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1617–1632, New York, NY, USA, jun 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3380595. URL <https://dl.acm.org/doi/10.1145/3318464.3380595>. 21, 85
- [9] Max Heimerl, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013. ISSN 21508097. doi: 10.14778/2536360.2536370. URL <http://dl.acm.org/citation.cfm?doid=2536360.2536370>. 2, 3, 20, 23, 26, 28, 58, 62, 63, 75, 85, 108
- [10] Ran Rui, Hao Li, and Yi-Cheng Tu. Join algorithms on GPUs: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2541–2550. IEEE, 2015. 2, 24, 33, 44, 57, 58, 62, 63, 64, 65, 74
- [11] Tim Kaldewey. GPU Join Processing Revisited. 3, 20, 21, 24, 28, 44, 79, 85
- [12] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpapothakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious Hash-Joins on GPUs. URL http://infoscience.epfl.ch/record/262530/files/sioulas{}_icde2019{}_gpujoin.pdf. 2, 21, 24, 33, 35, 36, 37, 38, 44, 57, 62, 65, 74, 79, 126
- [13] Tobias Lauer, Amitava Datta, Zurab Khadikov, and Christoffer Anselm. Exploring graphics processing units as parallel coprocessors for online aggregation. In *International Conference on Information and Knowledge Management, Proceedings*, pages 77–84, New York, New York, USA, 2010.

- ACM Press. ISBN 9781450303835. doi: 10.1145/1871940.1871958. URL <http://portal.acm.org/citation.cfm?doid=1871940.1871958>. 2, 25
- [14] Tomas Karnagel, Rene Mueller, and Guy M Lohman. Optimizing GPU-accelerated Group-By and Aggregation. Technical report. URL <http://www.adms-conf.org/2015/gpu-optimizer-camera-ready.pdf>. 25
- [15] Diego Tomé, Tom Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter Boncz. Optimizing Group-By And Aggregation using GPU-CPU Co-Processing. Technical report. 26
- [16] Tim Gubner Cwi and Peter Boncz. Exploring Query Execution Strategies for JIT, Vectorization and SIMD. Technical report. URL <https://t1mm3.github.io/assets/papers/adms17.pdf>. 2, 26
- [17] Nadathur Satish, Mark Harris, and Michael Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. Technical report. URL <https://ieeexplore.ieee.org/abstract/document/5161005/>. 2, 25
- [18] Andrew Davidson, David Tarjan, Michael Garland, and John D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *2012 Innovative Parallel Computing, InPar 2012*, 2012. ISBN 9781467326322. doi: 10.1109/InPar.2012.6339592.
- [19] Erik Sintorn and Ulf Assarsson. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. Technical report. URL <https://www.sciencedirect.com/science/article/pii/S0743731508001196>. 25
- [20] Duane G. Merrill and Andrew S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, volume 2010, pages 545–546, New York, New York, USA, 2010. Institute of Electrical and Electronics Engineers Inc. ISBN 9781450301787. doi: 10.1145/1854273.1854344. URL <http://portal.acm.org/citation.cfm?doid=1854273.1854344>. 2, 25
- [21] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524. ACM, 2008. 2, 24, 28, 33, 35, 58, 62

- [22] P Ghodsnia. An in-GPU-memory column-oriented database for processing analytical workloads. 3, 20
- [23] Isaac Gelado, Javier Cabezas, Nacho Navarro, John E. Stone, Sanjay Patel, and Wen Mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *ACM SIGPLAN Notices*, 45(3):347–358, mar 2010. ISSN 15232867. doi: 10.1145/1735971.1736059. URL <https://dl.acm.org/doi/10.1145/1735971.1736059>. 3, 20
- [24] Michela Becchi, Surendra Byna, Srihari Cadambi, and Srimat Chakradhar. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 82–91, New York, New York, USA, 2010. ACM Press. ISBN 9781450300797. doi: 10.1145/1810479.1810498. URL <http://portal.acm.org/citation.cfm?doid=1810479.1810498>. 20
- [25] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPURT-eraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 325–336, 2006. ISBN 1595934340. doi: 10.1145/1142473.1142511. URL <http://gamma.cs.unc.edu/GPUTERASORT>. 3, 20
- [26] Piotr Przymus and Krzysztof Kaczmarek. Dynamic compression strategy for time series database using GPU. In *Advances in Intelligent Systems and Computing*, volume 241, pages 235–244. Springer Verlag, 2014. ISBN 9783319018621. doi: 10.1007/978-3-319-01863-8_26. URL https://link.springer.com/chapter/10.1007/978-3-319-01863-8_{_}26. 3, 22
- [27] Piotr Przymus and Krzysztof Kaczmarek. Compression Planner for Time Series Database with GPU Support. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8920, pages 36–63. Springer Verlag, 2014. ISBN 9783662457603. doi: 10.1007/978-3-662-45761-0_2. URL https://link.springer.com/chapter/10.1007/978-3-662-45761-0_{_}2. 22
- [28] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache query co-processing on coupled CPU-GPU architectures. *Proceedings of the VLDB Endowment*, 8(4):329–340, 2014. 5, 22, 28, 31, 33, 58, 62, 63, 85, 87, 97

- [29] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. Light loss-less data compression, with GPU implementation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10048 LNCS, pages 281–294. Springer Verlag, 2016. ISBN 9783319495828. doi: 10.1007/978-3-319-49583-5_22. [3](#), [22](#)
- [30] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1603–1618, New York, New York, USA, may 2018. Association for Computing Machinery. ISBN 9781450317436. doi: 10.1145/3183713.3183734. URL <http://dl.acm.org/citation.cfm?doid=3183713.3183734>. [3](#), [23](#), [29](#)
- [31] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE, 2012. [3](#), [23](#), [25](#), [29](#), [33](#)
- [32] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Revisiting Hash Join on Graphics Processors: A Decade Later. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, pages 294–299. IEEE, 2019. [4](#), [35](#), [37](#), [57](#), [58](#), [61](#), [62](#), [75](#), [79](#), [96](#), [126](#)
- [33] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Revisiting hash join on graphics processors: a decade later. *Distributed and Parallel Databases*, pages 1–23, jan 2020. ISSN 15737578. doi: 10.1007/s10619-019-07280-z. URL <https://link.springer.com/article/10.1007/s10619-019-07280-z>. [4](#)
- [34] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. *dl.acm.orgPaperpile*, 2017. doi: 10.1145/3018743.3018756. URL <http://dx.doi.org/10.1145/3018743.3018756>. [4](#), [32](#), [66](#)
- [35] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. *dl.acm.orgPaperpile*, 53(4):752–768, jun 2018. doi: 10.1145/

