GPU-Acceleration of In-Memory Data Analytics

Evangelia Sitaridi

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

ABSTRACT

GPU-Acceleration of In-Memory Data Analytics

Evangelia Sitaridi

Hardware advances strongly influence the database system design. The flattening speed of CPU cores makes many-core accelerators, such as GPUs, a vital alternative to explore for processing the ever-increasing amounts of data. GPUs have a significantly higher degree of parallelism than multi-core CPUs but their cores are simpler. As a result, they do not face the power constraints limiting the parallelism of CPUs. Their trade-off, however, is the increased implementation complexity. This thesis adapts and redesigns data analytics operators to better exploit the GPU special memory and threading model. Due to the increasing memory capacity and also the user's need for fast interaction with the data, we focus on in-memory analytics.

Our techniques span different steps of the data processing pipeline: (1) Data preprocessing, (2) Query compilation, and (3) Algorithmic optimization of the operators. Our data preprocessing techniques adapt the data layout for numeric and string columns to maximize the achieved GPU memory bandwidth. Our query compilation techniques compute the optimal execution plan for conjunctive filters. We formulate *memory divergence* for string matching algorithms and suggest how to eliminate it. Finally, we parallelize decompression algorithms in our compression framework *Gompresso* to fit more data into the limited GPU memory. Gompresso achieves high speed-ups on GPUs over multi-core CPU state-of-the-art libraries and is suitable for any massively parallel processor.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my gratitude to my advisor Ken Ross for his guidance during my years at Columbia University. Ken always gave great insights during our research discussions because of his broad and deep knowledge of Database Systems and his sharp intellect. He inspired my interest in Hardware Accelerated Databases and I consider myself fortunate to have an advisor that always had time for discussion. I am also thankful for his understanding during the rewarding but also the harder times of my academic journey.

My sincere thanks go to Dr. Tim Kaldewey who was a very encouraging and hands-on mentor and collaborator sharing his valuable advice on all research, technical and personal levels. During my internships at the IBM Almaden Research Center, I had the chance to work with him, Dr. Rene Mueller and Dr. Guy Lohman. I consider myself lucky to collaborate on a project with them, an experience that I found genuinely exciting and I feel I gained valuable insights on real-world problems and on communicating my work more effectively. During my time in IBM Almaden, I also had the chance to interact with Dr. Ippokratis Pandis, who shared valuable advice on presenting my work in conferences.

I would like to thank Professor Luis Gravano for his resourceful comments and thought-provoking questions during the Database Group meetings and additionally for serving on my committee.

I am also grateful to Professors Mihalis Yannakakis and Eugene Wu for serving on my thesis committee. I appreciate the time they took out of their busy schedules to read through my thesis and share their knowledge and insights.

Many thanks deserve all the members of the Database Research Group of Columbia with whom I had the opportunity to overlap. I thank Bingyi, Fotis, Ioannis, Orestis, Pablo, Tom, and Wangda, for attending and giving valuable comments during my conference dry-run presentations. I especially thank Orestis Polychroniou, who was an excellent and inspiring

collaborator.

Professors Yannis Ioannidis and Alex Delis introduced me to the Database and the System research field during my B.Sc. and M.Sc. years. Their lectures sparked my interest in these two fields and I am grateful for their advice during my Ph.D. applications and studies.

The members of Computing Research Facilities of Columbia were extremely helpful when hardware failures came up. I am thankful to Daisy, Hitae, Jorge and Sean for their support.

I am grateful to the National Science Foundation (grants IIS-0915956 and IIS-1218222) for the continuous financial support during my studies. I would like to thank IBM for the Fellowship it awarded me during the fourth year of my Ph.D. I am also grateful for the hardware donations from NVIDIA, that provided the GPU hardware I used for my Experimental Evaluations. Finally, I would like to thank Onassis Foundation for the scholarship it awarded me for my Ph.D. research.

On a personal note, I would like to express my gratitude to my family and friends in Greece and USA for their continuous encouragement. The friends I met during my graduate years at Columbia: Anthie, Bingyi, Christian, Emilio, Georgia, John, Marios, Melanie, Moschoula, Orestis, and Theofilos were a great support system and I thank them for the fun moments we shared. My long-time friends Maria, Marialena, Marina, and Nisa kept me going during the hardest times and celebrated with me my aha! moments. My family members Eleni, Kostis, and Voula made my life more meaningful. I am, especially, forever indebted to my parents Despoina and Tasos for their generosity of spirit. They taught me while growing up the value of creative work and compassion. My thesis would not have been completed or even start without their unconditional love and support. One of my life purposes is repaying their love and dedication and living up to their moral standards.

To my parents, Despoina and Tasos

Chapter 1

Introduction

In the last two decades database systems faced new bottlenecks. The increase in memory sizes allowed small and moderately sized databases to fit in the RAM [Manegold *et al.*, 2000]. As a result, the main performance bottleneck shifted from disk access to memory access.

The increase of memory capacity creates new opportunities for businesses to take advantage of in-memory data analytics. In-memory data analytics speed up the performance of a wide range of applications, from data warehousing to real-time analytics.

In data warehousing, data is reviewed, aggregated and then processed. *Real-time analytics* involves up-to-the-minute fresh data [Cohen-Crompton, 2012]. Social media is a significant source for real-time analytics. Interactive data visualization is a powerful tool for a lot of industries [Tableau, 2016]. Twitter data can be used by journalists to generate visualizations related to important events, such as the elections. Visualizations of the most tweeted candidate capture election trends and enrich the content of an article. In-memory analytics for businesses provide more detailed reports by boosting database performance. Faster processing means that managers can understand customer behavior and eventually make high-value, fast and informed decisions. Regardless of the time sensitivity of the data, in-memory processing requires optimized implementations of key functions. In this thesis, we explore two key functions: Filtering and string matching predicates to explore interesting areas of the data.

Another shift in hardware performance was the flattening speed of single-core processors

that paved the way for the multi-core era. Parallelism is our main hope to meet the need for increasing performance on database workloads [Sutter, 2005]. To reach the maximum memory bandwidth of processors for in-memory workloads, fully parallel programs must be designed. There are different types of available parallelism that database systems exploit. *Thread-parallelism* involves the development of multi-threaded software partitioning a task into subtasks and concurrent threads executing the different subtasks in parallel. The main challenges in thread parallelism are load balancing to make sure all threads execute approximately the same amount of work and contention handling to avoid performance degradation because of resource sharing between threads. Database performance can be boosted by independent instructions in the code-path. Independent instructions can be evaluated in parallel because of the instruction pipelining in the CPUs. This potential overlap is called instruction level parallelism (ILP). For example, consider this pseudo-code snippet, applying a conjunctive filter on tuples of table R:

```
if(R.a < min_age)
  if(R.b < min_salary)
     add the tuple to results
```

The second if-statement instruction depends on the outcome of the first condition, limiting the ILP. Checking both conditions for every tuple, regardless of the outcome of the first condition, increases the ILP but also increases the memory traffic. The optimal execution depends on factors such as the memory latency and the condition selectivity. *Simultaneous multithreading* (SMT) on superscalar CPUs combines both thread-level and instruction-level parallelism by allowing instructions from multiple threads to be issued at a given cycle. *Data parallelism*, on the other hand, involves applying the same operation on different data elements. Challenges for data parallelism include transforming the control flow of the operators by removing branches so that the same code can be applied to different data elements. This optimization also results in increased ILP. Mainstream processors with SIMD instructions are the most common platforms for vectorized programs.

We are in the midst of a transition from the multi-core to the many-core era. While the number of transistors increases at the rate of Moore's law, the energy efficiency per transistor has been decreasing. Power constraints will eventually prevent all cores being

concurrently active [Esmaeilzadeh *et al.*, 2013]. Also, adding cores to a processor results in linear scaling only up to a certain number of cores. Many-cores have an especially high number of cores, which are typically simpler than the cores of traditional CPUs. Simpler cores do not have the same power constraints as more complex CPU cores. Many-cores trade-off ease of development for increased parallelism. GPUs and the Intel Xeon Phi are typical examples of many-core processors.

In this thesis, we focus on data processing powered by GPUs. GPUs are many-core processors with very fast memory suitable for memory bound database queries. In Section 1.1 we discuss our research motivation for GPU database processing to accommodate the increasing needs for lightning-fast processing. Section 1.2 describes the challenges posed by the special GPU processor architecture. In Section 1.3 we formulate the database setting and the problems we are tackling in the present thesis and Section 1.4 is our contribution statement.

1.1 Motivation

Hardware advances make hardware-aware software critical to avoid leaving database performance on the table [Breß *et al.*, 2014]. The exponential growth of data further pushes hardware acceleration for more efficient systems. GPUs have massive parallelism available and high memory bandwidth, matching the speeds of memory bound database workloads [Manegold *et al.*, 2000].

Integrating multiple CPUs can match the raw performance of a GPU processor but not the Performance per Watt. Energy efficiency of large scale data processing systems is becoming critical important with the increased availability of GPUs on the cloud. GPU instances on the cloud support the computing requirements of an increasing number of businesses without the upfront costs of dedicated server solutions[Amazon Web Services, 2016; Microsoft, 2016; MapD, 2016]. For the problems studied in this thesis, GPUs are always the most energy efficient option. Straightforward GPU implementations would not achieve performance speed-ups or energy savings against multi-core solutions but we leverage the GPU performance by adapting the input data layout in the GPU memory and the operator

algorithms.

As their memory capacity increases, GPUs are fit for in-memory processing of larger datasets. The NVIDIA K80 GPU has 24GB of available memory [NVIDIA, 2015e]. Our focus on GPUs is also supported by their increased commercial availability. Also, GPU programming interfaces are becoming more programmer-friendly so the learning curve for GPU parallel programming is now less steep [NVIDIA, 2016a].

1.2 GPU Architecture

This section gives a historical overview of the GPU architectural evolution. We then describe the main challenges that have to be tackled to leverage the performance of modern GPU processors.

1.2.1 GPU Historical Overview

GPUs were originally in the 1990s designed to offload 2D and 3D graphics rendering from the main CPU processor [Singer, 2013a]. The first GPUs did not have many cores, which were added later to be able to process multiple pixels in parallel. The term GPU was first officially used by NVIDIA during the launch of GeForce 256 [NVIDIA, 2016b]. GeForce 256 had a 32MB frame buffer, 5.312 GB/s memory bandwidth, and 220nm fabrication process [TechPowerUp, 2015]. The limitation of GPUs at the time was the limited programmability. While the OpenGL and DirectX APIs were being extended, the hardware had fixed functions and could not take advantage of the new API features.

To address that limitation, the next generation GPUs became programmable [Singer, 2013b]. At first, they only offered limited programmability with the use of *shaders*. Shaders were simple programs describing the traits of vertex or pixel data and replaced the component of the GPU hardware responsible for lighting and texture-mapping. The trend towards more extensive programmability of GPUs continued along with the available parallelism, which increased faster than Moore's law. The CUDA compute platform allowed GPU programming in a C-like language [NVIDIA, 2016a].

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 kernel execution [Glaskowsky, 2009]. Kernels were analogous to shaders in the previous GPU generations. Fermi also provided support for languages such as C++ and FORTRAN.

One of the bottlenecks in GPGPU computing has been the slow interconnection between the CPU and the GPU, which typically communicate through PCI Express (PCIe) links. Newer GPU generations will have faster links [NVIDIA, 2014] but the CPU-GPU interconnection speed will be still an order of magnitude slower than the device memory of the GPUs. For example, NVIDIA Tesla K80 has a memory bandwidth of 480 GB/s but the transfer bandwidth from and to the GPU is 16 GB/s in each direction [NVIDIA, 2015e]. NVLINK will be available on the most recent NVIDIA GPU architecture, Pascal. NVLINK is a faster and more energy-efficient alternative of PCIe reaching 80GB/s interconnection speed between the CPU and the GPU [NVIDIA, 2015d]. However, it will still present a potential bottleneck since Pascal GPU's memory bandwidth is projected to be 720 GB/s [NVIDIA, 2015d]. To eliminate this bottleneck, integrated GPUs were designed by AMD, integrating CPU and GPU cores on a single die [AMD, 2016]. However, their memory bandwidth is not nearly as fast as the bandwidth of a dedicated GPU. The high-end upcoming AMD Zen APU is projected to have 128GB/s memory bandwidth [Moammer, 2016]. Consequently, our general approach chose to focus on how to better exploit dedicated GPUs for database operators by storing the datasets in the GPU memory and minimizing the data transfers between the CPU and the GPU.

Based on the projected trends, the GPU available parallelism, and memory capacity will continue to improve [Moammer, 2015]. Newer memory technologies address the needs of memory-bound applications. 3D stacked RAM improves both capacity and speed by integrating multiple layers of DRAM components on the package along with the GPU [NVIDIA, 2015d].

Figure 1.1 shows the increase in memory speed and number of cores over recent years. GPUs are viewed as parallel vector processors with a special memory hierarchy designed

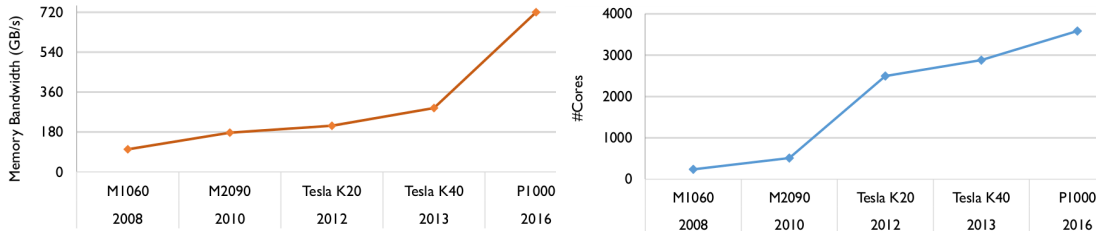


Figure 1.1: GPU Trends in memory bandwidth and number of cores.

for throughput rather than latency, unlike traditional CPUs.

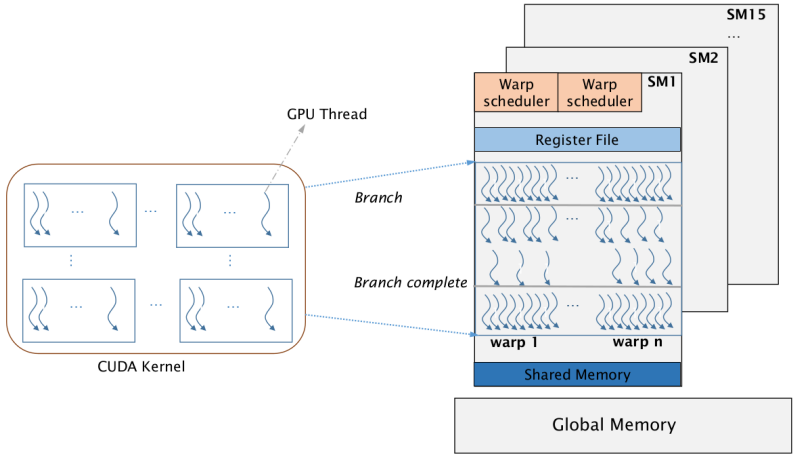


Figure 1.2: NVIDIA GPU Architecture overview.

1.2.2 GPU Architectural challenges

GPUs pose challenges to programmers trying to leverage their high performance. Figure 1.2 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[NVIDIA, 2016a; AMD, 2013].

A GPU has multiple streaming multi-processors (SM). The SM is the GPU component executing the GPU functions, called kernels. 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. Multiple warps can be organized in larger groups of threads, called *thread-blocks*. On the K40 GPU, there are 15 SMs, each capable of running

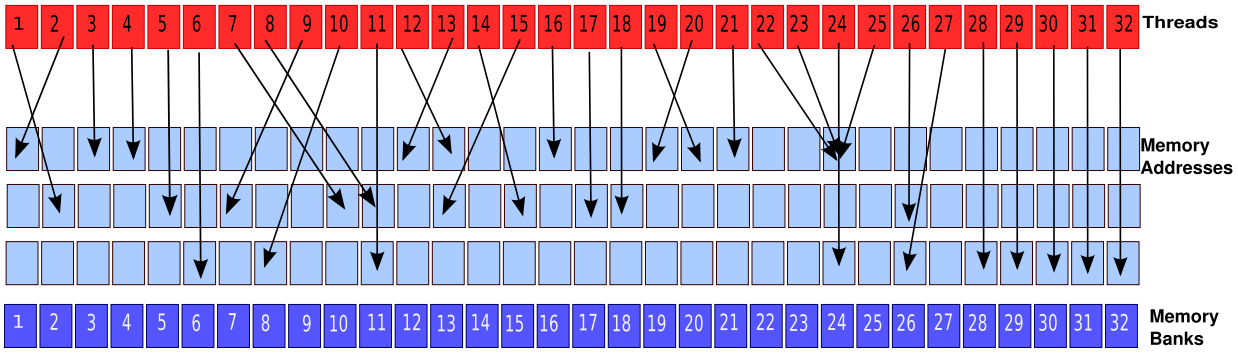


Figure 1.3: An example of bank/value access pattern for 32 threads in a warp.

64 concurrent warps.

Threads of 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. To accelerate algorithms including many branch instructions for GPUs, we have to rethink our approach by transforming the control flow of the code or designing an algorithm with fewer branches.

GPUs benefit significantly from instruction level parallelism because they can execute multiple instructions at a time if there are few or no instruction dependencies. Branches introduce instruction dependencies and consequently degrade execution efficiency. When few instructions depend on the outcome of a branch, a compiler would use *predication*. Predication allows each thread in a warp to either execute an instruction if the branch condition is true or do nothing (noop instruction) if the branch condition is false for this thread. Predication reduces the branch overhead and also increases the instruction level parallelism. However, depending on the instruction cost predication might increase the overall execution time. Our techniques can use additional workload information that is not available during compile time so it is complementary to the compiler optimizations.

GPUs have a radically different *memory hierarchy* from a traditional CPU [NVIDIA, 2015c]. *Global memory* is the largest type of memory, but it has high latency: 400–600

cycles. The scope of global memory is all GPU threads and it is cached in the L2 cache of the GPU, which is also shared among all threads. The L2 cache on a K40, which is shared by all processing elements has 12,288 128-byte cache lines.

Shared memory is used as a parallel, software controlled cache and its scope is a thread-block. Shared memory capacity is limited, on a K40 it is 16KB or 48KB depending on the kernel configuration. 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. Figure 1.3 illustrates a possible shared memory access pattern by the 32 threads in a warp. There are 2-way bank conflicts in banks 11, 13, 24, and 26. The three accesses to a single item by threads 22, 23 and 25 represent value conflicts. If these three accesses are reads, then there is no performance penalty; the system will broadcast the common data item to all three threads in one round. However, if these were write accesses, then two additional serialization rounds would be necessary for value conflicts.

Finally, *registers* are privately owned by each thread and store values that are used immediately.

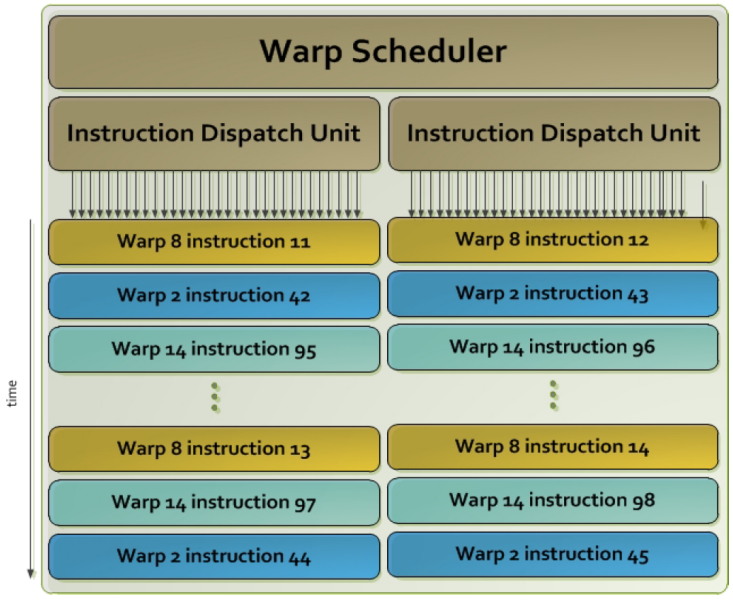


Figure 1.4: Thread scheduling in NVIDIA GPUs [NVIDIA, 2015c].

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. Figure 1.4 shows how instructions from different warps can be overlapped [NVIDIA, 2015c]. In the newer GPU generations, two instructions from each warp can be dispatched. When running at full capacity, there may be $15 \times 64 \times 32 = 30,720$ threads in flight on a K40. With many more threads than cache lines, any algorithm that tries to assign threads independent work is liable to thrash in the L2 cache if those threads each access even a single cache line. One possible optimization is reorganizing memory accesses so that threads access contiguous memory addresses. In that case, multiple memory accesses of the threads within a warp can be combined in few accesses from the global memory. This memory access pattern is called *memory coalescing*, and it achieves spatial data locality.

Compared to CPU processors, GPUs have limited memory capacity. Consequently, a single GPU can only fit small to moderate size databases. Data transfers between CPUs and GPUs can become a performance bottleneck in some workloads. For these workloads, we assume that only intermediate query results are exchanged between the CPU and the GPU, that are typically much smaller than the original data. To fit larger database workloads, we suggest a multi-GPU setting where the data is partitioned across different GPUs. For the case of compression workloads, where we show that data transfers is not the main bottleneck, we hide the data transfer latencies by using software pipelining.

CPU	GPU
Function	Kernel
Thread	Warp
SIMD lane	GPU thread
Branch prediction	Predication/Thread divergence
Low latency	High throughput
RAM	Global memory

Table 1.1: CPU-GPU analogies.

Finally, in Table 1.1 we summarize the analogies between CPUs and GPUs.

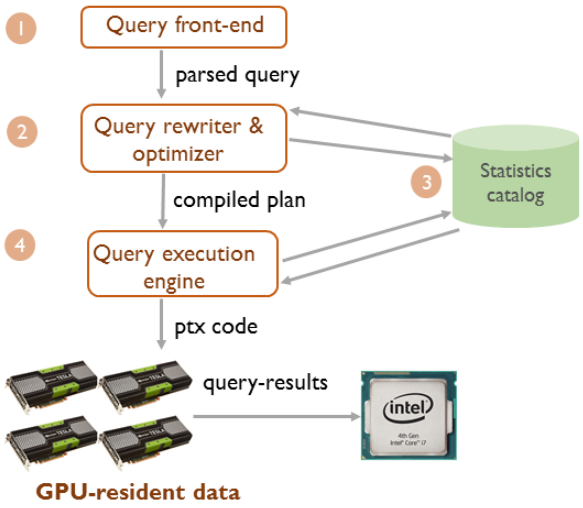


Figure 1.5: GPU Database system architecture.

1.3 Problem Setting and Context

In this chapter, we give a high-level overview of how our acceleration techniques and algorithms could be integrated into a full-fledged GPU database system. Our goal is to optimize query throughput and the query throughput per Joule. Based on our Experimental Evaluation, higher query throughput results in improved energy efficiency, so these goals are not contradicting.

Figure 1.5 shows the architectural components of a database system combining a traditional processor and an accelerator. As we describe in Section 4.5.1, we suggest executing a calibration step on a small sample of data to abstract the memory characteristics of the available GPU devices. The calibration must be executed when a new GPU is added to the system and update accordingly the query execution.

Physical Design We assume an in-memory setting. The input tables have numeric columns, string columns and potentially binary large objects (BLOBs). The string data is duplicated on both GPU device and the CPU RAM. A subset of the numeric columns is also duplicated on both the GPU and CPU RAM and the remaining data is stored in the CPU RAM. Determining which columns will be stored in the GPU device requires an analysis of

the workload to estimate which columns can benefit the most from GPU acceleration. We also analyze the workload to compute columns that are frequently aggregated, to reduce shared memory contention using our techniques described in Chapter 3.

The GPU resident data is loaded before the execution of the data analytics workload. During runtime, the data transfers between the CPU and the GPU only involve exchanging intermediate results. Our assumption is also in sync with the design of MapD, which is a database start-up doing real-time analysis on streaming data [Mostak and Graham, 2014]. The most frequently used data are stored in the GPU memory and only result-sets are exchanged between the GPU and the CPU.

Integer numeric columns are stored in a column-wise way. This matches the GPU architecture because different threads processing contiguous column values will coalesce their accesses to global memory, reducing the memory traffic. Column-stores are more suitable for read-intensive workloads accessing few columns of a large portion of the input tables. For string columns, our baseline layout stores the strings contiguously in memory. In Section 5 we evaluate different storage layouts and compare them against the baseline.

Columns that are larger, such as BLOBs, will be typically stored in a compressed format. Decompression performance can be more critical than compression performance because data will compressed only once but decompressed multiple times during system execution. In Chapter 7 we present our accelerated Gompresso framework, using GPUs to achieve 2X higher decompression performance than the multi-core performance of gzip.

In the case of updates, they are batched in the CPU, in a delta-store [Plattner, 2009], that is periodically merged with the GPU-resident data. Updates are typically only applied on the base data. Data is not indexed using typical row-store indexes, such as B-trees or hash-indexes. Our techniques could be combined with indexes, by querying the index structure and apply our selection optimization on the intermediate results returned by the index. Indexes, however, are not expected to improve the overall performance of the query workloads we study. Although there are efficient implementations of indexed key tree-search [Kim *et al.*, 2010], indexes are mostly beneficial for highly selective predicates while analytics queries scan fewer columns of a large portion of the involved tables. Also, indexing increases the maintenance cost when new table data are loaded in batches and space consumption

since they can be several times larger than the original data. Finally, indexes on GPUs are memory bound because of the irregular access pattern that results in cache thrashing. Efficient indexed search designs approach the compute bound limit but to avoid thrashing the throughput must be limited [Kim *et al.*, 2010].

There are, however, OLAP-friendly index structures, such as column zone maps or bitmap indexes. They are more space-efficient and at the same time can be used to accelerate queries selecting larger portions of data. Zone maps divide a table into continuous regions based on an attribute. These regions are called *zones* and record the minimum and the maximum attribute value in the zone [Graefe and Kuno, 2010]. Zone maps provide additional access paths, considered by the query optimizer since they allow to quickly skip over uninteresting zones of data for range queries. Bitmap indexes might also be constructed to accelerate range selections so they would be considered during the query optimization plan enumeration[Stockinger and Wu, 2006].

Query Optimization Initially, the query is submitted to the CPU, written in SQL or another high-level query language. The query is parsed to an internal representation and fed to the next step which is the typical query optimization exploring a space of alternative plans. Update queries are only applied to the delta-store, stored in the CPU main memory.

For read-only queries, the optimal plan determines the operator ordering and the device that each operator will execute. The query optimizer enumerates alternative plans corresponding to different operator orders and access paths. The optimizer has access to the data placement of the input table columns and consults the statistics catalog to estimate the cost of the alternative plans. The optimizer computes a query graph, representing the lower cost plan, defining the execution order and the device that each operator will execute on. Query optimization on a heterogeneous environment is out of the scope of this thesis but since the space of the plans considering different operator placements might explode for an increasing number of devices, we would need a heuristic to limit the space of alternatives. In Chapter 8 we discuss what cost models would be suitable for a heterogeneous database optimizer.

In particular, for complex predicates, where a subset of filters are applied on the CPU

and the remaining on the GPU the optimizer must select the optimal co-processing strategy. The strategy describes the order of the predicates are executed and whether the predicates are evaluated in sequence or in parallel across the available processors. The optimal choice depends on the predicate selectivity, predicate cost, and the processor performance and load. Section 5.3 describes the alternative strategies of CPU-GPU interaction for complex filters. A complete database system would additionally require monitoring the utilization of the available devices resources. In our setting, a single query is processed a time. However, databases could benefit by processing multiple queries at a time to maximize the GPU utilization, while avoiding shared resource contention [Wang *et al.*, 2014].

The optimized plan describes the form of the intermediate query results. The result-sets might be represented either by a row-identifier list or a bitmap, depending on which is smaller.

Code Generation The query graph representing the optimal plan must be now compiled to low-level code that might be mapped to one or multiple GPU kernels. Database systems have used LLVM to produce quickly code that would execute efficiently on the GPU and at the same time is portable across different GPU architectures [Suhan and Mostak, 2015]. Intermediate results between kernels would be written to the global memory. In this step, we would integrate our conjunctive selection optimization framework, presented in Chapter 4, to compute the optimal way to group different select predicates in different CUDA kernels. The execution engine must communicate with the statistics catalog to estimate the cost of different execution plans, taking into account indexes, such as zone maps on the involved attributes.

Query Run-time Our parallelism approach is simple for GPUs: For each kernel implementing a data processing operator, a GPU thread is processing a specified number of input elements. In case of operators with two inputs we split the work based on the size of the larger table. Query run-time on the GPU-device will use our optimized operator implementations for string matching, presented in Chapter 5. Based on our evaluation, our Pivot-KMP implementation for single-pattern matching and Pivoted-AC for multi-pattern matching predicates are the most efficient implementation for most queries.

1.4 Thesis Contribution

This thesis addresses performance bottlenecks that prevent in-memory analytics from taking full advantage of the GPU performance.

Database processing on GPUs is still a new topic that has a lot of research potential. The choice of the the sub-topics we worked on was based on the following motivation. First, we chose operators that are vital for the performance of database analytics and still have performance bottlenecks that previous work has not addressed. We concentrated on problems that not only have a practical applicability but also involve interesting abstractions capturing fundamental design features of GPUs, such as warped execution and throughput optimization, rather than architectural details that might be absent from future processors.

In our analysis we used profiling tools to identify individual operator performance bottlenecks in the baseline implementations but also algorithmic analysis to identify fundamental shortcomings of current methods. Using our insights on the GPU architecture we came up with a set of solutions that belong to different steps of the data processing workflow. Some solutions involved preprocessing of data by optimizing the memory layout. In other cases algorithmic redesign was essential, so after analyzing the average and worst-case algorithmic behavior of the available solutions, we adapted algorithms to match the GPU model. Within each thesis chapter we evaluate our algorithmic frameworks using micro-benchmarks and compare their performance against the baseline and other competitive existing solutions.

Our Experimental Evaluations are 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 GPU programming frameworks[NVIDIA, 2016a; AMD, 2013]. Most of our experiments were carried out on a single GPU. However, the operators we accelerated are "embarrassingly" parallel and it is trivial to extend them for multi-node GPU settings.

The following Sections discuss our specific research contributions. Unless otherwise stated, the software to carry out our evaluations was written by this thesis author. In all of the projects, discussions with my thesis advisor helped me shape the research problems and refine my experimental evaluations. I gratefully acknowledge the contribution of other collaborators in each individual subsection.

1.4.1 Shared Memory Joins & Aggregations

GPU threads can co-operate using shared memory, which is organized in interleaved banks and is fast only when threads read or modify addresses belonging to distinct memory banks. Our goal in Chapter 3 is to examine the role of bank and value conflicts for database processing on GPUs and to suggest bank-aware optimizations for improving data access behavior. We focus on two core database operators: foreign-key joins and grouped aggregation.

Aggregation is common in business analytics queries. Grouped aggregation is a helpful tool to summarize and explore data across different dimensions and potentially find unusual patterns. To achieve high performance we need fast access to the aggregate table. Aggregation on OLAP data warehouses, unless data is denormalized, requires several joins of the fact table to the dimension tables. On CPUs to achieve high performance for joins and aggregations, we try to fit the data in data caches [Manegold *et al.*, 2002] by partitioning. However, L2 caches on GPUs are not optimized for latency as in CPUs but for throughput [Mei and Chu, 2015]. To accelerate joins and aggregations we consider using the fast shared memory. For foreign key joins between a fact table and a dimension table, shared memory is used to contain the needed fragment of the dimension table. For grouped aggregation, shared memory contains the running aggregates for each group. In our baseline implementation scans through a fact table, consulting the structure in shared memory for each row. However, threads might access the same bank concurrently resulting in bank conflicts and degrading performance.

Our key insight is that if there is additional shared memory available beyond that needed for the basic structure described above, we can use that memory to store extra copies of the underlying data. For foreign key joins, we store duplicate dimension table rows; for grouped aggregation, we store duplicate groups. In either case, we make sure that the duplicates and the original item all occupy different banks. We modify the base fact table so that some rows refer to one of the duplicates rather than the original item, in order to reduce the number of bank and value conflicts. This modification is done once at data loading time, so the cost of optimization and the modification of the fact table is amortized over many queries.

In our Experimental Evaluation, we discover that:

- Columns written by various queries, for example potential group columns, should be optimized for writes
- For skewed data a 2× increase in footprint gives most of the benefits of bank optimization

1.4.2 Multi-Predicate Selection Execution

In Chapter 4 we are interested in the performance of a table scan combined with the application of a compound selection condition. While a filtering scan might be considered a "lightweight" operator compared to a join, it is important for a number of reasons. Large tables cannot be stored in the shared memory, because of its limited capacity so larger tables should be stored in the global memory of GPUs. As the initial operator in a query plan, a filtering scan will typically process a large volume of data compared with later operators that process filtered data. In data analytics scenarios where there is a limited space budget for secondary indices and/or queries touch relatively large segments of the data, scanning the data may be preferable to index-based access methods.

Two baseline implementations involve applying all filters in the same kernel function or apply each filter in a different kernel function with each kernel but the last writing the intermediate query results in the GPU device memory. The first approach increases memory cost because each kernel must push the intermediate results into the global memory. The second approach reduces the memory cost, but it might result in reduced thread utilization. For conjunctive filters, if the filter is not true for some threads in a warp, these threads will remain idle while the other threads in a warp are evaluating the remaining conditions. We need to find the trade-off of thread utilization and memory cost, maximizing the overall performance.

The entire space of scan plans involves alternative ways to group filters into kernels. Especially in the presence of expensive select conditions, finding the optimal plan is necessary to maximize query performance. We suggest an analytical model estimating the relative performance of different plans by capturing the memory architectural features of GPUs. Our cost model has two main components: The cost of writing the intermediate results communicated between kernels and the overlap degree of memory loads.

Our main results of our experimental evaluation are summarized:

- Our analytical cost model accurately predicts the relative performance of different execution plans
- Our optimization algorithm using our cost model achieves 2-3× speed-up over the baseline approaches

This work has been published in DaMoN workshop [Sitaridi and Ross, 2013].

1.4.3 String Matching Optimization

Chapter 5 focuses on the efficient implementation of string matching operators common in SQL queries. GPU-accelerated string matching had not been studied in a database setting. String matching algorithms have been extensively studied for CPUs but state-of-the-art algorithms have branches and irregular memory access patterns that are inefficient for GPUs. Due to different architectural features, the optimal algorithm for CPUs might be suboptimal for GPUs. In a naive string matching implementation threads process different strings, stored contiguously in the GPU global memory. GPUs achieve high memory bandwidth by running thousands of threads concurrently so it is not feasible to keep the working set of all threads in the cache. In the presence of loops and branches, threads in a group have to follow the same execution path; if some threads diverge then different paths are serialized.

Also, in the context of a database system we are often interested only in the first match. If some threads locate a match early on, they can stop scanning the string. However, because of the warped execution of GPUs, these threads will remain idle while the remaining threads are still scanning their input string. GPU string matching in the database context improves the memory access locality of the input strings while minimizing the effect of branches on string matching performance.

We study the cache memory efficiency of single and multi-pattern string matching algorithms for conventional and novel pivoted string layouts in the GPU memory. Our pivoted layout split the input strings into similarly sized pieces and interleaves the string pieces in the global memory. As a result, threads of a warp processing contiguous strings will coalesce their accesses to the global memory. However, depending on the chosen string matching

algorithm, some threads might fall behind and will end up accessing different string pieces, increasing the cache footprint. We define this effect as *memory divergence*. Because of the limited L2 capacity, memory divergence degrades the string matching performance.

We also evaluate the different matching algorithms in terms of average and worst case performance and compare them against state-of-the-art CPU and GPU libraries. To reduce thread divergence we split string matching into multiple steps.

Our experimental evaluation shows that thread and memory efficiency affect performance significantly and that our proposed methods outperform previous CPU and GPU algorithms in terms of raw performance and power efficiency.

We summarize below the results of our Experimental Evaluation:

- Interleaving the strings in the GPU memory results in reduced memory traffic and improved performance
- The Knuth-Morris-Pratt algorithm is a good choice for GPUs because its regular memory access pattern reduces memory divergence
- GPUs achieve on average 3× faster performance than CPUs and 1.5× reduced energy consumption
- CPUs still present a viable alternative platform for string matching because of their high Performance/$ ratio

Our work has been published in the Special Issue of the VLDB Journal on "Data Management on Modern Hardware" [Sitaridi and Ross, 2015].

1.4.4 SIMD-Accelerated Regular Expressions

CPU with SIMD extensions bear similarities in their approach to GPUs. In CPUs, the same instruction is broadcast to multiple execution units. A lane in a SIMD register corresponds to a GPU thread of a warp. Based on the above observations, we explore whether our concept of data parallel string matching can be extended for CPU processors with SIMD extensions. However, in recent mainstream CPUs, substring matching is supported in

hardware through a specialized SIMD instruction that is sufficient to cover most queries using the `like` keyword.

While a single instruction is enough to cover most queries with substring matching operators, more advanced predicates such as regular expression matching cannot be optimized as easily. Regular expressions in queries offer high expressive power inside the DBMS and are typically employed as selective filters. Generic regular expression depend on both the complexity of the underlying deterministic-finite-automaton (DFA) and the irregularity of the input. In Chapter 6, we show how to filter string columns against regular expressions by using branchless vectorized (SIMD) code to traverse through the DFA.

CPUs have larger caches so there is no cache thrashing when threads process independent input strings. Hence, we follow a different approach by accessing the input strings non-sequentially and eliminate the inherent need for branching, whether the string is accepted or rejected by the DFA.

Our evaluation on mainstream CPUs and co-processors shows our approach to be up to 5X faster compared to the scalar implementations, offering a crucial upgrade for DBMSs to support efficient regular expression matching in selections.

Our work has been accepted for publication at the DaMoN 2016 workshop.

1.4.5 Compression Acceleration

There exists a plethora of compression techniques, each having a different trade-off between its compression ratio (compression efficiency) and its speed of execution (bandwidth). Deflate algorithm, used by gzip, has fast decompression performance and a reasonable compression ratio and is thus widely used by many large scale systems [Ganelin *et al.*, 2016]. Therefore, we show how to leverage the massive parallelism provided by GPUs to accelerate Inflate, the Deflate decompressor.

We design a massively-parallel compression scheme that can be used to shrink the memory footprint of data in the GPU and better utilize its limited memory capacity. We are interested in developing a general-purpose compression framework that can support diverse Big Data workload formats ranging from unstructured text to matrix data. Most research so far has focused on the speed of compressing data as it is loaded, but the speed of decom-

pressing that data can be even more important for modern workloads – data is compressed only once when loaded into the database but repeatedly decompressed as it is read when executing analytics or machine learning jobs. Decompression speed is, therefore, crucial to minimizing the response time of these applications, which are typically data bandwidth bound. We are therefore interested in optimizing decompression performance, without underlying compression ratio and speed.

