ACCELERATION AND EXECUTION OF RELATIONAL QUERIES USING GENERAL PURPOSE GRAPHICS PROCESSING UNIT (GPGPU)

A Dissertation
Presented to
The Academic Faculty

By

Haicheng Wu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2015

ACCELERATION AND EXECUTION OF RELATIONAL QUERIES USING GENERAL PURPOSE GRAPHICS PROCESSING UNIT (GPGPU)

Approved by:

Dr. Sudhakar Yalamanchili, Advisor
Joseph M. Pettit Professor, School of ECE
Georgia Institute of Technology

Dr. Richard Vuduc
Associate Professor, School of CSE
Georgia Institute of Technology

Dr. Linda M Wills
Associate Professor, School of ECE
Georgia Institute of Technology

Dr. Santosh Pande
Associate Professor, School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
Associate Professor, School of Computer Science
Georgia Institute of Technology

Date Approved: November 5, 2015

To my parents and my wife

ACKNOWLEDGMENT

Six years' PhD study is a very long journey. When I started it six years ago, I could not believe I would still be on campus when I was 30. Now standing at the finish line and looking back, I would say I really enjoy the process and am proud of being a member of the Georgia Tech community. Of course without the many help from the people I know or I do not know, I could not reach this achievement, not to mention enjoying these six years.

I need to thank my advisor Dr. Sudhakar Yalamanchili the most. He agreed to take me to the CASL lab just after a phone call when I was in desperate need of a research advisor. He pointed me to a challenging but promising research area. Without his insightful guidance and patience, I would not have that many publications and would not be confident of introducing myself as the leading researcher in my research area. I should also thank Dr. Yalamanchili for his generous recommendations for the internship, fellowship, and my current full time job. More than that, I am also very grateful that Dr. Yalamanchili admits my wife in CASL and allows us to work together. What Dr. Yalamanchili provides me is so much more than I could ever expect to receive from an academic advisor. I would try my best to be a person as nice and helpful as Dr. Yalamanchili after I graduate from Georgia Tech.

Second, I should thank my previous advisor, Dr. Nate Clark. Although we work together only for one year, he trained me very well. It was Nate that taught me how to do academic research, how to write a paper, how to answer properly to the reviewers, etc. I still remember how frustrated I was when I wrote my first paper which was submitted three weeks after I arrived at Atlanta. It was painful at that time, but eventually paid off. Without this one year's training, I could not have started my work in CASL so smoothly. Moreover, Nate sold his car to me at an incredibly low price when I was financially unstable and in desperate need of a car.

Third, I should thank Dr. Guotong Zhou. She gave me the chance to be a Tech student through the Georgia Tech Shanghai program. She helped me to receive the scholarship to reduce my tuition burden and get through a lot of administrative problems when I transferred to the Atlanta campus. She gave me the help when I had to change academic advisors. She did much more than she should do for all GT-Shanghai students.

I also need to thank all of my committee members, Dr. Hyesoon Kim, Dr. Santosh Pande, Dr. Richard Vuduc, Dr. Linda Wills, and as well as Dr. Karsten Schwan. I know most of them very well in the six years and I even took classes from some of them. It was so sad that Karsten cannot show up in my defense. I wish I could defense earlier.

Six years ago, I could imagine that I would get a PhD one day (but not using six years), I would have fruitful research results, and I would settle down in the United States, but it was far beyond my imagination at that time that Jin would become my wife one day. Spending days and nights together with Jin for the last several years is the best thing ever happened in my life. She is not only a good wife, but also a first class researcher. She organizes my life, brainstorms with me, helps me out when I get stuck, encourages me when I was stressful. Without her, I am not complete.

Many thanks should also go to my parents. Their support is essential for my success. I left China in my family's most difficult time, but they gave me everything they could to help me get used to the life in Atlanta. My mother spent several months with me in the United States in the last six years. I understand how lonely they were when I was not around. Their sacrifice is what I can never pay back.

I also appreciate a lot the help that come from my friends and my lab mates, Greg Diamos, Andrew Kerr, Jeff Young, Minhaj Hassan, Eric Anger, Naila Farooqui, Si Li, William Song, Nawaf Almoosa, Chad Kersey. Greg was also my collaborator and mentor when I interned in NVIDIA. They are all good people and smart researchers. They are always there when I need them.

I am also thankful to all my collaborators, Greg Diamos, Daniel Zinn, Tim Sheard,

Sean Baxter, Molham Aref, Srihari Cadambi, Michael Garland. It is my great honor to work with all of you. Without you, I can never get these publications. I also would like to thank all other mentors, managers, and colleagues of mine during my internships. All the four internships I had in TI, NEC Labs, and NVIDIA were great experience for me.

I thank all my sponsors, LogicBlox, Intel, NVIDIA, NSF. It is a great honor to be a two time recipient of NVIDIA fellowship. The generous support from all these organizations play very important roles in supporting my PhD study both technically and financially.

Last but not least, I thank all the anonymous reviewers of my publications. Their comments, suggestions, and even criticisms significantly helped to shape my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

API application program interface. 16, 17, 27

APU Accelerated Processing Unit. 17

AST Abstract Syntax Tree. 24, 28, 29

BFS Breadth-first Search. 86

BSP bulk synchronous parallel. 4, 6

CFG Control-flow Graph. 10, 11, 24, 25, 34

CPU Central Processing Unit. xii, 1, 2, 4–9, 13, 15–17, 19, 20, 22, 34, 36, 37, 39–41, 43, 44, 47, 51, 77, 78, 98, 100, 102, 104–106, 110–112, 115–122, 124, 125

CSR compressed sparse row. 9, 85

CTA Cooperative Thread Arrays. 13, 16, 19, 52, 54, 56, 57, 59, 62–67, 70, 74, 75, 87, 96, 113, 123

DFG Data-flow Graph. 10, 24

DRAM Dynamic Random-access Memory. 15, 43

ECC error-correcting code. 110

GPGPU General Purpose GPU. 16, 86, 101, 103

GPU Graphics Processing Unit. x, xii, 1–11, 13, 15–23, 26–28, 30, 33, 34, 36, 37, 39–41, 43–52, 62, 65, 67, 68, 70–72, 74, 76, 77, 85, 86, 89–91, 96–106, 110–125

HPC High Performance Computing. 3, 4

HPRC High Performance Relational Computing. 1

ILP instruction level parallelism. 3, 87, 122, 123

IR Internal Representation. x, 7, 8, 19, 23, 25–27, 30, 34, 122

ISA Instruction Set Architecture. 13, 27

JIT just-in-time. 34, 51

LFTJ Leapfrog Triejoin. xi, xii, 6, 78, 81–84, 86, 98–102, 105–112, 116–121, 123

PCIe Peripheral Component Interconnect Express. xii, 4, 9, 15, 17–19, 36, 37, 41, 42, 45–47, 51, 68, 99, 101, 103, 105, 106, 114, 116, 120, 123–125

PTX Parallel Thread eXecution. 13, 26, 27, 34, 44, 49, 72, 73

RA Relational Algebra. 2, 3, 7–12, 17–19, 23–26, 30, 32, 43, 44, 50–54, 56, 57, 66, 74, 77

RISC reduced instruction set computing. 13

SMX Stream Multiprocessor. 13, 43

SSD solid-state drive. 103, 105, 107

STL Standard Template Library. 13

UVA Unified Virtual Addressing. 17, 19

SUMMARY

High Performance Relational Computing (HPRC) such as modern enterprise applications represents an emergent application arena that requires the processing of queries and computations over massive amounts of data. Large-scale, multi-GPU cluster systems potentially present a vehicle for major improvements in throughput and consequently overall performance. However, throughput improvement using GPUs is challenged by the distinctive memory and computational characteristics of Relational Algebra (RA) operators that are central to queries for answering business questions. This thesis first maps the relational computation onto GPUs by designing a series of tools and then explores the different opportunities of reducing the limitation brought by the memory hierarchy across the CPU and GPU system. Contributions from this thesis include:

- The design, implementation, and evaluation of Red Fox, a compiler and runtime infrastructure for executing relational queries on GPUs. Red Fox is comprised of (1) a language front-end for LogiQL which is a commercial query language, (2) an RA to GPU compiler, (3) optimized GPU implementation of RA operators, and (4) a supporting runtime. Its performance is evaluated on the full set of industry standard TPC-H queries on a single node GPU. Compared with a commercial LogiQL system implementation optimized for a state of art CPU machine, Red Fox on average is **11.20x** faster including PCIe transfer time. To the best of my knowledge, this is the first reported end-to-end compilation and execution infrastructure that supports the full set of TPC-H queries on commodity GPUs.

- Inspired in part by loop fusion optimizations in the scientific computing community, a new compiler technique called kernel fusion is designed as a basis for data movement optimizations. Kernel fusion fuses the code bodies of several GPU kernels to (1) reduce data footprint to cut down data movement throughout GPU and CPU

memory hierarchy, and (2) enlarge compiler optimization scope. The experiments on NVIDIA Fermi platforms demonstrate that kernel fusion can bring another 2.89x speedup in GPU computation on average across the micro-benchmarks tested.

- A CPU-based multi-predicate join algorithm is ported into GPU by exposing more fine-grained parallelism and adapting the memory access patterns to the GPU memory hierarchy. This is a good example of porting old sequential algorithm into GPUs. The new algorithm not only can aggregate the computation together to reduce the data movement overhead, but also can provide much better performance and be used with more flexibility compared with kernel fusion.

- The new GPU-based multi-predicate join algorithm is integrated into a multi-threaded CPU database runtime system that supports out-of-core data set. GPUs are viewed as another high throughput processors and the original partitioning and scheduling algorithms do not need to be modified. In the experiment, the input data are stored in SSD and two high-end GPUs are used in parallel. The integration proves the feasibility of using GPUs to accelerate existing database systems to solve real problems.

This thesis presents key insights, lessons learned, measurements from the implementations, and opportunities for further improvements.

CHAPTER 1

INTRODUCTION

High Performance Relational Computing (HPRC) has emerged as a discipline that mirrors high performance scientific computing in importance. HPRC promises to profoundly impact the way businesses grow, the way people access services, and how people generate new knowledge. This is especially true of the commercial enterprise space that is the engine of the economy. This enterprise software stack is a collection of infrastructure technologies supporting bookkeeping, analytics, planning, and forecasting applications for enterprise data. The task of constructing these applications is challenged by the increasing sophistication of the relational analysis required and the enormous volumes of data being generated and subject to analysis. Consequently, a growing demand exists for increased productivity in developing applications for sophisticated data-analysis tasks such as optimized search, probabilistic computation, and deductive reasoning. This demand is accompanied by the exponential growth in the target data sets and commensurate demands for increased throughput to keep pace with the growth in data volumes.

The explosive growth of Big Data in the enterprise has energized the search for architectural and systems solutions to sift through massive volumes of data. The use of programmable GPUs has appeared as a potential vehicle for high throughput implementations of enterprise applications with an order of magnitude or more performance improvement over traditional CPU-based implementations because discrete GPU accelerators provide massive fine-grained parallelism, higher raw computational throughput, and higher memory bandwidth compared to multi-core CPUs. This expectation is motivated by the fact that GPUs have demonstrated significant performance improvements for data intensive scientific applications such as molecular dynamics [2], physical simulations in science [3], options pricing in finance [4], and ray tracing in graphics [5]. It is also reflected in the emergence of accelerated cloud infrastructures for small and medium enterprise such as

Amazon's EC-2 with GPU instances [6]. Current EC2 8xlarge GPU instances are about 40% more expensive than 8xlarge CPU instances. Nominally, it is expected that if GPU implementations can provide significant speedup in excess of 40% relative to CPU implementations, enterprises will have a strong motivation to move to GPU-accelerated clusters if the software stacks can accommodate mainstream development infrastructures - a major motivation for the contributions of this thesis. A further motivation for the use of GPU accelerators is their energy efficiency. Nine out of the top ten supercomputers in the latest Green 500 use NVIDIA Tesla or AMD FirePro GPU cards [7].

While programming languages and environments have emerged to address productivity and algorithmic issues for these type of applications, the ability to harness modern high performance hardware accelerators such as GPU devices is nascent at best. Towards this end this dissertation researches the different aspects of using a GPU to accelerate relational computation including algorithm design, system implementation, and compiler optimizations. In particular, the targeted domain is on-line analytics processing wherein relational queries are processed over massive amounts of data. These queries are a mix of relational and arithmetic operators and are typically supported by a variety of declarative programming languages. The goal of this dissertation is to bridge the semantic gap between relational queries and GPU execution models and maintain significant performance speedup relative to the baseline CPU implementation. The target platforms are cloud systems comprising high performance multicore processors accelerated with GPUs, such as those from vendors NVIDIA, AMD, and Intel. The baseline implementation is executed on stock multicore blades that employs a runtime system that parcels out work units (e.g., a relational query over a data partition) to cores and manages out-of-core data sets. The envisioned accelerated configuration employs GPUs attached to the node that can also execute such work units where a relational query is comprised of a mix of Relational Algebra (RA) [8] (Set operators, PROJECT, SELECT, PRODUCT, and JOIN), arithmetic, and logic operators.

However, the application of GPUs to the acceleration of applications that perform relational queries and computations over massive amounts of data is a relatively recent trend and there are fundamental differences between such applications and compute-intensive scientific High Performance Computing (HPC) applications. Relational queries on the surface appear to exhibit significant data parallelism. Unfortunately, this parallelism is generally more unstructured and irregular than other domain specific operations, such as those common to dense linear algebra, complicating the design of efficient parallel implementations. RA operators also exhibit low operator density (operations per byte) making them very sensitive to and limited by the memory hierarchy and costs of data movement. Overall, the nature and structure of relational queries make different demands on the following:

- Traversals through the memory hierarchy: The low operator density of relational computation determines that the cost of moving data is much higher than the cost of computation. Moreover, relational computation tends to exhibit poor spatial and little temporal locality while modern processors and memory hierarchies are optimized for locality. Coupled with a high ratio of memory accesses, queries make poor use of memory bandwidth. When processing large amounts of data, reducing or hiding the data movement overhead is the fundamental challenge.

- Choice of logical and arithmetic operators: The computation of enterprise application is much simpler than HPC. For example, typically the largest part of a calculation is search which only requires address calculations, integer comparisons and conditional branches. Thus, there is not much room to improve the single thread instruction level parallelism (ILP) compared with the traditional scientific applications.

- Control flow structure: Computations over relations are highly data dependent. Control flow and memory access behaviors are difficult to predict while available concurrency is time varying, data dependent, and also difficult to predict. It is more challenging for GPU to handle these control flow when massive number of threads

run concurrently and synchronously.

- Data layout: Relational databases operate on relations or tables which is different from other HPC applications which commonly use sparse and dense multidimensional arrays. The independence between different rows or columns in a table is a source of potential parallelism in relational computation.

Compared with the baseline CPU only system, the target platform which is augmented by a GPU raises several unique problems as listed below.

- PCIe channel. Discrete GPUs communicate with the CPU through the PCIe bus which has much smaller bandwidth (16GB/s) compared with either the CPU memory bandwidth (100GB/s) or the GPU memory bandwidth (300GB/s). To make things worse, GPU memory capacity is much smaller than that of the CPU. The largest memory today found on a single GPU is 12GB. Due to the above two reasons, processing out-of-core data involves multiple data movement transfers over PCIe.

- Load Balance. Load balance is important for any parallel computation. GPUs host tens of thousands of fine grained threads that operate in synchronous thread groups. Small variations in load balance can produce large degradations in performance.

- GPU architecture. The GPU architecture is fundamentally different from the CPUs. Optimized for throughput rather than latency the GPU hosts a bulk synchronous parallel (BSP) execution model. Techniques that work well on CPU may not work well in a GPU. Thus, recompilation of an old code base may not lead to efficient performance. New algorithms and new data management techniques are needed.

The result of the above demands and challengers is significant algorithmic and engineering challenges to harnessing the throughput capabilities of GPUs. Consequently, two major components of the execution that need to be addressed are

- The efficient and effective execution of relational queries operating over data partitions to make use of the throughput of GPUs. This is essentially an algorithm issue.
- The management of large data sets whose size exceeds available memory.

To address these two components, four different but related steps are taken in this dissertation to effectively harness GPUs for relational computations.

First, an end-to-end compiler/runtime system called Red Fox [9] is designed to efficiently map full relational queries to GPUs. In Red Fox, an application is specified in LogiQL a declarative programming model for database and business analytics applications [10]. This development model for enterprise applications combines transactions with analytics by using a single expressive declarative language amenable to efficient evaluation schemes, automatic parallelization, and transactional semantics. The application then passes through a series of compilation stages that progressively lowers LogiQL into relational and arithmetic primitive operators, primitive operators into computation kernels, and finally kernels are translated into binaries that are executed on the GPU. A set of algorithm skeletons for each operator is provided as a CUDA [11] template library to the compiler. During compilation, the skeletons are instantiated to match the data structure format, types, and low level operations used by a specific operator. The application is then serialized into a binary format that is executed on the GPU by a runtime implementation. Red Fox is evaluated on the full set of TPC-H benchmark queries [12] executing on an NVIDIA GPU with small data sets which fit in the GPU memory.

Second, Kernel Weaver [13], an optimization module for Red Fox, is designed to address the challenge of optimizing data movement through the CPU-GPU memory hierarchy because relational computations are limited by each level of the memory. Considering the fact that Red Fox maps complex queries into dozens of dependent primitives which need to pass information by using large volume of intermediate data when the problem size is large. Kernel Weaver can automatically apply a cross-kernel optimization, kernel fusion [14], to these primitives. Kernel fusion is analogous to traditional loop fusion [15] that can fuses

small kernels into big ones. Its principal benefits are that it (1) reduces transfers of intermediate data through the CPU-GPU memory hierarchy, (2) reduces the overall memory data footprint of a sequence of kernels in each level of the memory hierarchy, and (3) increases the textual scope, and hence benefits, of many existing compiler optimizations.

Third, inspired by sequential LFTJ [1], a multi-predicate join algorithm optimized for the GPU is designed [16]. Similar to Kernel Weaver, this optimization fuses multiple join operations at the algorithmic level to reduce the data movement. However, this new algorithm also (1) uses a more compact TrieArray data structure, (2) eliminates intermediate data reorganization steps (e.g. sorting if using sort-merge join) when joining multiple predicates by different keys, (3) utilizes the GPU resource and memory bandwidth more efficiently.

The fourth and the last step is to handle out-of-core datasets, thereby freeing the implementation from the limitations of the on-board GPU memory and the size of the host memory. This extension is demonstrated by integrating the GPU-Optimized LFTJ with a CPU based database system. The CPU system is in charge of partitioning the data, assigning work loads to CPU and GPU cores.

1.1 Contributions

This thesis argues that *high throughput discrete GPUs can be used to accelerate relational computations for real world complex queries*. The specific contributions of this thesis are the following.

Red Fox compilation/runtime tool chain The main contribution of Red Fox is a solution for effectively executing full queries in heterogeneous clouds augmented with GPUs. and an implementation, demonstration, and evaluation of the solution. More specifically, Red Fox is an extensible compilation and execution flow for executing queries expressed in declarative/query languages on processors implementing the BSP [17] execution model

- specifically GPUs. It provides an engineering design that supports portability across multiple front-end languages and multiple back-end GPU architectures which is achieved via two standardized interfaces.

- Query Plan: A query plan format that decouples the language front-end from compiler optimizations making it possible to integrate other language front-ends, e.g., SQL [18] or NoSQL [19].

- Harmony IR [20]: An IR of operators implemented on the GPU that forms an interface with which to integrate (1) machine-specific optimizers and (2) different GPU language implementations for operators, e.g., CUDA or OpenCL [21] (Red Fox currently supports CUDA).

To the best of my knowledge, Red Fox is the first infrastructure that compiles and executes the complete TPC-H benchmark on GPUs. The result of evaluation proves the computation advantage and cost efficiency of running relational computations on a GPU.

Kernel Weaver optimization module Kernel Weaver design provides insight into how, why, and when kernel fusion works with quantitative measurements from implementations targeted to NVIDIA GPUs. The idea, the framework, and the algorithm for automated kernel fusion can also be applied to other application domain. In particular, the definition of basic dependencies between GPU kernels is general and can be used for other GPU compilation analysis and optimization passes. The quantification of the impact of kernel fusion on different levels of the memory hierarchy provides guidelines for future RA algorithm design.

Multi-predicate join algorithm This is a good example of how to port a sequential algorithm optimized for a CPU onto a GPU. The GPU algorithm changes the CPU tree traversal method from the depth first into breadth first to exploit fine grained parallelism. It also makes trade-offs between binary search and linear search to find the balance between computation complexity and good load balance and low control and memory divergence.

The evaluation with clique listing also demonstrates the possibility of using relational computation to solve graph problems on a GPU. This algorithm is also the fastest sort-merge join variant for GPUs to the best of my knowledge.

Out-of-core data set support The integration into a CPU based system and the extension to the support of out-of-core data set provides the design of a complete GPU-accelerated system that can be adapted a product. The reported end-to-end performance demonstrates the feasibility and efficiency of using GPUs to solve real world problem.

Collectively, these contribution will lead to the conclusion that GPUs can be used together with the existing database system and bring several times speedup.

1.2 Organization

The main body of this thesis introduces the above four steps in four chapters. The detailed organization is as follows.

Chapter 2 first briefly introduces the necessary background information for this thesis including the LogiQL query language, RA primitive operators, architecture and the programming model for NVIDIA GPUs. Next, this chapter describes the state-of-art in using GPU to accelerate relational computation.

Chapter 3 presents the design of Red Fox. It first introduces the structure of the two IRs of Red Fox which can be used to extend it to support different front-end languages or back-end devices. This chapter then describes each module of Red Fox including the language front-end, data structures, RA primitive algorithm library, RA-GPU compiler, and Harmony runtime. The experimental component compares the performance of Red Fox and a commercial CPU based database system over all 22 TPC-H queries. The experiment analysis also includes the cost comparison, discusses the individual query performance, breaks down the performance into individual queries and characterizes the impact of GPU architecture on the performance.

Chapter 4 describes kernel fusion optimization using Kernel Weaver for relational operators. This chapter first explains the source of the problem and then lists the six benefits of using kernel fusion. The discussion next presents the three steps to automate the process (1) finding primitives that can be fused, (2) choosing primitives that are profitable if fused together, and (3) generating the fused kernel code by interweaving the original source CUDA code. The experimental component lists the results of applying kernel fusion to several common combinations of RA primitives. The performance is analyzed to evaluate the benefits.

Chapter 5 describes the implementation of a multi-predicate join algorithm for GPUs. It first introduces the original CPU algorithm and then explains the use of the compressed sparse row (CSR) data structure. The chapter uses examples to explain how the CPU algorithm's tree traversal is changed to breadth first and subsequent algorithm design issues. The experiment section reports results for triangle listing and clique listing problems.

Chapter 6 is the last chapter of the main body of the thesis and discusses how to integrate the GPU multi-predicate join algorithm into a CPU runtime system. It first introduces the CPU partition algorithm, the system framework, and the use of double buffering to pipeline the GPU computation with the PCIe data transfer. The experimental component evaluates the performance by running triangle listing over large real and synthesized graphs in a GPU accelerated heterogeneous system.

Finally, Chapter 7 summarizes the role of GPUs in relational compaction pointing out the key aspects of using GPU in this application domain.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter first presents an overview of prior work in languages, algorithms, GPU architectures for relational computations on GPUs. The remainder of the chapter summarizes relevant related work.

2.1 Background

The background information needed by the rest of the thesis includes the LogiQL relational query language, RA primitive operators that queries can be translated into, and the GPU device that can be used to accelerate the processing of the RA primitives.

2.1.1 LogiQL

LogiQL is a variant of Datalog [22], with extensions (aggregations, arithmetic, integrity constraints and active rules) to support the development of an entire enterprise application stack: from user-interface controls, to workflow involving complex business rules, to sophisticated analysis over data. The declarative nature of LogiQL is the key reason for its suitability for rapid application development. LogiQL is a logic programming language, where computations over data are expressed in terms of logical relations among sets of data: e.g., conjunctions, implications, etc. Well-defined properties associated with logical relations, such as commutativity and associativity of conjunctions, readily expose data parallelism in LogiQL programs. Internally, relational data are organized as a key-value store [23]. The limited number of control operators (e.g., one sequencing operator), makes it easy to construct a finite Data-flow Graph (DFG) and Control-flow Graph (CFG).

Compared to popular imperative programming languages such as Java or C++, LogiQL

abstracts away much detail about the actual execution of a program from application developers: developers only specify logical relationships between data. Compared to emerging distributed programming languages for GPUs such as Map-Reduce [24], LogiQL expresses a richer set of relational and database operations that are cumbersome to implement in Map-Reduce.

2.1.2 Relational Algebra Primitives

Database programming languages are mainly derived from primitive operations common to first order logic. They are also declarative, in that they specify the expected result of the computation rather than a list of steps required to determine it. Due to their roots in first order logic, many database programming languages such as SQL and Datalog can be mostly or completely represented using RA. RA itself consists of a small number of fundamental operations, including PROJECT, SELECT, PRODUCT, SET operations (UNION, INTERSECT, and DIFFERENCE), and JOIN. These fundamental operators are themselves complex applications that are composed of multi-level algorithms and complex data structures. Given kernel-granularity implementations of these operations it is possible to compile many high level database applications into a CFG of RA kernels.

RA consists of a set of fundamental transformations that are applied to sets of primitive elements. Primitive elements consist of n-ary tuples that map attributes to values. Each attribute consists of a finite set of possible values and an n-ary tuple is a list of n values, one for each attribute. Another way to think about tuples is as coordinates in an n-dimensional space. An unordered set of tuples of this type specifies a region in this n-dimensional space and is termed a "relation". Each transformation in RA performs an operation on a relation, producing a new relation. Many operators divide the tuple attributes into *key* attributes and *value* attributes. In these operations, the key attributes are considered by the operator and the value attributes are treated as payload data that are not considered by the operation.

Table 1 lists the common RA operators and a few simple examples. In general, these operators perform simple tasks on a large amount of data. A typical complex relational

query consists of dozens of RA operators over massive data sets.

Table 1. The set of relational algebra operations. In the example, the 1st attribute is the "key". (Syntax: (x,y) – tuple of attributes; {(x1,y1),(x2,y2)} – relation; [0,2] – attribute index)

RA Operator	Description	Example
SET UNION	A binary operator that consumed two relations to produce a new relation consisting of tuples with keys that are present in at least one of the input relations.	x = {(2,b),(3,a),(4,a)}, y = {(0,a),(2,b)} UNION x y → {(0,a),(2,b),(3,a),(4,a)}
SET INTERSECTION	A binary operator that consumes two relations to produce a new relation consisting of tuples with keys that are present in both of the input relations.	x = {(2,b),(3,a),(4,a)}, y = {(0,a),(2,b)} INTERSECT x y → {(2,b)}
SET DIFFERENCE	A binary operator that consumes two relations to produce a new relation of tuples with keys that exist in one input relation and do not exist in the other input relation.	x = {(2,b),(3,a),(4,a)}, y = {(3,a),(4,a)} DIFFERENCE x y → {(2,b)}
CROSS PRODUCT	A binary operator that combines the attribute spaces of two relations to produce a new relation with tuples forming the set of all possible ordered sequences of attribute values from the input relations.	x = {(3,a),(4,a)}, y = {(True)} PRODUCT x y → {(3,a,True),(4,a,True)}
JOIN	A binary operator that intersects on the key attribute and cross product of value attributes.	x = {(2,b),(3,a),(4,a)}, y = {(2,f),(3,c),(3,d)} JOIN x y → {(2,b,f),(3,a,c),(3,a,d)}
PROJECT	A unary operator that consumes one input relation to produce a new output relation. The output relation is formed from tuples of the input relation after removing a specific set of attributes.	x = {(2,False,b),(3,True,a),(4,True,a)} PROJECT [0,2] x → {(2,b),(3,a),(4,a)}
SELECT	A unary operator that consumes one input relation to produce a new output relation that consists of the set of tuples that satisfy a predicate equation. This equation is specified as a series of comparison operations on tuple attributes.	x = {(2,False,b),(3,True,a),(4,True,a)} SELECT (key==2) x → {(2,False,b)}

Among all the RA primitive operators, JOIN is the most complex one and is more compute intensive than the rest of the RA primitives. Another problem of JOIN is that its output size can vary, i.e. between zero to the product of the sizes of the two inputs.

Conservatively reserving the largest possible memory space for storing the output wastes memory.

In addition to these operators, enterprise applications perform arithmetic computations ranging from simple operators such as aggregation to more complex functions such as statistical operators used for example in forecasting or retail analytics. Further, operators such as SORT and UNIQUE are required to maintain certain order amongst data elements and thereby can introduce certain ordering constraints amongst relations.

2.1.3 General Purpose GPUs

The current implementation targets NVIDIA GPUs and therefore the terminology of the bulk synchronous execution model underlying NVIDIA's CUDA language is adopted. Figure 1 shows an abstraction of NVIDIA's GPU architecture and execution model. A CUDA application is composed of a series of multi-threaded data parallel kernels. Data-parallel kernels are composed of a grid of parallel work-units called Cooperative Thread Arrays (CTA)s which in turn consist of an array of threads that may periodically synchronize at CTA-wide barriers. In the processors, threads within a CTA are grouped into logical units known as warps that are mapped to SIMD units called Stream Multiprocessor (SMX)s. Hardware warp and thread scheduling hides memory and pipeline latencies. Global memory is used to buffer data between kernels as well as to communicate between the CPU and GPU. Each SMX has a shared scratch-pad memory with allocations for each CTA and can be used as a software controlled cache. Registers are privately owned by each thread to store immediately used values.

Kernels are compiled to Parallel Thread eXecution (PTX), a virtual Instruction Set Architecture (ISA) that is realized on NVIDIA GPUs [25]. This PTX representation is a reduced instruction set computing (RISC) virtual machine similar to LLVM [26] that is amenable to classical compiler optimization. Based on CUDA, NVIDIA also distributes an open source template library–Thrust [27] which is very similar to the C++ Standard Template Library (STL) library and provides high performance parallel primitives such as

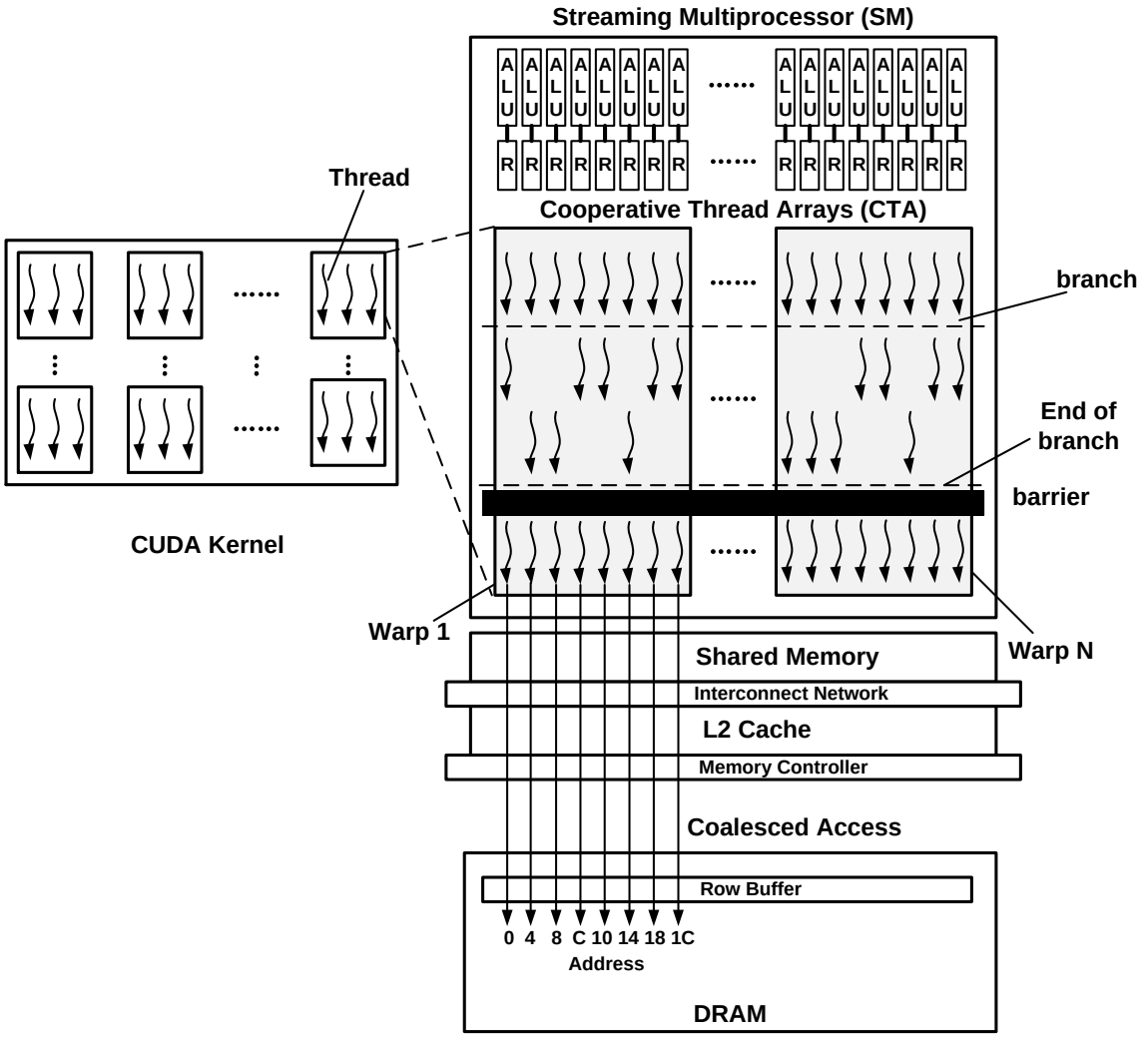

Figure 1. NVIDIA GPU Architecture and Execution Model.

SET operations. While the current implementation is based on CUDA, the use of bulk synchronous parallel model permits relatively straightforward support for industry standard OpenCL which is supported by NVIDIA, AMD and Intel GPUs.

Performance is maximized when all of the threads in the warp take the same path through the program. However, when threads in a warp do diverge on a branch, i.e., different threads take different paths, performance suffers because the execution of two paths is serialized. This is referred to as branch divergence. Memory divergence occurs when threads in a single warp experience different memory-reference latencies because the accessed data may locate in different levels of the memory hierarchy and the entire warp has

to wait until all memory references are satisfied.

GPU Dynamic Random-access Memory (DRAM) organizations favor coalesced memory accesses which are deep queues of coarse-grained bulk operations on large contiguous chunks of data, so that all data that is transferred to the row buffer is returned to the processor, and that it is accessed sequentially as a long burst. However, a large number of requests directed to addresses that map to different row buffers will force the memory controller to switch pages and trigger premature row buffer transfers, reducing effective bandwidth. Purely random accesses from active threads in a warp result in frequent row buffer transfers and address traffic, which significantly reduce effective DRAM bandwidth.

CUDA Stream is a feature provided by NVIDIA CUDA to increase the concurrency of using GPU. A CUDA stream represents a queue interface to the GPU device. Multiple streams can be established to a device and CUDA commands (PCIe transfer, CUDA kernel) in the same CUDA Stream are executed in order, while those in different streams can run concurrently relative to each other.

Compared with a traditional multi-core CPU, GPUs have a considerably larger number of (simpler) computation cores and higher memory bandwidth. However, discrete GPUs are interconnected with the CPU via the PCIe which has relatively much smaller bandwidth compared with either the CPU or GPU memory bandwidth. Moreover, the memory capacity of modern GPU boards is not large - 12GBytes is the largest today. Thus for practical data footprints, efficient staging and management of data movement between the host and GPU memories is very important.

Terms used to describe GPU abstractions such as data parallel threads and shared scratch-pad memory typically vary depending on the specific programming model being considered. CUDA typically uses the terms *thread* and *shared memory*, and OpenCL typically uses *work item* and *local memory*. The CUDA terminology is adopted in this thesis as is the implementation. However, the same concept and technology can also be applied to OpenCL and its supported devices.

2.2 Related Work

GPU-related database research is still in its very early stage. Previous research has focused mainly on algorithm design for primitives and has reported good performance compared with the CPU counterparts. Research groups have also been making an effort to design complete systems that can utilize the computation power of GPUs. Few of the groups can run complex real queries, besides which optimizations schemes are rarely reported.

2.2.1 RA Primitive Researches

Before NVIDIA introduced CUDA in 2006, General Purpose GPU (GPGPU) [28] programming relied on graphics application program interface (API)s such as DirectX [29] and OpenGL [30]. The limited support of programmability in hardware and software makes it very difficult to perform a simple task such as matrix multiplication on GPUs. During that period, Govindaraju et al. [31] designed GPU algorithms for several primitives such as selection and aggregation. They observed noticeable speedup compared with optimized CPU-based algorithms in some cases and that the GPU would have a bright future in the database area.

The introduction of CUDA and OpenCL makes programming for GPUs much more productive. Wu et al. [32] used CUDA to accelerate column scanning and get 5-6x speedup against an eight-core CPU. He et al. [33] used CUDA to design a series of join algorithms including sort-merge join, nested-loop join and hash join which achieved 2-7x speedup over their CPU baseline. Later, Trancoso et al. [34] also implemented nested-loop join and hash join in a GPU. They reported up to 21x speedup compared to a single core system. Sitaridi et al. [35] researched the impact of shared memory bank conflicts on the performance of join and aggregation. They proposed to reduce this problem by duplicating the data.

Baxter [36] designed the ModernGPU library including the sort-merge join algorithms. Both the sort and the merge parts were based on the merge path framework [37] which can balance the work load between CTAs and threads. Either sort or merge are implemented in several CUDA kernels and all the kernels are pushed to be memory bound. Moreover,

his JOIN algorithm calculates the output size first so that it does not need to conservatively allocate a large segment of memory. The join throughput of his implementation can reach about 35GB/s for 64-bit random keys in a Titan GPU.