- 3192366.3192404. URL <https://doi.org/10.1145/3192366.3192404>. 4, 32
- [36] Zhihao Jia, Yongkee Kwon, Galen Shipman LANL, Pat M Cormick LANL pat, lanlgov Mattan Erez, Alex Aiken, Galen Shipman, Pat M Cormick, and Mattan Erez. A Distributed Multi-GPU System for Fast Graph Processing. *dl.acm.orgPaperpile*, 11(3):2977–310, 2017. doi: 10.14778/3157794.3157799. URL https://dl.acm.org/ft_gateway.cfm?id=3173079&type=pdf. 4, 32
- [37] NVIDIA Collective Communication Library (NCCL) Documentation — NCCL 2.7.1 documentation. URL <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html>. 4, 31
- [38] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment*, 9(14):1707–1718, 2016. 5, 23
- [39] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. Hardware-conscious Query Processing in GPU-accelerated Analytical Engines. *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research*, page 9, 2019. URL <http://infoscience.epfl.ch/record/262529>. 5, 10, 29, 30, 57, 85, 89
- [40] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, pages 1603–1618, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3183734. URL <http://doi.acm.org/10.1145/3183713.3183734>. 5, 29, 30, 85
- [41] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *Proceedings of the VLDB Endowment*, 6(10):889–900, 2013. 5, 23, 29, 31, 33, 58, 62, 63, 87, 97
- [42] Modern Data Centers to Accelerate HPC Workloads NVIDIA, . URL <https://www.nvidia.com/en-us/data-center/>. 7
- [43] Welcome to AMD High-Performance Processors and Graphics, . URL <https://www.amd.com/en>. 7, 31

- [44] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1633–1649. Association for Computing Machinery, jun 2020. ISBN 9781450367356. doi: 10.1145/3318464.3389705. [10](#), [11](#), [21](#), [23](#), [31](#), [32](#), [125](#), [126](#)
- [45] NVIDIA DGX-1, . URL <https://www.nvidia.com/en-us/data-center/dgx-1/>. [10](#), [21](#), [58](#), [125](#)
- [46] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, jan 2020. ISSN 15582183. doi: 10.1109/TPDS.2019.2928289. [11](#), [31](#)
- [47] DGX White Paper: NVIDIA, . URL <https://www.nvidia.com/en-us/data-center/resources/dgx-1-system-architecture-whitepaper/>. [11](#)
- [48] POWER8 with NVIDIA NVLink Technology. URL https://www-355.ibm.com/systems/power/openpower/tgcmDocumentRepository.xhtml?aliasId=POWER8{_}with{_}NVIDIA{_}NVLink. [11](#)
- [49] A B MySQL. The world’s most popular open source database. *MySQL AB*, 2007. URL <https://ci.nii.ac.jp/naid/10016599862>. [17](#), [18](#)
- [50] Database Cloud: Oracle. URL <https://www.oracle.com/database/>. [17](#)
- [51] PostgreSQL: The world’s most advanced open source database. URL <https://www.postgresql.org/>. [17](#), [18](#)
- [52] Db2 Database - Overview - Singapore : IBM. URL <https://www.ibm.com/sg-en/products/db2-database>. [17](#)
- [53] Dennis G. Severance. A practitioner’s guide to data base compression tutorial. *Information Systems*, 8(1):51–62, jan 1983. ISSN 03064379. doi: 10.1016/0306-4379(83)90030-3. [17](#)
- [54] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the*

- ACM SIGMOD International Conference on Management of Data*, pages 671–682, New York, New York, USA, 2006. ACM Press. ISBN 1595934340. doi: 10.1145/1142473.1142548. URL <http://portal.acm.org/citation.cfm?doid=1142473.1142548>.
- [55] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, jan 1999. ISSN 00104620. doi: 10.1093/comjnl/42.3.193. URL <https://academic.oup.com/comjnl/article/42/3/193/336407>.
- [56] Application of data compression to a large bibliographic data base : Proceedings of the seventh international conference on Very Large Data Bases - Volume 7. URL <https://dl.acm.org/doi/10.5555/1286831.1286873>. 17
- [57] J Goldstein, R Ramakrishnan ... Conference on Data ..., and undefined 1998. Compressing relations and indexes. *ieeexplore.ieee.orgPaperpile*. URL <https://ieeexplore.ieee.org/abstract/document/655800/>. 17
- [58] Jiawei Han, Zhaohui Xie, and Yongjian Fu. Join Index Hierarchy: An Indexing Structure for Efficient Navigation in Object-Oriented Databases. Technical report. URL <https://ieeexplore.ieee.org/abstract/document/761666/>.
- [59] Gary Valentin, Michael Zuliani, Daniel C Zilio, Guy Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. Technical report. URL <https://ieeexplore.ieee.org/abstract/document/839397/>.
- [60] Goetz Graefe. Sorting And Indexing With Partitioned B-Trees. Technical report. URL <http://www-db.cs.wisc.edu/cidr/cidr2003/program/p1.pdf>. 17
- [61] S Robbins InfoQ News and Undefined 2008Paperpile. RAM is the new disk. 18
- [62] Peter A Boncz, Stefan Manegold, Martin L Kersten, and Others. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999. 18, 24