Straightforward parallelization methods, in which the input block is simply split into many, much smaller data blocks that are then processed independently by each processor, result in poorer compression efficiency, due to the reduced redundancy in the smaller blocks, as well as diminishing performance returns caused by per-block overheads. In order to exploit the high degree of parallelism of GPUs, with potentially thousands of concurrent threads, our implementation needs to take advantage of both intra-block parallelism and inter-block parallelism. For intra-block parallelism, a group of GPU threads decompresses the same data block concurrently. Achieving this parallelism is challenging due to the inherent data dependencies among the threads that collaborate on decompressing that block.

We propose and evaluate two approaches to efficiently parallelize Inflate on GPUs. The first technique exploits the SIMD-like execution model of GPUs to coordinate the threads that are concurrently decompressing a data block. The second approach avoids data dependencies encountered during decompression by pro-actively eliminating performance-limiting dependencies during the compression phase. The resulting speed gain comes at the price of a marginal loss of compression efficiency.

We also present **Gompresso/Bit**, the parallel implementation of an Inflate-like scheme [Deutsch, 1996] that is suitable for massively-parallel processors such as GPUs. We also implement **Gompresso/Byte**, based on LZ77 with byte-level encoding. It trades off slightly lower compression ratios for an average of 3× higher decompression speed.

Our main results are summarized below:

- **Gompresso**, running on an NVIDIA Tesla K40, decompresses two real-world datasets 2× faster than the state-of-the-art block-parallel variant of zlib running on a modern multi-core CPU, while suffering no more than a 10 % penalty in compression ratio.
- **Gompresso** also uses 17 % less energy by using GPUs, against state-of-the-art parallel

CPU libraries for decompression

Our work was accepted for publication at the ICPP 2016 conference.

1.5 Thesis Outline

In this Section, we describe the structure of the present thesis. Chapter 3 describes how to efficiently preprocess data to access efficiently shared memory for concurrent reads or updates.

In Chapter 4 we present our optimization execution algorithm for conjunctive selections, which uses an accurate analytical cost model to compare the relative performance of alternative plans.

Chapter 5 tackles the sub-string matching optimization for GPU databases by carrying out an extensive evaluation of alternative algorithms, string layouts, and parallelization methods. Chapter 6, motivated by the speed-ups of GPU sub-string matching, extends our ideas for more vectorized CPU-based regular expression matching.

In Chapter 7 we present our general purpose GPU compression framework, addressing the needs of Big Data workloads for high decompression speed.

Finally, in Chapter 8 we summarize the main conclusions of our research and outline future directions for GPU-powered data analytics and database processing.

Chapter 2

Related Work

In this Chapter we discuss related projects on database acceleration using GPUs. We also present key choices of more complete GPU research database systems.

2.1 Relational Operator Accelerator

We analyze further related projects to our contribution in the individual thesis chapters. Research on database processing on GPUs has demonstrated significant speed-ups [Fang *et al.*, 2007; Bakkum and Skadron, 2010; Diamos *et al.*, 2013]. The power of GPU processors has been exploited for the efficient implementation of different join algorithms [He *et al.*, 2008b]. GPU hash-join achieved practically full utilization of the PCIe bandwidth [Kaldewey *et al.*, 2012]. The input tables were stored in the CPU RAM and GPU was used to offload the requests from the probe table to the hash table, exploiting the fast random memory access of the GPU global memory. Conjunctive selections and aggregations were accelerated on earlier GPU processors with a different memory architecture [Govindaraju *et al.*, 2004].

P-ary search is an algorithm operating on sorted lists, designed to scale to the number of available processors [Kaldewey *et al.*, 2009; Kaldewey and Di Blas, 2011]. FAST, an architecture sensitive tree index suitable for CPU and GPU processors accelerated in-memory search [Kim *et al.*, 2010; Kim *et al.*, 2011].

Scatter and gather operations have been adapted for GPU processors to improve data

locality [He *et al.*, 2007]. Radix-sort was implemented with these scatter/gather operations while using shared memory to improve performance. A Map-Reduce framework has been suggested to facilitate programming of web analysis tasks on GPU processors without sacrificing performance [He *et al.*, 2008a].

An extended precision library has been implemented on GPUs and incorporated into a GPU-based query engine to achieve a significant performance improvement for scientific applications [Lu *et al.*, 2010].

GPUs also have had a commercial impact, especially on the database start-up market share. MapD is a Big Analytics platform taking into advantage the high parallelism and fast memory of GPU processors and uses indexes to search tweet contents [Mostak and Graham, 2014]. Other state-of-the-art GPU database systems are Parsteam [Michael Hummel, 2010], SQream [Ori Netzer, 2014] Parstream [Michael Hummel, 2010], GPUDB[GPUDB, 2016], and Jedox [Raue, 2010].

2.2 State-of-the-art GPU-Accelerated Database Systems

As we mentioned, there a lot of start-ups with commercial impact. Here, we focus on research databases systems because of their design being publicly available. Some of these start-ups include MapD [Mostak and Graham, 2014], SQream [Ori Netzer, 2014], .

We present now the research database systems and analyze them based on the following implementation choices:

- Cost models: What type of function is used to estimate the cost of alternative query execution plans?
- Storage layout: Is data stored in a row-wise or a column-wise fashion?
- Query execution: How are different query plans translated to GPU code? This step is related to the execution model chosen by the engine.
- Query scheduling: Is query processing mapping queries just to GPUs or does it balance the utilization of the available CPU and GPU processors?

- String matching: Is string matching supported and if so what type of layout is used for the string datasets?

CoGaDB CoGaDB optimizes co-processing for columnar hybrid CPU/GPU database systems.

Cost model: A self-learning approach is used to distribute the amount of work in heterogeneous architectures [Breß *et al.*, 2012]. This allows CoGaDB to be agnostic of the architectural details of the available processors and easily extensible.

Storage layout: Tables are stored in a column-wise way allowing threads to coalesce their accesses to the GPU memory [Breß *et al.*, 2014].

Query execution: The input is processed in an operator-at-a-time fashion, so an operator fully processes the input data before the result is pushed to the next operator.

Query scheduling: CoGaDB optimizes query plans to maximize parallelism across the available devices, while keeping track of the operator utilization.

String matching: CoGaDB uses dictionary compression on strings so only equality and inequality predicates are implemented.

Ocelot Ocelot is an extension of MonetDB that maps operators on different computer architectures in a hardware oblivious way. This means, that a single code-path is maintained for CPUs, GPUs, and potentially others types of architectures, such as FPGAs. The advantage of this approach is the reduced development overhead.

Cost model: Ocelot uses MonetDB optimization, which is based on analytical cost-models [Heimel *et al.*, 2013].

Storage layout: Ocelot is a column-store engine.

Query execution: Ocelot is a hardware oblivious extension of MonetDB using OpenCL. It inherits the operator-at-a-time execution of MonetDB. However, it uses lazy evaluation to match with the OpenCL programming model. Lazy evaluation in this context means the operators are scheduled but the engine does not wait for them to complete. At the time each operator is scheduled, it has a set of events on which it depends to ensure that all of its inputs are available before execution.

Query scheduling: Ocelot runs an entire query-plan on a single device, either on the CPU or the GPU.

String matching: Ocelot supports simple string operations, such as equality predicates.

WOW WOW is a research prototype system, demonstrated by IBM, running the full Star Schema Benchmark (SSB) at scale factor 1000[Mueller *et al.*, 2013].

Cost model: WOW focuses on execution optimization rather than traditional query optimization so it does not introduce a new cost model.

Storage layout: WOW uses a columnar layout.

Query execution: WOW has easily composable plans since each kernel implements a single operator, executed in an operator-at-a-time fashion. The execution engine uses late materialization.

Query scheduling: WOW uses GPU processors for group-by and join predicates. The select predicates are applied on the CPU using multiple threads.

String matching: SSB does not require any string matching operations.

GPUQP GPUQP is a relational in-memory database engine using both CPUs and GPUs for query processing [He *et al.*, 2009].

Cost model: GPU-QP uses analytical cost models to find the most cost-efficient plan.

Storage layout: GPU-QP uses the columnar layout for CPUs and GPUs.

Query execution: Execution is done operator-at-a-time. An operator might be partitioned and evaluated concurrently on the CPU and the GPU.

Query scheduling: A single query can be executed on the GPU, the CPU or on both processors concurrently.

String matching: To the best of our knowledge, GPUQP does not support string matching operations.

Virginian Virginian implements a subset of SQLite commands in the GPU. This subset involves filtering and aggregation operations [Bakkum and Chakradhar, 2012].

Cost model: Virginian is implemented within SQLite database. No additional query optimization techniques were described specific for GPU processing.

Storage layout: Virginian implements the *tablet* data structure, where fixed-length columns are stored in a column-wise way in the tablet. Variable-length columns, such as strings, are stored in a dedicated section of the tablet structure.

Query execution: Virginian follows the op-code model. All query operators are compiled into a single GPU kernel, to avoid the cost of writing the intermediate results.

Query scheduling: Queries are executed either in their entirety on the CPU or on the GPU.

String matching: The tablet structure supports storage of strings but string matching operators were left for future work [Bakkum and Chakradhar, 2012].

Red Fox Red Fox is a compiler and runtime execution framework for running relational queries on GPUs [Wu *et al.*, 2014].

Cost model: Red Fox is an execution environment rather than a full database system, so it does not do full optimization for GPUs.

Storage layout: Tables are stored in a key-value store. Keys and values are represented by densely packed arrays of tuples.

Query execution: An operator, depending on its implementation, is mapped to one or multiple kernels. For example, join operations are mapped to multiple kernels.

Query scheduling: Queries in Red Fox are only executed on the GPU.

String matching: Red fox supports LIKE operations. String are stored in a separate table with different sub-tables for strings having the same length.

Chapter 3

Shared Memory Joins & Aggregations

3.1 Introduction

GPUs have a radically different memory-hierarchy from a traditional CPU. As discussed in Section 1.2, global memory is the largest type of memory but it has high latency: 400–600 cycles. Shared memory is used as a parallel, software controlled cache. Its capacity on a high-end GPU is 16KB or 48KB depending on the kernel configuration. The access time of each shared memory bank is 4-bytes per 2 cycles. 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.

The C2070 GPU offers atomic operations on global and shared memory [1], where each thread that calls an atomic operation on a variable is promised that this variable will not be accessed by another thread until this operation is complete [NVIDIA, 2010]. Other threads trying to access the same address get serialized. This creates another possible form

[1]The K40 and K80 offer similar atomic operations. Our work was done when C2070 was the state-of-the-art GPU.

of contention between threads when at least one is writing data. We refer to this kind of serialization as a *value conflict*. Value conflicts can span warps, because atomic operations may still be in flight when new warps get scheduled to the SM.

Our goal is to examine the role of bank and value conflicts for database processing on GPUs, and to suggest bank optimizations for improving data access behavior. We focus on two core database operators: foreign-key join and grouped aggregation. For foreign key joins, shared memory is used to contain the needed fragment of the dimension table. For grouped aggregation, shared memory is used to contain the running aggregates for each group. In both cases, we make a scan through a fact table, consulting the structure in shared memory for each row.

Our key insight is that if there is additional shared memory available beyond that needed for the basic structure described above, we can use that memory to store extra copies of the underlying data. For foreign key joins, we store duplicate dimension table rows; for grouped aggregation, we store duplicate groups. In either case, we make sure that the duplicates and the original item all occupy different banks. We modify the base fact table so that some rows refer to one of the duplicates rather than the original item, in order to reduce the number of bank and value conflicts.[2] This modification is done once at data loading time, so the cost of optimization and the modification of the fact table is amortized over many queries.

3.2 Related Work

Alternative aggregation strategies on chip multiprocessors minimize thread-level contention on CPUs exhibiting different degrees of memory sharing between threads [Cieslewicz *et al.*, 2007]. A framework for parallel data-intensive operations automatically detects and responds to contention by cloning popular items at query time [Cieslewicz *et al.*, 2010]. This and two additional parallel aggregation strategies have been studied on a Nehalem processor [Ye *et al.*, 2011].

[2]For grouped aggregation, a final pass combines the aggregates for the duplicates into a single aggregate for each item.

Cuckoo hashing methods resolve hash collisions using multiple hash functions for each item instead of one [Pagh and Rodler, 2004; Erlingsson *et al.*, 2006; Ross, 2007]. During the insertion of an item, if none of the positions are vacant, the key is inserted in one of the candidate positions, selected randomly, displacing the key previously placed there. The displaced key is re-inserted in one of its alternate positions. This procedure is repeated until a vacant position is found or a maximum number of re-insertions is reached. Our method searches for the shortest relocation sequence that eliminates bank conflicts, instead of following a randomized procedure.

Data declustering techniques are used to distribute data partitions among multiple storage units, e.g., disks [Holland and Gibson, 1992] or servers. Replication and optimal replica placement of data items has been suggested to maximize resource utilization in the Kinesis distributed storage system [MacCormick *et al.*, 2009].

3.3 Problem Description

We assume an in-memory OLAP setting and a star schema. Using coding techniques commonly used in OLAP databases [Copeland and Khoshafian, 1985; Whang and Krishnamurthy, 1990; Pucheral *et al.*, 1990], we assume that foreign key columns and grouping columns are coded with consecutive integer codes starting from 0. That way, dimension tables and aggregate structures can be organized as simple arrays rather than as hash tables.[3] There is a direct relationship between the array index and memory bank, since memory banks on the C2070 are distributed in a round robin fashion every four bytes [4] [NVIDIA, 2015b]. We assume all data tables are stored columnwise with 4-byte datatypes, maximizing the potential for bank parallelism.

If d is a foreign key column, then the domain of d in the initial fact table will be the integers between 0 and $c-1$ where c is the cardinality of the referenced table. If d is a

[3]Such optimizations are particularly important in GPUs. If different threads in a warp need to follow hash overflow chains of different length, then the execution paths will diverge and threads will be partially serialized for the length of this divergence.

[4]Newer GPUs, such as the K40, support two banking modes: For 4-byte banks and an alternative banking mode for 8-byte banks. Our project was implemented when the C0270 was the state-of-the-art GPU.

grouping column, then the maximum value in the column is one less than the effective size of the aggregation table.[5] We will process a "warp's worth" of contiguous data at a time, a unit we shall call a "chunk." On the C2070, the chunk size is 32 data elements [6].

Now suppose that column d takes values 3 and 35 at two places in a single chunk of elements from column d. Because $3 \equiv 35 \mod 32$, both references will map to the same bank, leading to a bank conflict. For this example, we might create a new version of the element in slot 35, and put it in slot 3207, say, at the end of the table. In the fact table row with the conflict, we re-map 35 to 3207 and the conflict no longer holds because $3 \not\equiv 3207 \mod 32$. We keep track of this new row in slot 3207, which could be used for subsequent fact table rows as an alternative to slot 35 if slot 35 causes another conflict.

In the unbounded version of the problem, we do not limit the number of copies generated. If we're only concerned about bank conflicts within a warp, then 32 copies of each data item, one per bank, would guarantee that we could avoid bank conflicts altogether. In practice, fewer than 32 copies are needed. In the bounded version of the problem, we observe that the shared memory capacity puts a limit on how many values can be efficiently handled. Based on this capacity, we set a budget on the average number of copies. For example, a budget of 5 would mean that the total size of the table including duplicates cannot exceed 5 times the size of the table without duplicates.

In the aggregation case, where we need to perform writes on the shared-memory-resident array, we also need to create copies to resolve value conflicts, where the same value appears more than once in a warp. We will also extend the analysis beyond the warp, looking for value conflicts between nearby warps within a fixed "window," on the grounds that an in-flight atomic update of a value might conflict with that same value in subsequent warps. In the worst case, more than 32 copies may be needed to completely avoid value conflicts.

Data partitioning between threads is static. Each thread in a thread block processes a certain number of records, so we know beforehand which records a thread is going to process. We can detect bank conflicts by scanning chunk-by-chunk and inter-warp value

[5] If some intermediate values don't appear at all in the table, then the actual grouping cardinality may be smaller.

[6] Newer GPUs, such as the K40, have the same number of banks, thus using the same chunk size.

conflicts by remembering the set of values in the last chunks. The number of chunks we remember defines the optimization window. Our algorithm is easily extended for different regular access patterns, e.g., an access pattern where each thread reads four integers at a time using built-in CUDA vector types.

Static partitioning means that our optimization may not be effective if data items "move" from their original fact table grouping before being processed. This may limit our choices for other operators. For example, a selection operator that scanned the fact table and wrote an intermediate result containing only the matching records would change the chunking pattern. Alternative selection operators are compatible with retaining physical order. One option would be to combine the selection and aggregation into one joint kernel, so that the aggregation operator sees the data in the original locations. Another option would be to use a clustering scheme such as multidimensional clustering [Markl *et al.*, 1999; Padmanabhan and others, 2003] so that the records matching the selection conditions tend to be contiguous.

Theta	Distinct Banks	Write SR	Read SR
0.00	20.42	3.54	3.44
0.25	20.41	3.53	3.42
0.50	20.36	3.57	3.38
0.75	19.94	3.81	3.18
1.0	18.24	5.33	2.77

Table 3.1: Average number of distinct banks, read serialization rounds and write serialization rounds in a chunk.

We consider a variety of Zipfian distributions for the fact table column, ranging from $\theta = 0$ (uniform) to $\theta = 1$ (very skewed). To give a better sense of the problem, we provide in Table 3.3 some statistics for a column of cardinality 1024 for different θ parameters. Without performing any optimization, we analyze the column and compute how many read and write serialization rounds are required, together with the number of distinct banks in a chunk. Note that skew hurts write serialization due to an increase in the number of value conflicts, but helps read serialization due to improved locality (since shared items can be

broadcast to multiple threads).

3.4 Data Placement Algorithms

This Section discusses how to optimize data placement by creating extra copies of the table values accessed during query processing. Each subsection analyzes how to adapt the placement algorithm for different optimization cases.

3.4.1 Write Conflicts

3.4.1.1 Intra-Warp Optimization

Before we discuss our main algorithms, we remark that it might be possible to reduce bank and value conflicts by reordering the fact table so that rows that would cause a conflict in the current chunk are held back until a later chunk. Reordering can only be a partial solution, because if a value occurs with a frequency higher than 1/32, then value conflicts cannot be eliminated by simple reordering. Further, there are often criteria more important than bank conflicts for ordering a fact table, so assuming the ability to reorder the table may be unreasonable.

Depending on the data distribution and data ordering each value should be assigned a different number of copies. Intuitively, frequently occurring values should get more copies, because those items are more likely to conflict, and because the extra copies are the most valuable when they can be used by many rows. Rather than statically choose a prioritization scheme for the number of copies based on frequency, we use a dynamic scheme to determine the number of copies for each value based on the "demand" for extra copies.

Initially each value is assigned one copy. We process a chunk of fact table rows at a time, until each chunk has been processed. For each chunk we proceed as follows.

We first try to assign as many values in the chunk as we can to one of its available copies, without causing any bank or value conflicts relative to previous choices. If we succeed at assigning all values, we move to the next chunk. If not, which is more likely, some values remain whose copies conflict with previous assignments. For each such value v, we assign v into one of the occupied banks and unassign the value v' that was previously there. We

then try to reassign v' into one of its other copies, which could lead to a recursive sequence of reassignments. We do not consider reassignments at random. Instead, we use a breadth-first-search (BFS) algorithm to find the shortest sequence of reassignments that allows the value to be inserted without conflicts.

Assignment of a value in a chunk may fail for one of the following reasons: 1) the distinct number of banks among all copies in the chunk is less than the number of banks, or 2) none of the keys can be assigned to an empty bank, because an empty bank is not reachable given the current set of copies. In both cases, to resolve the conflict a new copy of the value is created in one of the empty banks. In this way, we generate new copies of the values that are hardest to place. If we have already spent our space budget, we place the item without creating a new copy, and accept that this chunk will need multiple serialization rounds.

Two important choices affecting the space and time efficiency of the algorithm are:

- When failing, multiple bank-slots might be available. We want to assign the same number of copies to each bank-slot so that the replicated table is stored contiguously in the memory without gaps. If more copies are assigned to certain banks then there will be "holes" in the memory in the less popular banks. These holes still consume shared memory, and should be avoided.

- The order according to which we insert the copies into the BFS queue matters. If we always enqueue the lower numbered banks first, then there is a high probability that the available slots upon failure will be the higher numbered slots, creating contention on those banks. To address this problem, we start each chunk from a different position enumeration of the copies.

Figure 3.1 shows how our algorithm processes a chunk of the second column of a fact table to eliminate bank conflicts. For simplicity, the chunk size in the example is 8 elements, equal to the number of threads in a warp and equal to the number of memory bank-slots. Each dashed edge links a placed value to its alternative bank locations. After the initial assignment, values 15 and 531 remain unassigned. In the next step, a relocation sequence is found for 531 that displaces 14, displacing 6 in turn. For value 15 there is no valid sequence of value movements because the only empty bank-slot 4 is unreachable, so a new copy of 15

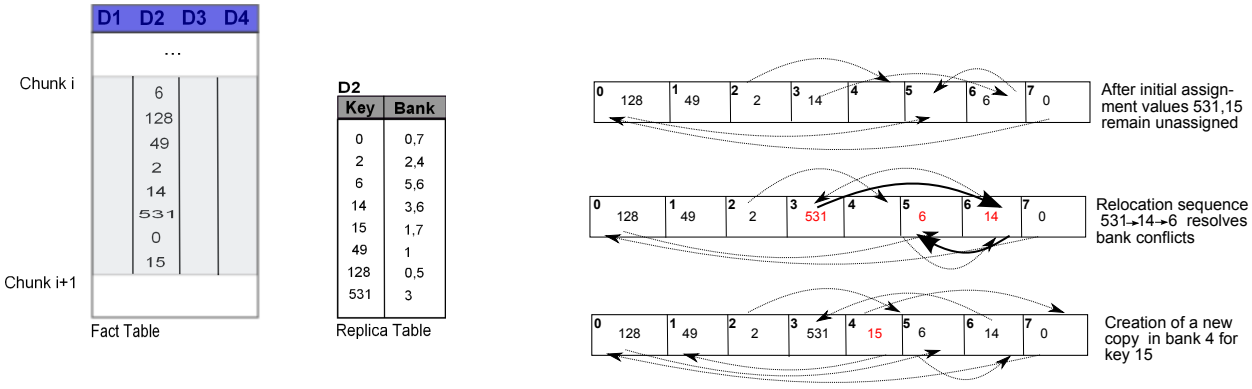


Figure 3.1: Bank optimization for 8-element chunks.

is created in bank 4.

3.4.1.2 Inter-Warp Optimization

Inter-warp value conflicts occur only between warps belonging to the same thread-block. We set a window size corresponding to the number of chunks prior to the current chunk to consider for value conflicts. (A window size of zero means that inter-warp value conflicts are ignored.) We shall examine the impact of window size experimentally. We extend the BFS algorithm so that copies that have previously been used within the current window are not used again in the current chunk.

In case of failure, we create a new copy as before. If we have already used the space budget we try again to place the value in the current chunk, ignoring inter-warp conflicts.

Finally, for a skewed dataset with a large window the number of copies for frequent values will also be high, increasing the search cost. To reduce this cost we consider first the copies that were least recently used, increasing the probability that a non conflicting assignment is found early.

3.4.2 Read Conflicts

In case of read conflicts the problem is relaxed due to the value multicasting performed in the hardware. If in a chunk there are some duplicate values, then all of those values can be assigned to the same bank without degrading the performance. This means that we can simply run the assignment algorithm for the first occurrence of the value in the chunk and

put the rest of the occurrences in the same bank.

3.5 Experimental Evaluation

3.5.1 Experimental Setup

For our experiments we used synthetic data following the zipf distribution for different θ parameters. The default number of distinct values in the zipf distribution was 1024. Each column is a 4-byte integer. We used our suggested technique to resolve bank and value conflicts for different table sizes (t, up to 200M which is the maximum number fitting in GPU memory), window sizes (w) and space budgets (b).

We ran separately the following queries on an OLAP star-schema:

```
Q1: SELECT SUM(D1.B)          Q2: SELECT A, COUNT(*)
    FROM F, D1                    FROM F
    WHERE F.A=D1.A                GROUP BY A
```

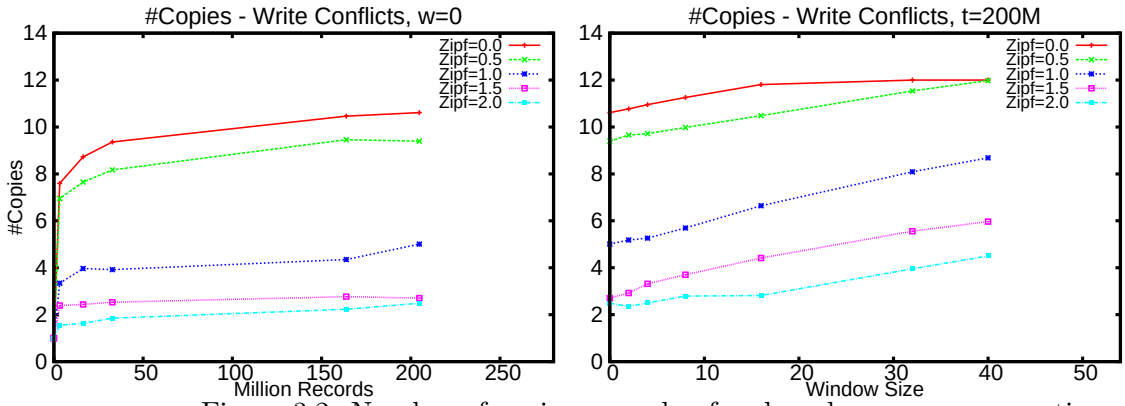


Figure 3.2: Number of copies per value for shared memory aggregation.

In both cases fact table `F` was stored in the global memory. In the first query dimension tables are stored in the shared memory to perform the foreign key join, and a scalar aggregate is generated. In the second query shared memory is used to store the aggregates local to a thread block. After each thread-block computes the sums in the shared memory, it merges the results for each copy to global memory; in the end, global memory contains

the correct aggregate sums. In what follows, results measuring read conflicts correspond to `Q1`, and results measuring write conflicts correspond to `Q2`.

Optimization was done on a dual-chip Intel E5620 CPU using 16 threads. GPU performance was measured on an Nvidia Tesla C2070 machine with 6GB of RAM and a nominal RAM bandwidth of 144GB/s. The GPU was configured to use 48KB of shared memory in each SM. Each thread-block processed 350K rows using 1024 threads. The number of thread blocks for a kernel was computed based on the number of table records.

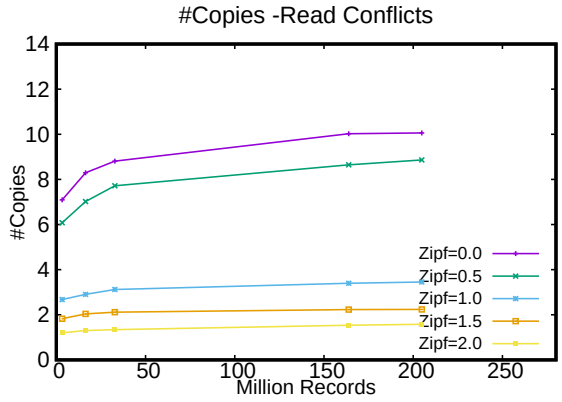


Figure 3.3: Number of copies per value for shared-memory joins.

3.5.2 Memory Footprint

Figures 3.2 and 3.3 show the number of copies per value for different optimizations with a very lenient space budget b of just under 12 copies. Writes generate more copies than reads, because the write value conflicts create additional constraints. For similar reasons, the number of copies increases as the window size is increased. As skew increases, the number of copies decreases significantly. Many copies of a few popular items is often enough to create a conflict-free access pattern. As the number of records increases, the number of copies also increases, but the increase is fairly mild after 50M records. Figure 3.4 shows the average number of copies per value and the total number of value replicas for varying column cardinality. As expected, for lower cardinalities the number of copies is higher because there is an also higher probability for value conflicts and this trend is more apparent when an

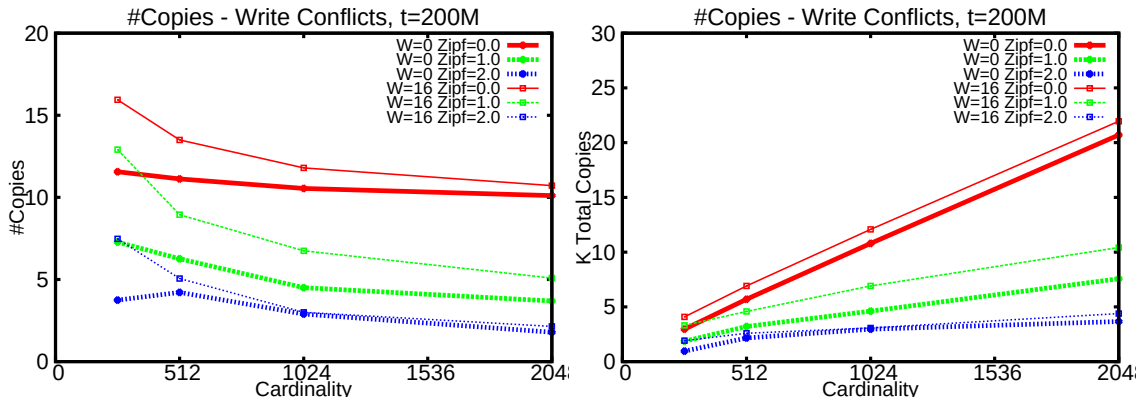


Figure 3.4: Number of copies per value for varying cardinality.

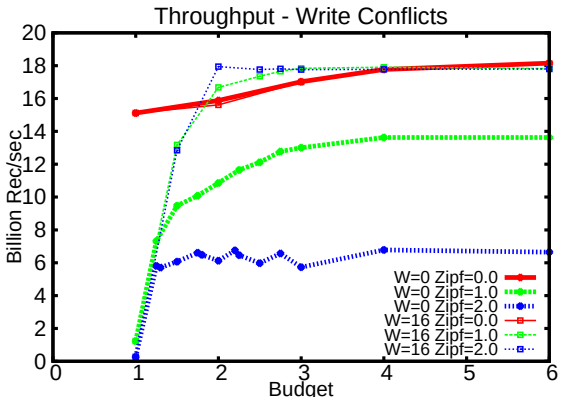


Figure 3.5: Throughput for different budgets.

optimization window is set.

Figure 3.5 shows how performance depends on the space budget. The space budget is the maximum allowed total footprint increase factor. For uniform data, we get close to maximum throughput at an average of 4 copies per value. For skewed data, even 2 copies per value gives good performance: our algorithm first creates copies for the frequent items that cause most of the conflicts. These results show that with a realistic (2–4X) increase in shared memory footprint, one can get most of the benefits of bank and value conflict avoidance.

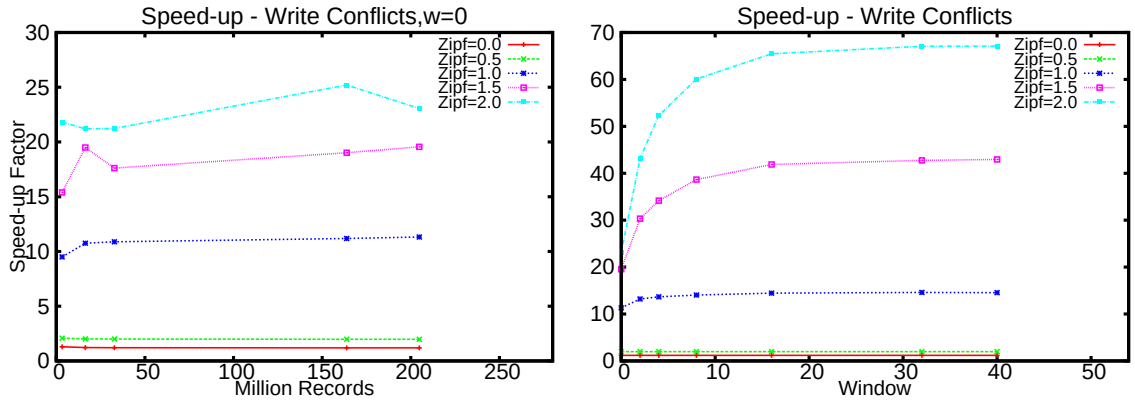


Figure 3.6: Speed-up for Write Conflicts.

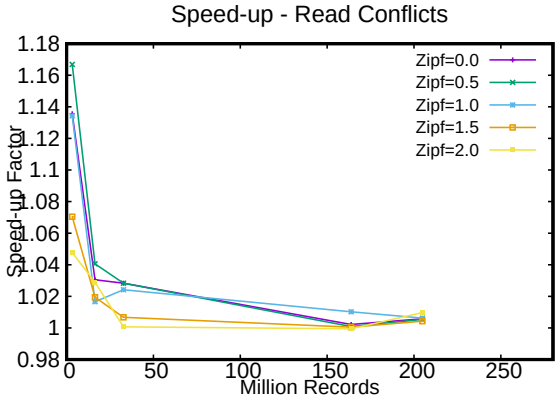


Figure 3.7: Speed-up for Read Conflicts.

3.5.3 Query Performance

Figures 3.6 and 3.7 shows the speed-up of query execution on the GPU for the configurations of Figure 3.2. The write speed-ups are particularly dramatic at high skew, highlighting the importance of addressing conflicts when there are heavy hitters. For uniform data, the speed-up factor is about 1.2, showing that optimizing for write conflicts is still important without heavy hitters. The window size is unimportant for uniform data, but is important for skewed data. Most of the benefit of windowing occurs with a window size of 16 chunks. For reads, the speed-up is much smaller, about 2% or less once there are enough records so that thread scheduling can hide the read serialization latency. As previously noted,

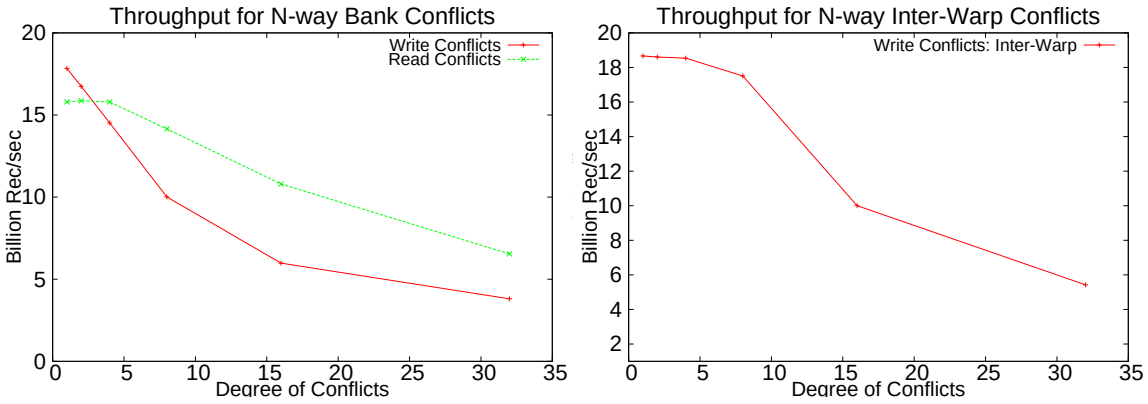


Figure 3.8: Throughput and profiler Counters for varying conflict degree, t=30M.

unlike for writes, skew helps reads because it provides more opportunities for values to be broadcast to multiple threads.

We describe a worst-case scenario for read conflicts where all threads in a warp read a different value on the same bank. We note that since read conflicts are just a sub-case of write conflicts, columns that are both read and written by various queries should be optimized for writes. We used the CUDA command line profiler to count the number of bank conflicts for different degrees of conflicts. The CUDA profiler reports counters per SM. We generated synthetic data causing a specified number of serialization rounds per warp. We were careful to make sure that all values are distinct within a chunk, to show the worst case scenario for read conflicts. Figure 3.8 shows the throughput and the number of conflicts as reported by the `l1_shared_bank_conflicts` counter of the CUDA Profiler [7] [NVIDIA, 2015a]. For the worst case of read conflicts the throughput is less than half of the conflict-free performance. As previously noted, the effect for write conflicts was more significant.

One may wonder whether an access pattern that reads many different elements concentrated in a few banks is realistic. After all, a simple randomization of the bank location would lead to a reasonable spread of items across banks. The following example suggests

[7] In the newest version of CUDA command line profiler the number of read and write bank conflicts are reported separately in counters shared_load_replay and shared_store_replay correspondingly.

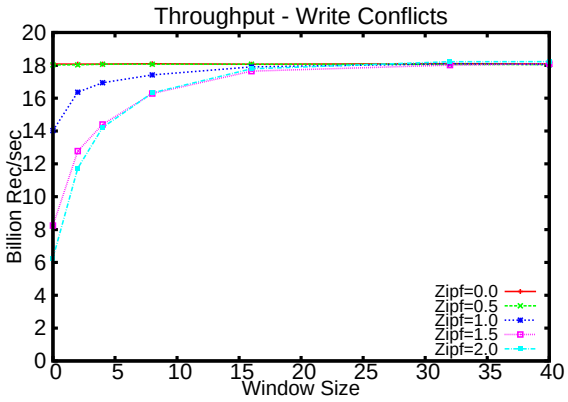


Figure 3.9: Throughput for different windows.

that degenerate cases may indeed arise in practice.

Consider a foreign key join where the foreign key consists of two attributes, x, and y. Imagine that the dimension table represents metadata about cells in an n by m grid, so there are nm rows in total. Suppose that the dimension table is clustered by (x, y). The fact table also contains attributes x and y, but the fact table is clustered by y. As we scan through the fact table we will be repeatedly (for each y value) touching n dimension table rows separated by m rows. If d is the greatest common divisor of m and 32, then this access pattern will use only $32/d$ banks. In the worst case, m is a multiple of 32, and only one bank is accessed.

In Figure 3.9 we show the actual throughput for different windows, where the table size is 200M records. We can process about 18 billion records per second, i.e., about 72GB/sec. To assess the importance of optimizing writes for bank conflicts, we repeated the experiment with a modified algorithm that optimizes for value conflicts but not bank conflicts. The results in Figure 3.10 show that there is a 20% drop in throughput relative to Figure 3.9. Write bank conflicts appear to be more significant than read bank conflicts. Atomic write operations take longer, since they need a read-modify-write cycle.

For $\theta = 2.0$ in Figure 3.10, the performance is better than expected. At this extreme level of skew, there are two heavy hitters that occur many times in each chunk. Both of these heavy hitters have copies in every bank, so they are easy to place. The only possible bank

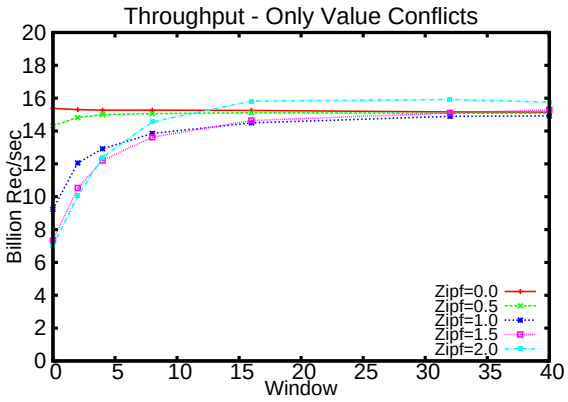


Figure 3.10: Optimizing for value conflicts only.