All the above algorithms assume the data fit and reside in the GPU memory. Considering the limited GPU memory capacity, these algorithms could not directly run a large data set. Moreover, the speedup would be reduced by including the PCIe data movement overhead since PCIe bandwidth is much smaller than GPU memory bandwidth. CUDA Unified Virtual Addressing (UVA) is a technique developed by NVIDIA in which programmers can use a single unified memory space to manage the data stored separately in the CPU and GPU. One advantage of UVA CUDA memory copy APIs is that the GPU can directly access the data allocated in the CPU memory, so that the input and output data do not have to have a local copy in the GPU, and a data set larger than GPU memory can be supported although data still has to be moved through the PCIe. Kaldewey et al. [38] designed a hash join algorithm by using UVA to support out-of-core data sets. Their measurements showed that the performance of the algorithm was close to the PCIe bandwidth limit.

Fused architectures that put the CPU and GPU on the same die are a recent trend in the heterogeneous system community. Intel GEN [39], AMD Accelerated Processing Unit (APU) [40] and NVIDIA Project Denver [41] represent the latest efforts from industry. More and better products will come in the future. At the cost of having a less powerful GPU accelerator due to the limitation of chip size, power, thermal, etc., the largest advantage provided by the fused architecture is that memory of the CPU and GPU will be shared (not completely shared for current products) and PCIe is no longer needed. He et al. [42] implemented a Hash Join algorithm in OpenCL that can utilize this feature and perform the computation in both the CPU and GPU. Their experiments performed on an AMD APU showed that the implementation was 1.53x or 1.35x faster than computing on CPU or GPU only respectively.

Broneske et al. [43] proposed a software engineering approach to design RA operators

for different processors including GPUs by using a single code base to reduce errors and improve maintainability.

2.2.2 Researches on GPU Accelerated Database System

Compared with the implementations of relational primitives, the system level research for executing queries on GPUs is much less mature. When executing a complete query involving many primitives, problems faced by a single primitive are amplified. To the best of my knowledge, Red Fox is the first system that can run complex queries on GPUs such as the full TPC-H benchmark. While the TPC-H website [44] lists and ranks reported large scale performance, none of the reference platforms use GPUs.

Pioneering early work in GPU database systems was GPUQP, developed by He et al [45] which has started to migrate to OpenCL [46]. Their system was built on top of the RA primitives they designed. However, they had not been addressing any runtime issue necessary to manage the execution of full scale queries and its interaction with the compilation. They only reported results for two TPC-H queries, Q1 and Q3, which were just slightly faster than the CPU counterparts. Subsequently, Fang et al. designed several data compression schemes based on GPUQP to reduce the PCIe data transfer for database queries [47]. The compression and decompression throughput on GPUs could reach 45 GB/s and 56 GB/s respectively.

Bakkum et al. also tried to run full queries on GPUs, but with a very different approach [48]. They modified the virtual machine infrastructure of SQLite to use GPUs to execute SQLite opcodes (not RA primitives). Their implementations achieved a 20-70x speedup for some simple synthesized queries compared with the stock SQLite. While novel, their implementations had not yet matured to complex operators (e.g., implementation supporting JOIN).

Recently, Yuan et al. [49] built a column-store GPU query engine, YDB, for data warehouse workloads and tested it with the Star Schema Benchmark [50], a benchmark modified from TPC-H. The engine they built also has a primitive library, which was implemented

in both CUDA and OpenCL. However, their engine lacks a runtime system and a flexible infrastructure (e.g., the two IRs in Red Fox). The performance Yuan et al. reported was 1.2-6.5x faster than their CPU baseline. The optimization suggestions they made (using UVA, invisible join, data compression) are also valuable for improving Red Fox and similar infrastructures. However, some of the observations the group mentioned are in conflict with measurements from Red Fox. For example, their statement that the performance gain from GPU hardware advancement is very small might suggest that their implementations of relational primitives are not fully optimized for the target hardware. This is also partially verified by their following work which is implemented by Wang et al. [51] which claimed that their memory usage is only 23%. To address this problem and pipeline the PCIe and GPU computation, they designed a framework, MultiQx-GPU to concurrently execute multiple queries by buffering data in CPU and an intelligent query scheduler. This thesis uses simple double buffer scheme to increase concurrency.

Martinez [52] et al. designed a Datalog engine for GPUs and could run it with simple Datalog rules. They designed algorithms and performed some simple optimizations for individual RA primitives and some fused operators for common patterns (e.g., two back-to-back selections). Compared with their work, LogiQL is more expressive and Red Fox uses a more efficient primitive design, has more optimizations, can automatically perform kernel fusion, and supports more complex queries.

Rauhe [53] et al. designed a multi-level parallelism framework to execute relational queries in GPUs and tested with only seven TPC-H queries due to the lack of support of functions such as string operations. They chose to manually write the OpenCL code for the queries instead of translating from the high level query language because they could apply all the techniques to improve the performance. The three levels in their platforms are CTA compute, CTA accumulation, and global accumulation. In the CTA compute phase, GPU threads execute the programs adopted from their earlier CPU implementation to a portion of the data. The results computed by the GPU threads are aggregated inside the CTAs and

then across the CTAs by another kernel. Their method did not optimize for GPUs and had problems such as load balance.

Pirk et al. [54, 55] have a very different approach to using the GPU. They partition the data bitwise and let the GPU run approximate computations upon the most significant bits of the data. Later on, the CPU refines the result by using the rest of the data. To do so, they implement each primitive into two versions, the approximate version and the refined version. They integrate their technique into MonetDB [56] and evaluate three TPC-H queries with upto 6x speedup with out of GPU core data sets.

Sitaridi et al. [57] proposed a method to create optimal query plans for executing selection that has compound conditions in GPUs and planned to extend it to other primitives in the future. Karnagel et al. [58] researched the placement of primitives in a heterogeneous system.

Two teams led by Heimel et al. and Breb et al. have a series of research efforts for accelerating database operations using GPUs. Together, they investigate the design space of using GPUs to accelerate database systems [59, 60].

Breb et al. designed a self-tuning query optimizer called HyPE [61, 62, 63, 64, 65, 66] that can assign primitives to run either on the CPU or GPU based on a GPU aware cost model. They also develop a column-store GPU-accelerated system CoGaDB [67] which is similar to Yap. CoGaDB uses Hype as its query optimizer. Running with the Star Schema benchmark, the performance of CoGaDB is on the same order as MonetDB. They also explore the security issues in using a GPU to process confidential data.

Heimel et al. first proposed to use a GPU to estimate data distribution to accelerate query optimization [68]. Next, they designed a portable platform called Ocelot [69, 70] as a module of MonetDB that can evaluate relational queries on different processors, CPU or GPU using OpenCL. They evaluate Ocelot on a subset of TPC-H queries and get comparable performance against the original MonetDB running on a multi-core CPU. Later, they use HyPE as query optimizer for Ocelot to further improve the performance [71, 72].

In summary, Red Fox compared with the earlier work is more complete since it supports all TPC-H queries. Red Fox is also more flexible because its modularized design supports both static and dynamic optimizations and can re-target different query languages or processors. The performance of Red Fox is also outstanding mainly because of its very efficient primitives. Researches such as GPU aware query optimization and data compression can also be used to enhance Red Fox. The following kernel fusion optimization and multiple-predicate join algorithm design focus on aggregating computation to reduce memory movement overhead. This contribution is unique and no other group has ever worked on it.

CHAPTER 3

RED FOX

The ability to use a GPU to accelerate Database applications is impeded in large part by the semantic gap between programming languages, models, and environments designed for productivity, and GPU hardware optimized for massive parallelism, speed, and energy efficiency. Towards this end, **Red Fox** is a system in which the productivity of declarative languages are combined with the throughput performance of modern high performance GPUs to accelerate relational queries. Queries and constraints in the system are expressed in a high-level logic-programming language called LogiQL . The relational data is stored as a key-value store to support a range of workloads corresponding to queries over data sets. The main contribution of this chapter is a solution for effectively mapping full queries and query plans to GPUs and an implementation, demonstration, and evaluation of the solution. As a point of comparison, a multicore CPU implementation of LogiQL provided by LogicBlox Inc. is used as performance baseline. Note that this chapter assumes all data required by the computation fitting in the GPU global memory because this chapter focuses on algorithmic aspects not data footprint. Chapter 6 discusses the situation when the data is larger than the GPU memory size.

3.1 System Overview

The current non-accelerated software system executes on stock cloud platforms comprised of multicore blades, e.g., Amazon EC2. The current system compiles queries operating over a data partition (a work unit) and dispatches them for execution on the cores. The longer term vision is to extend this dispatch capability to use GPU accelerators in addition to host cores. Red Fox only deals with the GPU compilation of queries in the context of this execution model while a discussion of the impact of this capability in the larger system is provided in Chapter 6.

The results in this chapter are reported for a stand-alone GPU implementation. The overall organization of Red Fox is illustrated in Figure 2. A LogiQL program is parsed and analyzed by the language front-end to produce an IR of the query plan that represents a plan of execution for a set of dependent and interrelated RA and arithmetic operators. The RA-Harmony compiler instantiates the query plan with executable CUDA implementations that are stored in the primitive library and converts it to the Harmony IR which is serialized into the binary that is executed by the runtime.

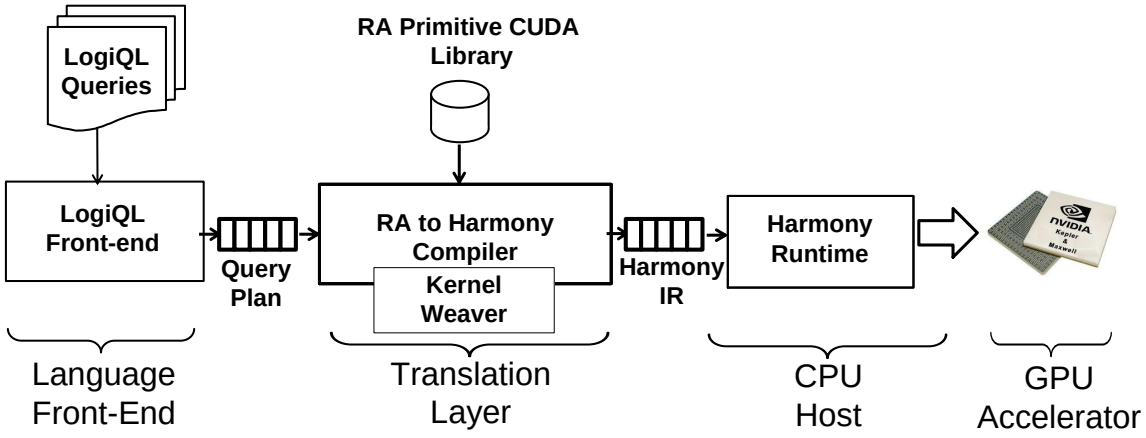

Figure 2. Red Fox platform.

The two IRs isolates the major components - the front-end, compiler, and runtime. This is to support the longer term goal of easier migration to other language front-ends and GPU backends (e.g., OpenCL). Collectively, these components implement a complete compilation chain from a LogiQL source file to a GPU binary.

3.1.1 Query Plan

Listing 3.1 is a simple example of a LogiQL program that classifies even and odd numbers. This example will be used throughout this chapter. A LogiQL file contains a number of declarations (lines 1, 5, 10, and 14) stating the type of relations and definitions (lines 12 and 15) stating how they are computed. For example, line 1 states that data type of *number* is 32-bit integer. Line 12 states that if m is odd and the next number after m is n, then n is even; similarly, Line 15 expresses that a number next to an even number is itself odd.

Together, they provide a **recursive definition** of the two relations. Line 2, 3, 6, 7, 11 explicitly assign initial data to *number*, *next*, and *even*. A LogiQL program starts with the initial data and iteratively derives facts for the other relations until it cannot derive any new facts. Note that if there are multiple rules having the same relation in the head, then the union of the derived data is computed. For example, *even* will contain 0 (as per Line 11) and other even numbers as of Line 12.

Listing 3.1. LogiQL Query Example

```
number(n) -> int32(n).                                    1
number(0).                                                2
number(1).                                                3
//other number facts elided for brevity                   4
next(n,m) -> int32(n), int32(m).                          5
next(0, 1).                                               6
next(1, 2).                                               7
//other next facts elided for brevity                     8
                                                          9
even(n) -> int32(n).                                      10
even(0).                                                  11
even(n) ←number(n),next(m,n),odd(m).                      12
                                                          13
odd(n) -> int32(n).                                       14
odd(n) ←next(m,n), even(m).                               15
```

The LogiQL front-end has two steps. In the first step, it parses a LogiQL source file into an Abstract Syntax Tree (AST) that stores information about relations and language clauses that operate on them. Clauses that contain multiple compound operations are broken down into atomic operations that can be executed individually. A list of all relations and their associated types is stored in this representation along with a DFG of the operations. In the second step, the front-end translates the AST into a CFG of RA operators. It performs a simple mapping from each LogiQL atomic operation to a series of abstract RA operators. Relations are also translated into equivalent types. All of this information is stored in a query plan.

The query plan consists two primary parts. The first part is the declarations of relations and the second part is a CFG of RA and other (e.g., arithmetic, aggregation, and string) operators. Figure 3 shows the query plan of the above LogiQL query example. The *variables* part lists all the used tuples (not single scalars) and the data types of each associated attribute. The *CFG* part is comprised of basic blocks and each basic block has several commands. Commands include

- RA and aggregation operations;
- data movement commands to make a duplicate of a variable;
- conditional and unconditional jumps to select the next basic block to execute;
- HALT command to terminate the execution.

Most of the work is performed by the relational operations. The MapFilter command is a combination of SELECT, PROJECT and arithmetic/string functions that is used by LogiQL . Similarly, the MapJoin command is a combination of JOIN, PROJECTION and arithmetic/string function. MapFilter and MapJoin will be further decomposed into operators in a later compilation stage. Moreover, recursive definitions are translated into loops in the query plan.

3.1.2 Harmony IR

The Harmony IR is adapted from [20] and represents a program as a CFG of side-effect-free kernels that operate on managed variables. The compiler maps RA operations to kernels and intermediate data structures to kernel variables. A runtime system is responsible for scheduling kernels on processors subject to control and data dependencies. It is also responsible for materializing variables and moving them between address spaces in a heterogeneous system.

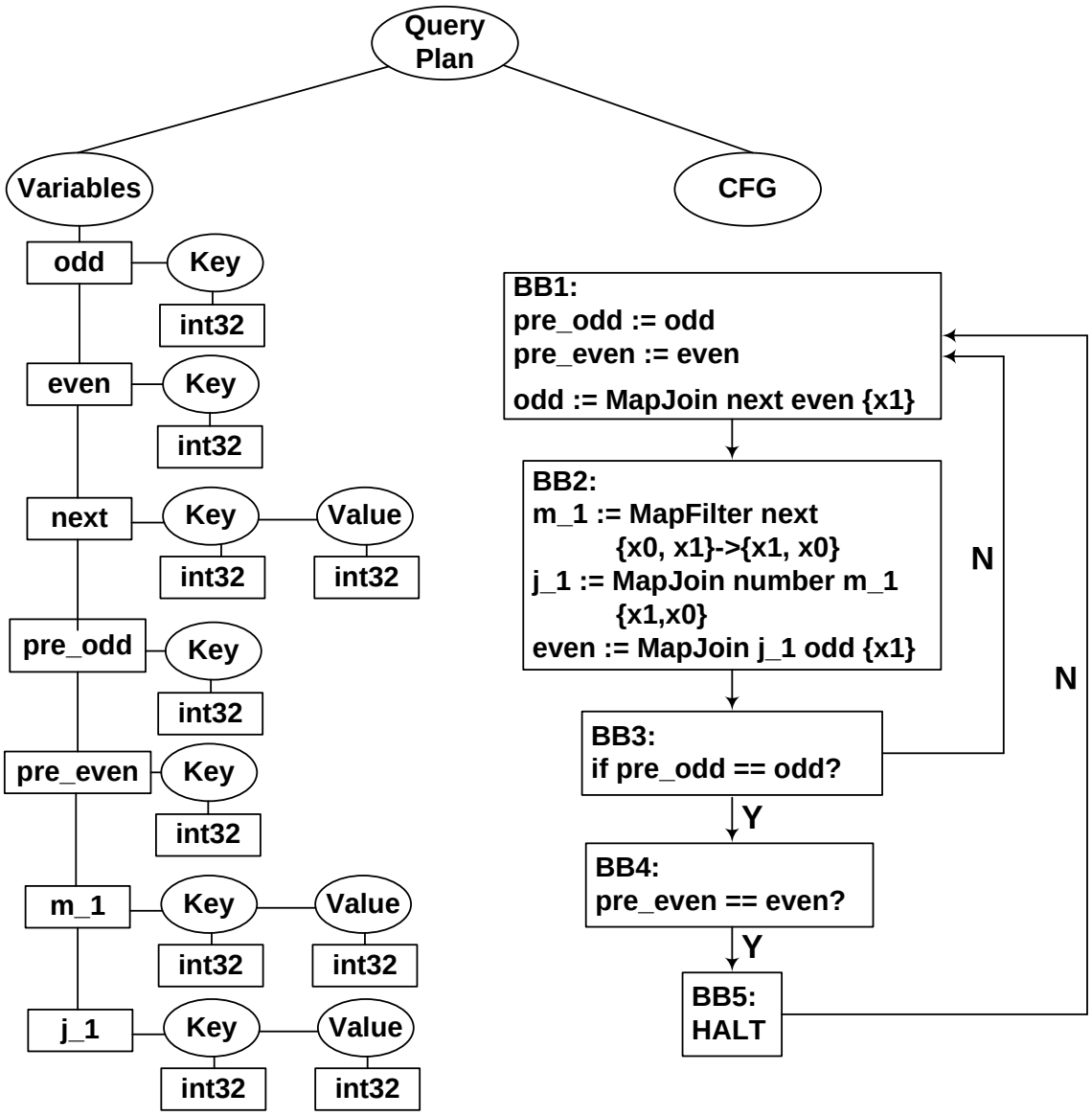

Figure 3. Example of a Query Plan (union of *even* and *odd* is omitted for brevity).

This high level representation lends itself to common program analysis and optimizations. For example, kernels can be scheduled statically subject to dependencies, and memory storage can be time multiplexed between variables using liveness information that directly falls out of the dataflow arcs between producing and consuming kernels.

Kernels can represent arbitrary operations. The compiler specializes skeleton implementations of RA algorithms first into templated CUDA source code, and then into a PTX kernel which is finally embedded in the Harmony IR. Kernels are executed on the GPU

hardware by first using the NVIDIA compiler to lower PTX to the native GPU ISA, and then by using the NVIDIA driver to launch the kernel.

Figure 4 is an example of translated Harmony IR. Operators in the query plan are broken into several kernels. Some temporary variables are added to store the intermediate data.

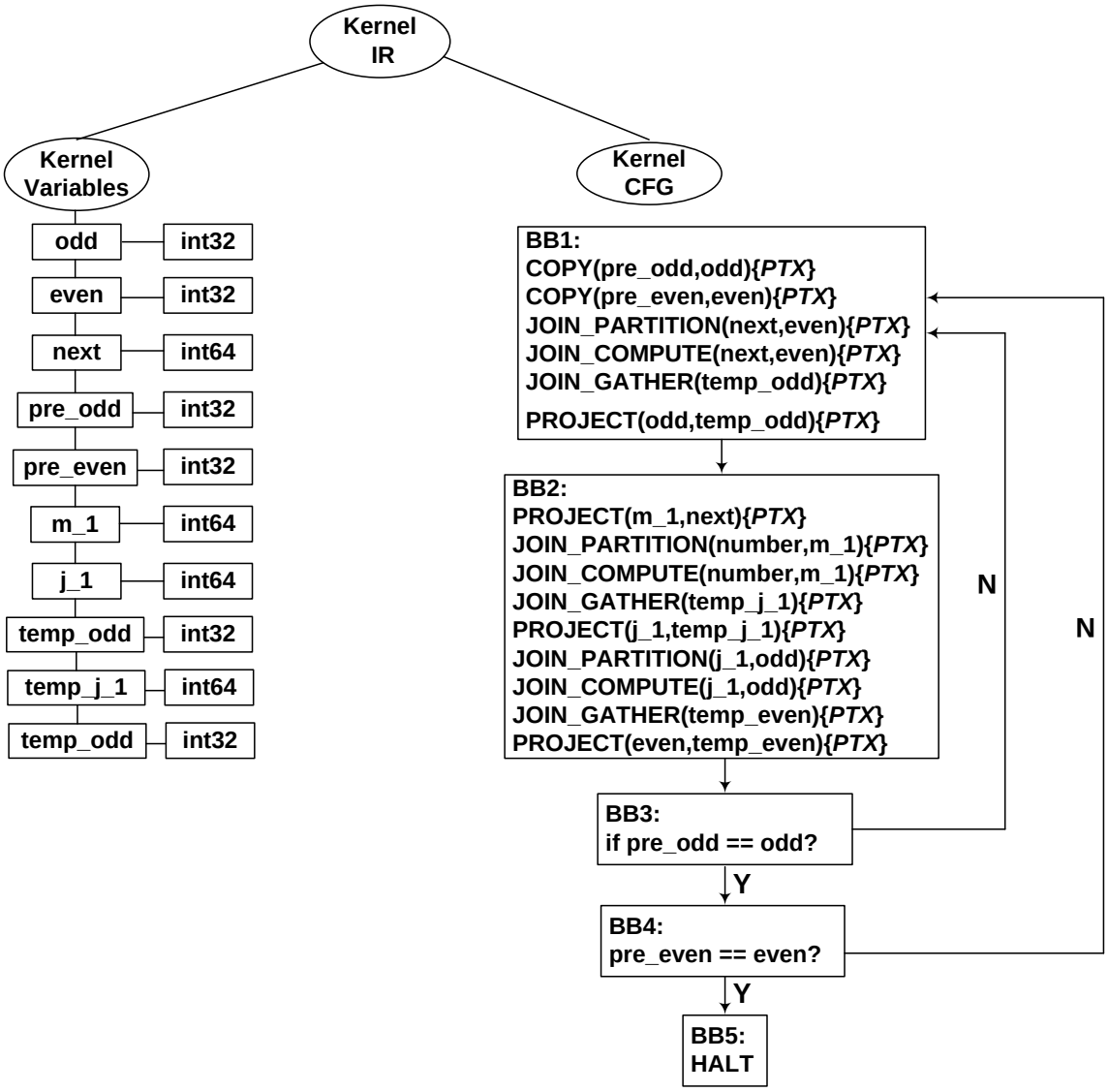

Figure 4. Example of the Harmony IR (union of *even* and *odd* is omitted for brevity).

To support devices other than NVIDIA GPUs, the kernels can be implemented in OpenCL. The back-end runtime can then be modified to launch kernels by calling the OpenCL runtime APIs instead of the CUDA driver APIs. These are engineering rather than conceptual challenges.

3.2 Implementation

The following sections introduce the three main modules of Red Fox.

3.2.1 LogiQL Front-end

The LogiQL front-end translates a LogiQL query into an GPU-executable query plan. Figure 5 shows the major functions of the front-end. The first and the second stages are provided by LogicBlox Inc. and they are described here for completeness. The pass manager and optimization passes are new contributions.

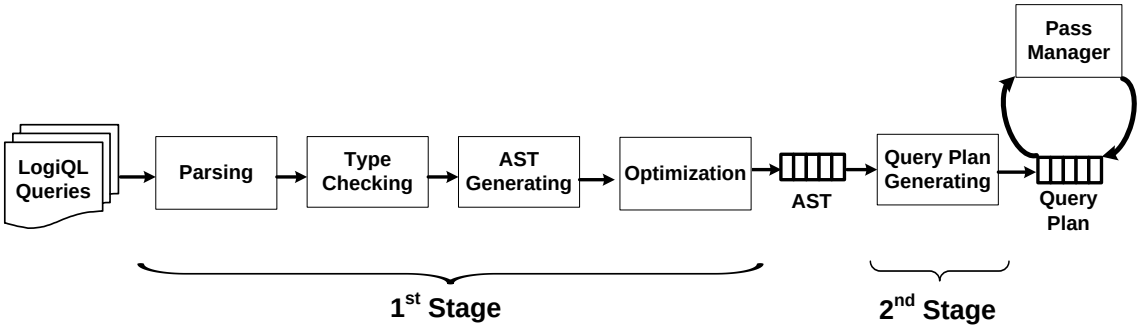

Figure 5. Compilation Flow of Red Fox Front-end.

In the first stage of the compilation, the front-end parses a LogiQL query consisting of a set of clauses, declarations, and constraints and builds an AST. In this process, types are strictly checked after parsing. The strong typing discipline of LogiQL guarantees that well typed LogiQL queries do not fail at runtime due to type errors. A complete set of type annotations for all LogiQL predicates and variables appearing in the query is added to the original query as a part of this process. In the odd-even classification example, any attempts to derive strings or float data into the integer only predicates (e.g., *odd*) is statically rejected, allowing the query evaluator to optimize the storage representation and selection of primitive operators without the need for the runtime to check at every step that correct values are used. The type checker also performs sound type inference, minimizing the amount of type annotations and declarations needed in the source query while preserving safety.

After type checking, high-level syntactic features (including disjunctive formulas, recursive definitions, complex expressions, automatic primitive operator overloading and conversions) are de-sugared into a core logical language. This language consists of an ordered set of executable logical clauses in dependency/execution order thus specifying high-level control flow. Within clauses, arithmetic and string operators are checked to ensure that no iteration over potentially infinite tuple spaces can occur, guaranteeing termination. Further simplification and optimization steps include common subexpression elimination, clause and predicate inlining, and generation of alternate indexes. The temporary result of the first stage is an intermediate AST containing information about the types and binding sites of variables, and information about control flow and potential parallelism as illustrated in Listing 3.2.

Listing 3.2. AST Example

```
Clauses1:                                                    1
  number(0). number(1). number(2).                           2
    number(3). number(4). number(5).                         3
  next(0,1). next(1,2). next(2,3).                           4
    next(3,4). next(4,5).                                    5
  even(0).                                                   6
                                                             7
Clauses2 {recursive}:                                        8
  for all {int32(n)}                                         9
    odd(n) ←exists {int32(m)}                                10
      next(m,n), even(m).                                    11
  for all {int32(n)}                                         12
    even (n) ←exists {int32(m)}                              13
      number(n),next(m,n),odd(m).                            14
                                                             15
Dependencies:                                                16
  Clauses2 ←Clauses1.                                        17
```

There are two clauses in Listing 3.2. The first clause contains data initializations of *number*, *next*, and *even*. The second clause describes how to compute *even* and *odd*. The first clause can be done in parallel since the initializations are independent of each other.

The second clause should be computed recursively because *odd* and *even* are mutually dependent. Clause 1 should be computed before Clause 2 because the computation of *even* and *odd* relies on *number* and *next*.

The second stage is to

1. map the predicates into tuple formats.

2. translate the clauses into a sequence of RA operations that can run on the GPU.

The ordering of Clause1, Clause2, and the implicit control flow of Clause2 is made explicit in a static, single assignment style. Figure 3 is an example of a translated query plan before any optimization. In this example, the data initialization part is omitted in the Figure. The recursive part is converted into loops over the section and checks for changes in the output relations after each iteration. Inside each basic block of Figure 3, the operators are ordered to respect the dependence requirements.

The end of the second stage is a pass manager which controls the transformation and analysis passes that run over the query plan IR. Currently, supporting passes include common (sub)expression elimination, several statistical passes and type inference passes which assign types and properties (e.g., uniqueness) to intermediate results. More relational and compiler optimizations will be added in the future.

3.2.2 RA-Harmony compiler

The RA-Harmony compiler translates a query plan to an executable GPU implementation exported in the Harmony IR format. The core part of the compiler is the primitive library. The job of the rest is to map the variables/operators in the query plan to data structure/CUDA implementation stored in the library. The primitive library, as shown in Figure 6 is comprised of three layers

1. The bottom layer deals with relation storage.

2. The middle corresponds to low level tuple operations that directly operate on tuple data, and

3. The top layer encompasses operator skeletons.

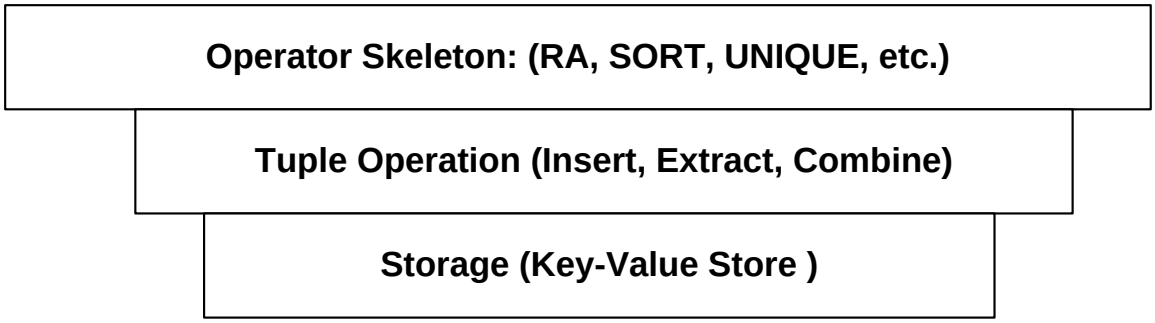

Figure 6. Three Layer Design of the RA-Kernel Compiler.

Relations are stored as key-value pairs. Both keys and values are represented by densely packed arrays of tuples. The sorted form allows for efficient array partitioning and tuple lookup operations. If the PROJECT operators change the key of a relation, the key tuple array and value tuple array will be reorganized to reflect the change. Figure 7 is an example of physical tuple data layout in the GPU memory. The padding zeros are used to pack the tuple data to the nearest 2^n byte boundary (n is the smallest integer necessary to store the tuple) to align the data storage and ease the system design. The tuple size is templated and the current system supports up to 1024-bit tuples which is very easy to extend if needed. Strings are stored separately in string tables with several string tables used to store different length strings. Only the string starting addresses are stored in the tuples. For example, if attribute 3 of Figure 7 is a string less than 32 characters, all the string contents in attribute 3 are stored in a string table whose entry has 32 bytes. The pointers to each entry are stored in the tuples rather than the string contents. The entry size of the largest string table is 128 bytes. If the string length is larger than 128 bytes or unknown beforehand, strings are stored in the 128-byte string table and one string might occupy multiple entries. A helper kernel is designed to set up the string tables.

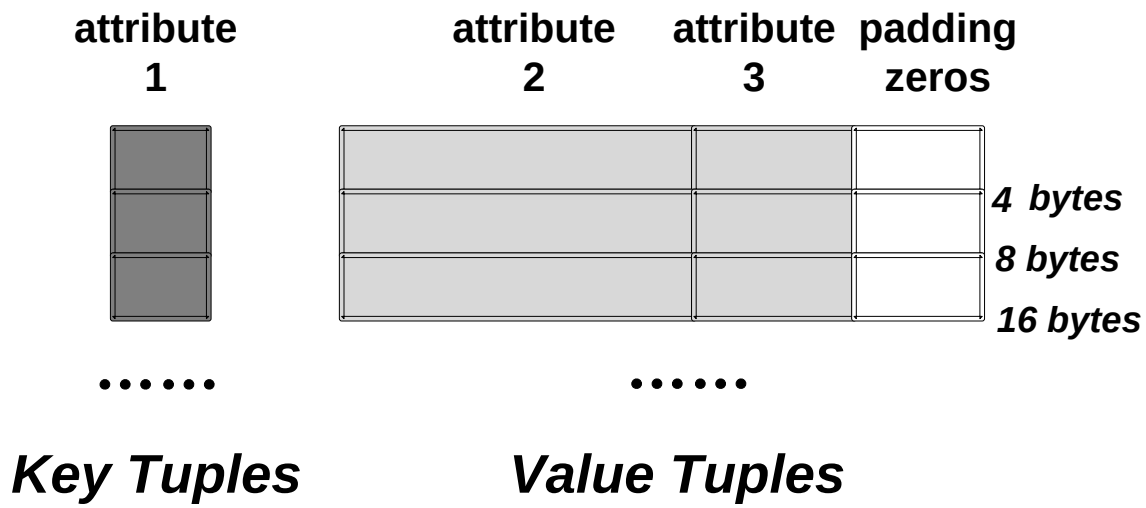

Figure 7. Example of Tuple Storage.

Low level tuple operators are called by the operator skeletons to manipulate tuples. These low level operations partially isolate the algorithm design and data storage and ease modification and optimization. Currently, RA operators uses three tuple operations:

1. Insert: insert an attribute into the tuple.
2. Extract: extract an attribute from the tuple.
3. Combine: combine two tuples by concatenating their value attributes which is used by the JOIN operator.

Finally, as to mapping the operators, a compound operator in a query plan such as Map-Filter or MapJoin is decomposed into SELECT or JOIN, PROJECT and arithmetic/string operators. Furthermore, SORT operators are added when some operators need sorted inputs. Currently, JOIN, AGGREGATION, and SET family require sorted inputs because of the chosen algorithm (introduced later in this subsection). Similarly, UNIQUE operators are added when the output data are required to retain uniqueness.

All operators are implemented using various algorithm skeletons that allow the same high level algorithm to be readily adapted to operations over complex data types. This

approach is commonly used in compilers for high level domain specific languages such as Copperhead [73], Optix [5].

The primitive library is very easy to integrate third party algorithms. Currently, Red Fox uses the algorithms designed by Diamos et al. [74] for PROJECT, PRODUCT, and SELECT. JOIN and merge sort algorithms are from ModernGPU library. The JOIN implementation is a variant of sort-merge join optimized for GPUs. They make trade-offs between computation complexity and memory access efficiency and scale well with high throughput. Red Fox chooses to use radix sort from Back40Computing library [75] when the sort key is short because radix sort has better computation complexity for short keys. Operators such as SET family, UNIQUE, and AGGREGATION (thrust::reduce_by_key) use implementations from NVIDIA's Thrust library. The remaining operators such as arithmetic (including datetime support), string operations (e.g., string append and substring) are data parallel operations and are re-implemented. Table 2 summarizes the algorithms stored in the primitive library and the source of the implementation.

Table 2. Algorithm Sources for Primitives.

Diamos et al.	SELECT, PROJECT, PRODUCT
ModernGPU	JOIN, merge sORT
Back40Computing	radix sort
Thrust	SET family, UNIQUE, AGGREGATION

Algorithm skeletons are CUDA implementations of operators that are templated on the tuple type and possibly the lower level operation type as well (e.g., comparison in SELECT). Once a operator has been mapped to a skeleton, the skeleton is instantiated for the data types of the relation, and the low level operations performed in the case of SELECT and PROJECT. Operators from Diamos et al. and ModernGPU library use the same three stage design (partition, compute, and gather) in the algorithms. Some operators such as JOIN involve more than one CUDA kernel. So, each operator may finally map

to more than one CUDA kernel. The instantiated skeletons are then compiled into PTX format by *NVCC* and stored using the Harmony IR format. Similarly, Thrust library calls are also compiled into binaries by *NVCC* and stored in the Harmony IR. Similar to the pass manager in the front-end, different optimizations (e.g., kernel fusion) can be applied over the Harmony IR.

3.2.3 Harmony Runtime

The runtime is responsible for executing Kernel binaries on a GPU device and performing data exchange between the CPU and GPU. Since the program is represented abstractly in terms of kernels and variables, the runtime has a fair degree of freedom in how it schedules or optimizes kernels and in how it allocates variables. A few potential optimizations are possible, motivated by the observations described in [76]. The runtime is designed to support those optimizations while the current implementation is optimized for deterministic debugging.

Figure 8 shows the structure of the runtime. Operationally, the runtime first loads the kernel binaries from the Harmony IR and forms them into a CFG of kernels. Kernels in each basic block in the graph are scheduled using a variant of list scheduling that attempts to minimize memory footprint by scheduling variable definitions and uses back-to-back. During scheduling, dataflow analysis is performed on the program to determine variable live ranges. Since variables represent complex data structures that may change size dynamically, explicit allocate and deallocate operations are inserted before definitions and after final uses respectively. The next step is to interpret the basic blocks in the CFG. When a basic block is selected for execution the first time, the PTX for the kernel is passed to the GPU driver for just-in-time (JIT) compilation. The runtime maintains a database of previously compiled kernels and the driver's compiler is invoked only when executing a new kernel. Finally, kernels in the basic block are submitted in-order to the GPU driver for execution. When the block completes, branching code at the end of the block determines the next block to begin executing or to halt the program.

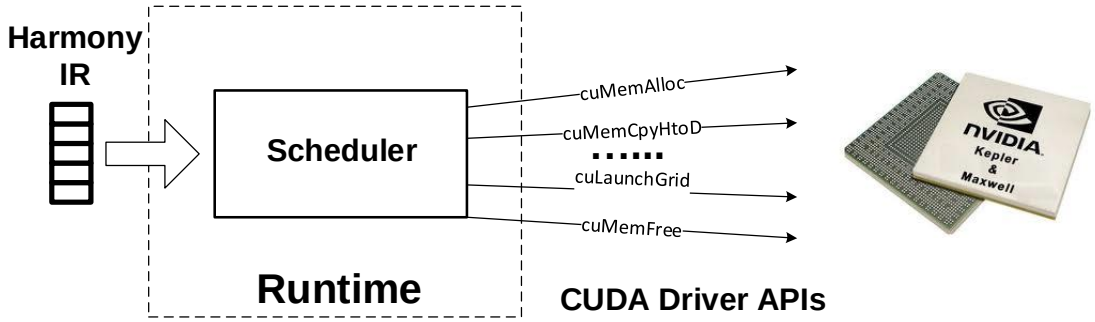

Figure 8. Red Fox Runtime.

