Interactive Data Analytics Using GPUs

by

Anil Shanbhag

B.Tech., Indian Institute of Technology Bombay (2014) S.M., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Certified by..... Samuel R. Madden Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Interactive Data Analytics Using GPUs

by

Anil Shanbhag

Submitted to the Department of Electrical Engineering and Computer Science on August 26, 2020, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Modern GPUs provide an order-of-magnitude greater memory bandwidth compared to CPUs. In theory, this means data processing systems can process O(TB) of data with sub 100ms latency, thereby enabling interactive query response times on analytical SQL queries. However, the massively parallel architecture of GPUs requires rearchitecting in-memory data analytics systems in order to achieve optimal performance. This thesis describes how we adapted and redesigned in-memory data analytics systems to better exploit the GPU's memory and execution model.

We present Crystal, a library of building blocks that can be used for writing high performance SQL query implementations for GPU. We use Crystal to implement basic SQL query operators and an analytical benchmark. We present theoretical models based on memory bandwidth as the critical bottleneck for query performance and show that implementations using Crystal are able to achieve these theoretical limits. We also present a study of the fundamental performance characteristics of GPUs and CPUs for database analytics. Our analysis shows that using modern GPUs vs CPUs can lead to a runtime gain equal to $1.5 \times$ bandwidth ratio of GPU to CPU ($25 \times$ in our setup) and be $4 \times$ more cost effective than CPUs. Finally, we used Crystal's design principles to develop massively parallel variants of two classic sequential algorithms: top-k and bit-packing based compression. Bitonic Top-K is a top-k algorithm based on bitonic sort that is $4 \times$ faster than previous approaches. GPU-FOR is a compression format that can be decompressed efficiently in parallel and can be used to fit more data into the limited GPU memory.

In summary, this thesis makes the case for using GPUs as the primary execution engine for interactive data analytics, and shows that implementations are efficient and practical.

Thesis Supervisor: Samuel R. Madden Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I thank my advisor Sam Madden for his guidance and support throughout my graduate studies. Sam always gave great insights and I have benefited greatly from his breadth of experience and deep intellect. Sam has always been supportive, given a great deal of freedom and I consider myself fortunate to have had him as my advisor.

I am also thankful to the rest of my committee members. I thank Xiangyao Yu for being a great collaborator and a good friend. Xiangyao offered unceasing help throughout the projects we collaborated on. I would like to thank Tim Kraska for his thought provoking discussions during the group meetings and additionally for serving on my committee.

I want to give special thanks to my collaborators over the years. I want to thank Alekh Jindal for being a hands-on mentor and guiding me through my first two years. I loved working with Holger Pirk who got me interested in in-memory databases. Holger's innovative ideas and unique insights made working with him a great learning experience. I enjoyed collaborating with Nesime Tatbul. Nesime showed me a different view of research from the industry side.

I owe gratitude to the amazing research group that I have been fortunate to be a part of: Favyen, Kapil, Oscar, Yi, Matt, Siva, Becca, Anant, Leilani, Joana, Manasi, Raul, Emanuel, Dong, Lei, Siva, and Wenbo. Together, we went through numerous submissions, internal feedback on papers and talks, and many fun group outings. Outside of work, I am blessed to have a fantastic set of friends. Special thanks to Nishant, Anurag, and Tuhin who have shared this journey with me. I also want to thank Srivatsa, Shelar, Prateesh, Shibani, Frisbee gang and Sangam group for making the past few years enjoyable.

As I complete my Ph.D. journey, I cannot help but look back and thank people who enabled me to get to MIT in the first place. I am grateful to Srikanth Kandula who backed me when I was a research novice and inspired me to do a Ph.D. I also owe deep gratitude to S. Sudarshan who mentored my undergraduate thesis and helped me write my first paper.

Finally, and more importantly, I can't find words to describe my gratitude to my family. My parents, Atmanand and Deepa, and my brother Anish for their constant support and encouragement. I am also thankful to my girlfriend, Swetha Itchapurapu, for her support and love during all my ups and downs. This thesis would not have been possible without them.

Contents

1	Intr	roduction	17
	1.1	GPU Architecture and Challenges	19
	1.2	Thesis Contributions	21
		1.2.1 Tile-based Execution Model for Query Processing	
		on GPUs	21
		1.2.2 Fundamental Performance Characteristics of GPUs and CPUs	
		for Database Analytics	22
		1.2.3 Novel Query Operators for GPUs	23
	1.3	Thesis Outline	26
2	Bac	ekground	27
	2.1	GPU Architecture	27
	2.2	Query Execution on GPU	30
Ι	Qı	uery Execution on GPU	32
3	GP	U Query Execution	33
	3.1	Introduction	33
	3.2	Background	36
		3.2.1 Query Execution on CPU	36
	3.3	Failure of the Coprocessor Model	37
	3.4	Tile-based Execution Model	38
	3.5	Crystal Library	43

	3.6	Opera	tors on GPU vs CPU	48
		3.6.1	Project	48
		3.6.2	Select	50
		3.6.3	Hash Join	52
		3.6.4	Sort	57
	3.7	Workl	oad Evaluation	61
		3.7.1	Workload	62
		3.7.2	Performance Comparison	62
		3.7.3	Case Study	64
		3.7.4	Cost Comparison	69
	3.8	Concl	usion	69
п	N	ovel	GPU Query Operators	71
4	Top	-K		73
	4.1	Introd	luction	73
	4.2	Backg	round	76
		4.2.1	Sorting on the GPU	76
		4.2.2	K-Selection	79
	4.3	Algori	thms	79
		4.3.1	Per-Thread Top-K	80
		4.3.2	Bitonic Top-K	81
	4.4	Optim	nization & Implementation	86
		4.4.1	Per-Thread Top-K	86
		4.4.2	Selection-based Top-K	87
		4.4.3	Optimizing Bitonic Top-K	87
		4.4.4	Database Integration	97
	4.5	Evalua	ation	99
		4.5.1	Setup	99
		152	Performance with Varving K	90

		4.5.3	Dependence on Data Type	100
		4.5.4	Dependence on Data Distribution	102
		4.5.5	Dependence on Data Size	104
		4.5.6	$Key(s) + Value \ldots \ldots$	105
		4.5.7	Comparison against CPU	106
		4.5.8	MapD Integration	107
	4.6	Cost I	Model	109
		4.6.1	Radix-based Top-K	109
		4.6.2	Bitonic Top-K	110
	4.7	Concl	usion	112
5	Dat	a Con	pression	113
	5.1	Introd	luction	113
	5.2	Backg	round	115
		5.2.1	Compression Techniques	116
		5.2.2	Query Execution on GPUs	118
	5.3	Fast E	Bit Unpacking	120
		5.3.1	Data Format	121
		5.3.2	Implementation	123
		5.3.3	Discussion	127
	5.4	Fast I	Delta Decoding	129
		5.4.1	Data Format	130
		5.4.2	Implementation	131
	5.5	Datab	base Integration	133
	5.6	Evalua	ation	136
		5.6.1	Setup	138
		5.6.2	Performance with Varying Bitwidths	138
		5.6.3	Dependence on Data distributions	140
		5.6.4	Performance on SSB	142
		5.6.5	GPU as a Coprocessor	144

\mathbf{A}	Per	-Thread Top-K Using Registers	153
	6.2	Heterogeneous Computing	150
	6.1	Multi-GPU Query Execution	148
6	Cor	clusion and Future Work	147
	5.7	Conclusion	145
		5.6.6 Discussion \ldots	145

List of Figures

1-1	Memory bandwidth growth on CPU and GPU over the past decade .	18
2-1	GPU Architecture overview	28
2-2	GPU Memory Hierarchy	29
3-1	Star Schema Benchmark Q1.1	37
3-2	Evaluation of the GPU coprocessor approach on the Star Schema	
	Benchmark	37
3-3	Running selection on GPU	41
3-4	Vector-based to Tile-based execution models	41
3-5	Query Q0 Kernel running y > 5 with tile size 16 and thread block size 4	42
3-6	Implementing queries using Crystal	43
3-7	Query Q0 Kernel Implemented with Crystal	45
3-8	Q0 performance with varying tile sizes	47
3-9	GPU vs CPU performance on the project microbenchmark $\ . \ . \ .$.	50
3-10	Implementing selection scan	51
3-11	GPU vs CPU performance on the select microbenchmark \ldots . \ldots	52
3-12	GPU vs CPU performance on the join microbenchmark	56
3-13	GPU vs CPU performance on the sort microbenchmark	60
3-14	Implementing selection scan	61
3-15	Star Schema Benchmark Queries	63
3-16	Star Schema Benchmark Q1.1 Execution Plan	65
3-17	Star Schema Benchmark Q2.1	66
3-18	Star Schema Benchmark Q2.1 Execution Plan	66

4-1	The Duality of Top-K and Sorting	74
4-2	Bitonic Sorting Network	77
4-3	Top-K Merge	83
4-4	Bitonic Top-K (K=4)	85
4-5	Combining Multiple Steps	90
4-6	Avoiding shared memory bank conflicts with padding $\ldots \ldots \ldots$	92
4-7	Bitonic Top-K performance varying the number of elements per thread	93
4-8	Comparison distance for local sort $k = 8, x = 4 \dots \dots \dots \dots$	94
4-9	Shared memory bank conflicts when comparing elements	94
4-10	Time taken with different k (32-bit float values) $\ldots \ldots \ldots \ldots$	100
4-11	Time taken with different k (32-bit integer values)	101
4-12	Time taken with different k (64-bit double values)	102
4-13	Top-K performance across different distribution	103
4-14	Performance with varying data size	104
4-15	Performance with different number of keys	105
4-16	Comparing GPU Top-K against CPU Top-K	106
4-17	Using Top-K kernel in MapD	107
4-18	Estimated vs actual runtimes for different K \hdots	111
5-1	Bit packing with vertical data layout	118
5-2	GPU-FOR Data Format	122
5-3	Example encoding with GPU-FOR	122
5-4	Decompression performance with varying number of data blocks per	
	thread block (D) \hdots	126
5-5	GPU-DFOR Data Format	130
5-6	Illustration of Prefix Sum Algorithm	132
5-7	Query Q0 Kernel Implemented with Crystal	134
5-8	Performance of the different compression algorithms on uniform data	
	with varying bit widths	139
5-9	Comparison of compression schemes on different data distributions	141

5-10	Compression waterfall for Star Schema Benchmark columns	143
5-11	Performance on Star Schema Benchmark queries with compressed	
	columns	143
6-1	NVIDIA A100 with NVLink GPU-to-GPU connections	149
A-1	Different Per-Thread Top-K Approaches	154

List of Tables

3.1	List of block-wide functions in Crystal	44
3.2	Hardware Specifications	63
3.3	Purchase and renting cost of CPU and GPU instance \ldots	69

Chapter 1

Introduction

Hardware trends have greatly influenced the development of data management systems. Historically, data was stored on (rotating) disks, and only small fractions would be kept in main memory. The increase in DRAM capacities along with the increase in the number of DIMM slots per machine has increased capacity and allowed systems to keep large fractions, and in some cases all of their data directly in RAM. Main memory is more than an order of magnitude faster than disk and allows random access. Thus, in comparison to disk-based systems, in-memory database systems offer significant performance improvements. A number of in-memory data analytics systems have been developed including Tableau Hyper [56], SAP HANA [29], and MonetDB [22].

The shift to in-memory systems has shifted the performance bottleneck from disk bandwidth to main memory bandwidth. Over the past decade, main memory bandwidth growth has trailed CPU performance growth. While CPU performance has grown at an average 60 percent per year, main memory bandwidth has grown at just 7 percent per year. In order to get the next order of magnitude of performance improvement and achieve interactive query performance, researchers have started looking beyond multi-core CPUs to many-core accelerators such as GPUs and Intel Xeon Phi.

Over the past decade, special-purpose graphics processing units have evolved into general purpose computing devices, with the advent of general purpose parallel pro-



Figure 1-1: Memory bandwidth growth on CPU and GPU over the past decade

gramming models, such as CUDA and OpenCL. Because of GPUs' high compute power, they have seen significant adoption in deep learning and high performance computing. However, GPUs also have significant potential to accelerate memorybound applications such as data analytics systems. Figure 1-1 shows the growth of GPU memory bandwidth over the years in comparison to memory bandwidth on CPU. GPUs can utilize High-Bandwidth Memory (HBM), a new class of RAM that has significantly higher throughput compared to traditional DDR RAM used with CPUs. A single modern GPU is capable of delivering up to 1.2 TBps of memory bandwidth and 16 Tflops of compute compared to 128GBps of memory bandwidth and < 1 Tflops on a single CPU.

The rise in memory bandwidth has been coupled with a rise in GPU memory capacity. Over the past decade, GPU memory capacity has increased from 4GB to 40GB on the latest Nvidia A100 GPU. The current HBM roadmap is expected to further double the capacity. The ability to equip a modern server with several GPUs (up to 32), means that it's possible to have hundreds of gigabytes of GPU memory on a modern server. This is sufficient for many analytical tasks; for example, one machine could host several weeks of a large online retailer's (with say 100M sales per day) sales data (with 100 bytes of data per sale) in GPU memory, the on-time flight performance of all commercial airline flights in the last few decades, or several billion tweets sent over the past few days. In-memory data analytics systems are typically bound by DRAM memory bandwidth [56] and hence can benefit from higher memory bandwidth of GPUs.

In this thesis, we focus on analytical query processing powered by GPUs. Section 1.1 describes the GPU architecture and presents the challenges posed by the massively parallel nature of GPUs for query processing. Section 1.2 presents the problem context and contributions of this thesis. Section 1.3 describes the outline of this thesis.

1.1 GPU Architecture and Challenges

When GPUs become popular in the 1990s, they were originally in the 1990s designed to offload 2D and 3D graphics rendering from the main CPU processor. The limitation of GPUs at the time was the limited programmability. The launch of the Nvidia CUDA computing platform in 2007 allowed GPU programming in a C-like language. The Fermi architecture, released in 2009, was seminal for General Purpose GPU (GPGPU) computing. Fermi was the first complete GPU architecture satisfying the requirements of demanding High-Performance Computing (HPC) applications. In addition to improved performance, Fermi had a true cache hierarchy, error-correcting code memory (ECC), and concurrent execution. Over the past decade, in addition to increase in raw compute and memory bandwidth of GPGPUs, there has been significant improvement in the compute capabilities of GPUs to make them useful in a broad range of domains. GPUs now support synchronization via locks and atomics, unified memory access to CPU memory, ability to do RDMA, etc. Despite these improvements, the GPUs pose challenges to programmers trying to leverage their high performance for query processing. Some key challenges in adapting query processing to the GPU are:

Thread Divergence: GPUs implement a Single Instruction Multiple Threads (SIMT)

architecture. A group of 32 threads called a *warp* start executing at the same program address but have their private register state and program counters so that each thread is free to branch independently. However, when threads in the same warp follow a different execution path, threads are serialized by the hardware. This phenomenon is called thread divergence. While the branch followed by a subset of the threads in the warp is executed, the remaining threads are idle, resulting in resource underutilization and performance degradation.

Massive Parallelism and Limited Synchronization: GPUs achieve a very high degree of parallelism by having many processing elements, each of which can have many warps in flight at any point in time. When running at full capacity, there may be 163840 threads in flight on a V100 GPU. Only small groups of threads (called thread blocks) can synchronize, there is no synchronization available across all threads.

Tiny Threads: GPU threads have significantly fewer resources per thread to store intermediates. On the Nvidia V100, each GPU thread can only store roughly 24 4-byte entries in its local cache compared to 32KB L1 cache available per core on a CPU.

Memory Access Pattern: GPUs have a different memory hierarchy compared to traditional CPUs. Global memory is the largest type of memory, but it has high latency: 400–600 cycles. It is important to organize memory accesses to global memory so that threads access contiguous memory addresses. In that case, multiple memory accesses of the threads within a warp need be combined in few accesses from the global memory. This memory access pattern is called memory coalescing, and it achieves spatial data locality. Shared memory is programmable cache available per SM. To maximize performance, shared memory is organized into 32 banks, so that all threads in a warp can access different memory banks in parallel. However, if two threads in a warp access different items in the same memory bank, a bank conflict occurs, and accesses to this bank are serialized, potentially hurting performance.

To exploit the full performance of a GPU in database analytics workloads, we need to rearchitect SQL query operators to be aware of and work around these challenges.

1.2 Thesis Contributions

This thesis addresses the question of how to run analytical queries efficiently on the GPU. The following sections discuss our specific research contributions.

1.2.1 Tile-based Execution Model for Query Processing on GPUs

Compared to CPU threads, individual GPU threads have limited resources. We show that treating GPU threads as independent execution units can lead to additional materialization and bad performance. To address this issue, we propose a tile-based execution model. Tile-based processing extends vector-based processing on the CPU where each thread processes a vector at a time to the GPU. Threads on the GPU are grouped into thread blocks. Threads within a thread block can communicate through shared memory and can synchronize through barriers. Hence, even though a single thread on the GPU has limited capacity, a single thread block can hold a significantly larger group of elements collectively between them in shared memory. We call this unit a Tile. In the Tile-based execution model, instead of viewing each thread as an independent execution unit, we view a thread block as the basic execution unit with each thread block processing a tile of entries at a time. One key advantage of this approach is that after a tile is loaded into shared memory, subsequent passes over the tile will be read directly from shared memory and not from global memory.

We show that queries can be broken into steps where each is a function that takes as input a set of tiles, and outputs a set of tiles. We call these primitives *block-wide functions*. A block-wide function is a device function¹ that takes in a set of tiles as input, performs a specific task, and outputs a set of tiles. Instead of reimplementing these block-wide functions for each query, which would involve repetition of non-trivial functions, we developed a library called **Crystal**.

Crystal is a library of templated CUDA device functions that implement the full set of primitives necessary for executing typical analytic SQL SPJA analytical queries.

 $^{^1\}mathrm{Device}$ functions are functions that can be called from kernels on the GPU

Crystal lets users write high performance kernel code. Block-wide functions in Crystal make it easy to use non-trivial functions and reduce boilerplate code. The primitives are composable and make it is fairly easy to implement query kernels of larger queries. We use Crystal to implement the standard SQL query operators (select, project, join, sort) and the Star-Schema Benchmark (SSB) on the GPU. We present cost models for the operators and queries that assume memory bandwidth is saturated and show that these query kernels do saturate memory bandwidth.

1.2.2 Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics

There is a lack of consensus on how much performance improvement can be obtained from using GPUs for data processing. Past works have divergent views on gains that can be achieved - ranging from $2 \times$ to $100 \times$. Past work frequently compares against inefficient baselines, e.g., MonetDB [83, 78, 46] which is known to be inefficient [56]. The empirical nature of past work makes it hard to generalize results across hardware platforms.

To understand the true nature of performance gains on GPU vs CPU, we implement fundamental SQL query operators like Select, Project, Join, and Sort and, a popular analytics workload called the Star Schema Benchmark (SSB) using Crystal and compare against efficient implementations of operators and of SSB on CPU. Given that in-memory analytics is typically memory bandwidth bound, we use evaluate using these optimized implementations of query operators and full SQL queries to show why performance gains of running on GPU vs CPU deviate from the memory bandwidth ratio of GPU to CPU. We also develop theoretical models of the runtime of these operators assuming memory bandwidth is saturated and show that implementations using Crystal are able to saturate memory bandwidth. We use models to explain why the runtime ratio of operators and queries on GPU to CPU differs from the memory bandwidth ratio.

For individual query operators, we observe that the performance gain is equal to or

less than the bandwidth ratio of the GPU to CPU. However, on full SQL queries the performance gain is actually around twice the bandwidth ratio due to the inability of the CPU to hide the memory latency associated with irregular memory access patterns seen in multi-join queries. On a the experimental setup we used, running queries on the GPU leads to 24x reduction in query runtime on the SSB workload compared to running the queries on the CPU.

1.2.3 Novel Query Operators for GPUs

In the previous section, we described the Crystal framework and how it can be used to implement the basic SQL query operators: Select, Project, Join and Sort. The massively parallel nature of GPUs makes it non-trivial to implement certain SQL operators. We present two novel query operators: a parallel top-k operator called Bitonic Top-K and parallel decompression operators for two bit-packed storage formats GPU-FOR and GPU-DFOR.

Top-K A common type of analytical SQL query involves running a top-k, i.e., finding the highest (or lowest) k of n tuples given a ranking function. Examples of top-k queries include asking for the most expensive products on an e-commerce site, the best-rated restaurants in a review site, or the worst performing queries in a query log. Top-k is a well studied problem in computer science in general and data management in particular since top-k calculation (order-by/limit clauses) is supported by virtually every data analytics system.

Top-K is a classic sequential problem that can be efficiently computed by maintaining a priority-queue (a.k.a. max-heap) of size k and making a single pass over the data with each entry inserted into this heap. The runtime of this approach is on the order of $n \log(k)$. This algorithm can be parallelized across m processors by logically partitioning the data, having each processor compute a per-partition top-k and computing the global top-k from the m per-partition heaps. While this method can be efficiently implemented on multi-core processors, it is not suited to the Single-Instruction-Multiple-Threads execution model of massively parallel systems, because the unpredictable execution flow leads to high branch divergence overhead. With the recent interest in GPU-based query processing, there is an obvious need for a efficient, massively parallel algorithm to solve the top-k problem.

One way to develop an intuition for the existence and even the characteristics of a solution to this problem is to consider the duality of top-k and sorting algorithms. When thinking about sorting in the context of massively parallel architectures, a popular algorithm is bitonic sort. Yet, there is no known corresponding top-k algorithm to bitonic-sort. We can, however, hypothesize that, like bitonic sort, it is likely to be based on bitonic merges and needs to incorporate a number of low-level optimizations to make it compute- as well as bandwidth-efficient.

We systematically develop this intuition into a working algorithm by extensively studying existing top-k solutions on GPUs and developing a novel solution targeted towards massively parallel architectures. We found that it is in fact based on bitonic merges and called it *Bitonic Top-K*. We investigate the characteristics of a number of other potential top-k algorithms for GPUs, including sorting and heap-based algorithms, as well as radix-based algorithms that use the high-order bits to find the top items. In the end, we find that bitonic top-k is up to 4 times faster than other top-k approaches and up to 15x faster than sorting for k up to 256.

Compression GPU-based databases aim to provide real-time analytics capabilities by using GPUs to store a large fraction (or all) of the working set. A key constraint in these systems is the GPU memory capacity. Currently, GPUs have at most 40 GB of memory which is used both to cache the working set and as scratch memory for query execution. GPU memory is $6 \times$ more expensive compared to CPU RAM and going outside a single GPU's memory incurs a penalty. Therefore, data compression is critical. Currently, GPU-based systems use simple compression schemes like fixedwidth dictionary encoding and run-length encoding (RLE) similar to CPU-based inmemory analytics systems, decompressing on-the-fly during query execution.

In this thesis, we introduce two new efficient compression schemes for GPUs: GPU-FOR which does bit-packing in conjunction with Frame-Of-Reference (FOR) and *GPU-DFOR* which uses delta encoding with bit-packing and FOR. Both these schemes are designed to offer improved compression ratios while still being able to decode them in parallel across thousands of threads at close to memory bandwidth speeds. GPU-FOR partitions the data into blocks, in each block encoding integers with the minimum bit size needed to represent a value in the block. It works well with uniform data and can handle skew. GPU-DFOR first delta-encodes a block of integers before using GPU-FOR. It is suited for sorted and semi-sorted columns.

Past works have looked at delta encoding, FOR, and variable length byte-aligned packing (NSV). These works found that achieving the minimum space cost by using a combination of compression schemes (e.g. delta+NSV) can degrade performance as intensive decompression overburdens the GPU. Hence previous work deemed these schemes GPU-unfriendly. The reason for bad performance was that these works treated threads on the GPU as independent execution units and hence required multiple passes to decode the compressed data. For example: to use a column encoded using delta encoded variable length byte-aligned packing in a query, these systems would first run a prefix-sum primitive to unpack the variable-length byte-packed data and write it to global memory, then do a second pass prefix-sum primitive to delta decode the data, again writing to global memory, and finally run a query kernel that reads from global memory the unpacked column. This approach has high overhead as the intermediate data is read/written to global memory multiple times. In our work, we treat thread blocks as the basic execution unit; each thread block collectively decodes one block of encoded entries. By treating thread blocks as the basic unit of execution, we are able to cache a block of data in on-chip caches and inline the multiple steps involved in decoding into a single kernel, resulting in a single pass over the data. We present a series of optimizations that enable us to decode at close to memory bandwidth speed. The performance of our schemes simplifies the choice of compression scheme to encode a column: simply choose the scheme with the smallest storage footprint. It eliminates the need for sophisticated compression planners used by past works to choose the right compression scheme.

To show that our compression schemes perform well and significantly reduce the

storage footprint of GPU-based systems, we present an integration of GPU-FOR and GPU-DFOR into the Crystal framework. We encapsulate the decompression into a device function that enables programmers to change a kernel operating on an uncompressed array to a compressed column with a single line of code. In the end, we find that our compression schemes can reduce the storage footprint by up to $10\times$ on certain data distributions; on the Star Schema Benchmark, the proposed scheme achieves 50% reduction in storage compared to no compression and 37% compared to existing GPU compression schemes with no impact on performance.

1.3 Thesis Outline

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

Part I of the thesis focuses on efficient query execution on the GPU. We describe the *Tile-based execution model* and present the Crystal library. We present efficient implementations of SQL query operators and an analytical workload. We use theoretical models to show these saturate bandwidth on the GPU and use them to compare performance of a GPU-based vs CPU-based analytical database.

Part II presents two novel query operators. Chapter 4 presents a holistic comparison of different top-k algorithms for GPUs and presents the *Bitonic Top-K* algorithm. Chapter 5 presents two bit-packing based compression schemes GPU-FOR and GPU-DFOR that can be used to store data columns on the GPU and presents optimized decompression routines for each.

In the end, we summarize the main conclusions of our research and outline future directions for GPU-powered data analytics and database processing.