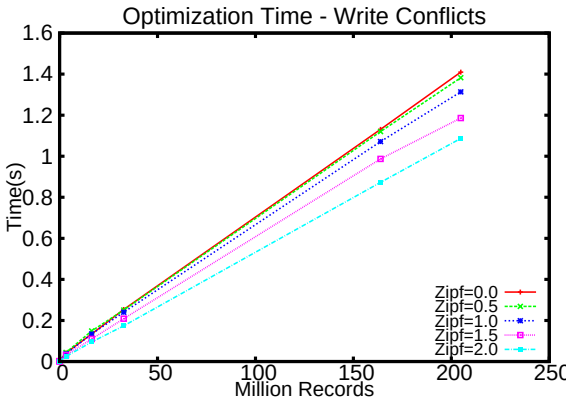


Figure 3.11: Optimization time.

conflicts come from the less frequent items, of which there are just a few in each chunk. The expected number of bank conflicts resembles the birthday paradox: The number of people in a group of size n having the same birthday as another member is proportional to n^2. By removing a subset of items this expectation also decreases quadratically. Thus high-skew distributions indirectly optimize for bank conflicts, even when only value conflicts are explicitly considered by the placement algorithm.

3.5.4 Optimization Speed

In Figure 3.11 we see the time performance of the optimization algorithm for write conflicts. The elapsed time is just a few seconds, even for moderately large window sizes. For increasing skew, the algorithm runs faster because it is easier to arrange the records in a chunk. Our algorithm adjusts to the data distribution by creating many copies for the frequent values, so we have the freedom to place them in any bank and only have to resolve conflicts with the non-frequent values. However, for increasing window sizes, skewed data needs longer optimization time because relocations of frequent values are expensive, due to the high number of copies these items have. Reassignments of frequent items occur when infrequent values with few copies have to displace them.

3.6 Summary & Conclusions

We defined the problem of bank conflicts and value conflicts for data-processing operators on GPU processors. We studied the impact on performance of those two contention factors for two popular OLAP operators on CUDA architecture. We suggested and evaluated a technique for resolving conflicts that can easily be configured for different memory access patterns and space budget requirements. Results indicate that columns that are written by various queries e.g., potential grouping columns, should be optimized for writes and that read conflicts should not be a high priority for bank optimization.

Chapter 4

Multi-Predicate Selections

4.1 Introduction

Here, we are interested in the performance on a GPU of a table scan combined with the application of a compound selection condition. While a filtering scan might be considered a "lightweight" operator compared to a join, it is important for a number of reasons. As the initial operator in a query plan, a filtering scan will typically process a large volume of data compared with later operators that process filtered data. In data analytics scenarios where there is a limited space budget for secondary indices and/or queries touch relatively large segments of the data, scanning the data may be preferable to index-based access methods. GPUs have significantly higher RAM bandwidth compared to CPUs, so achieving high memory bandwidth for GPU scans has a larger potential payoff. In the presence of expensive select conditions, optimally ordering and grouping of conditions is necessary to maximize query performance.

The following code snippets check the conjunction of two conditions on a two column-table stored in global memory. P1 uses branching-and in which a thread retrieves the value of the second column only if the first condition holds. P2 always checks both conditions, but does not require a branch instruction.

```
P1: check=(t1[tid]==val1)  && (t2[tid]==val2);
P2: check=(t1[tid]==val1)  &  (t2[tid]==val2);
```

Consider the execution of P1 on a GPU where the selectivity of both conditions is 0.25. During the first condition check the threads access contiguous memory locations so there is only one coalesced memory transaction per warp. During the second condition check there will be on average 8 threads of a warp for which the first check is successful, so an additional memory transaction will be issued per warp, even though only a subset of the threads evaluates the second condition.[1] In P2, all threads retrieve both values from the two columns but the memory requests can be scheduled more efficiently. We quantify this improvement by running short tests on our target machine. Depending on the selectivity of the first condition the improvement of P2 over P1 can be up to 25%.

An alternative implementation of this compound condition would apply each condition in a separate kernel, with the two kernels executed serially. The first kernel applies the first condition and outputs the records passing the check, which is the subset of records on which the second kernel will apply the second condition. In this way there will be reduced thread divergence but also increased traffic to the global memory.

Branches hurt the performance of CPU programs too. For datasets stored in main memory, branch mispredictions can significantly degrade program performance. Compound select conditions need to be optimized given the CPU, memory characteristics and condition selectivities [Ross, 2004]. Our approach is motivated by the CPU optimizations but there are significant differences since GPU branch divergence and CPU branch misprediction are different phenomena.

4.2 Related Work

A subset of SQLite commands has been implemented on GPUs [Bakkum and Skadron, 2010]. A query is translated to a single kernel executing all operators and operators are mapped to their corresponding opcodes. Some opcodes will only be executed by a subset of threads, for example when a filter condition is true for only a subset of threads in a warp,

[1]Compilers sometimes use branch predication so that some instructions in branches are not skipped, to maximize efficiency of instruction scheduling. Predication adds the overhead of pipeline slots with null operations in some threads on each case of a branch. Typically there is a limit on the number of instructions to which this heuristic is applied [NVIDIA, 2015c].

so thread divergence caused thread underutilization.

Evaluating multiple predicates in a single-kernel is reminiscent of the kernel fusion technique suggested for data warehousing applications [Wu *et al.*, 2012b]. Kernel fusion combines operators of a query execution plan into a single CUDA kernel. A framework optimizing which query operators to fuse has been suggested [Wu *et al.*, 2012a]. Kernel fusion results in reduced data transfer through the GPU memory hierarchy, but it might increase register and shared memory pressure. Here, we focus on the increased thread divergence caused by the fusion of multiple select operators, and on how branching reduces instruction scheduling efficiency. We assume a columnar memory table layout; [Diamos *et al.*, 2013] assumes a row-store with full records packed into a 32-bit word. We also assume that data are GPU memory resident and that no transfers are required between the CPU and GPU.

We suggest a software-based optimization to reduce thread divergence. Other software-based solutions eliminate thread divergence by thread-data remapping [Zhang *et al.*, 2010], which has been extended to remove additional code irregularities, e.g., irregular memory references [Zhang *et al.*, 2011]. Iteration delaying and branch distribution reduce the performance impact of branch divergence on programs [Han and Abdelrahman, 2011]. Loop-splitting reduces register pressure caused by thread divergence [Carrillo *et al.*, 2009]. Software-based branch predication has been suggested for AMD GPUs to reduce branch penalties [Taylor and Li, 2011].

Various hardware extensions have been suggested for thread divergence elimination not just for GPUs but for SIMD processors in general: dynamic regrouping of threads into new warps [Fung *et al.*, 2007], adjusting SIMD width based on branch or memory latency divergence [Meng *et al.*, 2010], and identifying reconvergence points for threads [Diamos *et al.*, 2011].

In this chapter, we study how branch penalties affect the performance of compound select conditions on a GPU and our solution is adapted from algorithms minimizing branch misprediction rate for main-memory databases running on a CPU [Ross, 2004]. The most efficient plan optimizes the combination of branching-and (&&) and logical-and (&) operators. We suggest two polynomial algorithms: The first also optimizes combinations of branching-and and logical-and operators. The second optimizes which conditions to fuse

into a single-kernel. In the presence of expensive predicates the optimal condition ordering is in ascending order of the rank metric: $\frac{selectivity-1}{cost-per-tuple}$ [Hellerstein, 1998]. Optimal execution strategies for select queries involve choosing between late and early materialization [Abadi *et al.*, 2007]. A query plan compilation framework has been suggested that, in addition to minimizing CPU costs, maximizes the lifetime of the data within the registers [Neumann, 2011].

4.3 Problem Setting

We assume an OLAP setting and a GPU-friendly column-store memory layout. Data is resident in GPU global memory. Queries posed are the conjunction of equality or range conditions on one or multiple columns. Compound select queries with many conditions are also common in scientific databases. We group all conditions on a single column into one so in the rest of this Chapter we assume that every condition is applied on a different column.[2] If there is an index on one or more attributes, we could apply the conditions on the indexed attribute(s) and use our technique for the remaining conditions.

4.4 Execution Strategies

4.4.1 Single-Kernel Plans

In a program implementing a conjunctive select query, all threads in a warp check 32 consecutive values of the column the first condition constrains. If the condition is not satisfied for some of the threads then these threads will not evaluate the remaining conditions. However, they will remain idle until all threads in the warp finish condition evaluation. Also, similarly to CPUs, the scheduling of instructions in a branch statement is less efficient. The choice of the most efficient plan depends on the selectivity and the predicate cost. The left-to-right order of conditions is determined by the rank metric mentioned above. We show below the code snippets for all single-kernel plans of a three-condition query.

[2]Extending the cost model to conditions like `c1[i]=c2[i]` that mention more than one column is straightforward.

```
S3:   check = c1[i]<v1 & c2[i]<v2 & c3[i]<v3
S21:  check = (c1[i]<v1 & c2[i]<v2) && c3[i]<v3
S12:  check = c1[i]<v1 && (c2[i]<v2 & c3[i]<v3)
S111: check = c1[i]<v1 && c2[i]<v2 && c3[i]<v3
```

Figure 4.1 shows the execution path of the three conditions in the S111 plan.

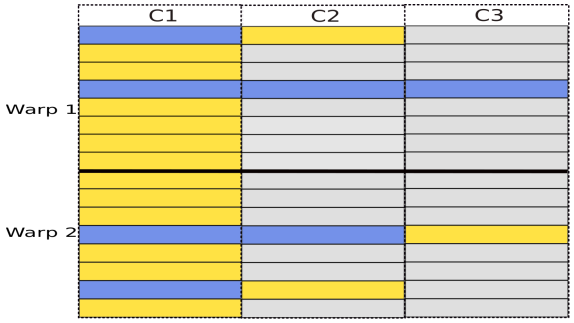


Figure 4.1: Execution of the S111 plan for two warps.

Using the implementation techniques described in [Ross, 2004], S3 needs no branches, and is thus called the "no-branch" plan. S21 and S12 each need one branch (depending on the compiler, which may choose predication rather than branching), and S111 needs two branches.

Figure 4.2 shows the performance in milliseconds of all plans for a conjunctive query computing a count aggregate with four conditions having the same selectivity, on a 128 million row table. Each plan is implemented in a separate CUDA kernel and they only vary in how they evaluate the four conditions. The no-branch plan S4 is the most efficient for selectivities ≥ 0.4 where the combined selectivity for the first three conditions is greater than $\frac{1}{32}$. In such a case, there will typically be at least one thread per warp satisfying the first three conditions. This means that one memory transaction has to be issued for each of the four columns for all plans, so the time cost of all plans remains constant after this value; the difference in performance comes from branch penalties. In the range [0.2-0.35] the optimal plan is S31. Again we note that for selectivity values ≥ 0.2 there will be one memory transaction for the first three columns. In the range 0.05-0.2 the optimal plan is S211. In CPUs that depend on branch prediction, the branch penalty is highest when the

probability that the branch being taken is close to 0.5 [Ross, 2004], a phenomenon we do not see on GPUs.

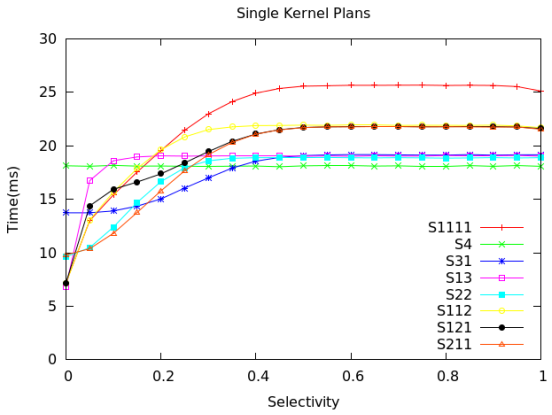


Figure 4.2: Time performance in ms of different plans for a query with four conditions.

4.4.2 Multiple-Kernel Plans

Depending on the selectivity and the evaluation cost of the conditions, it may be more efficient to execute the query by running separate kernels each checking a subset of the conditions. All but the last kernel write the intermediate query results to the global memory. Depending on the selectivities of the conditions, the extra write cost may be justified by the reduction in thread divergence: Every thread is active in later kernels testing later conditions. In presence of expensive predicates, fused kernels could exhibit higher resource underutilization.

Each query execution plan has one or more kernels, each checking one or more conditions. For example $[c_1][c_2c_3c_4]$ denotes a plan for a four-condition query consisting of two kernels. The first kernel evaluates the first condition and writes the positions/record-ids of the satisfying records into the global memory. The second fused kernel checks all three remaining conditions, with the best of the equivalent single-kernel plans described in the previous section. The space of alternative plans includes the different groupings of the n conditions. Because in optimal plans select conditions are ordered by the rank cost metric, we need only consider one left-to-right ordering of the conditions, as before. This means that plan $[c_1][c_2c_3c_4]$ can be denoted unambiguously by `K13`. We call the `K4` plan fuse-only, and the

rest of the plan space uses kernel fusion on a subset of the conditions.

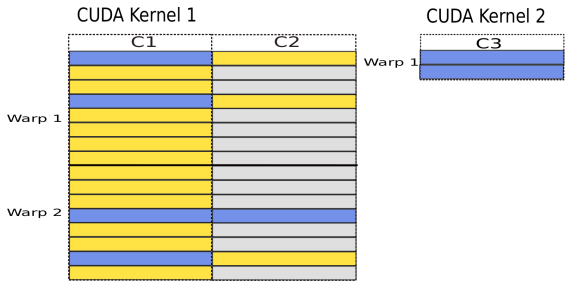


Figure 4.3: Execution of the K21 plan for two warps.

Figure 4.3 shows the execution path of the `K21` plan for three conditions.

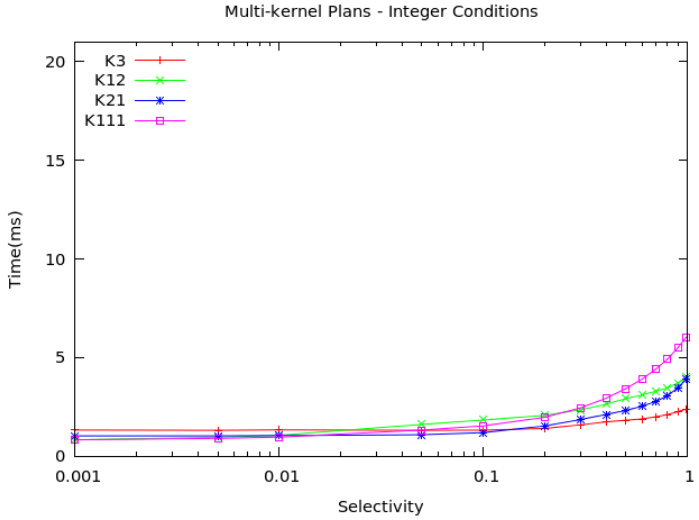


Figure 4.4: Performance of multi-kernel plans when varying the selectivity of the first condition.

Figure 4.4 shows the time cost of different plans, for a three condition query and a table with 12.8 million rows. We execute a conjunctive query and vary the selectivity of the first condition. The last two conditions have fixed selectivity 0.5 and the query output is a fourth column. In each kernel only the third condition includes a branch (`K3` is implemented as `S21`). All conditions are integer comparisons. Figure 4.5 the second and the third diagram the last predicate is a substring match on a string column with 8 and 16-character strings correspondingly. We observe that for varying selectivity and predicate

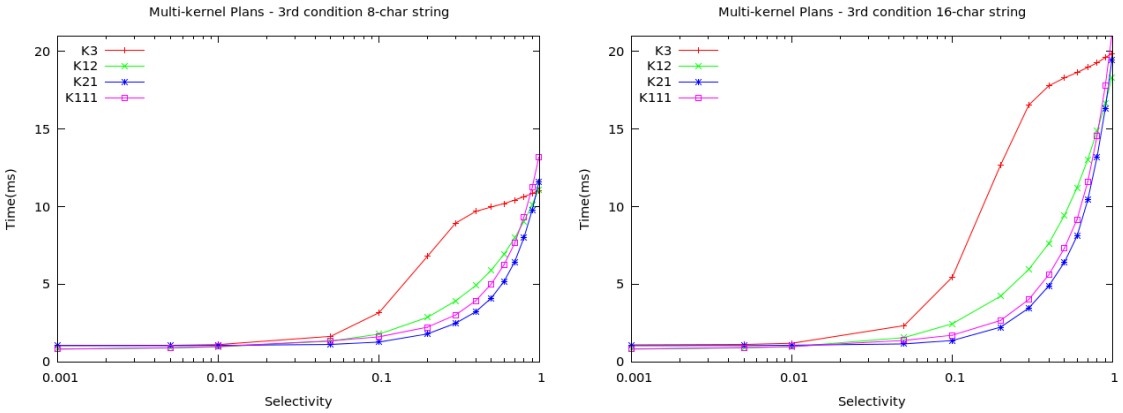


Figure 4.5: Performance of multi-kernel plans when varying the selectivity of the first condition.

evaluation cost different plans are optimal, so we should optimize plan selection.

In the left diagram for higher values of selectivity (less selective conditions) fuse-only plans are the fastest because multi-kernel plans have increased write cost. In the middle and last diagram K21 is typically the most efficient because it evaluates the expensive predicate on a reduced subset of rows. K111 also evaluates the expensive predicate for fewer rows but due to the increased write cost is faster only for very selective predicates (selectivity ≤ 0.01). In fuse-only plans, even if only one of the values a thread reads satisfies the first two conditions the remaining threads have to wait for this thread to evaluate the expensive string-match operation. For 16 character strings and selectivities in [0.1-0.4] range K3 is 3-5X slower than K21 and K111 plans although the gap narrows as selectivities increase.

4.5 Query Cost-Model

4.5.1 Single-Kernel Plans

Based on these results we conclude that conjunctive select queries need to be optimized. Our optimization algorithm uses a cost model estimating the global memory cost of different execution plans. Due to CUDA scheduling, the time cost for a two-column scan is less than twice the cost of a single-column scan when memory loads from both columns are branch free. If the memory load from the second column is included in a branch and there is at

least one thread in a warp fetching the second column value, the cost of the two-column scan is twice the cost of a single-column scan. We will describe now how we quantify the improvement from efficient instruction scheduling of branch-free memory loads. Say the time cost for a k-column table scan is $scan_k$. If r is the number of table rows and w is the warp size then the number of memory transactions for a sequential scan is $\frac{r}{w}$. We account for the improvement from scheduling compared to the expected performance by computing in advance (during a calibration phase) the following rate for each k:

$$r_k = \frac{scan_k}{k \times scan_1}$$

For example, the cost of the plan $(C_1$ `&` C_2 `&` $C_3)$ `&&` C_4 is:

$$r_3 \times \frac{3r}{w} + min(sel_1 * sel_2 * sel_3 * w, 1) * \frac{r}{w}$$

4.5.2 Multiple-Kernel Plans

The total cost for a multi-kernel plan is the sum of costs of all individual kernels which can be estimated as suggested in Section 4.5.1, plus the cost of writing the intermediate query results. In Appendix 8.4 we explain how shared memory is used as a buffer to achieve coalesced writes to global memory. While we have described a method that writes the record-ids of intermediate records, we also consider a variant that instead writes all of the column values that are needed in the subsequent kernels. The intermediate results are stored column-wise to be read efficiently from the subsequent kernels. While this variant may need to transfer more data, subsequent accesses will be better aligned for coalesced access.

The cost of writing the intermediate results for plan K111 and a query that selects all columns is:

$$write\ cost_{col} = \sum_{i=1}^{n-1} \frac{sel_i! \times r \times n}{w}$$

where $sel_i!$ is the combined selectivity for conditions 1 to i and $sel_0! = 1$. If the output of a kernel is a record-id list we have to add the cost of reading the list: $\frac{r}{w} \times sel_{i-1}!$, except for the first kernel which reads data sequentially. Also, all kernels but the last have to write

the record-ids of the satisfying rows:

$$write\ cost_{rid} = \sum_{i=1}^{n-1} \frac{sel_i! \times r}{w}$$

If we use record-id lists there will be sequential memory access only in the first kernel. For the subsequent kernels the threads will read in a coalesced way only the record-id lists containing the positions of satisfying records but the actual values might not be stored contiguously. To account for that we have to adjust the read cost by using the random access memory bandwidth of our target machine.

4.5.3 Model Calibration

To factor-in the improvement by efficient CUDA scheduling of memory transactions for branch-free code we calibrate our model by measuring the scan performance for 1 to n-column scans, where n is the number of conditions in the WHERE part of the query. The calibration step is short because it only has to run once, on a relatively small number of table records. We calibrate our model using tables containing 16 million rows. On our machine, for 4-byte integer scans and 1024-thread block size the rate r_2 for a two-column scan is 0.75. For three and four columns the rate is around 0.68. This means that when measuring the cost of no-branch plan $C_1\&C_2$ the weighed number of memory transactions accounted for by our model will be $2 \times 0.75 = 1.5$ rather than 2. The constants of our model depend on the chosen thread block size/configuration since the performance of CUDA kernels also depends on the kernel configuration. If there are multiple configurations to choose from, then we should compute the described rates for each, and when optimizing a query we must use the rates for the right configuration. Here, we are showing the results for 4-byte integer columns but our model can be used for different data-types.

4.6 Optimization Algorithm

Here we describe our suggested optimization algorithm minimizing branch penalties. We suggest two different methods. The first method computes the optimal execution plan for single kernel queries with n conditions and the second method is a multi-kernel optimization

```
Generate all no-branch plans for subsets p ∈ P, storing their costs in A[p], with null left
and right attributes
for r ∈ P in increasing order r right child of the plan do
    for l ∈ P in increasing order such that rank(l.last)+1=rank(r.first) do
        cost=(l && r).cost
        if cost < A[r ∪ l].cost then
            A[r ∪ l].cost=cost
            A[r ∪ l].left=l
            A[r ∪ l].right=r
        end if
    end for
end for
```

Figure 4.6: Single-Kernel Optimization Algorithm.

algorithm determining which conditions will be applied in a fused kernel. For a small number of conditions we could enumerate all possible plans, but the number of all possible plans grows factorially so we need more efficient alternative algorithms. Also the high performance of GPUs implies that optimization time must be small. We assume that attributes are not correlated. If this is not the case we might want to enumerate all possible condition orders and choose the one with the minimum estimated cost.

Both algorithms have the same complexity. To compute the optimal plan for a query with n conditions the time complexity is $O(n^3)$.

Single Kernel Optimization We present a polynomial dynamic programming algorithm for computing the optimal execution plan with n conditions. We let P denote consecutive nonempty subsets of the n conditions that have been ordered by the rank cost metric. For $n = 4$ there are 10 such subsets: 4 with a single condition, 3 with two conditions, 2 with three conditions, and one with four conditions. In general $|P| = n(n + 1)/2$.

In the initial step we generate all no-branch plans for subsets in P. Our algorithm has an outer and an inner loop corresponding to the right and left subplan. We iterate over

plans in increasing order, which means if p_1 is a subset of p_2 then p_1 comes earlier in the loop than p_2. We use any generated subplan as part of the larger plans that are generated in later iterations. In the end, the optimal plan for n conditions is contained in corresponding position of the A array. Each position of this array contains the left and right subplan of the optimal plan, and the cost of this plan. We can retrieve the optimal plan using the left and right subplan.

There are $O(n^2)$ iterations of the outer loop, and $O(n)$ iterations of the inner loop because of the requirement that rank(l.last)+1=rank(r.first). Thus the overall complexity is $O(n^3)$.

Multi-kernel Optimization The algorithm that determines the optimal plan by splitting query execution into multiple kernels is similar to the single-kernel optimization algorithm. We use the same algorithm skeleton by varying the cost estimation when combining two plans. When combining two plans the number of kernels in the combined plan will be the sum of the number of kernels in the left and right plan and the cost is the sum of their cost plus the intermediate result writing cost to the global memory. The optimal plan determines which conditions to fuse similarly to how an optimal single kernel plan uses logical-and (&) rather than the branching-and (&&) operator. Internally, the algorithm calls Algorithm 4.6 to find the best fused plan for the included conditions.

4.7 Experimental Evaluation

4.7.1 Experimental Setup

GPU performance was measured using 1024 threads on an Nvidia Tesla C2070 machine with 6GB of RAM and a nominal bandwidth of 144GB/s. Optimization was done on a dual-chip Intel E5620 with nominal bandwidth 25.6GB/s using a single-thread. We assume the data to be GPU resident so we do not measure the time needed to transfer the data to the GPU memory. For reference a single column scan of a 128 million row table computing a count aggregate takes 6.5 ms corresponding to a bandwidth of around 80GB/s. Our kernels are typically memory bound. CUDA C code was compiled using the `nvcc` compiler of the

CUDA toolkit 4.2 using full optimization. We used synthetic data with 128M rows and 4-byte integer columns. We ran the following handcompiled queries:

```
Q1: SELECT  C4 FROM  T
  WHERE  C1<v1 AND C2<v2 AND  C3<v3
Q2: SELECT COUNT(*) FROM  T
  WHERE  C1<v1 AND C2<v2 AND  C3<v3 AND  C4<v4
```

4.7.2 Cost Model Validation

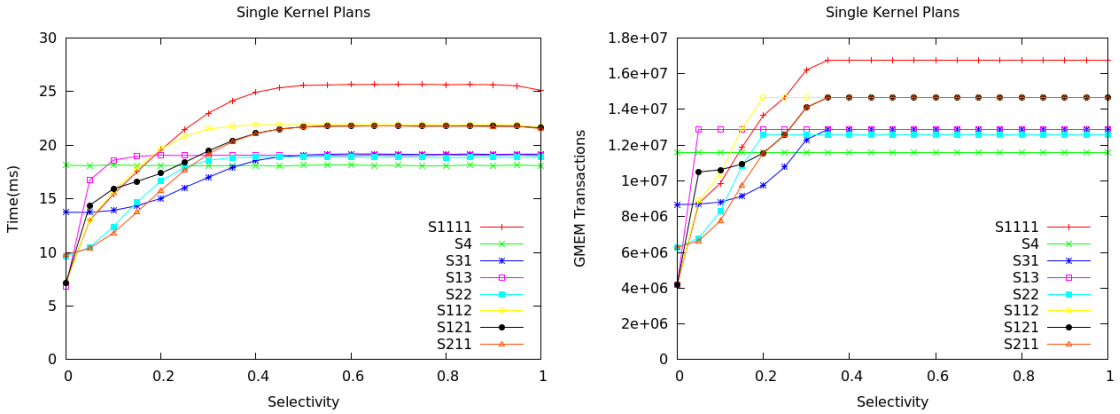


Figure 4.7: Actual and estimated performance for different single-kernel plans of Q2.

Figure 4.7 validates the model for conjunctive queries executed in a single kernel for query Q2, where all conditions have a common selectivity that is varied. We observe that the trends of different plans in the left diagram (actual times, the same as Figure 4.2) closely resemble those in the right diagram (optimizer estimates) proving that our cost model can be used in single kernel optimization. Different plans are optimal in different selectivity ranges. For completeness, Figure 4.8 shows the performance of the same plans on the CPU using 16 threads. CPU plans are 4.5–7 times slower, so even for relatively lightweight operators like compound selections, GPUs offer better performance due to higher memory bandwidth.

In Figure 4.9 the left figure shows the performance of different plans for Q1 using record-ids as the intermediate data representation between kernels. The three conditions

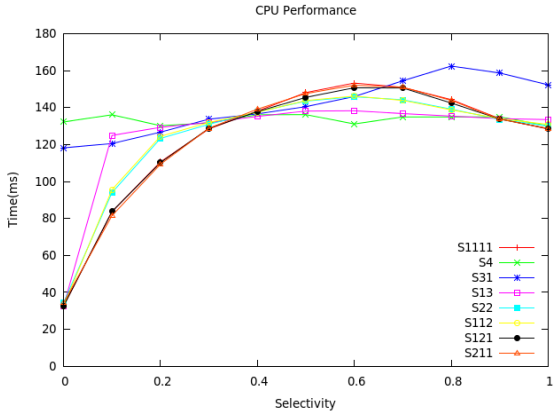


Figure 4.8: Time performance of different plans on the CPU for Q2.

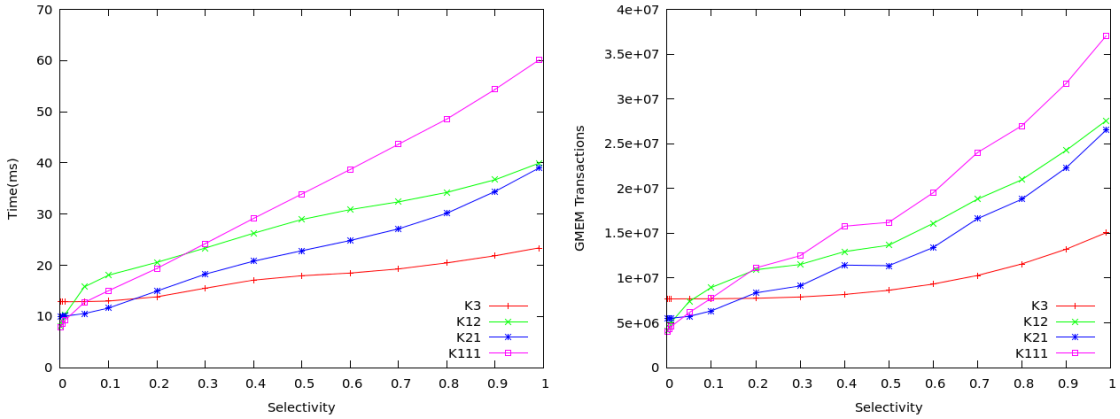


Figure 4.9: Actual and estimated performance of multi-kernel plans for Q1 varying the selectivity of all conditions.

have the same selectivity and the right diagram shows the weighed number of global memory transactions. The similar trends in the two diagrams indicate that memory cost is a valid metric for multi-kernel optimization too. We also note that for such queries it is faster to execute all conditions in a fused kernel when the conditions are not selective, although as we saw in Section 4.4.2 the case is different in the presence of expensive predicates. K3 is less efficient for very selective conditions: For selectivities below 0.01 the K111 plan is the fastest, and for selectivities between 0.01 and 0.1, K21 is the most efficient.

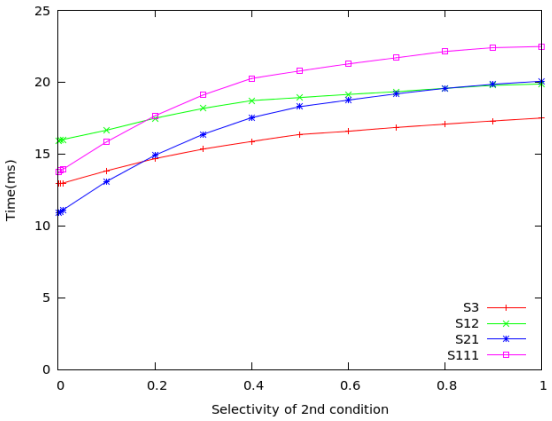


Figure 4.10: Time performance of single-kernel plans for Q1 for varying predicate selectivity.

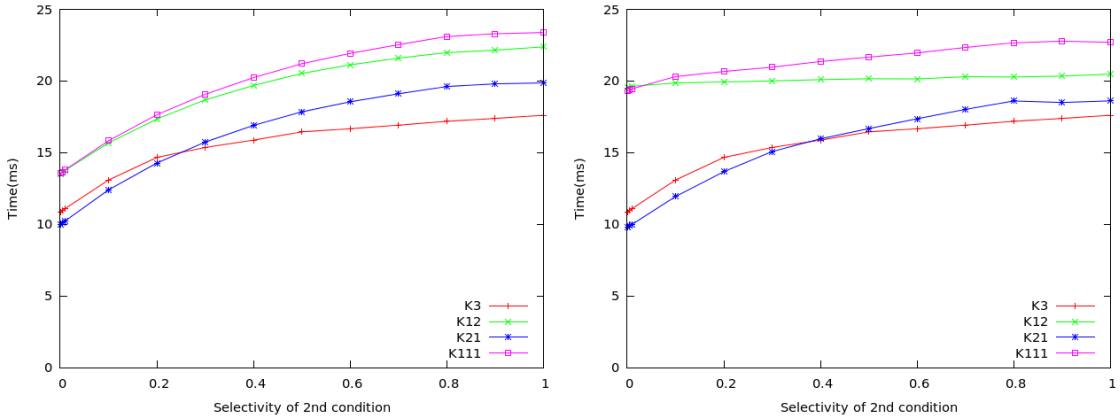


Figure 4.11: Time performance of multi-kernel plans for Q1 or varying predicate selectivity.

4.7.3 Query Plan Space

In the next experiment we vary the selectivity of the second condition while the first selectivity is fixed to 0.1 and the third to 0.8. Figure 4.10 shows the performance of single-kernel plans and Figure 4.11 the performance of multi-kernel plans using rid-lists for intermediate results and when writing full column values as intermediates. Multi-kernel plans seem to provide a performance improvement for selective conditions. For example the K21 plan is around 5% faster than the S21 plan for low selectivities. For multiple kernel plans, K21 is the optimal up to 0.2 when writing record-ids and 0.3 when writing column values, because the combined selectivity of the first two conditions is low and it is more efficient to write the reduced set of results in global memory and then apply the third condition. For most

plans it is slightly more efficient to write the column values than using rid-lists, except for selective queries on plans such as K111 and K12 where there is more intermediate data to be written. In general, the choice of intermediate result format would depend on the columns in the SELECT clause of the query and the conditions' selectivities.

4.7.4 Optimization Speed

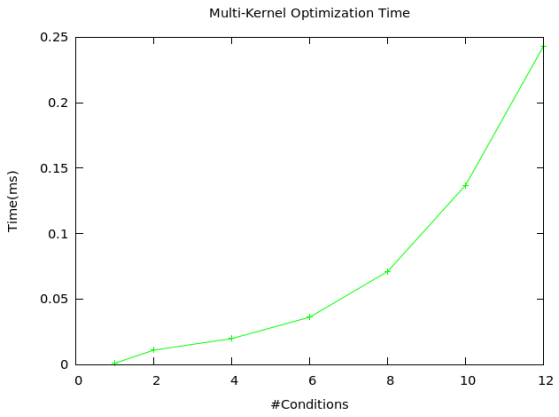


Figure 4.12: Execution time of the multi-kernel optimizer as a function of the number of conditions.

Figure 4.12 shows the performance of the multi-kernel optimizer (running on the CPU) as a function of the number of conditions. Optimization time even for relatively many conditions is a fraction of a millisecond suggesting that the optimization cost is low compared to the query cost even in high performance environments like GPUs.

4.8 Summary & Conclusions

In this Chapter we studied the effect of branch penalties and thread divergence on the performance of compound select conditions for tables stored in GPU global memory. We suggest a cost model that accurately predicts the performance of query execution plans with varying branching and divergence behavior. Our results suggest that different plans are optimal for different sets of selectivities and predicate evaluation costs, and that our optimization is able to find these plans.

Chapter 5

Sub-string Matching Acceleration

5.1 Introduction

Relational queries often contain string matching conditions within the LIKE predicate. For example, the following query appears as a subquery of Q16 of TPC-H [Transaction Processing Performance Council, 2014]:

```
select s_suppkey
from supplier
where s_comment like '%Customer%Complaints%'
```

The "%" character is a wildcard that can match an arbitrary number (including 0) of characters. In this way, the SQL LIKE predicate allows for a limited form of regular expression matching. When evaluating such queries, the system does not need to find all occurrences of a pattern in a string, or even the position of the match(es) within the string. It is therefore possible to apply optimizations for string matching that might not be possible in a more general context, such as terminating string matching early as soon as a match is found.

As discussed in Section 1.2, GPUs are becoming popular for data processing tasks because of their high memory bandwidth and abundant parallelism. Modern GPUs have a moderate amount of on-board RAM. For example, the Nvidia K40 has 12GB of RAM, with a potential access bandwidth of 288GB/sec. For the purposes of string matching, we pro-

pose that the GPU RAM be used to store the string columns for a database system, while the CPU RAM is used to store the remainder of the database. Multiple GPU cards can be combined to store and process (in parallel) larger string collections. String matching can then run at GPU speeds without needing large data transfers between the CPU and GPU. Only string identifiers would be communicated between the CPU and GPU.

5.1.1 String Matching Acceleration

5.1.1.1 Algorithmic Background

Indexes can speed up search dramatically for selective queries. For string data, the state-of-the-art index structure is the suffix tree [Weiner, 1973]. Unfortunately, even very efficient implementations of suffix trees (or suffix arrays [Ferragina and Manzini, 2005]) are an order of magnitude larger than the string data being indexed [Karkkainen and Ukkonen, 1996; Ferragina and Manzini, 2005; Tian *et al.*, 2005]. There are suffix arrays, a compressed form of suffix tree, which can have as low consumption as 1.25X depending on the number of bits used to encode a symbol [Ferragina and Manzini, 2005].

It is unclear whether allocating the extra space is a good use of resources. What is more, a suffix tree lookup does not provide a complete solution for SQL LIKE conditions. If there are many substrings within a LIKE clause, separated by "%" symbols, one would need to search the index once for each of the substrings, intersect the matching string identifiers, and then verify that the substrings occur in the correct order, without overlap.

String matching is a well-studied problem. Two classic algorithms are the Knuth-Morris-Pratt (KMP) algorithm [Knuth *et al.*, 1977] and the Boyer-Moore (BM) algorithm [Boyer and Moore, 1977]. To search for a sequence of strings as specified in a LIKE clause with "%" symbols, one can search for each string in turn using any string matching algorithm, starting each search where the previous search left off. On CPUs, empirical studies have shown that BM is generally superior to KMP [Crochemore and Lecroq, 1996] because it facilitates larger jumps through the target string. Parallelizing these algorithms on CPUs is straightforward: each available thread can be used to independently search a different string in the database.

A database system should be able to give robust and stable performance without per-

formance "surprises" for particular inputs. On GPUs the worst case is amplified due to the SIMT architecture. If one string in a warp exhibits worst-case behavior, then all threads in the warp will suffer the latency consequences. For this reason, we exclude algorithms (such as Boyer-Moore-Horspool [Horspool, 1980] and Quick-Search [Sunday, 1990]) whose worse-case behavior is $O(m \times n)$ where m is the pattern size and n the string size. We briefly describe several well-known string searching algorithms, and state their time complexity.

Boyer-Moore (BM) [Boyer and Moore, 1977]. BM preprocesses the pattern and creates two shift tables: BM bad character and BM good suffix tables. The good-suffix table stores the shift value for each character in the pattern. The bad-character table stores a shift value for each character in the alphabet, with the shift value being based on the occurrence of the character in the pattern. BM compares the pattern with the string from right to left. The worst-case performance is $O(n + m)$ steps [Apostolico and Giancarlo, 1986]. BM can skip over large parts of the input string if the last character of the pattern does not match the input string.

Knuth Morris Pratt (KMP) [Knuth *et al.*, 1977]. KMP preprocesses the pattern storing the necessary information in the partial match table. The partial match table, which we call *next*, stores how far we have to backtrack if a comparison of the current position in the pattern with the input fails. Unlike BM, KMP compares the pattern to the input string from left to right and in case of a failure it shifts the pattern rather than the input based on the partial match table. KMP has worst case $O(n + m)$ time complexity. In Section 5.3.3 we present and evaluate three alternative KMP implementations.

