Parallel graph processing on graphics processing units

Zhong, Jianlong

2014

Zhong, J. (2014). Parallel graph processing on graphics processing units. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/59975>

<https://doi.org/10.32657/10356/59975>

PARALLEL GRAPH PROCESSING ON GRAPHICS PROCESSING UNITS

JIANLONG ZHONG
SUPERVISOR: BINGSHENG HE

School of Computer Engineering

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

2014

Acknowledgments

First, I would like to express my sincere thanks to my supervisor Dr. HE Bingsheng for his remarkable guidance and generous help over the past four years. His expertise and invaluable advice has left a profound impact on my research career, especially the way I think of and solve new problems. His self-discipline and diligence has taught me how to bring out the best of myself under limited resources. I'm especially grateful for the great patience he showed when I had little progress, made mistakes, or even worked against him.

I appreciate my colleagues of Xtra Group for their help and companionship along the way. I'm honored to be the first student member of Xtra group and work with the following excellent new members: GONG Yifan, CHEN Changbing, HE Jiong, ZHOU Chi, CHEN Xuntao, HUYNH Bach Khoa, WEI Lei, NIU Zhaojie, LU Yanchao, LIU Cheng, LIU Haikun and Wang Zeke.

Special thanks should be given to NG-GOH Siew Lai, JIN Jiangming, ZHANG Yang, FAN Hongfei, TANG Shanjiang, XU Yi, LI Xiaosong, LAI Pan, ZHANG Wei and ZHAO Wenbo, my colleagues of the Parallel and Distributed Computing Center (PDCC) for their help.

I owe my deepest gratitude to my parents. The extraordinary perseverance they showed when facing difficulties in life and the great sacrifices they made for me have always been my inexhaustible motivation for working hard. I'm also lucky to have my brother as my partner in my journey to knowledge. Without the love and support from my family, today would not be possible.

My work is supported by the NTU Research Scholarship and a MoE AcRF Tier 2 grant (MOE2012-T2-2-067). I would also like to thank NVIDIA for donating hardware for my research.

Abstract

Graphs are de facto data structures for many applications, and efficient graph processing is a must for the application performance. Recently, the GPU (Graphics Processing Unit) has been adopted to accelerate many specific graph processing algorithms such as BFS and shortest path. This is because GPUs have an order of magnitude higher computational power and memory bandwidth compared to CPUs. However, superb hardware power of the GPU does not automatically lead to superb performance of graph processing. We are facing various challenges in efficiency and programmability of building parallel graph processing systems on GPUs. Despite the recently improved programmability of GPUs, it is difficult to write correct and efficient GPU programs and even more difficult for graph processing due to the irregularities of graph structures. Even worse, there lacks an efficient method and runtime system on the GPU to support concurrent graph processing tasks from multiple applications and/or users. To address those challenges, we develop the *Medusa* system to simplify parallel graph processing on the GPU and to support high-throughput executions of concurrent GPU tasks. This thesis presents the design, implementation and experimental evaluations of Medusa, followed by detailed case studies of Medusa in real-world graph applications.

To democratize GPU accelerated graph processing, Medusa proposes a programming framework to enable users to harnessing the power of GPUs by writing sequential C code. Particularly, Medusa offers a small set of APIs for developers to define their application logics, and embraces a runtime system to automatically execute the user-defined functions in parallel on GPUs. We further develop a series of graph-centric optimizations based on the architecture features of GPU for the efficiency of Medusa. With the optimized system on a single GPU, we further extend it to execute on multiple GPUs within a machine and develop optimizations on computation and data transfer.

With the improved programmability of Medusa, users can implement their graph processing algorithms and submit multiple tasks to the GPU for concurrent executions. It therefore requires an optimized runtime system for managing concurrent GPU tasks. As a matter of fact, GPUs are usually deployed in shared environments such as clusters and clouds. In such shared environments, many tasks are submitted to GPUs from different

users, and throughput is an important metric for performance and total ownership cost. Particularly, we propose *Kernelet* as the runtime back end system to optimize the throughput of concurrent kernel executions on the GPU. Kernelet embraces transparent memory and PCI-e data transfer management techniques, as well as dynamic slicing and scheduling techniques to support efficient concurrent kernel executions. With slicing, Kernelet divides a GPU kernel into multiple sub-kernels (namely slices). Each slice has tunable occupancy to allow co-scheduling with other slices for high GPU utilization. We develop a novel Markov chain-based performance model to guide the scheduling decision. Our experimental results demonstrate up to 31% and 23% performance improvement on NVIDIA Tesla C2050 and GTX680 GPUs for our benchmark kernels, respectively. Kernelet improves the throughput of concurrent Medusa applications by 29% on C2050.