3.3 TPC-H Performance

The experimental evaluation is conducted by compiling and executing all 22 queries of the TPC-H industry benchmark. TPC-H is a widely used benchmark for decision support systems. It is comprised of a set of business-oriented, complex, ad-hoc queries over large data sets. These queries answer important business questions by analyzing relations between customers, orders, suppliers and products using complex data types and multiple operators on large volumes of randomly generated data sets. Each query possesses unique features. For example, Figure 9(a) shows the generated query plan of Query 1 (Q1). Q1 provides a summary pricing report for all products shipped before a given date. Its query plan

1. concatenates several variables (e.g., *Status* and *Flag*) into a big table.
2. finds products whose *ShipDate* is before a constant date.
3. groups by *Status* and *Flag* and get pricing statistics.

The input data set used by the following experiments are all generated by the standard TPC-H data generator. The data generator has a parameter, *scale factor*, to control the input data set size. A scale factor of 1 is roughly equal to a 1GB database.

Finally, these queries are representative of industrial strength workloads. The complexity of the compiled queries ranges from 13 operators whose implementations include 56 CUDA kernels for query 13, to 150 operators whose implementations comprise 522 CUDA kernels for query 19.

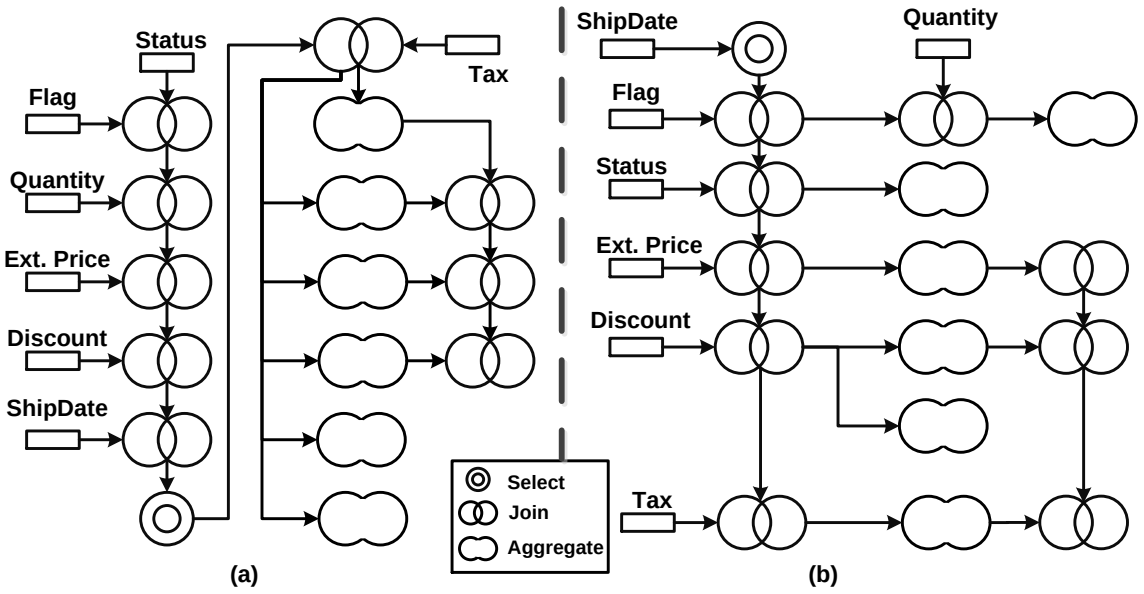

Figure 9. (a) Original Query Plan of Query 1; (b) Optimized Query Plan.

Table 3 provides the definition of the experimental platform. The NVIDIA GeForce GPU is attached as a device on the host PCIe channel. The overall cost of the system is about $2,400$ USD (GPU costs about $1,000$ USD). Scale Factor 1 TPC-H queries are evaluated in this section which fits in GPU memory. The query results are verified against a CPU implementation. Kernel Weaver is not used in this analysis.

Table 3. Red Fox experimental environment.

CPU	Intel i7-4771	GPU	GeForce GTX Titan
G++	4.6.3	NVCC	5.5
OS	Ubuntu 12.04	Thrust	1.7
CPU Mem	32GB	GPU Mem	6GB (288.4GB/s)
PCIe	3.0 x16		

The 22 queries cumulatively take 2.1 seconds including PCIe time with 1.6 seconds spent in GPU computation. Across the 22 queries, PCIe takes 11.7% of the total execution time. The PCIe transfers only occur at the beginning and the end of the query computation to move the input and output data sets between the CPU and the GPU. The allocated host memory used to buffer the data are pinned memory which can be read/written by GPU at

higher bandwidth than pageable host memory.

3.3.1 Performance Breakdown

Table 4 summarizes the performance of all 22 TPC-H queries for scale factor 1 (SF 1) whose data set size fits the GPU memory. The execution time in column 2 *includes* the PCIe transfer time and the GPU computation time. Column 3 reports the pure GPU computation time. The PCIe transfers only occur at the beginning and the end of the query computation to move the input and output data sets between the CPU and the GPU. The allocated host memory used to buffer the data are pinned memory which can be read/written by GPU at higher bandwidth than pageable host memory. The query results are verified against a CPU implementation. For SF 1, queries take from 0.01 seconds to 0.25 seconds to execute. Across the 22 queries, PCIe takes 25.0% of the total execution time. Several queries (Q4, Q7, Q15, Q18, and Q20) spend more than 33% of their time in PCIe transfers - motivating pipelined execution of PCIe transfers and GPU computation.

The Power metric and Price/Performance metric are two standard reporting conventions required by the TPC-H organization. The TPC-H power metric [1] (the higher, the better) measures the raw performance. It reports how many queries the system can execute back to back in one hour, i.e. the reverse of the query execution time geometric mean. The value of the power metric for Red Fox is **49,061 QphH@1GB** (w/ PCIe), or **67,245 QphH@1GB** (w/o PCIe). The TPC-H Price/Performance metric (the lower, the better) is the unit cost for performance. For Red Fox, the number is about **0.05 USD/QphH@1GB**, (w/ PCIe), or **0.04 USD/QphH@1GB** (w/o PCIe). The last two columns of Table 4 lists the TPC-H performance of a CPU-based system which will be discussed in Section 3.3.2.

Figure 10 shows the frequency of occurrence of each primitive (top part) and execution time breakdown (bottom part) across all the queries. The bar "others" includes arithmetic operations, string operations, etc. SORT is split into two parts: SORT_JOIN and

[1] TPC-H Power@Size = 3600 * SF / $\left(\prod_{i=1}^{22} T_{Qi}\right)^{1/22}$.

Table 4. TPC-H Performance (SF 1).

Query #	Execution Time (seconds) GPU (w/ PCIe)	GPU (w/o PCIe)	CPU Parallel	CPU Serial
Q1	0.23	0.18	2.76	18.60
Q2	0.03	0.02	0.41	2.35
Q3	0.10	0.07	2.88	4.74
Q4	0.06	0.04	0.34	2.59
Q5	0.08	0.06	1.19	19.68
Q6	0.11	0.08	0.91	11.50
Q7	0.08	0.05	0.62	4.87
Q8	0.12	0.09	1.17	12.25
Q9	0.15	0.11	2.00	132.7
Q10	0.12	0.10	0.75	9.35
Q11	0.01	0.01	0.27	2.76
Q12	0.25	0.21	1.31	10.54
Q13	0.08	0.06	0.60	2.38
Q14	0.14	0.11	0.82	3.22
Q15	0.07	0.04	0.59	2.11
Q16	0.02	0.01	1.21	4.65
Q17	0.10	0.08	0.19	43.12
Q18	0.03	0.02	0.51	4.86
Q19	0.19	0.14	1.80	40.67
Q20	0.05	0.02	0.27	21.57
Q21	0.09	0.06	2.25	18.32
Q22	0.02	0.02	0.78	2.97
Total	**4.49**	**3.96**	**23.65**	**375.89**

SORT_AGG. The former is the sorting before the JOIN and the latter is the sorting before AGGREGATION. Overall, the most widely used operator is PROJECT, followed by JOIN. SORT_JOIN is called about half as frequent as JOIN because one or both inputs of the JOINs are already sorted. This also happens to AGGREGATION and SORT_AGG. The least occurring operators are SET DIFFERENCE and UNIQUE since only a few queries use them. SET INTERSECTION is never called.

Each bar in the bottom part of Figure 10 represents the percentage of time spent in an operator across all 22 queries. Overall, most of the execution time is spent in SORT_JOIN (38%) and JOIN (31%) as expected. Therefore these operators deserve more attention in

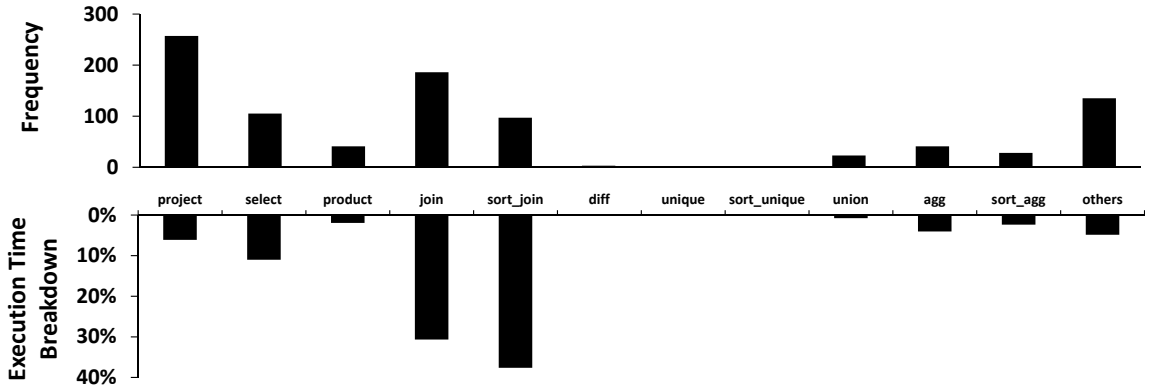

Figure 10. Red Fox operator frequency (a) and performance break down (b) for SF 1.

algorithm optimization on GPUs. Especially for the sort-merge join algorithm used in the paper, sorting takes much longer than the merging. Furthermore, although PROJECT and SELECT are used frequently, the percentage of execution time is relatively small. They are not computationally intensive. Consequently early ordering of SELECT and PROJECT operators in a query plan can significantly reduce the run time of downstream operators like JOIN and SORT by pruning data set sizes while also being computationally simple and embarrassingly parallel (across tuples).

3.3.2 Performance Comparison

The execution performance on GPUs is compared to that of the commercial LogiQL implementation, LogicBlox 4.0, on CPUs. Stock compilation is used without specific optimizations manually or otherwise targeted to TPC-H. All 22 queries are tested in one Amazon EC2 instance cr1.8xlarge (2× Intel Xeon E5-2670, 16 cores in total) with 32 threads to parallelize the query processing. The overall cost of this instance is about 6,000 USD excluding the network and software cost, which is 2.5x as expensive as the tested GPU system. Two Xeon E5-2670 CPUs cost about 3,000 USD which is three times the price of the GTX Geforce Titan card. The theoretical memory bandwidth of the Xeon CPU is 51.2 GB/s which is about 17% of that of the GPU device used in the evaluation.

The fourth column of Table 4 lists the absolute execution time of the commercial system. Figure 11 shows the relative speedup of individual queries achieved by Red Fox compared with this system for SF 1 data sets. It should be noted that the baseline commercial system employs novel optimizations that produces very efficient and often optimal or near optimal query plans for CPUs. The query plans produced for the GPU are *not* fully optimized and do not employ such industrial strength query plan optimizations. Thus, the speedup estimates are conservative. Section 3.3.3 discusses in greater detail aspects of improvements for GPUs which can produce additional factors of performance improvement.

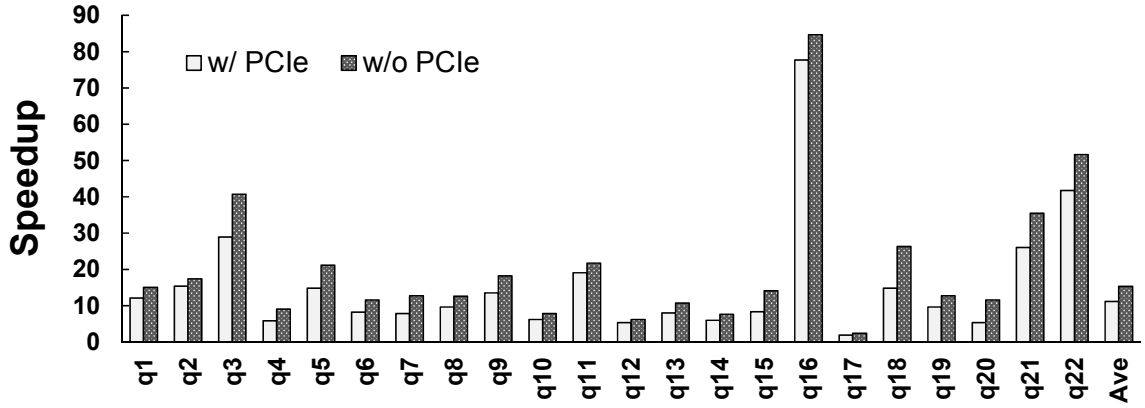

Figure 11. Comparison between Red Fox and parallel build of LogiQL (SF 1).

In Figure 11, the GPU performs better than the CPU for all queries. Q16 and Q22 have higher speedup in the GPU than the other queries. One common feature of these two queries that distinguishes them from the rest is that they concentrate on string processing on a large number of different short strings (less than or equal to 128 bytes). Q16 has three SELECTs performing regular expression searches (string notlike). Q22 focuses on substring and string matching. The CPU system utilizes the STL and BOOST library to perform the string operations. In the GPU, tuples are mapped to threads so that each thread performs the required string operation on one string. For example, in the case of regular expression search each thread performs the same search pattern upon its own string. Thus, the GPU threads may follow different code path depending on the string contents which

can vary a great deal. Severe branch and memory divergence are expected. The throughput of normal SELECTs for integer comparison is larger than 100GB/s. However, the throughputs of three SELECTs of Q16 are much smaller, i.e. 22GB/s, 17GB/s, and 5GB/s (the difference is caused by search pattern, string length, string content, etc.). Even then, Red Fox still outperforms the CPU system which is also limited by low branch prediction accuracy and CPU cache misses. Q17 has relatively smaller speedup which is less than 2x. The unoptimized query plan is the main reason. Section 3.3.3 will analyze the impact in more detail and discuss future improvements.

The last two bars in Figure 11 compare the TPC-H power metric between different execution configurations. The power metric for the parallel CPU system is 4,380 QphH@1GB. Red Fox is **11.20x** faster if including PCIe time or **15.35x** faster if just comparing the processor computation time. As to the TPC-H Price/Performance metric, the difference between the CPU and GPU would be 28.01x or 38.39x including or excluding PCIe. If only considering the cost of the processors, the GPU is 33.61x or 46.06x more cost efficient.

For completeness, Figure 12 includes the performance comparison between Red Fox and the *sequential* build of LogiQL 4.0 which runs one query on a single core of CPU because often database systems map one query to one thread when concurrently processing queries, i.e., in throughput optimized CPU designs. The last column of Table 4 is the raw performance data of the sequential CPU system. The CPU configuration of the sequential experiment is an Intel i7-920 with 12GB memory. Please note that the CPU is not as powerful as the server class CPU used in the parallel experiment. Overall on average, Red Fox is 114.00x faster with PCIe or 156.25x faster without PCIe.

3.3.3 Analysis and Future Improvement

The evaluation with small scale TPC-H queries demonstrates that the GPU is a credible algorithmic alternative to accelerate database workloads. This capability is the foundation of running large scale analysis. Moreover, the experience raises several improvement opportunities.

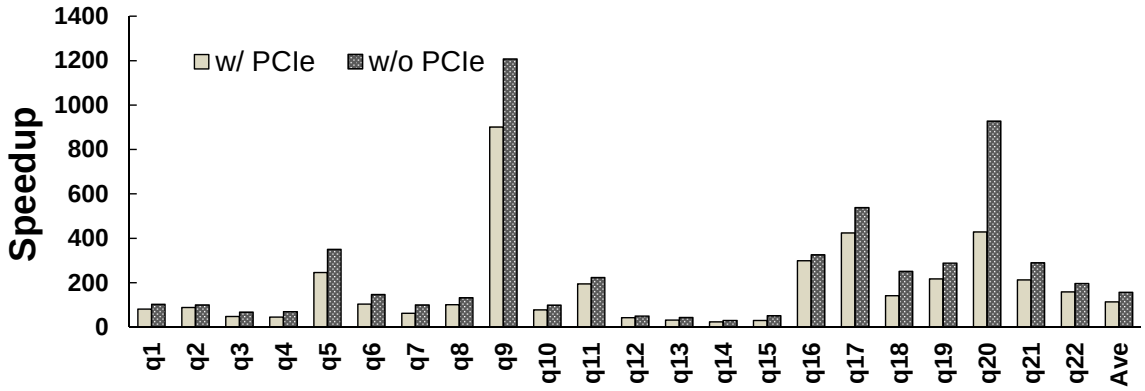

Figure 12. Comparison between Red Fox and sequential build of LogiQL (SF 1).

Red Fox currently does not include some standard optimizations utilized in the database community. Simply reordering the positions of operators in the query plan can significantly improve the performance. For example consider optimizations manually applied to the query plan in Figure 9(a) and shown in Figure 9(b). This was performed as follows.

1. move the SELECT operator to the beginning of the query;

2. perform the aggregations as soon as all the data are ready and discard the data immediately after aggregation.

The result is that the memory footprint of Q1 was reduced by half (due to reduction in size of intermediate data). and the execution time excluding PCIe transfer becomes 1.8X faster than the original.

Moreover, SORT is the most time consuming operator that needs to be further optimized. To reduce its cost, first grouping the JOIN operations by key attribute can minimize the number of intervening SORTs required. Second, some primitive implementations do not require pre-sorting such as the hash join operator. Third, the multi-predicate join algorithm as shown in Chapter 5 no longer needs to sort the data in the middle of the query execution.

More broadly, several hardware and software issues that are related to performance are discussed as below.

GPU DRAM System: RA operators are already memory bound in the GPU so that increasing the memory bandwidth can directly improve the performance. As to DRAM latency, it is less important than capacity or bandwidth because the contiguous tuple storage and the large amount of data make it relatively easy to hide.

SMX Microarchitecture: The instruction mix in database primitives is comprised mainly of integer and load/store instructions with relatively smaller percentage of floating point operations. Control and memory divergence occurs when searching or comparing data but their cost is not as significant as moving data. The goal of increasing the occupancy is to saturate SMX utilization or DRAM bandwidth since the basic strategy of RA primitive design is to increase core utilization until SMXs are saturated or the operations are memory bound. Thus, the shared memory and register files should be large enough to buffer the computation data especially in the cases when intermediate tuple sizes become large. Consequently, some emphasis on reducing tuple size when feasible is important for achieving high occupancy for given shared memory and register file sizes.

Data movement: For every query, Red Fox needs to launch many CUDA kernels and these kernels use global memory to communicate. Thus, there is a great deal of data movement between kernels. Consider Q1 that reads/writes 20GB from/into GPU memory. Suppose the memory bandwidth is 200GB/s, transferring these data would take 0.1 seconds which is about 1/2 of the total GPU computation time. In relational queries, data movement cost is amplified by the relatively fine grained nature of RA operators. Employing variants of classical loop fusion optimizations to kernels or using multi-predicate join operations, can improve performance.

In this accelerator configuration, the GPU performs operations over partitioned data sets and logically appears as a faster core to the host runtime. However, the runtime must account for the cost of data transfer. Thus, intelligent workload partitioning schemes (CPU vs. GPU) will have to make the decisions as to when accelerator usage is productive as a function of query plan and input data set characteristics. With appropriate changes in cost,

the model applies to fused parts where the CPU and GPU share the same memory hierarchy such as in AMD Fusion or Intel's Haswell. Finally, another approach to addressing the memory limitation of discrete parts is the use of global virtual memory (e.g., CUDA UVM). In this case, data movement is not managed by the application programmer, but rather by the system software/compiler.

3.4 Summary

This chapter presents the design of a GPU compiler/runtime framework for a high level declarative language commonly used for database and business analytics applications. The context is that of a cloud system where individual nodes are accelerated with GPUs and where the runtime system is targeted to multicore execution of queries over partitioned data sets and uses the GPU logically as a high speed accelerator core. This paper focuses on the compilation of industrial strength queries represented by the full TPC-H benchmark onto GPUs. Comparison with multithreaded host implementations demonstrates significant computing speedup is feasible for a declarative programming model for database and business analytics. The language is progressively parsed and lowered through a series of RA representations that eventually are mapped to PTX kernels embedded in the Harmony IR that is executed by an implementation of the runtime on a high-end discrete GPUs. The performance on the full set of TPC-H queries is reported which to the best of my knowledge is the first such implementation for GPUs. Analysis of the performance, the lessons learned and future directions are also provided at last.

CHAPTER 4

DATA MOVEMENT OPTIMIZATION

This chapter introduces how to aggregate the computation from different primitives together to reduce the costly data movement overhead between them. Chapter 3 introduces a system that can map the relational computations to GPUs. Its performance relies on the the individual implementation of the primitives comprising the query. With careful implementation, these primitives which have low operation density can be pushed to be memory bound in GPUs if the data is already in the GPU memory or PCIe bound if the data start from the host memory. This is true even for the most complex primitive, relational join. However, these primitives from the same query are separated, i.e. each primitive loads the input from the (device or host) memory in the beginning and stores the result into the memory. The input data might be generated by the earlier primitive and similarly the result data might be used by the later primitives. These round trip data movements can be optimized away if there is a good approach to cache the data closer to the processor than the memory.

As shown in Figure 13, internal to the GPU there exists a memory hierarchy that extends from GPU core registers, through on-chip shared memory, to off-chip global memory. However, the amount of memory directly attached to the GPUs (the off-chip global memory) is limited, forcing transfers from the next level which is the host memory that is accessed in most systems via PCIe channels. The peak bandwidth across PCIe can be up to an order of magnitude or more lower than GPU local memory bandwidth. Relational computation applications must stage and move data throughout this hierarchy. Consequently there is a need for techniques to optimize the implementations of relational applications considering both the GPU computation capabilities and system memory hierarchy limitations.

This chapter proposes the Kernel Weaver optimization framework and demonstrates the impact of kernel fusion for optimizing data movement in patterns of interacting operators

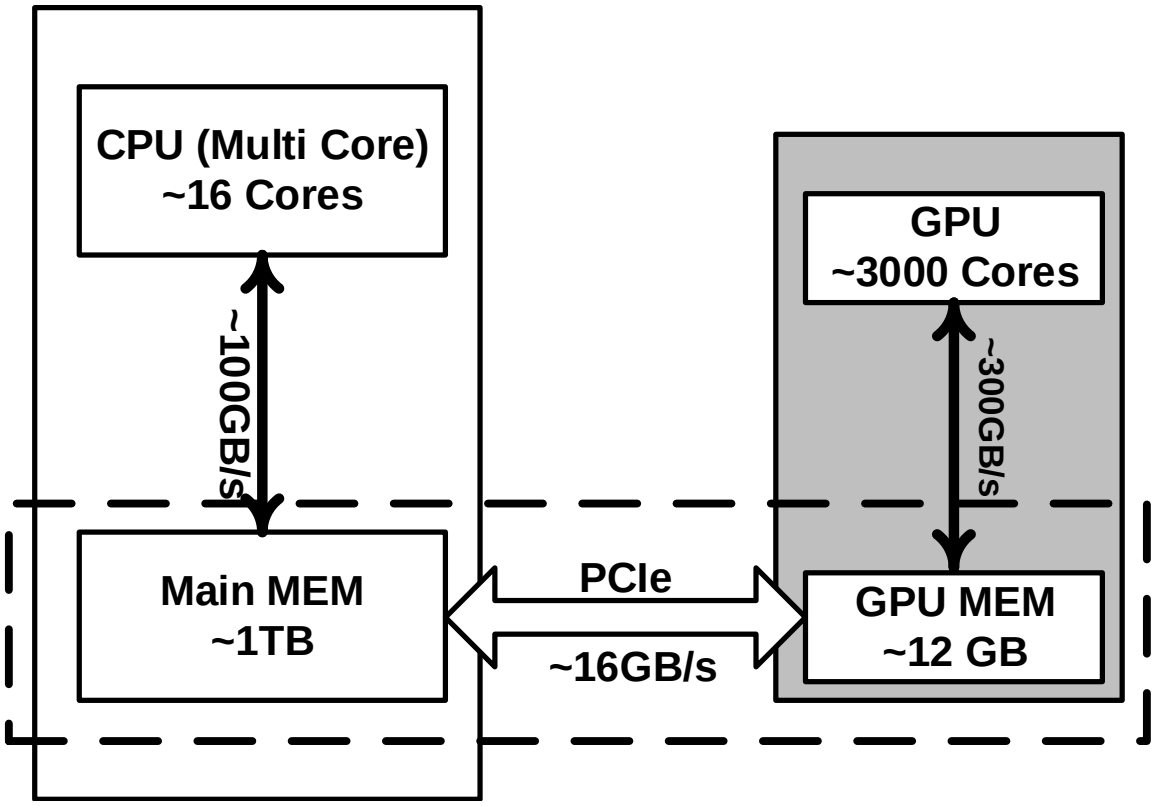

Figure 13. Memory hierarchy bottlenecks for GPU accelerators.

such as those found in the TPC-H benchmark suite.

4.1 Benefits

The idea of GPU kernel fusion comes from classic loop fusion optimization. Basically, kernel fusion reduces data flow between two kernels (via the memory system) having consumer-producer dependence by merging them into one larger kernel. Therefore, its benefits goes far beyond reduction in PCIe traffic.

Figure 14 depicts an example of kernel fusion. Figure 14(a) shows two dependent kernels - one for addition and one for subtraction. After fusion, a single functionally equivalent new kernel (Figure 14(b)) is created. The new kernel directly reads in three inputs and produces the same result without generating any intermediate data.

Kernel Fusion has six benefits as listed below (Figure 15). The first four stem from creating a smaller data footprint through fusion since it is unnecessary to store temporary

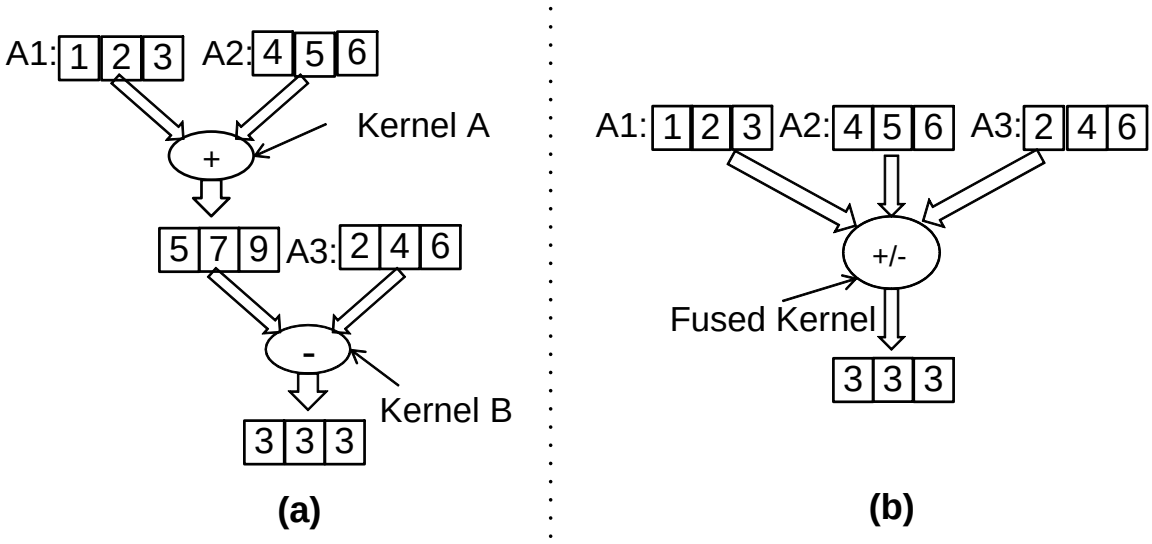

Figure 14. Example of kernel fusion.

intermediate data in global memory after each kernel execution, while the other two relate to increasing the compiler's optimization scope.

Smaller Data Footprint results in the following benefits:

- Reduction in PCIe Traffic: In the absence of fusion, if the intermediate data is larger than the relatively small GPU memory, or if its size precludes storing other required data, the intermediate data will have to be transferred back to the CPU for temporary storage incurring significant data transfer performance overheads (Figure 15(a)). For example, if kernels generating *A3* in Figure 14(a) need most of the GPU memory, the result of the addition has to be transferred to the CPU memory, and subsequently transferred back to the GPU before the subtraction can be executed. Fusion avoids this extra round trip data movement.

- Larger Input Data: Since kernel fusion reduces intermediate data thereby freeing GPU memory, larger data sets can be processed on the GPU which can lead to a smaller number of overall transfers between the GPU and CPU (Figure 15(b)). This benefit grows as the application working set size grows.

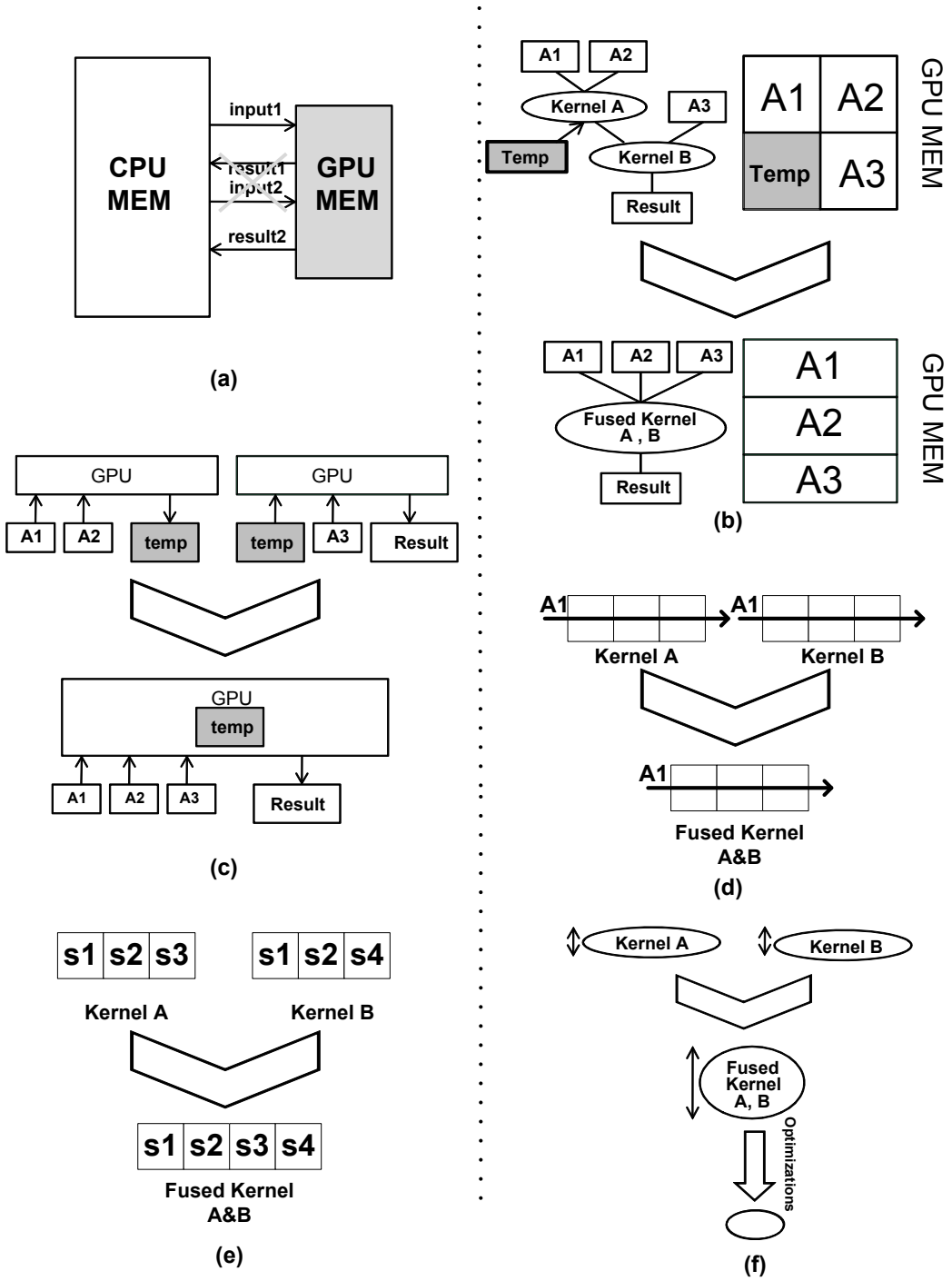

Figure 15. Benefits of kernel fusion: (a) reduce data transfer; (b) store more input data; (c) less GPU Memory access; (d) improve temporal locality; (e) eliminate common stages; (f) larger compiler optimization scope

- Temporal Data Locality: As in traditional loop fusion, access to common data structures across kernels expose and increase temporal data locality. For example, fusion

can reduce array traversal overhead and improve the cache performance when the array is accessed in both kernels (Figure 15(c)). Kernels that are not fused may have to access the GPU memory again if the data revisited across kernels is flushed from the cache.

- Reduction in Memory Accesses: Fusing data dependent (producer-consumer) kernels enables storage of intermediate data in registers or GPU shared memory (or cache) instead of global memory (Figure 15(d)). Moreover, kernels which are not fused have a larger cache footprint necessitating more off-chip memory access.

Larger Optimization Scope brings two benefits:

- Common Computation Elimination: When two kernels are fused, the common stages of computations are redundant and can be avoided. For example, the original two kernels in Figure 15(e) both have stages S1 and S2 which need to be executed only once after fusion.

- Improved Compiler Optimization Benefits: When two kernels are fused, the textual scope of many compiler optimizations are increased bringing greater benefits than when applied to each kernel individually.

  Table 5 compares the speedup of using the *O3* flag to optimize before and after fusion for a very simple, illustrative example. Without fusion, the two filter operations are performed separately in their own kernels (row 1, column 2). After fusion the two statements occur in the same kernel and are subject to optimization (row 2, column 2). The third and fourth columns show the number of corresponding PTX instructions produced by the compiler when using different optimization flags. Before optimization, the fused kernel has 5 more instructions than without fusion (10 vs. 5). Using compiler optimizations without fusion can reduce **40%** instruction count (from 5 to 3), while optimizing a fused kernel achieves a higher **70%** instruction reduction (10

down to 3). This simple example indicates that significant reduction in instruction counts are possible when applied to larger code segments.

Table 5. The impact of kernel fusion on compiler optimization

	Statement	Inst # (O0)	Inst # (O3)
not fused	if (d<THRESHOLD1) if (d<THRESHOLD2)	5×2	3×2
fused	if (d<THREASHOLD1 && d<THREADSHOLD2)	10	3

These benefits are especially useful for relational queries since RA operators are fine grained and exhibit low operation density (ops per byte transferred from memory). Fusion naturally improves operator density and hence performance.

Figure 16 is a simple example comparing the GPU computation throughput of back-to-back SELECTs (its implementation is briefly introduced in Section 4.2) with and without kernel fusion. Inputs are randomly generated 32-bit integers, the x-axis is the problem size which fits GPU memory, and kernels were manually fused in this example. On average, fusing two SELECTs achieves **1.80x** larger throughput while fusing three kernels achieves **2.35x**. Fusing three SELECTs is better since more redundant data movement is avoided and larger code bodies are created for optimization.

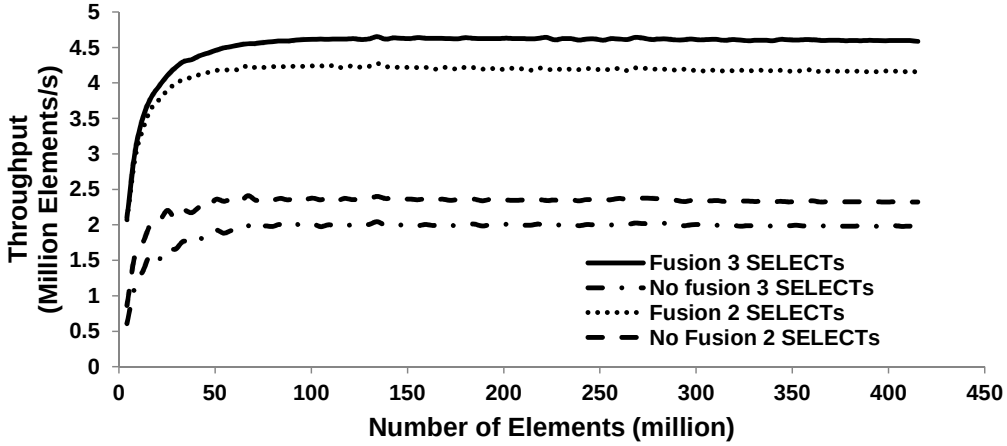

Figure 16. Performance Comparison between fused and independent SELECTs.

Intel had Sandy (and Ivy) Bridge architectures and AMD brought Fusion APUs to the market. Both designs put the CPU and GPU on the same die and removed the PCIe bus. In these systems, four out of the six benefits listed above still apply (excluding Reduction in PCIe Traffic and Larger Input Data). Thus, kernel fusion is still valuable.

While a programmer could perform a fusion transformation manually, database queries are typically supplied in a high level language like Datalog or SQL, from which lower-level operations are synthesized using a query planner and compiler. Automating this process as a compilation transformation is necessary to make GPUs accessible and useful to the broader community of business analysts and database experts. Moreover, running kernel fusion dynamically in a JIT creates opportunities to leverage runtime information for more effective optimizations.

4.2 System Overview

As illustrated in Figure 2, Kernel Weaver is implemented as an optimization module inside the RA-Harmony compiler of Red Fox which can automatically generate fused Harmony kernels for the runtime to launch. Note that each RA operator may be implemented as several CUDA kernels so that fusing operators requires coordinated fusion of several CUDA kernels.