Aho-Corasick (AC) [Aho and Corasick, 1975]. AC is a generalization of KMP for multiple patterns. The complexity is linear in the length of the patterns and length of the input string. It uses a deterministic finite automaton (DFA) for the matching process. An equivalent Non Deterministic Finite state Automaton (NFA) can be designed, which is typically more space efficient, but results in slower matching performance. In the context of database query processing we advocate the use of a DFA for string matching.

Multi-pattern matching has been accelerated on Cell Processors for Intrusion Detection Systems [Scarpazza *et al.*, 2007; Iorio and Lunteren, 2008]. Cell processors similarly to GPUs require memory coalescing and in order to use the SIMD instructions different streams

processed in parallel are interleaved in SIMD registers.

RE2 is an optimized high performance regular expression matching library developed by Google [RE2, 2014]. RE2 uses a DFA to detect whether a pattern appears in an input string. Boost is a set of libraries implementing, among others, the KMP and BM algorithms [Boost, 2014]. The SSE 4.2 instruction set provides the CMPISTRI instruction which implements a substring search for patterns fitting in an SSE register [Intel, 2011].

We focus on exact string matching but there is also a large body of work focusing on approximate string matching, allowing mismatches [Navarro, 2001]. Some popular algorithms that are used extensively in computational biology are Smith-Waterman and Needleman-Wunsch [Needleman and Wunsch, 1970; Smith and Waterman, 1981]. Approximate pattern matching can also be implemented using Hidden-Markov models [Bhargava and Kondrak, 2009].

5.1.2 String Matching on GPUs

Significant speed-ups were obtained for the GPU implementation of multi-pattern matching algorithms Wu-Manber [Pyrgiotis *et al.*, 2012] and Aho-Corasick [Lin *et al.*, 2010; Zha and Sahni, 2013]. Also a multi-pattern version of BM was implemented but had inferior performance to AC [Zha and Sahni, 2013]. Strings were stored contiguously and to reduce the global memory latency, input is first loaded in the shared memory to achieve memory coalescing. NFA matching has been implemented to reduce the space required for the automaton [Cascarano *et al.*, 2010]. String pivoting in 4-byte units has been suggested for NFA-based string matching to achieve coalescing [Zu *et al.*, 2012]. Offloading string matching onto GPUs has been used to accelerate a digital forensics tool [Marziale *et al.*, 2007] and in intrusion detection system computation [Fisk and Varghese, 2004; Jacob and Brodley, 2006; Vasiliadis *et al.*, 2011].

A simplified version of KMP in older GPU architectures did not result in performance speed-ups under normal load conditions [Jacob and Brodley, 2006]. KMP has been implemented on the GPU using shared memory to store the pattern and the partial match table [Bellekens *et al.*, 2013]. It faced similar issues with thread divergence and it had suboptimal performance compared to PFAC for multiple pattern searches. Loop unrolling in the

code of KMP resulted in a minor performance improvement. KMP has been implemented on multi-GPUs [Lin *et al.*, 2013c]. However, there were no exact performance numbers reported, just the speed-up compared to a CPU implementation. Various single pattern matching algorithms have been evaluated, taking advantage of the various GPU memories [Vasiliadis *et al.*, 2011].

Implementations of AC on GPUs have used device memory [Zha and Sahni, 2013] and texture memory to store the DFA [Lin *et al.*, 2010; Vasiliadis *et al.*, 2011]. There are multiple ways to parallelize string matching on a set of input strings by varying the mapping of threads to strings. For example in the PFAC library a thread is allocated for each input byte and each thread starts matching from the given byte offset [Lin *et al.*, 2010; Lin *et al.*, 2013a]. If no match is found, the threads terminate searching without backtracking the state automaton. PFAC also uses state number reordering to check more efficiently whether an accepting state has been reached, which we also use in our implementation of AC. PFAC has good performance for intrusion detection systems but its performance varies by more than order of magnitude for adversarial inputs [Lin *et al.*, 2013a]. (Our AC implementation uses shared memory to store the transition table. This is a realistic choice for database queries where there is a reasonable number of patterns in a single query.) PFAC uses texture memory to store the transition table but it caches the initial row in the shared memory.

The GPU implementations of Needlman-Wunsch and Smith-Waterman algorithms have resulted in high performance improvements over CPU implementations [Ligowski and Rudnicki, 2009; Liu *et al.*, 2009; Farivar *et al.*, 2012]. Hidden-Markov models are computationally intensive so evaluating them on GPUs resulted in significant performance speed-ups exploiting the high parallelism and memory bandwidth [Li *et al.*, 2009].

5.1.3 String matching on GPU Databases

Many GPU databases have been implemented following different co-processing approaches for the GPU and CPU processors [Breß *et al.*, 2014]. Typically GPU databases first apply dictionary compression on strings and process the compressed representations in the GPU memory [Breß *et al.*, 2014]. However, not all string predicates, such as wildcards, can

be answered using just the compressed representation of the strings. String processing has been identified either as an unsuitable application for GPU acceleration [Rauhe *et al.*, 2013] or it has been identified as a still open problem [Pirk *et al.*, 2014]. A data structure called tablet has been suggested, that handles variable-length data as strings for a GPU database [Bakkum and Chakradhar, 2012]. Redfox is a run time framework for database queries that runs all TPC-H queries [Wu *et al.*, 2014]. Strings are stored in different tables with each string table storing different length strings. Significant speedups were observed on queries containing string matching predicates but the matching algorithm was not specified. Each thread performed matching on independent strings resulting in branch and memory divergence. We consider alternative ways to map GPU threads to input strings and also alternative GPU device string memory layouts. MapD is a Big Analytics platform taking into advantage the high parallelism and fast memory of GPU processors and uses indexes to search tweet contents [Mostak and Graham, 2014]. Other state-of-the art GPU database systems are Parsteam [Michael Hummel, 2010] and SQream [Ori Netzer, 2014] but there is not available documentation on their string matching approach.

String processing has also been identified as an application that can be accelerated using vector processors [Zukowski, 2009]. GPU processors have different threads of the same warp executing in each step rather than different data lanes. We believe that our approach can also be adapted for SIMD processors.

5.1.4 Thread-Divergence on GPUs

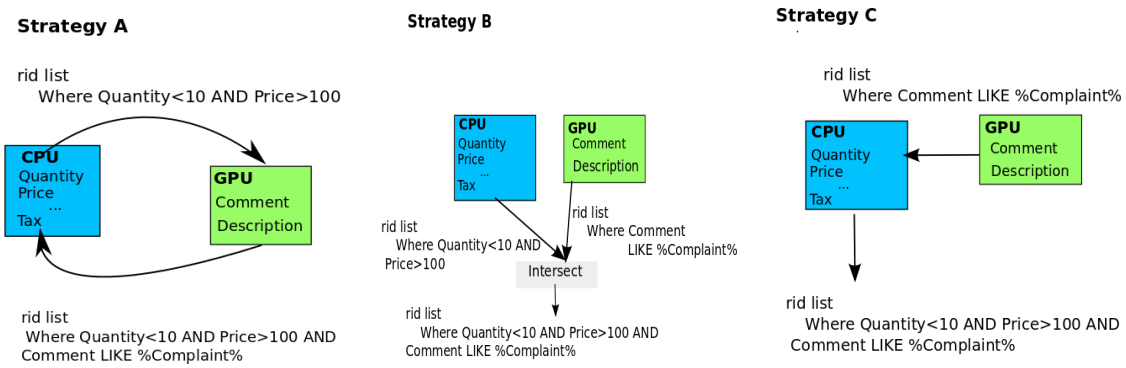
Chapter 4 discussed how compiling queries to multiple kernels and writing intermediate results to the global memory can reduce divergence. Another software based solution reduces thread and memory divergence for applications that can tolerate errors [Sartori and Kumar, 2013]. Different warp scheduling policies have been suggested to reduce resource utilization on GPUs [Narasiman *et al.*, 2011].

5.2 Cache Pressure

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. On the Nvidia K40, there are 15 processing elements called "stream multiprocessors" (SMs), each capable of running 64 concurrent warps. When running at full capacity, there may be $15 \times 64 \times 32 = 30{,}720$ threads in flight. The L2 cache, which is shared by all processing elements has 12,288 128-byte cache lines. With many more threads than cache lines, any algorithm that tries to assign threads independent work is liable to thrash in the L2 cache if those threads each access even a single cache line.

A second kind of cache pressure arises from the limited bandwidth of the L2 cache, which is 1024 bytes per cycle. The cache access granularity is 32 bytes. Each SM can dispatch two independent instructions per thread for each of four concurrent warps. If independent L2 accesses were to occur in every instruction of every thread of every concurrent warp, the bandwidth needed would be $15 \times 4 \times 2 \times 32 \times 32 = 122{,}880$ bytes per cycle. For a workload whose threads all access independent data, the GPU would only be able to sustain one L2 access every 120 instructions, on average.

Global memory is the most plentiful but also the slowest type of memory. GPUs coalesce the global memory accesses of the threads in a warp into as few transactions as possible. If all threads access the same cache line then there will only be one memory transaction, 32 times fewer than if all threads accessed different cache lines. Shared memory and L1 cache are on-chip fast memories but have limited capacity. Their combined size is 64KB per SM and they are two orders of magnitude faster than the global memory. We could use shared memory to achieve coalescing: All threads in an SM would load contiguous strings into the shared memory from global memory. Given 64 warps (2048 threads) in flight in each SM and up to 48KB of shared memory, all threads could read in a coalesced fashion strings of 24 characters. However, for longer strings the parallelism would be reduced, resulting in worse performance.



(a) Sends row-ids satisfying the two integer filters to the GPU; the GPU then searches only the listed strings for matches, and returns the corresponding row-ids.

(b) Executes the integer filters on the CPU and in parallel the string matching condition on all rows in the GPU; the results are intersected to compute the final result.

(c) This strategy first applies the string filter on the GPU and sends the matching row-ids to the CPU which can use this information to process less data.

Figure 5.1: Strategies for CPU-GPU interaction.

5.3 String matching Framework

There are multiple strategies to execute a complete query in a coprocessing environment with CPUs and GPUs. Suppose there are some integer and some string filters in the query, as for example in Q16 of TPC-H. The output of the algorithm is expected to be the row-identifiers of rows matching all of the specified conditions. Figure 5.1 shows three execution strategies and the data exchanged between the CPU and the GPU. The query optimizer is responsible for choosing a suitable strategy given selectivity and cost estimates for the various conditions.

Queries that require the matching string itself in the SELECT clause are handled outside the matching process.

5.3.1 Addressing Thread Divergence

Size Grouping. Thread divergence can occur when some threads have reached the end of their string, while others in the same warp have not. The threads that have finished sit

idle until all of the remaining threads in the warp complete their task, which may involve scanning a long string segment. For this reason we make sure that all threads in a warp operate on strings of similar length. We preprocess the string database, sorting the strings by length. If necessary, we pad some strings with a few nulls so that each string can be grouped with other strings of the same length into a warp.

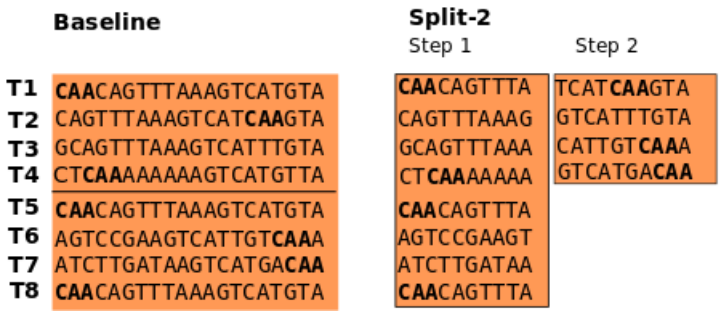


Figure 5.2: Execution of the baseline method and Split-2 method for search pattern 'CAA'.

Splitting. Even with equal-length strings, thread divergence can occur when some threads have found a match and sit idle, while others in the same warp have not. To address this kind of divergence, we consider the option of *splitting* strings. For example, we could split each string into two equally-sized pieces and initially perform string matching on the first half. Those strings for which a match has been found have their IDs added to the answer set, while strings for which a match has not been found have their IDs added to a pending set. In a second pass, we process the second halves of strings in the pending set in parallel, without ever looking at the second halves of strings that matched in the first pass. In this way, threads that match early do not have to sit idle for the entire string length. Splitting is illustrated in Figure 5.2.

We implement the multiple steps of split optimization in a single GPU function. In all steps but the last, the threads store in the shared memory the record-ids of the strings that have not matched the pattern. Different threads might process different parts of a string in distinct steps. We divide the shared memory in different buffers, one for each warp. For selective conditions the expected performance gain is higher because more strings are filtered in each step and there is lower shared memory writing cost.

Splitting may introduce a small overhead at split boundaries. Searching each piece of

the string (except the last) might need to process up to $m-1$ characters from the following piece, where m is the length of the pattern. When the following piece is later searched, some of those $m-1$ characters may be processed again. For KMP (but not BM) we avoid this boundary overhead by recording the search state for the last character of the previous string piece.

5.3.2 Addressing Cache Pressure

The observations of Section 5.2 suggest that allowing all threads to do independent work (the method of choice for CPUs) will not be sustainable on GPUs. Efficient GPU string matching will need some form of *locality* so that data from each cache line is useful for multiple threads.

Register Usage To reduce the number of memory loads from cache, we try to read as large a unit of string data as possible into GPU registers. Using an extended load instruction, it is possible to read up to 16 bytes of data at once into two registers. Depending on the memory access pattern of the string matching algorithm, this approach can reduce the traffic from the L2 cache, although it does not reduce the cache footprint needed by the collection of threads.

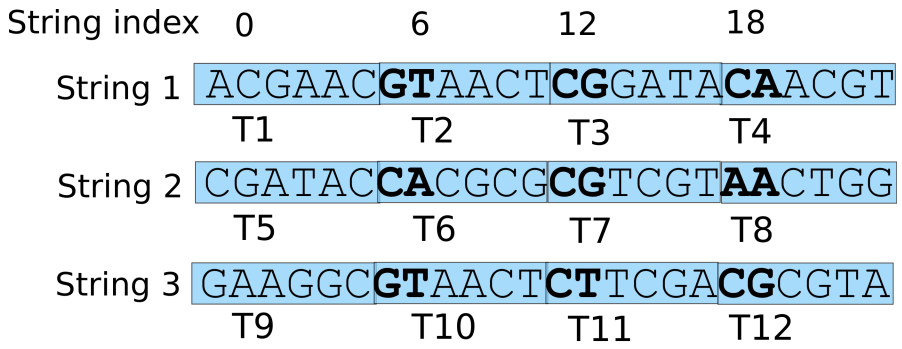


Figure 5.3: Execution of Seg 6-4 parallelism method for a pattern of three characters.

Segmentation. One way to enhance locality is to allocate multiple threads from a warp to different segments of the same string in such a way that the interval between consecutive threads on a string is less than a cache line. In that way, multiple segments of the string can be examined concurrently, and the cache misses needed to read the string are amortized over

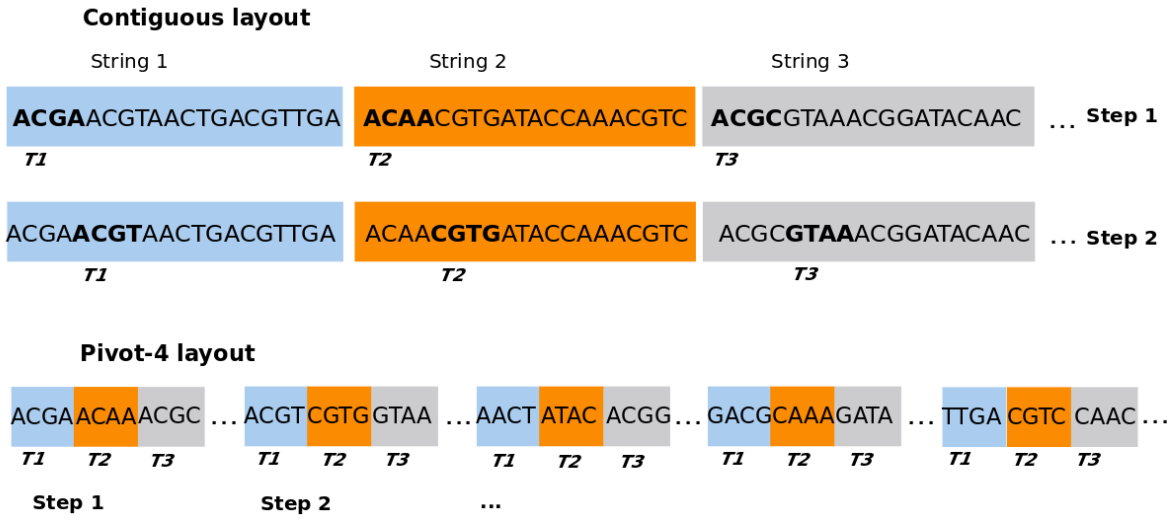


Figure 5.4: Contiguous and pivoted layout for 20-character strings and pivoted-piece of 4-characters.

multiple threads. For example, if four threads span a cache line, then the cache footprint would be reduced by a factor of 4. Threads processing the same string can coordinate and finish searching when one of them finds a match. For the segmented method, we will use the term "Seg-k-t" to denote a method using t threads per input string with a gap between threads of k bytes. Figure 5.3 demonstrates the execution of the Seg-6-4 method for string of 24 characters. Typically we choose larger gaps, but we show a small gap for the sake of simplicity.

While locality is improved in such a scheme, there is a boundary overhead for each segment. To detect patterns that span segment boundaries for a pattern of m characters all threads but the last will have to process the following $m-1$ characters. Because segments are processed in parallel, we miss opportunities to skip over the initial characters of each segment.

Pivoting. So far, we have implicitly assumed that strings are stored contiguously. However, in database applications preprocessing the data (e.g., by rearranging the string layout) is a viable option to maximize workload performance. We propose pivoting[1] the

[1]To the best of our knowledge string pivoting has been suggested only for fixed 1-byte or 4-byte units

	Independent	Segmented	Split	Pivot	Split+Pivot	Self-Pivot (Long Strings)
+Thread efficiency	Low	High	High	Low	High	High
+L2 efficiency	Low	High	Low	High	High	High
-Boundary overhead	No	Yes	KMP:No/ BM:Yes	No	No	Yes

Table 5.1: Advantages and drawbacks of each string matching optimization.

strings in the GPU global memory to eliminate cache thrashing and to facilitate coalesced loads. We divide the string into equally sized pieces and store each piece as if it was a different column of a column store. For example, suppose we choose a piece size of 4 bytes for strings 128 bytes long. The first "column" would contain the initial 4 characters of each string. The second "column" would contain the second 4 characters, and so on up to "column" 32. Figure 5.4 shows the contiguous and the pivoted layout for 20-character strings and 4-character pivoted pieces. In case strings have different lengths we need to pad them to have the same number of "columns". Threads would be assigned one per string, with threads in a warp processing consecutive strings. When they start, warps would read the initial 4 bytes from a contiguous group of strings, which is an ideal memory access pattern because (a) it is coalesced, and (b) it supports good spatial locality and a reduced cache footprint. As string matching progresses, threads may progress at different rates depending on the matching algorithm and pattern. We evaluate the effect on performance for different pivot widths.

In the case of pivoted strings, algorithms for which threads progress at different rates are vulnerable to a phenomenon we call *memory divergence*. If threads in a warp are accessing mostly different offsets within the pivoted layout, we will have lost most of the performance benefits of pivoting. As a result, there is an implicit advantage to methods that proceed in lockstep (or close to it) through the strings, even if it means shorter advances on each step. In fact, we show how one can slightly change the implementation of KMP to guarantee lockstep processing, perhaps at the expense of increased thread divergence.

[Scarpazza *et al.*, 2007; Iorio and Lunteren, 2008; Zu *et al.*, 2012], without studying the impact on cache behavior.

Memory divergence has not previously been studied for string matching applications.

Self-Pivoting: For long strings we suggest the following solution combining the best features of pivoting and segmentation: Segment the string into equal-sized pieces and store the segments of each string in a pivoted fashion. For example, suppose we segment a 4KB string into 32 segments of 128 bytes each. With a pivot width of 4 bytes, the first cache line would contain the first 4 bytes from each segment; the second cache line would contain bytes 5 though 8 for each segment, and so on. Threads in a warp process different segments of a single string in parallel, and achieve locality because the data needed by those threads at any point in time is concentrated in just a few cache lines (one if the threads manage to proceed in lock-step). Threads in a group coordinate and terminate when any of them locates the pattern.

Self-pivoting requires minimal padding, because each string can be divided into segments that differ in size by at most one symbol. Unlike pivoting across strings, self-pivoting retains good coalesced memory access behavior even for filtered string access (Strategy A in Figure 5.1).

Method	**Techniques**
Segmentation	Segmentation, Size grouping, Register Usage
Splitting	Splitting, Size grouping, Register Usage
Pivoting	Pivoting, Size grouping, Register Usage
Self-Pivoting	Self-Pivoting, Size grouping, Register Usage

Table 5.2: Set of techniques used in our string matching methods.

5.3.3 Addressing Memory Divergence

GPUs are sensitive to changes of source code even if the time complexity remains the same. However, alternative implementations of KMP have not been studied. We show below two alternative ways to code the KMP algorithm [Eppstein, 1996], assuming that the algorithm returns "true" after finding the first match. In both of these code fragments, `s` is the string being searched, `p` is the pattern, `nxt` is the partial match table, `m` is the length of the pattern, and `n` is the length of the string. For simplicity we omit optimizations such as reading many

characters at a time from memory into registers.

```
KMP_basic(char *s, char *p, nt *nxt, int m, int n){
int i=0;
int j=0;
while (i<n) {
    if (p[j]==s[i]) { //Divergent branch
       j++;
       i++;
       if (j==m)
           return true;
    } else {
       if (j != 0)   //Divergent branch
          j = nxt[j];
       else
          i++;
    }
}

KMP_step(char *s, char *p, int *nxt, int m, int n){
int j=0;
for (int i=0;i<n;i++) {
    while (j>=0 && p[j]!=s[i]) //Divergent loop
          j=nxt[j];
    j++;
    if (j==m)
          return true;
}
```

In the `KMP_basic` algorithm, if the while loop is executed in parallel by many threads, each thread either advances through the string or advances the offset within the pattern. Strings that advance the offset in the pattern will "fall behind" the other threads that

advance though the string. Unlike `KMP_basic`, a thread-parallel execution of the `for` loop in `KMP_step` waits for those threads that need to advance far enough through the pattern to advance to the next character. We also propose a hybrid method `KMP_hybrid` that has an outer loop like `KMP_step`, to ensure memory alignment, with an inner loop like `KMP_basic` to reduce thread divergence. The granularity of the outer loop should correspond to the pivot width in a pivoted layout; the following code segment corresponds to a pivot width of 4. In our string matching framework, we assume a little-endian representation similar to the NVIDIA GPU architecture.

```
KMP_hybrid(char *s, char *p, int *nxt, int m, int n){
int j=0;
for(int i=0;i<n;i+=4){
    unsigned t=*((unsigned *)s+(i>>2));
    for(int k=0;k<4;){
         if (p[j]==((char)t)) {//Divergent branch
              t>>=8; k++; j++;
              if (j==m)
                 return true;
         } else {
            if (j != 0)    //Divergent branch
                j = nxt[j];
            else {
                t>>=8; k++;
}}}}
```

In KMP_Basic and KMP_Hybrid the input is advanced in two cases: 1) If the comparison of the pattern to the input succeeds or 2) If j equals zero. If the input is not advanced then the jump table is accessed. Based on that observation we can remove some of the branches in the code to increase the string matching performance. We show below the version of KMP_Hybrid, with the eliminated branches:

```
KMP_hybrid(char *s, char *p, int *nxt, int m, int n){
int j=0;
for(int i=0;i<n && j<m;i+=4){
    unsigned t=*((unsigned *)s+(i>>2));
    for(int k=0;k<4 && j<m;){
          const int cmp=p[j]==((char)t);
          const int cmp2=cmp || j==0;
          if(cmp2)  {
                  t>>=8;
          } else {
                  j = nxt[j];
          }
          j+=cmp;
          k+=cmp2;
    }
    return j==m;
}
```

Removing the remaining branches resulted in inferior performance.

Figure 5.5 shows the execution of the Pivot-4 KMP-Hybrid method. Different threads are processing different strings. Threads start scanning the first character of the first pivoted piece. In the internal loop of KMP-Hybrid the input is advanced either if the comparison of the current pattern character to the input succeeds or if it fails on the first character of the pattern (j==0). T1, T3 match their input in first iteration to the pattern. In the second iteration the comparison fails for both T1 and T3 so they have to consult the *nxt* table to shift the pattern. In the third iteration T1 and T3 will compare the second input character again after shifting the pattern. Threads synchronize again in the seventh iteration after all threads process the first piece and scan the first character of the second pivoted piece.

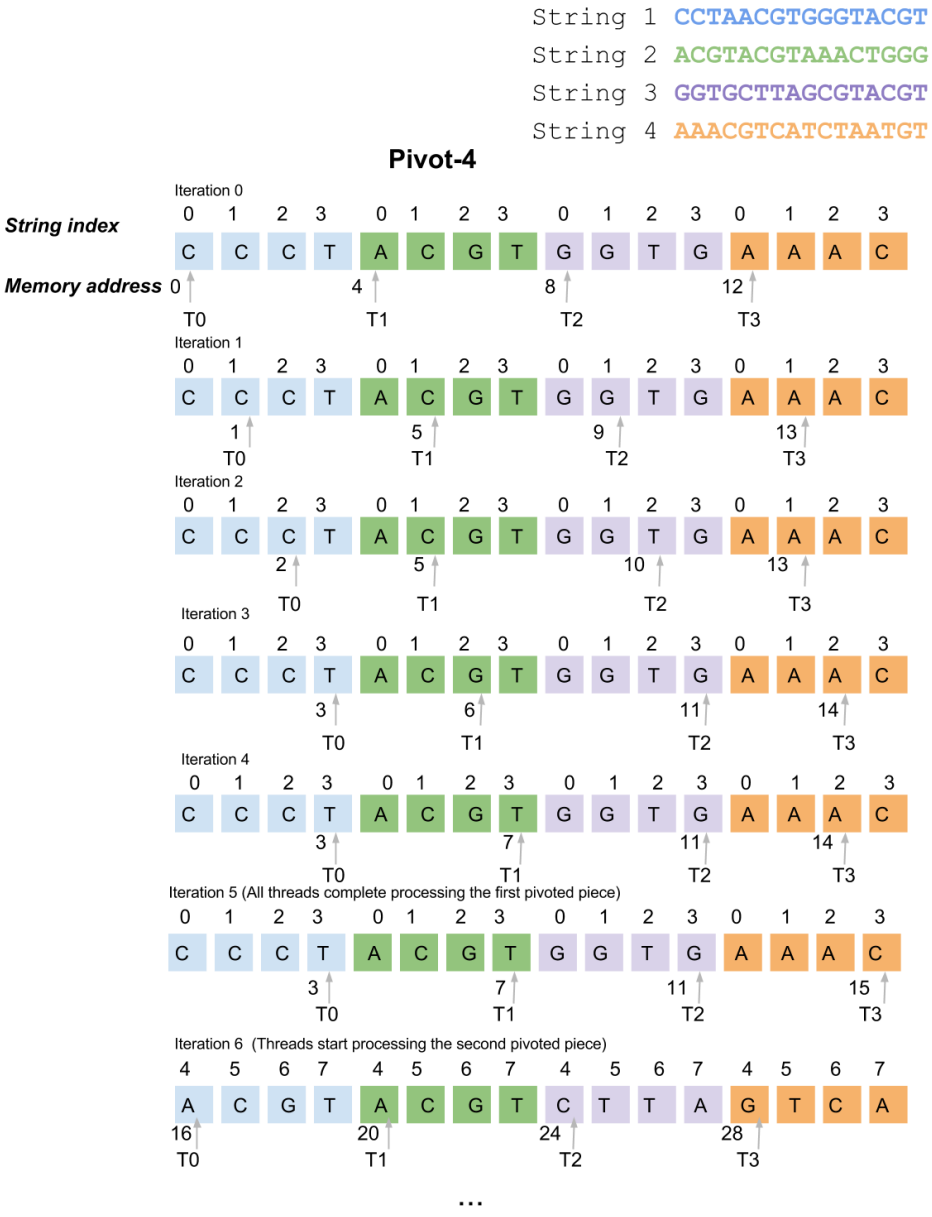


Figure 5.5: Execution of Pivot-4 method for 'ATG' pattern using the KMP-Hybrid string matching method and 4 GPU threads.

5.3.4 Combining Different Optimizations

In Table 5.1 we summarize the different optimization techniques we evaluate. Our suggested techniques can be combined to implement more complex methods. Table 5.2 summarizes

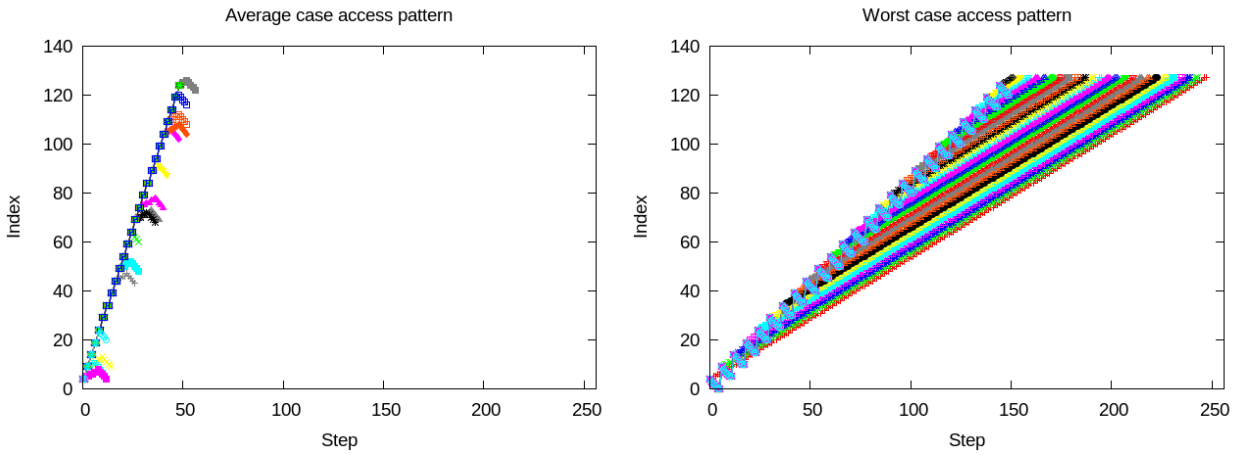


Figure 5.6: Memory access pattern for BM on an average case dataset and an adversarially generated input maximizing memory divergence. There are 32 curves in each subfigure showing the index accessed by each thread of a warp. The input strings are 128 characters/bytes long.

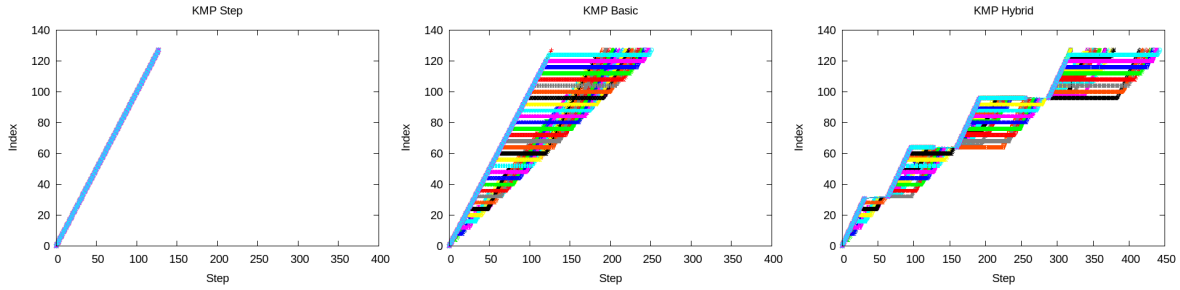


Figure 5.7: Memory access pattern for a worst case input of KMP.

the techniques used by the matching methods evaluated in our experiments. We always implement size grouping and use registers to reduce the global memory traffic. When using pivoting the size of the used registers is limited to the pivoting width: For example in Pivot-4 method we load in 4-byte registers four characters from the input string. We cannot use registers larger than the pivot width because this would require multiple loads from non-contiguous memory locations.

Splitting can be combined with segmentation: In each step, each piece of the string can be processed by multiple threads as in the segmented method. Splitting can also be combined with pivoting, although it does not lead to better performance compared to plain

pivoting. Splitting incurs the cost of writing the intermediate results in the shared memory but this additional cost is justified by the reduced string processing cost. However, the benefits of splitting are not as significant for pivoted layouts since the global memory cost of string processing is reduced. For the conventional layout there is a cache miss in each thread memory access while for the pivoted layout multiple thread accesses can be combined in a few global memory transactions. Also, the benefits of pivoting will be reduced in the subsequent steps because the strings being processed might not be contiguous. Self-pivoting is designed for longer strings by combining the best features of the segmented and pivoted methods.

5.3.5 Algorithm Analysis

Here, we analyze the worst-case behavior of KMP and BM string matching algorithms. We then provide a cache footprint analysis of the baseline and optimized string memory layouts.

5.3.5.1 Worst-Case Algorithmic Analysis

We are interested in constructing worst case inputs for the BM and KMP algorithms. We must avoid performance "surprises" on these inputs. The worst case for a single instance of the KMP algorithm is searching for a pattern with m repetitions of a single character in an input string which is a repetition of the following segment: the first $m-1$ characters match but the mth does not. For the warp level adversarial dataset the input strings are constructed in such way that the first thread in the warp matches the first 4 characters and fails, the second thread fails after the first 8 character comparisons and so on. This will result in memory divergence for Pivot-KMP_basic: Some of threads will fall behind, accessing different column segments. KMP_step will have increased thread divergence because threads will each execute a different number of iterations of the internal while loop.

For BM we construct a similar adversarial set of strings as in KMP with the only difference that the matching fails in the leftmost character rather than the rightmost because it compares the characters from right to left. In Figure 5.6 we show the memory access pattern of BM for two sets of input strings. There are 32 color bands in each of the subfigures corresponding to the 32 different threads in a warp. Different threads process 32 different

input strings, so the different curves show the string index accessed by each of the threads in the warp. The first set of strings is randomly generated. In the second dataset to evaluate the worst case memory access pattern of BM the strings are adversarially generated so that the divergence of BM memory access pattern is maximized. Figure 5.7 shows the memory access pattern of the KMP implementations. We note that different steps have different cost for the three KMP versions and more steps do not imply slower performance. With a pivoted layout, when concurrent threads access locations separated by more than the pivot width, additional cache misses will result.

5.3.5.2 L2 Cache analysis

Table 5.3 shows the L2 cache footprint for different parallelization methods.

Method	Cache footprint (MB)
Independent	**3.75**
Seg-64	**1.875**
Seg-32	0.938
Seg-16	0.469
Split-k	**3.75**
Pivot-2 (Ideal)	0.117
Pivot-4 (Ideal)	0.117
Pivot-8 (Ideal)	0.234
Pivot-16 (Ideal)	0.469
Pivot-32 (Ideal)	0.938
Pivot-k (Divergent)	**3.75**

Table 5.3: L2 cache footprint of different matching methods.

Segmentation and pivoting can reduce the cache footprint below the L2 capacity. Splitting does not impact the cache footprint, but could still be effective at reducing other forms of latency. Below $k = 4$, the ideal footprint of Pivot-k does not change because a single cache line can accommodate 32 threads' worth of 4-byte data. The numbers for pivoting labeled "ideal" reflect the footprint at the start of matching, and if matching proceeds at

precisely the same rate in each thread of a warp. The row labeled "divergent" corresponds to a situation in which threads progress at the sufficiently different pace that all threads are touching different cache lines. In practice, partial thread divergence may lead to intermediate cache footprint sizes. We will examine this issue in more depth in Sections 5.3.3 and 5.3.5.1. The cache footprint of self-pivoting is the same as regular pivoting.

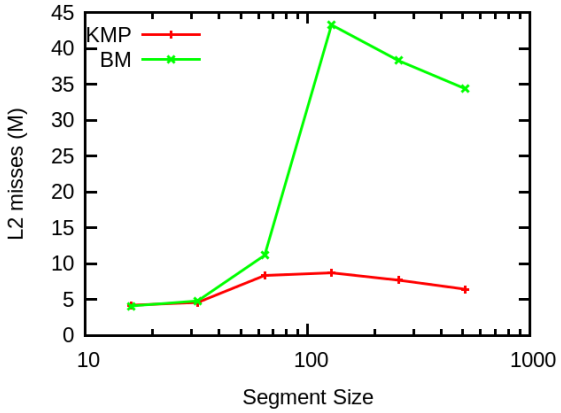


Figure 5.8: KMP and BM Seg-k-t L2 misses for varying segment size on a dataset of 512K strings with t=4.

To validate the analysis in Table 5.3, Figure 5.8 shows the number of L2 misses for varying segment size as measured on Nvidia K40. The string length is 1024 bytes, the pattern size is 4 bytes and the selectivity 0.9. For both algorithms L2 misses increase significantly at a segment size of 64, as predicted in Table 5.3.

5.4 Experimental Evaluation

5.4.1 Experimental Setup

GPU performance was measured on two GPUs: an Nvidia Kepler K40, and an Nvidia Tesla C2070. ECC was turned on. L1 caching is off by default on the K40. We turned off L1 caching on the C2070 because we observed that performance improved for bandwidth-bound algorithms as a result of the reduced bandwidth needed from the L2 cache.[2] Table

[2]When the L1 cache is turned on, 128 byte cache lines are sent from the L2. When L1 caching is off, 32 bytes are accessed at a time from the L2.

GPU	Tesla C2070	Tesla K40
Memory size	6GB	12GB
Frequency(MHz)	1150	745
Bandwidth (GB/s)	144	288
L2 (KB)	768	1536
L2 band. (B/cycle)	384	1024
Cores	448	2880
Shared mem. (B/cycle/bank)	2	4
SMs	14	15
Cost ($)	1250	3100

Table 5.4: GPUs used in our experiments.

5.4 contains the specifications of each GPU. The default machine is the K40 but we show some results for both machines to prove that our techniques and analysis are not specific to one machine, and to highlight situations where the machine used does matter.

CUDA C code was compiled using nvcc of CUDA toolkit 5.5 using the full optimization level. We also used loop unrolling in the string matching functions to reduce the number of executed instructions. For a more fair comparison against CPUs we use the base clock of the K40 GPU (745 MHz), although we noticed an improvement of 17% in performance when using the maximum clock frequency (875 MHz).[3] We used a dual socket Intel E5 2620 CPU to compare against the GPU performance. Each socket has 6 cores (12 threads) and a power rating of 85W. We used all 24 threads of the dual socket machine.

Strings are either random (for average-case performance measurements), constructed as described in Section 5.3.5.1 (for worst-case measurements) or generated from real-world datasets. String characters in our experiments are one byte so we use the terms character and byte interchangeably. To achieve a given selectivity, the pattern is inserted into random locations of a suitable number of randomly chosen strings. For some experiments we also vary the number of times that a pattern appears in a string. Table 5.5 summa-

[3]The higher frequency is a "boost" frequency that can only be used when there is power and temperature headroom.