Chapter 2

Background

In this chapter, we present background and prior work related to this thesis. In Section 2.1 we review the basics of the GPU architecture. Section 2.2 describes the relevant aspects of past approaches to running database analytics workloads using GPUs.

2.1 GPU Architecture

Figure 2-1 shows an abstraction of the GPU architecture and threading model. Our research is done on NVIDIA GPUs so we use the NVIDIA CUDA terminology but the abstractions of OpenCL APIs are similar so our techniques are applicable for both types of GPU paradigms.

A GPU has multiple streaming multi-processors (SM). The SM is the GPU component executing the GPU functions. GPUs implement a Single Instruction Multiple Threads (SIMT) architecture. The unit of execution in SIMT is a group of threads called a *warp*, which is typically 32 threads. On the Nvidia V100 GPU, there are 80 SMs, each capable of running 64 concurrent warps (2048 threads).

To run a function on the GPU, you start by writing a GPU function called a *kernel*. The kernel is invoked across a large number of threads. These threads are grouped into units called *thread blocks* where each thread block is of size 32-1024 threads.



Figure 2-1: GPU Architecture overview

Each thread block gets scheduled on a SM. Once scheduled, the thread block is divided into warps with each warp executing independently. Threads in a warp start executing the same program at the same program addresss but have their own private register state and program counters so that each thread is free to branch independently. However, when threads in the same warp follow different execution paths, their execution is serialized by the hardware. This phenomenon is known as *thread divergence*. While the branch followed by a subset of threads in the warp are executed, the remaining threads are idle leading to performance degradation and resource underutilization. Threads within a single thread block can synchronize using barriers.

Many database operations executed on the GPU are performance bound by the memory subsystem (either shared or global memory) [83]. In order to characterize the performance of different algorithms on the GPU, it is thus critical to properly understand its memory hierarchy. Figure 2-2 shows a simplified hierarchy of a modern



Figure 2-2: GPU Memory Hierarchy

GPU.

The lowest and largest memory in the hierarchy is the *global memory*. A modern GPU can have global memory capacity of up to 40 GB with memory bandwidth of up to 1500 GBps. Global memory has high latency: 400-600 cycles. The scope of global memory is all GPU threads. The device groups global memory loads and stores from threads in a single warp such that multiple loads/stores to the same cache line are combined into a single request. Maximum bandwidth can be achieved when a warp's access to global memory results in neighboring locations being accessed.

Accesses to global memory from a SM are cached in the L2 cache (L2 cache is shared across all SMs). The L2 cache on the V100 GPU, which is shared by all processing elements has 49,152 128-byte cache lines.

Each SM has a *shared memory* which is used as a parallel, software controlled cache and its scope is a thread block. Shared memory serves as a scratchpad that is controlled by the programmer. Shared memory capacity is limited, on the V100 GPU, it is 16KB or 48KB depending on the kernel configuration per SM. Hence, at full occupancy with 64 concurrent warps executing, there is space for 24 4-byte entries per thread in shared memory. To maximize performance, shared memory is organized into 32 banks, so that all threads in a warp can access different memory banks in parallel. However, if two threads in a warp access different items in the same memory bank, a bank conflict occurs, and accesses to this bank are serialized,

potentially hurting performance. Shared memory has an order of magnitude higher bandwidth than global memory but has much smaller capacity (a few MB vs. multiple GB).

Each SM also has an L1 cache. Accesses to global memory can optionally also in the L1 cache (L1 cache is local to each SM). On the V100 GPU, L1 cache size is 48KB or 16KB depending on the kernel configuration per SM.

Finally, *registers* are the fastest layer of the memory hierarchy. Registers are privately owned by each thread and store values that are immediately used by each thread. On the V100 GPU, the register file size is 64k 32-bit registers per SM. If a thread block needs more registers than available, register values spill over to local memory. Despite its name, local memory only means it is only accessible by the thread – it is stored off the SM in slow global memory.

2.2 Query Execution on GPU

With the slowing of Moore's Law, CPU performance has stagnated. In recent years, researchers have started exploring heterogeneous computing to overcome the scaling problems of CPUs and to continue to deliver interactive performance for database applications. In such a hybrid CPU-GPU system, the two processors are connected via PCIe. The PCIe bandwidth of a modern machine is up to 16 GBps, which is much lower than the memory bandwidth of either CPU or GPU. Therefore, data transfer between CPU and GPU is a serious performance bottleneck.

Past work in the database community has focused on using the GPU as a coprocessor, which we call the *coprocessor model*. In this model, data primarily resides in the CPU's main memory. For query execution, data is shipped from the CPU to the GPU over PCIe, so that (some) query processing can happen on the GPU. Results are then shipped back to the CPU. Researchers have worked on optimizing various database operations under the co-processor model: selection [73], join [37, 38, 43, 72, 66, 71, 82], and sort [33, 75]. Several full-fledged GPU-as-coprocessor database query engines have been proposed in recent years. Ocelot [39] provides a hybrid analytical engine

as an extension to MonetDB. YDB [83] is a GPU-based data warehousing engine. Both systems used an operator-at-a-time model, where an operator library containing GPU kernel implementations of common database operators such as scans and joins is invoked on batches of tuples, running each operator to completion before moving on to the next operator. To ensure kernel execution efficiency, tuples are pushed from one operator to another in batches which lets the system hide some of the CPU-GPU data transfer latency. In operator-at-a-time model, each operator needs to write out its output to global memory. As a result, query intermediates end up being written to and read from global memory multiple times which is inefficient.

Kernel fusion [81] attempted to hide inefficiency associated with running multiple kernels for each query like in the operator-at-a-time model. Kernel fusion fused operator kernels with producer-consumer dependency when possible to eliminate redundant data movement. As kernel fusion is applied as a post-processing step, it will miss opportunities where kernel configurations are incompatible. HippogriffDB [46] used GPUs for large scale data warehousing where data resides on SSDs. HippogriffDB claims to achieve $100 \times$ speedup over MonetDB when the ratio of memory bandwidth of GPU to CPU is roughly $5 \times$. More recently, HorseQC [31] proposes pipelined data transfer between CPU and GPU to improve query runtime. A contribution of this thesis is showing that the coprocessor model is slower than running the query efficiently directly on the CPU. To the extent past works showed gains, it is because of comparison against inefficient baselines.

Commercial systems like Omnisci [9], Kinetica [6], and BlazingDB [4] aim to provide real-time analytical capabilities by using GPUs to store large parts (or all) of the working set. The setting used in this thesis is similar to ones used by these systems. Although these systems use a design similar to what we advocate, some have claimed $1000 \times$ performance improvement by using GPUs [1] but have not published rigorous benchmarks against state-of-the art CPU or GPU databases, which is a contribution of this thesis.

Part I

Query Execution on GPU

Chapter 3

GPU Query Execution

3.1 Introduction

In-memory analytics is typically memory bandwidth bound. The improved memory bandwidth of GPUs has led some researchers to use GPUs as coprocessors for analytic query processing [83, 78, 46, 31]. However, previous work leaves several unanswered questions:

- GPU-based database systems have reported a wide range of performance improvement compared to CPU-based database systems, ranging from 2× to 100×. There is a lack of consensus on how much performance improvement can be obtained from using GPUs. Past work frequently compares against inefficient baselines, e.g., MonetDB [83, 78, 46] which is known to be inefficient [56]. The empirical nature of past work makes it hard to generalize results across hardware platforms.
- Past work generally views GPUs strictly as an coprocessor. Every query ends up shipping data from CPU to GPU over PCIe. Data transfer over PCIe is an order of magnitude slower than GPU memory bandwidth, and typically less than the CPU memory bandwidth. As a result, the PCIe transfer time becomes the bottleneck and limits gains. To the extent that past work shows performance improvements using GPUs as an coprocessor, much of those gains may be due

to evaluation against inefficient baselines.

• There has been significant improvement in GPU hardware in recent years. Most recent work on GPU-based database [31] evaluates on GPUs which have memory capacity and bandwidth of 4 GB and 150 GBps respectively, while latest generation of GPUs have 8× higher capacity and bandwidth. These gains significantly improve the attractiveness of GPUs for query processing.

In this chapter, we set out to understand the true nature of performance difference between CPUs and GPUs, by performing rigorous model-based and performancebased analysis of database analytics workloads after applying optimizations for both CPUs and GPUs. To ensure that our implementations are state-of-the-art, we use theoretical minimums derived assuming memory bandwidth is saturated as a baseline, and show that our implementations can typically saturate the memory bus, or when they cannot, describe in detail why they fall short. Hence, although we offer some insights into the best implementations of different operators on CPUs and GPUs, the primary contribution of this chapter is to serve as a guide to implementors as to what sorts of performance differences one should expect to observe in database implementations on modern versions of these different architectures.

Past work has used GPUs mainly as coprocessors. By comparing an efficient CPU implementation of a query processor versus an implementation that uses the GPU as a coprocessor, we show that GPU-as-coprocessor offers little to no gain over a pure CPU implementation, performing worse than the CPU version for some queries. We argue that the right setting is having the working set stored directly on GPU(s).

We developed models and implementations of basic operators: Select, Project, and Join on both CPU and GPU to understand when the ratio of operator runtime on CPUs to runtime on GPUs deviates from the ratio of memory bandwidth of GPU to memory bandwidth of CPU. In the process, we noticed that the large degree of parallelism of GPUs leads to additional materialization. We propose a novel execution model for query processing on GPUs called the *Tile-based execution model*. Instead of looking at GPU threads in isolation, we treat a block of threads ("thread block") as a single execution unit, with each thread block processing a tile of items. The benefit of this tile-based execution model is that thread blocks can now cache tiles in shared memory and collectively process them. This helps avoid additional materialization. This model can be expressed using a set of primitives where each primitive is a function which takes as input of set of tiles and outputs a set of tiles. We call these primitives *block-wide functions*. We present Crystal, a library of block-wide functions that can be used to implement the common SQL operators as well as full SQL queries. Furthermore, we use Crystal to implement the query operators on the GPU and compare their performance against equivalent state-of-the-art implementations on the CPU. We use Crystal to implement the Star-Schema Benchmark (SSB) [57] on the GPU and compare it's performance against our own CPU implementation, a state-of-the-art CPU-based OLAP DBMS and a state-of-the-art GPU-based OLAP DBMS. In both cases, we develop models assuming memory bandwidth is saturated and reason about the performance based on it.

In summary, we make the following contributions:

- We show that previous designs which use the GPU as a coprocessor show no performance gain when compared against a state-of-the-art CPU baseline. Instead, using modern GPU's increased memory capacity to store working set directly on the GPU is a better design.
- We present Crystal, a library of data processing primitives that can be composed together to generate efficient query code that can full advantage of GPU resources.
- We present efficient implementations of individual operators for both GPU and CPU. For each operator, we provide cost models that can accurately predict their performance.
- We describe our implementation of SSB and evaluate both GPU and CPU implementations of it. We present cost models that can accurately predict query runtimes on the GPU and discuss why such models fall short on the CPU.

3.2 Background

In Chapter 2, we reviewed the basics of the GPU architecture and described past approaches to running database analytics workloads on the GPU. In this section, we describe relevant aspects of past approaches to running database analytics workloads on the CPU.

3.2.1 Query Execution on CPU

Database operators have been extensively optimized for modern processors. For joins, researchers have proposed using cache-conscious partitioning to improve hash join performance [50, 19, 17, 16]. Schuh et al. summarized the approaches [69]. For sort, Satish et al. [68] and Wassenberg et al. [79] introduced buffered partitioning for radix sort. Polychroniou et al. [62] presented faster variants of radix sort that use SIMD instructions. Sompolski et al. [74] showed that combination of vectorization and compilation can improve performance of project, selection, and hash join operators. Polychroniou et al. [61] presented efficient vectorized designs for selections, hash tables, and partitioning using SIMD gathers and scatters. Prashanth et al. [52] extended the idea to generate machine code for full queries with SIMD operators. We use ideas from these works, mainly the works of Polychroniou et al. [62, 61] for our CPU implementations.

C-Store [77] and MonetDB [22] were among the first column-oriented engines, which formed the basis for analytical query processing. MonetDB X100 [21] introduced the idea of vectorized execution that was cache aware and reduced memory traffic. Hyper [56] introduced the push-based iteration and compiling queries into machine code using LLVM. Hyper was significantly faster than MonetDB and brought query performance close to that of handwritten C code. We compare the performance of our CPU query implementations against MonetDB [22] and Hyper [56].
```
SELECT SUM(lo_extendedprice * lo_discount) AS revenue
FROM lineorder
WHERE lo_quantity < 25
AND lo_orderdate >= 19930101 AND lo_orderdate <= 19940101
AND lo_discount >= 1 AND lo_discount <= 3;</pre>
```





Figure 3-2: Evaluation of the GPU coprocessor approach on the Star Schema Benchmark

3.3 Failure of the Coprocessor Model

While past work has claimed speedups from using GPUs in the coprocessor model, there is no consensus among past work about the performance improvement obtained from using GPUs, with reported improvements varying from $2 \times$ to $100 \times$.

Consider Q1.1 from the Star Schema Benchmark (SSB) shown in Figure 3-1. For simplicity, assume all column entries are 4-byte integers and L is the number of entries in **lineorder**. An efficient implementation on a CPU will be able to generate the result using a single pass over the 4 data columns. The optimal CPU runtime (R_C) is upper bounded by $16L/B_c$ where B_c is the CPU memory bandwidth. This is an upper bound because, if the predicates are selective, then we may be able to skip entire cache lines while accessing the lo_extendedprice column. In the coprocessor model, we have to ship 4 columns of data to GPU. Thus, the query runtime on the GPU (R_G) is lower bounded by $16L/B_p$ where B_p is the PCIe bandwidth. The bound is hit if we are able to perfectly overlap the data transfer and query execution on GPU. However, since $B_c > B_p$ in modern CPU-GPU setups, $R_C < R_G$, i.e., running the query on CPU yields a lower runtime than the running query with a GPU coprocessor.

To show this empirically, we ran the entire SSB with scale factor 20 on an instance where CPU memory bandwidth is 54 GBps, GPU memory bandwidth is 880 GBps, and PCIe bandwidth is 12.8 GBps. The full workload details can be found in Section 3.7.1 and the full system details can be found in Table 2. We compare the performance of the GPU Coprocessor with two OLAP DBMSs: MonetDB and Hyper. Past work on using GPUs as a coprocessor mostly compared their performance against MonetDB [83, 78, 46] which is known to be inefficient [56]. Figure 3-2 shows the results. On an average, GPU Coprocessor performs $1.5 \times$ faster than MonetDB but it is $1.4 \times$ slower than Hyper. For all queries, the query runtime in GPU coprocessor is bound by the PCIe transfer time. We conclude the reason past work was able to show performance improvement with a GPU coprocessor is because their optimized implementations were compared against inefficient baselines (e.g., MonetDB) on the CPU.

With the significant increase in GPU memory capacity, a natural question is how much faster a system that treats the GPU as the primary execution engine, rather than as an accelerator, can be. We describe our architecture for such a system in the next section.

3.4 Tile-based Execution Model

While a modern CPU can have dozens of cores, a modern GPU like Nvidia V100 can have 5000 cores. The vast increase in parallelism introduces some unique challenges for data processing. To illustrate this, consider running the following simple selection query as a micro-benchmark on both a CPU and a GPU:

QO: SELECT y FROM R WHERE y > v;

On the CPU, the query can be efficiently executed as follows. The data is partitioned equally among the cores. The goal is to write the results in parallel into a contiguous output array. The system maintains a global atomic counter that acts as a cursor that tells each thread where to write the next result. Each core processes its partition by iterating over the entries in the partition one vector of entries at a time, where a vector is about 1000 entries (small enough to fit in the L1 cache). Each core makes a first pass over the first vector of entries to count the number of entries that match the predicate d. The thread increments the global counter by dto allocate output space for the matching records, and then does a second pass over the vector to copy the matched entries into the output array in the allocated range of the output. Since the second pass reads data from L1 cache, the read is essentially free. The global atomic counter is a potential point of contention. However, note that each thread updates the counter once for every 1000+ entries and there are only around 32 threads running in parallel at any point. The counter ends up not being the bottleneck and the total runtime is approximately $\frac{D}{B_C} + \frac{D\sigma}{B_C}$ where D is the size of the column, and B_C is the memory bandwidth on the CPU.

We could run the same plan on the GPU, partitioning the data up among the thousands of threads. However, GPU threads have significantly fewer resources per thread. On the Nvidia V100, each GPU thread can only store roughly 24 4-byte entries in shared memory at full occupancy, with 5000 threads running in parallel. Here, the global atomic counter ends up becoming the bottleneck as all the threads will attempt to increment the counter to find the offset into the output array. To work around this, existing GPU-based database systems would execute this query in 3 steps as shown in Figure 3-3(a). The first kernel K_1 would be launched across a large number of threads. In it, each thread would read in column entries in a strided fashion (interleaved by thread number) and evaluate the predicate to count the number of entries matched. After processing all elements, the total number of entries matched by thread t. The second kernel K_2 would use the count array to compute

the prefix sum of the count and store this in another array \mathbf{pf} . Recall that for an array A of k elements, the prefix sum p_A is a k element array where $p_A[j] = \sum_{i=0}^{j-1} A_j$. Thus, the i^{th} entry in \mathbf{pf} indicates the offset at which the ith thread should write its matched results to in the output array \mathbf{o} . Databases used an optimized routine from a CUDA library like Thrust [11] to run it efficiently in parallel. The third kernel K_3 would then read in the input column again; here the i^{th} thread again scans the i^{th} stride of the input, using $\mathbf{pf}[i]$ to determine where to write the satisfying records. Each thread also maintains a local counter c_i , initially set to 0. Specifically for each satisfying entry, thread i writes it to $\mathbf{pf}[i] + c_i$ and then increments c_i . In the end, $\mathbf{o}[\mathbf{pf}[t]] \ldots \mathbf{o}[\mathbf{pf}[t+1] - 1]$ will contain the matched entries of thread t.

The above approach shifts the task of finding offsets into the output array to an optimized prefix sum kernel whose runtime is a function of T (where T is the number of threads $(T \ll n)$), instead of finding it inline using atomic updates to a counter. As a result, the approach ends up being significantly faster than the naive translation of the CPU approach to the GPU. However, there are a number of issues with this approach. First, it reads the input column from global memory twice, compared to doing it just once with the CPU approach. It also reads/writes to intermediate structures count and pf. Finally, each thread writes to a different location in the output array resulting in random writes. To address these issues, we introduce the **Tile-based execution model**.

Tile-based processing extends the vector-based processing on CPU where each thread processes a vector at a time to the GPU. Figure 3-4 illustrates the model. Threads on the GPU are grouped into thread blocks. Threads within a thread block can communicate through shared memory and can synchronize through barriers. Hence, even though a single thread on the GPU at full occupancy can hold only up to 24 integers in shared memory, a single thread block can hold a significantly larger group of elements collectively between them in shared memory. We call this unit a **Tile**. In the Tile-based execution model, instead of viewing each thread as an independent execution unit, we view a thread block as the basic execution unit with each thread block processing a tile of entries at a time. One key advantage of this



Figure 3-3: Running selection on GPU



Figure 3-4: Vector-based to Tile-based execution models.

approach is that after a tile is loaded into shared memory, subsequent passes over the tile will be read directly from shared memory and not from global memory, avoiding the second pass through global memory described in the implementation above.

Figure 3-3(b) shows how selection is implemented using the tile-based model. The entire query is implemented as a single kernel instead of three. Figure 3-5 shows a sample execution with a tile of size 16 and a thread block of 4 threads for the predicate y > 5. Note that this is just for illustration, as most modern GPUs would use a



Figure 3-5: Query Q0 Kernel running y > 5 with tile size 16 and thread block size 4

thread block size that is a multiple of 32 (the warp size) and the number of elements loaded would be 4–16 times the size of the thread block. We start by initializing the global counter to 0. The kernel loads a tile of items from global memory into the shared memory. The threads then apply the predicate on all the items in parallel to generate a bitmap. For example, thread 0 evaluates the predicate for elements 0,4,8,12 (shown in red). Each thread then counts the number of entries matched per thread to generate a histogram. The thread block co-operatively computes the prefix sum over the histogram to find the offset each thread writes to in shared memory. In the example, threads 0,1,2,3 match 2,1,4,3 entries respectively. The prefix sum entries 0,2,3,7 tell us thread 0 should write its matched entries to output at index 0, thread 1 should write starting at index 2, etc. We increment a global counter atomically by total number of matched entires to find the offset at which the thread block should write in the output array. The shuffle step uses the bitmap and the prefix sum to create a contiguous array of matched entries in shared memory. The final write step copies the contiguous entries from shared memory to global memory at the right offset.

By treating the thread block as an execution unit, we reduce the number atomic updates of the global counter by a factor of size of tile T. The kernel also makes a single pass over the input column with the **Gen Shuffled Tile** ensuring that the final write to the output array is coalesced, solving both problems associated with



(a) SELECT y FROM R WHERE y > v(b) SELECT y FROM R WHERE x > w AND y > v

Figure 3-6: Implementing queries using Crystal

approach used in previous GPU databases.

The general concept of the tile-based executing model i.e., dividing data into tiles and mapping threadblocks to tiles has been used in other domains like image processing [40] and high performance computing [11]. However, to the best of our knowledge this is the first work that uses it for database operations. In the next section, we present Crystal, a library of data processing primitives that can be composed together to implement SQL queries on the GPU.

3.5 Crystal Library

The kernel structure in Figure 3-5 contains a series of steps where each is a function that takes as input a set of tiles, and outputs a set of tiles. We call these primitives *block-wide functions*. A block-wide function is a device function¹ that takes in a set of tiles as input, performs a specific task, and outputs a set of tiles. Instead

 $^{^1\}mathrm{Device}$ functions are functions that can be called from kernels on the GPU

Primitive	Description	
BlockLoad	Copies a tile of items from global memory to shared memory. Uses vector	
	instructions to load full tiles.	
BlockLoadSel	Selectively load a tile of items from global memory to shared memory based on	
	a bitmap.	
BlockStore	Copies a tile of items in shared memory to device memory.	
BlockPred	Applies a predicate to a tile of items and stores the result in a bitmap array.	
BlockScan	Co-operatively computes prefix sum across the block. Also returns sum of all	
	entries.	
BlockShuffle	Uses the thread offsets along with a bitmap to locally rearrange a tile to create	
	a contiguous array of matched entries.	
BlockLookup	Returns matching entries from a hash table for a tile of keys.	
BlockAggregate	Uses hierarchical reduction to compute local aggregate for a tile of items.	

Table 3.1: List of block-wide functions in Crystal

of reimplementing these block-wide functions for each query, which would involve repetition of non-trivial functions, we developed a library called **Crystal**.

Crystal² is a library of templated CUDA device functions that implement the full set of primitives necessary for executing typical analytic SQL SPJA analytical queries. Figure 3-6(a) shows an sketch of the simple selection query implemented using blockwide functions. Figure 5-7 shows the query kernel of the same query implemented with Crystal. We use this example to illustrate the key features of Crystal. The input tile is loaded from the global memory into the thread block using BlockLoad. BlockLoad internally uses vector instructions when loading a full tile and for the tail of the input array that may not form a perfect tile, it is loaded in a striped fashion element-at-a-time. BlockPred applies the predicate to generate the bitmap. A key optimization that we do in Crystal is instead of storing the tile in shared memory, in cases where the array indices are statically known before hand, we choose to use registers to store the values. In this case, items (which contains entries loaded from the column) and **bitmap** are stored in registers. Hence, in addition to 24 4-byte values that a thread can store in shared memory, this technique allows us to use roughly equal amount of registers available to store data items. Next we use BlockScan to compute the prefix sum. BlockScan internally implements a hierarchical block-wide parallel prefix-sum approach [36]. This involves threads accessing bitmap entries of

²The source code of the Crystal library is available at https://github.com/anilshanbhag/crystal

```
1 // Implements SELECT y FROM R WHERE y > v
2 // NT => NUM_THREADS
3 // IPT => ITEMS_PER_THREAD
4 template<int NT, int IPT>
  __global__ void Q(int* y, int* out, int v, int* counter) {
     int tile_size = get_tile_size();
6
     int offset = get_tile_offset();
7
     __shared__ struct buffer {
8
       int col[NT * IPT];
9
       int out[NT * IPT];
10
    };
11
     int items[IPT];
12
     int bitmap[IPT];
13
     int indices[IPT];
14
15
    BlockLoadInt<NT, IPT>(col+offset,items,buffer.col,tile_size);
16
    BlockPredIntGT<NT, IPT>(items, buffer.col, cutoff, bitmap);
17
    BlockScan<NT, IPT>(bitmap,indices,buffer.col,
18
       num_selections,tile_size);
19
20
     if(threadIdx.x == 0)
21
       o_off = atomic_update(counter,num_selections);
22
23
    BlockShuffleInt<NT, IPT>(items, indices, buffer.out);
24
    BlockStoreInt<NT, IPT>(buffer.out,out + o_off,num_selections);
25
26 }
```

Figure 3-7: Query Q0 Kernel Implemented with Crystal

other threads — for this we load bitmap into shared memory, reusing buffer.col shared memory buffer used for loading the input column. Shared memory is order of magnitude faster than global memory, hence loads and stores to shared memory in this case do not impact performance. After atomic update to find offset in output array, BlockShuffle is used to reorder the array and finally we use BlockStore to write to output array. The code skips some minor details like when the atomic update happens, since it is executed on thread 0, the global offset needs to be communicated back to other threads through shared memory.

In addition to allowing users to write high performance kernel code that as we show later can saturate memory bandwidth, there are two usability advantages of using Crystal:

- Modularity: Block-wide functions in Crystal make it easy to use non-trivial functions and reduce boilerplate code. For example, BlockScan, BlockLoad, BlockAggregate each encapsulate 10's to 100's of lines of code. For the selection query example, Crystal reduces lines of code from more than 300 to less than 30.
- Extensibility: Block-wide functions makes it is fairly easy to implement query kernels of larger queries. Figure 3-6(b) shows the implementation of a selection query with two predicates. Ordinary CUDA code can be used along with Crystal functions.