As a case study, we collaborate with our colleagues to adopt Medusa in a real research project. Particularly, we adopt Medusa to implement GPU accelerated simulation of information propagation over large-scale and complex social networks. For a number of applications, Medusa is used as the base infrastructure, which allows our colleagues to develop additional application-specific optimizations for performance improvement. From this case study, we demonstrate that the programmability of Medusa improves the coding productivity of domain specific applications, and Medusa brings significant performance speedups by leveraging the GPUs.

Contents

Acknowledgments	i
Abstract	ii
List of Figures	vii
List of Tables	ix
List of Acronyms	x
1 Introduction	1
1.1 Motivation	1
1.2 Our Solution	4
1.3 Results and Contributions	6
1.4 Thesis Outline	8
2 Preliminaries and Related Work	10
2.1 Graph Processing	10
2.1.1 Graph Applications	10
2.1.2 Parallel Graph Processing	11
2.2 GPGPU	13
2.2.1 GPU Architecture	13
2.2.2 CUDA and OpenCL	17
3 System Overview	20
3.1 Programming Interface	20
3.2 Medusa Programming Workflow	22
3.3 Architectural Design	24
4 Medusa: Simplified Graph Processing on the GPUs	27
4.1 Introduction	27
4.2 Related Work	28
4.2.1 GPU Graph Processing	28
4.2.2 Multiple and Distributed GPU Processing	29
4.2.3 BSP Model for Graph Processing	30

4.3	System Design	31
4.3.1	Fine-Grained Graph APIs	32
4.3.2	GPU-Transparent Programming Interface	33
4.3.3	Storage Layout Optimizations	35
4.3.4	Graph-Aware Buffer Scheme	37
4.3.5	Multi-GPU Execution	39
4.4	Evaluation	43
4.4.1	Experimental Setup	43
4.4.2	Graph Primitives	44
4.4.3	Comparison with Manual Implementations	47
4.4.4	Impact of individual optimizations	50
4.4.5	Comparison with CPU-based Execution	51
4.4.6	Efficiency of Visualization Operations	52
4.4.7	Results on Multi-GPU Extension	53
4.5	Summary	56
5	Kernelet: High-Throughput Concurrent Kernel Execution	58
5.1	Introduction	58
5.2	Related Work	60
5.2.1	Scheduling on SMT CPUs	60
5.2.2	Concurrent Kernel Scheduling on GPUs	61
5.3	Problem Definition	63
5.4	System Overview	65
5.4.1	Design Rationales	66
5.4.2	Design Overview	67
5.5	Kernelet Methodology	67
5.5.1	Two-level Scheduling	68
5.5.2	Kernel Slicing	69
5.5.3	Kernel Co-Scheduling	73
5.5.4	Co-scheduling Space Pruning	74
5.5.5	Performance Model	76
5.6	Evaluation	83
5.6.1	Experimental Setup	83
5.6.2	Results on Kernel Slicing	87
5.6.3	Results on Model Prediction	87
5.6.4	Results on Kernel Scheduling	91
5.6.5	Results on Medusa Workload Scheduling	94
5.7	Summary	96

6	Case Study: Medusa-Based Information Propagation Simulations	97
6.1	Background	97
6.2	Techniques and Findings	99
6.2.1	Models of Information Propagation	99
6.2.2	Algorithm Optimizations	100
6.2.3	Simulation on Multiple GPUs	101
6.2.4	Findings in Viral Advertisement	102
6.3	Experience on Using Medusa	102
7	Conclusion and Future Work	105
7.1	Conclusions	105
7.2	Future Work	106
	Publications	108
A	CUDA Implementation of Vector Addition	110
	Bibliography	115

List of Figures

2.1	GPU as a coprocessor in modern machines.	13
2.2	The architecture of a streaming multi-processor (SM) in NVIDIA Fermi GPU [1]	14
2.3	CUDA threads organized in two dimensional thread blocks (right) and one dimensional grids (left).	17
3.1	User-defined functions in PageRank implemented with Medusa.	23
3.2	The key modules in Medusa.	24
4.1	Device data structure definitions and CUDA kernels generated from front end for PageRank.	34
4.2	Graph layouts: AA and CAA.	37
4.3	Graph aware buffer scheme.	38
4.4	Graph partitioning and replication (dashed circles represent the replicas)	40
4.5	Graph partitioning with multi-hop replication.	41
4.6	Performance comparison between Medusa and existing GPU implementation of SSSP [2].	48
4.7	Performance impact of different graph layouts on PageRank using RMAT as input.	50
4.8	Graph aware buffer scheme.	51
4.9	Performance speedup of Medusa running on the GPU over MTGL [3] running on 12 cores.	52
4.10	Performance speedup of Medusa running on the GPU over MTGL [3] running on one core.	53
4.11	Visualization results on DBLP co-authorship graph.	54
4.12	Execution time with increasing hops of replication.	55
4.13	Comparison between the predicted and actual measured TS	55
5.1	Application scenarios of concurrent kernel executions on the GPUs.	63
5.2	Kernel state transitions.	64
5.3	Design overview of Kernelet.	68