Kernel fusion is based on the multi-stage formulation of algorithms for the RA operators. Multi-stage algorithms are common to sorting [77], pattern matching [78], algebraic multi-grid solvers [79], or compression [80]. This formulation is popular for GPU algorithms in particular since it enables one to separate the structured components of the algorithm from the irregular or unstructured components. This can lead to good scaling and performance. Kernel fusion can now be explained as a process of weaving (mixing, fusing, and reorganizing) stages from different operators to generate new optimized operator implementations.

The high level description of the order and functionality of the stages will be referred

as an *algorithm skeleton*. In this chapter the algorithm skeletons are from Diamos et al. which are also used by Red Fox. These algorithms store relations as a densely packed array of tuples with strict weak-ordering as shown in Figure 7. All RA operator skeletons are comprised of three major stages, partition, compute and gather. The following briefly describes the functionality of each stage using the implementation of a simple operator - SELECT (Figure 17) - as an example.

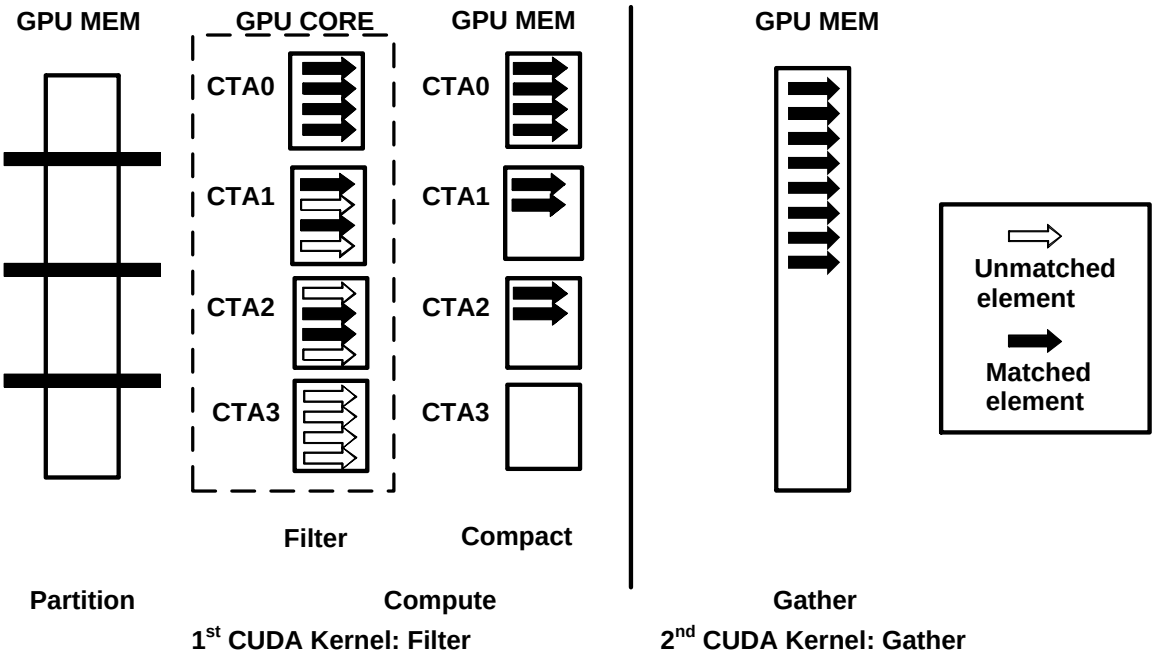

Figure 17. Example algorithm for SELECT

Partition: The input relations are partitioned into independent sections that are processed in parallel by different CTAs. For unary operators such as SELECT in Figure 17 the input relations can be evenly partitioned to balance the workload across CTAs. Binary operators such as JOIN and SET INTERSECTION are more complex in this stage since they need to partition both inputs and partitioning is based on a key value consequently producing unbalanced sizes of inputs to CTAs and resulting in unbalanced compute loads.

Compute: A function for each RA operator is applied to its partition of the inputs to generate independent results. Different RA operators are specialized to effectively utilize fine-grained data parallelism and the multi-level memory hierarchy of the GPU to maximize

performance. For example, the SELECT in Figure 17 first *filters* every element in parallel and then leverages the shared memory to *compact* [81] the filtered result in preparation for creating a contiguous output.

Gather: The results computed in individual partitions are gathered into a global dense sorted array of tuples by using a coalesced memory-to-memory copy.

Multi-stage RA operators are implemented as multiple CUDA kernels - typically one per stage. Kernel weaver fuses operators by interleaving stages and then fusing interleaved stages (their respective CUDA implementations) to produce a multi-stage implementation of the fused operator. A variety of alternative implementations can be used for the implementation of each stage and can be accommodated by the operator fusion process.

4.3 Automating Fusion

This section introduces the process of kernel fusion employed in Kernel Weaver. For simplicity, the initial description is based on each operator being implemented as a single data parallel kernel. Subsequently, higher performance multi-stage implementations of the RA operators will be described.

Three main steps to fuse operators are:

1. Using compiler analysis to find all groups of operators that can be fused.
2. Selecting candidates to fuse
3. Performing fusion and generating code for the fused operators.

4.3.1 Criteria for Kernel Fusion

The simple idea is to take two kernels say with 4096 threads each, and produce a single kernel with 4096 threads, where each thread is the result of fusing two corresponding threads in the individual kernels. Clearly, the data flow between the two fused threads must be correctly preserved. The classification below can be understood from the perspective of preserving this simple model of kernel fusion. The first consideration is finding feasible

combinations of data parallel kernels to fuse via compiler analysis, followed by the selection of the best options. To reduce the data movement overhead between kernels, two types of criteria for fusion of candidate kernels are that they possess

1. Same kernel configuration (CTA dimensions and thread dimensions).

2. Producer-consumer dependence.

The first criteria is similar to loop fusion that requires compatible loop headers (same iteration number, may need loop peeling to pre-transform the loop, etc.). Kernel fusion also requires compatibility between kernel parameters. The fused kernel will have the same kernel configuration as the candidates. The data parallel nature of RA operators make their implementation independent (with respect to correctness) of the kernel configuration. Thus, while too many or too few CTAs or threads may lead to inefficient use of resources, fusion can be performed correctly if the kernel configurations are the same. This work tests a set of micro-benchmarks (see Section 4.4) with a wide range of combinations of CTA dimensions and thread dimensions and picks one pair that works best in most cases.

The second criteria is due to the fact that the benefits listed in Section 4.1 are derived primarily from exploiting producer-consumer dependencies. Data dependence analysis is necessary to find candidate kernels. Producer-consumer dependence between two data parallel kernels can be classified into three categories as shown in Figure 18: *thread, CTA and kernel dependence*.

In the first category of *thread dependence*, each thread of the consumer kernel only consumes data generated by a single thread from the producer kernel. Figure 18(a) illustrates such an example with tuples containing two attributes, e.g., (1,T). Dependencies between producer and consumer kernels corresponding to unary RA operators such as SELECT and PROJECT, belong to this category because the operation on one input tuple is independent of the operation performed on any neighboring tuple. In this case, corresponding producer

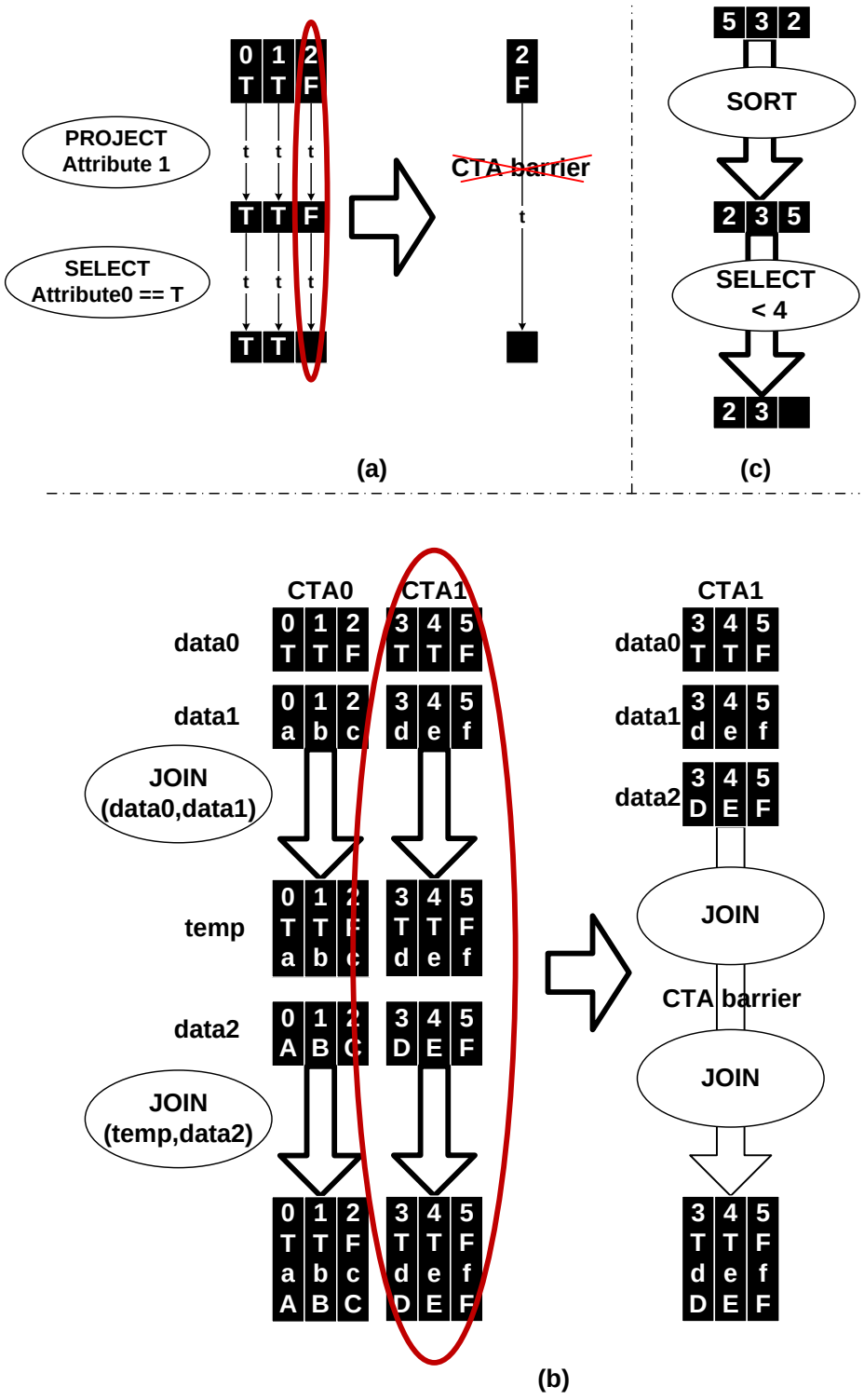

Figure 18. Example of three kinds of dependence: (a) thread dependence; (b) CTA dependence; (c) kernel dependence.

and consumer threads from each kernel can be fused without having to insert synchronization operations. This type of producer-consumer dependence between kernels is referred to as *thread dependence*.

The second category is wherein every CTA of the consumer kernel depends on the completion of a CTA of the producer kernel. Such dependencies are referred to as *CTA dependencies*. For example, this occurs between binary RA operators such as JOIN and SET INTERSECT that have a producer-consumer dependence. Consider, Figure 18(b) that illustrates a producer-consumer dependence between two JOIN operators. The first operator performs a JOIN operation across tuples from two input data sets, *data0*, and *data1*. Each CTA is provided a partition of input tuples, corresponding to some range of the key value used in the JOIN (in this example each tuple has a unique key value which is an integer). Thus, a thread in a CTA must compare the key values of tuples it is processing with the key values of tuples being processed by every other thread in the CTA, *and only within the CTA*. While such a partitioning of input tuples across CTAs produces unbalanced loads between CTAs, data dependencies between threads are confined to remain within a CTA. The producer CTA writes its tuples to shared memory where a CTA from the consumer kernel can now pick it up. A barrier synchronization is necessary after the producer operator before the consumer operator can start and corresponding threads from producer-consumer CTAs can be fused with appropriately placed barriers.

The third category is wherein the consumer kernel has to wait for the completion of all threads in the producer kernel, i.e., kernel fusion is not feasible. A typical example is where the producer kernel is a SORT operator (Figure 18(c)). The reasons are

1. It cannot be launched until all inputs arrive.

2. SORT shuffles all data and the following consumer operators need to wait for its completion before being able to start streaming data.

Such dependence is referred to as *kernel dependence*.

Note the three categories of dependencies are from the perspective of being able to fuse kernels by fusing corresponding threads within producer-consumer kernels. Accordingly, the dependencies are implicitly associated with the level of the memory hierarchy used to pass data. Fused threads across thread dependent kernels use the register file which is allocated by the thread. Fused threads across CTA dependent kernels use shared memory which is allocated by the CTA. Finally, according to the above classification, the kernels in a dependence graph that are candidates for kernel fusion only exhibit thread or CTA dependencies with other kernels, and are bounded by operators with kernel dependencies. Algorithm 1 formalizes the steps to find kernel fusion candidates. Its main idea is first removing operators causing kernel dependence from the graph and then finding the rest connected operators.

```
Input: a list of operators op
Output: a list of fusion candidate groups c
i = 0;
length = size of list op;
Topologically sort op;
while i ≠ length do
    class = classify dependence between op[i] and
            its predecessors and successors;
    if class == Kernel Dependence then
        delete op[i];
    end
    i = i + 1;
end
c = all connected subgraphs of op;
```

Algorithm 1: Searching for fusion candidates.

In Red Fox, the query plan generated by the language front-end consists of RA operators and their associated variables. This information is used to construct an RA dependence graph like the one shown in Figure 19(b). The nodes in the graph represent RA operators and the directional edges identify nodes with the producer-consumer dependencies. Candidate kernels meeting the above two criteria will be marked for the next process. The large circle bounded by SORT operators contains operators satisfying the dependence

requirement and are candidates for fusion. Instances supporting recursive queries (e.g. ancestor(a,c)←parent(a,b),ancestor(b,c)) may generate a dependence graph with enclosed loops. This work only considers acyclic graphs although often loop unrolling and related known optimizations can create acyclic dependence graphs.

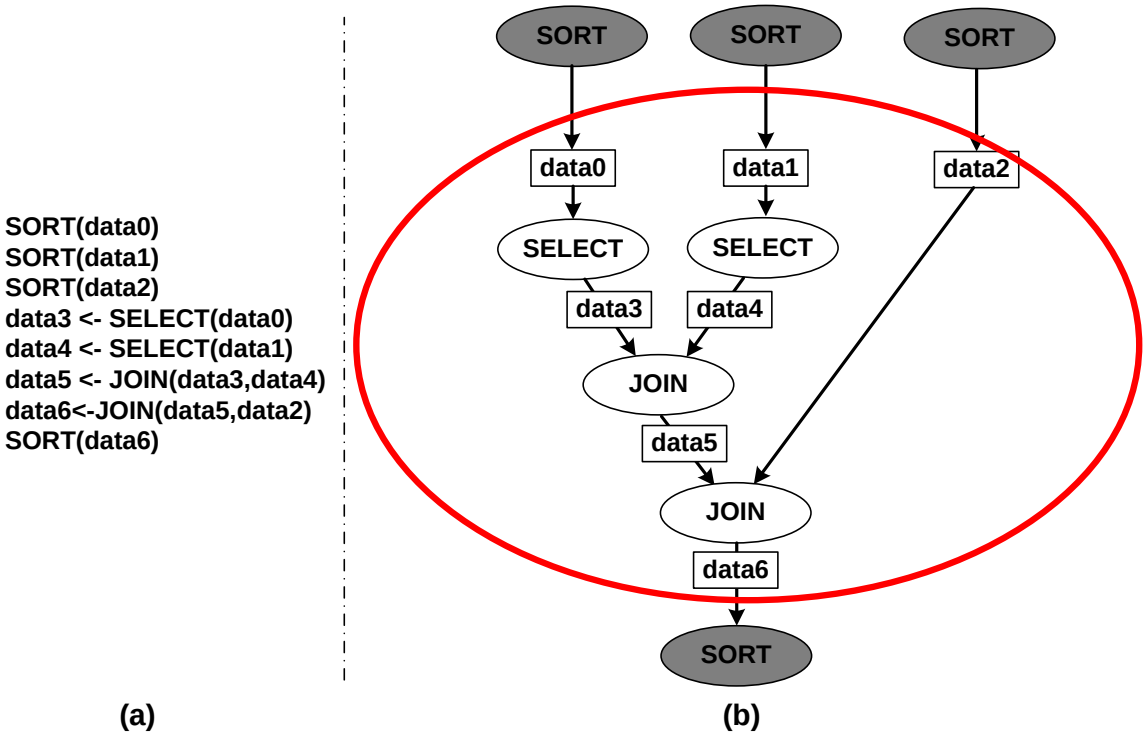

Figure 19. Example of constructing a dependence graph: (a) database program; (b) dependence graph.

4.3.2 Choosing Operators to Fuse

Fusing all the kernels meeting the criteria may not be practical. The main constraint on fusion is resource constraints - pressure on limited registers and limited amount of shared memory available within each stream multiprocessor. Fusion choices must also be ordered based on dependencies and performance impact. Accordingly candidate kernels are topologically sorted to honor the dependence requirement and a greedy heuristic is used to pick up kernels satisfying constraints. Algorithm 2 is the heuristic algorithm.

The heuristic algorithm searches for the longest contiguous sequence of operators that

```
Input: a list of candidate operators op
Input: resource budget b
Output: a list of fusion groups f
i = 0;
j = 0;
length = size of list op;
Topologically sort op;
while i ≠ length do
    add op[i] to f[j];
    cost = resource usage estimation of f[j];
    if cost > b then
        delete op[i] from f[j];
        j = j+1;
    else
        i = i + 1;
    end
end
```

Algorithm 2: Choosing operators to fuse.

can be fused, within resource constraints, i.e., fits within the shared memory and registers budgeted for each CTA. The intuition underlying the above method is that it is more important to fuse operators executed earlier than those executed later. The reason is that relational queries normally process large amounts of data. After several filtering and reduction operators, the data set is reduced significantly. Resource permitting, fusing the first few operators in the dependency graph provides the most benefit.

Figure 20 is an example that shows how the greedy heuristic of Algorithm 2 works. It starts from the candidates circled in Figure 19(b). Figure 20(a) first performs a topological sort on the dependence graph to produce a list of operators. If operators execute in this order, all dependencies will be honored. Starting from the top of the list, Figure 20(b) searches for the longest contiguous sequence of operators that can be fused, within resource constraints, i.e., fits within the shared memory and registers budgeted for each CTA (*data3* and *data4* become internal to the fused operator). In the example in Figure 20(c) the second JOIN cannot be added since the estimated shared memory resource usage is larger than the budget. The algorithm repeats the above process for the next operator, the second

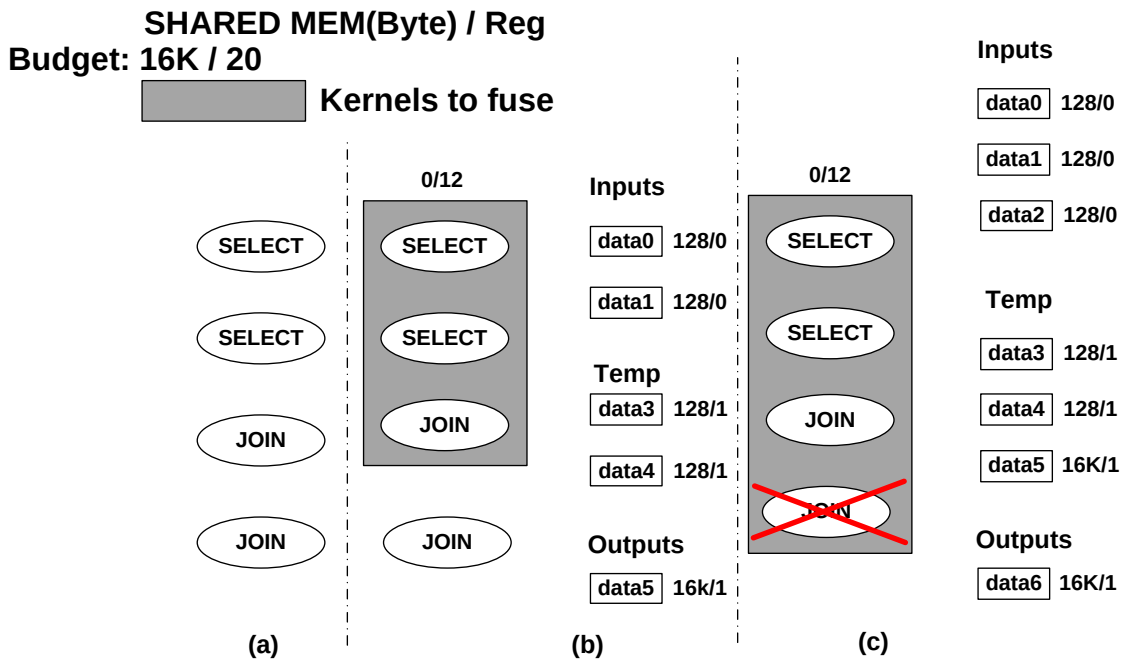

Figure 20. Example of choosing operators to fuse: (a) topologically sorted operators; (b) choose the first three operators to fuse; (c) refuse to fuse the fourth operator.

JOIN, until no more operators can be fused. Resource usage estimation is discussed in Section 4.3.3.3 after introducing code generation.

4.3.3 Kernel Weaving and Fusion

Given the dependence graph and candidate operators to fuse, the final step is performing the fusion. Recall that each operator is implemented as a multi-stage algorithm with three stages - *partition*, *computation*, and *gathering* - each of which is implemented as a CUDA data parallel kernel. The fused operator still has these three stages.

At a high level, fusion is achieved by two main steps:

1. Grouping the *partition*, *computation*, and *gathering* stages of the operators together (which is also referred to as interleaving).

2. Fusing the individual stages.

In other words, the *partition* stages of the candidate operators are fused together into a single data parallel kernel, which could be viewed as the *partition* stage for the newly

fused operator. Similarly, the *computation* and *gathering* stages are fused into a single fused *computation* and *gathering* stage respectively. For example, when two operators are fused, the fused operator will have the multi-stage structure shown in Figure 21 where the two compute stages are fused into one data parallel kernel (the fused partition and fused gather stages similarly represent fusion of individual partition and gather stages). The fused computation stage performs the computation stage of the original operators in the order of their dependencies. All intermediate data and data sizes are stored in the shared memory or registers. The fused operator may have multiple inputs and outputs.

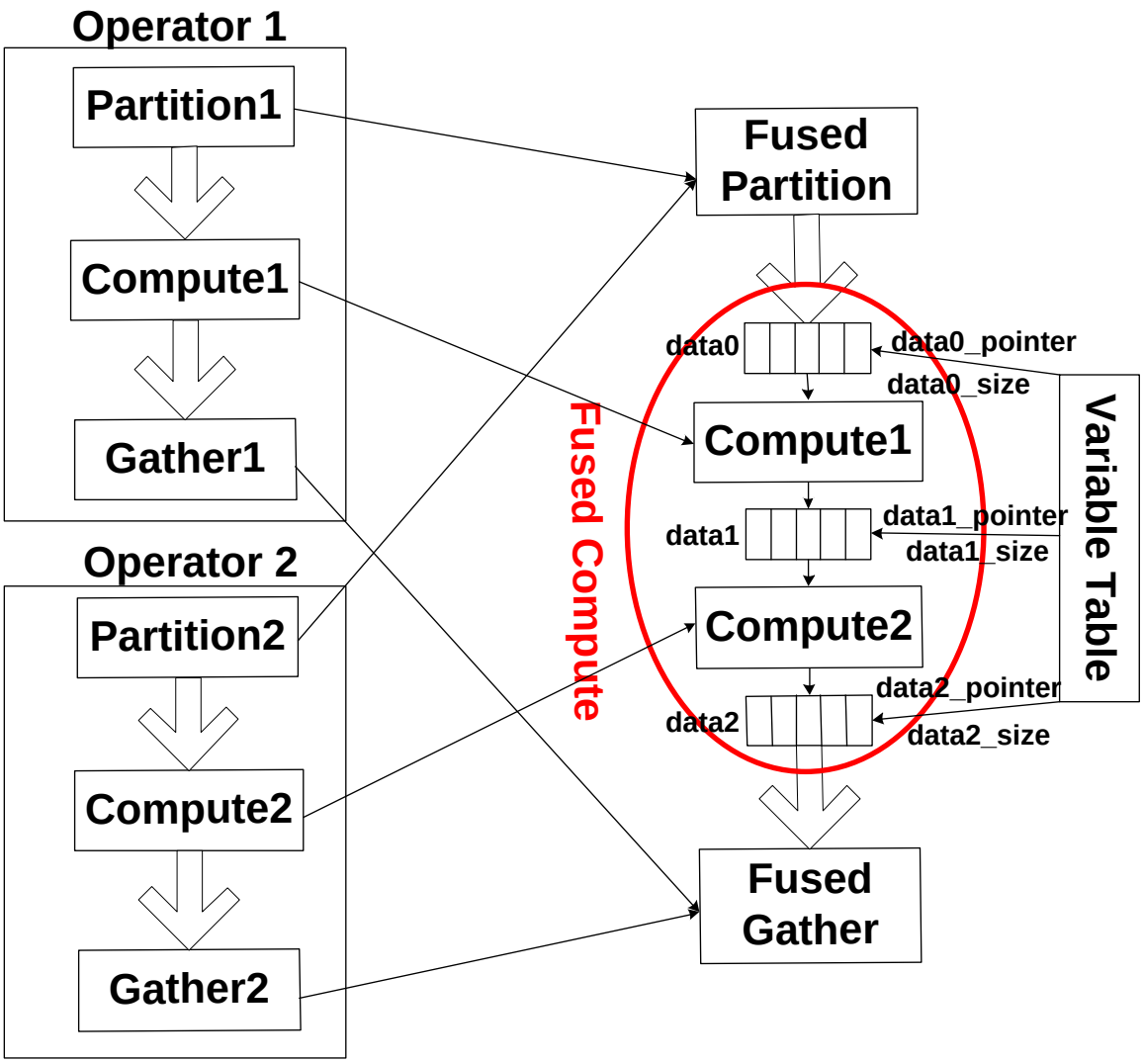

Figure 21. The structure of generated code (fusing two operators).

The above fusion process includes code generation for the fused operators. Code generation takes as input a description of a topologically sorted set of operators to be fused and their associated variables, and produces CUDA code for the data parallel kernels that implement the fused operator. The CUDA code is generated by concatenating the instantiated algorithm skeleton code of each stage, and connecting the outputs of one stage to the inputs of the next stage. A variable table, which records and tracks the use of variables between stages, is needed to instantiate the skeleton. Figure 21 shows how the variable table tracks the variables that hold result data and result size of each computation stage.

Fusing operators depends on whether thread dependence or CTA dependence exists between operators. We now describe in more detail how to fuse operators with thread and CTA dependencies.

4.3.3.1 Fusing Thread Dependent Only Operators

Unary operators SELECT and PROJECT exhibit thread dependence. The kernel configuration (CTA and grid dimensions) of both operators are equal. Therefore each thread in the producer operator is fused with a corresponding thread in the consumer operator.

The partition stage of the fused operator remains the same as that of the producer operator. The compute stage of the fused operator is a data parallel kernel with the same kernel configuration, where each thread is created as follows. Every thread first loads a tuple from its input partition into registers. The computation of corresponding producer and consumer threads are performed using these registers, i.e., SELECT or PROJECT in the correct order. These operators either discard data (SELECT) or discard attributes (PROJECT). The output of this sequence of operations is compacted into an output array. The gather stage accumulates all of the data from different threads in the fused compute stage into contiguous memory.

As shown in Figure 17, the computation stage of SELECT has two parts, filter and stream compaction. After kernel fusion, stream compaction is needed only when the SELECT result should be copied to GPU memory. Figure 22 is an example of fusing two

back-to-back SELECTs. Compared with Figure 17, only one filter operation is added. Moreover, the fused kernel only needs to read and write memory once rather than twice.

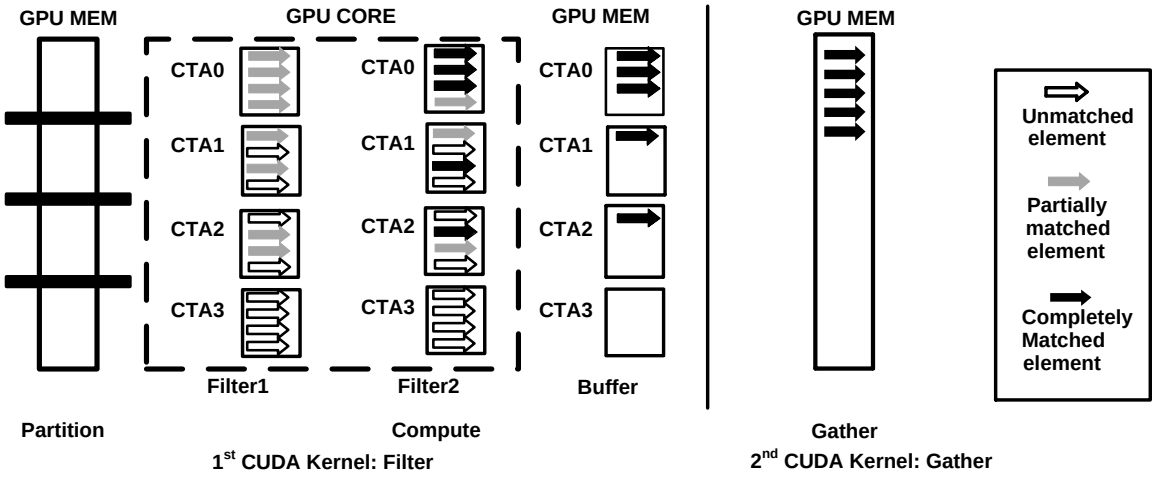

Figure 22. Example of fusing two SELECTs.

For PROJECT, its result tuple should be stored into a new register with a different data type since it contains less attributes. Thus, the operations after PROJECT have to use this new register instead.

4.3.3.2 Fusing CTA and Thread Dependent Operators

Binary relational operators are CTA dependent. This change increases the number of inputs of the fused operators and necessitates the following main distinctions in code generation:

1. Use binary search to partition inputs;
2. Use shared memory to support CTA dependence;
3. Synchronize two operators having CTA dependence.

Thus, code generation has to be extended to support the three differences. Figure 23 shows the generated code for the operators in Figure 19(b) and is used as example to explain the extensions.

The adapted approach is to maintain the independent operation of each CTA to be able to fuse corresponding CTAs from the producer and consumer operators. This is achieved

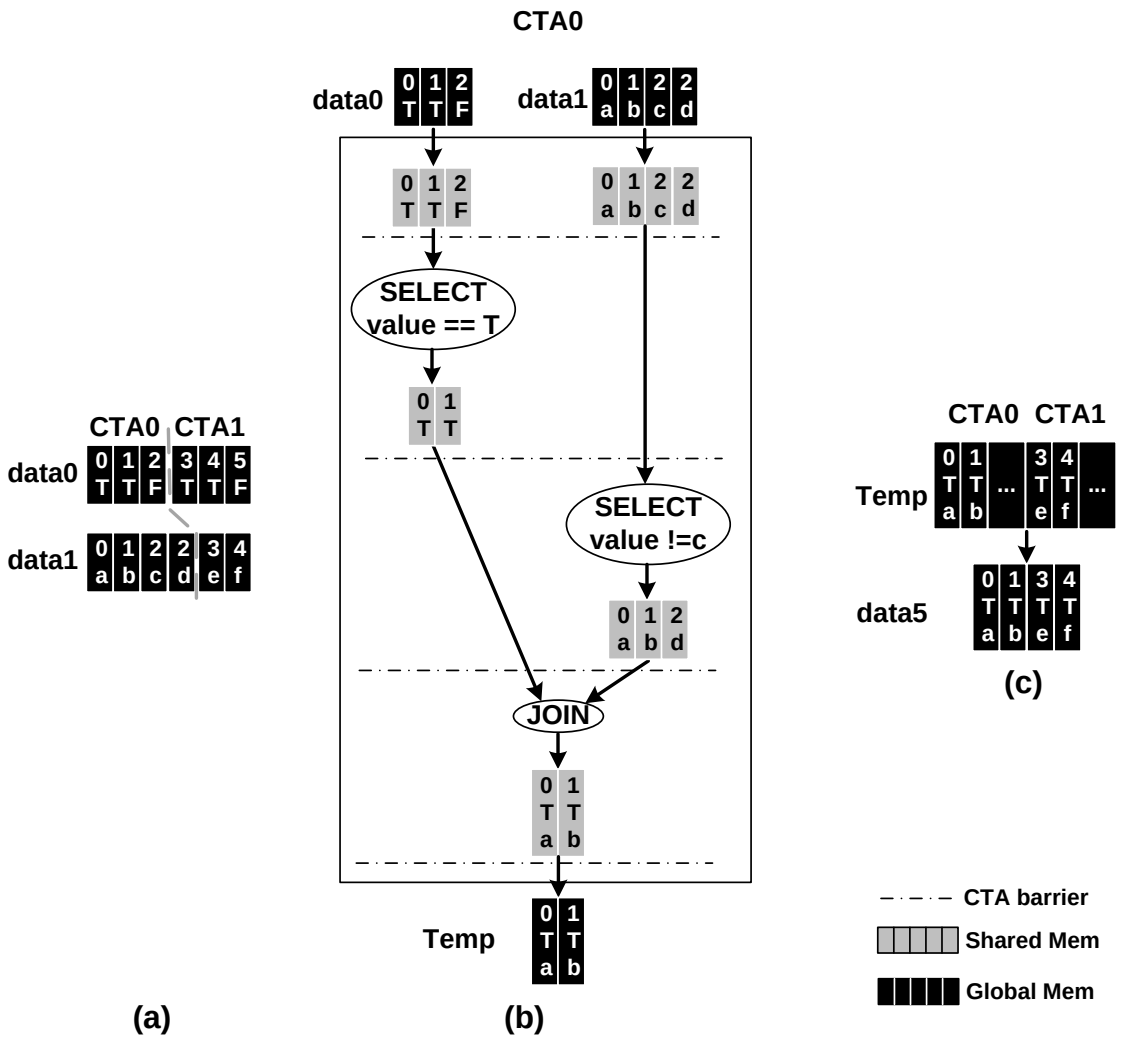

Figure 23. Example of Generated Code of Figure 20(b): (a) Partition two inputs; (b) Computation of one CTA; (c) Gather one output.

in the *partition* stage by partitioning the input set by key values. Each CTA then receives a set of tuples corresponding to a specific range of key value pairs. This is achieved using binary search [82] and both inputs of each binary operator are partitioned across CTAs. For example, in Figure 23(a), *data0* is first evenly partitioned into two parts bounded by pivot tuples. Then, a binary search is used to lookup the tuples in *data1* corresponding to the *key* attributes of *data0* pivot tuples. The partitioned data sizes of the two inputs provided to each CTA thus may differ (e.g. *data1*). However, when fusing two binary operators (e.g., two JOIN operators), three inputs need to be partitioned and each operator may use

different keys. For instance, one JOIN may use the first 2 attributes as a *key* and the other JOIN may only use the first attribute as *key*. In this case, the fused input stage will only use the first attribute as *key*. This preserves the independence of operation across CTAs.

Figure 23(b) is an example of the computation stage of one CTA. The other CTA works in exactly the same way but upon different data. In the beginning, each CTA first allocates a software controlled cache in shared memory for each input and then loads data into the cache (e.g. CTA0 loads in a portion of corresponding *data0* and *data1* divided as in Figure 23(a)). Afterwards, a CTA-wise fused computation performs fused operations upon cached data. Within a CTA, the generated code can perform all supported operations such as SELECT and JOIN. If two connected operators have CTA dependence (e.g. between SELECT and JOIN), the result data of the producer operator should be stored in a cache allocated in the shared memory, and the result size is stored in a register. To guarantee all threads within a CTA finish updating the cache, a CTA barrier synchronization is needed after the producer operation. If two dependent operators only exhibit thread dependence, they only need to use register(s) to pass value(s) and no synchronization is necessary. For example, the first operator in Figure 23 to execute is SELECT and it has CTA dependence relationship with its consumer JOIN. Thus, SELECT has to store its result in shared memory rather than the register. The second SELECT is handled in the same way. Thus, the inputs of JOIN all reside in the shared memory before its execution. After JOIN, the result is moved to GPU global memory.

The gather stage (Figure 23(c)) is the same as in the thread dependent only cases which packs the useful results generated by two CTAs into an output array.

4.3.3.3 Resource Usage

Code generation determines resource occupancy. As shown in Figure 20(c), some resources are used to store input, output, and intermediate temporary data. Others are used inside the computation.

Fusing thread dependent operators stores intermediate data in the registers. The number

of needed registers depends on the data type of the tuple which is provided by the database front-end. Fusing CTA dependent operators stores temporary data in the shared memory and temporary data size value in one register. Allocated shared memory size is a function of data type, input data size and operator type. For example, SET INTERSECT needs to allocate $\min(input1, input2)$ tuples for its output. The data variable and data size variable stored in registers are live until they are no longer needed.

Registers are also needed to perform partition, computation, and gather. The partition result, the beginning and the end position of all inputs, uses variables to pass to the computation stage. The liveness of the variables used inside each stage is the same as the scope of the stage. Thus, different variables of different stages can reuse the same registers. Therefore, the register usage of a fused operator is less than or equal to the maximum of the register usage in each stage plus the registers used to pass values between stages. The registers used by each stage can be determined as long as the data types of all tuples are known.

4.3.4 Extensions

The preceding three sections discussed how code is generated for RA operators having producer-consumer dependence. This method can be extended to support other dependence types or other operators.