- [63] Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*. URL <https://dl.acm.org/doi/10.5555/2591305.2591307>. 18
- [64] Redis. URL <https://redis.io/>.
- [65] memcached - a distributed memory object caching system. URL <https://memcached.org/>. 18
- [66] Bin Cui, Beng Chin Ooi, Jianwen Su, and Kian-Lee Tan. Contorting high dimensional data for efficient main memory KNN processing. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03*, page 479, New York, New York, USA, 2003. ACM Press. ISBN 158113634X. doi: 10.1145/872757.872815. URL <http://portal.acm.org/citation.cfm?doid=872757.872815>. 18
- [67] Jun Rao and Kenneth A. Ross. Making B+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data - SIGMOD '00*, pages 475–486, New York, New York, USA, 2000. Association for Computing Machinery (ACM). doi: 10.1145/342009.335449. URL <http://portal.acm.org/citation.cfm?doid=342009.335449>.
- [68] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings - International Conference on Data Engineering*, pages 38–49, 2013. ISBN 9781467349086. doi: 10.1109/ICDE.2013.6544812.
- [69] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *Proceedings - International Conference on Data Engineering*, pages 302–313, 2013. ISBN 9781467349086. doi: 10.1109/ICDE.2013.6544834.
- [70] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Designing fast architecture-sensitive tree search on modern multicore/many-core processors. In *ACM Transactions on Database Systems*, volume 36, pages 1–34, dec 2011. doi: 10.1145/2043652.2043655. URL <https://dl.acm.org/doi/10.1145/2043652.2043655>.

- [71] Bin Cui, Beng Chin Ooi, Jianwen Su, and Kian Lee Tan. Main memory indexing: The case for BD-tree. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):870–874, jul 2004. ISSN 10414347. doi: 10.1109/TKDE.2004.1318568.
- [72] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*. URL <https://dl.acm.org/doi/10.5555/645925.671362>.
- [73] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*. URL <https://dl.acm.org/doi/10.5555/645913.671312>. 18
- [74] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77–85, dec 2008. ISSN 00010782. doi: 10.1145/1409360.1409380. URL <https://dl.acm.org/doi/10.1145/1409360.1409380>. 18, 19
- [75] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. Technical report, 2016. URL <http://hdl.handle.net/11245/2.128089><http://hdl.handle.net/11245/1.380005>. 19
- [76] Peter A Boncz, Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution Software Testpilot View project Using GPUs in analytic DBMSes View project MonetDB/X100: Hyper-Pipelining Query Execution. Technical report, 2005. URL <https://www.researchgate.net/publication/45338800>.
- [77] Peter Boncz, Torsten Grust, Maurice Van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 479–490, 2006. ISBN 1595934340. doi: 10.1145/1142473.1142527. 18

- [78] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’neil, Pat O’neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-oriented DBMS. Technical report. 18, 22
- [79] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD-PODS’09 - Proceedings of the International Conference on Management of Data and 28th Symposium on Principles of Database Systems*, pages 1–2, New York, New York, USA, 2009. Association for Computing Machinery (ACM). ISBN 9781605585543. doi: 10.1145/1559845.1559846. URL <http://portal.acm.org/citation.cfm?doid=1559845.1559846>. 18
- [80] Martin Kaufmann. Storing and processing temporal data in a main memory column store. *Proceedings of the VLDB Endowment*, 6(12):1444–1449, aug 2013. ISSN 21508097. doi: 10.14778/2536274.2536333. URL <http://dl.acm.org/doi/10.14778/2536274.2536333>. 18
- [81] Christian Lemke, Kai Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding up queries in column stores: A case for compression. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6263 LNCS, pages 117–129. Springer, Berlin, Heidelberg, 2010. ISBN 3642151043. doi: 10.1007/978-3-642-15105-7_10. URL https://link.springer.com/chapter/10.1007/978-3-642-15105-7_{_}10. 18
- [82] SAP HANA: In-Memory Database. URL <https://www.sap.com/sea/products/hana.html>. 18
- [83] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases, VLDB ’01*, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1558608044.
- [84] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database - The end of a column store myth. In *Proceedings of the ACM*

- SIGMOD International Conference on Management of Data*, pages 731–741, 2012. ISBN 9781450312479. doi: 10.1145/2213836.2213946. [18](#)
- [85] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *Proceedings of the VLDB Endowment*, 6(14):1702–1713, sep 2013. ISSN 21508097. doi: 10.14778/2556549.2556555. URL <http://dl.acm.org/doi/10.14778/2556549.2556555>. [18](#)
- [86] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, aug 2009. ISSN 21508097. doi: 10.14778/1687627.1687671. URL <http://dl.acm.org/citation.cfm?doid=1687627.1687671>.
- [87] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, aug 2008. ISSN 21508097. doi: 10.14778/1454159.1454171. URL <http://dl.acm.org/citation.cfm?doid=1454159.1454171>.
- [88] Ziqiang Feng and Eric Lo. Accelerating aggregation using intra-cycle parallelism. In *Proceedings - International Conference on Data Engineering*, volume 2015-May, pages 291–302. IEEE Computer Society, may 2015. ISBN 9781479979639. doi: 10.1109/ICDE.2015.7113292.
- [89] Yinan Li and Jignesh M. Patel. BitWeaving: Fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 289–300, New York, New York, USA, 2013. ACM Press. ISBN 9781450320375. doi: 10.1145/2463676.2465322. URL <http://dl.acm.org/citation.cfm?doid=2463676.2465322>. [18](#)
- [90] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 743–754, New York, New York, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi:

- 10.1145/2588555.2610507. URL <http://dl.acm.org/citation.cfm?doid=2588555.2610507>. 18
- [91] PÅ Larson, EN Hanson, SL Price IEEE Data Eng. Bull., and undefined 2012. Columnar Storage in SQL Server 2012. *academia.eduPaperpile*. URL <http://www.academia.edu/download/48908164/DE2012Q1.pdf{#}page=17>.
- [92] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013. ISSN 21508097. doi: 10.14778/2536222.2536233. 19
- [93] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Shareddb: Killing one thousand queries with one stone. *Proceedings of the VLDB Endowment*, 5(6):526–537, feb 2012. ISSN 21508097. doi: 10.14778/2168651.2168654. URL <http://dl.acm.org/doi/10.14778/2168651.2168654>. 18
- [94] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. Technical report, . URL <https://ieeexplore.ieee.org/abstract/document/6816684/>. 19
- [95] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning Publisher Very Large Data Base Endowment Inc. (VLDB Endowment) Terms of Use Creative Commons Attribution-Noncommercial-Share Alike 3.0 Schism: a Workload-Driven Approach to Database Replication and Partitioning. Technical report, 2150. URL <http://dl.acm.org/citation.cfm?id=1920853http://hdl.handle.net/1721.1/73347http://creativecommons.org/licenses/by-nc-sa/3.0/>.
- [96] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and