	A1	A2	R1	R2	TPC-H
String size	1024	16–16K	500	4.51M	63,1024
Occurrences	0–32	0	N/A	N/A	N/A
Pattern size	1–32	15	10	20/25	N/A
Rows	512K	32K	3.52M	64	512K
Alphabet	26	4	128	4	N/A

Table 5.5: Workload parameters.

rizes the workload parameters for our datasets. Workload A1 reflects a textual search task. A2 reflects a search of an artificial DNA sequence. R1 is a set of wikipedia abstracts as downloaded from DBpedia [DBPedia, 2014]. For R1 we chose a set of 10-character patterns randomly selected from the input. All patterns correspond to low selectivity (<0.01). R2 searchehttps://www.facebook.com/s the genome sequence of Yersinia pestis [Trust Sanger Institute, 2001] which we have replicated to achieve full parallelism to simulate a realistic workload searching the genomes of many bacteria (one string per organism) for the given pattern. For the TPC-H workload we use the data generator provided by the TPC-H benchmark. We ran the sub-query of Q16, which we will be referring to as Q16_1:

```
select s_suppkey
from supplier
where s_comment like '%Customer%Complaints%'
```

We use scale factor of 53 during data generation for the Supplier column to produce 512K rows. A fixed number of rows is randomly selected from the TPC-H data generator to contain strings matching %Customer%Complaints%. We show the results for both the original s_comment column size and for a wider column size of 1024 characters.

Pattern preprocessing is done only once for a pattern so it has no performance impact. We store the jump-table in the shared memory in addition to the search pattern. We compare our GPU implementation to the performance of PFAC multi-pattern matching running on the GPU and a multi-threaded CPU implementation using three different CPU implementations for A1.

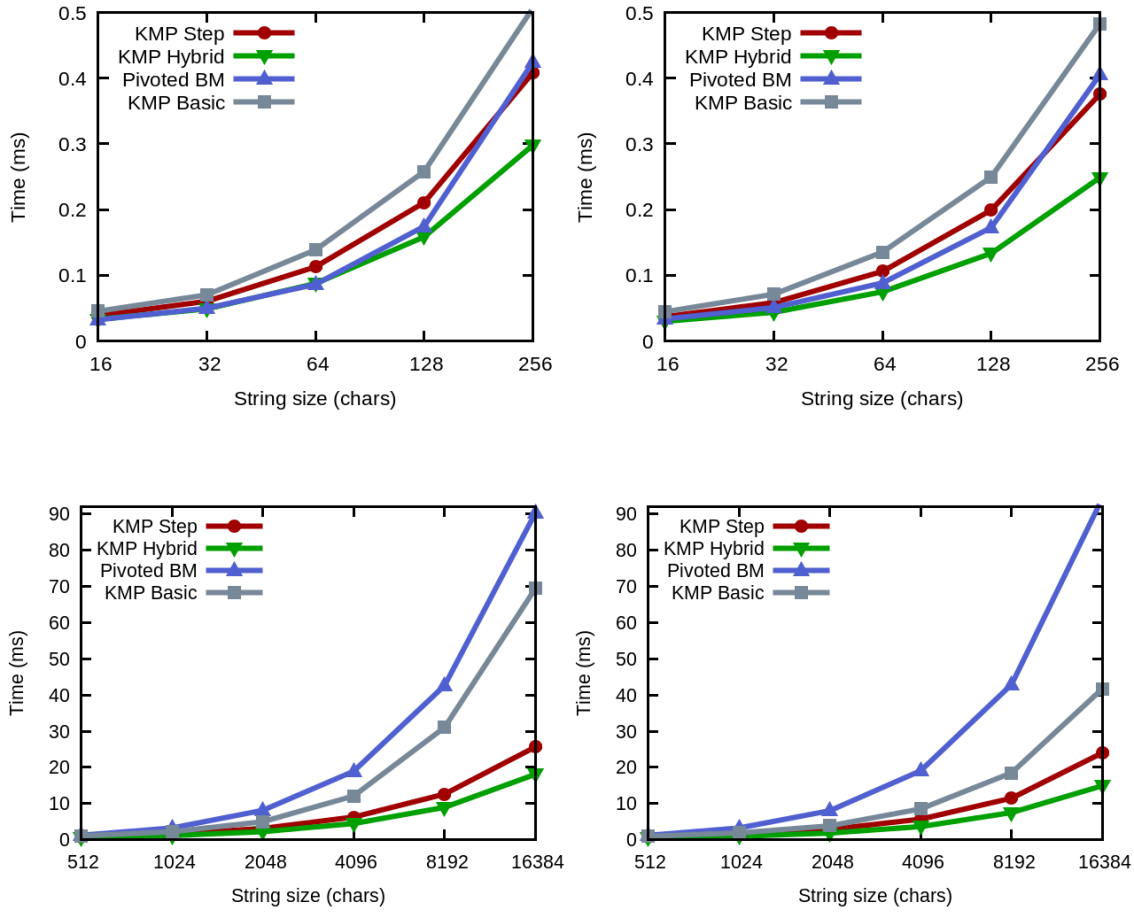


Figure 5.9: Performance as a function of string length for Pivot-4 (left) and Pivot-8 (right) layouts. The top row shows the results for shorter strings and the bottom row for longer strings.

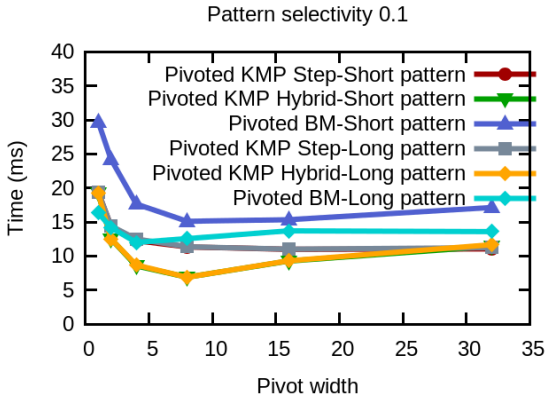


Figure 5.10: Time performance on A1 for varying pivoted width.

5.4.2 Comparing Algorithm Efficiency

Figure 5.9 shows the performance of all three versions of KMP and of BM on workload A2 as a function of the string length, using Pivot-4 and Pivot-8 layouts. The biggest impact of our methods is for longer strings where there is worse memory locality and higher memory divergence.

The top row shows the performance for strings up to 256 characters and the bottom for strings up to 16384 characters. For strings less than 64 characters BM has similar performance to KMP-Hybrid. For longer strings the performance of BM and KMP-Basic deteriorates due to memory divergence.

Pivot-8 is clearly superior to Pivot-4 for all methods. Figure 5.10 shows the performance of pivoted KMP and pivoted BM algorithms for varying pivot width on workload A1. We observe that KMP has stable performance regardless of the pivoted width and pattern length and in general it has superior performance to BM.

We repeated the experiment for workload R1, corresponding to a textual search on a real-world dataset, where the character distribution is non-uniform. In Figure 5.11 we show the corresponding results. We observe that the performance difference between KMP methods and BM is higher than the difference for A1. This happens because when some characters occur more frequently than others, as it is the case for real text, this resembles inputs of smaller alphabet size for which BM has less competitive performance.

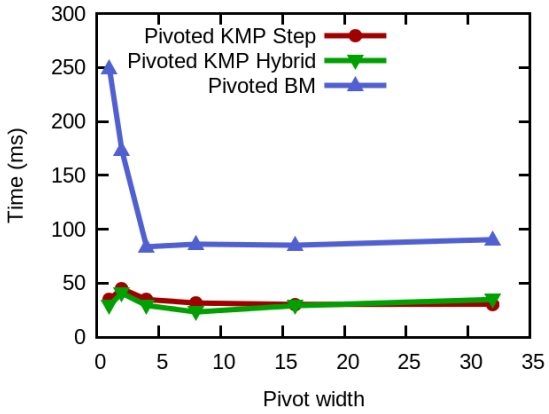


Figure 5.11: Time performance on R1 for varying pivoted width.

Taken together, the results of Figures 5.10 and 5.11 suggest that 8 is the best choice for pivoting width. We have repeated the same experiment for shorter strings (64 characters) suggesting the same pivoted width. The results also show that, at least for pivoted layouts, techniques that minimize memory divergence (`KMP_step` and `KMP_hybrid`) are superior. `KMP_hybrid` is better than `KMP_step` in Figure 5.9 because thread divergence is reduced. Threads do not have to proceed in lockstep at character granularity, just at 8-character boundaries. From now on, we will use just `KMP_hybrid` for single-pattern matching on pivoted layouts.

We also implemented pivoting itself on the GPU to transform strings from contiguous to pivoted representations. For a pivot width of 8, pivoting ran at 60GB/s on the K40, which is typically faster than string search.

5.4.3 Effect of Thread Divergence

We show the performance of split optimization on the K40 and C2070 for workload A1 in Figure 5.12.

Figure 5.12 compares the performance of independent string search to split-k for different k and for a varying number of repetitions of the pattern in the strings. Each subfigure in the top row corresponds to BM on the C2070 with a different operator selectivity: the first for selectivity 0.6, the second for 0.75, and the last for selectivity 0.9. For less selective conditions there is lower overhead of writing in shared memory in the intermediate steps

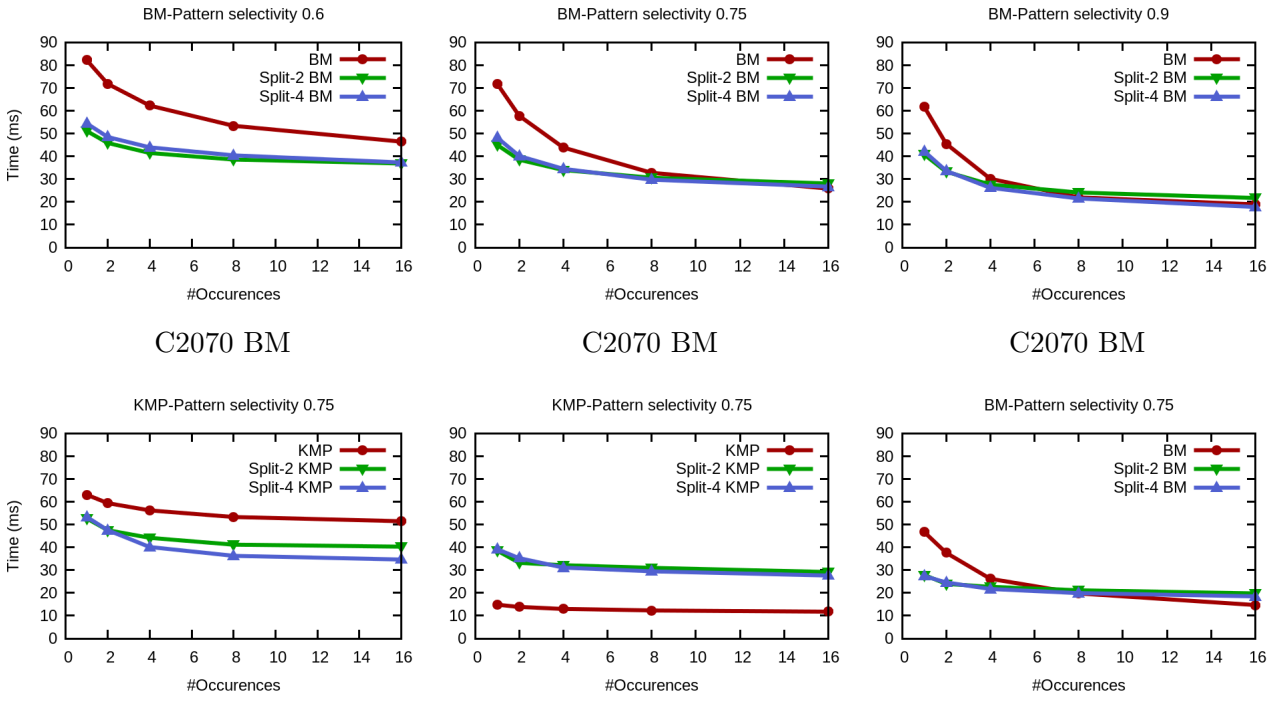


Figure 5.12: Performance of Split-k Optimization on BM for varying number of string occurrences in the input. The first subfigure is for selectivity 0.6, the second for 0.75, and the last for selectivity 0.9.

and more potential for bypassing computation, on the other hand the cost of string search is lower because more threads finish searching early. For selectivity 0.9 the speed-up is up to 40%, while for selectivity 0.6 the speed-up is 65%.

The right chart in the bottom row of Figure 5.12 shows corresponding results for BM at selectivity 0.75 on the K40 GPU, which are qualitatively similar to the C2070. The remaining two charts show the KMP performance on the two GPUs under various Split-k configurations. Surprisingly, the GPUs differ significantly in the relative performance. While split-k optimizations are helpful on the C2070, the same is not true on the K40. The reasons for this difference are subtle, and illustrate the complexities of optimizing GPU performance.

Consider the performance parameters of the two devices in Table 5.4. If an algorithm is memory bound, it can expect at best a 2X (288/144) improvement moving from the C2070 to the K40. Similarly, if the algorithm is bound by the L2 bandwidth, an improvement of 1.7X $((1024 * 745)/(384 * 1150))$ is possible. If the algorithm is bound by accesses to shared memory, then a factor of 2 improvement is possible. On the other hand, an algorithm that is not memory bound has the potential for a 4.2X $((2880 * 745)/(448 * 1150))$ speedup due to the much larger number of cores on the K40. The K40 can issue two instructions per warp so depending on how well the pipeline slots can be filled, the speedup potential is up to 8.4X. The KMP algorithms with the split optimization are shared-memory bound, because the intermediates used by this optimization need to be written to and read from shared memory. On the other hand, the KMP algorithm without the split optimization has been engineered to be cache resident and to avoid memory performance pitfalls. As a result it can achieve a speedup closer to the 4.2X potential speedup. BM, being memory bound due to the high cache miss rate, cannot achieve the same speedup.

5.4.4 Effect of Alphabet Size

Figure 5.13 shows the performance of the pivoted and unpivoted methods for varying alphabet size. We observe that BM methods, both pivoted and unpivoted depend on the alphabet size: For larger alphabets the performance improves, because the skipping distance increases. The performance improvement is more observable for pivoted-BM because

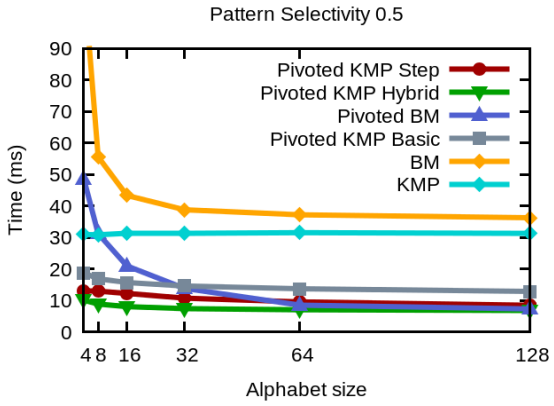


Figure 5.13: Time performance of pivoted and unpivoted methods for varying alphabet size and an 8-character pattern.

for large alphabets the memory divergence decreases and threads in a warp may skip entire pivoted pieces. KMP also slightly improves: The memory access cost remains the same because memory accesses are better coordinated (KMP-Hybrid and KMP-Step) but there are fewer partial matches so there are fewer jumps over the pattern table. The overall performance variation is less observable because the pattern and the jump table are stored in the shared memory which is faster. Finally, we observe that pivoted KMP-Hybrid is the faster matching method even for larger alphabets.

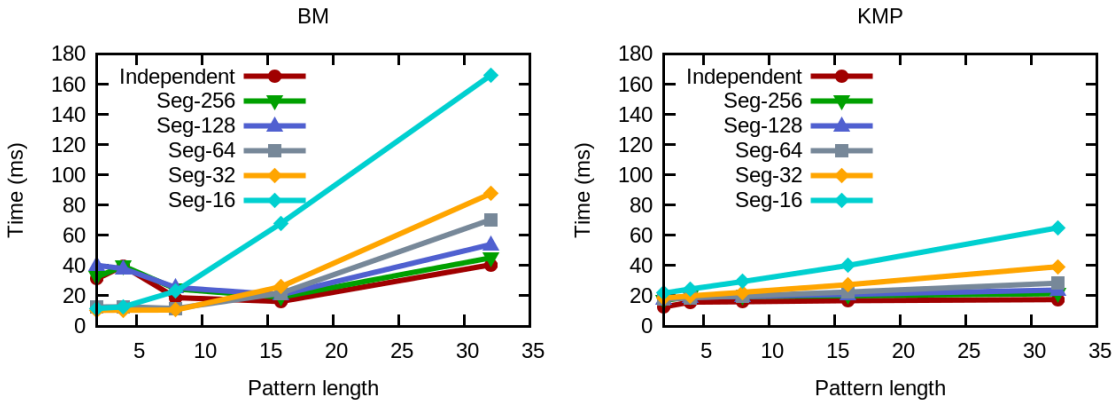


Figure 5.14: Performance for increasing pattern length for Independent and Seg-k-t (8) implementations.

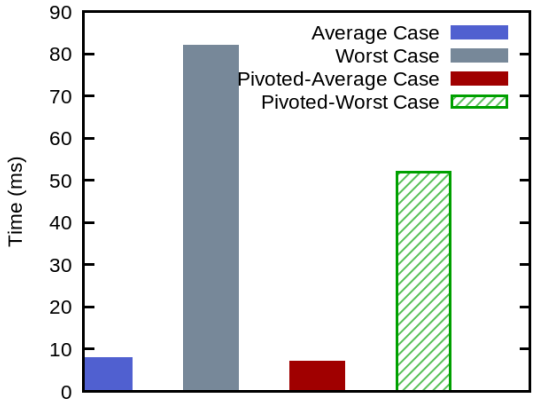


Figure 5.15: Performance of BM for average and adversarial input.

5.4.5 Segmentation

Figure 5.14 shows the performance of Seg-k-t for $t = 16$ for increasing pattern length on workload A1. For BM and patterns shorter than 16 characters the best method is Seg-32 because of the reduced cache misses, but for patterns longer than 16 characters the most efficient method is having threads matching independent strings. For BM the performance is initially increasing because BM can skip larger parts of the input but it starts decreasing because of the increased effect of boundary overhead. For KMP the optimal parallelism method is independent threads because we already reduce the memory cost by prefetching multiple characters into the registers. We also observe that KMP performance seems to depend less on the pattern length. When $t > 8$, we observed that the choice of t did not significantly affect the performance of string matching.

5.4.6 Worse-Case Performance

Figure 5.15 shows the performance of row-wise and Pivoted-8 BM for an average and worst-case input set of strings on workload A1 for a 32-character pattern.

The performance difference is about an order of magnitude. We also observe pivoting helps the performance of BM. This is because subset of the threads particularly at the beginning of the search process "jump" to the same pivoted column. However depending on the pattern length and the pivoted width the performance improvement might be less

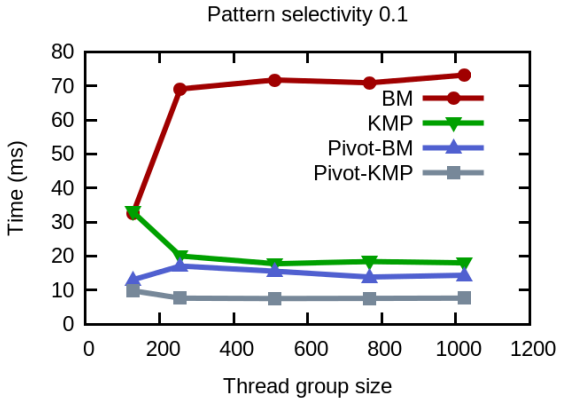


Figure 5.16: Performance of KMP and BM for varying group size.

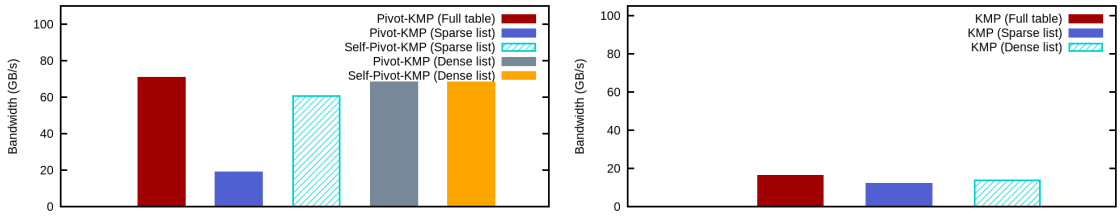


Figure 5.17: Bandwidth of string matching for sparse and dense record lists for pivoted (left) and unpivoted (right) KMP methods.

significant, as we noted in Figure 5.10. The performance difference for all versions KMP between the average and worst case is less than 2x making it a more robust choice.

5.4.7 Thread Group Size Tuning

Figure 5.16 shows the performance of (Pivot-8 and unpivoted) KMP and BM for varying group size on A1 for a 4-character pattern. `KMP_basic` is used for unpivoted and `KMP_hybrid` for Pivot-8. The optimal thread group size for unpivoted BM is 128 because it has larger cache footprint when there are more concurrent threads. For the Seg-k-t BM method the L2 footprint is reduced so the optimal group size is larger; we use the optimal thread group size for each method.

Co-processing performance Figure 5.17 shows the performance of Pivot-KMP, Self-Pivot-KMP and unpivoted KMP for two different access patterns. String matching is ap-

plied on a sparse and a dense subset of records rather than the full table. The sparse record list corresponds to 1% selectivity while the dense for 50% selectivity. These patterns correspond to queries where one or multiple predicates have already been applied. The list of the record identifiers are communicated from the CPU to GPU. In the pivoted methods the dense rid-list performance is hardly affected because we're still doing just a small number of memory transactions (2 rather than 1, since the selectivity is 0.5). As expected, the unpivoted method's performance is not significantly affected. The performance for the unpivoted method is still lower than the performance of the pivoted methods on sparse record lists.

For the sparse case threads in a warp will access different pivoted pieces so we are doing 32 transactions rather than 1 and pivoting performance degrades significantly. However, self-pivoting maintains most of the performance benefits even when a sparse subset of the strings is processed from a query.

Table 5.6 shows the performance of the three CPU-GPU coprocessing stategies for the following query:

```
Select count(*)
From Orders
Where o_comment not like '%special%packages%' AND o_orderstatus='F' AND
o_totalprice>50000
```

The dataset is generated by the TPC-H generator. The string column o_comment is stored in the GPU. The other two columns are stored in the CPU RAM. The fastest stategy

Strategy A	Strategy B	Strategy C
0.85 ms	1.02 ms	1.05 ms

Table 5.6: Time performance of the three alternative CPU-GPU interaction strategies.

is Strategy A because the predicates applied on the CPU are "cheap" to evaluate. For more expensive predicates we expect Strategy B to be more competitive.

5.4.8 Comparison with CPUs

Investing in the use of a GPU depends on more factors than just the raw query performance. We compare GPUs and CPUs holistically in terms of raw performance, performance per

$ and energy consumption for the subquery Q16_1. In addition to comparing CPU and a GPU processors, we also show the estimated performance of a combined system that uses both CPUs and a K40 GPU in the following way: It initially executes different instances of the same query on the CPUs and the GPU. We run a query per CPU hardware thread and another query instance on the CPU delegating its work to the GPU. The CPU threads use for each query the fastest CPU matching library among the algorithms that have linear time complexity. Whenever a query completes execution we start a new query to ensure that all processors are kept busy. This process is executed for five seconds and in the end we compute the average bandwidth and energy consumption per query. Typically the throughput of the combined system is the sum of the measured query throughput of all the processors when executing independently. There is only a small (5-10%) overhead when the CPU thread delegating its workload to the GPU has to copy back the query results.

We use all 24 CPU hardware threads and set the frequency policy to maximum.[4] We evaluate two different popular CPU libraries, RE2, and Boost and we show the performance of the fastest of the two. For RE2 each thread is independent operating on a separate re2 object, so the pattern is compiled once for each thread. For the Boost library we use the object-based interface for each method and the pattern is again compiled once for each CPU thread. We also implement a CPU matching method based on the CMPISTRI SSE instruction for patterns that fit in a SSE register. In the worst case this method has $O(n \times m)$ time complexity but in the special case of short patterns it has good performance. This method scans the string in segments of 16 bytes (the size of the SSE register) until a full match is found. If a partial match is found the CMPISTRI instruction returns an offset to the beginning of the partial match; we then load the next 16 bytes and check whether the following segment matches the remaining subpattern.

Tables 5.7 and 5.8 summarize our CPU versus GPU comparison for the two different column sizes of the TPC-H workload. We focus on longer strings so we use the results of Table 5.7 in our later analysis but the results also favour GPUs for the shorter string experiments.

[4]This the maximum base frequency, not a boosted frequency that depends on power or temperature headroom.

	CPU (RE2)	CPU (CMPISTRI)	GPU	CPU+GPU
Price ($)	952	952	3100	4052
Query Performance (GB/s)	40.75	43.1	98.7	138.7
Energy consumed (J)	2.09	1.97	1.27	1.53
Performance / $	42.8	45.28	31.89	34.23

Table 5.7: CPU versus GPU comparison for Q16_1 and string size 1024 bytes.

	CPU (Boost BM)	CPU (CMPISTRI)	GPU	CPU+GPU
Price ($)	952	952	3100	4052
Query Performance (GB/s)	20.87	28.36	80.56	100.2
Energy consumed (J)	0.24	0.18	0.1	0.15
Performance / $	21.92	29.8	26.01	26.87

Table 5.8: CPU versus GPU comparison for Q16_1 and original string size (63 bytes).

The power rating of the K40 is 235W. For the two CPUs the aggregate power rating is 170W. The energy consumption does not include the RAM memory energy consumption while for the GPU the rating includes all GPU components. Tables 5.7 and 5.8 show that the estimated GPU energy consumption is at least 1.55x less than the CPU energy consumption for long and medium length strings. For long strings the energy consumed for the Q16_1 execution is 1.97J for the CPU implementation based on CMPISTRI and 1.27J for the GPU, so the GPU consumes 1.55x times less the energy than the CPU even without including the energy consumed from the CPU RAM. We also note the the power efficiency for the latest GPUs seems to be improving: The K80 processor has 1.65x the memory bandwidth of the K40 (480 GB/s) and 1.73x the number of cores while the power rating is only 1.27x of that of the K40 used in our experiments [NVIDIA, 2015e].

We compute the (MB/s)/($) rate to quantify the performance per $. The price of the K40 is $3100. The price for each CPU processor is $421 and to that cost we must add the price for 12GB DDR3 RAM, which is $110, so the total price for the CPU system is $952.[5] Using the above costs the performance / $ rate for the CPU is 45.28 and for the

[5]Prices were taken on 05/08/2015 from amazon.com.

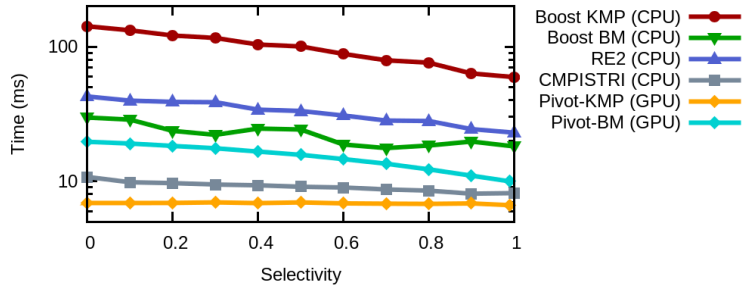


Figure 5.18: Time performance of CPU and GPU string matching for A1 and a 8-character pattern. The y-axis is logarithmic.

GPU 31.89 for long strings (Table 5.7). However, for medium length strings GPUs have similar performance per $ ratio: 26.01 versus 29.8 for the CPU system (Table 5.8). If we limit our comparison against CPU algorithms with linear worst-time complexity, the GPU has actually better performance per $. We found similar results for a Q13 TPC-H subquery, operating on shorter strings (average string length 47 characters). The results for this sub-query can be found in the Appendix.

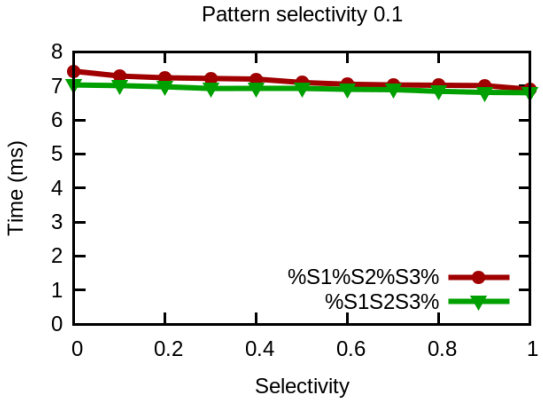


Figure 5.19: Time performance of KMP for two different predicates: '%S1%S2%S3%' vs. '%S1S2S3%'.

Figure 5.18 shows the performance of the CPU libraries and the GPU pivoted string matching methods for varying pattern selectivity. The performance difference between GPU and CMPISTRI is less significant for the synthetic dataset but Pivot-KMP method's performance is still 20-45% faster for any selectivity value. We also notice that RE2 has

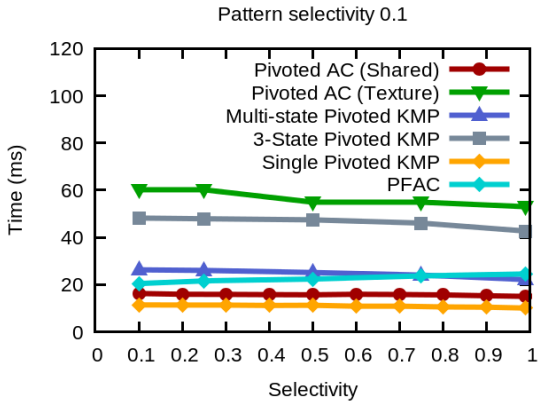


Figure 5.20: Performance of pivoted AC, pivoted KMP against PFAC for varying selectivity.

worse performance for the artificially generated dataset: This happens because RE2 can advance fast over a string if the first character of the pattern being searched does not appear in the input string. For example, when searching for the wildcard predicate '%Customer%Complaints' in Q16_1, character 'C' occurs only in about a third of the input strings so RE2 performance is relatively high. For the artificially generated dataset typically all 26 characters appear in each of the input strings so this optimization cannot be applied.

Figure 5.19 compares the performance of KMP for a LIKE `'%S1S2S3%'` query and a LIKE `'%S1%S2%S3%'` query. In the second query all string searches are implemented in one kernel. All three patterns are preprocessed and stored in an array of patterns. Each time a substring is found the array index is incremented and an overall match is found if all substrings are matched. KMP performance degrades slightly for queries involving a sequence of patterns because of the extra bookkeeping cost.

Figure 5.20 shows the performance of pivoted AC for varying selectivity, implementing the following LIKE predicate: LIKE `'%(S1|S2|S3)%'`. We compare the performance of AC to the performance of: a) KMP for the same query using split optimization, where each step matches one of the three patterns consecutively; b) single-pattern `KMP_hybrid` that performs a simpler query; c) `KMP_step` matching all three patterns by keeping a separate state for each pattern; and d) PFAC library using all optimizations [Lin *et al.*, 2013b]. We observe that AC, when storing the DFA in the shared memory, does not have significant overhead compared to single-pattern KMP, and that it is the superior method for multi-

Library	Machine	Cores	Bandwidth	Perf.(m=20)	Perf.(m=25)	(Avg Perf.)/$
Self-Pivot-KMP	Tesla C2070	448	144 GB/s	15.21GB/s	15.02GB/s	12.1
Self-Pivot-KMP	Kepler K40	2880	288 GB/s	55.37GB/s	55.13GB/s	17.82
Pivot-KMP	Tesla C2070	448	144 GB/s	14.93GB/s	15.27GB/s	12.08
Pivot-KMP	Kepler K40	2880	288 GB/s	54.53GB/s	55.06GB/s	17.67
PFAC	Kepler K40	2880	288 GB/s	26.3GB/s	26.3GB/s	8.5
KMP	Tesla K20m	2496	208 GB/s	19.76GB/s	N/A	7.39
BMH	GTX 280	240	141.7 GB/s	N/A	1.2GB/s	10

Table 5.9: Performance (GB/s) of our matching methods versus the published performance of other GPU libraries (bottom three lines).

	Short pat.(4)	Medium pat.(16)	Long pat. (64 chars)
Low sel. (0.05)	Pivot-KMP	Pivot-KMP	Pivot-BM
Medium sel. (0.5)	Pivot-KMP	Pivot-KMP	Pivot-KMP
High sel. (0.9)	Pivot-KMP	Pivot-KMP	Pivot-KMP

Table 5.10: Best average case performance for workload A1 for different query parameters.

pattern matching. PFAC loads the first row in the shared memory which we have not implemented in our AC implementation using texture memory. Our AC implementation using shared memory is slightly faster than PFAC for a small number of patterns. The most significant advantage is that its performance does not degrade dramatically for adversarial inputs: PFAC performance for worst-case inputs degrades by up to 20x [Lin *et al.*, 2013b], while our performance degrades by less than 2x.

Table 5.9 compares published performance numbers and the measured performance of our GPU implementation on workload R2. This table shows that we significantly outperform prior methods, even when adjusting for differences in GPU capabilities. Self-pivot KMP has similar performance to Pivot-KMP with the additional benefit that it can coalesce memory even when GPU operates on a subset of the table given by an rid-list. Table 5.10 summarizes which algorithm-parallelism combination is optimal for different query parameters under

[6]Prices were taken on 05/08/2015 from amazon.com.

workload A1. We observe that Pivoted KMP is the best algorithm because it has the best performance in 8/9 cases. The competing algorithm is Pivot-BM for low selectivity queries and longer patterns: BM can effectively skip over large segments of text. For the worst-case inputs KMP has always superior performance making it the best choice overall.

5.5 Conclusions and Future Work

We advocate using the GPU as a coprocessor for string matching and KMP as the preferred string matching algorithm. String matching is an interesting application to evaluate the effect of thread and memory divergence on GPU kernel performance which has a fair number of different dimensions. We suggest multiple parallelism methods for string matching and study the performance of the state-of-the art algorithms on two different GPUs. We analyze alternative string layouts in the global memory and suggest different performance optimizations for string matching algorithms. Our solution optimizes string search by selecting the right parallelism granularity and string layout for the different algorithms. The performance of our proposed methods exceeds that of other CPU and GPU implementations.

Chapter 6

SIMD-Accelerated Regular Expressions

6.1 Introduction

Modern hardware advances have made a substantial impact on the design and implementation of database systems. The increase in RAM capacity allows the average-sized database to fit in main memory. With main-memory-resident data, the performance bottleneck shifts from disk speed to the RAM bandwidth, an order of magnitude higher.

Improving CPU efficiency for in-memory query processing seeks to explore all kinds of parallelism provided by modern CPUs in order to saturate the RAM bandwidth. With the advent of multi-core CPUs, multi-threading is now the most fundamental optimization for performance-critical tasks.

In the context of databases, scan operators, besides using multiple threads, also utilize SIMD vector instructions to maximize efficiency. When the selective predicates are simple, such as comparing a constant against a numeric column, multi-threaded scans in SIMD code process data faster than can be loaded, reaching the RAM bandwidth bottleneck.

A significant collaborator in this project was Orestis Polychroniou. He contributed to shaping the research problem during our discussions and writing our solution overview. He also wrote the vectorized code for the regular expression matching.

6.1.1 Substring Matching

Many algorithms have been proposed to accelerate substring matching, the most well known being Knuth-Morris-Pratt [Knuth *et al.*, 1977] and Boyer-Moore [Boyer and Moore, 1977]. Both algorithms improve over the worst-case $O(n^2)$ brute-force solution, by employing pre-computed offset arrays, in order to achieve $O(n)$ worst case complexity. The pre-processing step is dependent on the pattern only and is trivial in databases where a single pattern is matched against many tuples. The implementation of Boyer-Moore for matching a single pattern is shown below. The `pat_jmp` and `sym_jmp` arrays are pre-computed once.

```
bool like(const char *string, const char *pattern,
          size_t str_len, size_t pat_len, [...]) {
   size_t i = pat_len - 1;
   while (i < str_len) {
      uint8_t b = string[i];
      size_t j = pat_len - 1;
      while (b == pattern[j]) {
         if (j == 0) return true;
         b = string[--i], j--; }
      i += max(pat_jmp[j], sym_jmp[b]); }
   return false; }
```

Luckily, recent mainstream CPUs offer a specialized SIMD instruction to process strings that can also be used to implement substring matching. Specifically, the SSE 4.2 128-bit SIMD instruction set in x86 processors provides the `cmp?str` instructions that can match against patterns that fit in a SIMD register. The algorithm resembles the brute force approach, but runs in $O(n)$ time for patterns up to 16 bytes.

```
bool like(const char *string, __m128i pat, [...]) {
   size_t i = 0;
   while (i + 16 < str_len) {
      __m128i str = _mm_loadu_si128(&string[i]);
      size_t j = _mm_cmpistri(pat, str, 12);
      if (j >= 16) i += 16;
      else {
```

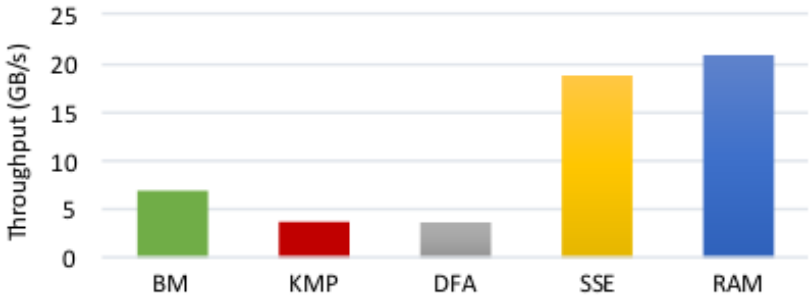


Figure 6.1: Substring matching for TPC-H Q13.

```
        if (j + pat_len <= 16) return true;
        i += j; } }
    if (i + pat_len <= str_len) {
       __m128i str = _mm_loadu_si128(&string[i]);
       size_t j = _mm_cmpestri(pat, pat_len, str,
                                  str_len - i, 12);
       if (j < 16 && j + pat_len <= 16) return true; }
    return false; }
```

Figure 6.1 shows the performance of different algorithms for substring matching on TPC-H Q13 (scalar factor 300) using multiple threads on a mainstream 4-core CPU. The query has a `like '%special%packages%'` operator that matches patterns `special` and `packages` in that order. By nesting two calls of substring matching that return the match position, we can implement a conjunction of pattern matches.

Substring matching without using the specialized hardware instruction is far from the RAM bandwidth, due to branch dependencies for every character of the input string. Knuth-Morris-Pratt (KMP) is very similar to a deterministic finite automaton (DFA) that matches the same pattern, but uses an ad-hoc jump table for failed matches. The DFA is implemented using a 2D transition table for each state times all possible values per string character. The number of states is equal to the pattern length. As expected, KMP and the DFA have similar performance since both scan the entire string if there no match. Boyer-Moore (BM) is much faster than KMP due to skipping a large portion of each input string. Still, we cannot saturate the RAM bandwidth, even with multiple threads. Note that the we

generate a bitmap and thus selectivity does not affect the performance.