The first extension is to support input dependence, i.e. operators shares the same inputs. The benefits of fusing these operators is that the input data shared by different operators only need to be loaded once, which is not as important as the case of producer-consumer dependence. Fusing operators having input dependence also increases the resource pressure. The modification to the above process is to detect input dependence when constructing the dependence graph. The code generation part can remain the same.

The second extension is to support simple arithmetic operations such as addition, subtraction, multiplication and division. These arithmetic operators are much simpler than RA operators. They have two inputs, but use even partitions to divide both inputs. The

dependence between them belongs to thread dependence and can use registers to store computation results.

4.4 Experimental Evaluation

Table 6 shows the experimental infrastructure. This section uses the old version of Red Fox to generate GPU binaries. All primitive algorithms used here were designed by Diamos et al.

Table 6. Kernel Weaver experimental environment.

CPU	2 quad-core Xeon E5520 @ 2.27GHz
Memory	48 GB
GPU	1 Tesla C2070 (6GB GDDR5 memory)
OS	Ubuntu 10.04 Server
GCC	4.4.3
NVCC	4.0

4.4.1 Micro-benchmarks

TPC-H queries are analyzed and some commonly occurring combinations of operators are identified that are potential candidates for fusion. From the 22 queries in TPC-H, Figure 24 illustrates some frequently occurring patterns of operators corresponding to different cases discussed in Section 4.3. In the figure, (a) is a sequence of back-to-back SELECT operators that perform filtering, for instance, of a date range. It only has thread dependence. (b) is a sequence of JOIN operations that creates a large table with multiple attributes, and exhibits CTA dependence. (c) corresponds to the JOIN of three small selected tables and has both thread and CTA dependence. (d) represents the case when different SELECT operators need to filter the same input data and has input dependence. (e) performs arithmetic computations such as $price \times (1 - discount) \times (1 + tax)$ which appears in several TPC-H queries. The PROJECTs in the figure discard their sources and only retain part of the result.

The above patterns can be further combined to form larger patterns that can be fused. For example, (a) and (b) can be combined to form (c).

In the following experiments, the tuple used in patterns (a)–(d) are 16 bytes. (e) uses single precision floating point values.

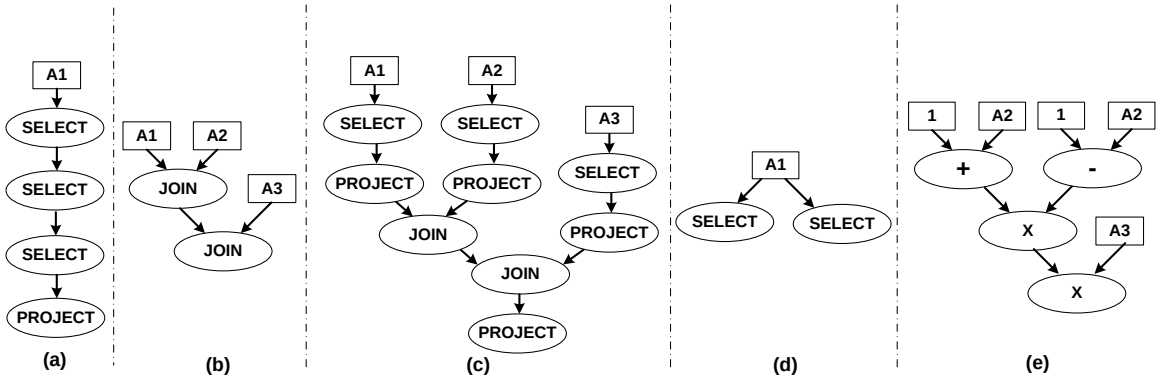

Figure 24. Common operator combinations from TPC-H.

4.4.2 Examples of Generated Code

Figure 25 shows the generated fused computation stage code of Figure 24(a) (only two SELECTs shown for brevity). It performs two SELECTs and a PROJECT. The first two filters operate on the value in *data_reg*, and store the filter result in *match*, which is later used to determine if follow-up operations are needed. The result of PROJECT is written to a new register *project_reg* since its data type is smaller than *data_reg*. The last step, stream compaction, dumps the value stored in this new register to the GPU's global memory. The generated code may be less compact than manually written code, but compilers such as *nvcc* can optimize it to produce high quality binary code.

4.4.3 Performance Analysis

The micro-benchmarks listed in Figure 24 are tested with inputs that fit in GPU memory. Figure 26 shows the speedup for GPU-only execution time (no PCIe transfer) with kernel fusion. The input data are randomly generated and then fed into the automatically generated fused code using the Red Fox compilation flow. The baseline implementation

```
if(begin_input + id < end_input)
{
  data_reg = begin_input[id];                    Load Data To Reg
  {
    unsigned char key = extract(data_reg);
    if(comp(key, 64))
    match = true;
  }                                              Filter0
  {
    if(match)
    {
      unsigned char key =extract(data_reg);
      if(comp(key, 64))
      match = true;
    }
  }                                              Filter1
  {
    if(match)
    {
      project_reg = project(data_reg, 0);
    }
  }                                              PROJECT
}
{
  unsigned int max = 0;
  unsigned int output_id = exclusiveScan(match, max, 0);
  if(match)
    buffer0[output_id] = project_reg;            Stream
  __syncthreads();                               Compaction
  if(threadIdx.x < max)
    begin_output0[outputIndex0 + threadIdx.x] = buffer0[threadIdx.x];
  outputIndex0 += max;
}
```

Figure 25. Example of generated computation stage source code of Figure 24(a).

for comparison directly uses the implementation from the primitive library without fusion. Similar to Figure 16, the performance data are averaged over a wide set of problem sizes (from 64 MB to 1 GB). On average, kernel fusion achieves a **2.89x** speedup. Cases (a) and (e) containing only thread dependence show the largest speedup, because they do not insert new synchronizations, and threads execute independently. Furthermore, (a) eliminates

three stream compaction stages and three gather stages after fusion. The speedup of case (d) is less than the rest because it has input dependencies and can only benefit from loading fewer inputs. (b) and (c) have CTA dependencies and need extra synchronizations which makes kernel fusion less beneficial than the thread dependence only cases. The speedup in case (c) is larger than (b) because (c) fuses some thread dependence operators.

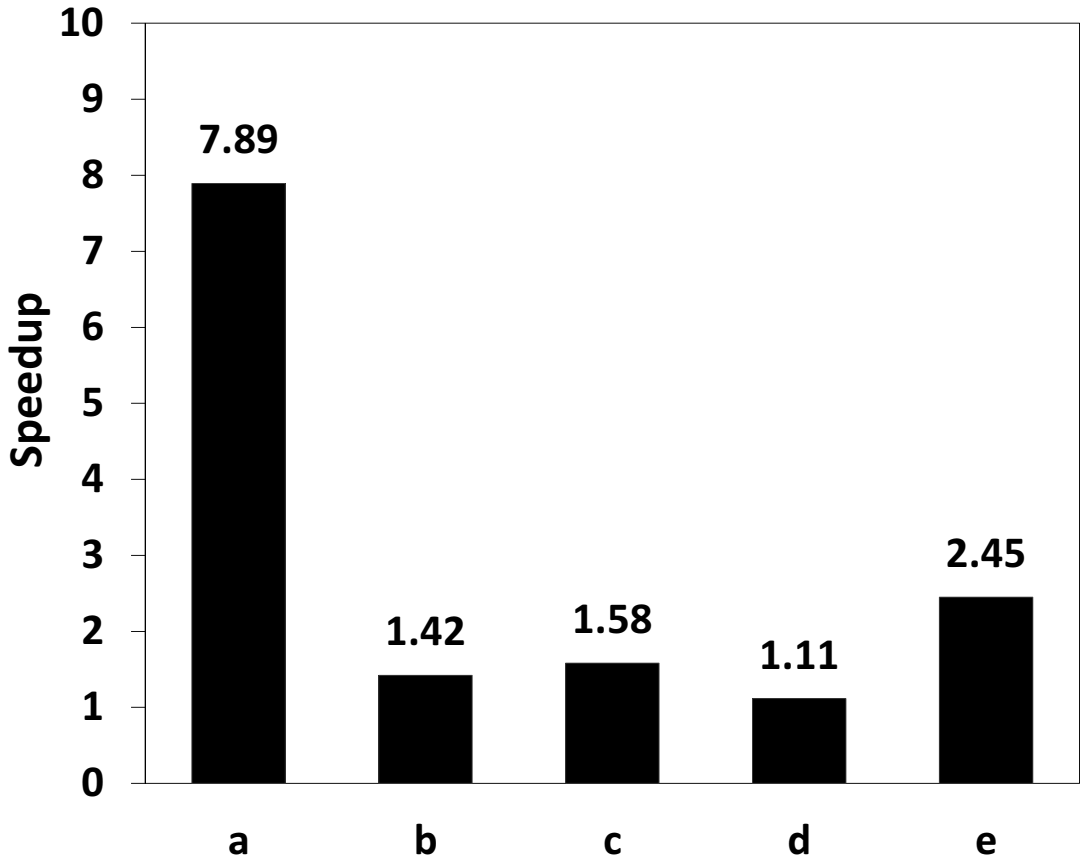

Figure 26. Speedup in execution time.

The next set of experiments examine the benefits claimed in Section 4.1, specifically the improvement in GPU global memory usage, total memory access cycles and compiler efficacy. Figure 27 shows the GPU global memory allocated and used with and without kernel fusion. The additional memory without fusion is attributed to large intermediate results. In pattern (d) however, the fused operator uses a little more memory because the fused compute stage has to store two outputs in memory for the future gather stage rather

than one. Similarly, Figure 28 shows the data for GPU memory access cycles (collected using the *clock()* intrinsic). On average, fusion reduces the GPU global memory access time by 59%. Finally, Figure 29 quantifies the impact of the compiler. All micro-benchmarks are compiled with *-O3* and *-O0* flags, both with and without kernel fusion. The figure shows the speedups achieved by *-O3* compared to *-O0*. Clearly, kernel fusion enables the compiler to perform better optimization.

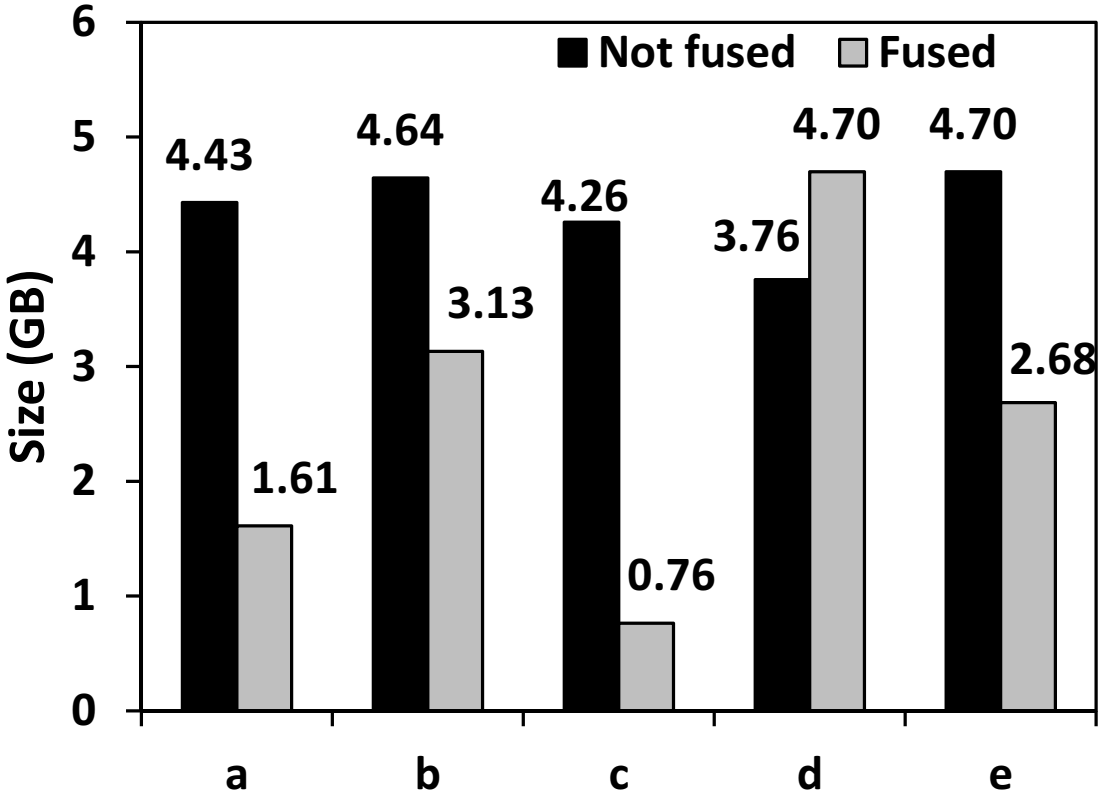

Figure 27. GPU global memory-allocation reduction.

When fusing two or three SELECTs (e.g., pattern (a)), the second or third SELECT might have some idle threads because some data are not matched in the earlier SELECT. This might impact the overall performance. Figure 30 examines the performance sensitivity of kernel fusion to the selection ratio (percentage of data matching selection condition) with randomly generated 32-bit integers. The results shows fusing two 10% SELECTs produces (more idle threads) 1.28x speedup while fusing two 90% SELECTs (less idle

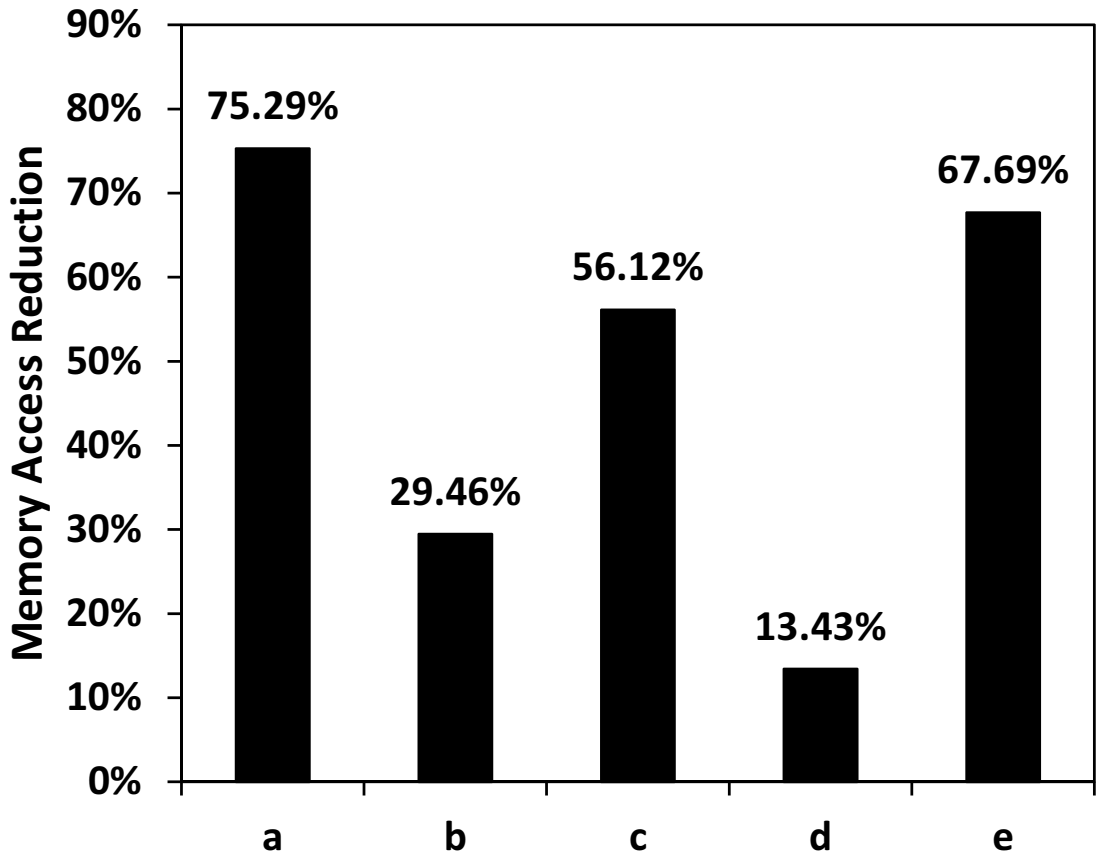

Figure 28. Reduced memory cycles with kernel fusion.

threads) produces 2.01x speedup. Thus, it is fair to say that idle threads may impact the performance but do not negate the benefits of fusion.

4.4.4 Resource Usage

Table 7 lists the GPU resource usage and occupancy (active warps / maximum active warps) of the individual operators and the fused patterns. Since resources are finite, utilizing too many resources per thread may decrease the occupancy. The resource information was collected from *ptxas* and occupancy is from *CUDA_Occupancy_calculator*. The top five rows list the resources used by individual operators (e.g. 1 PROJECT needs 11 PTX registers and 0 byte shared memory), and the bottom five columns show the usage of each pattern after kernel fusion (e.g. fused pattern (a) uses 22 PTX registers and around 2.3K shared

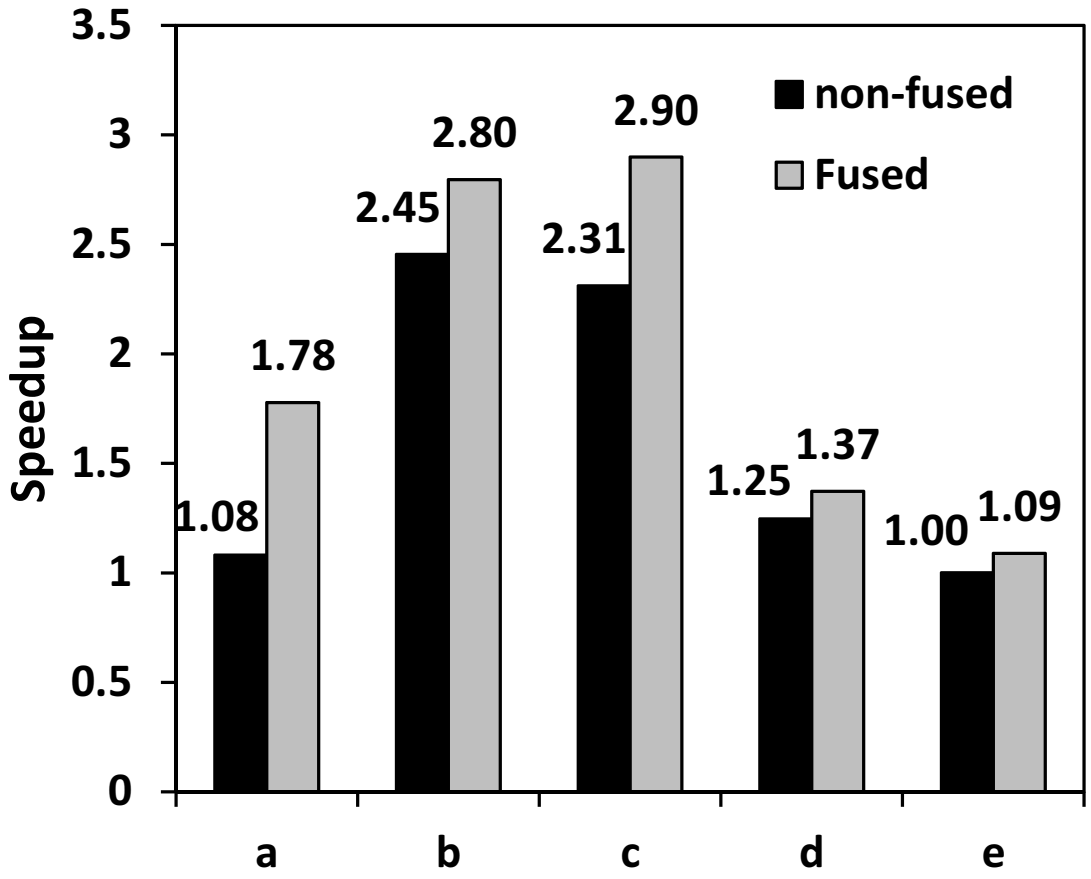

Figure 29. Comparison of compiler optimization impact.

memory). The statistics indicate that kernel fusion in most cases increases the resource usage which is the same as the impact of loop fusion, and consequently may lower occupancy (pattern (b) – pattern(e)). Taking pattern (b) as an example, it requires 55 PTX registers and about 23K shared memory with fusion. However, if running two JOINs back-to-back sequentially, each JOIN only needs 47 PTX registers and 13K shared memory. Pattern (a) will use less shared memory after kernel fusion than a single SELECT because

1. Thread dependence does not use shared memory to store temporary results.

2. The data type of fused results array buffered in shared memory uses smaller data type since PROJECT removes some attributes.

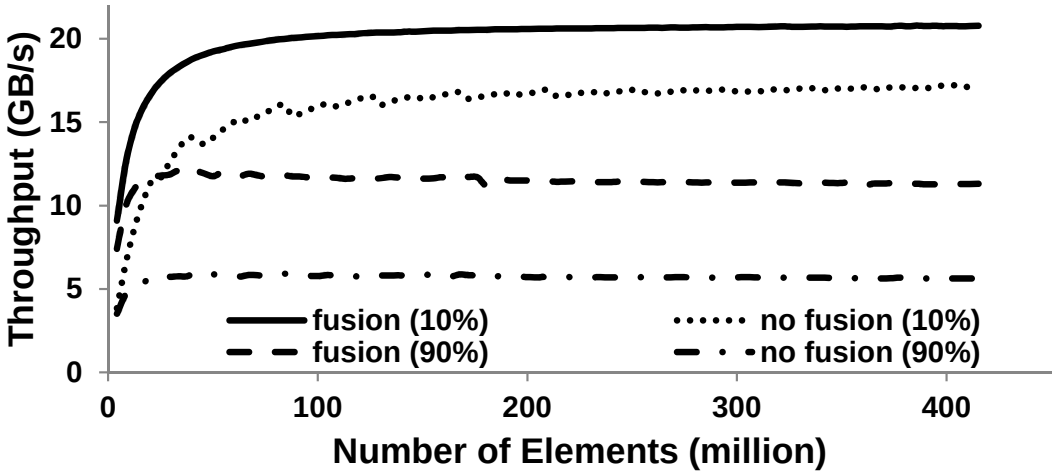

Figure 30. Sensitivity to selection ratio.

Table 7. Resource usage and occupancy of individual (top) and fused (bottom) operators.

	PTX Reg #	Shared Mem (Byte)	Occupancy (%)
PROJECT	11	0	100
SELECT	22	3848	88
JOIN	47	13580	38
+/-	10	0	100
Multiply	13	0	100
(a)	22	2308	88
(b)	55	23560	33
(c)	62	23048	17
(d)	30	4612	67
(e)	27	0	75

4.5 Summary

This chapter introduces a cross-kernel optimization framework, Kernel Weaver, that can apply kernel fusion optimization to improve the performance of RA primitives used in relational computations on GPUs. Kernel fusion aggregates larger body of code that can reuse as much data as possible. It can reduce the data traffic through the memory hierarchy caused by the I/O bound nature of database applications, and also enlarge the optimization scope.

To automate the process of kernel fusion, this chapter first classifies the producer-consumer dependence between RA operators into three categories: thread, CTA and kernel

dependence. Then, Kernel Weaver leverages the multi-stage algorithm design to weave stages from operators having thread and CTA dependence. The experiments shows that kernel fusion optimization brings **2.89x** speedup in GPU computation on average across the micro-benchmarks tested. The same technique can be applied to different domain, different representation format and different devices.

CHAPTER 5

GPU OPTIMIZED MULTIPLE-PREDICATE JOIN

This chapter introduces an alternative method that can aggregate the computation from different join primitives to increase the efficiency for GPUs to evaluate relational queries. The proposed method is an multi-predicate join algorithm which can execute relational join over multiple input relations by just one primitive.

In Chapter 3, the TPC-H results produced by Red Fox shows that

1. JOIN dominates the overall performance. If the GPU can efficiently execute JOIN, then most likely it can efficiently execute the whole relational query.

2. About half of the JOINs require the data to be sorted. Since SORT is much heavier than the merge phase of the sort-merge join, the overall time spent in sorting is close to the time spent in the merge part.

The Kernel Weaver optimization introduced in Chapter 4 can improve the performance across primitives including JOINs, but it still has following issues

1. Fusion cannot take place across the boundary with SORT operators which appear frequently in TPC-H queries.

2. Even fusing JOIN operators which execute on data sharing the same keys, the performance gain is limited which is much smaller than fusing operators having thread dependency.

3. The quality of automatically generated code although better, still leave room for optimization. Manually designed and carefully tuned code should do much better.

Thus, because of the importance of the JOIN and these limitations of Kernel Weaver, this chapter explores the opportunity of designing a *multi-predicate join algorithm* which

is not bounded by sort and has much better performance than binary joins. Specifically, this chapter adapted a worst-case optimal sequential CPU multi-predicate join algorithm to the GPU. Such multi-predicate join operators are quite powerful in that they can be used to implement most of the RA operators. To demonstrate the opportunities afforded by the multi-predicate join, graph workloads are analyzed in this chapter and the next. Specifically the fact that the explosive growth of graph sizes towards billions of nodes has pushed graph analysis to the forefront of numerous data intensive applications is exploited and is the focus of this chapter and the next.

In the rest of this chapter, Section 5.1 first introduces the clique problem and its relational solution. Section 5.2 reviews the original multi-predicate sequential join algorithm. Section 5.3 explains how to fundamentally transform the original algorithm to efficiently adapt it to the GPU. Finally, Section 5.4 evaluates the ported algorithm with some synthesized benchmarks.

5.1 Clique Problems

Clique listing such as finding triangles is an important operation for graphs; it is used as input for various graph properties and metrics such as the graph's clustering coefficient, triangular connectivity and others [83, 84]. Formally, given a graph $G = (V, E)$ its triangles are all sets $\{x_1, x_2, x_3\}$ such that $(x_i, x_j) \in E$. As is convention, each $\{a, b\}$ edge is as two directed edges (a, b) and (b, a) in a binary relation `E(x,y)`. The unique triangles are then computed via the following relational query expressed as a Datalog rule:

```
tr(x,y,z) ← E(x,y),E(y,z),E(x,z),x<y<z.                    1
```

Here, the first join `E(x,y),E(y,z)` "computes" all paths of length 2; these are then complemented by the third edge `E(x,z)` to form a triangle. The condition `x<y<z` is necessary since otherwise each triangle would occur 6 times. Similarly, 4-cliques are sets of four nodes such that each node is connected with each other. The Datalog rule to compute all unique 4-cliques is:

```
4cl(x,y,z,w) ← E(x,y),E(x,z),E(x,w),E(y,z),                    1
               E(y,w),E(z,w),x<y<z<w.                          2
```

5.2 Leapfrog Triejoin

LFTJ [1] is a worst-case optimal multi-predicate join algorithm for CPUs. Its basic building block is *leapfrog join*, which computes joins between *unary* relations. These are essentially multi-way-intersections. Then leapfrog join is generalized to multi-way-joins over general predicates to yield LFTJ.

A unary input relations R is accessed via a *linear iterator* interface that presents the data in *sorted order*. It is convenient to imagine the data to be stored in a sorted array of size $|R| + 1$ where the last cell is left empty. A linear iterator behaves much like a pointer into this array. It is initialized at the first element and provides methods for data access and iterator movement:

- *bool atEnd*() returns `true` if the iterator is positioned at the last array element (which does not correspond to a data value in R). Note that for an empty relation *atEnd*() will return `true` immediately after initialization.
- T *value*() returns the data value the iterator is positioned at. It must not be called if the iterator is *atEnd*().
- *void next*() moves the iterator to the next array cell. Like *value*(), this method must not be called if *atEnd*() is `true`.
- *void seek*(x) moves the iterator to the data value x. If x is not in R then the iterator is moved to the smallest element y that is larger than x; or to the last empty cell if such a y does not exist. Must only be called when the iterator is not *atEnd*() and the current *value*() is smaller than x.

The leapfrog join between a series of unary relations (e.g., R, S, and T) behaves somewhat similar to the merge-phase of merge-sort. The crucial difference is that leapfrog join

only searches for values that occur in all input relations. Thus, if one relation has a large value x, the algorithm can *skip forward* in the other relations to the value x. Skipping forward is done via the *seek*(.) operator. This leap-frogging motivates the name of the method.

The implementation of the leapfrog join is given in Figure 31. After **`leapfrog_init`**, the algorithm maintains the invariant that `Iter[p]` is the iterator with the smallest value while `Iter[p-1 mod k]` has the largest value; and the iterators in between are in ascending order. The core method is **`leapfrog_search`** where the algorithm repeatedly seeks the iterator with the smallest value to the iterator with the largest value until all iterators have the same value –a join result is found– or any of the iterators is *atEnd*() indicating that no result will be found.

For a fixed query, leapfrog join's runtime complexity is $O(N_{min} \log(N_{max}/N_{min}))$ where N_{min} and N_{max} are the cardinality of the smallest and largest relation, respectively [1]. This complexity bound holds if the following complexity bounds are satisfied for the linear iterator operations: *key*() and *atEnd*() need to be in $O(1)$; *next*() and *seek*(.) are required to be in $O(\log N)$ where N is the size of the relation. Furthermore, if m values are visited in ascending order, then amortized complexity of *seek*() and *next*() must not exceed $O(1 + \log(N/m))$. With an array representation, these bounds can easily be obtained. The bounds can also easily be obtained when data is stored in more standard paged data structures such as B-Trees.

To extend the linear iterators to allow iteration over arbitrary relations, the data of a relation R is organized as a Trie. The graph in Figure 32(a) is used throughout this chapter. This graph can be represented by a binary relation shown in Figure 32(b) in which each tuple has two numbers denoting the source and target nodes of the graph. When the relation is turned into a Trie as in Figure 32(c), every tuple in the relation is represented as a path from the root to a leaf node. Note that in the Trie,

1. All children of a node are *sorted* and *unique*,

Leapfrog join Algorithm as Linear Iterator

```
globals: Array Iter, integer p, bool atEnd

leapfrog_init():
  if any iterator is atEnd():
     atEnd := true
  else:
     sort Iter[0..k−1] by value() of each iterator
     p := 0; atEnd := false
     leapfrog_search()

leapfrog_search():
  max_value := Iter[(p−1) mod k].value()
  while true:
     min_value := Iter[p].key()
     if min_value == max_value:
        return
     else:
        Iter[p].seek(max)
        if Iter[p].atEnd():
           atEnd := true
           return
        else
           max = Iter[p].value()
           p := p + 1 mod k

leapfrog_next():
  Iter[p].next()
  if Iter[p].atEnd():
     atEnd := true
  else:
     p := p + 1 mod k
     leapfrog_search()

leapfrog_seek(seeked_value):
  Iter[p].seek(seeked_value)
  if Iter[p].atEnd():
     atEnd := true
  else:
     p := p + 1 mod k
     leapfrog_search()

leapfrog_atEnd(): return atEnd leapfrog_value(): return Iter[0].value()
```

Figure 31. Leapfrog-Join algorithm computing the intersection of k unary predicates given as an array `Iters`$[0..k-1]$ of linear iterators [1].

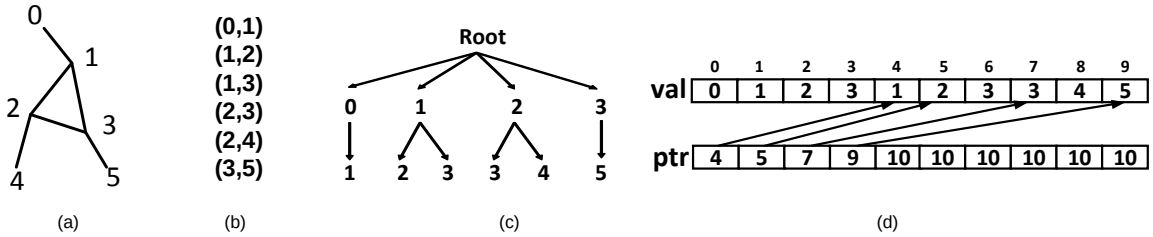

Figure 32. Sample graph (a), as binary relation (b), as Trie (c), and as TrieArray(d)

2. Children in the same level but belonging to different parent node may not be *sorted* and *unique*.

3. Only the children of a node are represented via a linear iterator and not necessarily all elements of a certain level; for example, if the iterator is positioned at the first 3 from left in the lowest level, and *next*() is called, *atEnd*() is true. To move to the neighboring 3, one needs to call *up*(), *next*(), and *open*() as explained below.

Besides the linear iterator operations, two more methods that allow moving up and down in the tree are needed to support the trie traversal:

- *open*() positions the iterator at the first children of the current node. This method may only be called if the iterator is not *atEnd*(); and
- *up*() moves the iterator back to the parent node.

LFTJ is a multi-predicate-join algorithm; that is it computes the result of Select-Project-Join queries directly without employing pair-wise joins. LFTJ can be explained by the example of

```
T(x,y) ← R(x,y),S(x),T(y)                                    1
```

LFTJ is configured by an order of the variables occurring in the body of the defining rule. Furthermore, the sequence of variables in each atom must be a subsequence of the chosen variable ordering. The example above chooses x, y, z because x, y in the first atom, x in the second atom, and y in the third atom are all subsequences of x, y, z.

The order of variables in join atoms can be permuted by deploying alternative indexes: $S(x, y)$ can be presented as $S'(y, x)$ with $S'(y, x) \leftarrow S(x, y)$. The chosen variable ordering, in general, influences the performance of the join evaluation. Finding a good ordering is usually deferred to a query optimizer.

LFTJ finds result tuples by using leapfrog joins to find assignments to the variables; one leapfrog join is used for each variable. Consider the example from above: first LFTJ performs a leapfrog join between $R(x, \_)$ and $S(x)$; this is done by opening the TrieIterators for R and S and using leapfrog join at the first level of the two Tries. However, whenever

a result c for x is found, the Trie-iterators for R and T are opened since these have the variable y. Then, a leapfrog join for y is performed. Once this is finished, the algorithm back-tracks to the join at level x, searches for the next match at level x and then descends again to level y once a match has been found.

For each atom in the rule body (e.g., R, S, and T) a TrieIterator is instantiated. Furthermore, an array of leapfrog joins is maintained, one join for each variable according to the variable order. The leapfrog join for a variable X has a pointer to the TrieIterators of those atoms in which X appears. In the above example, the leapfrog join for x points to the TrieIterators for R and S, while the leapfrog join for y points to the TrieIterators for R and T. LFTJ itself is implemented as a TrieIterator that presents the join result; so only the implementations for the TrieIterator interface need to be designed. A variable *depth* is used to keep track at which variable is currently operate (corresponding to the depth of the LFTJ-Trie). The methods *next*(), *seek*(), *atEnd*(), and *value*() are delegated to the leapfrog join for the active variable. The operation *open*() and *up*() are given in Figure 33. The result tuples can be found by following the TrieIterator.

Leapfrog Triejoin as Trie-Iterator

```
globals: Array LeapFrogs, integer d                                                    1
                                                                                       2
lftj_open():                                                                           3
  d := d + 1                                                                           4
  for each iter used in LeapFrogs[d]:                                                  5
     iter.open()                                                                       6
  LeapFrogs[d].leapfrog_init()                                                         7
                                                                                       8
lftj_up():                                                                             9
  for each iter used in LeapFrogs[d]:                                                  10
     iter.up()                                                                         11
  d := d - 1 // backtrack to previous var                                              12
                                                                                       13
lftj_value(): return LeapFrogs[d].leapfrog_value() lftj_next():                        14
LeapFrogs[d].leapfrog_next() lftj_seek(): LeapFrogs[d].leapfrog_seek()                 15
lftj_atEnd(): LeapFrogs[d].leapfrog_atEnd()                                            16
```

Figure 33. Leapfrog Triejoin Implementation

Figure 34 is an example showing the step by step operation of LFTJ for finding triangles in the graph. In this example, three relations are represented by three Triees, E(x,y), E(x,z) and E(y,z). The join operations performed on three variables x, y, and z occurs in three

corresponding levels as explained below.

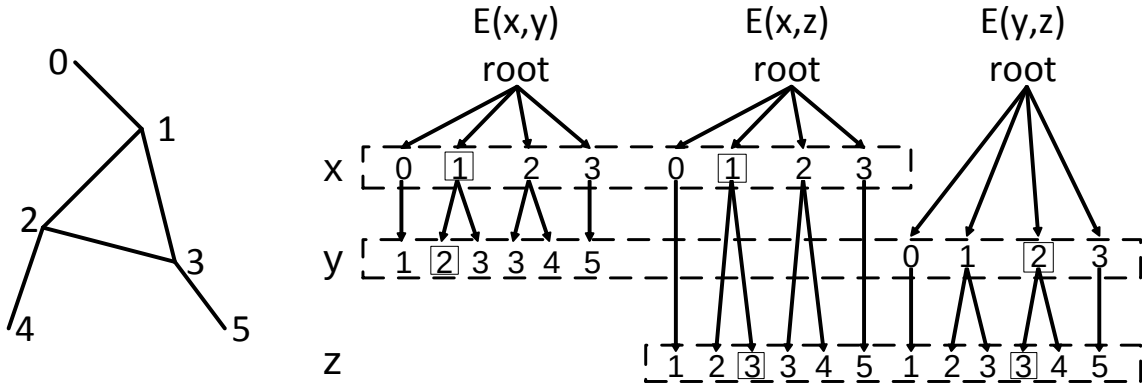

Figure 34. Example of using LFTJ to find triangles. Final results are marked by boxes

1. *open* level x. After this, both E(x,y) and E(x,z) point to the children nodes {0,1,2,3}. E(y,z) does not have variable x.

2. Use Leapfrog Join to find the first matched number, number 0, in the level x from Trie E(x,y) and E(x,z).

3. *open* level y. E(x,y) *open*s from the number 0, level x which is found in the previous step. Now E(x,y) is at level y and points to the child of the found node which is {1}. E(y,z) *open*s level y from the root level and it points to {0,1,2,3} at level y. E(x,z) does not have variable y.

4. Use Leapfrog Join to find the first matched number, number 1, in the level y between data set {1} from E(x,y) and {1,2,3,4} from E(y,z).