Crystal supports loading partial tiles like in Figure 3-6(b). If a selection or join filters entries, we use BlockLoadSel to load items that matched the previous selections based on a bitmap. In this case, the thread block internally allocate space for the entire tile, however, only matched entries are loaded from global memory. Table 3.1 briefly describes the block-wide functions currently implemented in the library.

To evaluate Crystal, we look at two microbenchmarks:

1) We evaluate the selection query Q0 with size of input array as 2^{29} and selectivity is 0.5. We vary the tile sizes. We vary the thread block sizes from 32 to 1024 in multiples of 2. We have three choices for the number of items per thread: 1,2,4. Figure 3-8 shows the results. As we increase the thread block size, the number of global atomic updates done reduces and hence the runtime improves until the thread block size approaches 512 after which it deteriorates. Each streaming multiprocessor on the GPU holds maximum of 2048 threads, hence, having large thread blocks reduces number of independent thread blocks. This affects utilization particularly when thread blocks are using synchronization heavily. Having 4 items per thread allows to effectively load the entire block using vector instructions. With 2 items per thread, there is reduced benefit for vectorization as half the threads are empty. With 1 item per thread there is no benefit. The best performance is seen with thread block size of 128/256 and items per thread equal to 4. In these cases, as we show later in Section 3.6.2, we saturate memory bandwidth and hence achieve optimal



Figure 3-8: Q0 performance with varying tile sizes

performance.

2) We evaluated the selection query Q0 using two approaches: independent threads approach (Figure 3-3(a)) and using Crystal (Figure 3-3(b)). The number of entries in the input array is 2^{29} and selectivity is 0.5. The runtime with the independent threads approach is 19ms compared to just 2.1ms when using Crystal. Almost all of the performance improvement is from avoiding atomic contention and being able to reorder matched entries to write in a coalesced manner.

Across all of the workloads we evaluated, we found that using thread block size 128 with items per thread equal to 4 is indeed the best performing tile configuration. In the rest of the chapter, we use this configuration for all implementations. All the implementations in this chapter were implemented by hand in CUDA C++ using Crystal. Since Crystal's block-wide functions are standard device functions, they can also called directly from LLVM IR.

In the next section, we show how to use these block-wide functions to build efficient operators on a GPU and compare their performance to equivalent CPU implementations.

3.6 Operators on GPU vs CPU

In order to understand the true nature of performance difference of queries on GPU vs. CPU, it is important to understand the performance difference of individual query operators. In this section, we compare the performance of basic operators: project, select, hash join, and sorting/partitioning on GPU and CPU with the goal of understanding how the ratio of runtime on GPU to runtime on CPU compares to the bandwidth ratio of the two devices. We use block-wide functions from **Crystal** to implement the operators on GPU and use equivalent state-of-the-art implementations on CPU. We also present a model for each of the operators assuming the operator saturates memory bandwidth and show that in most cases the operators indeed achieve these limits. We use the model to explain the performance difference between CPU and GPU. For the micro-benchmarks, we use a machine where GPU memory bandwidth is 880GBps and CPU memory bandwidth is 54GBps, resulting in a bandwidth ratio of 16.2 (see Section 3.7 for system details). In all cases, we assume that the data is already in the respective device's memory.

3.6.1 Project

We consider two forms of projection queries: one that computes a linear combination of columns (Q1) and one involving user defined function (Q2) as shown below:

```
Q1: SELECT ax_1 + bx_2 FROM R;
Q2: SELECT \sigma(ax_1 + bx_2) FROM R;
```

where x_1 and x_2 are 4-byte floating point values. The number of entries in the input array is 2^{29} . σ is the sigmoid function (i.e., $\sigma(x) = \frac{1}{1+e^{-x}}$) which can represent the output of a logistic regression model. Note that Q1 consists of basic arithmetic and will certainly be bandwidth bound. Q2 is representative of the most complicated projection we will likely see in any SQL query.

On the CPU side, we implement two variants: CPU and CPU-Opt. CPU uses a multi-threaded projection where each thread works on a partition of the input data. CPU-Opt extends CPU with two extra optimizations: (1) non-temporal writes and (2)

SIMD instructions. Non-temporal writes are write instructions that bypass higher cache levels and write out an entire cache line to main memory without first loading it to caches. SIMD instructions can further improve performance. With a single AVX2 instruction, for example, a modern x86 system can add, subtract, multiply, or divide a group of 8 4-byte floating point numbers, thereby improving the computation power and memory bandwidth utilization.

On the GPU side, we implement a single kernel that does two BlockLoad's to load the tiles of the respective columns, computes the projection and does a BlockStore to store it in the result array.

Model: Assuming the queries can saturate the memory bandwidth, the expected runtime of Q1 and Q2 is

$$runtime = \frac{2 \times 4 \times N}{B_r} + \frac{4 \times N}{B_w}$$

where N is the number of entries in the input array and B_r and B_w are the read and write memory bandwidth, respectively. The first term of the formula models the runtime for loading columns x_1 and x_2 , each containing 4-byte floating point numbers. The second term models the runtime for writing the result column back to memory, which also contains 4-byte floating point numbers. Note that this formula works for both CPU and GPU, by plugging in the corresponding memory bandwidth numbers.

Performance Evaluation: Figure 3-9 shows the runtime of queries Q1 and Q2 on both CPU and GPU (shown as bars) as well as the predicted runtime based on the model (shown as dashed lines). The performance of Q1 on both CPU and GPU is memory-bandwidth bound. CPU-Opt performs better than CPU due to the increased memory bandwidth efficiency. GPU performs substantially better than both CPU implementations due to its much higher memory bandwidth. The ratio of runtime of CPU-Opt to GPU is 16.56 which is close to the bandwidth ratio of 16.2. The minor difference is because read bandwidth is slightly lower than write bandwidth on the CPU and the workload has a read:write ratio of 2:1.

A simple multi-threaded implementation of Q2 (i.e., CPU) does not saturate mem-



Figure 3-9: GPU vs CPU performance on the project microbenchmark

ory bandwidth and is compute bound. After using the SIMD instructions (i.e, CPU-Opt), performance improves significantly and the system is close to memory bandwidth bound. The ratio of runtime of CPU-Opt to GPU for Q2 is 17.95. This shows that even for fairly complex projections, good implementations on modern CPUs are able to saturate memory bandwidth. GPUs do significantly better than CPUs due to their high memory bandwidth, with the performance gain equal to the bandwidth ratio.

3.6.2 Select

We now turn our attention to evaluating selections, also called selection scans. Selection scans have re-emerged for main-memory query execution and are replacing tradition unclustered indexes in modern OLAP DBMS [63]. We use the following micro-benchmark to evaluate selections:

Q3: SELECT y FROM R WHERE y < v;

where y and v are both 4-byte floating point values. The size of input array is 2^{29} . We vary the selectivity of the predicate from 0 to 1 in steps of 0.1.

To evaluate the above query on a multi-core CPU, we use the CPU implemen-

for each y in R:
output[i] = y
i += (y > v)

(a) With branching

(b) With predication

Figure 3-10: Implementing selection scan

tation described earlier in Section 3.4. We evaluate three variants. The "naive" branching implementation (CPU If) implements the selection using an if-statement, as shown in Figure 3-14(a). The main problem with the branching implementation is the penalty for branch mispredictions. If the selectivity of the condition is neither too high nor too low, the CPU branch predictor is unable to predict the branch outcome. This leads to pipeline stalls that hinder performance. Previous work has shown that the branch misprediction penalty can be avoided by using branch-free *predication* technique [64]. Figure 3-14(b) illustrates the predication approach. Predication transforms the branch (control dependency) into a data dependency. CPU Pred implements selection scan with predication. More recently, vectorized selection scans have been shown to improve on CPU Pred by using selective stores to buffer entries that satisfy selection predicates and writing out entries using streaming stores [61]. CPU SIMDPred implements this approach.

On the GPU, the query is implemented as a single kernel as described in Section 3.4 and as shown in Figure 3-3(b). We implement two variants: GPU If implements the selection using an if-statement and GPU Pred implements it using predication.

Model: The entire input array is read and only the matched entries are written to the output array. Assuming the implementations can write out the matched entries efficiently and saturate memory bandwidth, the expected runtime is:

$$runtime = \frac{4 \times N}{B_r} + \frac{4 \times \sigma \times N}{B_w}$$

where N is the number of entries in the input array, B_r and B_w are the read and write bandwidth of the respective device, and σ is the predicate selectivity.

Performance Evaluation: Figure 3-11 shows the runtime of the three algorithms



Figure 3-11: GPU vs CPU performance on the select microbenchmark

on CPU, two algorithms on GPU, and the performance models. CPU Pred does better than CPU If at all selectivities except 0 (at 0, CPU If does no writes). Across the range, CPU SIMDPred does better than the two scalar implementations. On GPU, there is no performance difference between GPU Pred and GPU If — A single branch misprediction does not impact performance on the GPU. Both CPU SIMDPred and GPU If/Pred closely track their respective theoretical models which assume saturation of memory bandwidth. The average runtime ratio of CPU-to-GPU is 15.8 which is close to the bandwidth ratio 16.2. This shows that with efficient implementations, CPU implementations saturate memory bandwidth for selections and the gain of GPU over CPU is equal to the bandwidth ratio.

3.6.3 Hash Join

Hash join is the most popular algorithm used for executing joins in a database. Hash joins have been extensively studied in the database literature, with many different hash join algorithms proposed for both CPUs and GPUs [17, 19, 44, 16, 24, 37]. The most commonly used hash join algorithm is the no partitioning join, which uses a

non-partitioned global hash table. The algorithm consists of two phases: in the *build phase*, the tuples in one relation (typically the smaller relation) are used to populate the hash table in parallel; in the *probe phase*, the tuples in the other relation are used to probe the hash table for matches in parallel. For our microbenchmark, we focus on the following join query:

Q4: SELECT SUM(A.v + B.v) AS checksum FROM A,B WHERE A.k = B.k

where each table A and B consists of two 4-byte integer columns k, v. The two tables are joined on key k. We keep the size of the probe table fixed at 256 million tuples, totaling 2 GB of raw data. We use a hash table with 50% fill rate. We vary the size of the build table such that it produces a hash table of the desired size in the experiment. We vary the size of the hash table from 8KB to 1GB. The microbenchmark is the same as what past works use [19, 17, 16, 69].

In this section, we mainly focus on the probe phase which forms the majority of the total runtime. We discuss briefly the difference in execution with respect to build time at the end of the section. There are many hash table variants, in this section we focus on linear probing due to its simplicity and regular memory access pattern; our conclusions, however, should apply equally well to other probing approaches. Linear probing is an open addressing scheme that, to either insert an entry or terminate the search, traverses the table linearly until an empty bucket is found. The hash table is simply an array of slots with each slot containing a key and a payload but no pointers.

On the CPU side, we implemented three variants of linear probing. (1) CPU Scalar implements a scalar tuple-at-a-time join. The probing table is partitioned equally among the threads. Each thread iterates over its entries and for each entry probes the hash table to find a matching entry. On finding a match, it adds A.v + B.v to its local sum. At the end, we add the local sum to the global sum using atomic instructions. (2) CPU SIMD implements vertical vectorized probing in a hash table [61]. The key idea in vertical vectorization is to process a different key per SIMD lane and use gathers to access the hash table. Assuming W vector lanes, we process W

different input keys on each loop iteration. In every round, for the set of keys that have found their matches, we calculate their sum, add it to a local sum, and reload those SIMD lanes with new keys. (3) Finally, CPU Prefetch adds group prefetching to CPU Scalar [24]. For each loop iteration, software prefetching instructions are inserted to load the hash table entry that will be accessed a certain number of loop iterations ahead. The goal is to better hide memory latency at the cost of increased number of instructions.

On the GPU side, we implemented the join as follows. We load in a tile of keys and payloads from the probe side using BlockLoad; the threads iterate over each tile independently to find matching entries from the hash table. Each thread maintains a local sum of entries processed. After processing all entries in a tile, we use BlockAgg to aggregate the local sums within a thread block into a single value and increment a global sum with it.

Model: The probe phase involves making random accesses to the hash table to find the matching tuple from the build side. Every random access to memory ends up reading an entire cache line. However, if the size of hash table is small enough such that it can be cached, then random accesses no longer hit main memory and performance improves significantly. We model the runtime as follows:

1) If the hash table size is smaller than the size of the K^{th} level cache, we expect the runtime to be:

$$runtime = max(\frac{4 \times 2 \times |P|}{B_r}, (1 - \pi_{K-1})(\frac{|P| \times C}{B_K}))$$

where |P| is the cardinality of the probe table, B_r is the read bandwidth from device memory, C is the cache line size accessed on probe, B_K is the bandwidth of level Kcache in which hash table fits and π_{K-1} is the probability of an access hitting a K-1level cache. The first term is the time taken to scan the probe table from device memory. The second term is the time for probing the hash table. Note that each probe accesses an entire cache line. If the size of level K cache is S_K and size of the hash table is H, we define cache hit ratio $\pi_K = min(S_K/H, 1)$. The total runtime will be bounded by either the device memory bandwidth or the cache bandwidth. Hence, the runtime is the maximum of the two terms.

2) If the hash table size is larger than the size of the last level cache, we expect the runtime to be:

$$runtime = \frac{4 \times 2 \times |P|}{B_r} + (1 - \pi)(\frac{|P| \times C}{B_r})$$

where π is the probability that the accessed cache line is the last level cache.

Performance Evaluation: Figure 3-12 shows the performance evaluation of different implementations of Join. Both CPU and GPU variants exhibit step increase in runtime when the hash table size exceeds the cache size of a particular level. On the CPU, the step increases happen when the hash table size exceeds 256KB (L2 cache size) and 20MB (L3 cache size). On the GPU, the step increase happens when the hash table size exceeds 6MB (L2 cache size).

We see that CPU SIMD performs worse than CPU Scalar, even when the hash table is cache-resident. CPU-SIMD uses AVX2 instructions with 256-bit registers which represent 8 lanes of 32-bit integers. With 8 lanes, we process 8 keys at a time. However, a single SIMD gather used to fetch matching entries from the hash table can only fetch 4 entries at a time (as each hash table lookup returns an 8 byte slot. i.e., 4-byte key and 4-byte value, with 4 lookups filling the entire register). As a result, for each set of 8 keys, we do 2 SIMD gathers and then de-interleave the columns into to 8 keys and 8 values. This added overhead of extra instructions does not exist in the scalar version. CPU SIMD is also brittle and not easy to extend to cases where hash table slot size is larger than 8 bytes. Note that past work has evaluated vertical vectorization with key-only build relations which do not exhibit this issue [61, 52]. Comparing CPU Prefetch to CPU Scalar shows that there is limited improvement from prefetching when data size is larger than the L3 cache size. When the hash table fits in cache, prefetching degrades the performs due to added overhead of the



Figure 3-12: GPU vs CPU performance on the join microbenchmark

prefetching instructions.

Due the step change nature of the performance curves, the ratio of the runtimes varies based on hash table size. When the hash table size is between 32KB and 128KB, the hash table fits in L2 on both CPU and GPU. In this segment, we observe that the runtime is bound by DRAM memory bandwidth on CPU and L2 cache bandwidth on the GPU. The average gains are roughly $5.5 \times$ which is in line with the model. When the hash table size is between 1MB and 4MB, the hash table fits in the L2 on the GPU and in the L3 cache on the CPU. The ratio of runtimes in this segment is $14.5 \times$ which is the ratio of L2 cache bandwidth on GPU to the L3 cache bandwidth on the CPU. Finally when the hash table size is larger than 128MB, the hash table does not fit in cache on either GPU or CPU. The granularity of reads from global memory is 128B on GPU while on CPU it is 64B. Hence, random accesses into the hash table read twice the data on GPU compared to CPU. Given the bandwidth ratio is 16.2x, we would expect it as roughly 8.1x, however it is 10.5x due to memory stalls. The fact that actual CPU results are slower than CPU Model is because the

model assumes maximum main memory bandwidth, which is not achievable as the hash table causes random memory access patterns.

Discussion: The runtime of the build phase in the microbenchmark shows a linear increase with size of the build relation. The build phase runtimes are less affected by caches as writes to hash table end up going to memory.

In this section, we modeled and evaluated the no partitioning join. Another variant of hash join is the partitioned hash join. Partitioned hash joins use a partitioning routine like radix partitioning to partition the input relations into cache-sized chunks and in the second step run the join on the corresponding partitions. Efficient radixbased hash join algorithms (*radix join*) have been proposed for CPUs [17, 19, 16, 24] and for the GPUs [66, 71]. Radix join requires the entire input to be available before the join starts and as a result intermediate join results cannot be pipelined. Hence, while radix join is faster for a single join, radix joins are not used for queries with multiple joins. While we do not explicitly model/evaluate radix joins, in the next section we discuss the radix partitioning routine that is the key component of such joins. That discussion shows that a careful radix partition implementation on both GPU and CPU are memory bandwidth bound, and hence the performance difference is roughly equal to the bandwidth ratio.

3.6.4 Sort

In this section, we evaluate the performance of sorting 32-bit key and 32-bit value arrays based on the key. According to literature, the fastest sort algorithm for this workload is the radix sort algorithm. We start by describing the Least-Significant-Bit (LSB) radix sort on the CPU [62] and on the GPU [53]. LSB radix sort is the fastest for the workload on the CPU. We describe why the LSB radix sort does poorly in comparison on the GPU and why an alternate version called Most-Significant-Bit (MSB) radix sort does better on the GPU [75]. We present a model for the runtime of the radix sort algorithm and then analyze the performance characteristics of radix partitioning on CPU vs GPU. Our implementations are primarily based on previous work but this is first time that these algorithm are compared to each other. The LSB radix sort internally comprises a sequence of radix partition passes. Given an array A, radix r, and start bit a, a radix partition pass partitions the elements of the input array A into a contiguous array of 2^r output partitions based on value of r-bits e[a:a+r) (i.e., radix) of the key e. Both on the CPU and GPU, radix partitioning involves two phases. In the first phase (*histogram phase*), each thread (CPU) / thread block (GPU) computes a histogram over its entries to find the number of entries in each partition of the 2^r partitions. In the second phase (*data shuffling phase*), each thread (CPU) / thread block (GPU) maintains an array of pointers initialized by the prefix sum over the histogram and writes entries to the right partition based on these offsets. The entire sorting algorithm contains multiple radix partition passes, with each pass looking at a disjoint sets of bits of the key e starting from the lowest bits e[0:r) to highest bits e[k-r:k) (where k is the bit-length of the key).

On the CPU, we use the implementation of Polychroniou et al. [62]. In the histogram phase, each thread makes one pass over its partition, for each entry calculating its radix value and incrementing the count in the histogram (stored in the L1 cache). For the shuffle phase, we first compute a prefix sum over the histograms of all the threads (a 2D array of dimension $2^r \times t$ where t is the number of threads) to find the partition offsets for each of the threads. Next, each thread makes a pass over its partition using gathers and scatters to increment the counts in its offset array and writing to right offsets in output array. The implementation makes a number of optimizations to achieve good performance. Interested reader can refer to [62] for more details.

On the GPU, we implemented LSB radix sort based on the work of Merrill et al. [53]. In the histogram phase, each thread block loads a tile, computes a histogram that counts the number of entries in each partition, and writes it out to global memory. Prefix sum is used to find the partition offsets for each thread block in the output array. Next, in the shuffling phase each thread block reads in its offset array. The radix partitioning pass described above need to do stable partitioning i.e., ensures that for two entries with the same radix, the one occurring earlier in the input array also occurs earlier in the output array. Now on the GPU, in order to ensure stable partitioning for LSB radix sort we need to internally generate an offsets array for each thread from the the thread block offset array. For an r-bit radix partitioning, we need 2^r size histogram per thread. A number of optimizations have been proposed to store the histogram efficiently in registers, details of which are described in [53]. Due to restriction on number of registers available per thread, stable radix partitioning pass can only process 7-bits at a time.

Recently, Stehle et al. [75] presented an MSB radix sorting algorithm for the GPU. The MSB radix sort does not require stable partitioning. As a result, in the shuffle phase, we can just maintain a single offset array of size 2^r for the entire thread block. This allows MSB radix sort to process up to 8-bits at a time. Hence, the MSB radix sort to sort array of 32-bit keys with 4 passes each processing 8-bits at a time. On the other hand, LSB radix sort can processes only 7-bits at a time, and hence needs 5 radix partitioning passes processing 6,6,6,7,7 bits each.

Model: In the histogram phase, we read in the key column and write out a tiny histogram. The expected runtime is:

$$runtime_{histogram} = \frac{4 \times R}{B_r}$$

where R is the size of the input array and B_r is the read bandwidth. In the shuffle phase, we read both the key and payload column and at the end write out the radix partitioned key and payload columns. If the step is memory bandwidth bound, the runtime is expected to be:

$$runtime_{shuffle} = \frac{2 \times 4 \times R}{B_r} + \frac{2 \times 4 \times R}{B_w}$$

where B_w is the write bandwidth.

Performance Evaluation: We evaluate the performance of histogram and shuffle phase of the three variants: CPU Stable (stable partitioning on CPU), GPU Stable



(b) Radix shuffling on CPU and GPU

Figure 3-13: GPU vs CPU performance on the sort microbenchmark

(stable partitioning on GPU), and GPU Unstable (unstable partitioning on GPU). We set the size of the input arrays at 256 million entries and vary the number of radix bits we partition on. Figure 3-13a shows the results for the histogram phase. Note that in the histogram phase there is no difference between GPU Stable and GPU Unstable. The histogram pass is memory bandwidth bound on both the CPU and GPU. Figure 3-13b shows the results for the shuffle phase. GPU Stable is able to partition up to 7-bits at a time whereas GPU Unstable is able to partition 8-bits at a time. CPU Stable is able to partition up to 8-bits a time while remaining bandwidth bound. Beyond 8-bits, the size of the partition buffers needed exceeds the size of L1 cache and the performance starts to deteriorate.

Now that we have the radix partitioning passes, we look at the sort runtime. On the CPU, we use stable partitioning to implement LSB radix sort. It ends up running 4 radix partitioning passes each looking at 8-bits at time. On the GPU, MSB radix

for each y in R:	<pre>for each y in R:</pre>	
if y > v:		
<pre>output[i++] = v</pre>	i += (y > v)	
(a) With branching	(b) With predication	

Figure 3-14: Implementing selection scan

sort also sorts the data with 4 passes each processing 8-bits at a time. The time taken to sort 2^{28} entries is 464ms on the CPU and 27.08ms on the GPU. The runtime gain is $17.13 \times$ which is close to the bandwidth ratio of $16.2 \times$.

3.7 Workload Evaluation

Now that we have a good understanding of how individual operators behave on both CPU and GPU, we will evaluate the performance of a workload of full SQL queries on both hardware platforms. We first describe the query workload we use in our evaluation. We then present a high-level comparison of the performance of queries running on GPU implemented with the tile-based execution model versus our own equivalent implementation of the queries on the CPU. We also report the performance of Hyper [56] on CPU and Omnisci [9] on the GPU which are both state-of-the-art implementations. As a case study, we provide a detailed performance breakdown of two of the queries to explain the performance gains. Finally, we present a dollar-cost comparison of running queries on CPU and GPU.

We use two platforms for our evaluation. For experiments run on the CPU, we use a machine with a single socket Skylake-class Intel i7-6900 CPU with 8 cores that supports AVX2 256-bit SIMD instructions. For experiments run on the GPU, we use an instance which contains an Nvidia V100 GPU. We measured the bidirectional PCIe transfer bandwidth to be 12.8GBps. More details of the two instances are shown in Table 2. Each system is running on Ubuntu 16.04 and the GPU instance has CUDA 10.0. In our evaluation, we ensure that data is already loaded into the respective device's memory before experiments start. We run each experiment 3 times and report the average measured execution time.

3.7.1 Workload

For the full query evaluation, we use the Star Schema Benchmark (SSB) [57] which has been widely used in various data analytics research studies [46, 83, 78, 31]. SSB is a simplified version of the more popular TPC-H benchmark. It has one fact table *lineorder* and four dimension tables *date, supplier, customer, part* which are organized in a star schema fashion. There are a total of 13 queries in the benchmark, divided into 4 query flights. In our experiments we run the benchmark with a scale factor of 20 which will generate the fact table with 120 million tuples. The total dataset size is around 13GB.

3.7.2 Performance Comparison

In this section, we compare the query runtimes of benchmark queries implemented using block-wide functions on the GPU (Standalone GPU) to an equivalent efficient implementation of the query on the CPU (Standalone CPU). We also compare against Hyper (Hyper), a state-of-the-art OLAP DBMS and Omnisci (Omnisci), a commercial GPU-based OLAP DBMS.

In order to ensure a fair comparison across systems, we dictionary encode the string columns into integers prior to data loading and manually rewrite the queries to directly reference the dictionary-encoded value. For example, a query with predicate $s_region = 'ASIA'$ is rewritten with predicate $s_region = 2$ where 2 is the dictionary-encoded value of 'ASIA'. Some columns have a small number of distinct values and can be represented/encoded with 1-2 byte values. However, in our benchmark we make sure all column entries are 4-byte values to ensure ease of comparison with other systems and avoid implementation artifacts. Our goal is to understand the nature of the performance gains of equivalent implementations on GPU and CPU, and not to achieve best storage layout. We store the data in columnar format with each column represented as an array of 4-byte values. On the GPU, we use a thread block size of 256 with tile size of 1024 (= 4×256) resulting in 4 entries per thread per tile.