- Ryan Stutsman. The case for RAMClouds. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, jan 2010. ISSN 0163-5980. doi: 10.1145/1713254.1713276. URL <https://dl.acm.org/doi/10.1145/1713254.1713276>.
- [97] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, Michael J Franklin, and Scott Shenker. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Technical report. URL <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>.
- [98] GA Agha the MIT Press and undefined 1986. A model of concurrent computation in distributed systems. *ci.nii.ac.jpPaperpile*. URL <https://ci.nii.ac.jp/naid/10000004223/>. 19
- [99] Marcin Zukowski, Mark Van De Wiel, and Peter Boncz. Vectorwise: A vectorized analytical DBMS. In *Proceedings - International Conference on Data Engineering*, pages 1349–1350, 2012. doi: 10.1109/ICDE.2012.148. 19
- [100] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A data platform based on the scaling-up approach. In *Proceedings of the VLDB Endowment*, volume 11, pages 663–676. VLDB Endowment, 2018. doi: 10.14778/3184470.3184471. 19
- [101] Per Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. SQL server column store indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1177–1184, 2011. ISBN 9781450306614. doi: 10.1145/1989323.1989448. 19
- [102] Thomas Willhalm, Ismail Oukid, Ingo M Uller, and Franz Faerber. Vectorizing Database Column Scans with Complex Predicates. Technical report. 19
- [103] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Rossy. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 2015-May, pages 1493–1508, New York, New York, USA, may 2015. Association for

- Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2747645. URL <http://dl.acm.org/citation.cfm?doid=2723372.2747645>. 19
- [104] Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop - The Databricks Blog. URL <https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html>. 19
- [105] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases. *Proceedings of the VLDB Endowment*, 11(1):1–13, sep 2017. ISSN 21508097. doi: 10.14778/3151113.3151114. URL <https://doi.org/10.14778/3136610.3136611>. 19
- [106] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, pages 311–326, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2882925. URL <http://doi.acm.org/10.1145/2882903.2882925>. 19
- [107] Skye Wanderman-Milne and Nong Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, 37(1):31–37, 2014.
- [108] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 613–624. IEEE, 2010.
- [109] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, apr 2014. ISSN 0949-877X. doi: 10.1007/s00778-013-0348-4. URL <https://doi.org/10.1007/s00778-013-0348-4>.
- [110] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. Tupleware:” Big” Data, Big Analytics, Small Clusters. In *CIDR*, 2015.
- [111] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast Queries over Heterogeneous Data Through Engine Customization. *Proc.*

- VLDB Endow.*, 9(12):972–983, aug 2016. ISSN 2150-8097. doi: 10.14778/2994509.2994516. URL <https://doi.org/10.14778/2994509.2994516>.
- [112] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, number CONF, 2015.
- [113] A Kohn, V Leis, and T Neumann. Making Compiling Query Engines Practical. *IEEE Transactions on Knowledge and Data Engineering*, page 1, 2019. doi: 10.1109/TKDE.2019.2905235. [19](#), [89](#)
- [114] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment*, 11(13):2209–2222, sep 2018. ISSN 21508097. doi: 10.14778/3275366.3284966. URL <http://dl.acm.org/citation.cfm?doid=3275366.3284966>. [19](#)
- [115] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building Efficient Query Engines in a High-level Language. *Proc. VLDB Endow.*, 7(10):853–864, jun 2014. ISSN 2150-8097. doi: 10.14778/2732951.2732959. URL <http://dx.doi.org/10.14778/2732951.2732959>. [19](#)
- [116] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD ’16*, pages 1907–1922, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2915244. URL <http://doi.acm.org/10.1145/2882903.2915244>. [19](#)
- [117] Ruby Y Tahboub, Grégory M Essertel, and Tiark Rompf. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD ’18*, pages 307–322, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196893. URL <http://doi.acm.org/10.1145/3183713.3196893>. [19](#)

- [118] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017. [19](#)
- [119] Peter Bakkum and Srimat Chakradhar. Efficient Data Management for GPU Databases. Technical report. [20](#), [31](#)
- [120] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144. IEEE, 2011. [20](#)
- [121] Dana Schaa and David Kaeli. Exploring the multiple-GPU design space. In *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, 2009. ISBN 9781424437504. doi: 10.1109/IPDPS.2009.5161068. [20](#)
- [122] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5): 879–899, 2008. ISSN 00189219. doi: 10.1109/JPROC.2008.917757. [20](#)
- [123] J Cohen and M Jeroen Molemaker. A fast double precision CFD code using CUDA. *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pages 414–429, 2009. [20](#)
- [124] Georg Dotzler, Ronald Veldema, and Michael Klemm. *IWMSE: JCuDaMP: OpenMP/Java on CUDA*. 2010. ISBN 9781605589640.
- [125] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the ACM/IEEE SC 2004 Conference: Bridging Communities*. Institute of Electrical and Electronics Engineers Inc., 2004. ISBN 0769521533. doi: 10.1109/SC.2004.26. [20](#)
- [126] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing 2008, HPDC’08*, pages 165–174, New York, New York, USA, 2008. ACM Press. ISBN