5. *open* level z. E(x,y) does not have any data at level z. So, it remains at the matched node at level y. E(x,z) *open* the child node {1} from number 0 at level x which is found in step 2. E(y,z) *open* the child nodes {2,3} at level z.

6. Use Leapfrog Join to intersect two data sets at level z {1} from E(x,z) and {2,3} from E(y,z). The intersection cannot find any result.

7. E(x,z) calls *up* to return to number 0 at level x. E(y,z) calls *up* to return to number 1 at level y.

8. Use Leapfrog Join to find the second matched number in the level y between data set {1} from E(x,y) and {1,2,3,4} from E(y,z). The search hits the end and returns with no result.

9. E(x,y) calls *up* to return to number 0 at level x. E(y,z) calls *up* to return to the root node.

10. Use Leapfrog Join to find the next matched number, number 1, in level x from E(x,y) and E(x,z).

11. *open* level y again. E(x,y) *open*s from number 1, level x. Now E(x,y) points to the child nodes {2,3} at level y. E(y,z) *open*s from the root level and points to child nodes 0,1,2,3 at level y.

12. Use Leapfrog Join to find the first matched number, number 2, in the level y between data set {2,3} from E(x,y) and {1,2,3,4} from E(y,z).

13. *open* level z again. E(x,z) *open*s the child nodes {2,3} at level z from level x. E(y,z) *open*s the child nodes {3,4} at level z.

14. Use Leapfrog Join to intersect two data sets at level z {2,3} from E(x,z) and {3,4} from E(y,z). The first matched number is number 3. Because level z is the bottom level, this is the first found triangle which is labeled as number 1 at level x, number 2 at level y, and number 3 at level z.

15. Continue the above steps until traversals of the three tries complete. This example only contains one triangle.

The above example shows that LFTJ can be viewed as cooperatively traversing several trees in a depth first order. The most important method is *seek* which is used to find

matched values among several data sets. Since all the data is sorted, the *seek* method can be implemented by any O(log(n)) algorithm such as binary search.

5.3 GPU-Optimized Multi-predicate Join

The new approach, referred as *GPU-optimized* uses a carefully designed strategy that is specifically optimized for GPUs. It uses an efficient data structure for the GPU to access the data and fundamentally changes the original algorithm to but employs the main principles.

5.3.1 TrieArray Data structure

As shown in Figure 32(d), the *GPU-optimized* approach uses a data structure called TrieArray. The structure is inspired by the commonly used CSR format used for graphs and matrices. The TrieArray structure uses two flat arrays: *val* and *ptr* because accessing contiguous data in flat arrays is very efficient in a GPU. As an example, the TrieArray for the Trie in Figure 32(c) is shown in Figure 32(d). Every node in the Trie is labeled by an index number in a breadth first way, starting from left to right and from top to bottom. The first array, *val* stores the Trie node values. val[i] stores the node value of the *i*-th node. The total length of this array equals the number of nodes. The second array, *ptr*, stores the information related with the Trie structure. ptr[i] identifies the index of the first child of the *i*-th node. The difference between two adjacent number in *ptr* is the child number of the left node. For example, the children of the first node starts from 4 and the children of the second node starts from 5. Thus, the children number of the first node is $5 - 4$ which is 1. The nodes in the bottom layer which do not have child nodes do not store anything in the *ptr* array. The last entry in this array stores the total number of the nodes so that the last entry in the second last level can correctly get its children number. Thus, array *prt* is shorter than the array *val* by the number of last level node number minus one. Last but not least, TrieArray is more compact and uses less memory than the original relation table. Thus It should take less time to access TrieArray than to access flat table.

5.3.2 Algorithm Overview

The *GPU-optimized* approach targets to pursue the best performance. At a high level, the GPU-optimized approach makes two significant changes to the original sequential LFTJ algorithms.

1. The new algorithm changes the depth first traversal into breadth first traversal. The advantage of breadth first is that it can expose more fine grained parallelism because nodes in the same level can be processed in parallel. The downside is that this approach has a larger memory footprint because it needs to maintain a work list to hold these nodes.

2. The new algorithm replaces most of the O(log(n)) search with O(n) linear search. O(log(n)) method has better computation complexity, but it has random memory access pattern which is detrimental in GPUs.

To traverse the Tries in a breadth first order, the algorithm starts from the top level, all GPU threads work together to horizontally intersect one level of nodes belonging to different predicates. The algorithm continues in subsequent levels and finishes when the bottom level is processed. Thus, the original problem is divided into three sub-problems.

1. Node expansion - Similar to regular Breadth-first Search (BFS), the GPU-Optimized approach needs to expand the child nodes from the parents nodes in parallel.

2. Intersection - The expanded child nodes from different predicates are intersected to find the matched values. The output is a bitmask to indicate the positions of the matched nodes.

3. Filter - The bitmask generated above is used to filter several arrays to get the result.

The first two sub-problems are hot research topics in GPGPU research. Here, two high performance primitives, *vectorized sorted search* and *load-balancing search* from the

ModernGPU library are used to assist the processing of these two sub-problems. These two primitives are both designed based on the merge path [37] framework which partitions the workload among each CTA and then among each thread inside the CTAs such that workload balance is guaranteed. The computational complexity of both algorithms is $O(N)$ where N is the input size. Moreover, these two primitives are also optimized for coalesced memory accesses, bank conflicts, ILP, etc. A brief introduction is as follows.

1. Vectorized Sorted Search - This primitive reads in two sorted arrays and locates the lower/upper bounds of each element of the first array in the second array. If elements of both input arrays are unique, *vectorized sorted search* can be directly used to intersect two arrays since a simple check of the lower/upper bound with the search key can indicate if the match exists or not.

2. Load-Balancing Search - It is the reverse operation of exclusive scan. For example, the children count of the first level in Figure 32, is an array of {1,2,2,1}. The exclusive scan of the children count is {0,1,3,5} which corresponds to the position in the output array where the child nodes can be expanded into. Running load-balancing search over the exclusive scan result will generate another array {0,1,1,2,2,3} where each value corresponds to the parent id that can expand its children to this position. For example, the second element has value 1 means parent 1 can expand its children to this position. Load-balancing search is used by several other primitives of the ModernGPU library to balance workloads.

When processing the predicates level by level, some nodes in the Tries are filtered out. Every predicate has a result array to store the index of remaining elements. This array gets updated after every intersection. Since several Tries are traversed at the same time, there must be some way to connect their individual result arrays. In the new join algorithm, elements at the same position of each index array point to the subtrees that will be expanded and intersected in subsequent levels. These elements are referred as *associated* elements.

Predicates that have been intersected are referred as *processed* while the rest are referred as *not processed*. According to the status of the two predicates to be intersected (either *not processed* or *processed*), three cases which use different intersecting procedures need to be considered. The problem of finding triangles (Figure 34) in a graph is shown below as an example to explain these three cases, followed by the description of a general algorithm.

5.3.3 Examples of Finding Triangles

Case 1: *Not Processed* intersects *Not Processed* Processing the top level, level *x*, involves the two predicates `E(x,y)` and `E(x,z)` which have not been intersected. In fact, this case only happens when processing the highest level of two predicates because child level has to be processed after its parent. Moreover, the top level data are always stored consecutively in the array `val`. Therefore, the procedure only requires a normal pairwise intersection. Since the data from each predicate are unique, one *vectorized sorted search* can find all the results. As to the clique-problems, two predicates (e.g. `E(x,y)` and `E(x,z)`) contain the same values in level *x* so that the intersection results are identical to the inputs. Thus, the intersection can actually be skipped. As shown in Figure 35, in the result arrays belonging to predicates `E(x,y)` and `E(x,z)` respectively, elements in the same position point to the matched values. For example, the first elements of two result index arrays are both 0, which indicates the first elements of `E(x,y)` and `E(x,z)` are matched (the matched value is 0).

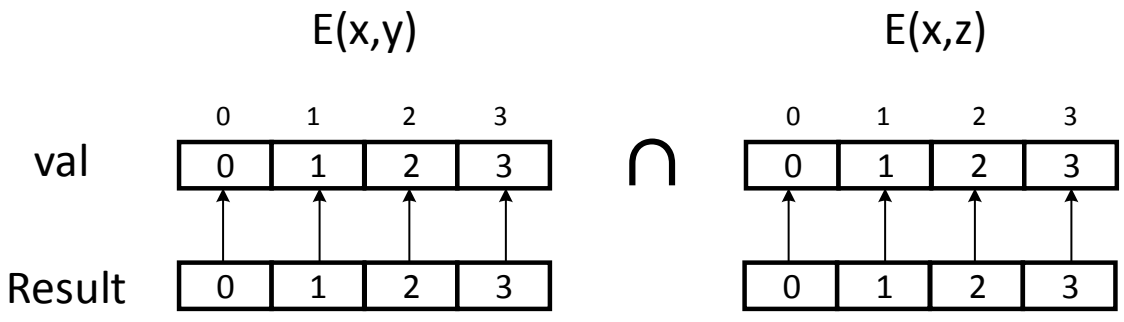

Figure 35. Processing Level *x*

Case 2: *Processed* intersects *Not Processed* Processing the second level, level *y*, requires intersecting the predicate `E(x,y)` which is processed in the top level and the predicate `E(y,z)` which is a new input to the procedure. As explained in the overview, three steps are used in this process:

1. Expanding the child nodes of `E(x,y)` from level *x* to level *y*.
2. Searching the matched values in `E(y,z)`.
3. Generating the result index arrays for all three predicates.

The purpose of expanding children is that the following intersection step can read in the input as a contiguous array so that the intersection step can evenly partition the input and access the memory with coalesced accesses. The overhead of expanding depends on how many parents need to be expanded and how many children each parent has. Note that the parents to be expanded may not be consecutive (consecutive in this example but not in the general case) and can cause non-coalesced memory accesses. However, the memory access pattern is not completely random because child nodes of the same parent are stored together. Also note that the expanded children come from different parents so that they may not be sorted. A most straightforward approach to expand the children of `E(x,y)` from level *x* to level *y* is

1. Running exclusive scan of the children count to calculate the output position;
2. Mapping the parents to GPU threads such that each thread expands one parent and stores the children to the calculated position.

A major drawback of this approach is that the workload of each thread depends on its child count which varies substantially. Instead, *load-balancing search* is used to guarantee that each GPU thread expands the same number of children. Figure 36 shows the major computations to calculate the addresses of the children to load the values. In this example, six threads independently expand six children.

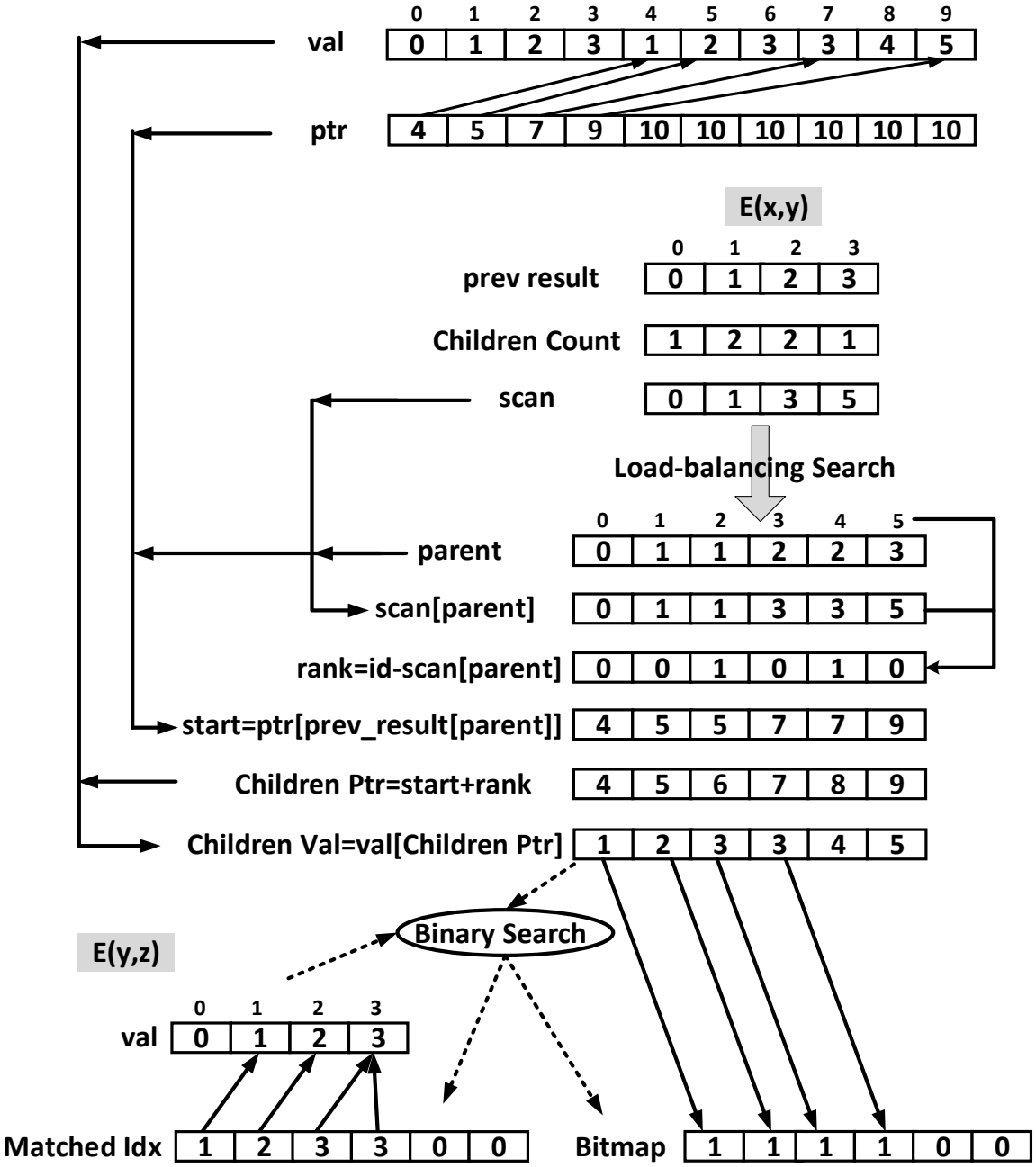

Figure 36. Child Expansion and Binary Search of Level *y*

Usually, the expanded number of child nodes is larger than the parent number because a parent usually has more than one children. In some cases, the expanded children number can be even larger than the GPU memory capacity. Thus, the algorithm compares the total children number (calculated by the exclusive scan) with the free memory size before

physically expanding the children. If the number of child nodes is larger, the algorithm will expand the child nodes in multiple rounds. In each round, it partitions the parent nodes and only expands those that for which the number of child nodes fit in free GPU memory. In Figure 37, the expanded child count is six and the free memory can only hold three nodes. Thus, the first round expands the first three parents and the second round expands the last parent.

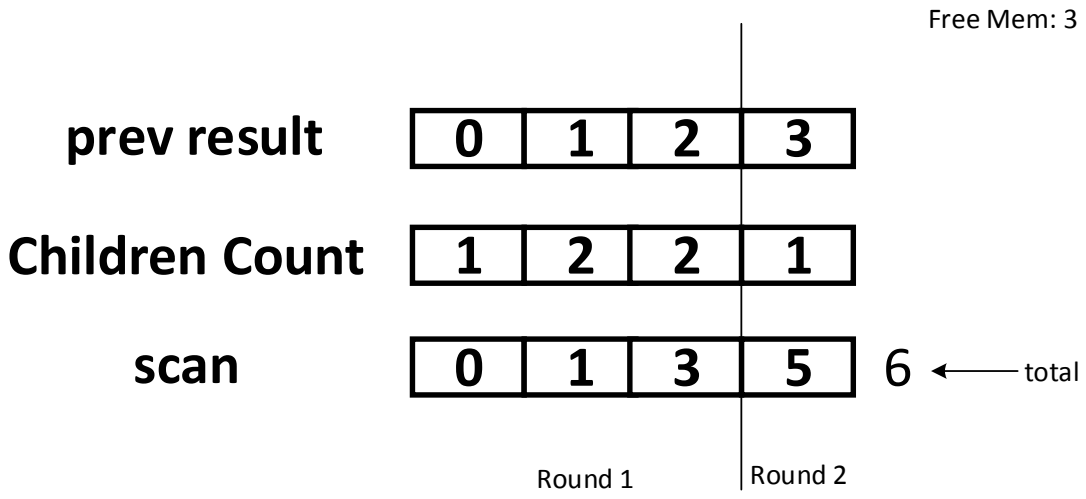

Figure 37. Example of using two rounds to expand children

After expanding E(x,y), the expanded child nodes of E(x,y) needs to intersect with the y level of `E(y,z)` which is not processed before. *Vectorized sorted search* cannot be used here because one of the inputs (the expanded children from `E(x,y)`) may not be sorted. Instead, a traditional binary search is used. The result of binary search is

1. A bitmap that shows if a child of `E(x,y)` finds a match.
2. An array that records the matched indices in `E(y,z)`.

The last step is generating the new result index arrays for all predicates. Elements of index arrays generated from `E(x,y)` and `E(y,z)` are already *associated* and point to the same values. While not directly involved in the computation of level *y*, the result index array of `E(x,z)` from the previous intersection should be updated such that it is also *associated*

with the newly generated index array of `E(x,y)` and `E(y,z)` Each updated index array element of `E(x,z)` is *associated* with the parent of each result index array element of `E(x,y)`. Specifically in this example, the updated index array elements of `E(x,z)` should also point to the parents of the new results of `E(x,y)`. The bitmap generated by the binary search is used to filter several different arrays to create the final new results. The detailed process is shown in Figure 38. The filtering is implemented as a stream compaction process.

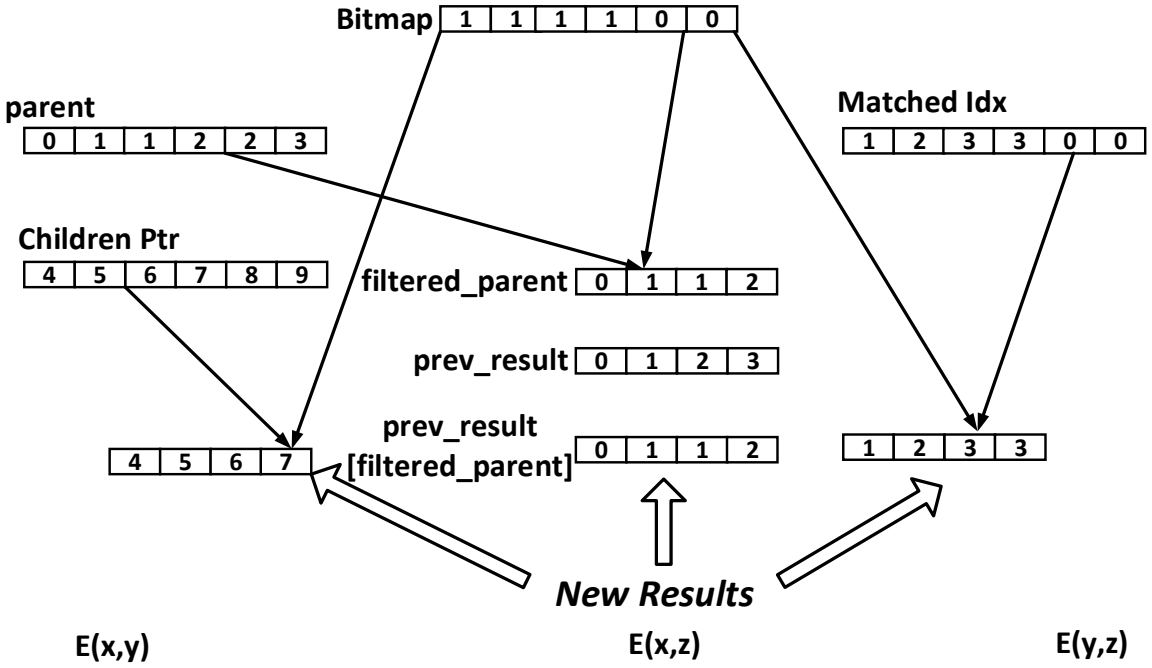

Figure 38. Generating New Results of Level *y*

Case 3: *Processed* intersects *Processed* The last level, level *z*, of the triangle example involves two predicates `E(x,z)` and `E(y,z)` that have already been processed. The computation flow is similar to processing level *y*, first expanding children for both predicates, then intersecting them and finally generating the results. The child expansion step is shown in Figure 39. It should be noted that some parents have been expanded twice. For example, the `prev_result` of the predicate `E(x,z)` in the figure has two indices having the same number 1 and this parent node is expanded twice. Redundant expansion consumes more memory space but eases the intersection part because it can perform better load balancing by using merge-path and more coalesced memory accesses. Similar to Case

2, the free memory size is also checked and only children fitting in memory are going to be expanded. The expanded children of both predicates may not be sorted. However, the intersections only apply to the nodes whose parents are *associated* (i.e., the subtrees LFTJ is chasing down) which is indicated by the sorted parent id. Furthermore, children of the same parent are sorted and unique. So, wrapping the value of each children values with their parent id and treating it as a single value (i.e., parent_id.child_value), the inputs to be intersected can still be considered as sorted and unique. Therefore a simple *vectorized sorted search* can find all matches. Figure 40 shows the major steps in this cases. The output of the intersection is again a bitmap and an array records all matched indices. These two outputs are again used to generate the results of this level as shown in Figure 41 which is similar to level *y*.

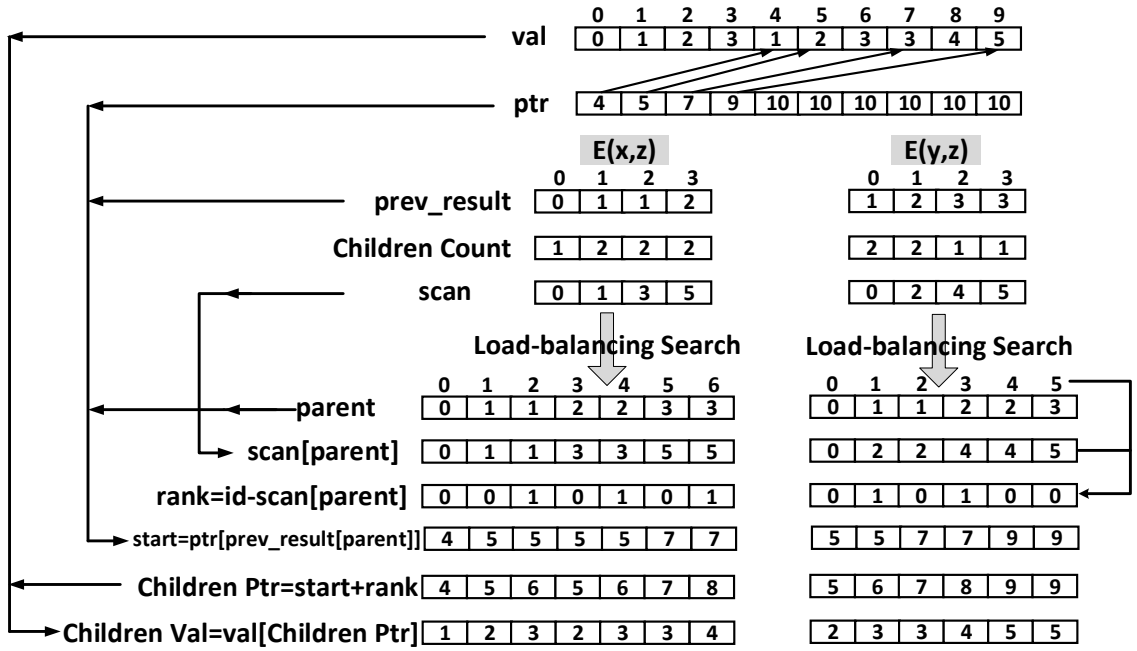

Figure 39. Child Expansion of Level *z*

The operations performed by the three cases involve several CUDA kernels. Kernels pass data via global memory.

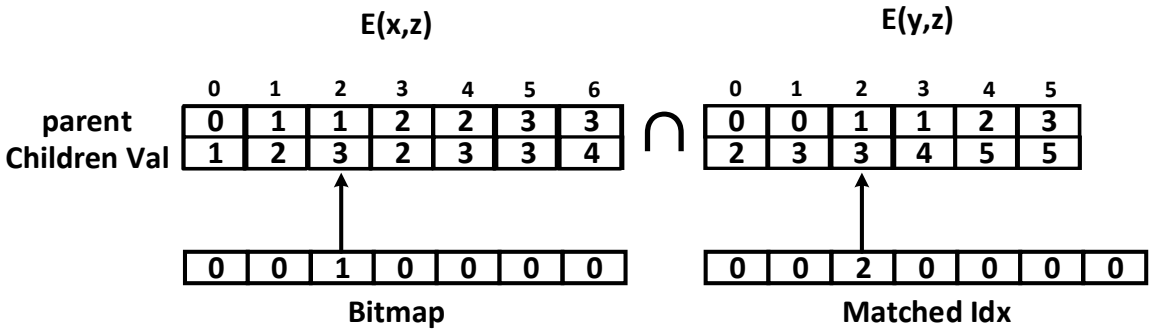

Figure 40. Intersection of Level z

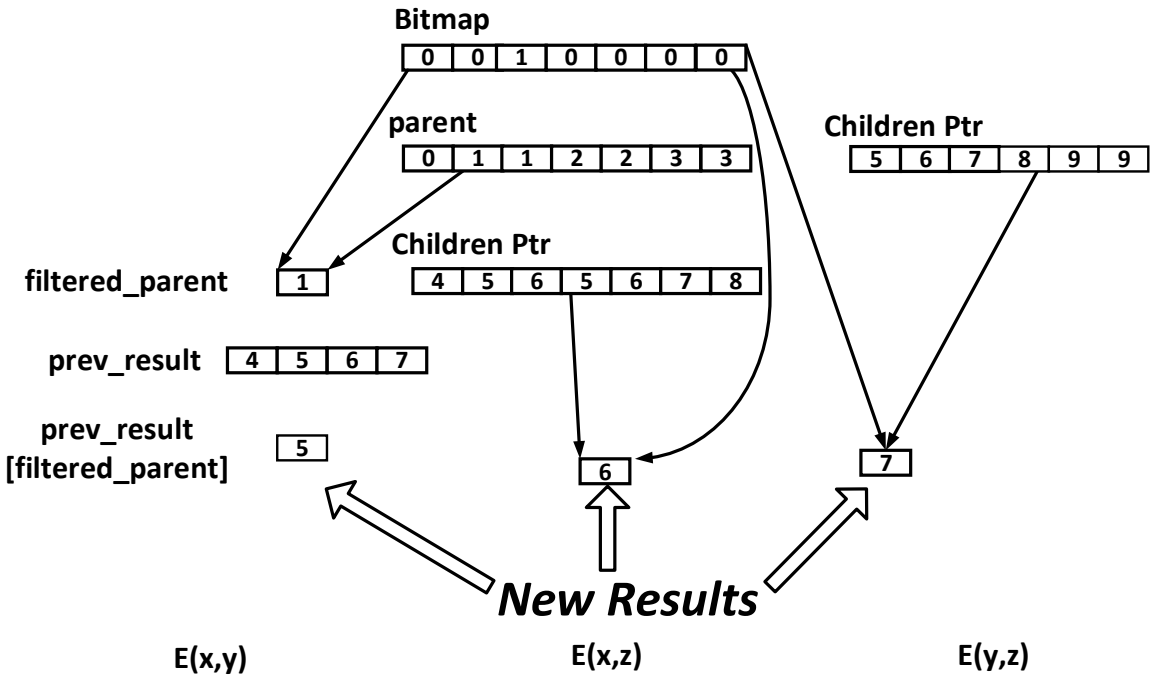

Figure 41. Generating New Results of Level z

5.3.4 General Join Algorithm

The three levels in the example of finding triangles show how the algorithm handles three different cases. As a general multi-predicate join, every level may handle more than two predicates. The combination of any two predicates always falls into one of the above three cases. The general algorithm that processes one level is described as below:

- Classify the predicates according to their status, *processed* or *not processed*.
- For all the *not processed* predicates, run the pairwise intersection as Case 1 to find

common values of all new predicates. The memory accesses are mostly coalesced.

- For all the *processed* predicates, run the pairwise intersection as Case 3. The intersections can reduce the result size. Some of the memory accesses are index-based indirect accesses and may not be coalesced.

- Use the costly binary search as in Case 2 to intersect *not processed* and *processed* predicates.

The above algorithms aggressively selects predicates that can be processed by the most efficient intersections and use at most one binary search in one level. Figure 42 shows how the four clique problem can be solved. Three intersections belong to Case 1, another three belong to Case 3, and only two binary searches are used. The pairwise intersection that in the same level are not independent. Some intermediate results of the first intersection can be reused by the later intersections. For example, the later intersection does not need to expand nodes that has already been expanded by the first intersection. Instead, it only needs to filter the existing expanded children. If there are more than two predicates involved in one case, the algorithm sorts the predicates according to their length and starts from the shortest then to the longer ones. The reasons for this are

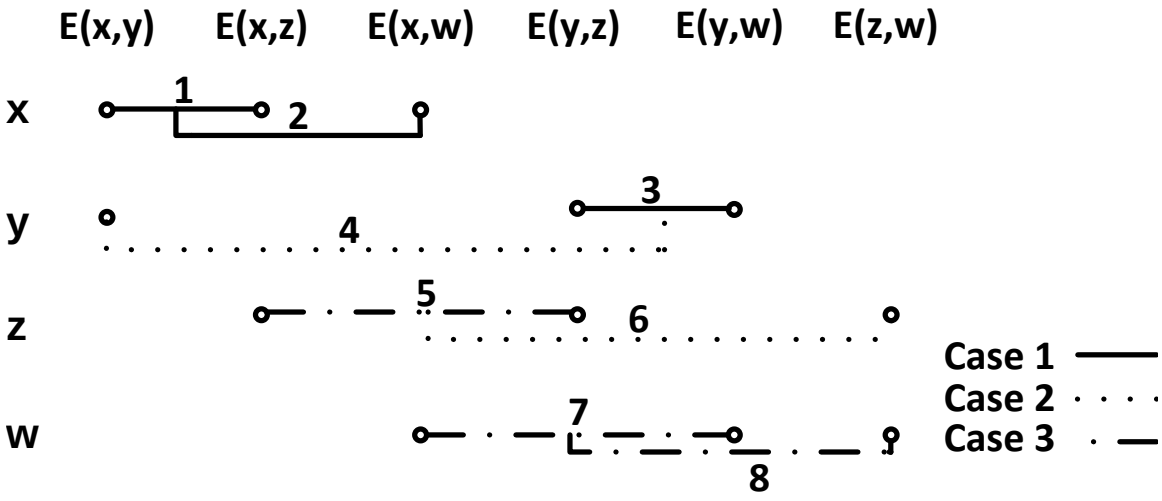

Figure 42. Intersections to solve 4-clique problem. Numbers are the orders to perform the intersection

1. The computation complexity of Case 3 is O(n) and the complexity of Case 2 is O(log(n)). Running short ones earlier can reduce the execution time.

2. The short one generates short bitmasks which can reduce the memory footprint and also the time to scan the bitmask.

GPU-Optimized follows the same programming style as ModernGPU and uses the same optimization techniques such as register blocking, texture memory, etc. Moreover, kernels that use load-balancing searches use merge-path to map array elements to GPU threads as many ModernGPU primitives do. The other kernels such as filtering kernels, array elements are evenly assigned to CTAs and then to threads.

Compared with other GPU algorithms designed for pairwise relational join, *GPU-optimized* approach

- processes all the predicates at the same time. Take triangle listing as example, when intersecting E(x,y) and E(y,z) at level y, E(x,z) is also filtered. Thus, *GPU-optimized* has less searches to perform.

- replaces heavy data reorganization (sorting or rebuilding hash table when join keys are changed) with binary searches.

- has relatively smaller memory footprint because of the compact TrieArray data structure.

5.3.5 Trade-off Between Binary Searches and Linear Searches

In the previous example, linear search is used when two arrays are both sorted because it has good load balance and good memory access patterns. However, in some cases when the two sorted arrays are different (e.g. different distributions or different lengths), adding some binary searches can be rewarding mainly because of its good computation complexity although always using binary searches is slower in the GPU as shown in the experiment section.

For example, when intersecting two arrays as shown in Figure 43 where one is much longer than the other one, using linear search needs to compare all 12 numbers, but it only requires two binary searches. In this extreme example, the good computation complexity of binary search may outperform the GPU efficient linear search. Thus it might be beneficial to combine these two technologies to get better overall performance.

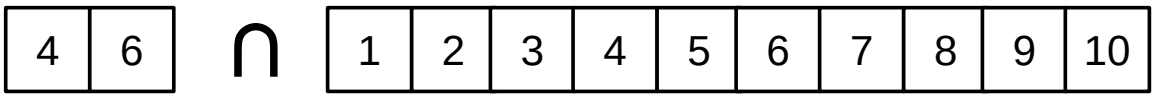

Figure 43. Example of Tradeoff between Linear Search and Binary Search.

There are at least two ways of mixing binary search with linear search as shown in Figure 44

- Only some elements uses binary search. For example, if the first and last elements of the short array use binary search, the linear search scope in the longer array can be restricted. In Figure 44(a), the linear search only needs to compare data in the gray area. The cost of binary search is negligible, but its benefit can be large. This method is used in GPU-optimized by default. Certainly, some other elements in the short array can also use binary search to further shrink the search scope, but the cost of binary search will increase.

- Only run the first several iterations of binary search. The first several iterations of binary search can skip large segments of data. For example, the first iteration can get rid of half of the data while the last iteration can only discard one data item. Thus, every element in the short array can use the first several iterations to locate a small area that its match may exist and the following linear search can be used to find the precise position. For example, only the gray areas need to be checked in Figure 44(b). This method should have relatively higher overhead than the first one because more random memory accesses are used.

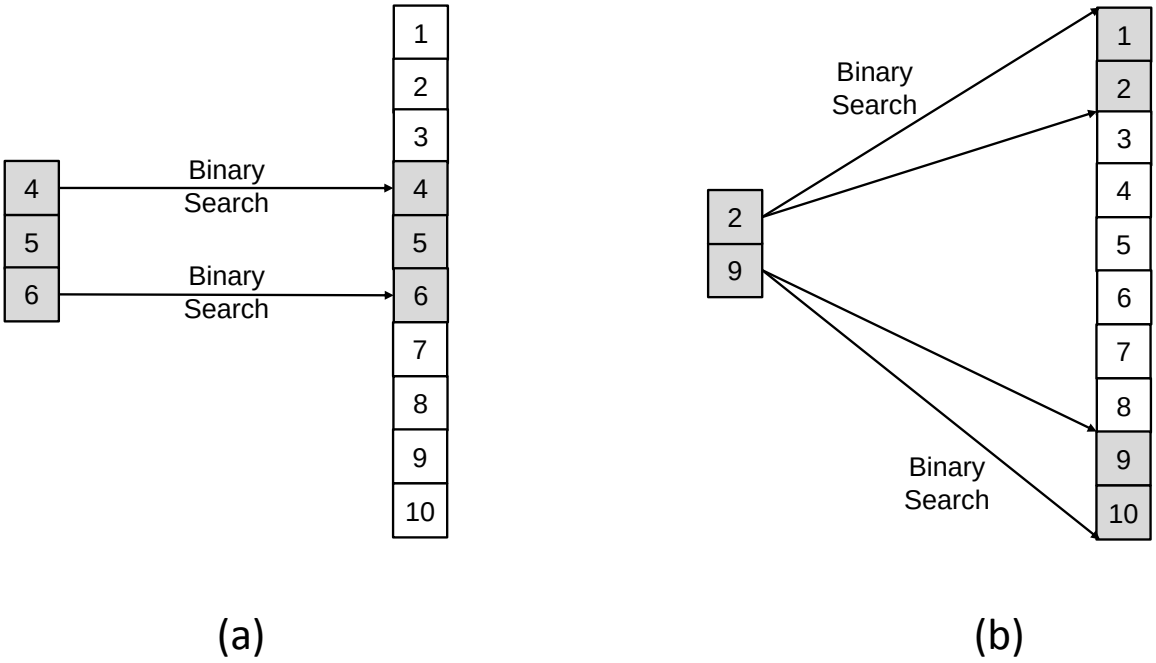

Figure 44. Two methods of mixing linear search with binary search. (a) Some elements uses binary search; (b) Run the first several iterations of binary searches.

The selection between the two methods and the number of binary searches that should be used depends on the data distribution and requires detailed workload characterization. This remains a future research topic.

5.4 Experimental Evaluation

The performance of four approaches are experimentally evaluated in this section: LFTJ-CPU, LFTJ-GPU, GPU-optimized and regular pairwise relational joins. LFTJ-CPU runs the original LFTJ algorithm on a multi-core CPU. It parallelizes the sequential algorithm by evenly partitioning the data in the first layer and maps them to the CPU threads. LFTJ-GPU works in the similar way as LFTJ-CPU, but replaces the CPU threads by the GPU threads without any further optimizations. LFTJ-CPU and LFTJ-GPU completely uses binary searches to seek data. The benchmarks focus on the two queries that find triangles and four-cliques in randomly generated graphs. Red Fox is used to perform regular sort-merge joins from the ModernGPU library. The inputs to Red Fox are flat arrays as shown

in Figure 32(b) and the other three approaches use the TrieArray structure. The results of different approaches are verified with the LogicBlox platform.

Table 8 lists the characteristics of the evaluation environment. The high-end NVIDIA GeForce GPU is attached as a device on the host PCIe channel. Synthetic graph workloads are generated as follows. Graph sizes range from 10K edges to 100M edges; Edges are randomly placed between nodes. Nodes are stored as 64-bit integers. The number of nodes is maintained as $\text{numEdges}^{(C-1)/(2C-4)}$, where C is 3 for triangles or 4 for 4-cliques. Therefore, each node in triangle workload has 2 edges on average and 4-clique has $2 \times \text{numEdges}^{1/4}$ edges. Note how nodes for the 4-clique dataset have increasingly connected edges as the size of the graph grows. This will stress the memory footprint of the GPU-optimized approach and Red Fox. The degree-distribution is chosen such that the number of found triangles or 4-cliques is very sparse, ranging from 0 to 3.