Platform	CPU	GPU
Model	Intel i7-6900	Nvidia V100
Cores	8 (16 with SMT)	5000
Memory Capacity	64 GB	32 GB
L1 Size	32KB/Core	$16 \mathrm{KB/SM}$
L2 Size	256KB/Core	6MB (Total)
L3 Size	20MB (Total)	-
Read Bandwidth	53GBps	880GBps
Write Bandwidth	55GBps	880GBps
L1 Bandwidth	-	$10.7 \mathrm{TBps}$
L2 Bandwidth	-	2.2TBps
L3 Bandwidth	157GBps	-

Table 3.2: Hardware Specifications



Figure 3-15: Star Schema Benchmark Queries

Figure 5-11 shows the results. Comparing Standalone CPU to Hyper shows that the former does on an average 1.17x better than the latter. We believe Hyper is missing vectorization opportunities and using a different implementation of hash tables. The comparison shows that our implementation is a fair comparison and it is quite competitive compared to a state-of-the-art OLAP DBMS. We also compared against MonetDB [22], a popular baseline for many of the past works on GPU-based databases. We found that the Standalone CPU is on an average $2.5 \times$ faster than MonetDB. We did not include it in the figure as it made the graph hard to read. We also tried to compare against Pelaton with relaxed-operator fusion [52]. We found that the system could not load the scale factor 20 dataset. Scaling down to scale factor 10, its queries were significantly slower (>5×) than Hyper or our approach.

Comparing Standalone GPU to Omnisci, we see that our GPU implementation does significantly better than Omnisci with an average improvement of around $16 \times$. Both methods run with the entire working set stored on the GPU. Omnisci treats each GPU thread as an independent unit. As a result, it does not realize benefits of blocked loading and better GPU utilization got from using the tile-based model. The comparison of Standalone GPU against Omnisci and Standalone CPU to Hyper serve as a sanity check and show that our query implementations are quite competitive.

Comparing Standalone GPU to Standalone CPU, we see that the Standalone GPU is on average $25 \times$ faster than the CPU implementation. This is higher than the bandwidth ratio of 16.2. This is surprising given that in Section 3.6 we saw that individual query operators had a performance gain equal to or lower than the bandwidth ratio. The key reason for the performance gain being higher than the bandwidth ratio is the better latency hiding capability of GPUs. To get a better sense for the runtime difference, in the next subsection we discuss models for the full SQL queries and dive into why architecture differences leads to significant difference in performance gain from the bandwidth ratio.

3.7.3 Case Study

The queries in the Star Schema Benchmark can be broken into two sets: 1) the query flight q1.x consists of queries with selections directly on the fact table with no joins and 2) the query flights q2.x, q3.x, q4.x consist of queries with no selections on fact table and multiple joins — some of which are selective. In this section, we analyze the behavior q2.1 in detail as a case study. Due to space constraints, we do not model all queries, instead we focus two queries q1.1 and q2.1 (one from each bucket). For each query, we build a model assuming the query is memory-bandwidth bound, derive the expected runtime based on the model, compare them against the observed runtime, and explain the differences observed.

Q1.1: Figure 3-1 shows the query. The query filters the lineorder tables based on 3 columns. The selectivity of predicates on lo_orderdate, lo_discount, and lo_quantity are 1/5, 3/11, and 1/2 respectively. Figure 3-16a shows the query plan on the CPU and Figure 3-16b shows the implementation on GPU using block-wide



Figure 3-16: Star Schema Benchmark Q1.1 Execution Plan

functions. The total number of cache lines accessed is approximately:

$$n = \frac{4|L|}{C} + \min(\frac{4|L|}{C}, |L|\sigma_1) + \min(\frac{4|L|}{C}, |L|\sigma_1\sigma_2) + \min(\frac{4|L|}{C}, |L|\sigma_1\sigma_2\sigma_3)$$

|L| is the cardinality of the lineorder table, C is the cache line size and $\sigma_1, \sigma_2, \sigma_3$ are selectivity of the 3 predicates. The four terms represent the number of cache lines accessed from the four columns: lo_orderdate, lo_discount, lo_quantity and lo_extendedprice respectively. For each column except the first, the number of cache lines accessed is the minimum of: 1) accessing all cache lines of the column $(\frac{4|L|}{C})$ and 2) accessing a cache line per entry read $(|L|\sigma)$. Given n, the query runtime is $n * C/B_r$ where B_r is the memory read bandwidth of the respective device. Plugging in the values we get the expected runtime on CPU and GPU as 26.7 ms and 1.98 ms compared to the actual runtime of 40.2 ms and 2 ms. The GPU runtime observed is close to the runtime expected based on the model. There is significant difference between the observed and expected runtimes on the CPU. The key reason for it is that the 2nd and 3rd selection result in more irregular memory accesses with prefetchers

```
SELECT SUM(lo_revenue) AS revenue, d_year, p_brand
FROM lineorder, date, part, supplier
WHERE lo_orderdate = d_datekey
AND lo_partkey = p_partkey AND lo_suppkey = s_suppkey
AND p_category = 'MFGR#12' AND s_region = 'AMERICA'
GROUP BY d_year, p_brand
```

Figure 3-17: Star Schema Benchmark Q2.1



Figure 3-18: Star Schema Benchmark Q2.1 Execution Plan

are less able to effectively predict and hence result in pipeline stalls. The performance gain observed is $20\times$, only slightly higher than that bandwidth ratio of GPU over CPU.

Q2.1: Figure 3-17 shows the query: it joins the fact table lineorder with 3 dimension tables: supplier, part, and date. The selectivity of predicates on p_category and s_region are 1/25 and 1/5 respectively. The subsequent join of part and supplier have the same selectivity. We choose a query plan where lineorder first joins supplier, then part, and finally date, this plan delivers the highest performance among the several promising plans that we have evaluated. Figure 3-18 shows the query plan.

The cardinalities of the tables lineorder, supplier, part, and date are 120M, 40k, 1M, and 2.5k respectively. The query runs build phase for each of the 3 joins to build their respective hash tables. Then a final probe phase runs the joins pipelined. Given the small size of the dimension tables, the build time is much smaller than the

probe time, hence we focus on modeling the probe time. On the GPU, each thread block processes a partition of the fact table, doing each of the 3 joins sequentially and updating a global hash table at the end that maintains the aggregate. Past work [51] has shown that L2 cache on the GPU is an LRU set associative cache. Since hash tables associated with the **supplier** and **date** table are small, we can assume that they remain in the L2 cache. The size of the **part** hash table is larger than L2 cache. We model the runtime as consisting of 3 components:

1) The time taken to access the columns of the fact table:

$$r_1 = \left(\frac{4|L|}{C} + \min(\frac{4|L|}{C}, |L|\sigma_1) + \min(\frac{4|L|}{C}, |L|\sigma_1\sigma_2) + \min(\frac{4|L|}{C}, |L|\sigma_1\sigma_2)\right) \times \frac{C}{B_r}$$

where σ_1 and σ_2 are join selectivities associated with join with supplier and part tables respectively, |L| is the cardinality of the lineorder table, C is size of cache line, and B_r is the global memory read bandwidth. For each column except the first, the number of cache lines accessed is the minimum of: 1) accessing all cache lines of the column $\left(\frac{4|L|}{C}\right)$ and 2) accessing a cache line per entry read $(|L|\sigma)$.

2) Time taken to probe the join hash tables:

$$r_2 = (2 \times |S| + 2 \times |D| + (1 - \pi)(|L|\sigma_1)) \times \frac{C}{B_r}$$

where |S| and |D| are cardinalities of the supplier and date table, $(|L|\sigma_1)$ represents the number of lookups into the parthash table and π is the probability of finding the part hash table lookup in the L2 cache.

3) Time taken to read and write to the result table:

$$r_3 = |L|\sigma_1\sigma_2 \times \frac{C}{B_r} + |L|\sigma_1\sigma_2 \times \frac{C}{B_w}$$

The total runtime on GPU is $r_1 + r_2 + r_3$. The key difference with respect to CPU is that on the CPU, all three hash tables fit in the L3 cache. Hence for CPU, we would have $r_2 = (2 \times |S| + 2 \times |D| + 2 \times |P|)$. To calculate π , we observe that

the size of the part hash table (with perfect hashing) is $2 \times 4 \times 1M = 8MB$. With the supplier and date table in cache, the available cache space is 5.7MB. Hence the probability of part lookup in L2 cache is $\pi = 5.7/8$. Plugging in the values we get the expected runtimes on the CPU and GPU as 47 ms and 3.7 ms respectively compared to actual runtime of 125 ms and 3.86 ms.

We see that the model predicted runtime on the GPU is close to the actual runtime whereas on the CPU, the actual runtime is higher than the modeled runtime. This is in large part because of the ability of GPUs to hide memory latency even with irregular accesses. SIMT GPUs run scalar code, but they "tie" all the threads in a warp to execute the same instruction in a cycle. For instance, gathers and scatter are written as scalar loads and stores to non-contiguous locations. In a way, CPU threads are similar to GPU warps and GPU threads are similar to SIMD lanes. A key difference between SIMT model on GPU vs SIMD model on CPU is what happens on memory access. On the CPU, if a thread makes a memory access, the thread waits for the memory fetch to return. If the cache line being fetched is not in cache, it leads to a memory stall. CPU have prefetchers to remedy this, but prefetchers do not work well with irregular access patterns like join probes. On the GPU, a single streaming multiprocessor (SM) usually has 64 cores that can execute 2 warps (64 threads) at any point. However, the SM can keep > 2 warps active at a time. On Nvidia V100, each SM can hold 64 warps in total with 2 executing at any point in time. Any time a warp makes a memory request, the warp is swapped out from execution into the active pool and another warp that is ready to execute ends up executing. Once the memory fetch returns, the earlier warp can resume executing at the next available executor cores. This is similar to swapping of threads on disk access on CPU. This key feature allows GPUs to avoid the memory stalls associated with irregular accesses as long as enough other threads are ready to execute. Modeling query performance of multi-join queries on CPUs is an interesting open problem which we plan to address as future work.

	Purchase Cost	Renting Cost
CPU	\$2-5K	0.504 per hour
GPU	PU + 8.5K	3.06 per hour

Table 3.3: Purchase and renting cost of CPU and GPU instance

3.7.4 Cost Comparison

The chapter has so far demonstrated that GPUs can have superior performance than CPUs for data analytics. However, GPUs are known to be more expensive than CPUs in terms of cost. Table 3.3 shows both the purchase and renting cost of CPU and GPU that match the hardware used in this chapter (i.e., Table 3.2). For renting costs, we use the cost of EC2 instances provided by Amazon Web Services (AWS). For CPU, we choose the instance type r5.2xlarge which contains a modern Skylake CPU with 8 cores, with a cost of \$0.504 per hour. For GPU, we choose the instance type p3.2xlarge whose specs are similar to r5.2xlarge plus it has an Nvidia V100 GPU, with a cost of \$3.06 per hour. The cost ratio of the two systems is about $6 \times$. For purchase costs, we compare the estimate of a single socket server blade to the same server blade with one Nvidia V100 GPU. The cost ratio of the two systems at the high end is less than $6\times$. The average performance gap, however, is about $25 \times$ according to our evaluation (cf. Section 3.7.2), which leads to a factor of 4 improvement in cost effectiveness of GPU over CPU. Although the performance and cost will vary a lot across different CPU and GPU technologies, the ratio between the two will not change as much. Therefore, we believe the analysis above should largely apply to other hardware selection.

3.8 Conclusion

This chapter compared CPUs and GPUs on database analytics workloads. We demonstrated that running an entire SQL query on a GPU delivers better performance than using the GPU as an accelerator. To ease implementation of high-performance SQL queries on GPUs, we developed Crystal, a library supporting a tile-based execution model. Our analysis on SSB, a popular analytics benchmark, shows that modern GPUs are $25 \times$ faster and $4 \times$ more cost effective than CPUs.

Part II

Novel GPU Query Operators
Chapter 4

Top-K

4.1 Introduction

A common type of analytical SQL query involves running a top-k, i.e., finding the highest (or lowest) k of n tuples given a ranking function. Examples of top-k queries include asking for the most expensive products on an e-commerce site, the best-rated restaurants in a review site, or the worst performing queries in a query log. Top-k is a well studied problem in computer science in general and data management in particular since top-k calculation (order-by/limit clauses) is supported by virtually every data analytics system. There are many instances of the problem and a diversity of efficient solutions (see [42] for a survey).

A naïve method for finding the top-k elements is to sort them and return the first k. However, sorting does more work than necessary, as there is no need to sort the elements beyond the top-k. A better approach is to maintain a priority-queue (a.k.a. max-heap) of size k and inserting greater elements while removing lesser ones. The runtime of this approach is in the order of $n \log(k)$. This algorithm can be parallelized across m processors by logically partitioning the data, having each processor compute a per-partition top-k and computing the global top-k from the m per-partition heaps. While this method can be efficiently implemented on multicore processors (see Section 4.5.7), it is not suited to the Single-Instruction-Multiple-



Figure 4-1: The Duality of Top-K and Sorting

Threads execution model of massively parallel systems¹. With the recent interest in GPU-based query processing [55, 83, 39, 60, 13, 49], there is an obvious need for a efficient, massively parallel algorithm to solve the top-k problem. In fact, we found that two of the most mainstream GPU programming frameworks (Tensorflow and Arrayfire) [5, 3] have open feature requests to add a top-k operator.

One way to develop an intuition for the existence and even the characteristics of a solution to this problem is to consider the duality of top-k and sorting algorithms. We illustrate this duality in Figure 4-1: the corresponding sort algorithm to priority queues is heapsort. In fact, one may view heapsort as the construction of a priority queue with k = n and the subsequent extraction of the elements in sorted order. This, of course, hides many implementations details but helps to form an intuition. When thinking about sorting and top-k in the context of massively parallel architectures, one finds that the textbook massively parallel sorting algorithm is bitonic sorting. Yet, there is no known corresponding top-k algorithm to bitonic-sort. We can, however, hypothesize that, like bitonic sort, it is likely to be based on bitonic merges and needs to incorporate a number of low-level optimizations to make it compute- as well as bandwidth-efficient.

In this work, we systematically develop this intuition into a working algorithm by extensively studying existing top-k solutions on GPUs and developing a novel solution

¹the unpredictable execution flow leads to high branch divergence overhead

targeted towards massively parallel architectures. We found that it is in fact based on bitonic merges and called it *bitonic top-k*. We investigate the characteristics of a number of other potential top-k algorithms for GPUs, including sorting and heapbased algorithms, as well as radix-based algorithms that use the high-order bits to find the top items. In the end, we find that bitonic top-k is up to 4 times faster than other top-k approaches and upto 15x faster than sorting for k up to 256.

These new algorithms have the potential to directly impact the performance of modern GPU-based database systems: all of the systems we are aware of (PG Strom, Ocelot, and MapD) currently use sort or transfer the entire dataset to the CPU for top-k calculation. They could, thus, directly obtain the benefits of our approach by integrating our algorithm. While we do not explicitly study means of mitigating the PCI-E bottleneck, having an efficient GPU-based top-k operator will allow these system to transfer less data through the PCI-E bus and, thus, achieve higher performance.

In summary, we make the following contributions:

- We study the performance characteristics of a variety of different top-k algorithms on a variety benchmarks, varying the data-set size, the value of k, the type of data (ints vs floats), and the initial distribution of data.
- We develop a novel, massively parallel, algorithm for the efficient evaluation of top-k queries.
- We devise a number of optimizations (in part based on known techniques, in part entirely novel) and show that our new bitonic top-k algorithm generally outperforms all other algorithms, often by a factor of 4x or more, for values of k up to 256. Furthermore, we demonstrate its robustness against skewed input data distributions.
- We demonstrate that the algorithm is able to be integrated into existing systems (specifically, MapD.)
- Finally, we develop detailed cost models for our bitonic top-k as well as other

algorithms, and show that these cost models can accurately predict runtimes, which is valuable when a query planner needs to choose a top-k implementation.

Before describing the details of our algorithms, optimizations, and experiments, we begin with a discussion of existing sorting and top-k algorithms for GPUs.

4.2 Background

4.2.1 Sorting on the GPU

Many sorting algorithms have been proposed over the years. The early implementations were often based on bitonic sort [18, 58, 34]. Later, radix-based sort algorithms were proposed which perform better than bitonic sort [67, 54, 76].

Bitonic Sort Bitonic sort is based on bitonic sequences, i.e. concatenations of two subsequences sorted in opposite directions. Given a bitonic sequence S with length $l = 2^r$, S can be sorted ascending (or descending) in r steps. In the first step the pairs of elements (S[0], S[l/2]), (S[1], S[l/2+1]), ..., (S[l/2-1], S[l-1]) are compared and exchanged if the second element is smaller than the first element. This results in two bitonic sequences, (S[0], ...S[l/2-1]) and (S[l/2], ...S[l-1]) where all the elements in the first subsequence are smaller than any element in the second subsequence. In the second step, the same procedure is applied to both the subsequences, resulting in four bitonic sequences. All elements in the first subsequence are smaller than any element in the second, all elements in the third subsequence are smaller than any element in the third and all elements in the third subsequence are smaller than any element in the fourth subsequence. The third, fourth, ..., r-th step follow similarly. Processing the r-th step results in 2^r subsequences of length 1, thus the sequence Sis sorted.

Let A be the input array to sort and let $n = 2^k$ be the length of A. The process of sorting A consists of k phases. The subsequences of length 2 (A[0], A[1]), (A[2], A[3]), ..., (A[n-2], A[n-1]) are bitonic sequences by definition. In the first phase these subsequences are sorted (as described above) alternating ascending and



Figure 4-2: Bitonic Sorting Network

descending. This creates bitonic subsequences of length 4, (A[0], A[1], A[2], A[3]), ..., (A[n-4], A[n-3], A[n-2], A[n-1]). In the second phase these subsequences of length 4 are sorted alternating ascending and descending, resulting in subsequences of length 8 being bitonic sequences. In the i-th phase of bitonic sort the total number of subsequences being sorted is 2^{k-i} and the length of each of these subsequences is 2^i , thus the i-th phase consists of i steps. After the (k-1)-th phase the array A is a bitonic sequence. A is sorted in the last phase k.

In every step n/2 compare/exchange operations are processed. There are logn phases, with the *i*-th phase having *i* steps. Thus, the number of comparisons is $O(nlog^2n)$. Hence, bitonic sort is slower than other O(nlogn) sort algorithms on a serial CPU. The advantage of bitonic sort is that it can be easily parallelized on SIMT and SIMD architectures and requires less inter-process communication. Figure 4-2a shows the bitonic sorting network for an arbitrary sequence of size 8. There $log_2 8 = 3$ phases, where phase *i* has *i* steps. Every step consists of 8/2 = 4 comparisons.

Sorting in Figure 4-2a(a) follows the process described above: in phase 1, elements 0 and 1 are compared and sorted in asscending order; elements 2 and 3 are sorted descending; elements 4 and 5 ascending, and so on. Each of these comparisons can be done in parallel on separate threads. At the end phase 1, there are 4 sorted sequences of length 2. In phase 2, with step size 2, first elements 0 and 2 and 1 and 3 are compared and sorted descending, while 4 and 6 and 5 and 7 are sorted ascending. These comparisons can also be done in parallel. Then, phase 2 with step size 1 is executed, such that elements 0 and 1 and 2 and 3 are sorted descending, and 4 and 5 and 6 and 7 are sorted ascending. Again these comparisons are parallelized. At the end of phase 2, we are left with two length 4 sorted lists. Finally, phase 3 merges these two lists using decreasing step sizes from 3 to 1.

The fastest implementation of bitonic sort is the one proposed by Peters et al.[58]. The bitonic top-k algorithm discussed later re-uses some of the ideas from their paper.

Radix Sort Radix sorting is based on the reinterpretation of a k-bit key as a sequence of d-bit digits, which are considered one a time. The basic idea is, that splitting the k-bit digits into smaller d-bit digits results in a small enough radix $r = 2^d$, such that keys can be partitioned into r distinct buckets. As sorting of each digit can be done with an effort that is linear in the number of keys n, the whole sorting process has a time complexity of $O(\lceil k/d \rceil n)$. Iterating over the keys' digits can be performed from the most-significant to the least significant digit (MSD radix sort [76]) or vice versa (LSD radix sort [67, 54]).

In either case, the first step is to compute a histogram of the input values in a sequential scan. As the histogram reflects the number of keys that shall be put into each of the r buckets, computing the exclusive prefix-sum over these counts yields the memory offsets for each of the buckets. Finally, the keys are scattered into the buckets according to their digit value. Recursively repeating these steps on subsequent digits for the resulting buckets ultimately yields the sorted sequence. The best performing sort algorithm today is based on MSD radix sort [76].

4.2.2 K-Selection

The k-selection problem asks one to find the k-th largest value in a list of n elements. Having a solution to the k-selection problem, one can easily find the top-k elements by possibly making one additional pass over the data. Alabi et.al [15] studied this problem extensively. Apart from the sort and choose the k-th element, they studied two other algorithms: Radix Select and Bucket Select.

Radix Select: Radix select follows from the MSD radix sort algorithm. Like the MSD radix sort, it operates as a sequence of steps, each of which processes a d-bit digit. It performs the same histogram and prefix sum steps. However, instead of writing out all the entries partitioned into buckets, radix select uses the histogram to find the bucket B containing the k^{th} -largest entry. It then writes out only the entries of B and continues to examine the next d-bit digit of the elements in *only* the matched bucket.

Bucket Select: Instead of creating the buckets based on radix bits, bucket select tries to be more robust by computing the buckets based on the min-max values. The algorithm makes an explicit first pass over the dataset to calculate the min and max values. Subsequently we execute a series of passes. Each pass is three step: create multiple buckets equally spaced out between min and max and, compute the number of entries in each bucket per thread. Second, do a prefix sum and find the bucket with the k^{th} largest element. Finally, read the input and write out elements of the matched bucket. We run the next pass on the entries of the matched bucket.

4.3 Algorithms

Based on the discussion so far, we have 3 algorithms to find top-k:

- Sort and Choose: Use radix sort to sort the entire vector and select the top-k elements from it.
- Using Radix Select: We can use the radix-based selection algorithm to get the k^{th} largest element and use that to find the top-k by making one additional pass

Algorithm	1:	Per-Thread	Top-K
-----------	----	------------	-------

over the input array.

• Using Bucket Select: We can do the same as above, this time using Bucket Select instead of Radix Select.

In this section, we describe two new algorithms for finding the top-k elements. In the first algorithm, each thread independently maintains the top-k elements it has seen so-far and finds the global top-k amongst the local (per thread) top-ks. Second, we present the bitonic top-k algorithm which is based on bitonic sorting. For ease of presentation, our description assumes tuples consisting only of a key. Of course, real applications may need to perform top-k on other settings, including (key,value) pairs, multiple keys, and different data types and distributions; our evaluation shows that our algorithms cover all of these cases (Section 4.5).

4.3.1 Per-Thread Top-K

A single-threaded version of top-k would maintain the top-k elements in a min-heap and update it for every new element seen. The natural way to parallelize is to partition the input, calculate the top-k per partition and calculate the global top-k from those as a final reduction step. Algorithm 1 shows the pseudocode that would run in parallel in each thread (nt threads are run in parallel). We use a min-heap per thread to maintain the top-k elements seen by that thread so far. After initializing the heap, we iterate over the elements starting from t in steps of number of threads. This (coalesced) memory access pattern has been shown to benefit memory access on the GPUs [35]. We check if the current element is larger than the minimum value among the top-k seen. If so, we pop the minimum and add the current element. Finally, we write out the top-k values to O in a coalesced manner. This approach is efficient in terms of memory usage. It makes one full read pass over the global memory and writes significantly less data. However, it suffers from thread divergence and occupancy issues, discussed in greater detail in Section 4.4.1.

4.3.2 Bitonic Top-K

While a full bitonic sort is a solution to the top-k problem, it performs a significant amount of unnecessary work in sorting the entire input, just as heap sort is much less efficient than using a priority queue to select the top-k.

In bitonic sort, we start from an unsorted array which is equivalent to sorted sequences of length 1 and construct longer sorted sequences of length 2,4, ... up to n, at which point the entire list is sorted. Our basic approach is to develop an algorithm that performs as little unnecessary work as possible but maintains the massively parallel nature as bitonic sort. To achieve this, we decompose the complex bitonic sort operation into a series of parallel steps with different comparison distances. We carefully reassemble the steps into three operators that, in combination, allow the efficient fully parallel calculation of the top-k elements of a vector. These operators are *local (bitonic) sort, merge* and *rebuild*.

In *local sort*, we generate sorted sequences of size k using (partial) bitonic sort. In the *merge*, we bitonically merge two sorted sequences of size k, thus creating two bitonic sequences, where the first sequence contains the k greatest (w.l.o.g) and the second sequence contains the k least elements. In *rebuild* we sort the sequence containing the greatest (w.l.o.g.) elements; the second sequence containing the smaller k elements is discarded. While sorting, we exploit the fact that the output of the second operator already satisfies the bitonic property. At this point, we have effectively halved the problem size. We recursively apply the *merge* and *rebuild* operators to the Algorithm 2: Bitonic Top-K Local Sort

Input : List L of length n**Output:** L with sorted sequences of length k1 int t = getGlobalThreadId();2 for $len \leftarrow 1$; len < k; $len \leftarrow len \ll 1$ do dir \leftarrow len \ll 1; 3 for $inc \leftarrow len; inc > 0; inc \leftarrow inc \gg 1$ do $\mathbf{4}$ int low \leftarrow t & (inc - 1); 5 int $i \leftarrow (t \ll 1) - low;$ 6 bool reverse \leftarrow ((dir & i) == 0); 7 $x0, x1 \leftarrow L[i], L[i + inc];$ 8 bool swap \leftarrow reverse \oplus (x0 < x1); 9 if swap: $x0, x1 \leftarrow x1, x0;$ $\mathbf{10}$ $L[i], L[i + inc] \leftarrow x0, x1;$ 11

(halved) sequence until we are left with only k elements which form the top-k. The resulting algorithm performs no unnecessary work and has the massive parallelism of bitonic sort. In the rest of this section, we describe the individual operators in more detail.