9781595939975. doi: 10.1145/1383422.1383443. URL <http://portal.acm.org/citation.cfm?doid=1383422.1383443>. 20
- [127] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 94–103, New York, New York, USA, 2010. ACM Press. ISBN 9781605589350. doi: 10.1145/1735688.1735706. URL <http://portal.acm.org/citation.cfm?doid=1735688.1735706>. 20
- [128] Peter Benjamin Volk, Dirk Habich, and Wolfgang Lehner. GPU-Based Speculative Query Processing for Database Operations. Technical report. URL <http://www.vldbarc.org/archives/workshop/2010/proceedings/files/vldb{ }2010{ }workshop/ADMS{ }2010/adms10-volk.pdf>. 20, 22
- [129] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, page 17. ACM, 2017. 21, 24, 33, 35, 40, 44
- [130] GPUDirect: NVIDIA Developer. URL <https://developer.nvidia.com/gpudirect>. 21
- [131] Jing Li, Hung Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriff DB: Balancing I/O and GPU bandwidth in big data analytics. In *Proceedings of the VLDB Endowment*, volume 9, pages 1647–1658. Association for Computing Machinery, oct 2015. doi: 10.14778/3007328.3007331. URL <http://dl.acm.org/doi/10.14778/3007328.3007331>. 21
- [132] Nvidia dgx-2. <https://www.nvidia.com/en-us/data-center/dgx-2/>, . 21, 125
- [133] Nvidia dgx station. <https://www.nvidia.com/en-us/data-center/dgx-station/>, . 21, 74, 82, 125
- [134] NVLink & NVSwitch:Advanced Multi-GPU Systems: NVIDIA, . URL <https://www.nvidia.com/en-sg/data-center/nvlink/>. 21, 31
- [135] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34(4):21:1—21:39, dec 2009. ISSN

- 0362-5915. doi: 10.1145/1620585.1620588. URL <http://doi.acm.org/10.1145/1620585.1620588>. 21, 22, 23, 26, 28, 31, 85
- [136] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. Technical report. URL <https://ieeexplore.ieee.org/abstract/document/1617427/>. 21
- [137] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009. ISSN 21508097. doi: 10.14778/1687627.1687671.
- [138] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006. ISBN 1595934340. doi: 10.1145/1142473.1142548. 21
- [139] Stavros Harizopoulos, Velen Liang, Samuel Madden, and Daniel J Abadi. Performance Tradeoffs in Read-Optimized Databases. Technical report, 2006. URL <http://nms.csail.mit.edu/~stavros/pubs/vldb2006.pdf>. 22
- [140] Eyal Rozenberg CWI, Amsterdam ERozenberg, cwini Peter Boncz CWI, and Amsterdam PBoncz. Faster across the PCIe bus: A GPU library for lightweight decompression including support for patched compression schemes. *dl.acm.orgPaperpile*, may 2017. doi: 10.1145/3076113.3076122. URL <http://dx.doi.org/10.1145/3076113.3076122>. 22
- [141] Jun Rao and Kenneth A Ross. Cache Conscious Indexing for Decision-Support in Main Memory. Technical report, 1998. URL <https://academiccommons.columbia.edu/doi/10.7916/D8T441ZB>. 22
- [142] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. k-ary search on modern processors. In *Proceedings of the 5th International Workshop on Data Management on New Hardware, DaMoN 2009 in Conjunction with ACM SIGMOD/PODS Conference*, pages 52–60, 2009. ISBN 9781605587011. doi: 10.1145/1565694.1565705. 22
- [143] M G Ivanova, M L Kersten, N J Nes, and R A P Gonçalves. An architecture for recycling intermediates in a column-store. *ACM Trans. Datab. Syst.* 35,

- 4, *Article*, 24(4), nov 2010. doi: 10.1145/1862919.1862921. URL <http://doi.acm.org/10.1145/1862919.1862921>. 22
- [144] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. Density-based clustering using graphics processors. In *International Conference on Information and Knowledge Management, Proceedings*, pages 661–670, 2009. ISBN 9781605585123. doi: 10.1145/1645953.1646038. 22
- [145] Luke J. Gosink, Kesheng Wu, E. Wes Bethel, John D. Owens, and Kenneth I. Joy. Data parallel bin-based indexing for answering queries on multi-core architectures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5566 LNCS, pages 110–129, 2009. ISBN 3642022782. doi: 10.1007/978-3-642-02279-1_9. 22
- [146] Witold Andrzejewski and Robert Wrembel. GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6262 LNCS, pages 315–329. Springer, Berlin, Heidelberg, 2010. ISBN 3642152503. doi: 10.1007/978-3-642-15251-1_26. URL https://link.springer.com/chapter/10.1007/978-3-642-15251-1_{_}26. 22
- [147] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 339–350, 2010. ISBN 9781450300322. doi: 10.1145/1807167.1807206. 22
- [148] T Kaldewey, J Hagen, A Di Blas, E Sedlar First USENIX Workshop On, and undefined 2009. Parallel search on video cards. *usenix.orgPaperpile*. URL https://www.usenix.org/event/hotpar09/tech/full_{_}papers/kaldewey/kaldewey_{_}.html. 22
- [149] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *Proceedings of the VLDB Endowment*, 6(12):1374–1377, 2013. ISSN 21508097. doi: 10.14778/2536274.2536319. 23, 28

- [150] Sebastian Breß, Henning Funke, and Jens Teubner. Robust Query Processing in Co-Processor-accelerated Databases. *dl.acm.orgPaperpile*, 26-June-20:1891–1906, jun 2016. doi: 10.1145/2882903.2882936. URL <http://dx.doi.org/10.1145/2882903.2882936>. 23
- [151] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD-PODS’09 - Proceedings of the International Conference on Management of Data and 28th Symposium on Principles of Database Systems*, pages 283–295, 2009. ISBN 9781605585543. doi: 10.1145/1559845.1559877.
- [152] Ingo Müller, Cornelius Ratsch, and Franz Faerber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. *researchgate.netPaperpile*, 2014. doi: 10.5441/002/edbt.2014.27. URL https://www.researchgate.net/profile/Ingo_Mueller/publication/259935043_Adaptive_String_Dictionary_Compression_in_In-Memory_Database_Systems/links/54cf8a610cf298d65663cc7a/Adaptive-String-Dictionary-Compression-in-In-Memory-Database-Systems.pdf. 23
- [153] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. ISSN 03600300. doi: 10.1145/375360.375365. 23
- [154] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *ACM SIGPLAN Notices*, volume 47, pages 129–139, aug 2012. doi: 10.1145/2370036.2145833. URL <https://dl.acm.org/doi/10.1145/2370036.2145833>.
- [155] Cheng Hung Lin, Sheng Yu Tsai, Chen Hsiung Liu, Shih Chieh Chang, and Jyuo Min Shyu. Accelerating string matching using multi-threaded algorithm on GPU. In *GLOBECOM - IEEE Global Telecommunications Conference*, 2010. ISBN 9781424456383. doi: 10.1109/GLOCOM.2010.5683320.
- [156] Niccolo Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. INFAnt: NFA pattern matching on GPGPU devices, oct 2010. ISSN 01464833. URL <https://dl.acm.org/doi/10.1145/1880153.1880157>. 23