Table 8. Experimental Environment.

CPU	Intel i7-4771	GPU	GeForce GTX Titan
G++	4.6.3	NVCC	6.0
OS	Ubuntu 12.04	Driver	331.62
CPU Mem	32GB	GPU Mem	6GB (288.4GB/s)
PCIe	3.0 x16		

In the experiments, all GPU approaches assume the input data are sorted and transformed into the TrieArray and reside in the GPU memory. PCIe transfer time is not included in the following experiments since this chapter focuses on in-core algorithms.

Figure 45(a) shows the performance for finding triangles. For all tested graphs, GPU-optimized is faster than the other three approaches which demonstrates that all optimizations work well. Compared with LFTJ-GPU, GPU-optimized uses a similar data structure but can more efficiently utilize GPU resources. Compared with Red Fox, GPU-optimized works on a more compact data structure and uses binary searches to replace sorting. Note that Red Fox cannot run 100M edge graphs because the intermediate data generated by the

first binary join is larger than the GPU memory capacity. Averaging from 10K to 100M edges, GPU-optimized is 1.95x faster than LFTJ-GPU, 6.37x faster than LFTJ-CPU, and 3.00x faster than Red Fox (10K to 30M). When running LFTJ-optimized with the largest graph, the internal node expansion requires more memory than the GPU can provide, then extra overhead occurs to expand nodes in multiple rounds as introduced earlier. Thus, GPU-optimized has relatively sharp decline at 100M edge graph , but it is still 7.8x faster than the CPU approach. It is also interesting to notice that LFTJ-GPU is 3.12x faster than LFTJ-CPU although they are originated from the same algorithm; which demonstrates raw GPU performance.

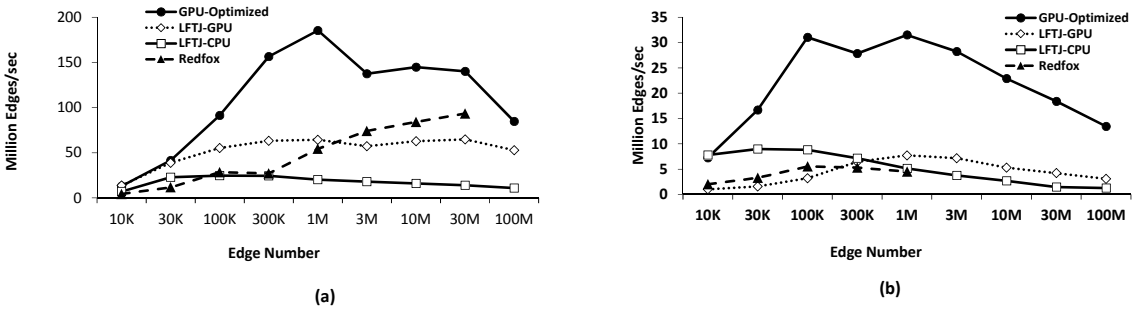

Figure 45. Experimental Results: (a) Triangle Throughput; (b) 4-Clique Throughput.

Figure 45(b) compares the performance of finding 4-cliques. Recall that 4-clique involves much more computation than 3-clique so that the achieved throughput for all four approaches is much lower than for the triangle problem. Here, the advantage of GPU-optimized is more evident. GPU-optimized has to expand nodes in multiple rounds when the graph has 1M or more edges. Red Fox can run up to 1M edges. Averaging from 10K to 100M, GPU-optimized is 5.87x faster than LFTJ-GPU, 6.19x faster than LFTJ-CPU, and 5.32x faster than Red Fox (10K to 300K). When running 100M edge graph, LFTJ-optimized is 11x faster than the CPU approach.

Most effort of the time in GPU-Optimized is spent on improving memory access patterns and load balance between parallel threads. To verify the effectiveness of the used approaches, *nvprof* is used to measure the *warp execution efficiency* (average active threads

per warp ÷ maximum number of threads per warp) and *ld/st average replay* (average number of replays for each load/store instruction executed when there is bank conflict or non-coalesced memory access). For comparison, Wang et al. [85] measured the above metrics for a wide range of irregular GPGPU applications including pairwise binary search based relational join. In the 30M triangle problems, *warp execution efficiency* across all kernels are as high as 95% (100% is ideal) which is as high as many regular GPGPU applications. As to *ld/st average replay* (0 is ideal), binary search kernel is 31 and node expansion kernel is 19 as expected. Excluding these kernels, the average number across all kernels is down to 0.96 from 4.62. 4-cliques problem has similar *warp execution efficiency* (94% overall for 3M), but much better *ld/st average replay* number (0.63). The reason is that it only uses 2 binary searches and its node degree is much larger which is good for coalesced memory access pattern.

The peak throughput of GPU-Optimized in triangle example is around 190 MEdges/sec when the graph has 1M edges which equals to 5.3 GB/s input/output throughput. Similarly, the peak of 4-cliques (also at 1M edge graph) is 32 Medges/sec which is around 0.6 GB/s. For complex queries such as clique listing which involves a lot of searching, the achieved throughput is smaller than the bandwidth of PCIe-3.0.

The GraphLab distribution provides a tool for counting triangles. It is used to compare with the GPU triangle performance (on the same hardware). This tool does not use general-purpose join algorithms to compute triangles; instead a a vertex-centric programming model is used where the triangle counting algorithm is written directly in C++. 4-cliques performance is not compared because the lack of a highly tuned 4-cliques implementation for GraphLab.

In comparison with GraphLab, the load-time is excluded and only the triangle counting time is included. For the 30M (100M) datasets, GraphLab took 7.8s (42.0s) while LFTJ-CPU required 2.1s (9.2s). For smaller datasets LFTJ-CPU was even faster, averaging an improvement of 7X for data size from 10K to 100M. To compare relative speeds without

multi-core-effects, both implementations are tested with the configuration to use only a single-thread. Here, the runtime-numbers for the 30M (100M) dataset are: 27.4s (124s) for GraphLab and 12.8s (51.9s) for LFTJ-CPU.

It is important to note that

1. GraphLab allows using cluster-parallelism and will then scale to much larger datasets.

2. GraphLab does not require the input data to be sorted as LFTJ does. However, even with sorting LFTJ-CPU outperforms GraphLab on our single machine. Focusing on the 30M (100M) dataset: using the CUDA thrust library, sort takes 0.64s (2.5s) on the GPU; Building the TrieArray data structure is not parallelized or optimized. The serial CPU implementation here takes 0.23s (0.76s).

5.5 Summary

This chapter presents a multi-predicate join algorithms for the GPU by adapting a worst case optimum CPU algorithm into the GPU. The new algorithm exposes more fine-grained parallelism and improves the memory access patterns. Its performance is evaluated by running triangle listing and 4-cliques listing in undirected graphs on synthetic datasets. The benchmarks show that the GPU-Optimized algorithm outperforms the naively parallelized original LFTJ implementation when run on the GPU, which in turn outperforms the same implementation run on the CPU, which outperforms GraphLab on the same hardware. Compared with binary joins used by Red Fox, the multi-predicate join is faster and can run much larger graphs.

CHAPTER 6

SUPPORTING OUT-OF-CORE DATA SETS

The last part of the dissertation focuses on the problem of executing relational queries over out-of-core data sets in a heterogeneous system which resembles relational computations in a real world system. The previous three chapters show that GPUs can improve the performance of relational computations when data fit in the GPU memory and the achieved throughput is smaller than PCIe bandwidth when the query is complex. When the data set is larger than the GPU memory size or even the host memory size, the impact of PCIe and the disk I/O have to be considered.

Figure 46 shows the whole memory hierarchy of such a heterogeneous system. This chapter focuses on using solid-state drive (SSD) as hard disk so that SSD is shown as the last level of storage. Comparead with traditional hard disk, SSD has less access time and latency. Its power consumption is also low. Therefore, SSD may replace hard disk for relational computation in the future when SSD price drops since database applications are normally considered as I/O bound. In Figure 46, the higher level of storage is closer to the GPU computation cores and usually has higher bandwidth, but smaller capacity. The bandwidth of the shared memory and register file of GPU can be over 1TB/s. Thus, the higher level storage can be viewed as the cache for the lower level storage. Note again, the bandwidth of PCIe is much smaller than the GPU host memory which makes it an inevitable bottleneck for GPGPU applications.

In order to achieve the best performance, executing complex queries over large amount of data in a heterogeneous system as shown in Figure 46 requires using the high throughput of GPU to overcome the data movement overhead through the whole memory hierarchy. More specifically, it involves the following aspects.

- Partition the data into slices that fit the GPU memory. A tradeoff has to be made between total partition number and partition size in order to minimize data overlapping

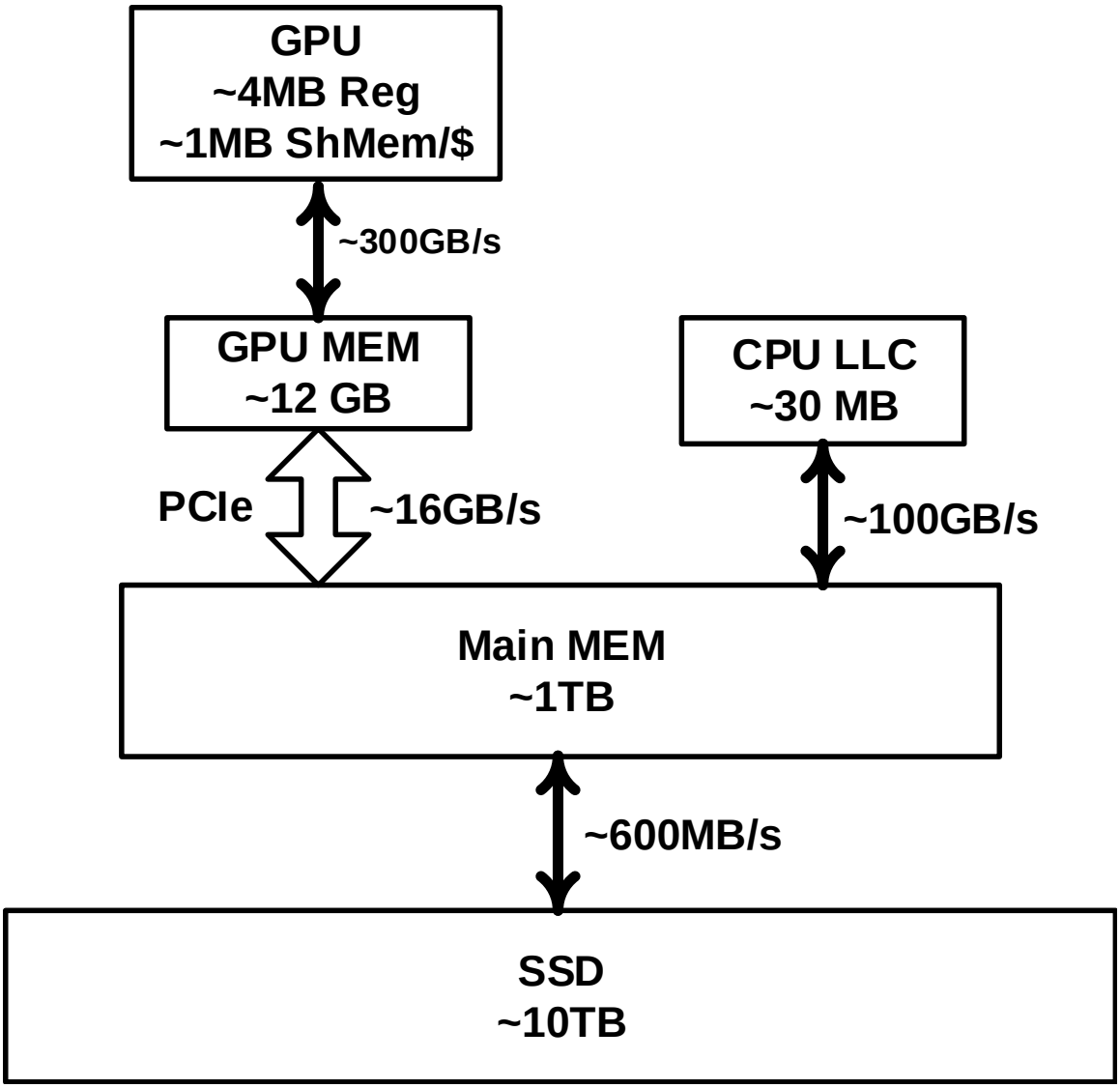

Figure 46. The memory hierarchy of a GPU accelerated heterogeneous system.

between different partitions.

- Schedule the primitives so that the result of previous primitives can be cached in the GPU memory and used by the following primitives. Otherwise, the GPU data have to be swapped with the CPU data which may decrease the performance.

- If the result or intermediate data of a primitive is too large to be held in the GPU memory, a solution is required to decide when and how to overflow these data to the host memory.

- If more than one GPUs are used, the partition algorithm is also responsible for workload balance among these GPUs.

- Large amount of data need to be moved from SSD to GPU cores. Data movement overhead in each level of storage should be either reduced through algorithm optimization or hidden by the computations.

This chapter introduces the design of a framework that supports relational computations over out-of-core data set. By implementing this framework and evaluating it in terms of the effectiveness in utilizing memory bandwidth that are brought up by the above data movement problems, it provides the measurement, learnt lessons, insights that are valuable for designing a real commercial system.

A prototype runtime system from LogicBlox Inc. is leveraged as the CPU-side data and task manager to control the execution of relational computations on GPUs. As shown in Figure 47, in the extended system, the GPU is treated as another powerful computation core in addition to the CPU cores. Thus, the existing CPU side partition and schedule can be used directly without any change as long as GPU shares the same programming interface with CPU. The reason of choosing this prototype system is that it uses TrieArray as data structure and LFTJ as main primitive which are compatible with the highly efficient GPU-optimized LFTJ designed in Chapter 5. By using this prototype, it is also demonstrated that the integration of GPUs into an existing CPU based system can reuse most of the existing components which substantially reduces the engineering effort. The chapter makes the following specific contributions:

- Integration of a complex GPU algorithm into a complete CPU system;

- Design and implementation of the double buffer approach to hide the PCIe and CPU overhead by the GPU computation;

- Demonstration of using multiple-GPUs in a heterogenous system to execute relational computations;

- Quantification of the efficiency of using GPUs to evaluate out-of-core data sets;

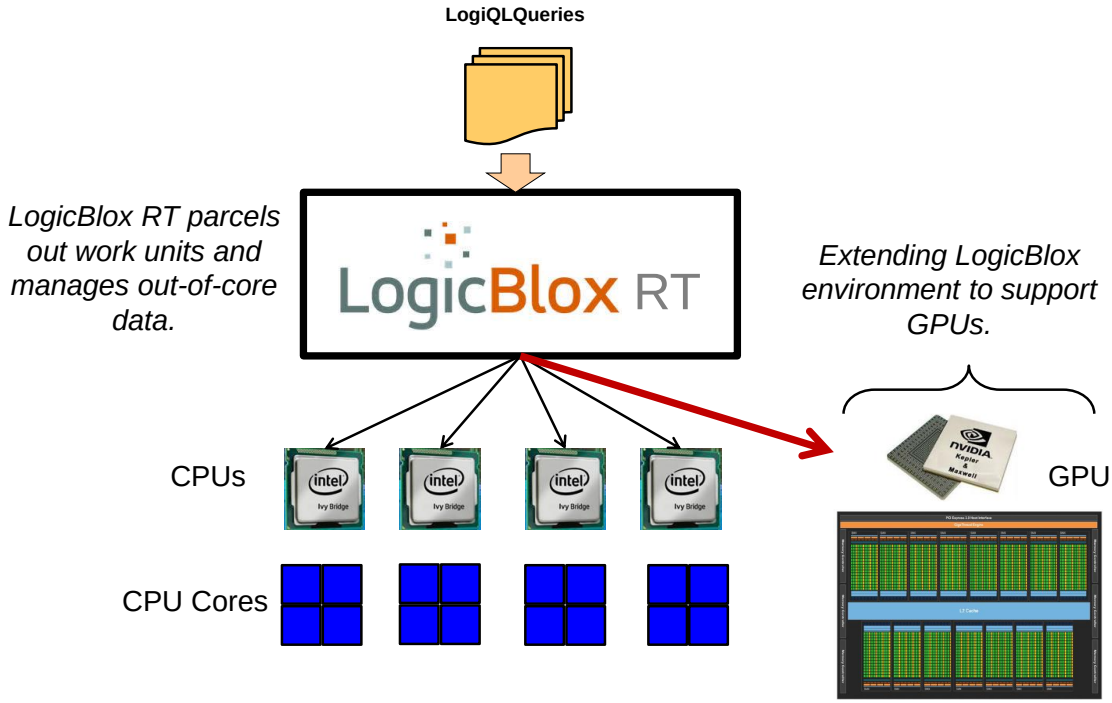

Figure 47. GPU is viewed as another high throughput core.

This chapter first briefly introduces the partition algorithm from LogicBlox Inc. in Section 6.1. Next, Section 6.2 introduces how to connect two GPUs to the CPU system to accelerate the execution of LFTJ. After that, Section 6.3 introduces how to use CUDA streams to hide the PCIe data movement latency. The last section shows the experiment result.

6.1 Introduction of the Partition Algorithm

This chapter describes the procedure that the GPU-Optimized LFTJ algorithm is integrated into a prototype runtime system to handle out-of-core data sets. Figure 48 shows the execution flow of this runtime when running a LogiQL rule such as triangle listing over out-of-core data set. The first step is to parses the input commands. The available commands include transforming data format, printing status, rule evaluation, etc. If the command is rule evaluation, it is fed into the slicer with necessary information carried by the command

parameters such as rule configuration and input data. As discussed previously, out-of-core input data sets reside on the SSD, which are partitioned by the slicer to find a portion that can be evaluated by the following in-memory LFTJ. After the partitioned data are evaluated, LFTJ returns to the runtime and the runtime calls the slicer again to find the next partition to process. This procedure continues until all the data are processed.

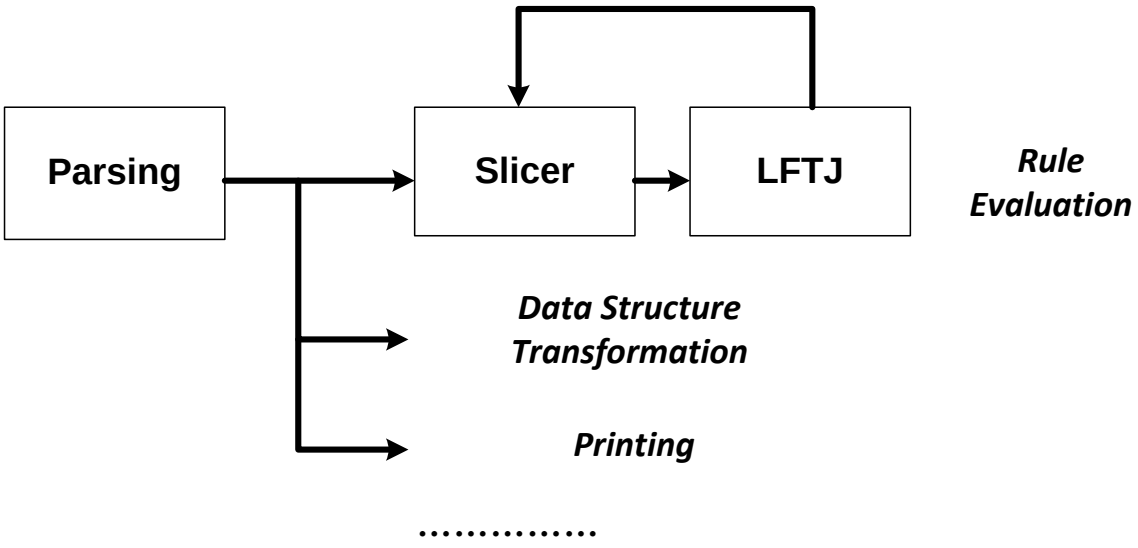

Figure 48. The execution flow of the prototype runtime.

One of the most important part of this runtime system is the slicer. It executes a partition algorithm [86] which manages the data movement through the hierarchy shown in Figure 47. This algorithm is developed by LogicBlox Inc. and is described here for completeness of the chapter.

In the beginning, the out-of-core data are stored in the TrieArray format as defined in 5.2 in the secondary storage which is SSD in this chapter. The TrieArray has n variables in the order of $x_1, \ldots, x_n$. In the case of triangle listing, the three variables are x, y, z in Figure 34. Directly executing LFTJ over these data causes excessive I/O operations because the LFTJ algorithm may access arbitrary data in the TrieArray at anytime which leads to frequent page faults.

The partition algorithm partitions the n-dimensional search space into *boxes*. Each

box has a lower bound and a higher bound for each variable (dimension) so that each box contains only the data fitting the host memory to be processed by an in-memory LFTJ algorithm. Figure 49 uses triangle listing as an example to illustrate this process. Figure 49(a) is the graph used for this example. Figure 49(b) is the corresponding Trie. Figure 49(c) is an example of dividing the data into seven boxes. The upper part of Figure 49(d) shows one of the boxes. In this box, every variable x, y, z is given a lower bound and a higher bound. The default lower bound is $-\infty$ and the default higher bound is ∞. The three predicates E(x,y), E(x,z), E(y,z) used to compute the triangle listing must slice the data so that every variable x, y, z falls into the scope limited by the lower and higher bound. The bottom of Figure 49(d) shows the *slices* of E(x,y), E(x,z), E(y,z) of this box. The three sliced tries are much smaller than the original one so that they can be stored in smaller buffers.

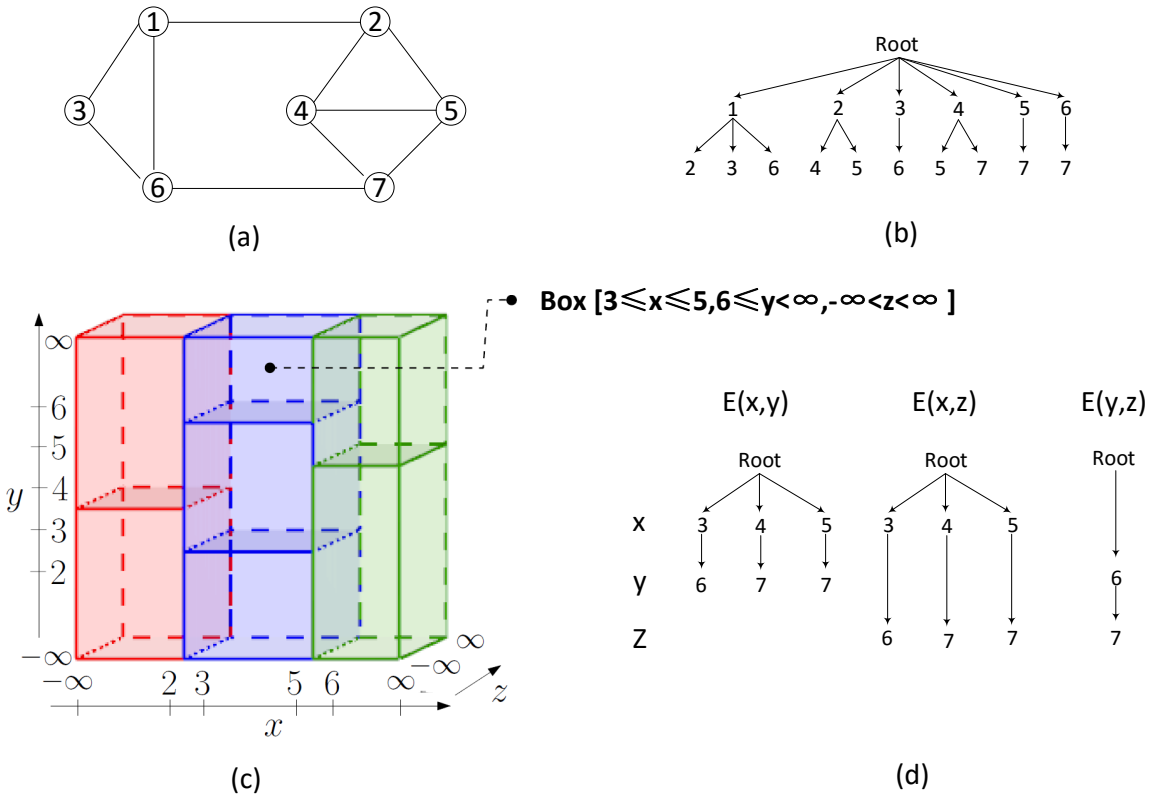

Figure 49. Example of boxing for using LFTJ to list triangles. (a) original graph. (b) TrieArray representation. (c) Boxed search space. (d) Example box and Trie slices.

Figure 50 is another example of the boxing procedure. This example partitions the data

of Figure 32(c) into two boxes. The first box is $[-\infty < x \leq 1, -\infty < y < \infty, -\infty < z < \infty]$ and the second is $[1 < x \leq \infty, -\infty < y < \infty, -\infty < z < \infty]$. This partition only splits the data of variable x into halves and the other two variables are remained intact. Figure 50 shows the original data on the left, the three slices of the first box at the top and the three slices of the second box at the bottom. Note that the the tries built for different boxes have some duplicated data which is E(y,z) in this example because in-memory LFTJ needs data from all predicates to conduct its computation and neither variable y nor z are partitioned in this example.

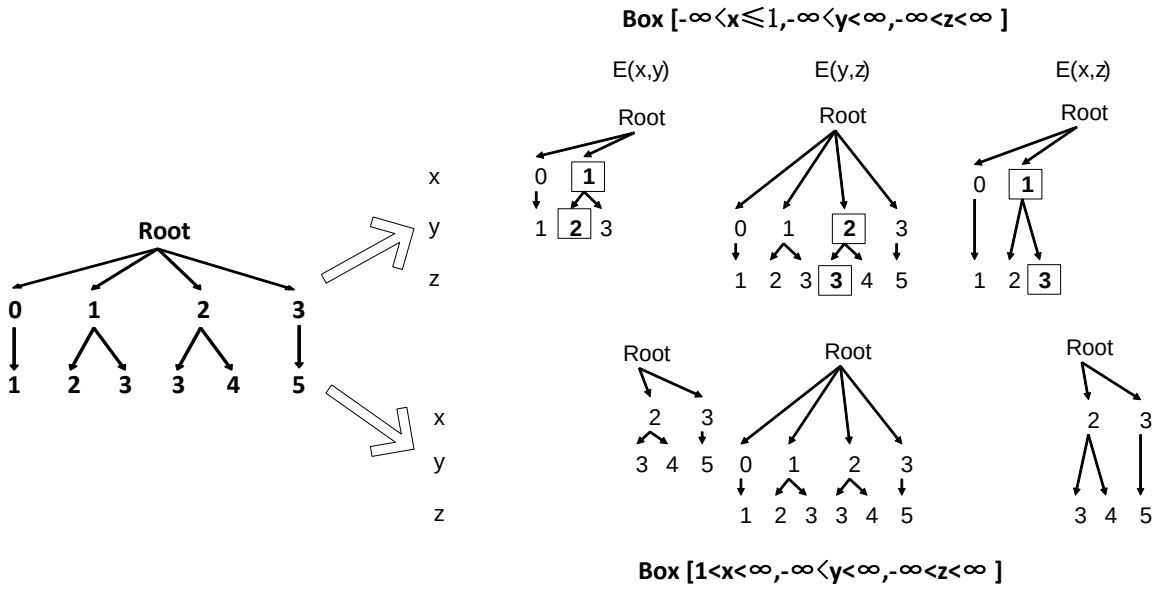

Figure 50. Example of partitioning Figure 32(c) into two boxes. Only the first box contains a triangle which is marked by a square.

The partition algorithm itself is straightforward and elegant. The input of the algorithm is the lower bounds for all variables and available memory size to store the partitioned data. The algorithm finds the upper bounds for all variables so that the box formed by the lower and upper bounds creates the largest slices whose total size is smaller than the given memory size. This procedure is also referred as *probing*. The initial lower bounds are $-\infty$. The algorithm finds one set of upper bounds and forms one box. The found upper bounds of the current box become the lower bounds of the next box. The partition algorithm is

called by the runtime until the found upper bounds are ∞. The process of searching the upper bounds is implemented as a simple binary search. When a box is found, new slices of the predicates are created in the CPU memory by copying the data of each variable falling between the lower and upper bound from the original TrieArray into the TrieArrays of the new slices. The copied *ptr* array of the TrieArray are adjusted accordingly. More details of the algorithm including the pseudo code can be found in [86]. [86] also proves the following features regarding its I/O efficiency:

- The partitioning algorithm is I/O efficient and worst-case optimal.
- It still maintains the original CPU computation complexity of LFTJ.
- When running triangle listing over large graphs, it can match the I/O complexity of the state-of-art specialized graph algorithm [87].

When the system contains more than one processor, e.g. two GPUs, each processor requires its own slicer to partition the data. Every slicer is executed by a separate CPU thread which is controlled by the main CPU thread of the runtime. The slicer is implemented as multi-thread safe so that each box can only be fetched by one slicer. The current implementation of the slicer is simplified and has load balance issue. Some slicers may have more partition work to do than others. This problem may even be intensified if the computation processors have large difference in computation throughput (e.g. CPU vs. GPU).

6.2 System Overview

The experiment environment of this chapter is the same as Chapter 5 except that it uses two GPUs: a GTX Titan and a Tesla K40c. Compared with GTX Titan, K40c has more memory (12GB), larger number of cores (2880), and smaller memory bandwidth after enabling error-correcting code (ECC). Overall, it has similar throughput as the GTX Titan when running LFTJ.

The overall system diagram is shown in Figure 51. The runtime executed by the main thread spawns several new threads to manage the attached GPUs if the received command is to run relational computation. The number of spawned threads equals to the number of GPUs used to perform the computation. Therefore, each spawned thread only needs to manage one GPU. Note that the GPU management threads run on the CPU rather than on the GPUs, and the CUDA kernels invoked by these threads are executed on the GPUs. The spawned thread needs to perform several management tasks including creating CUDA context, binding itself to a specific GPU, explicitly managing the GPU memory, and transferring data between CPU and CPU. One of its most important tasks is to invoke the slicer and GPU-Optimized LFTJ. After the slicer gets the new slices of TrieArrays stored in the host memory, these slices are copied to the GPU memory. Since the GPU-Optimized LFTJ shares the same interface with the original CPU LFTJ, the GPU multi-predicate join can be integrated directly to the runtime without any changes. The iterations between these tasks are similar as the original CPU execution flow shown in Figure 48 where the GPU management threads complete after all the data are processed.

One input parameter of the slicer to be determined is the size of the memory space required to store the slices which is referred as box size. If using smaller box size to store the partitioned data, more boxes have to be created. The tradeoff between box number and box size is that smaller box size is easier for achieving load balance and large box has lower partitioning overhead. The smaller the box is, the less time is needed to process the data in the box so that the processors have less idle time waiting for other busy processors. However, using smaller box size requires calling more box procedures which leads to more data movement overhead.

Usually the smaller the box is, the higher ratio of the duplicated data exist in the slices and the larger the number of boxes grows. For example, imagine the x variable of Figure 32(c) is splitted into four partitions or four smaller boxes, with each box having a E(y,z) slice which is not affected by the partition and is as large as the original trie. Figure 52(a)

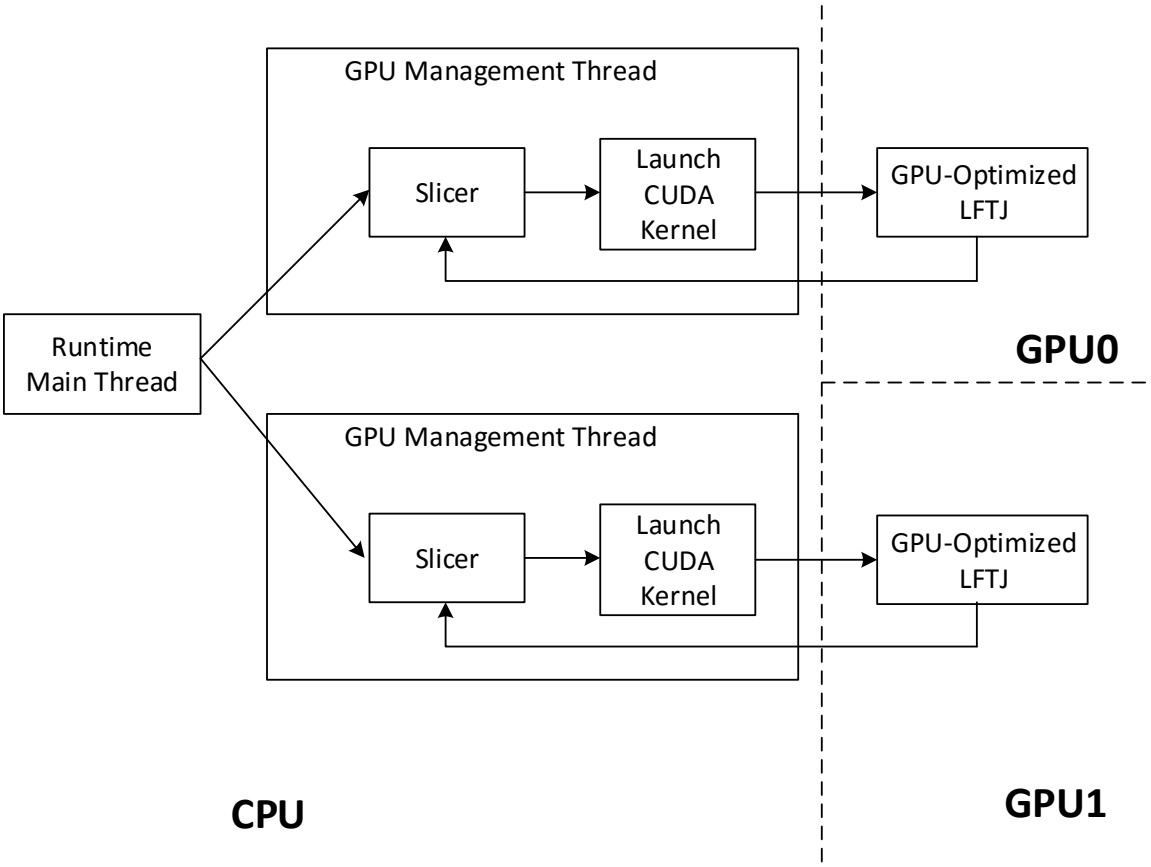

Figure 51. The execution flow of paritioning data in the CPU and running LFTJ on two GPUs.

shows the relations between box size and box number of graph LJ, one of the benchmark graphs that are introduced in Section 6.4. The horizontal axis is the box size where from the left to the right each box size is half of the previous one. The solid line represents the the resulting box number for different box size. The dotted line grows in the power of two. The figure demonstrates that the box number grows far more than twice if reducing the box size by two.

Figure 52(b) shows the relations between box size and the total execution time which equals to the product of box number and processing time per box. If the processing time of each box is linear with the box size, the total execution time dramatically increases as the box size goes down. As shown in Chapter 5, the actual processing time of LFTJ in GPU is between $O(n)$ and $O(nlog(n))$ which means using large box size needs much less total execution time. Plus the less overhead of partitioning itself, the advantage of using large

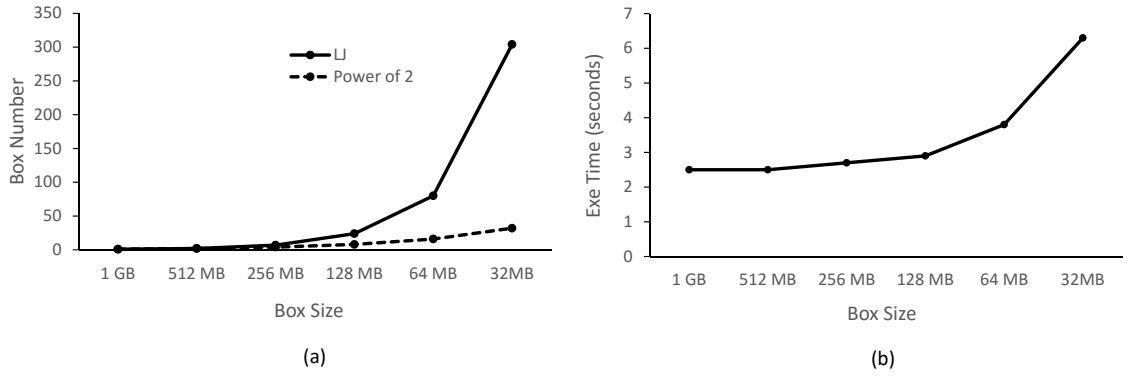

Figure 52. Relations between box size, box number and execution time.

box size is overwhelming. Thus, this chapter chooses to use large slice size to reduce the box number.

6.3 Concurrency Improvement

CUDA streams can be used to further improve the performance by increasing the concurrency when running large input data sets. However, concurrent execution on the GPU is not always beneficial. When executing two kernels concurrently on a device, each kernel nominally has access to only half the resources (CTAs and threads). Figure 53 uses the simple SELECT operator from [74] to illustrate this point. The line *no stream (old)* is the original implementation of SELECT that passes 50% of the elements. The line *no stream (new)* is the same implementation as *no stream (old)* but uses half the number of threads and CTAs. The performance of *(new)* is worse than *(old)*. The line *stream* uses CUDA stream to concurrently run two independent SELECTs each using the same design as *(new)* (the element number in the Figure is the total element number of both SELECTs). The performance of *(stream)* is better than *(new)* since two SELECTs can run concurrently. However, *stream* is worse than *(old)* when number of elements exceeds 8 million. It shows that concurrency is beneficial only when number of elements is small because less data parallelism exists. For large numbers of elements, concurrent stream execution is not advantageous. In cases like this, using more threads in a single kernel is better than using a smaller number of threads

in concurrent kernels.