(1) Local Sort The goal of this operator is to generate sorted runs of length k alternating between ascending and descending, starting from an unsorted array. Algorithm 2 shows the pseudocode. The unsorted sequence is equivalent to sorted sequence of length len = 1. Starting from len = 1, we generate sorted sequences of length $len = 2, 4 \dots k$. When len = k, we are done. This is the outer loop on line 3. When len = x, two neighboring sorted sequences of length x form a bitonic sequence of length 2x and can be sorted in log(x) + 1 steps. This is handled by the inner loop on line 4. In the first step, when inc = len, we compare pairs of elements $(L[0], L[len]), (L[1], L[1 + len]), \dots (L[len - 1], L[2len - 1])$. This is done in parallel, and each thread compares one pair of elements. In general, thread t compares element L[i] to L[i + len] where the index i is calculated as a function of t and inc as shown in lines 5 - 6. The elements are compared and exchanged (12-13) (if needed) and written back to the original array (14-15). The direction of exchange is determined by len. When len = x, we want to generate alternating ascending descending sorted



sequences of length 2x, i.e.: the direction changes every dir = 2 * len elements (Line 3). The actual direction of comparison is determined by whether (i/dir) is odd or even (Line 7). *Phase 1* in Figure 4-4 illustrates the accesses of *Local Sort* operator to generate find the top-4 of 16 elements.

(2) Merge At the end of the *local sort*, we have alternating ascending descending sorted (i.e., bitonic) sequences of length k. We compare neighboring sequences pairwise and select the larger element in each pair. While we do not know how many elements of each of the sequences are selected, we know that the top-k elements were selected and that they form a bitonic sequence. This is *the key insight of our work*. To illustrate it, consider Figure 4-3 which illustrates the calculation of a top-8: in Figure 4-3b (after the merge step), all elements on the left are amongst the top-8 because they are greater than their comparison partner which implies that they are greater than all elements to the left (or right, respectively) of their comparison partner (due to the bitonic property). This step halves the top-k candidate set. Algorithm 3 shows the pseudocode.

Algorithm 4: Bitonic Top-K Rebuild

Input : List L with bitonic sequences of length k**Output:** L with sorted sequences of length k1 int t = getGlobalThreadId();2 int len $\leftarrow k \gg 1$; **3** int dir \leftarrow len \ll 1; 4 for $inc \leftarrow len; inc > 0; inc \leftarrow inc \gg 1$ do int low \leftarrow t & (inc - 1); $\mathbf{5}$ int $i \leftarrow (t \ll 1) - low;$ 6 bool reverse \leftarrow ((dir & i) == 0); 7 $x0, x1 \leftarrow L[i], L[i + inc];$ 8 bool swap \leftarrow reverse \oplus (x0 < x1); 9 10 if swap: $x0, x1 \leftarrow x1, x0;$ $L[i], L[i + inc] \leftarrow x0, x1;$ 11

(3) Rebuild The input to rebuild is a list L with bitonic sequences of length k instead of an unsorted sequence in the local sort operator. As a result, we can generate sorted sequences of length k in log(k) steps by applying the inner loop of the local sort starting with len = k/2. For completeness, Algorithm 4 shows the pseudocode. The flow is the same as in local sort. A combination of merge and rebuild reduces a list of length n with sorted sequences of length k to a list of length n/2 with sorted sequences of length k. Merge and rebuild are repeated till we have a list of length k.

Analysis In *local sort*, every step does n/2 comparisons. There are *logk* outer loop iterations, with the *i*-th one having *i* steps. In *merge*, we do n/2 comparisons. In *rebuild*, we have *logk* steps of n/2 comparisons. Each time *merge* runs, the list size halves. *Merge* and *rebuild* run multiple times till we get a list of size k. The total number of comparisons are $O(nlog^2k)$. The runtime of bitonic top-k, like that of the bitonic sorting network is independent of the data distribution and depends only on |L| and k.



(b) Visualisation (gray: inactive, orange: candidates)

Figure 4-4: Bitonic Top-K (K=4)

4.4 Optimization & Implementation

In this section, we describe a number of optimizations – at both logical and implementation levels – that we applied to the different methods to optimize performance. All the performance numbers in this section are from running algorithms on a dataset of 2^{29} floating point values generated from a uniform distribution U(0,1) on a Nvidia Titan X Maxwell GPU (see Section 4.5.1 for details about the hardware setup). Numbers on more diverse data are given in Section 4.5.

4.4.1 Per-Thread Top-K

To implement the per-thread top-k algorithm (Algorithm 1) efficiently, we use shared memory to store the heap. Each thread block allocates an array of size k * wg in shared memory where wg is size of the thread block. Each thread maintains its own heap in shared memory using an array of size k. In order to avoid bank conflicts, we store the array striped, where thread t uses entries sdata[t + wg*i] where sdata is the shared memory array used by the thread block, and, i varies from 0...k. Since wg is always a multiple of 32, each thread's array maps to one shared memory bank and multiple threads in a warp updating their respective arrays does not cause shared memory bank conflicts.

The implementations suffer from two problems:

Thread Divergence: Heap updates are data dependent. On the GPU, a warp (32 threads) runs in a SIMT model. As a result, even if one thread wants to update its entries, all the other threads in the warp have to follow the same instruction path, leading to slowdown.

Occupancy: The shared memory used per thread increases with k. As the shared memory used by a block increases, the number of concurrent warps that can be run (occupancy) reduces. Beyond a point, the occupancy reduction leads to the GPU not having enough active warps to saturate the global memory bandwidth. For $k \geq 512$, even using the minimum thread block size of 32, we would need 64KB of shared memory, which is greater than 48KB available per thread block on our GPU.

We also implemented the per-thread top-k algorithm using registers and found its performance to be inferior. Appendix A contains a more detailed discussion and performance comparison of the register-based version.

4.4.2 Selection-based Top-K

The radix select and bucket select implementations used come from the GGKS package [14]. We revised the implementation of radix select to use 8-bit digits (based on MSD radix sort [76]) instead of 4-bit digits in the original code. This results in 4 passes for 32 bit (int and float) keys. Each pass can reduce the data size. However, if after the prefix sum we see no data reduction, the clustering step is skipped and we simply re-use the input in the next pass. Bucket select also divides the data into 16 buckets at a time and selects one bucket containing the k-th element. The interested reader can refer to [15] for more implementation details. The radix select implementation would write out the entire input array after each pass and then update the array pointer to point to the bucket containing k^{th} element. We fixed this inefficiency to only write out the right bucket.

Given the k^{th} highest element X, we can make an additional pass over the data to find the top-k elements. However this is not necessary. Once we select the bucket containing X, when scanning the array the second time to write out the tuples that fall into the bucket, we can also write out the elements in the higher buckets to a separate *result* array. In the last pass, we copy over all the elements in the identified bucket with value less than X to *result* and pad *result* with X to make it of size k. This eliminates the last pass we previously had to find the top-k elements given X.

4.4.3 Optimizing Bitonic Top-K

In this section we discuss a number of optimizations we devised to achieve close-tooptimal performance with our new bitonic top-k algorithm. While some of these are inspired by similar optimizations for other algorithms, to the best of our knowledge, none of them are applied in the context of top-k calculation. However, since our optimizations may be applicable to other problems, we include a paragraph on novelty and applicability in the description of each optimization. To give an impression of the importance of each optimization, we end every section with a graph indicating the the effect of optimization on the runtime for the case of finding top-32 elements in the dataset described at the start of this section.

Operating in Shared Memory The first optimization can be applied to each of the three operators individually: instead of reading/writing data after each massively parallel step to global memory, we do it once per operation. The data required is loaded into shared memory at the beginning of the operation. All the operation's intermediate steps happen in shared memory. At the end, the result is written back to global memory. For example, the *local sort* operation in Figure 4-4a has 3 intermediate steps. With this optimization, each threadblock would read the required data to shared memory, run the 3 steps within shared memory and then write back results at the end. Recall that the shared memory is an order of magnitude faster than global memory. This optimization shifts global memory reads/writes to shared memory reads/writes, thereby improving performance.



This results in a significant performance improvement from 521ms to 122ms. The *local sort* operator becomes shared memory bound while the *merge* and *rebuild* are still global memory bound.

Note that this optimization is contingent on k being less than or equal to $2*\max$ thread block size (= 2048 on modern GPUs). It also cannot be applied to all steps of a general bitonic sort algorithm with steps with *inc* up to n/2, because this would require loading the entire array into shared memory, but this is not a limitation in our bitonic top-k alogorithm as long as k is small enough. This optimization has been applied to bitonic sort to minimize accesses to the global memory [58].



Merging Operators As discussed in Section 4.3.2, our bitonic top-k algorithm can be broken into three operations: (1) *local sort* to create sorted sequences of length k, (2) *merge* two sorted sequences of length k to create a bitonic sequence of length k and (3) *rebuild* a bitonic sequence of length k after a merge. While the *local sort* operation is only executed once in the beginning, the *merge* and *rebuild* phase are alternated until the result is found.

The naive implementation would run a kernel per operator. However, there are no cross thread block dependencies across each of the kernels. This leads to a significant optimization potential: multiple operators can be fused into a single kernel and shared memory can be used to communicate results between operators. In addition to reducing kernel invocation overhead, this optimization eliminates global memory traffic due to intermediate results.

Each *merge* halves the number of elements. In order to ensure that each thread in the last operation in the fused kernel has work to do, we need to ensure that number of data items per thread is atleast 2^x where x is the number of *merge* phases in the fused kernel. We found the optimal number of processed of data items per thread to be 8. Beyond that, doubling the number of elements per thread doubles the number of shared memory bank conflicts and yields no performance improvement. Since each *merge* halves the number of elements per thread, processing 8 elements per thread allows us to have three (i.e., ld(8)) *merge* phases per kernel. This leads to two separate kernels: the first performing *local sort* followed by two *merge-rebuild* operators and a single *merge* (SortReducer). The second kernel performs three *rebuild-merge* operator sequences (**BitonicReducer**). To the best of our knowledge, this is a novel optimization.

This optimization reduces the runtime of top-32 from 122ms to 48.15ms. Both kernels (and as a result the entire application) are now shared memory bandwidth



Figure 4-5: Combining Multiple Steps

bound. The SortReducer kernel and BitonicReducer kernel achieve shared memory bandwidth of 2.75TBps and 2.7TBps respectively. This is greater than 90% utilization of the 2.9TBps peak bandwidth of shared memory observed on repeated read workload. We, therefore, shift our attention towards optimizing shared memory accesses.

Combining/Sequentializing Multiple Steps For the next optimization, we rearrange the assignment of data items to threads to reduce the amount of memory traffic. Figure 4-5a shows the default assignment (threads are color-coded), each thread reads two values from shared memory, compares them and writes them back to shared memory. As each thread is responsible for 8 elements, it does the same for 3 other pairs. If, however, we process more than two values per thread per round, the read and write operations can be shared. In Figure 4-5b, e.g., the orange thread reads the values at positions 0, 2, 4 and 6 and performs two comparisons on each. This halves the shared memory traffic and can be generalized to more elements (see Figure 4-5c). While this (partially) serializes the processing (from three fully parallel steps with four operations each to 12 sequential operations) it does not increase the overall number of comparisons. This optimization is similar to optimization 1 which combines multiple steps that read and write to global memory to read and write to shared memory. Instead here, we combine multiple steps that read and write to shared memory to work in registers. This reduces the runtime to 33.7ms.



Do work before writing Conventional wisdom is to copy a chunk of data from global to shared memory in a coalesced manner and processing data only in shared memory. However, by rejecting this common wisdom, we can reduce shared memory accesses. Each thread loads 8 consecutive elements from global memory into registers, perform all intermediate steps required to create local sorted sequence of length 8 without hitting shared memory and then write to shared memory. Note that as a result of this optimization, accesses to global memory are no longer coalesced because threads access data elements at a stride of 8. However, this does not lead to any noticeable performance difference on modern GPUs due to their data caches. This optimization is likely to be widely applicable. In our experiments, it yields an effective reduction of the runtime to 27.1ms.



Breaking Conflicts with Padding In this and the following section, we introduce three optimizations that help avoiding memory bank conflicts. While most current



Figure 4-6: Avoiding shared memory bank conflicts with padding

GPUs have 32 shared memory banks and warps of 32 threads, illustrating the effects of our optimizations on 32 memory banks would unnecessarily inflate the size of our figures. For that reason, we assume 8 memory banks (and warps of size 8) for the illustrations (note that the experiments are conducted on a real GPU with 32 memory banks).

The first optimization is an instance of a widely known technique: padding arrays to avoid memory conflicts. A shared memory array of size n can be viewed as a 2D array of dimensions $\left[\frac{n}{8}, 8\right]$ (where 8 is the number of banks). The key idea is to allocate slightly more memory to create a larger array of dimensions $\left[\frac{n}{8}, 9\right]$. The extra column added does not store any elements, however, it helps break shared memory conflicts. Figure 4-6 shows the accesses performed by a combined step combining *inc* = 2, 1 at time step 0 after padding. The grayed out cells do not hold any values and are simply space overhead. Each thread wants to read 4 contiguous elements. Thread 0 wants to read entries 0-3, thread 2 wants to read 8-12. Without padding these two threads would conflict (0 and 8 are in the same bank). The figure illustrates how the padding prevents the conflicts (thread 0 and 2, access different memory banks after padding). This decreases the runtime of top-32 to 22.3ms. Note that padding does not help bitonic sorting due to its global memory bandwidth boundness. In contrast, the bitonic top-k is shared memory bandwidth bound.

Padding also has a second benefit: it allows us to merge more operators into a kernel. Recall that processing more than 8 elements caused conflicts (discussed with



operator merging earlier in this section) and that this effect limited us to merge only three operators.

With padding, this is no longer true which allows us to merge four or even more steps (processing 16 or more elements per thread). However, beyond 16, the number of allocated registers forces the compiler to reduce occupancy leading to a performance penalty: Figure 4-7 shows the performance when varying the number of processed elements (B). There is virtually no benefit when increasing B from 16 to 32 and a detriment when increasing B to 64. We, thus, fixed B to 16.



Figure 4-7: Bitonic Top-K performance varying the number of elements per thread

Chunk Permutation Figure 4-8 illustrates the shared memory access pattern of the local sort operation after applying the optimizations discussed so far. Here, each outlined shape represents an operation with no accesses to shared memory (shared memory access is performed at the edges of each shape), the axes represent iterations of sequential loops within the kernel, and the numbers the distance in the input array of the compared elements. While most of the kernels are bank-conflict-free, we observe that, when the comparison distance is four, the memory accesses cause bank conflicts. To illustrate this, consider Figure 4-9a: it illustrates the comparisons



Figure 4-8: Comparison distance for local sort k = 8, x = 4



Figure 4-9: Shared memory bank conflicts when comparing elements

that are performed in the red box in Figure 4-8 (a pair-wise comparison of elements with a distance of four). The figure indicates the memory accesses of the threads in a warp: each arrow represents the comparison performed in one thread with the colors indicating the time (and thus the order) of the accesses. We observe that, despite padding, the memory accesses at clock time 0 overlap with respect to their memory bank. We can avoid this by changing the memory locations each thread reads from (and writes to). We call this optimization *Chunk Permutation* and illustrate it in Figure 4-9b: instead of reading from conflicting banks at clock 0, each thread accesses a different memory bank. One may notice that there is still overlap between the accessed values. However, these accesses are performed at different times. As is obvious the figure, there are no conflicts by observing that there are no two identically colored boxes in a column.

While we illustrated the chunk permutation optimization using the example of the

last step in the local sort, the problem occurs whenever the comparison distance of a combined step is greater than 1. This makes it widely applicable for our case and even more broadly (e.g., for bitonic sort). For our application it removes all the remaining memory bank conflicts in the *local sort* operator for all $k \leq 256$ and improves the performance of top-32 from 17.8ms to 16ms. The effect is more pronounced at higher k, e.g., improving top-128 performance by roughly 20 percent. This optimization is novel. The broader idea of re-arranging chunks to avoid bank conflicts could be applied to other algorithms that suffer from shared memory bank conflicts.

Reassigning Partitions The last optimization we developed is targets the assignment of data items to threads after the first reduction: since the reduction halves the number of elements but the number of threads remains the same, there is less work per thread. This leads to fewer steps being merged because the number of steps that can be merged is the logarithm of the number of input data items per thread. To maintain the same number of input data items per thread after the reduction, we have half the threads perform all the work. While this leaves half of the threads without work, the reduction in shared memory traffic due to larger combined steps outweighs that cost. This optimization further improves the performance to 15.4ms. This optimization is novel and maybe applicable to kernels that reduce input data in phases.

Discussion:

Memory Usage. Memory usage is of critical importance for GPU-based data management systems. For a dataset of size n, out-of-place bitonic top-k uses one additional buffer of size n/8. This is significantly less than sort and selection-based methods which require an additional buffer of size n.

Data larger than GPU memory. When data is larger than can fit in GPU memory, data needs to moved to the GPU via the PCI bus. There is a significant amount of research on reducing pressure on that bottleneck using asynchronous transfers [76, 34], approximation [59], compression [65, 28] and cost-aware device selection [23]. While we do not explicitly address the PCI-bottleneck in this chapter, the reductive nature of top-k queries makes it trivial to process the data in memory-size chunks and overlap computation with transfer (similar what is done for sorting [76]).

Bitonic Top-K on CPU. The bitonic top-k algorithm presented in the chapter can be adapted to run on the CPU as well. The bitonic top-k algorithm is reductive, it reduces an array of size n to an array of size k containing the top-k elements. To make use of all the cores available, we partition the input array into equal sized partitions and let each core independently process the partition to emit the top-k. The top-k elements emitted by the individual core are combined in a final global step to find the global top-k.

On each core, we further break down the input partition into vectors of fixed size (in the implementation we use 2048 elements as the vector size). We process the input partition in phases. The first phase does the function of the SortReducer. It reads in the unsorted input partition, one vector at a time and outputs $(1/16)^{th}$ of the input containing bitonic sequences of length k. The subsequent phases do the function of BitonicReducer. They read in the input containing bitonic sequences of length k, one vector at a time, and outputs $(1/16)^{th}$ of the input containing bitonic sequences of length k. Algorithm 5 shows the pseudocode.

On the GPU, each vector is processed in parallel by a thread block. Each thread of the thread block reads in 16 elements from shared memory and runs a combined step and outputs it back to shared memory. However, on the CPU, on each core, we process the vector in a single threaded fashion. The thread reads in 16 elements at a time from main memory and runs a combined step and outputs it back to shared memory. The reason we process small sized vectors (here of size 2048) is so that the data is cached in L1 cache. This allows random accesses in the vector to not incur latency of main memory read.

Modern CPUs also have support for Single Input Multiple Data(SIMD) instructions. The bitonic sorting network used to process a combined step can be implemented using SIMD instructions for improved performance. In the implementation we use 128-bit SSE-based implementation of [25]. Also, some of the optimizations details in Section 4.4.3 are not needed on the CPU. In particular, padding and chunk

```
Algorithm 5: CPU Bitonic Top-K Thread
```

```
Input : Input Parititon S of length n; int k
   Output: List O of the top-k elements per thread
1 int numElements \leftarrow n;
2 int numVectors \leftarrow numElements / vectorSize;
3 int temp[2][n/16];
4 int current \leftarrow 0;
5 for i \leftarrow 0; i < num Vectors; i \neq 1 do
       SortReducer(S, temp[current], i, k)
7 numElements \leftarrow numElements / 16;
s while numElements >= vectorSize do
9
       for i \leftarrow 0; i < num Vectors; i \neq 1 do
           BitonicReducer(temp[current], temp[1-current], i, k);
10
       numElements \leftarrow numElements / 16;
11
       numVectors \leftarrow numElements / vectorSize;
12
       current \leftarrow 1 - current;
\mathbf{13}
14 O \leftarrow sort(temp[current], numElements);
```

permutation are not useful on the CPU as there is no notion of bank conflict.

The bitonic top-k algorithm is not work-efficient. It does $O(n(logk)^2)$ number of comparisons as shown in Section 4.3.2. This is strictly worse than heap-based methods which do O(nlogk) number of comparisons. However, bitonic top-k can leverage SIMD instructions to improve performance. Overall, in the case when lots of heap insertions occur (e.g.: when the input data is sorted), the performance of bitonic top-k is close to that of heap-based methods despite the larger number of comparisons. Further, bitonic top-k could be better on platforms with wider vector instruction support like AVX-512 in Intel Knights Landing processors. We plan to explore this in the future.

4.4.4 Database Integration

Having developed a highly optimized massively parallel top-k implementation, we were naturally interested in its usability in a full system. As a proof of concept, we integrated the bitonic top-k kernel into MapD, an open source GPU database [55]. In this section, we discuss two optimization opportunities that can we used in the

context of database analytics to improve performance.

Fusing with filter A common query template is to find the top-k items in a subset of the data satisfying a selection predicate. The easy way to execute this is to have a seperate kernel execute the filter and have the subsequent top-k kernel use the output to find the top-k items. GPU-based databases end up doing this currently as they treat the top-k kernel(done using sort) as a blackbox. We can optimize this by fusing the select into the bitonic top-k routine.

Each thread block running the SortReducer kernel reads in 16nt elements and writes out nt elements where nt is the number of threads in the thread block. One way to fuse the kernels is to read in 16nt elements, apply the filter predicate and run the SortReducer on the matched elements. However, the SortReducer kernel is then effectively running on s * 16nt where s is the selectivity. As shown in previous section, having 16 elements per thread is crucial to the performance of SortReducer as it enables it to run combined steps. The FusedSortReducer instead uses the selection step as a buffer filler. It reads in nt elements at a time into shared memory, applies the filter predicate to find the number of matches elements, computes a prefix sum and then writes it out into a shared memory buffer of size 16nt. It then reads in the next batch of nt elements till we have more than 15nt elements matched. The rest of the entries are padded with min/max value so that they never show up in the top-k results. The SortReducer then works on the buffer of 16nt elements and writes out nt elements contain the top-k.

Custom Ranking Function A custom ranking function is an order by clause of the form $f(A_1, A_2, A_3...)$ where f is any function and $A_1, A_2, ...$ are columns of A. The ranking function can be evaluated at the start of SortReducer kernel instead of running it as a separate project step which outputs the value of the function.

4.5 Evaluation

In this section, we compare the performance of the five different algorithms we presented in Section 4.3:

- 1. Sort: Sorting to find top-k
- 2. PerThread TopK: Using a heap per thread to find top-k
- 3. Radix Select: Adapting radix select to find top-k
- 4. Bucket Select: Adapting bucket select to find top-k
- 5. Bitonic TopK: Using the bitonic top-k algorithm

after applying the optimizations in Section 4.4 varying the following parameters: 1. the value of K 2. the key data type 3. the data distribution 4. the data size 5. the number of key and value columns and finally 6. the device (CPU vs. GPU). After that, we show the performance achieved by integrating BitonicTopK in the MapD database by evaluating top-k queries on a twitter dataset.

4.5.1 Setup

All the results are averages of 3 runs on a single socket Intel i7-6900 @ 3.20GHz (Skylake with 8 Cores, 16 hardware threads) with Nvidia GTX Titan X Maxwell GPU running on Ubuntu 15.10 (Kernel 4.2.0-30) and CUDA 8.0.

4.5.2 Performance with Varying K

We generate 2^{29} random uniformly distributed (U(0, 1)) floats and observe the performance of the different algorithms with K varying from 1 to 1024 in powers of 2. Figure 4-10 shows the results.

Memory Bandwidth shows the time taken to read the entire data from global memory. Since all of the data needs to be read at-least once, this constitutes a lower bound on the runtime of any algorithm. In reality, most algorithms would write/read intermediate data and have other overheads. We observe that the runtime of the **Sort** method is virtually constant across k since it has to sort the entire input irrespective of K.



Figure 4-10: Time taken with different k (32-bit float values)

Radix Select and Bucket Select take almost the same time across K as expected. The latter does worse than the former due to the use of more expensive atomic operations. When k = 1, Bucket Select is fast as it terminates after finding the min-max of the array and directly returns it as the result.

PerThread TopK line has steep slope rising from k = 32, this is due to reduced occupancy and thread divergence as explained earlier in Section 4.4.1. The approach fails for K > 256 due to the required amount of shared memory. For K = 512, even with the minimum thread block size 32, we need 512 * 32 * 4 = 64KB (each key is 4 bytes) which exceeds the available 48KB per thread block.

Finally, Bitonic does better than all the other algorithms for $K \leq 256$. For K > 256, the Radix Select method does better.

4.5.3 Dependence on Data Type

Next, we run the algorithms on a dataset with 2^{29} unsigned integers drawn from $U(0, 2^{31} - 1)$ (see Figure 4-11). The time taken by all methods except Radix Select is virtually identical to that observed with *float* data type. Radix Select does better



Figure 4-11: Time taken with different k (32-bit integer values)

because with uniformly distributed data, the number of eliminated tuples per scan is maximal (a reduction of $256 \times$ assuming 8-bit radices).

Second, we run the algorithms on 2^{28} doubles drawn from U(0, 1). The size of the data is the same, however the word size of each key has increased. Figure 4-12 shows the results. The **Sort**-based approach has to perform twice as many scans (since the number of digits has doubled) but scan fewer values. However, processing 64-bit values is significantly more expensive than 32-bit values on most GPUs which explains the cost increase. **Radix Select** has the same issue, however, this effect is less pronounced as the algorithm operates on a smaller number of elements in subsequent passes. **Bucket Select** ends up being slightly faster than with floats as the number of keys has reduced resulting in smaller number of atomic operations. The **PerThread TopK** line is similar to line seen with *float* shifted to the left and slightly lower: this is natural since there less processing needs to be performed for every read byte. For each K, the method uses twice as much shared memory when processing doubles compared to processing floats. Thus, the approach fails earlier (for K > 128). Finally, **Bitonic TopK** remains largely unchanged as the data size is the same and the cost are dominated by the memory bandwidth.