- [157] Evangelia A. Sitaridi and Kenneth A. Ross. GPU-accelerated string matching for database applications. *VLDB Journal*, 25(5):719–740, oct 2016. ISSN 0949877X. doi: 10.1007/s00778-015-0409-y. URL <https://link.springer.com/article/10.1007/s00778-015-0409-y>. [23](#)
- [158] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an efficient hash table on the GPU. In *GPU Computing Gems Jade Edition*, pages 39–53. Elsevier, 2012. [23](#), [35](#), [58](#), [62](#)
- [159] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013. [23](#), [28](#), [31](#), [58](#), [62](#), [85](#)
- [160] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013. [23](#), [24](#)
- [161] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009. [23](#)
- [162] Spyros Blanas, Yinan Li, and Jignesh M Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 37–48. ACM, 2011.
- [163] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013. [44](#)
- [164] Bingsheng He and Qiong Luo. Cache-oblivious databases: Limitations and opportunities. *ACM Transactions on Database Systems (TODS)*, 33(2):8, 2008.

- [165] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17, 2007.
- [166] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1961–1976. ACM, 2016. [24](#), [44](#), [74](#)
- [167] Holger Pirk, Stefan Manegold, and Martin Kersten. Accelerating Foreign-Key Joins Using Asymmetric Memory Channels. In *ADMS*, 2011. [24](#), [58](#), [62](#)
- [168] Peter Kipfer, Mark Segal, and Rüdiger Westermann. UberFlow: A GPU-Based Particle Engine. Technical report, 2004. URL https://dl.acm.org/doi/abs/10.1145/1058129.1058146?casa={_}token=zTU0Y3f9cBkAAAAA:jkhAdBijIq268cwTirtoKs0WfT0tJpUKb1g9TnR-ynYNSRn9yjMTUU1u02WHJzeVJcQ1jwKKGb0
[24](#)
- [169] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses, SIGGRAPH 2005*. Association for Computing Machinery, Inc, jul 2005. doi: 10.1145/1198555.1198797.
- [170] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPURT-eraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 325–336, 2006. ISBN 1595934340. doi: 10.1145/1142473.1142511. URL <http://gamma.cs.unc.edu/GPUTERASORT>.
[24](#)
- [171] K. E. Batcher. Sorting networks and their applications. page 307. Association for Computing Machinery (ACM), 1968. doi: 10.1145/1468075.1468121. [24](#)
- [172] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with CUDA based on bitonic sort. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6067 LNCS, pages 403–410. Springer, Berlin, Heidelberg, 2010. ISBN 364214389X. doi: 10.

- 1007/978-3-642-14390-8.42. URL https://link.springer.com/chapter/10.1007/978-3-642-14390-8_{ }42. 25
- [173] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. GPU sample sort. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010. ISBN 9781424464432. doi: 10.1109/IPDPS.2010.5470444. 25
- [174] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan Primitives for GPU Computing Publication Date Scan Primitives for GPU Computing. 2007. doi: 10.2312/EGGH/EGGH07/097-106. URL <https://escholarship.org/uc/item/8051p6nd>. 25
- [175] D Cederman and P Tsigas. GPU-Quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14(4), 2009. ISSN 10846654. URL <http://doi.acm.org/10.1145/1498698.1564500>. 25
- [176] CUDA Toolkit 6.5: NVIDIA Developer. URL <https://developer.nvidia.com/cuda-toolkit-65>. 25
- [177] Emanuele Manca, Andrea Manconi, Alessandro Orro, Giuliano Armano, and Luciano Milanesi. CUDA-quicksort: An improved GPU-based implementation of quicksort. *Concurrency Computation*, 28(1):21–43, jan 2016. ISSN 15320634. doi: 10.1002/cpe.3611. 25
- [178] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 351–362, New York, New York, USA, 2010. ACM Press. ISBN 9781450300322. doi: 10.1145/1807167.1807207. URL <http://portal.acm.org/citation.cfm?doid=1807167.1807207>. 25
- [179] Dharendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of GPU Based Sorting Algorithms. *International Journal of Parallel Programming*, 46(6):1017–1034, dec 2018. ISSN 08857458. doi: 10.1007/s10766-017-0502-5. 25
- [180] Bonan Huang, Jinlan Gao, and Xiaoming L.X. An empirically optimized radix sort for GPU. In *Proceedings - 2009 IEEE International Symposium*