6.1.2 Regular Expression Matching

While a single instruction is enough to cover most queries with substring matching operators, more advanced predicates such as regular expression matching cannot be optimized as easily. Popular databases offer regular expression matching predicates such as `regexp_like` in Oracle DB, or `rlike`/`regexp` in MySQL. For example, this MySQL query counts the employees with valid e-mail addresses:

```
select count(*) from employees
where email regexp  # or "rlike"
'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+.[A-Za-z]2,4$'
```

To match a string against a regular expression, we typically construct a DFA. DFAs have a number of states that transition to other states based on the next character of the string. Because each character is processed only once, DFAs are worst-case $O(n)$ to the input string length. The DFAs are represented by a $s \times c$ transition table having s states and c possible character values. The number of states s is dependent on the complexity of the regular expression.

We show a DFA that validates e-mails in Figure 6.2. The DFA has 9 states and `S` is the starting state. The double-circled states `T2`, `T3`, and `T4` are also accepting states.

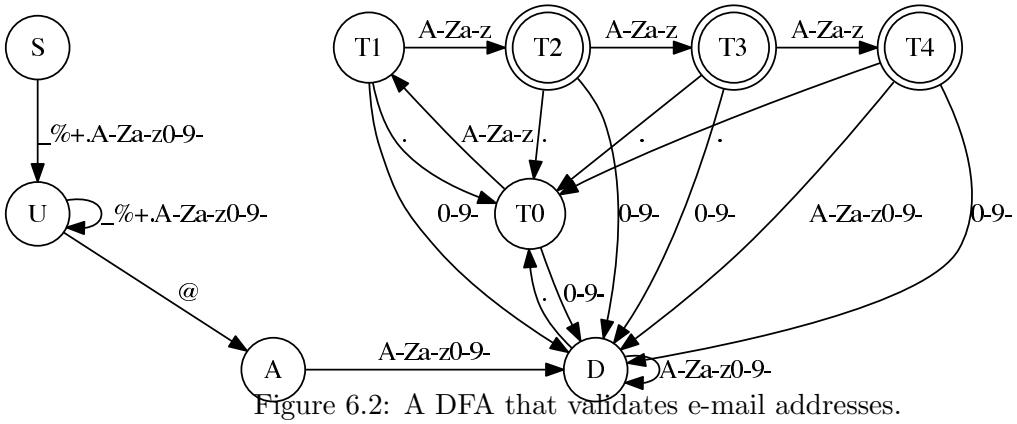


Figure 6.2: A DFA that validates e-mail addresses.

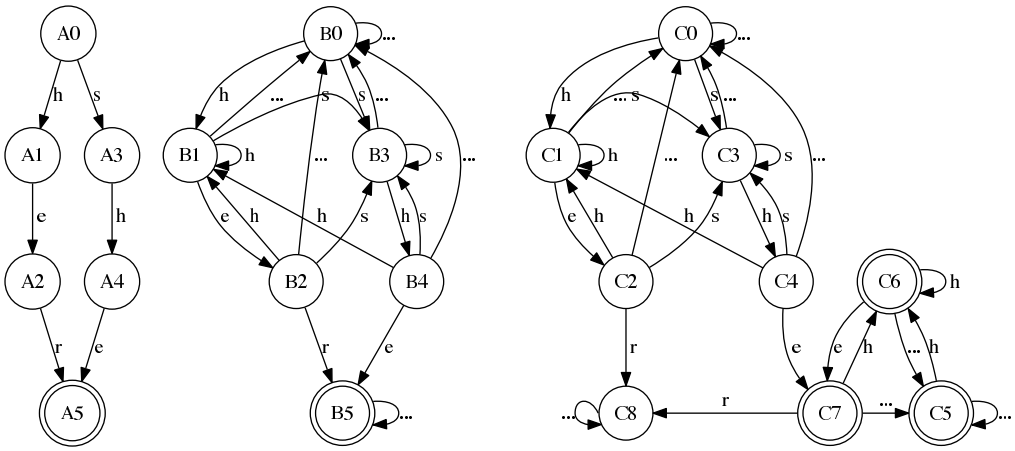


Figure 6.3: DFAs for combinations of: she, her.

All regular expressions have a DFA that matches them. A DFA can be costructed automatically from a regular expression and the number of states can be minimized, which is still a pre-processing step in the context of databases. Regular expressions of course cover all logical combinations substring matching. For example, the selection filter `like '%special%' or like '%packages%'` can either use two calls of substring matching and combine their result, or use a DFA to match both words simultaneously. The DFA matches input strings in linear time regardless of the number of patterns, but its size grows if more words have to be matched. A well known algorithm for multi-pattern substring matching is Aho-Corasick [Aho and Corasick, 1975]. Aho-Corasick places all patterns in a trie, implemented as a DFA, and keeps a separate transition table for failed matches, similar to KMP. However, the extra table can be encoded in the DFA transitions. Then, Aho-Corasick becomes identical to the minimal DFA. However, unlike Aho-Corasick, DFAs can accept all logical combination of positive and negative patterns. Figure 6.3 shows: (i) the trie that accepts `her` or `she` (`A` states), (ii) a DFA that accepts the same as substrings (`B` states), and (iii) a DFA that accepts `she` but rejects `her` as substrings (`C` states).

In Figure 6.2, there is an implicit extra state that works as a reject *sink*. An e-mail is invalid if we encounter an invalid character and can stop the DFA traveral immediately. In Figure 6.3 `B5` is an accept sink and `C8` is a reject sink. A DFA can have both. These special states allow us to accept or reject a string early, which is crucial for performance in some

DFAs, but can also introduce branch mispredictions.

We show how to implement regular expression matching via DFA traversal in vectorized SIMD code. Our implementation traverses the DFA for multiple strings at a time. We employ non-contiguous loads (gathers) to access different parts of multiple input strings without assuming any particular alignment. We also use gathers to traverse the DFA for multiple strings in parallel. Finally, we use branchless vectorized code to store the rids of matching tuples, while dropping strings that reach a sink state early.

Our approach works on both recent mainstream CPUs (Intel Haswell) and co-processors (Intel Xeon Phi) and is independent of the SIMD length. Our experimental evaluation shows that our method achieves more than 5X improvement on co-processors and significant improvement (50% to 80% of the memory bandwidth) on mainstream CPUs, offering a crucial tool to support fast regular expression matching.

In Section 6.2 we present related work. In Section 6.3 we describe our vectorized implementation for regular expression matching, including details such as how to access the input strings, how to traverse the DFA, and how to drop early failures. In Section 6.4 is our evaluation and we conclude in Section 6.5.

6.2 Related Work

As seen in the introduction, several algorithms have been proposed for substring matching. Knuth-Morris-Pratt [Knuth *et al.*, 1977] and Boyer-Moore [Boyer and Moore, 1977] are the most well known for single pattern matching, while Aho-Corasick [Aho and Corasick, 1975] and Commentz [Commentz-Walter, 1979] support multiple patterns. To convert a regular expression to a DFA with minimum number of states, we first translate it to an NFA, then convert to a larger DFA, and then minimize the states [Hopcroft, 1971]. This process is not time-critical in databases where expensive pre-processing is acceptable.

SIMD vector instructions were used to parallelize business text analytics workloads [Salapura *et al.*, 2012]. In Chapter 5 we used GPUs for efficient substring matching. Cell SPEs were used for fast network filtering [Kulishov, 2009]. Cell processors also accelerated DFA matching on multiple inputs [Scarpazza *et al.*, 2007; Iorio and Lunteren, 2008]. Com-

plementary techniques parallelized DFA matching within the same input [Mytkowicz *et al.*, 2014]. A space-efficient technique to split Aho-Corasick in two steps involving a DFA and an NFA that can also use SIMD instructions was proposed [Yang *et al.*, 2010]. Large DFAs were partitioned to fit in the Xeon Phi co-processor cache [Tran *et al.*, 2014]. Nevertheless, DFAs used in databases rarely exceed the CPU cache size. Advanced vector instructions, such as gathers, have been used to optimize in-memory database operators [Polychroniou and Ross, 2014; Polychroniou *et al.*, 2015].

6.3 Implementation

Regular expressions have a single DFA with the mininum number of states that matches them. What makes the DFA deterministic is that there is only one possible transition per state and character. Thus, we can represent the DFA as a $s \times c$ array with s states and c transitions where c is the size of the alphabet. Here we will use $c = 256$ to cover every value of each byte. Each state can either be accepting or rejecting. To avoid explicitly storing this information, we reorder the states placing the first s_{rej} rejecting states in rows $[0, s_{rej})$ and the rest s_{acc} accepting states in rows $[s_{rej}, s_{acc} + s_{rej})$. We use special numbers for sink states without storing any transitions, as these states only loop back to themselves.

Because the 2D-array representation of the DFA has always c transitions per state, we can map the 2D-array into an 1D-array and use arithmetic to compute the indexes. We start from the initial state and combine the state number with the next byte to find the position of the transition. The loop stops either when the string ends or if we reach one of the two sink states. The scalar code is shown below:

```
bool rlike(const uint8_t *string, size_t str_len,
           const ssize_t *dfa, [...]) {
   size_t i = 0, s = initial_state;
   do {
       s = dfa[(s << 8) | string[i]];
   } while (0 <= (ssize_t) s && ++i != str_len);
   return s + 1 > reject_states; }
```

The snippet is inlined in a loop that scans over the string column and stores the rids

of the accepted strings. A few optimizations we do to minimize the number of branches is to set the transtions to the negative and the positive sink to `-1` and `-2` respectively. The `0 <= (ssize_t) s` signed integer comparison tests whether the state is a sink or not. The `s+1 > reject_states` unsigned integer comparison tests whether the state is in the range $[-1, s_{rej})$, thus the string should be rejected (we use unsigned arithmetic and exploit the `s+1` unsigned overflow). By minimizing the branch tests and using simple arithmetic to access the DFA transition table, we ensure that the scalar code is as fast as possible.

If the number of states is small, we can write the DFA as a byte array (if $s < 255$) and shrink its memory footprint to fit it in the cache. In the database context, where the regular expression is specified in the query, the DFA is small and will probably fit in the L1 cache. Thus, performance is determined primarily and the L1 access latency for accessing new states, and by the number branch mispredictions if the strings are accepted or rejected early. Note that while there are no explicit `if` statements in the code, branches still occur if strings fail early. If we always have to process the entire string (e.g. because there are no transitions to sink states), the inner loop will execute a specific number of repeats and will not incur any branch mispredictions on advanced CPUs.

At first glance, optimizing the above snippet looks hardly possible, especially given that there are really no compute instructions to vectorize. For instance, there is no hash function compute. We will show that even such simple loops can get significant speedups from vectorized SIMD code. The only assumption we make is that the strings have fixed lengths, which is a common assumption of main-memory column stores to allow accessing string columns using record identifiers (rids). If the strings are shorter, they can be padded with some special character that is not part of the regular expression and transitions to a sink state in the DFA.

The fundamental principle in the vectorized approach is to process a different string per vector lane. This has been shown in previous work [Polychroniou and Ross, 2014; Polychroniou *et al.*, 2015] to work very well, especially on the co-processors that use very simple cores augmented with large vector units. Also, the number of operations drop to $O(1/W)$ (W the number of vector lanes) compared to the scalar code. Strings that reach some sink state should be not continue traversing the DFA. Thus, we need to access the input

out-of-order, creating new rids to replace the rids of strings that have finished processing. We also store the rids of processed strings that reached an accepting DFA state. A simplified version of the vectorized approach is shown below.

$\vec{r} \leftarrow rids_{in}[0]$ ▷ *rids of strings being processed*

$\vec{s} \leftarrow s_{initial}$ ▷ *state of strings being processed*

$\vec{o} \leftarrow 0$ ▷ *offset in strings being processed*

$i \leftarrow W$, $j \leftarrow 0$ ▷ *input & output index*

while $i \leq N$ **do**

$\vec{d} \leftarrow \text{strings}[\vec{r} \cdot l + \vec{o}]$ ▷ *gather next character(s)*

$\vec{s} \leftarrow \text{dfa}[(\vec{s} << 8) \text{ — } \vec{d}]$ ▷ *transition to next state(s)*

$m \leftarrow (\vec{s} + 1 > s_{reject})$ & $(\vec{o} = l)$ ▷ *check if accepted*

$\text{rids}_{out}[j] \leftarrow_m \vec{r}$ ▷ *store rid(s) (if accepted)*

$j \leftarrow j + |\vec{m}|$ ▷ *update output index*

$m \leftarrow (\vec{s} < 0) \text{ — } (\vec{o} \geq l)$ ▷ *check if finished*

$\vec{r} \leftarrow_m \text{rids}_{in}[0]$ ▷ *replace rid (if finished)*

$i \leftarrow i + |\vec{m}|$ ▷ *update input index*

$\vec{s} \leftarrow m \ ? \ s_{initial} : \vec{s}$ ▷ *reset DFA state (if finished)*

$\vec{o} \leftarrow m \ ? \ 0 : \vec{o}$ ▷ *reset in-string offset (if finished)*

end while

The notation is taken from earlier work [Polychroniou *et al.*, 2015] and is summarized here for clarity. $\vec{x} \leftarrow A[\vec{i}]$ is a gather operation using $\vec{i}$ for the indexes. $\vec{x} \leftarrow_m A[i]$ is a selective load where only the lanes specified in bitmap m are replaced with data loaded sequentially string from memory location A. $A[i] \leftarrow \vec{d}$ is a selective store where only the lanes specified in m are stored sequentially starting from A. $\vec{x} \leftarrow m \ ? \ \vec{y} : \vec{z}$ copies the values of each lanes from either $\vec{y}$ or $\vec{z}$, based on the bitmap m. The notation $|m|$ denotes the number of set bits in m. Scalar values in vector operations are broadcast to all lanes of a vector. For example, $\vec{x} \leftarrow \vec{x} + c$ adds c to all lanes of $\vec{x}$.

In the example algorithm, the rids are explicitly loaded from an input array. If the strings are just scanned in order, we can implicitly generate rids incrementing from 1 to N. Figure 6.4 illustrates this functionality. The output rids are also generated out-of-order.

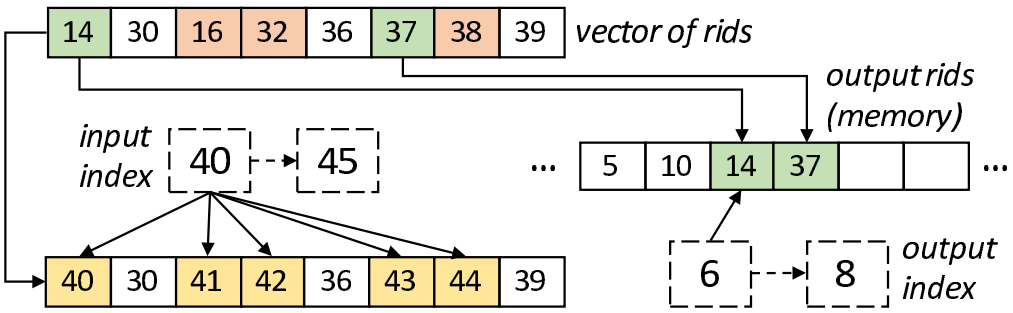


Figure 6.4: Selective loads & stores of rids.

The difference of the above code with the scalar code is that it converts all conditional control flow into branchless data flow. Since the input is no longer accessed in order, we use vector gather to load the bytes from the strings, compared to the scalar code that processed a single string and accessed the bytes sequentially.

The inefficiency of the algorithm shown above is that it only processes one byte at a time from each string. In string processing, we expect to process a non-trivial portion of each string to match the regular expression. Thus, processing more than one byte for each time we load new rids from input and store rids of matching strings to the output. Instead of doing a cache access for one byte per string, we could instead load multiple consecutive bytes, at least as big as a processor word (32-bit or 64-bit). CPU caches are equally fast whether we access 1 byte, or 8 bytes (aligned). Even aligned 32-byte vector accesses can be equally fast in the latest CPUs.

When gathering string data from arbitrary offsets, the accesses may not be aligned in 4-byte word boundaries. For example, if the string length is 15, the second string will start from the 15^{th} byte. Even if the processor allows scalar loads of 4-byte words or vectors loads to be unaligned, vector gathers of words may still require to be aligned on word boundaries. Mainstream CPUs (Intel Haswell) support unaligned gathers in SIMD (AVX 2), thus we can directly use 64-bit unaligned gathers. Xeon Phi co-processors however, require the gathers to be aligned and thus we have to do the alignment manually. To implement unaligned gathers in software, we issue aligned word gathers and align the bytes manually. In detail, we align the byte offset to an 4-byte int offset, gather 2 consecutive 4-byte ints per string, and then align the 4-byte ints using variable-stride shifts. The process is shown in Figure 6.5.

If we issue the minimum two 4-byte gathers, the number of usable bytes varies depending on the possible alignments of strings. If the (fixed) string length is a multiple of 4, then all strings will be aligned in 4-byte word boundaries and all 8 bytes are valid, unless we exceed the string length. If the string length is a multiple of 2, then all srings are aligned in 2-byte boundaries and at least 6 bytes are valid. Otherwise, at least 5 bytes are valid.

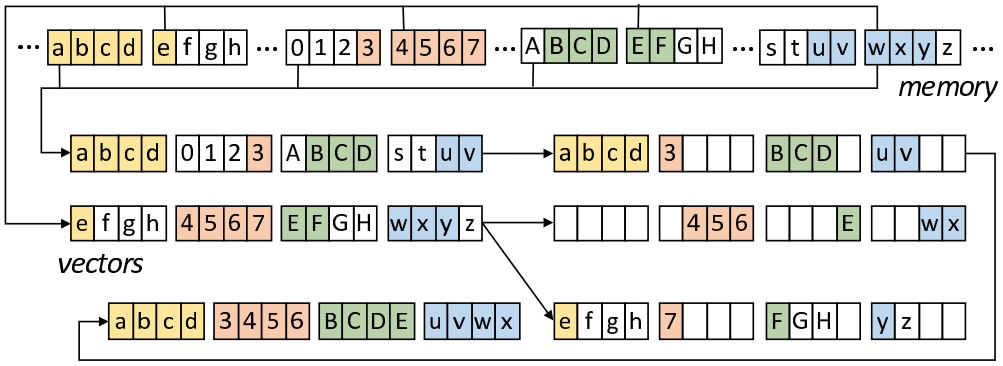


Figure 6.5: Unaligned vector gathers in Xeon Phi.

The Xeon Phi code for gathering string bytes is shown below.

```
// compute index:  rid * length + offset
__m512i p = _mm512_fmadd_epi32(rid, len, off);
// gather 8 bytes per string
__m512i p4 = _mm512_srli_epi32(p, 2);
__m512i w1 = _mm512_i32gather_epi32(p4, &strs[0], 4);
__m512i w2 = _mm512_i32gather_epi32(p4, &strs[4], 4);
// compute right shift strides:  (3 & p) << 3
__m512i sr = _mm512_and_epi32(p, m3);
shr = _mm512_slli_epi32(shr, 3);
// compute left shift strides:  32 - shr
__m512i shl = _mm512_sub_epi32(m32, shr)
// align first word:  w1 = (w1 >> shr) | (w2 << shl)
w1 = _mm512_or_epi32(_mm512_srlv_epi32(w1, shr),
                     _mm512_sllv_epi32(w2, shl));
// align second word:  w2 >>= shr
w2 = _mm512_srlv_epi32(w2, shr);
```

A guide to the vector intrinsics is available online.[1] If a compact main-memory layout of the string column is not important, we can pad the strings to lengths that are multiples of 4 to avoid the manual alignment and use 8 out of 8 gathered bytes. The same technique can be used in CPU with unaligned gather support to reduce the latency.

To traverse the DFA, we use inner loops for each word of bytes that was gathered. The loop repeats are only dependent on the fixed string length and thus should not incur branch mispredictions. While we avoid reloading rids from the input or storing rids to the input inside these loops, we still have to check whether the string has reached a sink state or the end of the string. The Xeon Phi code is shown below. In some extreme cases, it may be faster to test whether all vector lanes are invalid and exit the inner loops. On average, however, we expect the strings to be larger than 5–8 bytes, thus winning back the overhead of a few redundant loops after reaching a sink state or the end of the string.

```
// isolate next byte per string
__m512i b = _mm512_and_epi32(w1, mFF);
// compute index in transition table
__m512i p = _mm512_slli_epi32(s, 8);
p = _mm512_or_epi32(p, b);
// gather new states (assuming 8-bit DFA array)
s = _mm512_mask_i32extgather_epi32(s, k, p, dfa,
    _MM_UPCONV_EPI32_SINT8, 1, 0);
// increment offset for valid lanes using a -1 mask
off = _mm512_mask_sub_epi32(off, k, off, m1);
// shift word to get next string byte
w1 = _mm256_srli_epi32(w1, 8);
// update valid lanes: check for sink state (s > -1)
k = _mm512_mask_cmpgt_epi32_mask(k, cur, m1);
// update valid lanes: check for end of string
k = _mm512_mask_cmpgt_epi32_mask(k, len, off);
```

Finally, we note that the gathers to the DFA transition table cannot be optimized. Even if the DFA is a table of bytes, there is no use for the nearby bytes that would fit in the same

[1] software.intel.com/sites/landingpage/IntrinsicsGuide/

processor word. An interesting observation is that if the hardware does not support single byte gathers, the cost of converting (4-byte) int gathers to bytes using shifting is expensive and adds significant overhead to the critical path. Xeon Phi supports this functionality but the latest CPUs (AVX 2) do not. Thus, on the CPU implementation we found that storing the transition table of small DFAs using 4-byte ints rather than bytes makes traversal faster, even if it quadruples its size. The increase in the DFA footprint will in most cases not affect performance, because CPU gathers (AVX 2) are equally fast in the L1 and the L2 cache [Hofmann and others, 2014].

A common optimization applied to vector implementations is to unroll all instructions by repeating them for multiple instances of data. Unrolling hides the latencies among instructions, and improves performance even in aggressively out-of-order CPUs with simultaneous multi-threading (SMT). We can apply 2-way loop unrolling here by generating rids from 1 to N and N to 1 in the same loop, until the two rid offsets meet in the middle. In the end, a few rids will be processed in scalar code but their number is trivial ($< 4W$). The number of variables that hold the state of the two instances is doubled and thus we must ensure that the number of physical registers suffices to avoid register spilling.

Overall, the vectorized implementation improves regular expression matching via DFA traversal when the DFA is cache-resident. If the DFA does not fit in the cache (which is rare), we expect the performance to be dominated by RAM accesses. Note that reducing the RAM latency through pre-fetching is not possible here because we cannot predict the state at which the DFA will be in a generic way. A more practical solution is to partition the DFA and apply our technique to each cache-resident partition. Nevertheless, reducing the DFA footprint is out of the scope of this thesis.

6.4 Experimental Evaluation

Our evaluation was done on two platforms. The first platform is an Intel Xeon E3-1275v3 CPU with 4 Intel Haswell cores and 2-way SMT running at 3.5 GHz. The CPU has accesses to 32 GB of 1600 MHz DDR3 ECC RAM with a peak load bandwidth of 21.8 GB/s and supports 256-bit SIMD (AVX 2). It runs Linux 4.2 and we compile using GCC 5.2 with `-O3`.

The second platform is an Intel Xeon Phi 7120P co-processor with 61 modified P54C cores and 4-way SMT running at 1.238 GHz. The co-processor has access to 16 GB of on-chip GDDR5 RAM with a peak load bandwidth of 212 GB/s and supports 512-bit SIMD. It runs embedded Linux 2.6 and we compile using ICC 17 with `-O3`. We also tested ICC on the CPU, but GCC was marginally faster.

All figures show the performance of three methods on both platforms. The three methods shown are the scalar implementation (`Scalar`), the vectorized implementation without loop unrolling (`Vector (x1)`), and vectorized implementation 2-way loop unrolled (`Vector (x2)`). In all cases, we scan a fixed-length string column with synthetically generated data tailored to each regular expression to meet specific criteria per experiment. The method scan over the column and the rids of the strings that match the regular expression. The rids are implicitly generated from 0 to N-1. Each method is run in a shared nothing fashion using multiple threads, but the DFA is shared (read-only). The DFAs are compressed to byte if the number of states is small enough unless we run on the CPU where 32-bit gathers are faster. Unless otherwise specified, we use all available hardware threads, the selectivity is 1%, and the string length is 32.

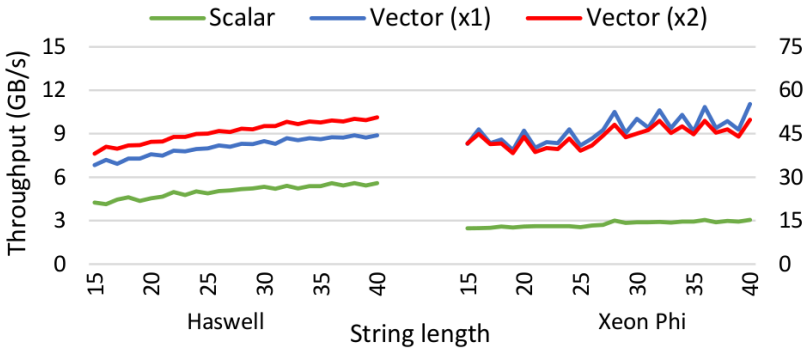


Figure 6.6: Varying string lengths (URL validation)

Figure 6.6 shows the throughput of regular expression matching by varying the string length. The DFA validates whether the input string is a valid URL using a complicated regular expression. The DFA has 90 states (64 rejecting states, 26 accepting states, and the reject sink) and has a memory footprint of 23 KB if stored as a byte array. The selectivity is 1% and we process half the string bytes on average to reach the reject sink state. The speedup on the Haswell CPU is 1.75–1.9X and increases with the string length. The scalar

code uses up to 25% of the memory bandwidth while the vector code uses up to 45% of the bandwidth. Loop unrolling boosts the vectorized method up to 14% On the Xeon Phi co-processor, the vectorized code is 3.1–3.7X faster and increases the bandwidth usage from 7% to 26%. Loop unrolling is actually slower on the Xeon Phi since each core executes in-order and has 4-way SMT. Both platforms are severely compute-bound since we access the cache per byte of input. Xeon Phi exhibits spikes due to unaligned gathers being slightly faster when the strings are 4-byte aligned.

In Figure 6.7, we use the same DFA with the same settings, but we fix the string length to 32 and vary the failure point of the input strings. The failure point represents the number of bytes processed per string, or the average number of transitions in the DFA until we reach the reject sink state.

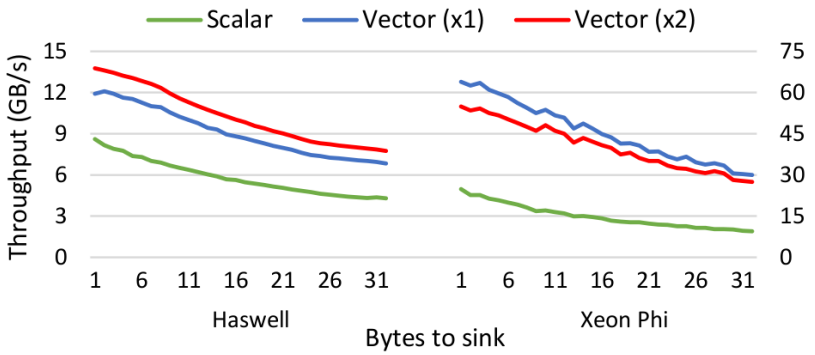


Figure 6.7: Varying the failure point (URL validation)

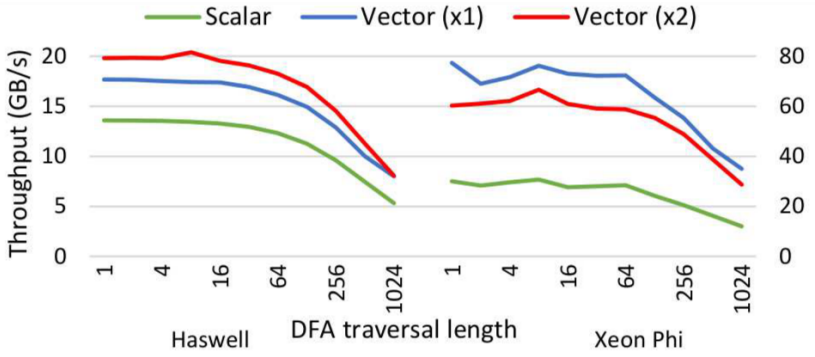


Figure 6.8: Varying the failure point (URL validation, long strings)

The vectorization speedup in the CPU is 1.6–1.85X and increases the bandwidth usage from a 26% to an 47% averaged across all failure points for strings of length 32. Loop unrolling

boosts performance up to 13%. In the co-processor, the vectorized code is 2.7–3.3X faster and uses up to 26% of the bandwidth. Loop unrolling again reduces performance. In Figure 6.8, we fix the string length to 1024 characters. Similarly, loop unrolling boosts performance on Haswell while it reduces performance on Xeon-Phi. For both processors, the performance drop is more dramatic when more than 64 characters are processed on average. The cache line size in both processors is 64 bytes so when more than 64 bytes are processed per string, the total number of memory accesses increases.

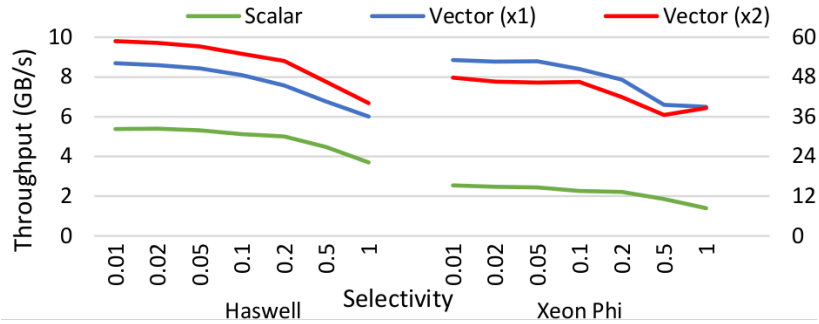


Figure 6.9: Varying the selectivity (URL validation)

In Figure 6.9, we use the same URL-validating DFA. The string length is fixed to 32 and we vary the selectivity rate. For the strings that do not match, we process half the bytes on average to traverse the DFA until the rejection is determined. In the CPU, we get 1.75–1.8X vectorization speedup and increase the bandwidth usage from 25% to 45% in low selectivities. The throughput drops by 32% when the selectivity reaches 100%. When the selectivity is near 0%, up to 45% of the load bandwidth is utilized. In the co-processor, the vectorized code is 3.5–4.7X faster, which is maximized when the selectivity is close to 100%. The speedup is maximized on high selectivity because the entire string must be processed to determine if it is a valid URL. Since we are compute-bound by the DFA accesses in the cache per byte of input, generating an array of rids for the accepted strings, does not affect performance unless the strings are very short.

Figure 6.10 varies the DFA size using multi-pattern substring matching. We vary the number of words that are inserted in the DFA so that its states are 10^k and in some cases exceed the cache size. The selectivity is 1% but the inputs are generated by appending valid dictionary words picked randomly, in order to ensure that we traverse long paths in

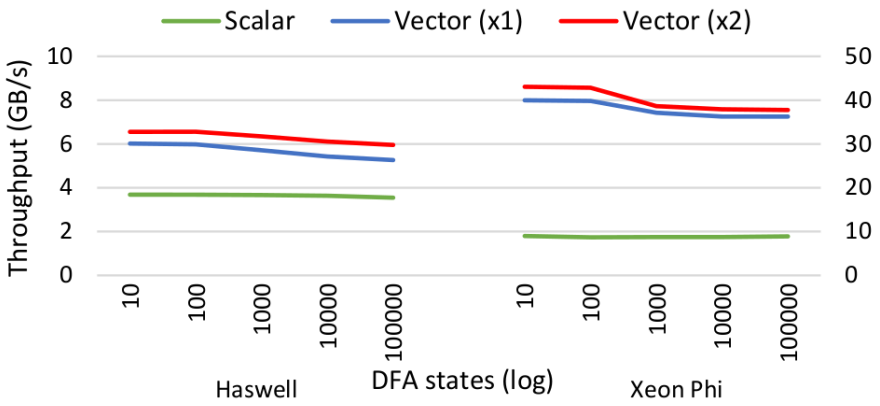


Figure 6.10: Varying the DFA size (DFA for multi-pattern matching using English dictionary words)

the DFA. In the CPU, the speedup is 1.75–2.5X and is maximized when the DFA is large. This implies that out-of-cache access latencies are exacerbated when tied with control flow dependencies, which is also supported by the fact that loop unrolling improves performance up to 45% on larger DFAs. In the co-processor, the speedup is 1.05–3.7X and is maximized when the DFA is small. Eliminating control flow dependencies is not useful on the in-order cores of Xeon Phi.

Figure 6.11 shows the scalability of all methods using a small multi-pattern matching DFA that contains both positive and negative examples. We use this DFA to illustrate that our approach is more general than multi-pattern matching. Performance scales almost linearly with the number of threads, even using SMT. On the Xeon Phi, this is expected due to high-latency vector instructions. On the Haswell, this means that the operation is compute-bound. Interestingly, using both SMT threads and loop unrolling gives marginal improvement thus we largely saturate instruction-level and thread-level parallelism and are bound by the cache accesses.

6.5 Conclusions

We presented generic vectorized implementations of regular expression matching for in memory string analytics. Our implementations are based on DFAs and are efficient both for general regular expression predicates and multi-pattern matching. We describe how to

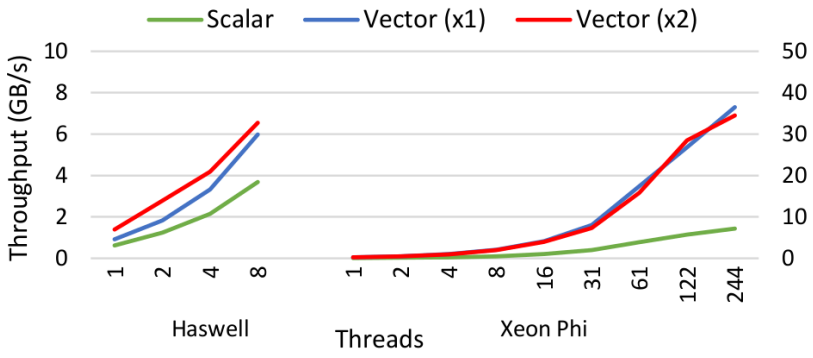


Figure 6.11: Scalability (multi-pattern matching both positive and negative using English dictionary words)

handle irregular memory accesses both on the input strings and the DFA state transition table using gather intrinsics. Our solution achieves up to 1.9X speed-ups on multi-core processors and 5X on many-core accelerators highlighting the impact of vectorization on analytics wokloads that are bound by cache accesses rather than compute instructions.

Chapter 7

Decompression Acceleration

This research project is joint work with my mentors during my IBM summer internships and my advisor Ken Ross. During my IBM internship in summer 2013, I was mentored by Rene Mueller, Tim Kaldewey, and Guy Lohman. They designed the project specifications and contributed to the bibliographic research by suggesting related work. Rene Mueller and Tim Kaldewey provided technical mentorship to profile existing compression frameworks. Their technical contribution also extended to refining my software design of our GPU compression framework. Tim Kaldewey programmed the first implementation of our CPU compression framework. Finally, all of my collaborators contributed to the final presentation of our work.

7.1 Introduction

With exponentially-increasing data volumes and the high cost of enterprise data storage, data compression has become essential for reducing storage costs in the Big Data era. There exists a plethora of compression techniques, each having a different trade-off between its compression ratio (compression efficiency) and its speed of execution (bandwidth). Most research so far has focused on the speed of compressing data as it is loaded, but the speed of decompressing that data can be even more important for Big Data workloads – usually data is compressed only once at load time but repeatedly decompressed as it is read when executing analytics or machine learning jobs. Decompression speed is therefore crucial to

minimizing response time of these applications, which are typically I/O-bound.

In an era of flattening processor speeds, parallelism provides our best hope of speeding up any process. In this work, we leverage the massive parallelism provided by Graphics Processing Units (GPUs) to accelerate decompression. GPUs have already been successfully used to accelerate several other data processing problems, while concomitantly providing a better Performance/Watt ratio than conventional CPUs, as well. However, accelerating decompression on massively parallel processors like GPUs presents new challenges. Straightforward parallelization methods, in which the input block is simply split into many, much smaller data blocks that are then processed independently by each processor, result in poorer compression efficiency, due to the reduced redundancy in the smaller blocks, as well as diminishing performance returns caused by per-block overheads. In order to exploit the high degree of parallelism of GPUs, with potentially thousands of concurrent threads, our implementation needs to take advantage of both intra-block parallelism and inter-block parallelism. For intra-block parallelism, a group of GPU threads decompresses the same data block concurrently. Achieving this parallelism is challenging due to the inherent data dependencies among the threads that collaborate on decompressing that block.

We propose and evaluate two approaches to address this intra-block decompression challenge. The first technique exploits the SIMD-like execution model of GPUs to coordinate the threads that are concurrently decompressing a data block. The second approach avoids data dependencies encountered during decompression by pro-actively eliminating performance-limiting back-references during the compression phase. The resulting speed gain comes at the price of a marginal loss of compression efficiency. We also present **Gompresso/Bit**, a parallel implementation of an Inflate-like scheme [Deutsch, 1996] that aims at high decompression speed and is suitable for massively-parallel processors such as GPUs. We also implement **Gompresso/Byte**, based on LZ77 with byte-level encoding. It trades off slightly lower compression ratios for an average 3× higher decompression speed.

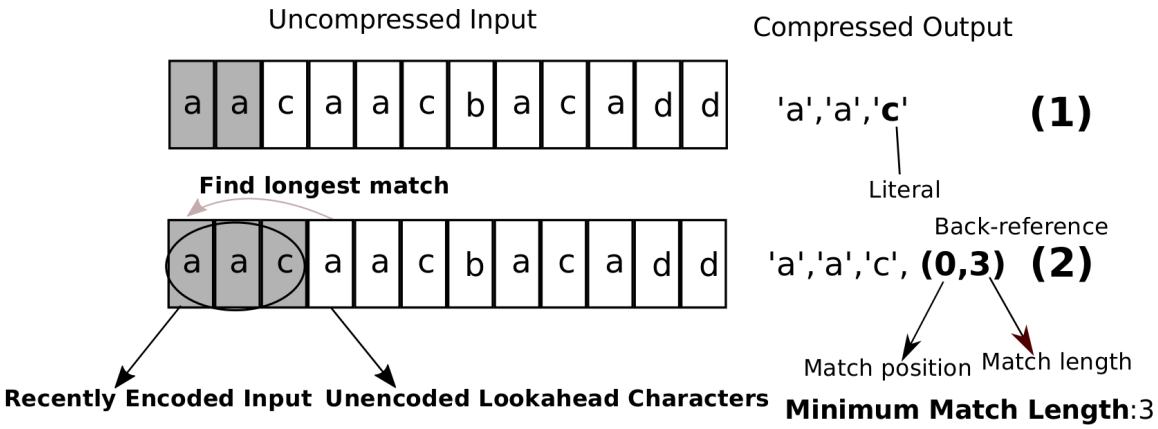


Figure 7.1: Illustration of LZ77 compression: (1) Literal is emitted because there was no match for 'c'. (2) Back-reference is emitted for a match on 'aac'.