Figure 4-12: Time taken with different k (64-bit double values)

4.5.4 Dependence on Data Distribution

Keeping the data size fixed at 2^{29} , we examine the performance of algorithms with varying k on 2 distributions:

- Huge Floats: Floating point numbers from $U(0, 10^6)$
- Increasing: Sorted floating point numbers from U(0,1)
- *Bucket Killer*: Contains all 1s(floats) except 4 numbers, each of which differ from 1.0 in one 8-bit digit. This minimizes the reduction achieved in a single radix-scan.

Figure 4-13 shows the results. The only algorithms that do not change based on the distribution of elements are **Sort** and **Bitonic TopK**. Both perform precisely the same operations.

With huge floats (figure 4-13a), the performance of all the methods except Radix Select is the same as in Figure 4-10. Radix Select does worse than in the U(0, 1)case as the first pass yields very little reduction. Where in the U(0, 1) case, we could (after calculating the histogram) determine all values have the same radix and elide the clustering pass, no such optimization is possible in this case. The clustering pass,



Figure 4-13: Top-K performance across different distribution



Figure 4-14: Performance with varying data size

thus, induces costs but creates very little benefit.

The *increasing* (figure 4-13b) distribution leads to **PerThread TopK** performing up to 3x worse while the other algorithms see no change. This is because **PerThread TopK**'s performance is dependent on number of heap inserts. With increasing distribution, each element causes a heap insert making it a near worst case for the algorithm.

For most selection algorithms, it is relatively easy to identify distributions which will cause worst case behaviour for the algorithms. *Bucket killer* is the adversarial distribution for Radix Select. With *bucket killer* (figure 4-13c), Radix Select ends up taking the same time as Sort because each radix pass leads to only one number being removed from consideration (the one which differs from 1 at that 8-bit digit). Each pass ends up reading and writing the entire dataset like in Sort. Bucket Select also experiences a 2x slowdown due to less data reduction in the intermediate steps. Note that, due to the predictable pattern of the bitonic merges, there is no adversarial input distribution for the Bitonic TopK approach making it a very robust option.

4.5.5 Dependence on Data Size

To show the performance of the algorithms across different data sizes, we run them with a fixed k = 64 and choose a data set of random floats drawn from U(0, 1) with varying data sizes ranging from 2^{21} to 2^{29} . Figure 4-14 shows the results. Bitonic



Figure 4-15: Performance with different number of keys

TopK and Sort grow linearly with input size. PerThread TopK maintains top-k per thread and runs a fixed number of threads to keep all the GPU cores busy. With larger data sizes, the number of elements processed by each thread increases. Also, for uniform distribution the probability of a heap insert decreases as more and more data is seen. This results in the initial outward bulge. Radix Select and Bucket Select grow linearly for larger data sizes. At data sizes below 2^{24} , the time taken by prefix sum (which is a constant across data sizes) becomes significant leading to flattening of the lines.

4.5.6 Key(s)+Value

So far, we used tuples with just a key. However, many applications would require key+value or multiple keys+value. In this section, we show the performance of **Radix** Select and Bitonic TopK with key + value (KV), two keys + value (KKV) and, three keys + value (KKKV). Each key is a float drawn from U(0, 1) and value is a 4 byte integer. Size of the elements in the dataset is 2^{28} . Figure 4-15 shows the results. Both the methods show a linear increase in the runtime due to increased data sizes as we go from KV to KKKV. The cut-off point remains the same across the different key counts. We do not show the results for the other methods for readability.

We do not show experiments with larger value payloads as it is always better to pass around the tuple id and construct the full tuple at the end of top-k. For example, consider a dataset with 10 million tuples of 4 byte key, 12 byte payload. Running topk on (key,id) instead of (key,payload) halves the data size moving around. Assembling the result at the end takes virtually no time.

4.5.7 Comparison against CPU

In this section, we compare the performance of CPU-based top-k to the GPU-based top-k. For CPU-based top-k, we have two heap-based methods: one using C++ STL priority queue as a min-heap (STL PQ) and, second a hand-optimized min-heap (Hand PQ). For each element, we check it against the heap minimum by comparing with the root of the heap. If its greater, we pop the root (the minimum) and insert the new element. We also show the CPU version of bitonic top-k. For GPU-based top-k, we show Bitonic TopK and Radix Select.



Figure 4-16: Comparing GPU Top-K against CPU Top-K

First, we compare them on a dataset of 2^{29} floats drawn from uniform distribution U(0, 1). Figure 4-16(a) shows the results. As the data is uniformly distributed, most of the elements get discarded when checked against the heap minimum and very few trigger a heap insertion. To illustrate this note that, for this dataset, with k = 32, each core looks at 671k elements and ends up doing about 500 insertions (including the first 32 elements that always get inserted). The performance is, thus, likely to be memory bound. Bitonic TopK does 3x better than Hand PQ when k = 32. Bitonic top-k on the CPU does significantly worse than heap-based methods as it does significantly



(a) Get top-k most retweeted tweets in a time range

(b) Find top-k most popular tweets

Figure 4-17: Using Top-K kernel in MapD

more computation than heap-based methods which do just 500 insertions.

Next, we consider the same dataset but sorted in increasing order. Figure 4-16(b) shows the results. Since the data is sorted, each element causes a heap pop/insert. This is close to the worst case. Bitonic TopK and Radix Select take the same time while the CPU algorithms do significantly worse. Bitonic TopK does 60x better than Hand PQ and 120x better than STL PQ when k = 32. Time taken by bitonic top-k on the CPU is close to that of Hand PQ despite doing more comparisons. This is due to the use of SIMD instructions.

As empirically demonstrated in this section, Bitonic TopK is the best performing approach for smaller K ($K \leq 256$) and Radix Select for larger K. To provide an analytical argument in support of these findings and to predict the performance on different hardware, we develop a hardware-conscious cost model in Section 4.6.

4.5.8 MapD Integration

To evaluate the performance improvement got from Bitonic TopK in a real world setting, we evaluate the system on a twitter dataset consisting of 250 million tweets from May 2017. We evaluate four queries:

1) SELECT id FROM tweets WHERE tweet_time < X

ORDER BY retweet_count DESC LIMIT 50

The query finds the top 50 most retweeted tweets in a specified time range. We vary the time range to have selectivity from 0 to 1 in steps of 0.1. MapD by default runs the filter on the time range followed by sort on the GPU. It then copies the top-k tweet ids and assembles the tweet (Filter+Sort). We evaluate two alternatives: 1) replace the sort by bitonic top-k (Filter+Bitonic TopK), 2) combined kernel that runs filter and bitonic top-k together (Combined Bitonic TopK). Figure 4-17a shows the results. Bitonic top-k based methods out-perform the existing methods. The filter fusion optimization saves the time to write out to and read in from global memory of the filtered id, retweet count entries. At selectivity 1, the filter fusion optimization reduces the total kernel runtime (time spent on GPU) by 30% and the end-to-end runtime by 23%.

2) SELECT id FROM tweets

ORDER BY retweet_count + 0.5 * likes_count DESC LIMIT K

The query finds the most popular tweets based on a complex ranking function. MapD by default runs a projection step that computes the value of the ranking function followed by a sort step (Project+Sort). We evaluate two alternatives: 1) replace sort with bitonic top-k (Project+Bitonic TopK), 2) a combined kernel that computes the value of the ranking function inside the SortReducer (Combined Bitonic TopK). Figure 4-17b shows the results. The combined kernel saves on having to having to write out and read in the projected rank value. This reduces the runtime of the combined method by 10ms compared to Project+Bitonic TopK.

3) SELECT id FROM tweets WHERE lang='en' OR lang='es'

ORDER BY retweet_count DESC LIMIT K

The query finds the top K tweets by retweet count in english or spanish language. We evaluate the same 3 methods used in query (1). The filter has a set selectivity of around 80%. We see the same trend as in the previous query. The combined kernel saves on having to read/write filtered id,retweet count entries. This reduces the runtime by 16ms compared to Filter+Bitonic TopK across all K.

SELECT uid, COUNT() AS num_tweets FROM tweets

GROUP BY uid ORDER BY num_tweets DESC LIMIT 50

The query finds the top 50 users by tweet count. There are about 57 million unique users in the dataset. By default in MapD, the query execution takes 97ms of which
the sort step takes 44ms. Using bitonic top-k reduces the runtime by 39% as it reduces the time taken by the sort step by 38ms. A query which finds say the 50 most popular hash tags would not benefit as much from bitonic top-k as the most of the time is spent in the group by step.

4.6 Cost Model

Due to space constraints, we limit our modeling efforts to the two best-performing algorithms (see last section): Radix Select and Bitonic TopK. We model them using hardware parameters we determined empirically using benchmarks provided by Nvidia. The parameters are 1. the global memory bandwidth (B_G) , 2. the shared memory bandwidth (B_S) , 3. the key size in bytes (w), 4. the the input data size in bytes (D) and 5. the total number of threads (n_t) .

4.6.1 Radix-based Top-K

Radix-based top-k (Radix Select) operates as a series of passes, each pass looking at one digit of 8 bits. Each pass reduces the data size and the total number of passes is at most w/8. Pass *i* involves:

• Read the input for the pass from global memory to write out the number of entries per digit value per thread (total: 16 integers per thread). D_{iI} is the input size for the pass in bytes. $D_{iI} = D$ for the first pass.

$$T_{i1} = \frac{D_{iI}}{B_G} + \frac{16 * 4 * n_t}{B_G}$$

• Calculate the prefix sum to find the digit value *d* containing the k-th value.

$$T_{i2} = \frac{2 * 16 * 4 * n_t}{B_G}$$

 Scan the input and write out entries with digit value d to another array in global memory. Let η_i be the fraction of entries with digit value d. Note that this step is skipped if $\eta_i = 1$.

$$T_{i3} = \frac{D_{iI}}{B_G} + \eta_i \frac{D_{iI}}{B_G}$$

The total time of pass i is $T_i = T_{i1} + T_{i2} + T_{i3}$. The total cost is the sum of the time taken by the individual passes.

4.6.2 Bitonic Top-K

Bitonic top-k runs a sequence of kernels: first the SortReducer kernel, followed by a series of BitonicReducer kernels. Let x be the number of elements per thread. Each kernel reduces the problem size by a factor of x. For every kernel, there are two components that can dominate performance depending on K: global memory access or shared memory access. Due to the high parallelism and the low overhead of context switches, the GPU will effectively hide the cost of the less expensive of these two behind the more expensive. The cost is, thus, the maximum of the two.

We start with the global memory access cost of the SortReducer kernel. The kernel makes one scan of the input from global memory and writes out 1/x of the input back (to global memory). The global memory data access time thus straight forward to model:

$$T_g = \frac{D}{B_G} + \frac{1}{x} \frac{D}{B_G}$$

The shared memory data access time is harder to estimate: in addition to the number of accesses, we need to take the number of shared memory bank conflicts into account. Since bank conflicts occur whenever two values on the same bank are accessed, we need to take the specific addresses of memory accesses into account.

The time taken if the kernel is bound by shared memory bandwidth is the sum of the time taken by each combined step:

$$T_s = \Sigma_i \delta_i \frac{D_{Ii} + D_{Oi}}{B_s}$$

where δ_i is the number of shared memory bank conflicts for one warp and, D_{Ii} and



Figure 4-18: Estimated vs actual runtimes for different K

 D_{Oi} are size of data read and written by the phase respectively. Applying this to find T_s for SortReducer finding the top-32, we get $T_s = 17.5D/B_s$.

The estimated time taken by the *SortReducer* kernel is $max(T_g, T_s)$. For the Titan X Maxwell, $B_S = 2.9TBps$ and $B_G = 251GBps$. The estimated total time is max(8.96ms, 12.1ms) = 12.1ms which is close to the actual runtime of 14.2ms. The cost for BitonicReducer can be estimated in a very similar way except that it directly starts with len = k/2.

Figure 4-18 compares the actual time of the methods versus the predicted time based on the models for finding top-k on a dataset with 2^{29} floating point numbers drawn from U(0, 1) with varying K. The predicted times show the same trends as the observed times and the cutoff point remains the same. Both the models underestimate the time taken. This is because a kernel bound by global or shared memory may not achieve the maximum possible bandwidth. For example, the first kernel of radixbased top-k should take 8.6ms based on model while in reality it takes 9.8ms and, the effective shared memory bandwidth used by the *SortReducer* kernel for k = 32is around 2.5TBps versus the maximum 2.9TBps.

As demonstrated in this section, bitonic top-k is not only experimentally faster but also theoretically more efficient than the best alternative we evaluated.

4.7 Conclusion

Data analytics on GPUs is increasingly common, and a frequently analytics task is to rank a set of data items according to some attribute and extract the top-k values. In this chapter, we presented many algorithms to efficiently compute top-k on GPUs, including a new algorithm based on bitonic sort. Through an extensive performance evaluation of a number of different algorithms, we showed that our bitonic-top-k algorithm is an order of magnitude faster than the fastest algorithms based on fully sorting a list of elements, and, depending on the value of k, several times faster than several other algorithms for efficiently computing top-k. We also presented a cost model that accurately predicts the performance of several algorithms with respect to k, allowing a query optimizer to choose the best top-k implementation for a particular query.

Chapter 5

Data Compression

5.1 Introduction

Several commercial systems, including Omnisci [9], Kinetica [6], and BlazingDB [4], aim to provide real-time analytics capabilities by using GPUs to store a large fraction (or all) of the working set. A key constraint in these systems is the GPU memory capacity. Currently, GPUs have at most 40 GB of memory which is used both to cache the working set and as scratch memory for query execution. GPU memory is $6\times$ more expensive compared to CPU RAM [70] and going outside a single GPU's memory incurs a penalty. Therefore, data compression is critical. Currently, GPUbased systems use simple compression schemes like fixed-width dictionary encoding and run-length encoding (RLE) [9, 83] similar to CPU-based in-memory analytics systems, and decompressing on-the-fly during query execution. To the best of our knowledge, no GPU database today uses bit-packed schemes which have been shown to achieve the best compression ratios [27] on the CPU but are non-trivial to decode in parallel across thousands of threads.

In this chapter, we introduce two new efficient compression schemes for GPUs: GPU-FOR which does bit-packing in conjunction with Frame-Of-Reference (FOR) and GPU-DFOR which uses delta encoding with bit-packing and FOR. Both these schemes are designed to offer improved compression ratios while still being able to decode them in parallel across thousands of threads at close to memory bandwidth speeds. GPU-FOR partitions the data into blocks, in each block encoding integers with the minimum bit size needed to represent a value in the block. It works well with uniform data and can handle skew. GPU-DFOR first delta-encodes a block of integers before using GPU-FOR. It is suited for sorted and semi-sorted columns.

Past works [28, 46] have looked at delta encoding, FOR, and variable length byte-aligned packing (NSV). These works found that achieving the minimum space cost by using a combination of compression schemes (e.g. delta+NSV) can degrade performance as intensive decompression overburdens the GPU. Hence previous work deemed these schemes GPU-unfriendly. The reason for bad performance was that these works treated threads on the GPU as independent execution units and hence required multiple passes to decode the compressed data. For example: to use a column encoded using delta encoded variable length byte-aligned packing in a query, these systems would first run a prefix-sum primitive to unpack the variable-length byte-packed data and write it to global memory, then do a second pass prefix-sum primitive to delta decode the data which is written to global memory, and finally the query kernel would read from global memory the unpacked column — the intermediate data is read/written to global memory multiple times. In our work, we treat a *thread block* as the basic execution unit and each thread block collectively decodes one block of encoded entries. By treating the thread block as the basic unit, we are able to cache a block of data in on-chip caches and inline the multiple steps involved in decoding into a single kernel, resulting in a single pass over the data. We present a series of optimizations that enable us to decode at close to memory bandwidth speed. The performance of our schemes simplifies the choice of a compression scheme to encode a column — we choose the scheme with the smallest storage footprint. It eliminates the need for sophisticated compression planners used by past works to choose the right compression scheme.

To show that our compression schemes perform well and and significantly reduce the storage footprint of GPU-based systems, we present an integration of GPU-FOR and GPU-DFOR into the Crystal framework [70]. We encapsulate the decompression into a device function that enables programmers to change a kernel operating on an uncompressed array to a compressed column with a single line of code. In the end, we find that our compression schemes can reduce the storage footprint by up to $10 \times$ on certain data distributions; on the Star Schema Benchmark, the proposed scheme achieves 50% reduction in storage compared to no compression and 37% compared to existing GPU compression schemes with almost no impact on performance.

In summary we make the following contributions:

- We present two bit-packing based compression schemes GPU-FOR and GPU-DFOR that can be used to store data compactly on the GPU.
- We develop a series of optimizations that allow us to decode the encoded data on-the-fly at close to memory bandwidth speed.
- We present an integration of GPU-FOR and GPU-DFOR into the Crystal framework and demonstrate ease of use.
- We present an evaluation on multiple synthetic benchmarks and on the Star Schema Benchmark (SSB). On SSB our schemes can achieve significant (37% on SSB) space savings and just 4% loss in performance compared to existing methods.

The rest of the chapter is organized as follows: related work and background are discussed in Section 5.2. We present the data format and the unpacking implementation on the GPU for GPU-FOR and GPU-DFOR in Section 5.3 and Section 5.4, respectively. In Section 5.5, we discuss the database integration. In Section 5.6, we evaluate the performance and compression ratio of binary packing against other schemes on the GPU. Finally, we conclude in Section 5.7.

5.2 Background

In this section, describe relevant aspects of past approaches to data compression on both GPUs and CPUs.

5.2.1 Compression Techniques

Compression techniques are heavily exploited in modern column-store databases for efficient query processing. These databases store relational data in a decomposition storage model (DSM) [26] where a *n*-attribute relation is replaced by *n* arrays, one for each attribute. Since each attribute is stored separately as a sequence of values, we can use lossless compression techniques to store them compactly. Based on the compute intensity of decompression, lossless compression techniques are categorized into two buckets: *lightweight* and *heavyweight*. Lightweight algorithms are mainly used in in-memory column stores while heavyweight algorithms like Huffman [41] and Lempel Ziv [84] (together with lightweight techniques) are used in disk-based column stores. In this chapter we focus on lightweight techniques. We show later in Section 5.6 that most of the compression gains are achieved with just lightweight techniques for our workload.

There are five basic lightweight techniques to compress a sequence of values: frame-of-reference (FOR) [32, 85], delta coding (DELTA) [45], dictionary compression (DICT) [12, 85], run-length encoding (RLE) [12], and null suppression (NS) [12].

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

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

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

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

NS removes leading zeros from an integer's bit representation. NS is useful when a

column contains many small integers.

FOR, DELTA, DICT, and RLE work at the logical level where a sequence of values is compressed into another sequence. NS addresses the physical level of bits with the basic idea of removing leading zeros in the bit representation of small integers. There are many different NS techniques proposed which can broadly be categorized as (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned. Bit-aligned NS algorithms compress an integer to a minimal number of bits, byte-aligned NS compress an integer with a minimal number of bytes, and word-aligned NS encode as many integers as possible into 32/64-bit words. The NS algorithms also differ in their data layout. We distinguish between *horizontal layout* and *vertical layout*. In the horizontal layout, the compressed representation of subsequent values is situated in subsequent memory locations. In the vertical layout, each subsequent value is stored in a separate memory word in a striping fashion.

Researchers have proposed a number of NS algorithms for compressing columns in main memory database management systems (DBMS) on CPUs. SIMD-Scan [80] stores column values in a tightly bit-packed horizontal layout, ignoring any byte boundaries and uses SIMD instructions to scan a column of entries. For example, to store a column of 11 bits in memory, the first value is put in the 1st to 11th bits whereas the second value is put in the 12th to 22nd bits and so on. Such a bit-packed layout incurs overhead to unpack the data before processing and does not saturate memory bandwidth. In the example above, several SIMD instructions have to be spent to align eight 11-bit values with the eight 32-bit banks of a register. Li and Patel [47] proposed the Horizontal Bit-Parallel (HBP) and Vertical Bit-Parallel (VBP) storage layouts. HBP is a word-aligned layout that packs as many entries as possible into the same word. In the example, 5 11-bit entries would be packed in 64-bit word and the remaining 9 bits wasted. Due to padding, HBP does not achieve compact storage. In VBP, if a processor uses S-bit words, it groups S entries of k bits each into k processor words such that the ith processor word contains the ith bit of each entry. Reconstructing/looking-up a value under the VBP layout is expensive though. That is because the bits of a value are spread across k words. ByteSlice [30]



Figure 5-1: Bit packing with vertical data layout

improves on VBP. It groups S/8 entries to: (1) an S-bit memory word contains bytes from S/8 different values; (2) bytes of a k-bit entry are spread across $\lfloor k/8 \rfloor$ words. ByteSlice stripes by byte, hence while scan is faster than VBP, the storage footprint is also larger than VBP.

The best performing NS scheme that also achieves good compression ratios is SIMD-BP128 [45] (and its variants). SIMD-BP128 processes data in blocks of 128 integers at a time and stores these integers in a vertical layout using the number of bits required for the largest of them. Figure 5-1 illustrates the vertical layout where the first four integers Int1, Int2, Int3, Int4 start out in four different 32-bit words. Int5 is immediately adjacent to Int1, Int6 is adjacent to Int2, etc. Each lane has 32 integers. The used bit width is stored in a single byte, whereby 16 of these bit widths are followed by 16 compressed blocks. SIMD-BP128 achieves better compression than Li and Patel [47] and ByteSlice, and can be decoded at memory bandwidth speed.

In Section 5.3.3, we discuss why directly translating SIMD-BP128 to the GPU leads to bad performance. GPU-FOR uses a horizontal layout similar to SIMD-Scan, however the decoding algorithm is novel and contains optimizations tailored to the GPU architecture. GPU-FOR achieves a better compression ratio than HBP as it does not use padding. We think VBP is not well suited for the GPU architecture as it is not easily vectorizable. In our evaluation, we compare against byte-aligned null suppression which achieves the same compression as ByteSlice.

5.2.2 Query Execution on GPUs

With the slowing of Moore's Law, CPU performance has stagnated. In recent years, researchers have begun to explore heterogeneous computing as a way to overcome

the scaling problems of CPUs and to continue to deliver interactive performance for database applications. Ocelot [39] provides a hybrid analytical engine as an extension to MonetDB. YDB [83] is a GPU-based data warehousing engine. HippogriffDB [46] used GPUs for large scale data warehousing where data resides on SSDs. More recently, HorseQC [31] proposes pipelined data transfer between CPU and GPU to improve query runtime. All these works have focused on using GPU as a coprocessor, where data is stored primarily on the CPU side and moved to the GPU at query execution time. Recent work [70] has shown that GPU as a coprocessor is slower than just running queries on the CPU, instead a better model is to store the working set directly on the GPU memory. The memory capacity of GPUs has increased significantly over the years, today a GPU can have up to 40GB of High Bandwidth Memory (HBM), which is likely to further increase as HBM technology improves. It is possible to attach up to 20 GPUs to a single socket CPU allowing the user to aggregate enough memory to store large datasets. Commercial systems like Omnisci [9], Kinetica [6], and BlazingDB [4] use this philosophy and aim to provide real-time analytical capabilities by using GPUs to store large parts of the working set. This chapter focuses on implementing compression schemes efficiently on the GPU so that more data can be cached on the GPU with minimal performance degradation. The compression schemes are beneficial to systems that use GPU as a coprocessor as well as they help reduce the data transfer time between CPU and GPU (see Section 5.6.5).

Researchers have looked at data compression for GPUs in the past. Yuan et al. [83] studied effect of RLE and DICT compression on query execution. Fang et al. [28] extended the work and studied a larger set of compression schemes like DICT, FOR, RLE, Null Suppression with Fixed Length (NSF), and NS with Variable Length (NSV). In NSF, all values are encoded with the number of bits being multiple of 8 to ensure output values are byte-aligned. As GPU is byte-addressable, they claimed that this achieves good decompression performance. NSV uses a variable number of bytes per entry. For each value, it stores the number of bytes used to store value (1, 2, 3, or 4) followed by bytes of the output value. Jing et al. [46] also studied compression, however they reused the implementation of Fang et al.. A key issue with past work is that they treated a cascade of compression schemes as independent layers. For example, data compressed with DELTA + FOR + NSF would run the DELTA decoding kernel, followed by FOR kernel and NSF kernel, finally using the uncompressed column in the actual query execution. Each kernel would read and write the entire column to global memory. As a result, cascades of compression schemes would achieve lower performance while likely having better compression ratios introducing a cost-benefit problem. Hence, both works proposed compression planners that based on a cost-model decided which cascades to use to minimize space cost while also ensuring query performance is not impacted significantly.

Compared to past work, this chapter focuses on building efficient decompression routines that can decode (variable length) *bit-aligned* null suppression schemes. Past works have looked only at byte-aligned schemes (NSF/NSV) and didn't evaluate bit-aligned schemes that we describe in this chapter which we show achieve better compression ratios. We also show our compression schemes (which are a cascade of basic compression schemes) can be decoded in a single pass over the data and inline during query execution at close to memory bandwidth speed. This eliminates the need for complicated compression planners. Commercial systems currently only implement fixed length dictionary encoding and would also benefit from our work.

5.3 Fast Bit Unpacking

In this section, we describe the GPU-FOR compression format, which uses bit-packing in conjunction with Frame-of-Reference (FOR) to store data compactly on the GPU and the fast bit unpacking routine used to decompress it efficiently on the GPU. GPU-FOR can be used to efficiently compress attributes of type integer, decimal, or dictionary-encoded string (i.e. sequence of integers) in a column store. At query time, the query executor will need to decompress data and run the query on the decompressed data. Hence, optimizing the performance of decompression is critical for analytic workloads. In contrast, data is only compressed once as it is loaded into the GPU, so optimizing the performance of compression is not as important. In the rest of the section, we first describe the bit-packed representation we use and then describe the kernel implementation on the GPU. We present a series of optimizations which allow us to decode bit-packed data while saturating memory bandwidth.