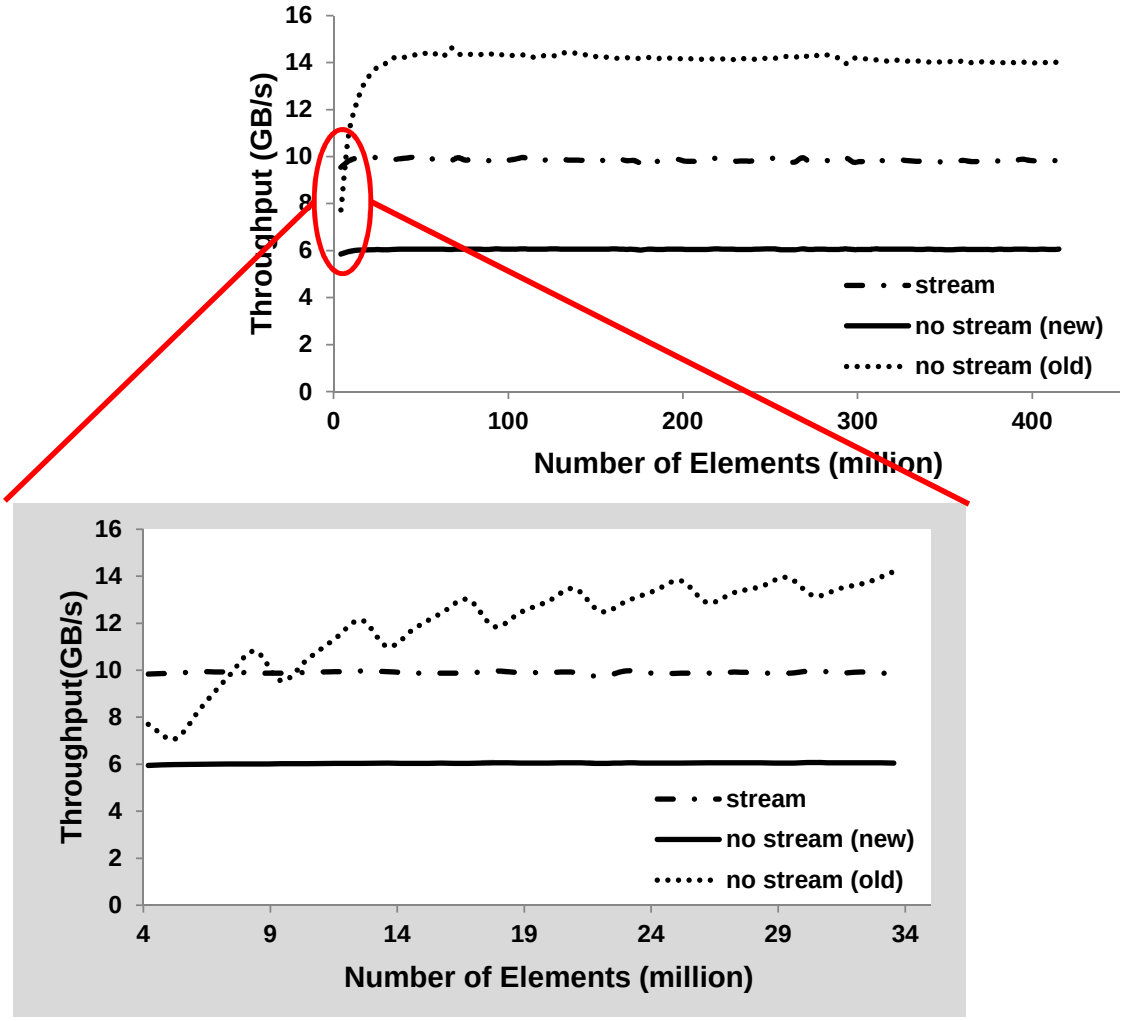

Figure 53. Performance of concurrently executing two selections (Tesla C2070).

Alternatively, another way to concurrently use the GPU is to use double buffering to partition the data set into segments and overlap the data transfer of one segment with the execution of another segment. Thus, the PCIe transfer is hidden by overlapping communication and computation. This is especially useful when the element number is large. This method is called Kernel Fission below because it is an opposite operation to Kernel Fusion.

The experiment GPU device in this section, the NVIDIA Tesla C2070, can overlap two PCIe transfers with a computation kernel which means the following three events can happen at the same time: one stream is downloading data to GPU, the second stream is

computing and the third stream is uploading result to the CPU. For such a device, at least three streams are needed to fully utilize its concurrency capacity.

Figure 54 compares the performance of kernel fission with serial execution using one SELECT operator. The data set used here is very large exceeding the size of GPU memory since these are the cases of interest. Note that the GPU's 6GB memory can hold less than 1.5 billion 32-bit integers. On average, the throughput achievable with kernel fission is **36.9%** better than the baseline version which demonstrates that pipelining can provide significantly improved performance over simply concurrently executing the kernels (*stream* of Figure 53).

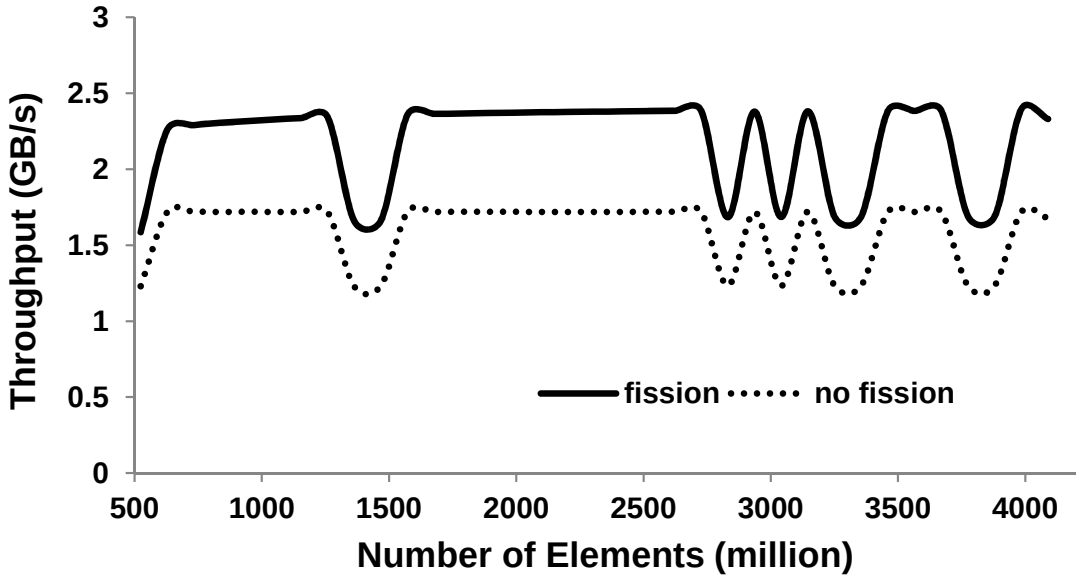

Figure 54. Performance of kernel fission.

In principle, the execution time of kernel fission is the maximum of CPU→GPU transfer time, GPU compute time, and GPU→ CPU transfer time. Taking SELECT as an example, the maximum is typically the input transfer time because the result of SELECT is smaller than the input, and the operator itself is computationally simple. Thus, the performance of running one SELECT with kernel fission is relatively insensitive to the fraction of the operator taken by the filter operation. The drawback of kernel fission is that

- It has to use pinned memory to transfer data which may hurt the CPU performance by reducing the available memory of CPU to perform other critical system tasks.
- More GPU memory has to be used to buffer the input and/or output data.

As to triangle listing, three events can run concurrently, CPU partitioning, PCIe transferring, and GPU computation. Considering the fact that GPU computation dominates the overall time, the integrated system put the first two events in one CUDA Stream and LFTJ execution in the other stream as shown in Figure 55. Thus, the time spent in preparing data for GPU is hidden by the computation.

Figure 55. Use double buffer to hide CPU partitioning and PCIe data movement.

6.4 Experiment Evaluation

This section focuses on the triangle listing problem on a heterogeneous system that has a multi-core CPU and two GPUs: one GTX Titan and one K40c. The experiments investigates and analyzes the overhead of partitioning, its behavior with limited available main memory and its performance compared to best-in-class competitors.

The evaluation environment is introduced in the Section 6.2. The same set of graphs used to evaluate the CPU partition algorithm [86] is evaluated again to test the efficiency of GPUs. Table 9 lists the characteristics of these graphs. The first row of the graph shows the graph name. Both real-world and synthetic input data of varying sizes are used. The real-world data include graphs from online community: "LiveJournal (LJ)", "Orkut" and "Twitter". The smallest dataset is "LJ", which contains the friendship graph of the online blogging community LiveJournal [88, 89]. Next, "Orkut" is the friend-ship graph of the free online community Orkut [90, 88]. "TWITTER" is one of the largest freely available graph data sets which contains the as-of-2010 "follower" relationships [91, 92]. The TWITTER graph is a power law graph in which some nodes have much larger node degree than the orther nodes. GPU-Optimized LFTJ is not expected to perform well in this graph since the computation complexity of linear search is too inefficient for this type of workload. "RAND" and "RMAT" are synthesized graphs and have better understood characteristics. Each of them comes in a medium-sized version (RAND16 and RMAT16) with 16 million nodes and 256 million edges and a large version (RAND80 and RMAT80) with 80 million nodes and 1.28 billion edges. In the RAND dataset, the edges are created by uniformly randomly selecting two endpoints from the graph's nodes. The RMAT data contains graphs created by the Recursive Matrix approach as proposed by Chakrabarti et al. [93] which closely matches real-world graphs such as computer networks, or web graphs. The graph is generated by using the data generator available at [94] with its default parameters. The LiveJournal and the synthetic graphs were also used by the MGT work in [87] and earlier work [83] to evaluate out-of-core performance for the triangle listing problem. The second row of Table 9 shows the graph size after transforming them into TrieArray data structure.

The execution time for the TrieArray-based implementation of LFTJ and two competing algorithms on the above experiment data sets with various configurations and memory restrictions is measured and presented. The TrieArray-based experiments are conducted on both CPU and GPUs.

Table 9. Characteristics of the used data sets. The CSV sizes refer to the raw data stored as flat array in ASCII. TA stands for the TrieArray representation stored in binary. $|V|$ is the vertex number in the graph. $|E|$ is the edge number. #Δ is the number of found triangles.

	LJ	Orkut	RAND16	RMAT16	RAND80	RMAT80	Twitter
csv	500MB	1.8GB	4.1GB	4.0GB	22GB	22GB	25GB
TA	315MB	1.2GB	2.3GB	2.2GB	11GB	11.2GB	10GB
$\|V\|$	4 Mio	3 Mio	16 Mio	16 Mio	80 Mio	80 Mio	42 Mio
$\|E\|$	35 Mio	117 Mio	256 Mio	256 Mio	1.28 Bill	1.28 Bill	1.2 Bill
$\frac{\|E\|}{\|V\|}$	8.7	38.1	16	16	16	16	28.9
#Δ	178 Mio	628 Mio	5457	2.2 Mio	5491	884,555	35 Bill

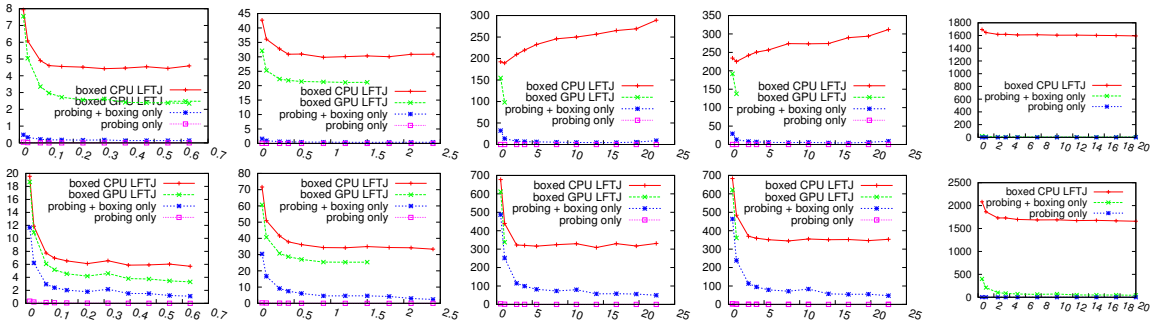

Figure 56. Computation and I/O overhead of the partition algorithm. Omitted graphs for {RAND—RMAT}16 as the look like the "80" variants. The X axises are the fraction of the input data size. For example, 0.1 means the available memory for boxing is 10% of the total input size. The Y axises are the execution time in seconds.

Figure 56 shows the computation and I/O overhead of the partition algorithm. The X-Axis varies according to the box size available for the partition shown in GB. The individual points range from $5, 10, \ldots$ up to 200% of the input data size in TrieArray representation. The first row of Figure 56 tests the computation overhead of the partition algorithm. LFTJ can only use memory up to the size of a fraction of the input during execution. To further (almost completely) remove I/O, all input data are *cat*-ted to `/dev/null`, which essentially pre-loads the Linux file-system cache. The partition is performed in one CPU thread and CPU LFTJ runs with eight threads to fully exploit the four cores of the CPU. Four variants are executed for every benchmark: (1) the full CPU LFTJ, (b) the full GPU LFTJ, (3) searching and copying the partitioned data without running LFTJ (probing + boxing), and

(d) only searching without copying input data nor running LFTJ (probing only).

The CPU work performed for searching and copying is trivial in comparison to the work done by the CPU or GPU join evaluation, even when the box sizes are limited to as little as 5% of the size of the input. The second top lines of the five figures in the first row represent the performance of executing GPU-Optimized LFTJ on K40c. The largest box size supported by GPU is much smaller than CPU because GPU has much smaller memory size and GPU-Optimized LFTJ uses much larger memory footprint. The GPU number is not shown in the TWITTER graph because GPU performance is more than 10x slower than its CPU counterpart for this power-law graph. However, GPU is 1.46-2.95x faster for the remaining graphs even when CPU uses 200% boxes. The CPU overhead of searching and copying is still less than 15% when LFTJ is performed in GPU. Results from [86] show that the general LFTJ together with its partition algorithm is three times slower than the specialized graph algorithm in the out-of-core setting. Using GPUs can reduce the gap or even match with these specialized algorithm. Both the productivity and performance portability provided by this framework enables the domain expert to still use their general tools while enjoying the computation power provided by the accelerators such as GPUs.

The performance of the partition algorithm when disk I/O needs to be performed is the main concern of evaluating out-of-core data. The same experiments as the first row of Figure 56 are conducted again but all linux system caches are cleared before starting a run. The total amount of memory used for the program (data+instructions) and any caches used by the operating system to buffer I/O on behalf of the program are further limited by using Linux's *cgroup* feature. The actual used limit is the value given to the partition and shown on the X-Axis plus a fixed 100MB (that accounts for the output buffer and the size of the executable). Results of this set of experiments are shown in the second row of Figure 56. These results are end-to-end which include all the overhead that happens in real world when running out-of-core data set. In this scenario, the cost of searching the box is still very cheap even for the 5% memory setting; Copying the data now has noticeable costs

for low-memory settings (25% and below) when executing LFTJ in CPU. However, even then, it is mostly dominated by the time to actually perform the in-memory joins. This is even more so for the real-world data sets. If executing LFTJ in GPU, the total execution time is still 1.30-1.73x faster than using the same box size to run CPU LFTJ for the first four graph. The ratio of searching and copying increases since GPU is faster in joining. Note PCIe overhead is not included in the paritioning part, but included in the GPU LFTJ execution. In two real-world graphs, LJ and Orkut, the overhead is still less than 1/3, while in two synthetic graphs, the overhead is between 66%-75% which indicating that the I/O is the bottleneck. In the latter cases, two levels of partition can be used: first find a large box from the secondary storage and then find small boxes to send to GPU from this large box. This should significantly reduce the I/O cost because the second row of Figure 56 proves finding large box has much smaller overhead than finding small box from SSD and the first row of figures proves that partitioning from memory has negligible cost.

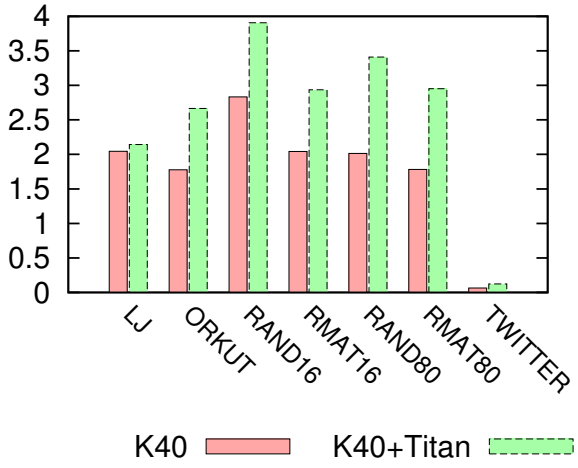

Figure 57. Speedup of using one or two GPUs against one CPU.

The last set of experiment demonstrates the capability of multi-GPUs on the boxing algorithm. Figure 57 shows the speedup of using one K40 and using both GPUs over the baseline which executes LFTJ in CPU by using 1GB box. K40 and Titan both use 512MB boxes so that the smallest graph, LJ, needs two boxes and can be executed concurrently in two GPUs. Memory is not restricted in this set of experiment. One K40 can still bring

around 2x speedup. When concurrently executing LFTJ on two GPUs, the speedup can reach 2-4x. Using two GPUs is always faster than using one GPUs while the scalability differs from one graph to another. For graph LJ, the use of an extra GPU only improves the performance by 5% because there is not enough data to saturate both GPUs. For all other benchmarks, the extra GPU can bring at least 40% additional speedup, particularly the largest TWITTER graph which has 90% additional speedup because of the low cost of boxing and full utilization of two GPUs executing the join algorithm.

6.5 Summary

This chapter discusses the problem of using GPU to accelerate the relational computation with the out-of-core data and provides one detailed solution which allows seamless integration of the GPU accelerated LFTJ algorithm with the advanced CPU data partition algorithm. This is a demonstration that the GPU accelerated database system can still reuse the major CPU components. Using triangle listing as an example, GPU can improve the performance of general relational join to catch up with the state-of-the-art specialized graph algorithms. Thus, domain expert users can solve graph problems with high productivity by simply executing relational computations in the proposed system. The overall integrated system can be optimized in several aspects to further improve the performance:

- Improve the partitioning algorithm to reduce data duplications between different slices.
- Improve the load balance between different slicer so that each processor only has to process the amount of data that is proportional to its throughput.
- Add a feedback-driven control system to adapt to different workloads, for example, by enabling detection of power law graphs and offloading more computation to the CPU.

CHAPTER 7

CONCLUSION

7.1 Summary

This thesis discusses several different aspects of running relational computations on GPUs including the basic query compilation, sophisticated memory optimization, complex algorithm design, and integration with existing CPU database system. As a conclusion, this thesis demonstrates that GPUs can bring significant speedup because of the potential parallelism existing in the relational computations, but requires extended effort in memory allocation and accesses due to the nature of low computation density and large volume of data to be processed. The insights of this thesis are summarized as below.

First, the static compilation of relational computations is a straightforward process by progressively lowering the domain specific language to high level of IR, then low level of IR, and finally the binaries that can be executed directly on the device. It is very similar to other language complies that are specifically targeted at GPUs such as Python [73] and JavaScript [95]. It is also similar to the compilation flow of traditional database system, although with a different backend which requires different primitive implementations. Therefore, the compilation itself is not the obstacle of using GPU in database. In fact, the existing query language front end and query optimization of the traditional CPU database system can still be utilized when executing relational computations on GPUs.

Second, the significant speedup of the GPU accelerated system mainly results from the efficient primitive design. The low computation density makes all primitives including join or even multi-predicate join very easy to implement with a simple and straightforward approach. However, an efficient GPU implementation could be substantially more complicated as the algorithm has to take care of various aspects including memory access patterns, load balancing, fine-grained parallelism, occupancy, ILP, etc. The nature of low computation density actually makes the algorithm design more difficult because the performance of

the primitives are more sensitive to the achieved memory throughput and workload imbalance. This thesis uses the following techniques to improve the performance:

- Use the shared memory to stage data to fully utilize memory coalescing. The algorithm is carefully adapted to reduce the use of random memory accesses.
- Use a sophisticated partition algorithm to assign workloads to the CTAs and then to the threads in the CTAs to guarantee the load balance in every level.
- Fully exploit the fine-grained parallelism. While it is easy for simple primitives such as SELECT and PROJECT, significant algorithm redesign are required for complex primitives such as multi-predicate join algorithm.
- Reuse shared memory and registers to increase the occupancy.
- Use loop unrolling, register blocking, and avoid bank conflict to improve the ILP.

By using the above techniques, all primitives are implemented as several kernels and most of these kernels are bounded by the memory. Only kernels that still require random memory accesses such as binary search kernels cannot fully utilized the GPU memory bandwidth.

Third, aggregating the computation can reduce the memory movement overhead and reduce the memory footprint. Kernel fusion and GPU-optimized multi-predicate join algorithm are two different approaches proposed in this thesis to achieve this goal. Kernel fusion explicitly removes the round trip data movement and memory allocation of intermediate data by staging the data closer to the computation cores. GPU-optimized LFTJ algorithm collaboratively processes all the predicates to reduce the size of the intermediate data. The first approach is more flexible and the second approach has better performance.

Fourth, although PCIe and disk I/O could be the throughput bottleneck of individual standard primitive on GPU, they are not the limitation for the complex queries such as TPC-H or clique listing which are the focus of this thesis. The throughput of individual

standard primitive on GPUs is much higher than the bandwidth of PCIe. For example, binary join can reach over 30GB/s. Therefore, PCIe or disk I/O is the bottleneck depends on where the data are stored. However, if the computation for the same piece of data is complex enough, the achieved throughput is much smaller than an individual primitive and the GPU computation can dominate the overall performance. For this type of relational computation, it is important to (1) cache the data in GPU so that more computation can be executed over the same piece of data and (2) reduce the memory footprint so that more data can be processed.

Fifth, the GPU can be viewed as another high throughput core when adding it to an existing CPU-based system. However, the runtime system should consider the difference of the cost model as wells as the data transfer overhead between CPU and GPU.

7.2 Future Work

Looking forward, there are still many areas that can be explored. First of all, many primitives are required to handle different data distribution which determines the memory access patterns. As GPUs are more sensitive to the memory access patterns than CPUs, algorithm designs for such primitives as well as choosing the right primitives (either statically or dynamically) are the important problems that need solutions.

Second, applying many existing database technologies to GPUs would also provide many new opportunities. For example, data compression and decompression can reduce the data movement across the CPU-GPU memory hierarchy. However, compression and decompression are highly data dependent so that it is not easy to implement them on GPUs. Another example is query optimizations which are traditionally used to find efficient query plans to execute on CPUs. Adding GPUs to the whole picture would introduce more variables when optimizing the query plans.

Third, GPU architecture is evolving in a very fast pace. Every new generation of GPU requires fundamentally redesign of the primitives. For example, tuning for NVIDIA Kepler

GPUs is very different from tuning for NVIDIA Fermi GPUs mainly because of the different ratio between register file and shared memory. Future GPUs from NVIDIA, AMD, and Intel will introduce the stacked memory which has much higher capacity and bandwidth. These changes are fundamental to relational computation and could bring significant speedup. However, the penalty of not efficiently utilizing the new memory hierarchy also becomes higher on these new generation of GPUs. Therefore, efficient primitive design and memory optimization are more critical on these new devices. Other examples of new GPU features that may have significant impacts on relational computations are unified virtual memory, integrated CPU-GPU system and high bandwidth connection such as NVLink [96]. These features can enlarge the GPU memory address space and reduce or remove the drawbacks of PCIe.

Last, while not investigated or discussed in this thesis, changes in GPU architecture could potentially make GPU more suitable for relational computations. For example, near memory computation and power consumption are two areas that are worthwhile to explore. The former could substantially change the way that computation and memory operations interact with each other, which may lead to performance improvement. The latter would require changes in the microarchitecture or circuit level in order to increase the energy efficiency of relational computations.

Overall, GPUs provide new opportunities for relational computations and demonstrate their efficiency in the prototyped system illustrated in this thesis. However, using GPUs in real database system products would require to solve many more remaining research and engineering problems.

REFERENCES

[1] T. L. Veldhuizen, "Triejoin: A simple, worst-case optimal join algorithm," in *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pp. 96–106, 2014.

[2] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *J. Comput. Phys.*, vol. 227, pp. 5342–5359, May 2008.

[3] J. Mosegaard and T. S. Sørensen, "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu," in *Proceedings of the 11th Eurographics Conference on Virtual Environments*, EGVE'05, (Aire-la-Ville, Switzerland, Switzerland), pp. 105–111, Eurographics Association, 2005.

[4] S. Solomon, R. Thulasiram, and P. Thulasiraman, "Option pricing on the gpu," in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pp. 289–296, Sept 2010.

[5] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: A general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, pp. 66:1–66:13, July 2010.

[6] Amazon, "Amazon elastic compute cloud user guide for linux." `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-ug.pdf`, 2014.

[7] CompuGreen, "The green500 list - november 2014." `http://www.green500.org/printpdf/16929?green500from=1&green500to=100`, November 2014.

[8] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377–387, June 1970.

[9] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili, "Red fox: An execution environment for relational query processing on gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 44:44–44:54, ACM, 2014.

[10] S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: An interactive tutorial," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, (New York, NY, USA), pp. 1213–1216, ACM, 2011.

[11] NVIDIA, "Cuda c programming guide." `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3YfVqMLiQ`, 2015.

[12] Council, T.P.P., "Tpc benchmark h, standard specification revision 2.16.0." `http://www.tpc.org/tpch/spec/tpch2.16.0.pdf`, 2013.

[13] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 107–118, IEEE Computer Society, 2012.

[14] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar, "Optimizing data warehousing applications for gpus using kernel fusion/fission," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, (Washington, DC, USA), pp. 2433–2442, IEEE Computer Society, 2012.

[15] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, (London, UK, UK), pp. 301–320, Springer-Verlag, 1994.

[16] H. Wu, D. Zinn, M. Aref, and S. Yalamanchili, "Multipredicate join algorithms for accelerating relational graph processing on gpus," in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 1, 2014.*, pp. 1–12, 2014.

[17] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, Aug. 1990.

[18] C. J. Date and H. Darwen, *A Guide to the SQL Standard (4th Ed.): A User's Guide to the Standard Database Language SQL*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[19] M. Stonebraker, "Sql databases v. nosql databases," *Commun. ACM*, vol. 53, pp. 10–11, Apr. 2010.

[20] G. F. Diamos and S. Yalamanchili, "Harmony: An execution model and runtime for heterogeneous many core systems," in *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, (New York, NY, USA), pp. 197–200, ACM, 2008.

[21] Khronos OpenCL Working Group, "The OpenCL Specification, version 2.0," October 2014.

[22] H. Gallaire and J. Minker, eds., *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, 1977*, Advances in Data Base Theory, (New York), Plemum Press, 1978.

[23] M. Seeger and S. Ultra-Large-Sites, "Key-value stores: a practical overview," *Computer Science and Media*, 2009.

[24] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 260–269, ACM, 2008.

[25] NVIDIA, "Parallel thread execution isa." http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz3YfVqMLiQ, 2015.

[26] LLVM Project, "Llvm language reference manual." http://llvm.org/docs/LangRef.html, 2015.

[27] NVIDIA, "Thrust." http://docs.nvidia.com/cuda/thrust/#axzz3YfVqMLiQ, 2015.

[28] C. J. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architectures for general-purpose computing: A framework and analysis," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, (Los Alamitos, CA, USA), pp. 306–317, IEEE Computer Society Press, 2002.

[29] A. Sherrod and W. Jones, *Beginning DirectX 11 Game Programming*. Boston, MA, United States: Course Technology Press, 1st ed., 2011.

[30] Khronos Groups, "The opengl shading language." http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf, 2014.

[31] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, (New York, NY, USA), pp. 215–226, ACM, 2004.

[32] R. Wu, B. Zhang, M. Hsu, and Q. Chen, "Gpu-accelerated predicate evaluation on column store," in *Proceedings of the 11th International Conference on Web-age Information Management*, WAIM'10, (Berlin, Heidelberg), pp. 570–581, Springer-Verlag, 2010.

[33] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 511–524, ACM, 2008.

[34] P. Trancoso, D. Othonos, and A. Artemiou, "Data parallel acceleration of decision support queries using cell/be and gpus," in *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09, (New York, NY, USA), pp. 117–126, ACM, 2009.

[35] E. A. Sitaridi and K. A. Ross, "Ameliorating memory contention of olap operators on gpu processors," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, (New York, NY, USA), pp. 39–47, ACM, 2012.

[36] S. Baxter, "Modern gpu." http://nvlabs.github.io/moderngpu/, 2013.

[37] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: A gpu merging algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 331–340, ACM, 2012.

[38] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, (New York, NY, USA), pp. 55–62, ACM, 2012.

[39] Intel, *Intel Open Source Graphics Programmers Reference Manual (PRM) for the 2013 Intel Core Processor Family, including Intel HD Graphics, Intel Iris Graphics and Intel Iris Pro Graphics*, 2014.

[40] N. Brookwood, "Amd fusion family of apus: Enabling a superior, immersive pc experience." http://www.amd.com/us/Documents/48423_fusion_whitepaper_WEB.pdf, 2010.

[41] B. Daily, "project denver processor to usher in new era of computing." http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/, 2011.

[42] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled cpu-gpu architecture," *Proc. VLDB Endow.*, vol. 6, pp. 889–900, Aug. 2013.

[43] D. Broneske, S. Breß, M. Heimel, and G. Saake, "Toward hardware-sensitive database operations," in *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pp. 229–234, 2014.

[44] Council, T.P.P., "Tpc-h - top ten performance results - non-clustered." http://www.tpc.org/tpch/results/tpch_perf_results.asp?resulttype=noncluster, 2013.

[45] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, pp. 21:1–21:39, Dec. 2009.

[46] S. Zhang, J. He, B. He, and M. Lu, "Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures," *Proc. VLDB Endow.*, vol. 6, pp. 1374–1377, Aug. 2013.

[47] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proc. VLDB Endow.*, vol. 3, pp. 670–680, Sept. 2010.

[48] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, (New York, NY, USA), pp. 94–103, ACM, 2010.

[49] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on gpu devices," *Proc. VLDB Endow.*, vol. 6, pp. 817–828, Aug. 2013.

[50] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "Performance evaluation and benchmarking," ch. The Star Schema Benchmark and Augmented Fact Table Indexing, pp. 237–252, Berlin, Heidelberg: Springer-Verlag, 2009.

[51] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang, "Concurrent analytical query processing with gpus," *Proc. VLDB Endow.*, vol. 7, pp. 1011–1022, July 2014.

[52] C. A. Martınez-Angeles, I. Dutra, V. S. Costa, and J. Buenabad-Chávez, "A datalog engine for gpus," *CHRISTIAN-ALBRECHTS-UNIVERSITAT ZU KIEL*, p. 239, 2013.

[53] H. Rauhe, J. Dees, K.-U. Sattler, and F. Faerber, "Multi-level parallel query execution framework for cpu and gpu," in *Advances in Databases and Information Systems* (B. Catania, G. Guerrini, and J. Pokorn, eds.), vol. 8133 of *Lecture Notes in Computer Science*, pp. 330–343, Springer Berlin Heidelberg, 2013.

[54] H. Pirk, T. Sellam, S. Manegold, and M. Kersten, "X-device query processing by bitwise distribution," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, (New York, NY, USA), pp. 48–54, ACM, 2012.

[55] H. Pirk, S. Manegold, and M. Kersten, "Waste not...efficient co-processing of relational data," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pp. 508–519, March 2014.

[56] P. Boncz, "Monet: a next-generation database kernel for query-intensive applications," 2002.

[57] E. A. Sitaridi and K. A. Ross, "Optimizing select conditions on gpus," in *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, (New York, NY, USA), pp. 4:1–4:8, ACM, 2013.

[58] T. Karnagel, M. Hille, M. Ludwig, D. Habich, W. Lehner, M. Heimel, and V. Markl, "Demonstrating efficient query processing in heterogeneous environments," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 693–696, ACM, 2014.

[59] S. Bre, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, "Exploring the design space of a gpu-aware database architecture," in *New Trends in Databases and Information Systems* (B. Catania, T. Cerquitelli, S. Chiusano, G. Guerrini, M. Kmpf, A. Kemper, B. Novikov, T. Palpanas, J. Pokorn, and A. Vakali, eds.), vol. 241 of *Advances in Intelligent Systems and Computing*, pp. 225–234, Springer International Publishing, 2014.

[60] S. Bre, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, "Gpu-accelerated database systems: Survey and open challenges," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV* (A. Hameurlain, J. Kng, R. Wagner, B. Catania, G. Guerrini, T. Palpanas, J. Pokorn, and A. Vakali, eds.), vol. 8920 of *Lecture Notes in Computer Science*, pp. 1–35, Springer Berlin Heidelberg, 2014.

[61] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake, "Automatic selection of processing units for coprocessing in databases," in *Proceedings of the 16th East European Conference on Advances in Databases and Information Systems*, ADBIS'12, (Berlin, Heidelberg), pp. 57–70, Springer-Verlag, 2012.

[62] S. Breß, S. Mohammad, and E. Schallehn, "Self-tuning distribution of db-operations on hybrid CPU/GPU platforms," in *Proceedings of the 24th GI-Workshop "Grundlagen von Datenbanken 2012", Lübbenau, Germany, May 29 - June 01, 2012*, pp. 89–94, 2012.

[63] S. Breí, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake, "Efficient co-processor utilization in database query processing," *Inf. Syst.*, vol. 38, pp. 1084–1096, Nov. 2013.

[64] S. Bre, E. Schallehn, and I. Geist, "Towards optimization of hybrid cpu/gpu query plans in database systems," in *New Trends in Databases and Information Systems* (M. Pechenizkiy and M. Wojciechowski, eds.), vol. 185 of *Advances in Intelligent Systems and Computing*, pp. 27–35, Springer Berlin Heidelberg, 2013.

[65] S. Breßand G. Saake, "Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms," *Proc. VLDB Endow.*, vol. 6, pp. 1398–1403, Aug. 2013.

[66] S. Bre, N. Siegmund, L. Bellatreche, and G. Saake, "An operator-stream-based scheduling engine for effective gpu coprocessing," in *Advances in Databases and Information Systems* (B. Catania, G. Guerrini, and J. Pokorn, eds.), vol. 8133 of *Lecture Notes in Computer Science*, pp. 288–301, Springer Berlin Heidelberg, 2013.

[67] S. Bre, "The design and implementation of cogadb: A column-oriented gpu-accelerated dbms," *Datenbank-Spektrum*, vol. 14, no. 3, pp. 199–209, 2014.

[68] M. Heimel and V. Markl, "A first step towards gpu-assisted query optimization," in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012.*, pp. 33–44, 2012.

[69] M. Heimel, "Designing a database system for modern processing architectures," in *Proceedings of the 2013 SIGMOD/PODS Ph.D. Symposium*, SIGMOD'13 PhD Symposium, (New York, NY, USA), pp. 13–18, ACM, 2013.

[70] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *Proc. VLDB Endow.*, vol. 6, pp. 709–720, July 2013.

[71] M. Heimel, F. Haase, M. Meinke, S. Breß, M. Saecker, and V. Markl, "Demonstrating self-learning algorithm adaptivity in a hardware-oblivious database engine," in *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pp. 616–619, 2014.

[72] S. Breß, B. Köcher, M. Heimel, V. Markl, M. Saecker, and G. Saake, "Ocelot/hype: Optimized data processing on heterogeneous hardware," *Proc. VLDB Endow.*, vol. 7, pp. 1609–1612, Aug. 2014.

[73] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: Compiling an embedded data parallel language," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 47–56, ACM, 2011.

[74] G. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili, "Relational algorithms for multi-bulk-synchronous processors," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 301–302, ACM, 2013.

[75] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[76] G. Diamos and S. Yalamanchili, "Speculative execution on multi-gpu systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, 2010.

[77] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for gpgpu stream architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 545–546, ACM, 2010.

[78] P. D. Vouzis and N. V. Sahinidis, "Gpu-blast: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.

[79] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[80] R. A. Patel, A. Davidson, Y. Zhang, J. D. Owens, and J. Mak, "Parallel lossless data compression on the gpu," in *In Innovative Parallel Computing*, p. 9, 2012.

[81] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, (New York, NY, USA), pp. 159–166, ACM, 2009.

[82] R. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," in *Combinatorial Pattern Matching* (S. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, eds.),

vol. 3109 of *Lecture Notes in Computer Science*, pp. 400–408, Springer Berlin Heidelberg, 2004.

[83] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, (New York, NY, USA), pp. 672–680, ACM, 2011.

[84] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, vol. 407, pp. 458–473, Nov. 2008.

[85] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured GPU applications," in *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, pp. 51–60, 2014.

[86] D. Zinn, "General-purpose join algorithms for listing triangles in large graphs," *CoRR*, vol. abs/1501.06689, 2015.

[87] X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 325–336, ACM, 2013.

[88] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," 2014.

[89] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, MDS '12, (New York, NY, USA), pp. 3:1–3:8, ACM, 2012.

[90] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, (New York, NY, USA), pp. 29–42, ACM, 2007.

[91] Twitter, "2010 twitter data set," 2010.

[92] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?," in *WWW '10: Proceedings of the 19th international conference on World wide web*, (New York, NY, USA), pp. 591–600, ACM, 2010.

[93] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining.," in *SDM*, vol. 4, pp. 442–446, SIAM, 2004.

[94] "R-mat data generator."

[95] J. Wang, N. Rubin, and S. Yalamanchili, "Paralleljs: An execution framework for javascript on heterogeneous systems," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, (New York, NY, USA), pp. 72:72–72:80, ACM, 2014.

[96] NVIDIA, “Nvidia nvlink high-speed interconnect: Application performance,” 2014.