- on Parallel and Distributed Processing with Applications, ISPA 2009*, pages 234–241, 2009. ISBN 9780769537474. doi: 10.1109/ISPA.2009.89. 25
- [181] Thrust: NVIDIA Developer. URL <https://developer.nvidia.com/thrust>. 25
- [182] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *ACM SIGGRAPH 2005 Courses, SIGGRAPH 2005*, page 206, New York, New York, USA, jul 2005. Association for Computing Machinery, Inc. doi: 10.1145/1198555.1198787. URL <http://portal.acm.org/citation.cfm?doid=1198555.1198787>. 25
- [183] Guilan Wang and Guoliang Zhou. GPU-based aggregation of on-line analytical processing. In *Communications in Computer and Information Science*, volume 288 CCIS, pages 234–245. Springer, Berlin, Heidelberg, 2012. ISBN 9783642319648. doi: 10.1007/978-3-642-31965-5_28. URL https://link.springer.com/chapter/10.1007/978-3-642-31965-5_{_}28. 25
- [184] Haicheng Wu, Ferst Drive, Gregory Diamos, Sean Baxter, Michael Garland, Tim Sheard, Molham Aref, and Sudhakar Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. *Proceeding of the International Symposium on Code Generation and Optimization (CGO)*, pages 44:44–44:54, 2014. URL <http://doi.acm.org/10.1145/2544137.2544166>. 26, 28, 30, 33, 85
- [185] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *SOSP’11 - Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 233–248, 2011. ISBN 9781450309776. doi: 10.1145/2043556.2043579. 26
- [186] Teng Li, Esam El-Araby, Vikram K Narayana, and Tarek El-Ghazawi. Evolutionary Design Space Exploration View project Hybrid Photonic-Plasmonic Non-blocking Broadband 5x5 Router for Optical Network View project GPU Resource Sharing and Virtualization on High Performance Computing Systems. *ieeexplore.ieee.orgPaperpile*, 2011. doi: 10.1109/ICPP.2011.88. URL <https://www.researchgate.net/publication/221083375>.

- [187] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. *GViM: GPU-accelerated Virtual Machines*. 2009. ISBN 9781605584652. [26](#)
- [188] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, pages 217–228, 2011. ISBN 9781450305525. doi: 10.1145/1996130.1996160. URL www.hpcinthecloud.com/offthewire/Amazon-. [26](#)
- [189] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532, 2014. ISSN 10459219. doi: 10.1109/TPDS.2013.257. [26](#)
- [190] Johns Paul, Bingsheng He, and Chiew Tong Lau. Query Processing on OpenCL-based FPGAs: Challenges and Opportunities. Technical report. [27](#)
- [191] Hongwen Dai, Zhen Lin, Chao Li, Chen Zhao, Fei Wang, Nanning Zheng, and Huiyang Zhou. Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls. In *Proceedings - International Symposium on High-Performance Computer Architecture*, volume 2018-February, pages 208–220. IEEE Computer Society, mar 2018. ISBN 9781538636596. doi: 10.1109/HPCA.2018.00027. [27](#)
- [192] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011. [27](#)
- [193] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *Proceedings of the 2011 International Conference on High Performance Computing and Simulation, HPCS 2011*, pages 24–32, 2011. ISBN 9781612843810. doi: 10.1109/HPCSim.2011.5999803. [27](#)

- [194] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *Proceedings of the VLDB Endowment*, 4(5):314–325, 2011. ISSN 21508097. doi: 10.14778/1952376.1952381. [27](#)
- [195] Kaibo Wang, Kai Zhangz, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, jul 2014. ISSN 21508097. doi: 10.14778/2732967.2732976. URL <http://dl.acm.org/doi/10.14778/2732967.2732976>. [27](#)
- [196] Zhila Nouri and Yi Cheng Tu. GPU-based parallel indexing for concurrent spatial query processing. In *ACM International Conference Proceeding Series*, pages 1–12, New York, NY, USA, jul 2018. Association for Computing Machinery. ISBN 9781450365055. doi: 10.1145/3221269.3221296. URL <https://dl.acm.org/doi/10.1145/3221269.3221296>. [27](#)
- [197] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. Harmonia: A high throughput B+tree for GPUs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 133–144, New York, NY, USA, feb 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295704. URL <https://dl.acm.org/doi/10.1145/3293883.3295704>. [27](#)
- [198] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. GPUQP: Query Co-processing Using Graphics Processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07*, pages 1061–1063, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247606. URL <http://doi.acm.org/10.1145/1247480.1247606>. [28](#), [93](#)
- [199] Philip Garcia and Henry F. Korth. Pipelined hash-join on multithreaded architectures. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware, DaMoN '07*, page 1, New York, New York, USA, 2007. ACM Press. ISBN 9781595937728. doi: 10.1145/1363189.1363191. URL <http://portal.acm.org/citation.cfm?doid=1363189.1363191>. [29](#)
- [200] Jens Teubner and Rene Mueller. How soccer players would do stream joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 625–636, New York, New York, USA, 2011. ACM Press. ISBN

9781450306614. doi: 10.1145/1989323.1989389. URL <http://portal.acm.org/citation.cfm?doid=1989323.1989389>. 29, 125
- [201] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, jun 2011. ISSN 2150-8097. doi: 10.14778/2002938.2002940. URL <http://dx.doi.org/10.14778/2002938.2002940>. 30, 108
- [202] OmniSci. <https://www.omnisci.com/>. 2019. 30
- [203] Thomas Neumann and Viktor Leis. Compiling Database Queries into Machine Code. *IEEE Data Eng. Bull.*, 2014. 30
- [204] Sparsh Mittal and Jeffrey S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, 47(4), jul 2015. ISSN 15577341. doi: 10.1145/2788396. URL <http://dx.doi.org/10.1145/2788396>. 30
- [205] Sebastian Breß. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, nov 2014. ISSN 1618-2162. doi: 10.1007/s13222-014-0164-z. URL <https://link.springer.com/article/10.1007/s13222-014-0164-z>. 30
- [206] Stefan L Manegold Peter Boncz Martin Kersten. Generic Database Cost Models for Hierarchical Memory Systems. Technical report, 2002. URL <http://monetdb.cwi.nl>. 31
- [207] S Breß, F Beier, H Rauhe, KU Sattler, E Schallehn Information Systems, and undefined 2013. Efficient co-processor utilization in database query processing. *ElsevierPaperpile*. URL <https://www.sciencedirect.com/science/article/pii/S0306437913000732>. 31
- [208] Sebastian Breß, Ingolf Geist, Eike Schallehn, Maik Mory, and Gunter Saake. A framework for cost based optimization of hybrid CPU/GPU query plans in database systems * †. Technical Report 4, 2012. URL <https://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-f650e8a0-6c37-486f-bfb1-929a5a74c2b9>. 31
- [209] The AMD FX APU — AMD, . URL <https://www.amd.com/en/products/fx-processors-laptop>. 31