7.2 Related Work

A common type of data compression replaces frequently-occurring sequences with references to earlier occurrences. This can be achieved by maintaining a dictionary of frequently-occurring patterns, such as in the LZ78 [Ziv and Lempel, 1978] and LZW [Welch, 1984] algorithms, or by maintaining a sliding window over the most recent data, as in the LZ77 algorithm [Ziv and Lempel, 1977]. A further space reduction can be achieved by encoding individual characters or dictionary symbols as variable-length code words. This so-called *entropy encoding* assigns code words with fewer bits to more frequently occurring symbols. Popular entropy encoding schemes are Huffman [Huffman, 1952] or Arithmetic [Rissanen, 1976] coding.

Compression schemes typically combine a dictionary-based technique and entropy coding. We study a variant of the popular DEFLATE [Deutsch, 1996] compression scheme, which is used in the gzip, ZIP, and PNG file formats. More precisely, we focus on the decompression process, called Inflate. DEFLATE uses the LZ77 dictionary scheme followed by Huffman coding. The dictionary in LZ77 is a sliding window over the recently processed input. The LZ77 compressor produces a stream of token symbols, in which each token is either a **back-reference** to a position in the sliding window dictionary, or a **literal** containing a sequence of characters, if that sequence does not appear in the sliding window.

A back-reference is represented as a tuple with the offset and the length of a match in the sliding window. Figure 7.1 illustrates both token types in a simple example. In the first step, the sequence 'aa' has already been processed and is in the sliding window dictionary. 'caac...' is the input to be processed next. Since the window does not contain any sequence starting with 'c', LZ77 emits a literal token 'c' and appends 'c' to the window. The sequence to be processed in the subsequent step is 'aacb...'. 'aac' is found in the window at position 0. The match is 3 characters long, hence, LZ77 emits back-reference token (0,3).

The resulting stream of literal and back-reference tokens are then converted into sequences of codewords by an entropy coder. DEFLATE uses Huffman coding, which yields code words with varying bit-lengths. We also consider entropy encoding that operates at the level of bytes rather than bits. This sacrifices some compression efficiency for speed. Existing dictionary-based compression schemes that use byte-level coding are LZRW1 [Williams, 1991], Snappy [Gunderson, 2015], and LZ4 [Collet, 2015a]. We refer to the implementation using bit-level encoding as **Gompresso/Bit**. Similarly, the entropy encoder in **Gompresso/Byte** operates at the byte level.

Although there are numerous compression schemes, we focus in this section on just the parallelization attempts of the best-known compression schemes.

Parallel CPU Implementations A parallel implementation for CPUs of gzip compression in the pigz library [Adler, 2015] achieves a linear speed-up of compression with the number of CPU cores. Decompression in pigz, however, has to be single-threaded because of its variable-length blocks. Another CPU compression library, pbzip [Gilchrist and Nikolov, 2015], parallelizes the set of algorithms implemented by the bzip2 scheme. The input is split into data blocks that can be compressed and decompressed in parallel. As already described in the Introduction, this inter-block parallelism alone is insufficient and results in poor performance on GPUs.

Hardware-Accelerated Implementations Parallelizing compression schemes within a block is a bigger challenge for massively-parallel processors. For example, the GPU implementation of bzip2 did not improve performance against the single-core CPU bzip2 [Patel *et al.*, 2012]. The major bottleneck was the string sort required for the Burrow-

Wheeler-Transform (BWT) compression layer. Future compressor implementations could be accelerated by replacing string sort with suffix array construction [Deo and Keely, 2013; Edwards and Vishkin, 2014; Wang *et al.*, 2015].

Most research has focused on accelerating compression, rather than decompression [Ozsoy and Swany, 2011]. Here, we address the thread dependencies that limit the parallelism of the LZ77 decompression. In our implementation each thread writes multiple back-reference characters at a time, avoiding the high per character cost. A parallel algorithm for LZ decompression, depending on the type of data dependencies, does not guarantee efficient GPU memory access[Agostino, 2000]. Huffman encoding is typically added to improve the compression ratio[Ozsoy *et al.*, 2014]. However, decoding is hard to parallelize because it has to identify codeword boundaries for variable-length coding schemes. Our parallel decoding method splits data blocks into smaller sub-blocks to increase the available parallelism. We trade-off a little of compression efficiency but only make only one pass over the encoded data. Alternative parallel decoding algorithms do not affect the compression ratio but they require multiple passes to decode the data for BWT decompression: A first pass to determine the codeword boundaries and a second for the actual decoding [Edwards and Vishkin, 2014].

Simpler compression schemes have been implemented on GPUs in the context of a database system [Fang *et al.*, 2010], but while these algorithms achieve good compression ratios for database columns, they are not efficient for Big Data workloads that might be unstructured. FPGAs and custom hardware have also been used to accelerate compression, resulting in high speed-ups [Xilinx, 2015; Abdelfattah *et al.*, 2014]. However, these hardware devices have very different characteristics and constraints than GPUs, so their parallelization techniques generally aren't applicable to GPUs.

7.3 Gompresso Overview

In this section, we provide an overview of **Gompresso**, which exploits parallelism between and also within data blocks. The most important design goal for **Gompresso** is a high decompression speed, while maintaining a "reasonable" compression ratio. **Gompresso**

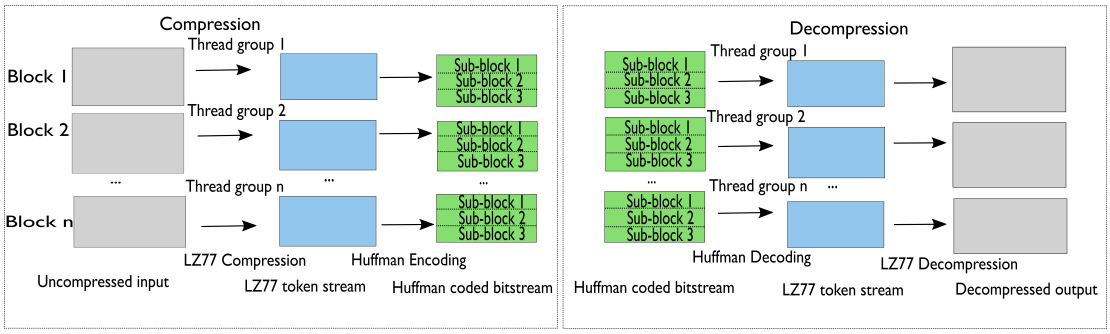


Figure 7.2: **Gompresso** splits the input into equally-sized blocks, which are then LZ77-compressed independently and in parallel. In **Gompresso/Bit**, the resulting token streams are further split into equal-sized sub-blocks and Huffman-encoded. The inverse process follows for decompression.

implements both compression and decompression, and defines its own file format. Figure 7.2 gives an overview of the **Gompresso** compression and decompression algorithms. We first briefly outline the parallel compression phase before describing parallel decompression, which is our focus.

7.3.1 Parallel Compression

In the first step, **Gompresso** splits the input into equally-sized data blocks, which are then compressed independently and in parallel. The block size is a configurable run-time parameter that is chosen depending on the total data size and the number of available processing elements on the GPU. Each block is LZ77-compressed by a group of threads using an exhaustive parallel matching technique we described earlier [Sitaridi *et al.*, 2013]. For **Gompresso/Byte**, the pipeline ends here, and the resulting token streams are written into the output file using a direct byte-level encoding. **Gompresso/Bit** requires an additional step in which the tokens are encoded using a Huffman coder. Similar to DEFLATE, **Gompresso/Bit** uses two separate Huffman trees to facilitate the encoding, one for the match offset values and the second for the length of the matches and the literals themselves. Both trees are created from the token frequencies for each block. To facilitate parallel decoding later on, the tokens of the data blocks are further split into smaller sub-blocks during encod-

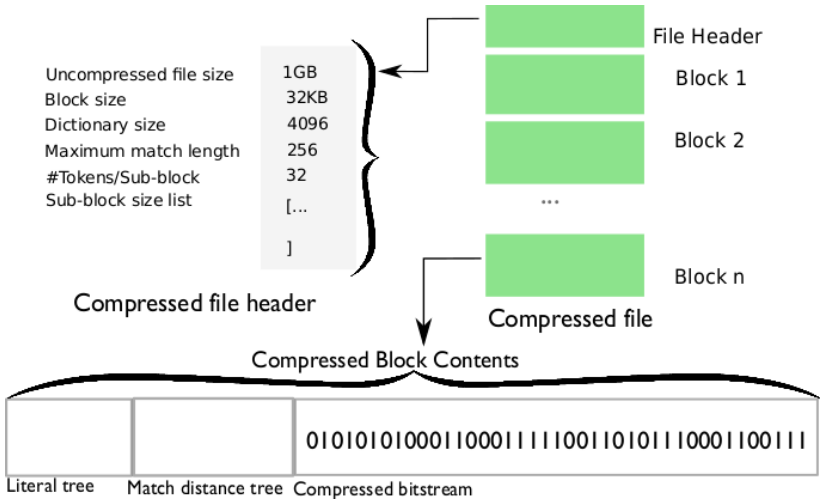


Figure 7.3: The **Gompresso** file format, consisting of: (1) a file header, (2) a sequence of compressed data blocks, each with its two Huffman trees (**Gompresso/Byte** does not use Huffman trees.) and encoded bitstream.

ing. A run-time parameter allows the user to set the number of sub-blocks per data block; more sub-blocks per block increases parallelism and hence performance, but diminishes sub-block size and hence compression ratio. Each encoded sub-block is written to the output file, along with its compressed size in bits. The parallel decoder can determine the location of the encoded sub-blocks in the compressed bitstream with this size information. Finally, the Huffman trees are written in a canonical representation [Huffman, 1952]. Figure 7.3 shows the structure of the compressed file format in detail.

7.3.2 Parallel Decompression

Gompresso/Byte can combine decoding and decompression in a single pass because of its fixed-length byte-level coding scheme. The token streams can be read directly from the compressed output. **Gompresso/Bit** uses a variable-length coding scheme for a higher compression ratio, and therefore needs to first decode the bitstream into a stream of tokens before proceeding with the LZ77 decompression. **Gompresso** assigns a group of GPU threads to collaborate on the Huffman decoding and LZ77 decompression on the independently compressed data blocks. This permits an additional degree of parallelism within

data blocks.

7.3.2.1 Huffman Decoding

Each thread of a group decodes a different sub-block of the compressed data block. The starting offset of each sub-block in the bitstream is computed from the cumulative sub-block sizes in the file header. All sub-blocks of a given data block decode their bitstreams using look-up tables created from the same two Huffman trees for that block and stored in the software-controlled, on-chip memories of the GPU. We can retrieve the original token symbol with a single lookup in each table, which is much faster than searching through the (more compact) Huffman trees, which would introduce branches and hence divergence of the threads' execution paths. The output of the decoder is the stream of literal and back-reference tokens, and is written back to the device memory.

7.3.2.2 LZ77 Decompression

Each data block is assigned to a single GPU warp (32 threads operating in lock-step) for decompression. We chose to limit the group size to one warp in order to be able to take advantage of the efficient voting and shuffling instructions within a warp. Larger thread groups would require explicit synchronization and data exchange via on-chip memory. We found that the potential performance gain by the increased degree of parallelism is canceled out by this additional coordination overhead.

We first group consecutive literals into a single literal string. We further require that a literal string is followed by a back-reference and vice versa, similar to the LZ4 [Collet, 2015a] compression scheme. A literal string may have zero length if there is no literal token between two consecutive back-references. A pair consisting of a literal string and a back-reference is called a *sequence*. We assign each sequence to a different thread (see Figure 7.4). In our experiments, we found that this grouping results in better decompression speed, since it not only assigns each thread a larger unit of work but its uniformity suits the lock-step execution model of the GPU. All threads in the warp concurrently alternate between executing instructions for string literals and for back references. For each sequence, its thread performs: (a) read its sequence from device memory and compute the start position

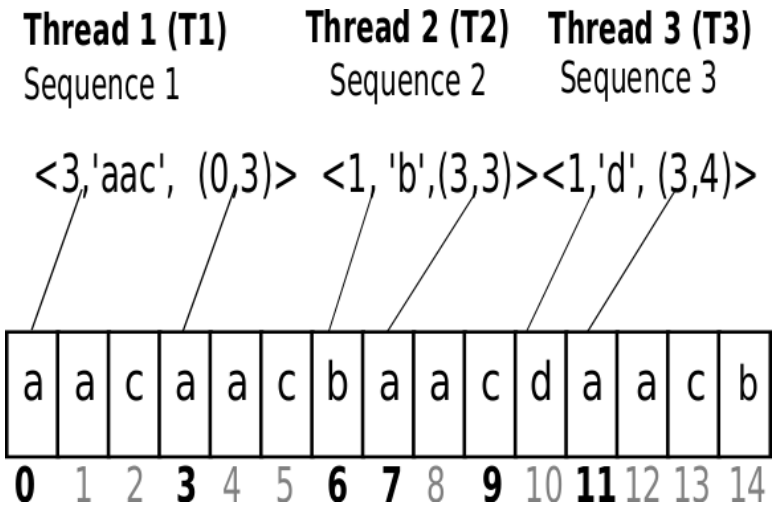


Figure 7.4: Decompression of 3 sequences by 3 threads. Numbers at the bottom show the positions in the uncompressed output, and those in bold indicate the start write positions of each token. For simplicity, we indicate each match position as a global offset, though **Gompresso** uses thread-relative distances.

of its string literal, (b) determine the output position of its literal, and copy its string literal to the output buffer, and (c) resolve and write its back-reference. We now describe each step in more detail:

Reading sequences Each warp uses the block offset to determine the location of the first decoded token in the device memory. Each thread in the warp will read a different sequence (see Figure 7.4). The threads then need to determine the start location of their literal strings in the token stream. This is accomplished by computing an intra-warp exclusive prefix sum from the literal lengths of their sequences, in order to locate the start positions from which they can copy their literal strings. We use NVIDIA's shuffle instructions to efficiently compute this prefix sum without memory accesses, a common GPU technique.

Copying literal strings Next, the threads compute write positions in the decompressed output buffer. Since all blocks, except potentially the last, have the same uncompressed size, the threads can also easily determine the start position of their block in the uncompressed output stream. The start position of each thread's literal string is determined by a second

exclusive prefix sum, which is then added to the start position of the block. This prefix sum is computed from the total number of bytes that each thread will write for its sequence, i.e., the length of its literal string plus the match length of the back-reference. Once the source and destination positions are determined from the two prefix sums, the threads can copy the literal strings from the token stream into the output buffer.

Copying back-references This is the most challenging step for parallel decompression, because of the data dependencies between threads in a warp. These dependencies arise when a back-reference points to another back-reference, and thus cannot be resolved before the former has been resolved. We address these nested back-references in Section 7.4. After all the back-references have been resolved, the warp continues with the next 32 sequences.

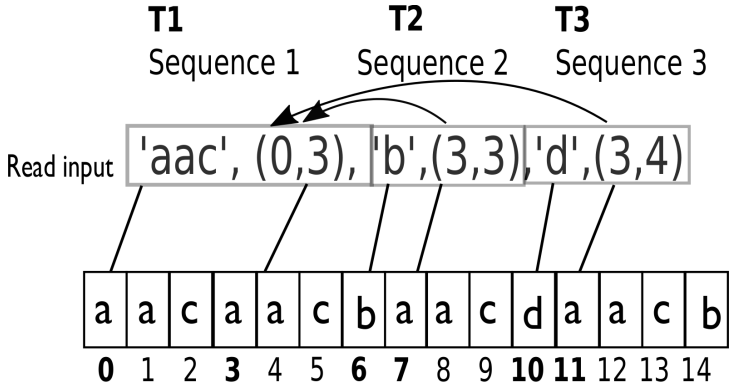


Figure 7.5: Nested back-references: back-references in Sequence 2 and 3 depend on Sequence 1, and cannot be resolved before the output of Sequence 1 is available.

7.4 Data Dependencies in Nested Back-references

Before processing a back-reference, the data pointed to by this reference needs to be available in the output. This introduces a data dependency and stalls threads with dependent references until the referenced data becomes available. The problem is illustrated in Figure 7.5. Threads T2 and T3 will have to wait for T1 to finish processing its sequence, because they both have back-references that point into the range that is written by T1. Resolving

back-references sequentially would produce the correct output, but would also under-utilize the available thread resources. To maximize thread utilization, we propose two strategies to handle these data dependencies. The first strategy uses warp shuffling and voting instructions to process dependencies as soon as possible, i.e., as soon as all of the referenced data becomes available. The second strategy avoids data dependencies altogether by prohibiting construction of nested back-references during compression. This second approach unfortunately reduces compression efficiency somewhat, which we will quantify experimentally in Section 7.5.

```
1:  function MRR(HWM, read_pos, write_pos, length)
2:      pending ← true                          ▷ thread has not written any output
3:      do
4:          if pending and read_pos+length≤HWM then
5:              copy length bytes from read_pos to write_pos
6:              pending ← false
7:          end if
8:          votes ← ballot(pending)
9:          last_writer ← count_leading_zero_bits(votes)
10:         HWM ← shfl(write_pos+length, last_writer)
11:     while votes> 0                          ▷ Repeat until all threads done
12:     return HWM
13: end function
```

Figure 7.6: Multi-Round Resolution (MRR) Algorithm.

7.4.1 MRR Strategy

Figure 7.6 shows the Multi-Round Resolution (MRR) algorithm for iterative resolution of nested back-references, which is executed by every thread in the warp. We follow the GPU programming convention in which each of the variables is *thread-private* unless it is explicitly marked as locally or globally shared. The Boolean variable *pending* is initially set on 2 and is cleared once the thread has copied its back-reference to the output (line 6).

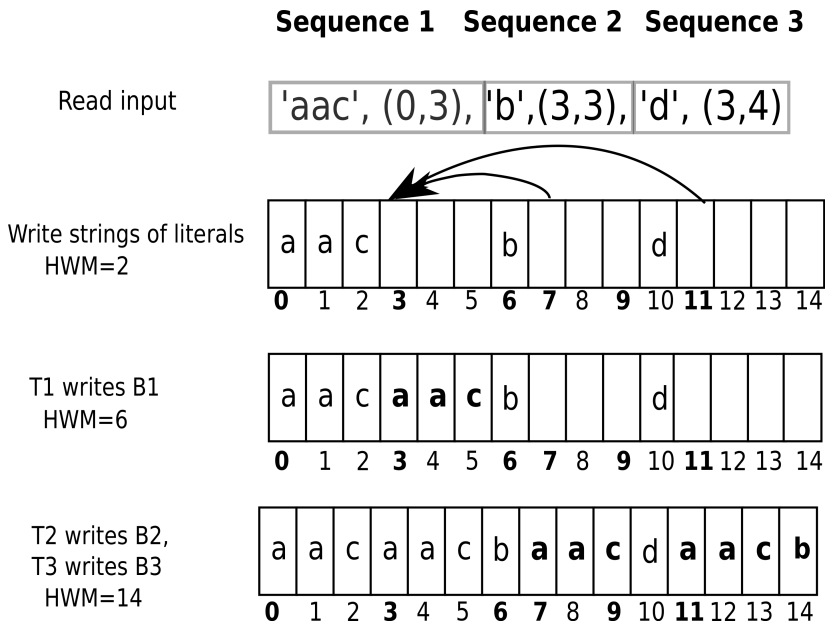


Figure 7.7: Multi-Round Resolution (MRR) execution.

Before calling MRR, all threads have written their literal string from their sequence to the output, but no thread in the warp has written a back-reference yet. In order to determine when the referenced data becomes available, the threads keep track of the high-water mark (HWM) position of the output that has been written so far without gaps. A back-reference whose referenced interval is below the HWM can therefore be resolved. In each iteration, threads that have not yet written their output use the high-water mark (HWM) to determine whether their back reference can be resolved (line 4). If so, they copy the data from the referenced sequence to the output, and indicate that they completed their work (lines 5 and 6).

The HWM is updated at the end of each iteration. The algorithm determines the last sequence that was completed by the warp, and sets the HWM past the highest write position of that sequence's back-reference. The threads can determine the last sequence without accessing shared memory by exploiting the warp-voting instruction `ballot` on the `pending` flag (line 8). This produces a 32-bit bitmap that contains the `pending` states of all threads in this warp. Each thread receives this bitmap and then counts the number of leading zeros in the bitmap in order to determine the ID of the `last_writer` thread that

completed the last sequence. A subsequent warp-shuffle instruction broadcasts the new HWM computed by the `last_writer` thread to all other threads in the warp (line 10). The iteration completes when all threads have processed their back-references.

Figure 7.7 illustrates the execution of MRR, for the set of 3 sequences from Figure 7.5. Initially, all threads write in parallel their string of literals. In the next step, T1 copies the back-reference of Sequence 1. In the last step, after Sequence 1 has been processed, the dependencies of T2 and T3 are satisfied, so both threads can proceed to copy their back-references.

At least one back-reference is resolved during each iteration which guarantees termination of the algorithm. The HWM increases strictly monotonically. The degree of achievable parallelism depends on nesting of back-references. As soon as the referenced ranges falls below the HWM they can be resolved simultaneously. Back-references that do not depend on data produced by other back-references from the same warp can be resolved in one round leading to maximum parallelism of the warp. In the worst-case scenario all but one back-reference depends on another back-reference in the same warp. MRR then leads to sequential execution. The next section describes a strategy that avoids this scenario.

7.4.2 DE Strategy

In this strategy, we trade off a little compression efficiency to avoid MRR's run-time cost of iterativelydetecting and resolving dependencies during decompression. During compression, we prohibit nested back-references that would create data dependencies within the same warp. This doesn't eliminate *all* nested back-references, only those that would depend on other back-references within the same warp. Prohibiting these same-warp back-references generally results in a slightly lower compression ratio and more effort during compression, due to the additional checking and bookkeeping. As we will show in Section 7.5, the degradation in compression ratio and compression speed is acceptable. In return, however, we get a 2–3× gain in decompression speed.

Dependency elimination works as follows: For every group of 32 sequences that will eventually be decompressed by the same warp of threads, we only look for dictionary matches below a certain warp high-water mark (warpHWM). By choosing the warpHWM to be the

```
1: pos ← 0
2: while pos<blocksize do
3:     warpHWM ← pos
4:     s ← 0
5:     literal_str ← ""
6:     while s < 32 do
7:         match ← find_match_below_hwm(dict, input, warpHWM)
8:         if match found then
9:             emit_sequence((literal_str, match))
10:            update_dictionary_with_backref(dict, match)
11:            pos ← pos + match.length
12:            s ← s + 1
13:            literal_str ← ""
14:        else
15:            b ← get next byte from input
16:            literal_str ← literal_str | b
17:            update_dictionary_with_literal_byte(dict, b)
18:            pos ← pos + 1
19:        end if
20:    end while
21: end while
```

Figure 7.8: Modified LZ77 compression algorithm with Dependency Elimination (DE).

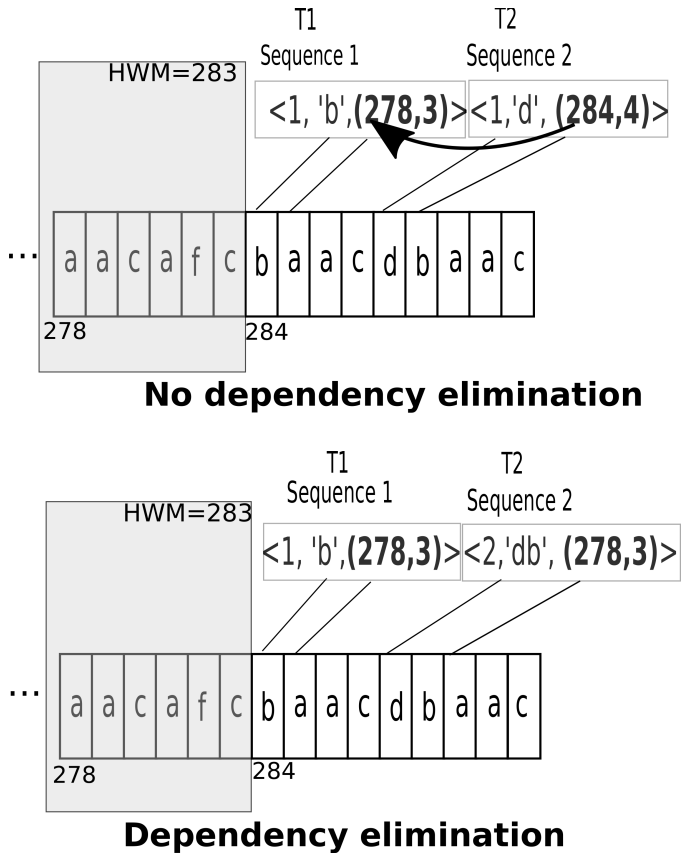


Figure 7.9: Resulting token stream without and with dependency elimination (DE).

cursor position in the input that has been completed previously by the warp, we avoid back-references that would otherwise lead to data dependencies. Figure 7.8 shows the modified LZ77 compression algorithm. The warpHWM is updated only after a group of 32 sequences have been completely processed (line 3). Threads that cooperate in the compression perform the string matching in parallel in `find_match_below_hwm` (line 7). They only look for a match below the current warpHWM. If no match is found, the next input byte is added to the literal string (line 16) and to the dictionary (line 17). Otherwise, if a match is found, the thread closes and emits the output sequence comprising the current literal string and the found match as a back-reference (line 9). Then the dictionary is updated with the found match. The variable "pos" keeps track of the cursor position in the processed input. Figure 7.9 illustrates the algorithm with an example. The dependency of T2 on T1 is avoided by choosing a shorter match in the back-references for Sequence 2.

Since our **Gompresso** work is focused on decompression, our implementation of the compressor is not as highly optimized as the most commonly used data compression libraries. We decided to implement the DE algorithm in the LZ4 compression library (CPU-only) [Collet, 2015a] in order to measure the impact that the dependency elimination has on compression speed and the resulting compression ratio. In addition to the DE algorithm itself, we also had to implement the logic for `find_match_below_hwm()` (line 7) by modifying the match-finding component in the LZ4 library so that it only returns matches below a certain HWM. To find matches, the compressor of the LZ4 library uses a hash table, a common choice for single-threaded implementations of LZ-based compression. The key in the hash table is a string of three bytes (trigram). The value is the most recent position in the input in which that trigram was encountered. This most recent position needs to be compared with the warpHWM. We modified the existing hash replacement policy to replace an occurrence with a more recent one only if the original entry is at more than some number of bytes behind the current byte position. We use a constant value for this "minimal staleness", which we determined experimentally. By testing different values ranging from 64–8 K on different datasets, we determined that 1 K results in the lowest compression ratio degradation.

7.5 Experimental Evaluation

7.5.1 Experimental Setup

We evaluate **Gompresso** using two different datasets. The first is a 1 GB XML dump of the English Wikipedia. The second dataset is the "Hollywood-2009" sparse matrix from the *University of Florida Sparse Matrix Collection*, stored as a 0.77 GB Matrix Market file format. Both sets are highly compressible. For comparison, the gzip tool achieves a compression ratio of 3.09:1 for the former and 4.99:1 for the latter, using the default compression level setting (–6). The performance measurements are conducted on a dual-socket system with two Intel E5-2620 v2 CPUs, 2×6 cores running 24 hardware threads. We add an NVIDIA Tesla K40 with 2,880 CUDA cores to the system for the GPU measurements. The device is connected via a PCI Express (PCIe) 3.0 x16 link with a nominal bandwidth of

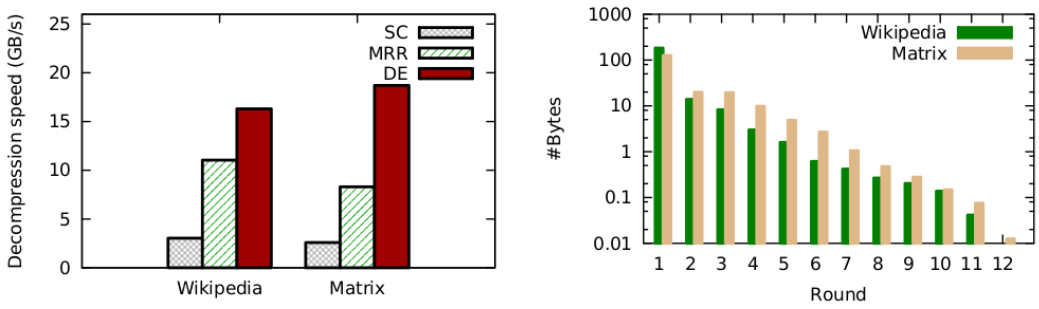


Figure 7.10: (a) Decompression speed of **Gompresso/Byte** (data transfer cost not included), using different dependency resolution strategies for the two datasets. (b) Number of bytes processed on each round of MRR.

16 GB/sec in each direction. We report bandwidth numbers that include PCIe transfers. In cases in which the PCIe bandwidth becomes the bottleneck, we report the bandwidth with input and output data residing in the GPU's device memory. ECC is turned on in our measurements. We determine the decompression bandwidth as the ratio of the size of the *uncompressed data* over the total processing time. Unless otherwise noted, we are using a data block size of 256 KB and a sliding window of 8 KB. For compression, we look at the next 64 bytes in the input for each match search in the 8 KB window. To facilitate parallel Huffman decoding in **Gompresso/Bit**, we split the sequence stream into sub-blocks that are 16 sequences long.

7.5.2 Data Dependency Resolution

7.5.3 Performance Impact of Nested back-references

We first focus on just the LZ decompression throughput of **Gompresso/Byte**, i.e., with no entropy decoding, for different resolution strategies in Figure 7.10. *Sequential Copying (SC)* is our baseline, in which threads copy their back-references in a sequential order without intra-block parallelism. The figure shows that Dependency Elimination (DE) is the fastest strategy for decompression. It is at least 5× faster than SC. We place the compressed input and the decompressed output in device memory in this setup, and ignore PCIe transfers. The figure shows that the decompression throughput is higher than the theoretical maximal

Figure 7.11: Series of sequences inducing 32 and 16 rounds of resolution.

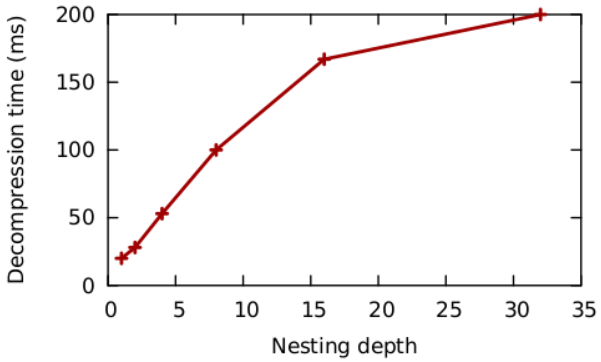


Figure 7.12: Decompression speed of MRR as a function of the number of resolution rounds, for an artificially generated dataset.

bandwidth of the PCIe link. As expected, Multi-Round Resolution (MRR) performs better than SC due to the higher degree of parallelism, while DE out-performs MRR because it achieves an even higher degree of parallelism.

Figure 7.10 shows the average number of bytes that are resolved from back-references in each round. For example, for round 2, we sum the number of bytes copied by the active threads in the second round divided by the number of MRR iterations executed for a dataset. The lower performance of MRR was surprising, given that we observed relatively few bytes processed after the first round. However, what limits performance is the number of rounds. For the Wikipedia dataset, the average number of resolution rounds is around 3, and for the Matrix dataset, 4.

To better understand the performance impact of multiple passes, we created a collection of artificial 1 GB datasets that induce a specified depth of back-reference nesting. We generate each dataset such that it leads to the desired depth. The general idea is as follows:

we repeat a 16-byte string with a one-byte change occurring in an alternating fashion at the first and last byte position. We chose the length of 16 to be close to the average back-reference match length in the two real datasets used in our evaluation. A separator byte, chosen from a disjoint set of bytes, is used to prevent accidental and undesired matches that cross different instances of the repeating string. Figure 7.11 illustrates how sequences of nested back-references are created. We show two small examples for strings of length four, rather than 16 bytes, for space reasons. The separator bytes are printed in black, while the repeating string is shown in green and orange colors. The arrows show the dependencies in the MRR algorithm. LZ decompression of the dataset shown on top in Figure 7.11 will incur data dependencies of all 32 threads in the warp except the first, whose dependency does not cause a stall because it points to the data that was processed by this warp previously. The nesting depth in a warp is 32, so completing the resolution requires 32 rounds. In order to generate datasets with a smaller nesting depth, we alternate multiple distinct repeated strings. For example, two repeated strings result in depth 16, four repeated strings in depth 8, and so on. For a depth of 16, in each round, two back-references are copied, one for each repeated string. These two strings are marked in green and orange in the lower example in Figure 7.11. Figure 7.12 shows the decompression time for different nesting depths. The decompression time increases sharply until about 16 rounds. The primary reason for the slower performance of MRR is that all threads in a warp have to wait until the entire warp's back-references have been resolved. Threads that resolve on the first round will be underutilized while other threads do work in subsequent passes.

We also implemented an alternative variant of MRR that wrote nested back-references to device memory during each round. Each round is performed in a separate kernel. Later passes read unresolved back-references and all threads in a warp can be doing useful work. Because of the overhead of writing to and reading from memory, together with the increased complexity of tracking when a dependency can be resolved, the alternative variant did not improve the performance of MRR.

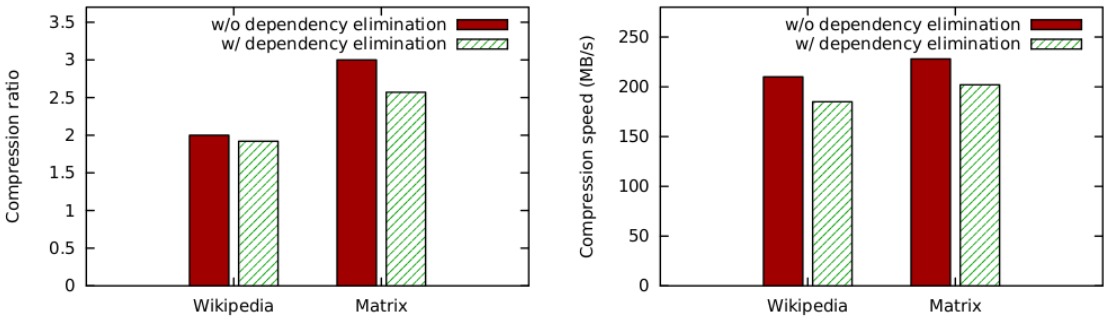


Figure 7.13: Degradation in compression efficiency and speed for DE method.

7.5.4 Impact of DE on Compression Ratio and Speed

Figure 7.13 shows the degradation in compression ratio and compression speed when eliminating dependencies using the Dependency Elimination (DE) algorithm we implemented by modifying the LZ4 library. The maximum degradation is 13 % in compression speed and 19 % in compression ratio, which is acceptable since we are aiming at fast decompression. In the remaining experiments, we use the DE method for decompression.

7.5.5 Compression Framework Tuning

Figure 7.13 shows the degradation in compression ratio and compression speed when eliminating dependencies using the Dependency Elimination (DE) algorithm we implemented by modifying the LZ4 library. The maximum degradation is 13 % in compression speed and 19 % in compression ratio, which is acceptable since we are aiming at fast decompression. In the remaining experiments, we use the DE method for decompression.

7.5.6 Dependency on Data Block Size

Figure 7.14 shows the decompression speed and compression ratio for different data block sizes. Larger blocks increase the available parallelism for Huffman decoding because there are more parallel sub-blocks in flight. Threads operating on sub-blocks that belong to the same data block share the Huffman decoding tables, which are stored in the software-controlled, on-chip memory of the GPU. This intra-block parallelism leads to a better utilization of the GPU's compute resources by scheduling more data blocks on the GPU's

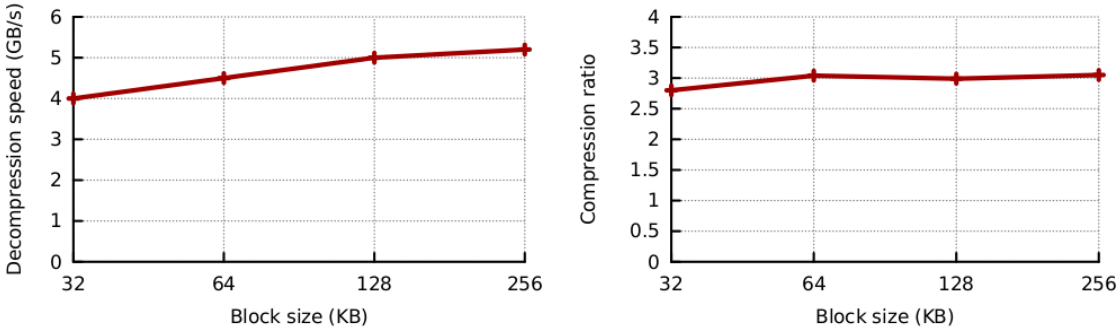


Figure 7.14: Decompression speed (data transfer cost included) and ratio of **Gompresso/Bit** for different block sizes.

processors for concurrent execution (inter-block parallelism). The space required by the Huffman decoding tables in the processors' on-chip memory limit the number of data blocks that can be decoded concurrently on a single GPU processor.

Each Huffman decoding table has 2^{CWL} entries, where CWL is the maximum codeword length. To fit the look-up tables in the on-chip memory, we are using limited-length Huffman encoding with a maximum length of CWL = 10 bits. Figure 7.14 shows that the compression ratio only marginally degrades for smaller blocks, so the space overhead of storing the block header for each compressed data block is not significant.

7.5.7 GPU vs. Multi-core CPU Performance

Lastly, we compare the performance of **Gompresso** to state-of-the-art parallel CPU libraries regarding decompression speed and overall energy consumption. We used a power meter to measure energy consumption at the wall socket. For CPU-only environments, we physically removed the GPUs from our server to avoid including the GPU's idle power. We parallelize the single-threaded implementations of the CPU-based state-of-the-art compression libraries by splitting the input data into equally-sized blocks that are then processed by the different cores in parallel. We chose a block size of 2 MB, as this size resulted in the highest decompression speeds for the parallelized libraries. Once a thread has completed decompressing a data block, it immediately processes the next block from a common queue. This balances the load across CPU threads despite input-dependent processing times for

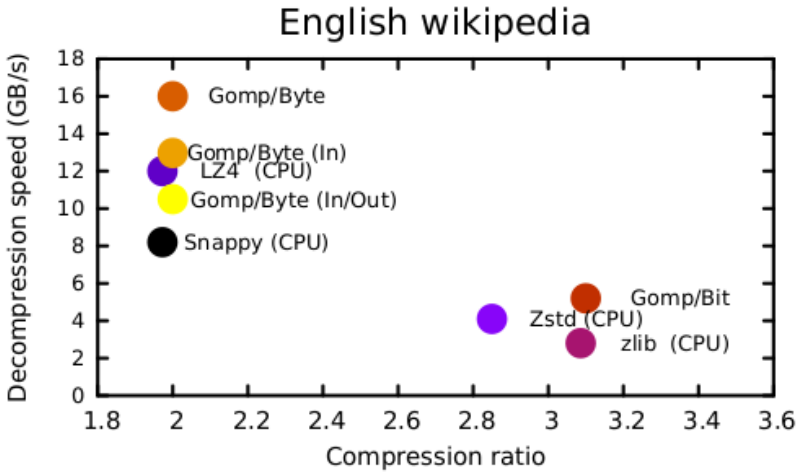


Figure 7.15: GPU vs multicore CPU performance. Cost for transferring data to and from the GPU is included for **Gompresso/Bit**. For **Gompresso/Byte**, we show the performance both including and not including data transfers.