5.3.1 Data Format

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

In bit-packing, a sequence of values is encoded with fixed bit size b. Choosing a common bit size b for an entire array would mean that the occurrence of a single large value would increase the number of bits needed to encode the values. Hence, bitpacking is generally used in conjunction with FOR encoding. In GPU-FOR, the array of values is partitioned into *blocks*. We use blocks of 128 integers. The range of values in the block is first found and then all the values are written in reference to the minimum value: for example, if the values in a block are integers in the range [100,130], then using a reference of 100, we can store them using 5 bits ($log_2(130 + 1 - 100)$). Each block is further divided into sequences of 32 integers called *miniblocks*. For each miniblock, we choose a bit-width based on the maximum number of bits needed to encode the largest value. Each bitwidth can be stored in 1 byte. We store the bitwidths of 4 miniblocks at the start of the block using a single integer. The choice of the size of the miniblock and the reason for storing bitwidths together is to ensure they align on 32-bit boundaries. This allows us to use 32-bit arithmetic while decoding and makes shared memory accesses (which are aligned to 32-bit boundaries) efficient.

The bit-packed array needs to be decoded in parallel across a large number of threads. For this, we store the start index of the blocks in a separate array called block starts. Finally we store the metadata associated with the encoding: block size (i.e., the number of integers within each block), miniblock count (i.e., number of



Figure 5-2: GPU-FOR Data Format



Figure 5-3: Example encoding with GPU-FOR

miniblocks per block), and the total count (i.e., total number of integers in the data array) in the header. Figure 5-2 shows a schematic of the format we use to store data.

Figure 5-3 shows an example of encoding 16 integers into a block with 2 miniblocks. The minimum value in the block (i.e., 99) is used as the reference. We calculate the difference of the block values from the reference. Each miniblock contains 8 integers. We see that the first miniblock needs 2-bits per block while the second miniblock needs 4-bits per block. We encode each miniblock with their respective bitsizes and store the reference and bitwidths at the start of the block.

The key difference between GPU-FOR and state-of-the-art bit-packing algorithms for CPUs like SIMD-BP128 is that GPU-FOR uses horizontal data layout to store the entries while SIMD-BP128 uses vertical data layout. We will discuss later in Section 5.3.3 the reason for this choice.

Algorithm 6: Fast Bit Unpacking on GPU — The following code runs on each of the 128 threads within a threadblock in parallel.

```
Input : int[] block starts; int[] data; int block id;
             int thread id
   Output: int item
1 int block start = block starts[block id];
2 unit * data block = \& data[block start];
3 int reference = data block[0];
4 uint miniblock_id = thread_id/32;
5 uint index_into_miniblock = thread_id \& (32 - 1);
6 uint bitwidth\_word = data\_block[1];
7 unit miniblock\_offset = 0;
s for j = 0; j < miniblock\_id; j++ do
       miniblock offset += (bitwidth word & 255);
9
      bitwidth word \gg = 8;
10
11 unit bitwidth = bitwidth word \& 255;
12 unit start bitindex = (bitwidth * index into miniblock);
13 unit header offset = 2;
14 uint start_intindex = header_offset + miniblock_offset + start_bitindex/32;
15 uint64 \ element\_block = data\_block[start\_intindex]
    (((uint64)data\_block[start\_intindex + 1]) \ll 32);
16 start\_bitindex = start\_bitindex & (32-1);
17 unt element = (element block & (((1 \leq bitwidth) - 1) \leq start bitindex)) \gg
    start bitindex;
18 item = reference + element;
```

5.3.2 Implementation

In this section, we describe a number of optimizations at the implementation level that we applied to achieve decompression at close to GPU memory bandwidth speed. These optimizations are inspired by similar optimizations for other algorithms. However, to the best of our knowledge, none have been applied in the context of parallel decompression of bit-packed data. To give an impression of the importance of each optimization, we end every subsection with the time taken to decode a compressed dataset of 500 million integer values drawn from a uniform distribution $U(0, 2^{16})$. The details of the experimental setup can be found in Section 5.6.1.

Base Algorithm: Algorithm 6 shows the pseudocode that would run in parallel on each thread (n threads are allocated for n-element dataset). Each threadblock (of size 128 threads) is assigned to decode a block (of 128 elements) with each thread decoding one element in the block based on its index. Each thread starts by reading the block start pointer of the block to find where in the data array the block starts (line 1–2). Each thread then reads in the bitwidth_word, uses it to compute the offset of its miniblock in the data array (miniblock_offset) (lines 7–10). In computing the miniblock offset, we use the fact that if entries in a miniblock are encoded with b bits, then the miniblock occupies b bytes (since there are 32 entries per miniblock). Next, we compute the offset in bits within the miniblock (line 12). Since the entries are bit-packed, they are not byte-aligned and can span byte boundaries. Using starting bit index, we calculate the starting integer index (start_intindex) of the entry (lines 13-14). We then load an 8-byte block starting at start_intindex (element_block) (line 15). This block contains the entire element, we use bitshift arithmetic to extract the entry (lines 16–17). Finally, we add reference and return the result. The result resides in a register and is used subsequently during query execution. In Section 5.5, we describe in greater detail how the algorithm is used during query execution.

This algorithm takes 18 ms to decompress the dataset described at the start of the section. Reading an uncompressed dataset of 500 million 4-byte integers takes 2.4 ms. This means decompressing the dataset is $7.5 \times$ slower than reading the uncompressed data. We use a number of optimizations detailed below to bridge the gap:

Optimization 1: Operating in Shared Memory

Each thread makes multiple requests to the data array which sits in global memory. Since, all the requests made by all threads within a threadblock touch one data block, in this optimization, we load the entire block into shared memory once at the start of the operation. Each threadblock starts by reading block_start[BlockId] and block_start[BlockId+1] to determine the boundaries of the data block to be processed and then loads it into its shared memory in a coalesced manner. All subsequent requests are made to the data block in shared memory.

Recall that the shared memory is an order of magnitude faster than global memory. This optimization shifts global memory reads to shared memory reads, thereby improving performance. This optimization results in runtime reduction from 18ms to 7ms on the sample dataset.

Optimization 2: Processing Multiple Blocks

The granularity of reads from global memory is 128 bytes [70]. Maximum bandwidth is achieved when warps' (groups of 32 threads) accesses to global memory result in neighboring locations being accessed. The best case scenario is when 32 threads access a 4-byte integer array of size 32, resulting in a perfect 128 byte access. In the sample dataset, if all integers end up being encoded with 16 bits, the block size is 258 bytes (2 bytes for block header + 256 bytes for miniblocks). When a threadblock of size 128 reads in the data block from global memory, some warp accesses do not result in an aligned full segment being read from global memory. The same issue occurs when we access the block_start array, we are reading in only two values from global memory, again leading to loss of efficiency.

In this optimization, we attempt to reduce the impact of these irregular accesses to global memory by processing multiple data blocks per threadblock. Each threadblock is assigned D(=2/4/8/16/32) data blocks to process. At the start, each threadblock reads in D + 1 block_start entries from global memory. Next they read in the data blocks block_start[D×BlockId] and block_start[D×BlockId + D] from global memory into shared memory. As a result, we have reduced the number of irregular accesses to both the block_start and the data array.

Figure 5-4 shows the runtime for decompression of the sample dataset with varying D. As we can see from the figure, the largest reduction comes from going from D = 1 to D = 4. Going from D = 4 to D = 16 improves the performance, however the improvement is marginal. Finally, when we go to D = 32 the performance deteriorates significantly. This is because the result of the decompression is stored in registers. While increasing D reduces the number of irregular accesses, the number of registers required and the shared memory requirement increases proportional to D. Each thread on the GPU has limited amount of registers and shared memory available. On an Nvidia V100 GPU, each thread can use 65 registers and 48 bytes of shared memory per thread at full occupancy. As a result, when we go to D = 16, each thread requires 64 bytes of shared memory which reduces occupancy slightly. When we go to D = 32, each thread requires 128 bytes of shared memory which results in



Figure 5-4: Decompression performance with varying number of data blocks per thread block (D)

significant reduction in occupancy and register spilling — hence the slowdown.

When we run full SQL queries, we have to store D values per output column in registers until the end of the query. We noticed in our evaluation on the Star Schema Benchmark (discussed later in Section 5.6.4) that there is little difference in performance with D = 4/8 and choosing D > 8 leads to deterioration in performance. This is because each query has 3-4 output columns and choosing higher values of Dleads to register spilling and reduced occupancy. Hence, we choose to simply use D = 4 in the rest of the chapter. Note that D is a parameter and users can choose higher value of D in case they are just decoding a single column.

Optimization 3: Precomputing Miniblock Offsets

Computing the miniblock_offset involves a for loop (lines 8–11 in Algorithm 6). We can make two observations: (1) miniblock offsets are a exclusive prefix sum over the bitwidths array (for example, if the bitwidths used by 4 miniblocks within a block are 7, 8, 9, and 10, the miniblock offsets are thus 0, 7, 15, and 24); (2) with D = 4, there are only D * 4 = 16 unique miniblocks offsets to compute, while Algorithm 6 performs this computation on all 128 threads redundantly. In this optimization, we reduce the compute load of the algorithm by precomputing the D * 4 miniblock

Algorithm 7: Precomputing Miniblock Offset — The following code
runs on each of the first $4 \times D$ threads within a threadblock.

Input : int[] s_block_starts; int[] s_data; int thread_id Output: int[] s_offsets; int[] s_bitwidths; 1 int block_index = thread_id / 4; 2 int miniblock_index = thread_id 3 uint bitwidth_word = s_data[s_block_starts[block_index] - s_block_starts[0] + 1]; 4 uint miniblock_offsets = (bitwidth_word << 8) + (bitwidth_words << 16) + (bitwidth_word << 24); 5 uint miniblock_offset = (miniblock_offsets >> (miniblock_index << 3)) & 255; 6 uint bitwidth = (bitwidth_word >> (miniblock_index << 3)) & 255; 7 s_offsets[thread_id] = miniblock_offset; 8 s_bitwidths[thread_id] = bitwidth;

offsets once at the start and storing them in shared memory. Algorithm 7 shows the pseudocode for the optimization. It runs on the first D * 4 = 16 threads (i.e. thread_id $\in [0, 16)$). All the array prefixed by s_ are in shared memory. We start by assigning each thread one miniblock offset/bitwidth pair to compute (lines 1-2). Each such thread loads the corresponding bitwidth word (line 3) and computes a prefix sum over it using bitshift arithmetic (line 4). Finally we extract the relevant offset and bitwidth for the miniblock and store it in shared memory (lines 5-8). These values are read by each thread when they need it. The optimization eliminates the for loop in lines 8–11 in Algorithm 6 and reduces the runtime from 2.39ms to 2.1ms, which is lower than the time taken to read the uncompressed data.

5.3.3 Discussion

GPU-FOR vs SIMD-BP128: As described earlier in Section 5.2.1, there are two variants of bit-packing based on the data layout: *horizontal* and *vertical*. On the CPU, the fastest bit-packing/unpacking routine is SIMD-BP128 [45]. SIMD-BP128 stores integers in a vertical layout. It uses SSE instructions with each SSE register holding 4 32-bit integers. The data is encoded with 4 lanes each with 32 integers allowing the data to be decoded efficiently by mapping each lane to a different vector lane of the SSE register. Each block encodes 128 integers. To ensure 16-byte alignment, SIMD-BP128 groups 16 blocks together, storing the bitwidths used in each block at the start. This is similar to GPU-FOR format with each block having 16 miniblocks, with each miniblock having 128 integers and encoded with a vertical layout.

On the GPU, if we consider a SIMD lanes as equivalent to a GPU threads in a warp, we can directly translate the SIMD-BP128 style vertical storage layout to the GPU. Let's call this GPU-SIMDBP128. We go from having 4 lanes on the CPU to 32 lanes on the GPU (warp size is 32 threads). As a result, on a typical thread block size of 128, with each thread having 32 integers to ensure their lane terminates in 32-bit boundaries, we would need a block size of 4096 vs 128 on the CPU. We implemented GPU-SIMDBP128 and compared the performance of GPU-FOR vs GPU-SIMDBP128 on the same microbenchmark. GPU-FOR (with D = 16) takes 1.55ms compared to GPU-SIMDBP128 which takes 4.3ms. Hence GPU-SIMDBP128 is 2.7x slower than GPU-FOR.

On the GPU, vertical packing like in SIMD-BP128 is slower because the number of registers available per thread is limited. Decoding the vertical layout would require space for 32 4-byte entries and 32 registers to store output. Similar to the case when D = 32, this leads to reduced occupancy. Furthermore, if we have a query with only 3 columns needed for result computation, we would need more than 2× the registers available per thread resulting in significant register spilling. To get a sense for the performance impact, we evaluated the Star Schema Benchmark q1.1 (described later in Section 5.6.4) with columns encoded using GPU-FOR vs with columns encoded using GPU-SIMDBP128. The query uses 4 columns. The performance with GPU-SIMDBP128 was 14x slower than with GPU-FOR. Another downside of using GPU-SIMDBP128 is the large block size (4096 vs 128). Large block sizes mean that one skewed value could lead to large bitwidth for the entire block, reducing compression gains.

A CPU has low compute to bandwidth ratio and each CPU core has a large L1 cache. This leads to bitpacking schemes with vertical layout like SIMD-BP128 (which has lower compute intensity but higher storage requirement) perform better than schemes with a horizontal layout like SIMD-Scan [80]. On the GPU, the compute

to bandwidth ratio is higher and each GPU thread has limited resources. This results in bitpacking with horizontal layout like GPU-FOR performing better than using vertical layout on the GPU.

Bit Packing without Miniblocks: Instead of having 4 miniblocks, one could instead just have one miniblock encoded with a single bitwidth. There is no difference in terms of memory space overhead as both schemes store a bitwidth(s) as a single 4byte integer. However, there is reduced computation as we don't have to calculate the miniblock offsets. We implemented this scheme and found the performance to be marginally better. The performance on the sample dataset improves from 2.1ms to 2ms. When we experimented further to see if it is possible to reduce runtime by reducing compute load by having a single bitwidth across blocks or using zero as reference, we could not achieve any further improvement. This leads us to believe that the performance is close to saturating bandwidth given our global memory access pattern.

5.4 Fast Delta Decoding

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

The sequential form of delta encoding requires just one subtraction per value $(\delta_i = x_i - x_{i-1})$. During decoding, we require one addition per value $(x_i = \delta_i + x_{i-1})$. For an array A of k elements, the prefix sum p_A is a k-element array where $p_A[j] = \sum_{i=0}^{j-1} A_j = p_A[j-1] + A_j$. Hence, the process of decoding delta encoded data is



Figure 5-5: GPU-DFOR Data Format

equivalent to computing the prefix sum. Efficient parallel prefix sum routines have been proposed [36] that could be used to decode delta encoded data on the GPU. A simple approach to delta encode + bit-pack the data would be to do it as two separate steps: first compute the deltas for the entire array and then bit-pack the deltas. The decoding would then be a two-step process: the first pass would use the bit unpacking routine described in Section 5.3.2 to decode the deltas and write it to global memory; the second pass would use the prefix sum routine to decode the data. This is the approach used by past work [28, 46] and is inefficient as it requires multiple passes over the data. Later in this section we describe how we can combine the delta decoding step and the bit unpacking step into a single pass.

Note that delta encoding is used only for sorted or partially-sorted data e.g.: to encode the primary key and secondary keys in databases, to encode posting lists in search workloads. Using it for unsorted data could lead to worse compression ratios compared to simply bit-packing the data. To illustrate this, consider a block of integers drawn uniform randomly from [0, 32). The integers can be bitpacked with 5 bits. However the deltas will be in the domain [-31, 31] and would require 6 bits per integer.

5.4.1 Data Format

Delta encoding the entire array as $x_0, \delta_1, \delta_2...$ makes it hard to decode in parallel as decoding the n^{th} block requires the $(n-1)^{th}$ block to have been decoded already. To

enable parallel decoding, we build on GPU-FOR encoding scheme (Section 5.3.1) by partitioning the array into sets of D blocks where each block contains 128 integers and delta encoding each set of D blocks independently (where D is the number of blocks processed per threadblock). Figure 5-5 shows the data format. Encoding xintegers generates x - 1 deltas. Hence during encoding, we pad the deltas with 0 to ensure every block has 128 entries. We store the first value separately before every D^{th} block, with start pointers still pointing to the start of each block.

5.4.2 Implementation

With the data format described above, each tile of D blocks can be decoded independently. During decoding, we first start by loading the D block segments into shared memory and use the fast bit unpacking routine (described in Section 5.3.2) to decode the deltas.

After bit unpacking the deltas, we have D deltas per thread. The output data entries can be calculated using prefix sum over the deltas of all threads within the threadblock. We can use block-wide prefix sum to compute the prefix sum over the deltas based on the work-efficient prefix sum algorithm proposed by Blelloch et al. [20]. For an array of n integers, the algorithm is able to compute the prefix sum of the array in parallel using $\Theta(\log n)$ steps. We start with loading the computed deltas into shared memory to create a contiguous array of deltas for all D blocks. All operations in the algorithm are done in place on the array in shared memory. The algorithm consists of two phases: the up-sweep phase and the down-sweep phase. Each phase consists of a series of steps where each step is a set of additions in parallel across threads. The additions when visualised form a tree pattern as shown in Figure 5-6. In the up-sweep phase, we traverse the tree from leaves to root computing partial sums at the internal nodes of the tree. Figure 5-6a illustrates the up-sweep phase on an array with 8 elements. There are 3 steps. In each step, we do a set of additions in parallel across threads and synchronize the threads after each step. In the downsweep phase, starting with the result of the up-sweep phase, we traverse back down the tree from the root, using the partial sums from the reduce phase to build the



(b) Down-Sweep phase

Figure 5-6: Illustration of Prefix Sum Algorithm

prefix sum result in place on the array. Figure 5-6b shows the down-sweep phase for the example. We start by inserting a zero at the last entry (root of the tree). On each step, each node at the current level passes its own value to its left child, and the sum of its value and the former value of its left child to its right child. In the end, each thread reads D entries back into registers and returns it as a result which will be used in the rest of the query. There are a number of optimizations done to achieve good performance (e.g., using a technique called padding to break shared memory bank conflicts) that we do not touch upon. Interested reader can refer to [36] for more details.

Although prefix-sum has been used widely in libraries like Thrust [11], a key ob-

servation we make is that it is sufficient to do delta coding in each set of D blocks separately. This allows us to get away with doing prefix sum entirely within a threadblock in shared memory. Doing prefix sum over an entire array is much more expensive and involves multiple passes over data. It also allows us to fuse bit unpacking and delta decoding steps into a single kernel which allows our implementation to perform decompression in a single pass over the data blocks in global memory compared to multiple passes required by previous works [28, 46]. The bit unpacking and delta decoding share the same shared memory buffer. The total resource requirement of the kernel is D 4-byte entries in shared memory and D registers to store the output per thread.

Compared to decoding GPU-FOR, decoding GPU-DFOR involves significantly more operations in shared memory. When used to decompress the example dataset described earlier in Section 5.3, the above algorithm takes 4.45ms. This is approximately $2\times$ slower than decompression of the dataset encoded with GPU-FOR while the compressed dataset size is only 6% larger. This is because the decompression of GPU-DFOR is bound by the shared memory bandwidth. GPU-DFOR does better than GPU-FOR on sorted and semi-sorted datasets. Consider a dataset of n = 500 million integers with entries from 1 to n sorted. This dataset when compressed using GPU-DFOR uses 1.8 bits per integer vs 7.8 bits per integer used by GPU-FOR. The runtime of GPU-DFOR is still $2\times$ slower than GPU-FOR as it still does the same number of operations in shared memory. However, when used in a larger kernel, GPU-DFOR is faster than GPU-FOR as shared memory will likely not be the bottleneck in the larger kernel. We will discuss the performance characteristics in greater detail in Section 5.6.3.

5.5 Database Integration

Given the efficient massively parallel bit-unpacking implementations described in the previous sections, we were naturally interested in its usability in a full system. As a proof of concept, we implemented the decompression routines as CUDA device func-

```
1 // Implements SELECT y FROM R WHERE y > v
2 // NT => NUM_THREADS
3 // IPT => ITEMS_PER_THREAD
4 template<int NT, int IPT>
  __global__ void Q(int* y, int* out, int v, int* counter) {
     int tile_size = get_tile_size();
6
     int offset = get_tile_offset();
\overline{7}
     __shared__ struct buffer {
8
       int col[NT * IPT];
9
       int out[NT * IPT];
10
     }:
11
     int items[IPT];
12
     int bitmap[IPT];
13
     int indices[IPT];
14
15
     BlockLoadInt<NT, IPT>(col+offset,items,buffer.col,tile_size);
16
     BlockPredIntGT<NT, IPT>(items, buffer.col, cutoff, bitmap);
17
     BlockScan<NT, IPT>(bitmap,indices,buffer.col,
18
       num_selections,tile_size);
19
20
     if(threadIdx.x == 0)
21
       o_off = atomic_update(counter,num_selections);
22
23
     BlockShuffleInt<NT, IPT>(items, indices, buffer.out);
24
     BlockStoreInt<NT, IPT>(buffer.out,out + o_off,num_selections);
25
26 }
```

Figure 5-7: Query Q0 Kernel Implemented with Crystal

tions¹ and show how they can be used with an existing GPU analytical engine. In particular, we chose Crystal [70], an open-source GPU analytics framework developed recently.

Crystal is a library of templated CUDA device functions that implement the full set of primitives necessary for executing typical select-project-join-aggregation (SPJA) analytical queries. Crystal is based on the idea of a *tile-based execution model*. In such a model, instead of viewing each thread as an independent execution unit, a thread block is viewed as the basic execution unit with each thread block processing a tile of entries at a time. A tile is simply a collection of $NT \times IPT$ elements where NT is the number of threads in a threadblock and IPT is the number of items per

 $^{^1\}mathrm{Device}$ functions are functions that can be called from kernels on the GPU

thread. Previous work [70] has shown that SQL query operators and SPJA queries implemented with Crystal can saturate memory bandwidth and thereby deliver anorder-of-magnitude speedup compared to CPU-based implementations.

The pseudo code in Figure 5-7 shows how the following example selection query is implemented in Crystal.

QO: SELECT x FROM R WHERE y > v;

The pseudo code uses the following block functions: BlockLoad loads a tile of data from global memory into the thread block (line 16). BlockPred applies the predicate to the tile and generates a bitmap (line 17). BlockBitmapLoad selects data from the tile using the bitmap (line 18). BlockScan implements hierarchical parallel prefixsum within the tile (line 19). The atomic update in line 23 determines the offset to which to write the matched results in the output array. BlockShuffle reorders the selected items into a contiguous array (line 25). Finally, BlockStore stores data into the output array (line 26). The modular nature of Crystal allows users to write high performance kernel code easily, reduces boilerplate code and makes it easy to use non-trivial functions.

We have implemented the decompression routines for GPU-FOR and GPU-DFOR as CUDA device functions LoadBitPack and LoadDBitPack respectively. These functions can be used in queries implemented in Crystal easily and can be used more broadly in any CUDA kernel as well. To integrate them into Crystal, the only required changes are to replace the load routines in Figure 5-7 with LoadBitPack. Below are the changes involved to make the kernel operate on bit-packed data:

```
1 ...
2 LoadBitPack<NT, IPT>(y.col, y.block_start,items,buffer.col,tile_size);
3 ...
4 LoadBitPack<NT, IPT>(y.col, y.block_start,items,buffer.col,tile_size);
5 ...
```

In this case IPT is the same as D, the number of blocks processed per threadblock. As can be seen from the example, the LoadBitPack device function encapsulates all the complexity and hides it from the end user. The user can run the query on compressed data by just changing a single line of code.

One key drawback of bit-packed data is that it lacks random access. Accessing any element requires loading the entire data block. As a result, when selections or joins filter data entries, we still have to read the entire column. As we show in the next section, this does not lead to material impact on performance because: 1) granularity of access from global memory is 128B, as a result random accesses are less beneficial in the first place and 2) analytics queries are mostly scan oriented, hence reduction in data size reduces the total data read which often compensates for loss of efficiency in case of a selective filter. Note that this would not work well for OLTP workloads which are characterized by point accesses. GPUs' are in general not suited and not used for OLTP workloads.

Since the routines LoadBitPack and LoadDBitPack are ordinary device functions, they can be used directly in user's CUDA code in conjunction with other GPU frameworks like Thrust [11] and they can also be called directly from NVVM (a compiler internal representation based on LLVM IR designed to represent GPU compute kernels) [8].

5.6 Evaluation

In this section, we compare the performance of the 4 main compression schemes used to store columns on the GPU:

- None: Data is stored as 4-byte integers.
- NSF: Null suppression with fixed length encoding. The entire array is encoded as 1, 2 or 4 byte entries depending on the maximum number of bits needed for any integer in the column.
- GPU-FOR: Using bit-packing to compress the data following the algorithm discussed in Section 5.3.

• GPU-DFOR: Using delta encoding and bit-packing to compress the data following the algorithm discussed in Section 5.4.

In addition, on two microbenchmarks we also compare the performance of two more compression schemes:

- RLE: Represents runs of the same value as a pair: (value, run-length). Values and run lengths are stored in two separate columns.
- NSV: Represents each value with a variable number of bytes (1,2,3 or 4). In a separate array it maintains the number of bytes used using 2 bits per value. This scheme is used to handle skew.

GPU-FOR and GPU-DFOR are novel to this work and we use the decompression routines discussed previously to decode them. The rest of the compression schemes are from past works [28, 46, 9] and we use improved versions of decompression routines proposed by them. For each compression scheme, we report three different metrics: **Compression Rate:** The average number of bits required for each integer after compression is applied (bits/int).

Aggregation Time: The time to load the compressed data from global memory, apply the decompression, sum the values, and store the total as 8-byte entry in global memory. We measure the raw throughput of the decompression algorithm in isolation. Decompression Time: The time required to load the compressed data from global memory, apply the decompression algorithm, and store the uncompressed data to global memory. This runtime is more reflective of the performance of the decompression algorithm when it is part of a larger kernel.