- [210] Kai Zhang, Jiayu Hu, Bingsheng He, and Bei Hua. DIDO: Dynamic Pipelines for In-Memory Key-Value Stores on Coupled CPU-GPU Architectures. 2017. doi: 10.1109/ICDE.2017.120. [31](#)
- [211] PCIe 4.0: Benchmark for Component Performance: AMD Partner Hub. URL <https://www.amd.com/en/partner/pcie-benchmark-for-component-performance>. [31](#), [126](#)
- [212] Chengxin Guo, Hong Chen, Feng Zhang, and Cuiping Li. Distributed join algorithms on multi-CPU clusters with GPUDirect RDMA. In *ACM International Conference Proceeding Series*. Association for Computing Machinery, aug 2019. ISBN 9781450362955. doi: 10.1145/3337821.3337862. URL <https://doi.org/10.1145/3337821.3337862>. [32](#), [58](#), [61](#), [75](#)
- [213] RAPIDS Accelerates Data Science End-to-End, . URL <https://developer.nvidia.com/blog/gpu-accelerated-analytics-rapids/>. [32](#)
- [214] Sebastian Breß, Eike Schallehn, and Ingolf Geist. *Towards Optimization of Hybrid CPU/GPU Query Plans in Database Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-32518-2. [33](#)
- [215] Hannes Rauhe, Jonathan Dees, Kai-Uwe Sattler, and Franz Faerber. Multi-level parallel query execution framework for cpu and gpu. In *East European Conference on Advances in Databases and Information Systems*, pages 330–343. Springer, 2013.
- [216] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.
- [217] Holger Pirk, Stefan Manegold, and Martin Kersten. Waste not... Efficient co-processing of relational data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 508–519. IEEE, 2014. [33](#)
- [218] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. Relational joins on GPUs: A closer look. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2663–2673, 2017. [33](#), [44](#), [74](#)
- [219] Gregory Frederick Diamos, Haicheng Wu, Ashwin Lele, and Jin Wang. Efficient relational algebra algorithms and data structures for gpu. Technical report, Georgia Institute of Technology, 2012. [33](#)

- [220] M Seliuchenko, O Lavriv, O Panchenko ... Radio Electronics & ..., and undefined 2016. Enhanced multi-commodity flow model for QoS-aware routing in SDN. *ieeexplore.ieee.orgPaperpile*. URL <https://ieeexplore.ieee.org/abstract/document/7739623/>. 60, 69
- [221] N Farrugia, JA Briffa, V Buttigieg 2019 IEEE Congress On, and undefined 2019. Solving the Multi-Commodity Flow Problem using a Multi-Objective Genetic Algorithm. *ieeexplore.ieee.orgPaperpile*. URL <https://ieeexplore.ieee.org/abstract/document/8790160/>.
- [222] Moses Charikar, Yonatan Naamad, Jenifer Rexford, and X. Kelvin Zou. Multi-commodity Flow with In-Network Processing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11409 LNCS, pages 73–101. Springer Verlag, 2019. ISBN 9783030197582. doi: 10.1007/978-3-030-19759-9_6. 60, 69
- [223] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. Track Join: Distributed Joins with Minimal Network Traffic. *dl.acm.orgPaperpile*, pages 1483–1494, 2014. doi: 10.1145/2588555.2610521. URL <http://dx.doi.org/10.1145/2588555.2610521>. 64
- [224] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. Technical report, . URL <https://ieeexplore.ieee.org/abstract/document/6816684/>.
- [225] Lukas Rupperecht, William Culhane, and Peter Pietzuch. Squirrel-join: Network-aware distributed join processing with lazy partitioning. In *Proceedings of the VLDB Endowment*, volume 10, pages 1250–1261. Association for Computing Machinery, aug 2017. doi: 10.14778/3137628.3137636. 64
- [226] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. *dl.acm.orgPaperpile*, 12-16-March-2016, feb 2016. doi: 10.1145/2851141.2851145. URL <http://dx.doi.org/10.1145/2851141.2851145>. 66
- [227] TPC-H. <http://www.tpc.org/tpch/>, 1999. 75, 86, 108, 110
- [228] Amazon ec2. <https://aws.amazon.com/ec2/instance-types/p3/>. 83

- [229] Max Heimerl, Martin Kiefer, and Volker Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1477–1492, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2749438. URL <http://doi.acm.org/10.1145/2723372.2749438>. 85
- [230] Star Schema Benchmark. <https://www.cs.umb.edu/~poneil/StarSchemaB.PDF>, 2009. 86, 108
- [231] Evangelia A Sitaridi and Kenneth A Ross. Optimizing Select Conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321969. doi: 10.1145/2485278.2485282. URL <https://doi.org/10.1145/2485278.2485282>. 96
- [232] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *Proc. VLDB Endow.*, 11(1):1–13, sep 2017. ISSN 2150-8097. doi: 10.14778/3151113.3151114. URL <https://doi.org/10.14778/3151113.3151114>. 96
- [233] A Crotty, A Galakatos, and T Kraska. Getting Swole: Generating Access-Aware Code with Predicate Pullups. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1273–1284, 2020. 96
- [234] David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016. ISSN 1572-5286. doi: <https://doi.org/10.1016/j.disopt.2016.01.005>. URL <http://www.sciencedirect.com/science/article/pii/S1572528616000062>. 105
- [235] NVLink and NVSwitch: Advanced Multi-GPU Systems: NVIDIA, . URL <https://www.nvidia.com/en-sg/data-center/nvlink/>. 126
- [236] EGX A100 : Real Time AI Processing at the Edge: NVIDIA. URL <https://www.nvidia.com/en-sg/data-center/products/egx-a100/>. 127

- [237] Jiasi Chen and Xukan Ran. Deep Learning With Edge Computing: A Review. *Proceedings of the IEEE*, 2019. ISSN 15582256. doi: 10.1109/JPROC.2019.2921977. [127](#)