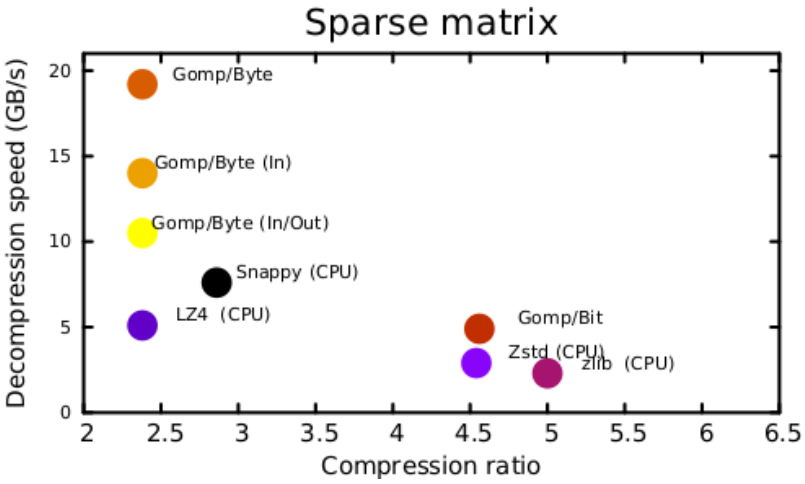


Figure 7.16: GPU vs multicore CPU performance. Cost for transferring data to and from the GPU is included for **Gompresso/Bit**. For **Gompresso/Byte**, we show the performance both including and not including data transfers.

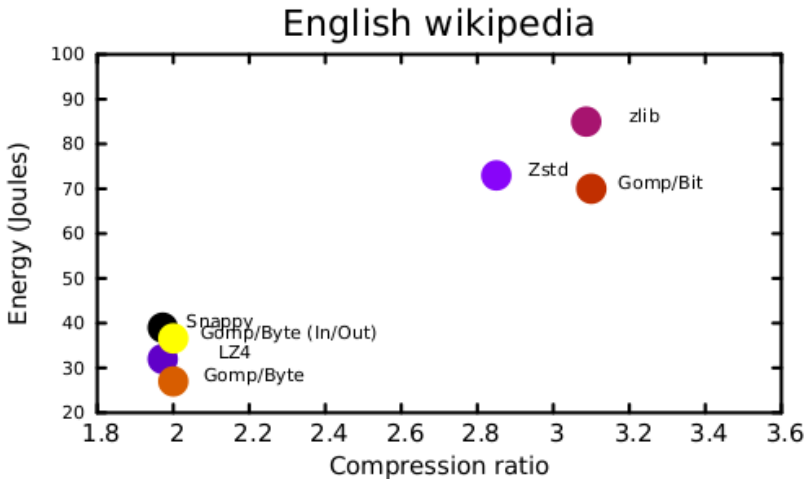


Figure 7.17: GPU vs. multicore CPU energy consumption.

the different data blocks.

Figure 7.15 shows the trade-offs between decompression speed and compression ratio. In addition to the measurements of our **Gompresso** system, we include the performance of two byte-level compression libraries (LZ4, Snappy) and for two libraries using bit-level encoding (gzip, zlib) for comparison. Zstd implements a different coding algorithm on top of LZ-compression that is typically faster than Huffman decoding, and we include it in our measurements for completeness [Collet, 2015b]. zlib implements the DEFLATE scheme for the CPU. For the GPU measurements, we show the end-to-end performance, including times for: (a) both compressed input and uncompressed output over PCIe, marked *(In/Out)* in Figure 7.15; (b) only the input transfers, marked as *(In)*; and (c) ignoring data transfers altogether, marked as *No PCIe*.

For **Gompresso/Byte**, PCIe transfers turned out to be the bottleneck. In separate bandwidth tests, we were able to achieve a PCIe peak bandwidth of 13 GB/sec. **Gompresso/Bit**, though not PCIe-bound, is still 2× faster than zlib and **Gompresso/Byte** is 1.35× faster than LZ4. For the matrix dataset, the decompression speed of **Gompresso/Bit** is around 2× faster than zlib. There is around 9 % degradation in compression ratio because we use limited-length Huffman coding. Although it lowers the compression efficiency, it enables us to fit more Huffman decoding tables into the on-chip memory.

Finally, we compare the energy consumed to decompress the Wikipedia dataset. In general, faster decompression on the same hardware platform results in improved energy efficiency. This is because the power drawn at the system level, i.e., at the wall plug, does not differ significantly for different algorithms. More interesting is the energy efficiency when comparing different implementations on different hardware platforms, e.g., a parallel CPU vs. a GPU solution. Figure 7.17 shows the overall energy consumption versus the compression ratio for **Gompresso** and a number of parallelized CPU-based libraries. **Gompresso/Bit** consumes around 1.2× less energy than the parallel zlib library. It also has similar energy consumption to Zstd, which implements a faster coding algorithm.

7.6 Summary & Conclusions

The exponentially-increasing data volumes of the Big Data era make data compression essential to control storage costs. The time to analyze such large data volumes is largely dependent upon how fast it can be accessed and decompressed. As processor speeds plateau, the best hope for significantly speeding up these operations is massively parallelizing them. Unfortunately, straightforward "divide and conquer" techniques that assign each data block to one thread don't sufficiently exploit massively-parallel computing platforms such as GPUs, so more sophisticated algorithms that achieve parallelism *within* blocks of data are required.

We developed techniques for massively parallelizing decompression using GPUs, implemented those techniques in two variants of the **Gompresso** system, and evaluated their performance in terms of decompression speed, the resulting compression ratio, and the energy consumed in the process. Decompression involves two operations that are inherently hard to parallelize: Huffman decoding and resolution of back-references in the typical LZ-style decompression. We presented one solution to parallelizing Huffman decoding by using parallel sub-blocks, and two techniques to resolve back-references in parallel. The first technique, called Multi-Round Resolution (MRR), exploits fast data-exchange primitives on GPUs in an algorithm that iteratively resolves back-references. The second, called Dependency Elimination (DE), is a simple technique during compression to limit back-references within a set of sub-blocks, thereby eliminating data dependencies that will stall parallelism

among collaborating threads that are concurrently decompressing that set of sub-blocks.

We implemented the above techniques in two variants of decompression for GPUs, called **Gompresso/Bit** and **Gompresso/Byte**, where **Gompresso/Bit** adds parallel Huffman decoding to LZ-style decompression. **Gompresso**, running on an NVIDIA Tesla K40, decompressed two real-world datasets 2× faster than the state-of-the-art block-parallel variant of zlib running on a modern multi-core CPU, while suffering no more than a 10 % penalty in compression ratio. **Gompresso** also uses 17 % less energy by using GPUs, when evaluated against state-of-the-art parallel CPU libraries for decompression.

Future work includes determining the extent to which our parallelization techniques can be applied to alternative coding and context-based compression schemes, and evaluating their performance. While our performance improvements over CPU-based algorithms are impressive, we continue to study innovative ways to significantly increase both the speed and efficiency of decompression using massively-parallel hardware.

Chapter 8

Concluding Remarks and Future Work

In the present thesis, we proposed a set of techniques addressing fundamental limitations of in-memory analytics operators on GPUs. Our techniques extend to different steps of query processing: Data preprocessing, data storage layout, query plan compilation, and algorithmic optimization of data processing operators. Our novel compression framework can be used to fit more data in the GPU memory or as a stand-alone tool. Our techniques are based on the SIMT architecture of GPUs and their memory hierarchy, designed for high throughput rather than low latency. We focus on fundamental features of GPUs so we project that our techniques will be useful for future GPU generations. The next sections suggest interesting directions for future work on GPU data analytics.

8.1 GPU Query Execution Optimization

We suggested an execution optimization algorithm for conjunctive selections on GPUs. Our solution uses an accurate analytical model to compare the performance of alternative conjunctive scan selection plans based on the predicted memory performance involving the number of memory accesses, the memory locality and the latency hiding factor of independent memory requests from the GPU hardware. Our model abstracts the characteristics of the GPU memory hierarchy that impact the selection performance. Further work would

model the performance of more complex queries on the GPU involving joins, aggregations, and disjunctions. The ordering of the operators would be decided by the query optimizer, as in traditional database systems. An underlying execution optimizer would determine the optimal query execution plan using the cardinality estimation component and our calibrated model of the GPU memory characteristics.

Query execution also is responsible for translating queries from SQL to low-level code. A first approach translates SQL queries to source code languages such as C++. However, LLVM query compilation is more efficient [Suhan and Mostak, 2015]. LLVM query compilation translates SQL queries to intermediate architecture independent code and has the additional advantage of portability across different architectures, CPUs or different GPU processor types. However, even if LLVM compilation is relatively more efficient, it might pose an overhead for queries with low latency requirements, such as in real-time analytics applications. Similar queries or queries with a similar structure and different constant parameters might be reoccurring so caching queries can reduce the query compilation cost.

8.2 String Matching Acceleration

We suggested software techniques reducing thread divergence of string matching and reorganize the string layout to optimize memory accesses for the GPU memory hierarchy. String data is amenable to compression enabling larger datasets to fit in the GPU and in some cases compressing strings also improves pattern matching performance. Decompressing text before performing string matching is the straightforward approach to compressed pattern matching. More sophisticated algorithms for compressed pattern matching, search strings either by skipping the decompression step or by only decompressing segments of the input strings. Compressed pattern matching algorithms have been designed for different types of text compression: such as Huffman Coding [Klein and Shapira, 2005] and LZ-compression[Farach and Thorup, 1995; Gawrychowski, 2011]. Accelerating string matching on different forms of compressed would not only evaluate the raw performance of algorithms for different formats but also would produce a time-space timeline of the evaluated algorithms. Compressed pattern matching would revisit the optimal layout for strings in the

GPU device memory since compressed text would be bit-aligned rather than byte-aligned.

8.3 Massively Parallel Lossless Compression

Our work focused on accelerating the Deflate decompressor. We identify critical opportunities for hardware-accelerated data compression. One important direction for future work is exploring alternative coding schemes and evaluate whether they change the performance skyline of compression frameworks. We showed how to parallelize Huffman decoding by spitting data blocks into smaller sub-blocks, that can be decoded in parallel by different parallel processors. This parallelization technique could be applied to other variable length coding algorithms. The main alternative coding approaches involve Arithmetic Coding [Rissanen, 1976] and Finite State Encoding [Duda, 2013].

Another research opportunity lies in exploring the efficiency of application-specific compression methods on massively parallel processors. Application-specific compression schemes become increasingly popular for large-scale data systems because of the higher resulting compression efficiency[Jun *et al.*, 2012]. The question remaining is how context-aware techniques can be adapted for GPUs.

Finally, we evaluated Gompresso in the context of a GPU. Gompresso, however, is generally designed for massively parallel architectures so it would be worth evaluating the use of Xeon Phi for decompression acceleration. Xeon Phi has many simpler cores, as GPUs. In Section 1.2 we discussed the similarities of GPUs to vector processors. However, the internals of GPUs and Xeon Phi are significantly different leaving to future work to determine whether our techniques would be efficient on Xeon Phi.

8.4 Heterogeneous Computing Data Analytics

Our techniques result in significant speed-ups over the CPU counterparts. However, in some cases, CPUs are still a viable alternative because of their better Performance/$ ratio. Based on this observation, in the next years, we expect database systems to be designed for heterogeneous processing. This trend is supported by the increase of the GPU memory capacity and the faster interconnection between the CPUs and the GPUs.

Designing heterogeneous systems that utilize efficiently all available processors require cost models to compare the absolute performance of queries on the available processors. In Chapter 4, we discussed the accuracy of analytical models predicting the relative performance of selection plans. Learning-based approaches have been suggested as an alternative solution to traditional computational cost models, which are more computationally expensive but can seamlessly adapt to new hardware. Learning models can also select the optimal operator variants in a hardware-oblivious way [Rosenfeld *et al.*, 2015] for a subset of operators. A natural way to extend this line of work is incorporating into the execution optimization framework more database operators, as for example joins. Our conjunctive selection optimization presents a middle ground between hardware-oblivious approaches and hardware-aware approaches: We suggest running a calibration step on each available device capturing critical features of the processor, such as the potential to overlap memory requests, the performance of atomic updates and the random access memory performance. Running this calibration step does not require prior knowledge of the device features and does not pose an overhead, as it only has to be executed on a small sample of data. Our conjunctive selection strategy could be extended for more general and complex queries. Additionally, it could incorporate into the cost model alternative methods to write the intermediate results in the GPU global memory, such as bitmaps [Rosenfeld *et al.*, 2015]. After extending our approach for general queries, the next step is comparing hardware-oblivious optimization approaches to an approach as ours that captures the underlying hardware characteristics for decision support benchmarks, such as TPC-H [Transaction Processing Performance Council, 2014].

At a practical level, the cost of increased code complexity must be quantified for heterogeneous systems. Integration of multiple processors means more code paths, increasing the cost of code maintenance. The alternative is having a single code-path for all processors and using the driver of each processor to translate it to low-level code. For example, OpenCL code can be executed, basically unmodified on CPUs and GPUs. It would be essential for future research to further evaluate how much performance is left on the table when using a single code-path.

Minimizing idle power is also another objective of CPU/GPU co-processing. Adding

one or multiple or GPUs to a system means more power wasted if these processors remain idle. This effect would diminish one of the advantages of using accelerators, which is reduced energy consumption. We could increase the processor utilization by monitoring the resource usage of the available processors and using this information for query scheduling, ultimately improving overall performance. Learning models could be used to schedule queries to processors during query runtime based on the operator execution time, processor load and data transfer cost between the main processor and the processors [Breß *et al.*, 2012]. Efficient resource usage monitoring is important for multi-query processing since processing only a single query at a time might underutilize processor resources. To evaluate alternative scheduling methods, we have to rethink the nature of the cost models for heterogeneous query processing. This reevaluation would involve adapting for GPUs computational models suggested for CPUs [Wu *et al.*, 2013; Li *et al.*, 2014] and comparing them to learning models focusing on hybrid CPU/GPU processing [Breß *et al.*, 2013].

Bibliography

[Abadi *et al.*, 2007] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, pages 466–475, 2007.

[Abdelfattah *et al.*, 2014] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL. In *IWOCL*, pages 4:1–4:9, 2014.

[Adler, 2015] Mark Adler. Parallel gzip. `http://zlib.net/pigz/`, 2015. Accessed: 2015-04-02.

[Agostino, 2000] S. De Agostino. Speeding up parallel decoding of LZ compressed text on the PRAM EREW. In *SPIRE*, pages 2–7, 2000.

[Aho and Corasick, 1975] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[Amazon Web Services, 2016] Amazon Web Services. High performance computing. `http://aws.amazon.com/hpc/`, 2016.

[AMD, 2013] AMD. OpenCL programming guide. `http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf`, 2013.

[AMD, 2016] AMD. AMD A-Series desktop APUs. `http://www.amd.com/en-us/products/processors/desktop/a-series-apu`, 2016.

[Apostolico and Giancarlo, 1986] Alberto Apostolico and Raffaele Giancarlo. The Boyer Moore Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.

[Bakkum and Chakradhar, 2012] Petter Bakkum and S Chakradhar. Efficient data management for GPU databases, 2012.

[Bakkum and Skadron, 2010] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*, pages 94–103, 2010.

[Bellekens *et al.*, 2013] X Bellekens, I Andonovic, Rc Atkinson, C Renfrew, and T Kirkham. Investigation of GPU-based pattern matching, 2013.

[Bhargava and Kondrak, 2009] Aditya Bhargava and Grzegorz Kondrak. Multiple word alignment with Profile Hidden Markov Models. In *ACL SRW*, pages 43–48, 2009.

[Boost, 2014] Boost. C++ libraries. `http://www.boost.org/`, 2014.

[Boyer and Moore, 1977] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[Breß *et al.*, 2012] Sebastian Breß, Felix Beier, Hannes Rauhe, Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. *Automatic Selection of Processing Units for Coprocessing in Databases*, pages 57–70. 2012.

[Breß *et al.*, 2013] Sebastian Breß, Eike Schallehn, and Ingolf Geist. *Towards Optimization of Hybrid CPU/GPU Query Plans in Database Systems*, pages 27–35. 2013.

[Breß *et al.*, 2014] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-accelerated database systems: Survey and open challenges. *T. Large-Scale Data- and Knowledge-Centered Systems*, 15:1–35, 2014.

[Carrillo *et al.*, 2009] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for GPGPU. In *ACM conference on Computing frontiers*, page 147150, 2009.

[Cascarano *et al.*, 2010] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. iNFAnt: NFA pattern matching on GPGPU devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, 2010.

[Cieslewicz *et al.*, 2007] John Cieslewicz, Kenneth A. Ross, and Ioannis Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, pages 2:1–2:10, 2007.

[Cieslewicz *et al.*, 2010] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD*, pages 483–494, 2010.

[Cohen-Crompton, 2012] Jen Cohen-Crompton. Real-time data analytics: On demand vs. continuous, 2012.

[Collet, 2015a] Yann Collet. LZ4–extremely fast compression. `https://github.com/Cyan4973/lz4`, 2015. Accessed: 2015-04-02.

[Collet, 2015b] Yann Collet. Zstandard–fast and efficient compression algorithm. `https://github.com/Cyan4973/zstd`, 2015. Accessed: 2015-04-02.

[Commentz-Walter, 1979] Beate Commentz-Walter. A string matching algorithm fast on the average. In *ICALP*, pages 118–132, 1979.

[Copeland and Khoshafian, 1985] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[Crochemore and Lecroq, 1996] Maxime Crochemore and Thierry Lecroq. Pattern-matching and text-compression algorithms. *ACM Comput. Surv.*, 28(1):39–41, 1996.

[DBPedia, 2014] DBPedia. DBpedia 2014 datasets. `http://wiki.dbpedia.org/Downloads2014`, 2014.

[Deo and Keely, 2013] Mrinal Deo and Sean Keely. Parallel suffix array and least common prefix for the GPU. In *PPoPP*, pages 197–206, 2013.

[Deutsch, 1996] Peter Deutsch. DEFLATE compressed data format specification version 1.3. RFC 1951 (Informational), may 1996.

[Diamos *et al.*, 2011] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD re-convergence at thread frontiers. In *MICRO*, page 477488, 2011.

[Diamos *et al.*, 2013] Gregory Diamos, Haicheng Wu, Jin Wang, Ashwin Lele, and Sudhakar Yalamanchili. Relational algorithms for multi-bulk-synchronous processors. *SIGPLAN Not.*, 48(8), 2013.

[Duda, 2013] Jarek Duda. Asymmetric numeral systems as close to capacity low state entropy coders. *CoRR*, abs/1311.2540, 2013.

[Edwards and Vishkin, 2014] James A. Edwards and Uzi Vishkin. Parallel algorithms for Burrows–Wheeler compression and decompression. *TCS*, 525:10 – 22, 2014.

[Eppstein, 1996] David Eppstein. Design and analysis of algorithms lecture notes., 1996.

[Erlingsson *et al.*, 2006] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. A cool and practical alternative to traditonal hash tables. In *Workshop on Distributed Data and Structures*, 2006.

[Esmaeilzadeh *et al.*, 2013] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.

[Fang *et al.*, 2007] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. GPUQP: Query co-processing using graphics processors. In *SIGMOD*, pages 1061–1063, 2007.

[Fang *et al.*, 2010] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, September 2010.

[Farach and Thorup, 1995] Martin Farach and Mikkel Thorup. String matching in lempel-ziv compressed strings. In *STOC*, pages 703–712, 1995.

[Farivar *et al.*, 2012] R. Farivar, H. Kharbanda, S. Venkataraman, and R.H. Campbell. An algorithm for fast edit distance computation on GPUs. In *Innovative Parallel Computing*, pages 1–9, May 2012.

[Ferragina and Manzini, 2005] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[Fisk and Varghese, 2004] Mike Fisk and George Varghese. Technical report, 2004.

[Fung *et al.*, 2007] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*, pages 407–420, 2007.

[Ganelin *et al.*, 2016] Ilya Ganelin, Ema Orhian, Kai Sasak, and Brennon York. *Spark: Big Data Cluster Computing in Production*. Wiley, 2016.

[Gawrychowski, 2011] PawełGawrychowski. Pattern matching in lempel-ziv compressed strings: Fast, simple, and deterministic. In *Proceedings of the 19th European Conference on Algorithms*, pages 421–432, 2011.

[Gilchrist and Nikolov, 2015] Jeff Gilchrist and Yavor Nikolov. Parallel bzip2. `http://compression.ca/pbzip2/`, 2015. Accessed: 2015-04-02.

[Glaskowsky, 2009] Peter N. Glaskowsky. NVIDIAs fermi: The first complete GPU computing architecture. Technical report, 2009.

[Govindaraju *et al.*, 2004] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, pages 215–226, 2004.

[GPUDB, 2016] GPUDB. A distributed database for manycore devices. `http://www.gpudb.com/`, 2016. Accessed: 2016-04-26.

[Graefe and Kuno, 2010] Goetz Graefe and Harumi Kuno. Tlkds. chapter Fast Loads and Queries, pages 31–72. 2010.

[Gunderson, 2015] Steinar H. Gunderson. Snappy a fast compressor/decompressor. `https://github.com/google/snappy`, 2015. Accessed: 2015-04-01.

[Han and Abdelrahman, 2011] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. In *GPGPU*, pages 3:1–3:8, 2011.

[He *et al.*, 2007] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *SC*, pages 46:1–46:12, 2007.

[He *et al.*, 2008a] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce framework on graphics processors. In *PACT*, pages 260–269, 2008.

[He *et al.*, 2008b] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[He *et al.*, 2009] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *TODS*, 34(4):21:1–21:39, 2009.

[Heimel *et al.*, 2013] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *VLDB*, 6(9):709–720, 2013.

[Hellerstein, 1998] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. *TODS*, 23(2):113–157, 1998.

[Hofmann and others, 2014] Johannes Hofmann et al. Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips. *CoRR*, arXiv:1401.7494, 2014.

[Holland and Gibson, 1992] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. *SIGPLAN Not.*, 27(9):23–35, 1992.

[Hopcroft, 1971] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, 1971.

[Horspool, 1980] R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[Huffman, 1952] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E.*, pages 1098–1102, Sep 1952.

[Intel, 2011] Intel. Intel 64 and IA-32 architectures software developer's manual, 2011.

[Iorio and Lunteren, 2008] Francesco Iorio and Jan Van Lunteren. Fast pattern matching on the Cell broadband engine. In *ISCA*, 2008.

[Jacob and Brodley, 2006] N. Jacob and C. Brodley. Offloading IDS computation to the GPU. In *ACSAC*, pages 371–380, 2006.

[Jun *et al.*, 2012] S. W. Jun, K. E. Fleming, M. Adler, and J. Emer. ZIP-IO: Architecture for application-specific compression of Big Data. In *FPT*, pages 343–351, 2012.

[Kaldewey and Di Blas, 2011] Tim Kaldewey and Andrea Di Blas. Large scale GPU search. In *GPU Computing Gems - Jade Edition*. Morgan Kaufmann, Waltham, MA, 2011.

[Kaldewey *et al.*, 2009] Tim Kaldewey, Jeff Hagen, Andrea Di Blas, and Eric Sedlar. Parallel search on video cards. In *HotPar*, pages 9–9, 2009.

[Kaldewey *et al.*, 2012] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU join processing revisited. In *DaMoN*, pages 55–62, 2012.

[Karkkainen and Ukkonen, 1996] Juha Karkkainen and Esko Ukkonen. Sparse suffix trees. In *Computing and Combinatorics*, volume 1090 of *LCNS*, pages 219–230. 1996.

[Kim *et al.*, 2010] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[Kim *et al.*, 2011] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Designing fast architecture-sensitive tree search on modern multicore/many-core processors. *TODS*, 36(4):22:1–22:34, 2011.

[Klein and Shapira, 2005] Shmuel T. Klein and Dana Shapira. Pattern matching in huffman encoded texts. *Information Processing & Management*, 41(4):829 – 841, 2005.

[Knuth *et al.*, 1977] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[Kulishov, 2009] Feodor Kulishov. DFA-based and SIMD NFA-based regular expression matching on Cell BE for fast network traffic filtering. In *SIN*, pages 123–127, 2009.

[Li *et al.*, 2009] Jun Li, Shuangping Chen, and Yanhui Li. The fast evaluation of hidden Markov models on GPU. In *ICIS*, volume 4, pages 426–430, Nov 2009.

[Li *et al.*, 2014] Jiexing Li, Jeffrey Naughton, and Rimma V. Nehme. Resource bricolage for parallel database systems. *Proc. VLDB Endow.*, 8(1):25–36, 2014.

[Ligowski and Rudnicki, 2009] L. Ligowski and W. Rudnicki. An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IPDPS*, pages 1–8, 2009.

[Lin *et al.*, 2010] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and J.-M. Shyu. Accelerating string matching using multi-threaded algorithm on GPU. In *GLOBECOM*, pages 1–5, 2010.

[Lin *et al.*, 2013a] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. Accelerating pattern matching using a novel parallel algorithm on GPUs. *TC*, 62(10):1906–1916, 2013.

[Lin *et al.*, 2013b] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. Accelerating pattern matching using a novel parallel algorithm on GPUs. *TC*, 62(10):1906–1916, Oct 2013.

[Lin *et al.*, 2013c] Kuan-Ju Lin, Yi-Hsuan Huang, and Chun-Yuan Lin. Efficient parallel Knuth-Morris-Pratt algorithm for multi-GPUs with CUDA. In *Advances in Intelligent Systems and Applications*, volume 21, pages 543–552. 2013.

[Liu *et al.*, 2009] Yongchao Liu, Douglas L. Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):1–10, 2009.

[Lu *et al.*, 2010] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *DaMoN*, pages 19–26, 2010.

[MacCormick *et al.*, 2009] John MacCormick, Nicholas Murphy, Venugopalan Ramasubramanian, Udi Wieder, Junfeng Yang, and Lidong Zhou. Kinesis: A new approach to replica placement in distributed storage systems. *Trans. Storage*, 4(4):11:1–11:28, 2009.

[Manegold *et al.*, 2000] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, December 2000.

[Manegold *et al.*, 2002] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[MapD, 2016] MapD. Cloud leadership: Gpus + ibm softlayer. http://www.mapd.com/blog/2016/05/19/cloud-leadership-gpus-ibm-softlayer/, 2016.

[Markl *et al.*, 1999] V. Markl, F. Ramsak, and R. Bayer. Improving OLAP performance by multidimensional hierarchical clustering. In *IDEAS*, pages 165–177, 1999.

[Marziale *et al.*, 2007] Lodovico Marziale, Golden G. Richard III, and Vassil Roussev. Massive threading: Using GPUs to increase the performance of digital forensics tools. *Digital Investigation*, 4, Supplement(0):73 – 81, 2007.

[Mei and Chu, 2015] Xinxin Mei and Xiaowen Chu. Dissecting GPU memory hierarchy through microbenchmarking. *CoRR*, abs/1509.02308, 2015.

[Meng *et al.*, 2010] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, pages 235–246, 2010.

[Michael Hummel, 2010] Michael Hummel. ParStream - a parallel database on GPUs. `http://www.nvidia.com/content/gtc-2010/pdfs/4004a_gtc2010.pdf`, 2010.

[Microsoft, 2016] Microsoft. AzureCon on demand. `https://azure.microsoft.com/en-us/documentation/videos/azurecon-2015-applications-that-scale-using-gpu-compute/`, 2016.

[Moammer, 2015] Khalid Moammer. NVIDIA confirms pascal launching in 2016. `http://wccftech.com/nvidia-volta-succeed-pascal-2018/`, 2015.

[Moammer, 2016] Khalid Moammer. AMD Zen APU featuring HBM spotted. `http://wccftech.com/xbox-one-may-be-getting-a-new-apu-based-on-amds-polaris-architecture/`, 2016.

[Mostak and Graham, 2014] Todd Mostak and Tom Graham. Map-D data redefined. `http://on-demand.gputechconf.com/gtc/2014/webinar/gtc-express-map-d-webinar.pdf`, 2014.

[Mueller *et al.*, 2013] Rene Mueller, Tim Kaldewey, Guy M. Lohman, and John McPherson. Wow: What the world of (data) warehousing can learn from the world of warcraft. In *SIGMOD*, pages 961–964, 2013.

[Mytkowicz *et al.*, 2014] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *ASPLOS*, pages 529–542, 2014.

[Narasiman *et al.*, 2011] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU performance via large warps and two-level warp scheduling. MICRO, pages 308–317, 2011.

[Navarro, 2001] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[Needleman and Wunsch, 1970] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[Neumann, 2011] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.

[NVIDIA, 2010] NVIDIA. TESLA C2050 / C2070, 2010.

[NVIDIA, 2014] NVIDIA. NVIDIA NVlink high-speed interconnect: Application performance. Technical report, 2014.

[NVIDIA, 2015a] NVIDIA. Command line profiler, 2015.

[NVIDIA, 2015b] NVIDIA. Cuda C best practices guide, 2015.

[NVIDIA, 2015c] NVIDIA. NVIDIA CUDA C programming guide, 2015.

[NVIDIA, 2015d] NVIDIA. NVIDIAs next-gen Pascal GPU architecture to provide 10X speedup for deep learning apps. `https://blogs.nvidia.com/blog/2015/03/17/pascal/`, 2015.

[NVIDIA, 2015e] NVIDIA. Tesla K80 GPU accelerator, 2015.

[NVIDIA, 2016a] NVIDIA. CUDA parallel programming. `http://www.nvidia.com/object/cuda_home_new.html`, 2016.

[NVIDIA, 2016b] NVIDIA. GeForce 256. `http://www.nvidia.com/page/geforce256.html`, 2016. Accessed: 2016-04-24.

[Ori Netzer, 2014] Ori Netzer. Getting Big Data done on a GPU-based database. `http://on-demand.gputechconf.com/gtc/2014/presentations/S4644-big-data-gpu-based-database.pdf`, 2014.

[Ozsoy and Swany, 2011] A. Ozsoy and M. Swany. CULZSS: LZSS lossless data compression on CUDA. In *CLUSTER*, pages 403–411, 2011.

[Ozsoy *et al.*, 2014] Adnan Ozsoy, D. Martin Swany, and Arun Chauhan. Optimizing LZSS compression on GPGPUs. *Future Generation Comp. Syst.*, 30:170–178, 2014.

[Padmanabhan and others, 2003] Sriram Padmanabhan et al. Multi-dimensional clustering: A new data layout scheme in DB2. In *SIGMOD*, 2003.

[Pagh and Rodler, 2004] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[Patel *et al.*, 2012] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D. Owens. Parallel lossless data compression on the GPU. In *Innovative Parallel Computing*, 2012.

[Pirk *et al.*, 2014] H. Pirk, S. Manegold, and M. Kersten. Waste not...; efficient co-processing of relational data. In *ICDE*, pages 508–519, 2014.

[Plattner, 2009] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *SIGMOD*, pages 1–2, 2009.

[Polychroniou and Ross, 2014] Orestis Polychroniou and Kenneth A. Ross. Vectorized Bloom filters for advanced SIMD processors. In *DaMoN*, 2014.

[Polychroniou *et al.*, 2015] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[Pucheral *et al.*, 1990] Philippe Pucheral, Jean-Marie Thévenin, and P. Valduriez. Efficient main memory data management using the DBgraph storage model. In *VLDB*, pages 683–695, 1990.

[Pyrgiotis *et al.*, 2012] ThemistoklisK. Pyrgiotis, CharalamposS. Kouzinopoulos, and KonstantinosG. Margaritis. Parallel implementation of the Wu-Manber algorithm using the OpenCL framework. In *AIAI*, volume 382, pages 576–583. 2012.

[Raue, 2010] Kristian Raue, 2010.

[Rauhe *et al.*, 2013] Hannes Rauhe, Jonathan Dees, Kai-Uwe Sattler, and Franz Faerber. Multi-level parallel query execution framework for CPU and GPU. In *Advances in Databases and Information Systems*, volume 8133, pages 330–343. 2013.

[RE2, 2014] RE2. Regular expression library. `https://github.com/google/re2`, 2014.

[Rissanen, 1976] Jorma J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, May 1976.

[Rosenfeld *et al.*, 2015] Viktor Rosenfeld, Max Heimel, Christoph Viebig, and Volker Markl. The operator variant selection problem on heterogeneous hardware. In *ADMS*, pages 1–12, 2015.

[Ross, 2004] Kenneth A. Ross. Selection conditions in main memory. *TODS*, 29(1):132–161, 2004.

[Ross, 2007] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, April 2007.

[Salapura *et al.*, 2012] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira. Accelerating business analytics applications. In *HPCA*, pages 1–10, 2012.

[Sartori and Kumar, 2013] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *TMM*, 15(2):279–290, 2013.

[Scarpazza *et al.*, 2007] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Peak-performance DFA-based string matching on the Cell processor. In *IPDPS*, pages 1–8, 2007.

[Singer, 2013a] Graham Singer. The history of the modern graphic processor. `http://www.techspot.com/article/650-history-of-the-gpu/`, 2013.

[Singer, 2013b] Graham Singer. The history of the modern graphic processor, Part 2. `http://www.techspot.com/article/653-history-of-the-gpu-part-2/`, 2013.

[Sitaridi and Ross, 2013] Evangelia A. Sitaridi and Kenneth A. Ross. Optimizing select conditions on GPUs. In *DaMoN*, pages 4:1–4:8, 2013.

[Sitaridi and Ross, 2015] Evangelia A. Sitaridi and Kenneth A. Ross. GPU-accelerated string matching for database applications. *The VLDB Journal*, pages 1–22, 2015.

[Sitaridi *et al.*, 2013] Evangelia A. Sitaridi, Rene Mueller, and Tim Kaldewey. Parallel lossless compression using GPUs. In *GPU Technology Conference GTC'13*, 2013.

[Smith and Waterman, 1981] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[Stockinger and Wu, 2006] Kurt Stockinger and Kesheng Wu. Bitmap indices for data warehouses. *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, 2006.

[Suhan and Mostak, 2015] Alex Suhan and Todd Mostak. MapD: Massive throughput database queries with LLVM on GPUs. `https://devblogs.nvidia.com/parallelforall/mapd-massive-throughput-database-queries-llvm-gpus/`, 2015.

[Sunday, 1990] Daniel M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

[Sutter, 2005] Herb Sutter. The free lunch is over. *Dr Dobb's*, 30(3), 2005.

[Tableau, 2016] Tableau. How people use Tableau. `http://www.tableau.com/stories`, 2016. Accessed: 2016-04-26.

[Taylor and Li, 2011] Ryan Taylor and Xiaoming Li. Software-based branch predication for AMD GPUs. *SIGARCH Comput. Archit. News*, 38(4):66–72, 2011.

[TechPowerUp, 2015] TechPowerUp. NVIDIA GeForce 256 SDR. `https://www.techpowerup.com/gpudb/731/geforce-256-sdr`, 2015. Accessed: 2016-04-24.

[Tian *et al.*, 2005] Yuanyuan Tian, Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3):281–299, 2005.

[Tran *et al.*, 2014] N. P. Tran, D. H. Choi, and M. Lee. Optimizing cache locality for irregular data accesses on many-core Intel Xeon Phi accelerator chip. In *HPCC*, pages 153–156, 2014.

[Transaction Processing Performance Council, 2014] Transaction Processing Performance Council. TPC-H benchmark. `http://www.tpc.org/tpch/`, 2014.

[Trust Sanger Institute, 2001] Trust Sanger Institute. Yersinia Pestis chromosome. `ftp://ftp.sanger.ac.uk/pub/project/pathogens/yp/Yp.dna`, 2001.

[Vasiliadis *et al.*, 2011] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Parallelization and characterization of pattern matching using GPUs. In *IISWC*, pages 216–225, 2011.

[Wang *et al.*, 2014] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with GPUs. *Proc. VLDB Endow.*, 7(11):1011–1022, July 2014.

[Wang *et al.*, 2015] Leyuan Wang, Sean Baxter, and John D. Owens. Fast parallel suffix array on the GPU. In *Euro-Par*, pages 573–587, 2015.

[Weiner, 1973] Peter Weiner. Linear pattern matching algorithms. In *Swat*, pages 1–11, 1973.

[Welch, 1984] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[Whang and Krishnamurthy, 1990] Kyu-Young Whang and Ravi Krishnamurthy. Query optimization in a memory-resident domain relational calculus database system. *TODS*, 15(1):67–95, 1990.

[Williams, 1991] R.N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference, 1991. DCC '91.*, pages 362–371, Apr 1991.

[Wu *et al.*, 2012a] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *MICRO*, pages 107–118, 2012.

[Wu *et al.*, 2012b] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *IPDPSW*, pages 2433–2442, 2012.

[Wu *et al.*, 2013] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proc. VLDB Endow.*, 6(10), 2013.

[Wu *et al.*, 2014] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red Fox: An execution environment for relational query processing on GPUs. In *CGO*, page 44:4444:54, 2014.

[Xilinx, 2015] Xilinx. GUNZIP/ZLIB/Inflate data decompression core. `http://www.xilinx.com/products/intellectual-property/1-79drsh.html`, 2015. Accessed: 2015-11-14.

[Yang *et al.*, 2010] Y. H. E. Yang, V. K. Prasanna, and C. Jiang. Head-body partitioned string matching for deep packet inspection with scalable and attack-resilient performance. In *IPDPS*, pages 1–11, 2010.

[Ye *et al.*, 2011] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, pages 1–9, 2011.

[Zha and Sahni, 2013] Xinyan Zha and S. Sahni. GPU-to-GPU and host-to-host multipattern string matching on a GPU. *TC*, 62(6):1156–1169, 2013.

[Zhang *et al.*, 2010] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining GPU applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *ICS*, pages 115–126, 2010.

[Zhang *et al.*, 2011] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, pages 369–380, 2011.

[Ziv and Lempel, 1977] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–343, May 1977.

[Ziv and Lempel, 1978] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, 24(5):530–536, Sep 1978.

[Zu *et al.*, 2012] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *PPoPP*, pages 129–140, 2012.

[Zukowski, 2009] Marcin Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, Universiteit van Amsterdam, 3 2009.

Appendix

Shared memory is divided into equally-sized buffers equal to the number of warps in a thread block. When threads in a warp read a warp's worth of data a local prefix sum is computed for the values satisfying the condition and these values are written in the shared memory buffer of this warp. To compute the local prefix sum we use CUDA intrinsics ballot (__ballot()) and population count (__popc()). Ballot function returns a 32-bit integer which combines the condition outcome from each thread, where the i-th bit is set if the condition is true for the corresponding thread in the warp. The population count function computes how many bits were set. When the buffer of each warp is full the threads of this warp write its contents to the global memory in a coalesced way.