The rest of the section is organized as follows: we discuss the setup in Section 5.6.1. In Section 5.6.2 we evaluate the performance of the algorithms on synthetic dataset with varying bitwidths. In Section 5.6.3, we evaluate the impact of different data distributions. In Section 5.6.4, we evaluate the impact on performance on the Star Schema Benchmark. In Section 5.6.5, we discuss the case when GPU is used as a coprocessor. Finally, in Section 5.6.6, we discuss miscellaneous topics.

5.6.1 Setup

For the experiments, we use a virtual machine instance that has an Nvidia V100 GPU which is connected to the CPU via PCIe3. The Nvidia V100 GPU has 32 GB of HBM2 memory. The global memory read/write bandwidth is 880 GBps. The bidirectional PCIe transfer bandwidth is 12.8 GBps. The system is running on Ubuntu 16.04 and the GPU instance uses CUDA 10.0. In our evaluation, we ensure that data is already loaded into the GPU memory before experiments start. We run each experiment 3 times and report the average measured execution time.

5.6.2 Performance with Varying Bitwidths

We generate 15 unsorted datasets each with 250 million entries, such that all data elements in the *i*-th dataset have exactly *i* effective bits, i.e., the value range is $[2^{i-1}, 2^i)$ for i = 2, 4, ..., 30. Within these ranges, the values are uniformly distributed.

Figure 5-8 (a-c) shows the results for the four compression algorithms. The bitpacking schemes achieve the finest possible granularity and thus can perfectly adapt to any bit width. Consequently, the compression rate is a linear function of the bitwidth. The overhead for GPU-FOR is 0.75 bit per int (1 block start word + 1 reference word + 1 bitwidth word per block of 128 integer entries). The overhead for GPU-DFOR is 0.81 bit per int (0.75 + 1 first value word per D = 4 blocks). As the data is not sorted, the deltas vary in the range $[-2^i, 2^i)$ and require one additional bit; our experiments below show the benefit of GPU-DFOR on sorted data.

Figure 5-8b shows the aggregation time for the different schemes. The performance of NSF is a staircase pattern where the runtime is based on whether the entry size is 1, 2, or 4 byte. None and NSF saturate memory bandwidth. The performance of GPU-FOR is close to that of operating on uncompressed data. GPU-FOR does a significant amount of computation per entry which is roughly constant across varying bitwidths, compared to NSF which does a single copy followed by aggregation. Hence, in isolation GPU-FOR's performance does not vary much across bitwidths. GPU-FOR achieves bandwidth of 700 GBps which is close to the max memory bandwidth of



(c) Decompression Time

Figure 5-8: Performance of the different compression algorithms on uniform data with varying bit widths

880 GBps. The performance of GPU-DFOR is bound by shared memory bandwidth. Hence, in isolation, GPU-DFOR performs worse than the other schemes for this experiment.

In Figure 5-8c, we can see that the decompression performance of the bit-packed schemes looks better as the compute load remains the same while we do one additional global memory operation (store) per integer. The performance of NSF is again a staircase pattern. GPU-FOR does slightly worse than NSF, however the worst case gap is 15% achieved at bitwidth 7. The gap is due to slightly larger data size and irregular access pattern associated with accessing the block_starts array used to find the block offsets in the data array. The performance of GPU-DFOR is comparable to GPU-FOR in this case. In general, when GPU-DFOR is used as part of a larger kernel, shared memory may not be a bottleneck. Hence the additional shared memory passes done in GPU-DFOR to decode the deltas may not impact the actual performance of the kernel.

5.6.3 Dependence on Data distributions

To test the robustness of the compression schemes, we test their performance using three distributions. For each distribution, we maintain the array size n fixed at 250 million entries. The distributions are as follows:

- D1: a sorted array where we vary the number of unique values from 4 to 2²⁸. Typically a table is sorted based on one column, which D1 is designed to resemble. For this distribution, we also compare against RLE.
- D2: a normal distribution with a standard deviation of 20 and mean varying from 64 to 2³⁰.
- D3: a Zipfian distribution with the exponent *alpha* characterizing the distribution varying from 1 to 5 (1 is least skewed, 5 is most skewed). D3 resembles dictionary encodings of tweets or text corpora where distribution of words follows Zipf's law. For this distribution, we also compare against NSV.



Figure 5-9: Comparison of compression schemes on different data distributions

The results for D1 can be found in Fig. 5-9 (a-c). The bit-aligned algorithms GPU-FOR and GPU-DFOR achieve better compression ratios compared to None and NSF due to use of FOR. As the number of unique values increases beyond 2^{22} , the block of 128 integers is likely to have different values. As the dataset is sorted, GPU-DFOR can encode such cases with fewer bits compared to GPU-FOR. In the extreme case, when number of unique values equals 2^{28} i.e., each value is unique and the array is sorted, GPU-DFOR encodes the data with just 1.8 bits per int vs 7.8 bits per int used by GPU-FOR. The performance of GPU-FOR and GPU-DFOR (Fig. 5-9 (b-c)) is bound by shared memory bandwidth which results in a gap in performance being smaller than the gap in compression rate in comparison to the other two schemes. **RLE** achieves better compression rates compared to the bit-aligned algorithms when the number of distinct values is less than 2^{22} , beyond that GPU-DFOR does better. The key issue with RLE is that it is $3 \times$ slower to decompress compared to the bit-aligned schemes (Fig. 5-9(c)). Decompressing RLE is a 4-step process, which even after optimizations is similar to GPU-FOR making 4 passes over an array of size n. RLE decoding cannot be inlined with query execution and requires an additional memory buffer array of size n for storing intermediates. Interested readers can refer to [28] for the decompression algorithm. GPU-DFOR achieves better performance across the entire range compared to RLE and is competitive in terms of compression rate — hence it is a better choice.

For D2 (Fig. 5-9 (d-f)), we can make the same general observations. When using GPU-FOR/GPU-DFOR, each block's entries generally lies within 3 standard deviations of the mean and occasional occurrence of a value outside this range does not move the compression rate significantly. For mean greater than 2^{16} , the bit-aligned schemes achieve $3 \times$ reduction in storage footprint compared to the other schemes and show-cases the use of FOR.

For D3 (Fig. 5-9 (g-i)), we see that the bit-aligned schemes can adapt to change in skew and achieve both better compression rate and lower runtime compared to NSF and NSV. NSV is better at adapting to skew compared to NSF, however its performance is significantly worse compared to all the other schemes. Decoding NSF suffers from the same issues that affect RLE, it requires multiple steps that lead to multiple reads and writes, the decoding can't be inline with query execution and it requires buffer space for intermediates. The bit-aligned schemes are superior to NSV across all metrics.

5.6.4 Performance on SSB

For the full query evaluation, we use the Star Schema Benchmark (SSB) [57] which has been widely used in various data analytics research studies [31, 46, 78, 83]. SSB is a simplified version of the more popular TPC-H benchmark. It has one fact table *lineorder* and four dimension tables *date, supplier, customer, part* which are organized as a star schema. There are a total of 13 queries in the benchmark, divided into 4 query flights. In our experiments we run the benchmark with a scale factor of 20 which will generate the fact table with 120 million tuples. The total dataset size is around 13GB.

Figure 5-10 shows the column sizes after compression using the different encodings. Between GPU-FOR, GPU-DFOR, and NSF, GPU-FOR achieves the lowest storage footprint for all columns except lo_orderkey on which GPU-DFOR does better. As discussed before, GPU-DFOR does better on sorted columns. In total, using GPU-FOR achieves a



Figure 5-10: Compression waterfall for Star Schema Benchmark columns



Figure 5-11: Performance on Star Schema Benchmark queries with compressed columns

37% reduction in data size compared to NSF and 52% reduction compared to None. In additional to the schemes above, we also added in GPU-FOR+GZ, which represents using GPU-FOR (or GPU-DFOR whichever is smaller) followed by gzip. A similar scheme (bit-packing + FOR + gzip) is used by default to encode all integer columns in Apache Parquet [2, 10], a common columnar storage format used in disk-based column stores. We can see from the figure that GPU-FOR achieves most of the compression gains achievable with GPU-FOR+GZ leading to only an additional 14% reduction in storage space compared to GPU-FOR. Gzip works well when there are repetitions in the data. Hence, columns like **orderkey** benefit from gzip as after applying the frame of reference operation of GPU-FOR; there are repeated strings in the binary data that can be efficiently compressed by gzip.

For the runtime comparison, we compare the performance of Crystal with None and NSF encoding against Crystal with the decompression routines for GPU-FOR. We also compare against OmniSci, a commercial GPU-based OLAP DBMS. Figure 5-11 shows the runtime comparison of different compression schemes. OmniSci does the worst as it does not use the tile-based execution model and instead operates each thread independently. Crystal-based schemes perform significantly better. Previous work [70] has shown that queries implemented with Crystal achieve a theoretical lower bound for query runtime derived from the fact that memory bandwidth is saturated. Among all the data encoding schemes, NSF achieves the best performance. However, the gap between NSF and GPU-FOR is < 10% for queries q2.1 .. q4.3. This is because these queries have multiple joins and the runtime is dominated by random accesses into hash tables.

Queries q1.1, q1.2, q1.3 are selection queries and have no joins. These queries follow the same template with q1.3 more selective than q1.2 which is more selective than q1.1. The runtime of GPU-FOR is a constant across queries as it does not support random accesses of entries. Both NSF and None are able to skip cache lines when the query is very selective. For these queries, NSF does best. However, the gap between the NSF and GPU-FOR on an absolute scale is less than 1 ms. Comparing the geometric means of runtime across the entire workload, GPU-FOR is 4% slower than NSF while having 37% smaller storage footprint.

5.6.5 GPU as a Coprocessor

Many systems use the GPU strictly as a coprocessor [31, 83, 39]. These systems move data from CPU to GPU across an interconnect like PCIe when processing every query. In this setting, the compression schemes discussed in this chapter are equally beneficial as the runtime is bound the time taken to ship data over the interconnect (transfer time). GPU-FOR/GPU-DFOR achieve the best compression rate across a variety of data distributions and using them would reduce the amount of data moving across the slow PCIe bus thereby reducing transfer time. To evaluate this, we ran the SSB q1.1 with data stored on the CPU and the columns encoded using GPU-FOR and compared against using NSF and None. The query runtime with None, NSF, and GPU-FOR are 154ms, 96ms, and 63ms respectively. The query runtime is bounded by the time taken for data transfer over PCIe. Query runtime with GPU-FOR is 34% lower
than with NSF.

5.6.6 Discussion

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

Choice of Compression Scheme: As shown in the SSB benchmark and in the microbenchmarks, the performance of GPU-FOR and GPU-DFOR is competitive with NSF when used as part of a larger kernel. As a result, the rule of thumb is to simply use the compression scheme that has the lowest storage footprint for each column independently. This means using GPU-FOR on all columns except sorted / semi-sorted columns like the primary and secondary sort keys for which GPU-DFOR would generate a more compact representation.

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

Compression Speed: Data compression is a one-time activity that happens on the CPU side. GPU-FOR and GPU-DFOR compression can be done efficiently on the CPU. On a machine where maximum memory bandwidth achievable on a single core is 25GBps, GPU-FOR and GPU-DFOR compression algorithms achieve bandwidth of 16GBps and 14GBps respectively.

5.7 Conclusion

GPU-based analytical systems have demonstrated significant speedups over main memory databases. The key constraint in these systems is the limited GPU memory capacity. This chapter presents two efficient massively parallel implementations of bit unpacking routines: GPU-FOR and GPU-DFOR. Together these schemes achieve a 37% reduction in storage footprint on SSB compared to existing schemes with almost no impact on performance. These results show that our algorithms make it practical to use bit-packing for compression on GPU for the first time.

Chapter 6

Conclusion and Future Work

In this thesis, we proposed the tile-based execution model for efficiently executing queries on the GPU. We presented Crystal, a library of block-wide functions that can be used to implement a wide variety of SQL queries on GPU. The crystal library is modular, extensive and highly optimized for the massively parallel SIMT architecture of GPUs.

We presented theoretical models for query performance on GPUs and CPUs assuming memory bandwidth is saturated. We showed that query implementations using Crystal achieve the theoretical minimum runtime predicted by the models and hence are optimal. We use the models to explain the performance difference between query execution on a GPU vs on a CPU. Our analysis on a popular analytics benchmark shows that using modern GPU vs a CPU can lead to a runtime gain equal to $1.5\times$ the bandwidth ratio of GPU to CPU ($25\times$ in our setup) and be $4\times$ more cost effective than CPUs. This makes a strong case for using GPUs as the primary execution engine when the working set fits into GPU memory.

Finally, we used the idea of tile-based execution to develop massively parallel variants of two classic sequential algorithms: top-k and bit-packing based compression. We presented a new algorithm based on bitonic sort for finding top-k entries called Bitonic Top-K. The bitonic top-k algorithm is up to a factor of $15\times$ faster than sort and $4\times$ faster than a variety of other possible implementations for small k values. For compression, we presented the GPU-FOR and the GPU-DFOR lossless compressed storage formats for storing data columns on the GPU, and decompression routines to decompress them on-the-fly during query execution. Together these schemes achieve a 37% reduction in storage footprint on a standard analytics workload compared to existing schemes with almost no impact on performance.

The next sections suggest interesting directions for future work on GPU data analytics.

6.1 Multi-GPU Query Execution

Our work focused on optimizing query execution on GPUs. Today it is possible to attach multiple GPUs (upto 32) onto a single host machine. This allows the system to use the several hundreds of gigabytes of HBM available to cache large parts of the working set directly on the GPU. Partitionable workloads, like those executed on a star schema, can be adapted to run on a multi-GPU setup by partitioning the fact table across the multiple GPUs and maintaining a copy of the dimensions tables on each GPU. Once the data is partitioned, we can adapt existing query implementation with Crystal to run on the multi-GPU setup by invoking one kernel per partition in parallel from the CPU. In the end, the aggregate results (which are usually small) can be merged on the CPU. As the invocations happen in parallel, performance scales linearly with the number of GPUs.

There are two issues with this approach: 1) we have to maintain a copy of the all dimension tables on each GPU, which can be wasteful and 2) workloads aren't always perfectly partitionable and may require data shuffling to compute the query result. The conventional way of transferring data across GPUs involves moving data through the host CPU. To move data from GPU₁ to GPU₂, we would first do a GPU₁ device to host CPU transfer (D-to-H) and then a host CPU to GPU₂ device transfer (H-to-D). We end up with two transfers over the PCIe bus which is quite slow. More recently, NVLink [7] and NVSwitch [7] interconnects enable fast GPU-to-GPU communication (D-to-D).

The Nvidia NVLink is a GPU-to-GPU interconnect that is significantly faster



Figure 6-1: NVIDIA A100 with NVLink GPU-to-GPU connections

and more energy-efficient than PCIe. Latest generation NVLink achieves 600GBps of bandwidth spread across 12 links. Figure 6-1 shows a configuration of 4 Nvidia A100 connected using NVLink. In the above configuration, each GPU pair has bidirectional bandwidth of 200GBps which is an order of magnitude more than PCIe bus bandwidth of 13GBps. Note that while NVLink is fast, the NVLink interconnect bandwidth is lower than the GPU memory bandwidth on A100 GPU of 1500 GBps. The Nvidia NVSwitch is an alternative interconnect available on the high-end DGX systems that enables all-to-all GPU communication between a group of 16 GPUs at 600GBps bandwidth.

The high interconnect bandwidth can be used to address both of the issues described earlier. The dimensions tables do not have stored on each GPU. CUDA supports direct memory accesses to memory on another GPU, hence the data columns can be accessed on demand. The high bandwidth reduces the overhead of doing shuffles for joins and sorts across multiple GPUs.

The multi-GPU architecture leads to interesting challenges around orchestrating query execution across GPUs, multi-tenancy and data placement which could make for interesting future work.

6.2 Heterogeneous Computing

This thesis shows that query execution on GPUs leads to significant speed-up vs using CPUs. However, when the data is large and used less frequently, CPUs are still a viable alternative because of their better Performance/\$ ratio.

When evaluating queries with multiple joins, each join (when using hash join) involves probing a hash table (built on entries from the dimension table) based on entries from the fact able. This step is referred to as the probe step and is usually the bottleneck in analytical workloads like the TPC-H, TPC-DS or Star Schema Benchmark. The probes are expensive as each probe into the hash table is in essence a random accesses that ends up fetching an entire cache line from memory.

In recent years, there has been an effort by hardware manufacturers to increase the CPU-GPU interconnect speed. Today, IBM POWER9 CPU supports NVLink interconnect to Nvidia GPUs. This interconnect has bandwidth of 75GBps, significantly higher than PCIe bandwidth. NVLink also gives the GPU direct access to pageable CPU memory. GPU load, store, and atomic operations are translated into CPU interconnect commands by the NVLink Processing Unit (NPU).

The fast interconnect coupled with increasing memory capacity of GPUs could be used to accelerate query performance vs a CPU-only DBMS even on workloads whose working set is larger than GPU memory. Recent work by Lutz et al. [48] showed $4\times$ join performance speed-up using the hybrid system with NVLink interconnect and data stored on the CPU vs optimized CPU-only implementations. The work uses the GPU to execute a hash join. The GPU first loads the build table via the interconnect and builds the hash table in GPU memory. Then the fact table is loaded via the interconnect and we probe the hash table previously built. In this implementation, all the random access associated with the hash join happen in the fast GPU memory instead of CPU memory, thereby resulting in significant speedup.

In the coming years we expect database systems to be designed for heterogeneous processing on hybrid systems. Designing heterogeneous systems that utilize efficiently all available resourcess on CPU and GPU presents numerous challenges including cost modelling, query scheduling, data placement, etc which could make for interesting future work.

Appendix A

Per-Thread Top-K Using Registers

Registers are the fastest layer of the memory hierarchy. However, as noted in Section 2.1, current generation GPUs do not have thread-local memory. A thread-local array can be made to use registers only if all its accesses are statically known. Without it, the compiler is forced to allocate the array in local memory which is off-chip and has a severe negative impact on performance. This prevents us from implementing a heap using registers as array accesses made during heap updates cannot be statically determined. We found that we can still maintain top-k per thread in registers by maintaining a list of top-k seen so far as a list and, keeping the index and value of the minimum value.

T buf[k]; T minValue; int minIndex;

If the element seen is greater than *minValue*, we update *minIndex* and find the new *minIndex*, *minValue* as follows:

```
minValue = xi
for j in range(0,k):
    if j == minIndex: buf[j] = xi
    if buf[j] < minValue:
        minIndex, minValue = j, buf[j]</pre>
```



Figure A-1: Different Per-Thread Top-K Approaches

While iterating over the elements of the buffer array creates overhead in the order of k, it allows the compiler to place the elements of *buf* in registers. The faster data accesses counteract the overhead for low values of k. For high values of k, the limited number of available registers forces the compiler to allocate some of the entries of *buf* in local memory even if the access is implemented in the manner described.

Figure A-1 compares the time taken by the register-based version to the shared memory-based version to find the top-k from 2^{29} floating point numbers with varying k. We vary the distribution: (a) *Uniform*: numbers drawn from a uniform distribution U(0, 1), (b) *Increasing*: numbers from U(0, 1) sorted increasing and, (c) *Decreasing*: numbers from U(0, 1) sorted decreasing.

The register-based top-k is slower than the equivalent shared-memory based topk method for larger k because the register-based method starts spilling registers to local memory, which leads to significant slowdown. This is evident in the sharp slope going from k = 32 to k = 64 in the graph. Comparing the *increasing* and *decreasing* distribution, we see that the gap between the methods widens. This is because *increasing* has every number updating the top-k. Updates are more expensive in the list compared to the heap. In *decreasing*, there are no heap updates after inserting the first k elements.

Bibliography

- [1] 1000X faster data exploration with GPUs. https://www.omnisci.com/blog/ mapd.
- [2] Apache Parquet. https://parquet.apache.org/.
- [3] Arrayfire discussion on top-k. http://bit.ly/2lLuFS1.
- [4] BlazingDB. https://blazingdb.com.
- [5] Issue to add gpu verion of top-k to tensorflow. https://github.com/ tensorflow/tensorflow/issues/5719.
- [6] Kinetica. https://kinetica.com/.
- [7] NVLink and NVSwitch Advanced Multi-GPU Systems. https://www.nvidia. com/en-us/data-center/nvlink/.
- [8] NVVM IR. https://docs.nvidia.com/cuda/nvvm-ir-spec/.
- [9] OmniSci. https://omnisci.com.
- [10] Parquet Encoding Format. https://github.com/apache/parquet-format/ blob/master/Encodings.md.
- [11] Thrust. https://thrust.github.io/.
- [12] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006* ACM SIGMOD international conference on Management of data, pages 671–682. ACM, 2006.
- [13] Martín Abadi et al. Tensorflow: A system for large-scale machine learning. In OSDI, 2016.
- [14] Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. GGKS: Grinnell GPU k-selection. http://code.google.com/p/ggks/, 2010.
- [15] Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 2012.

- [16] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. Multicore, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.
- [17] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Mainmemory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373. IEEE, 2013.
- [18] Kenneth E Batcher. Sorting networks and their applications. In Proceedings of the spring joint computer conference, 1968.
- [19] 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.
- [20] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989.
- [21] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyperpipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.
- [22] Peter Alexander Boncz et al. Monet: A next-generation DBMS kernel for queryintensive applications. Universiteit van Amsterdam [Host], 2002.
- [23] Sebastian Bre
 ß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In SIGMOD. ACM, 2016.
- [24] 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.
- [25] 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. *PVLDB*, 2008.
- [26] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In Acm Sigmod Record, volume 14, pages 268–279. ACM, 1985.
- [27] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.
- [28] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. PVLDB, 3(1-2):670–680, 2010.

- [29] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. ACM Sigmod Record, 40(4):45–51, 2012.
- [30] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the* 2015 ACM SIGMOD International Conference on Management of Data, pages 31-46, 2015.
- [31] 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, pages 1603–1618. ACM, 2018.
- [32] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*, pages 370–379. IEEE, 1998.
- [33] Naga Govindaraju et al. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.
- [34] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In SIGMOD, 2006.
- [35] Mark Harris. Optimizing cuda. SC07: High Performance Computing With CUDA, 2007.
- [36] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. GPU gems, 3(39):851–876, 2007.
- [37] 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, 2008.
- [38] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *PVLDB*, 2013.
- [39] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 2013.
- [40] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. Highperformance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.
- [41] David A Huffman. A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9):1098–1101, 1952.

- [42] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A survey of top-k query processing techniques in relational database systems. *CSUR*, 2008.
- [43] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In DaMoN, 2012.
- [44] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively parallel numa-aware hash joins. In *In Memory Data Management and Analysis*, pages 3–14. Springer, 2015.
- [45] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.
- [46] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.
- [47] Yinan Li and Jignesh M Patel. Bitweaving: fast scans for main memory data processing. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pages 289–300, 2013.
- [48] 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 2020 ACM SIGMOD International Conference on Management of Data, pages 1633–1649, 2020.
- [49] James Malcolm et al. Arrayfire: a gpu acceleration platform. In SPIE, 2012.
- [50] Stefan Manegold, Peter A Boncz, and Martin L Kersten. Optimizing database architecture for the new bottleneck: memory access. *Proceedings of the VLDB Endowment*, 9(3):231–246, 2000.
- [51] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 2016.
- [52] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.
- [53] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.
- [54] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 2011.
- [55] Todd Mostak. An overview of mapd (massively parallel database). White paper, Massachusetts Institute of Technology, 2013.

- [56] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. Proceedings of the VLDB Endowment, 4(9):539–550, 2011.
- [57] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference* on Performance Evaluation and Benchmarking, pages 237–252. Springer, 2009.
- [58] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with cuda based on bitonic sort. *Parallel Processing and Applied Mathematics*, pages 403–410, 2010.
- [59] Holger Pirk, Stefan Manegold, and Martin Kersten. Waste not... efficient coprocessing of relational data. In *ICDE*. IEEE, 2014.
- [60] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo-a vector algebra for portable database performance on modern hardware. *PVLDB*, 2016.
- [61] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIG-MOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.
- [62] Orestis Polychroniou and Kenneth A Ross. A comprehensive study of mainmemory partitioning and its application to large-scale comparison-and radixsort. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014.
- [63] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.
- [64] Kenneth A Ross. Selection conditions in main memory. ACM Transactions on Database Systems (TODS), 29(1):132–161, 2004.
- [65] Eyal Rozenberg and Peter Boncz. Faster across the pcie bus: a gpu library for lightweight decompression: including support for patched compression schemes. In DaMoN. ACM, 2017.
- [66] 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.
- [67] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing*, 2009. IPDPS 2009. IEEE International Symposium on, pages 1–10. IEEE, 2009.

- [68] 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 2010 ACM SIGMOD International Conference on Management of data*, pages 351–362. ACM, 2010.
- [69] 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.
- [70] Anil Shanbhag, Xiangyao Yu, and Samuel Madden. A study of the fundamental performance charecteristics of gpus and cpus for database analytics. In Proceedings of the 2020 International Conference on Management of Data. ACM, 2020.
- [71] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. Technical report, 2019.
- [72] Evangelia A Sitaridi and Kenneth A Ross. Ameliorating memory contention of olap operators on gpu processors. In *Proceedings of the Eighth International* Workshop on Data Management on New Hardware, pages 39–47. ACM, 2012.
- [73] 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, page 4. ACM, 2013.
- [74] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop* on Data Management on New Hardware. ACM, 2011.
- [75] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In SIGMOD. ACM, 2017.
- [76] Elias Stehle and Hans-Arno Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. In SIGMOD. ACM, 2017.
- [77] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.
- [78] 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.
- [79] Jan Wassenberg and Peter Sanders. Engineering a multi-core radix sort. In *European Conference on Parallel Processing*, pages 160–169. Springer, 2011.

- [80] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: ultra fast in-memory table scan using onchip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.
- [81] 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. IEEE, 2012.
- [82] 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.
- [83] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proceedings of the VLDB Endowment*, 2013.
- [84] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.
- [85] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ramcpu cache compression. In 22nd International Conference on Data Engineering (ICDE'06), pages 59–59. IEEE, 2006.