5.4	An example of kernel slicing.	70
5.5	Index rectification of <i>MatrixAdd</i> kernel based on PTX code.	72
5.6	Correlation between MUR/PUR and <i>CP</i>	75
5.7	Warp state transition diagram.	78
5.8	Sliced execution overhead with varying slice size on both C2050 and GTX680.	87
5.9	Comparison between predicted and measured single kernel execution IPCs on two GPUs.	88
5.10	Comparison between predicted and measured concurrent kernel execution IPCs on two GPUs with optimal slice ratio.	89
5.11	Comparison between predicted and measured concurrent kernel execution IPCs on two GPUs with fixed one-to-one slice ratio.	89
5.12	Comparison between predicted and measured concurrent kernel execution IPCs with/without considering uncoalesced access on C2050.	90
5.13	Comparison between predicted and measured concurrent kernel execution IPCs without considering multiple warp schedulers on GTX680.	90
5.14	Comparison between predicted and measured <i>CP</i> on C2050.	91
5.15	Comparison between different scheduling methods on both C2050 and GTX680.	92
5.16	CDF (cumulative distribution function) of execution time of MC(1000).	92
5.17	Execution time comparison of different Kernelet variants on C2050 ($sf = 1$).	93
5.18	Impact of kernel co-scheduling for different sf on C2050.	94
5.19	Sequential and concurrent execution of Medusa PageRank and Medusa BFS on Random.2MV.64ME	95
5.20	Sequential and concurrent execution of Medusa PageRank and Medusa BFS on RMAT.2MV.64ME	95
6.1	Execution time per simulation step.	101
A.1	CUDA implementation of vector addition (Part I).	110
A.2	CUDA implementation of vector addition (Part II).	111
A.3	CUDA implementation of vector addition (Part III).	112
A.4	CUDA implementation of vector addition (Part IV).	113
A.5	CUDA implementation of vector addition (Part V).	114

List of Tables

2.1	Comparison of hardware features of CPUs and GPUs	15
3.1	User-defined APIs in the EMV model	21
3.2	System provided APIs and parameters in Medusa	21
4.1	Summary of techniques used in Medusa and their advantages	31
4.2	Notations in the cost model	42
4.3	Characteristics of graphs used in the experiments	44
4.4	Traversed edge per second (10^6 TEPS) comparison with manual implemen- tations [2, 4].	48
4.5	Coding complexity of Medusa implementation and manual implementations.	49
4.6	Traversed edge per second (10^9 TEPS) comparison with Merrill et al.'s pa- per [5].	49
4.7	Performance comparison of varying the number of GPUs with/without over- lapping.	54
5.1	Notations in the performance model	77
5.2	GPU configurations.	83
5.3	Specification of benchmark applications and thread configuration (#threads per thread block $\times$ #thread blocks).	84
5.4	Memory and computational characteristics of benchmark applications. . . .	85
5.5	Workload configurations.	86
5.6	Number of kernels pruned with varying α_p and α_m on C2050.	92

List of Acronyms

AA adjacency array. 36

AOS array of structure. 25

APSP all-pair shortest path. 29

APU Accelerated Processing Unit. 110

BFS breadth first search. 1

BS Black Scholes. 85

BSP bulk synchronous parallel. 4

CAA column-major adjacency array. 37

CDF cumulative distribution function. 93

CUDA Compute Unified Device Architecture. 17

DSL domain-specific language. 32

DVFS dynamic voltage/frequency scaling. 3

ELL edge-oriented layout. 36

EMV Edge-Message-Vertex. 4

FCFS first come, first serve. 69

GBSP BSP-based graph processing model. 28

GPGPU General-Purpose computation on Graphics Processing Units. 1

HPC high performance computing. 11

HY hybrid layout. 36

ICM Independent Cascade Model. 102

LS load/store. 16

LTM Linear Threshold Model. 102

MBM maximal bipartite matching. 44

MM matrix multiplication. 85

MRIQ magnetic resonance imaging-Q. 85

MTGL Multi-Threaded Graph Library. 6

MUR Memory-bandwidth Utilization Ratio. 76

PC pointer chasing. 85

PTX Parallel Thread Execution. 19

PUR Pipeline Utilization Ratio. 76

RGEM responsive GPGPU execution model. 63

SAD sum of absolute differences. 85

SFU special function unit. 16

SIMD Single Instruction Multiple Data. 11

SM streaming multiprocessor. 13

SMT Simultaneous Multi-Threading. 59

SNS social networking services. 11

SOA structure of array. 25

SP scalar processor. 14

SPMD Single Program Multiple Data. 4

SSSP single source shortest path. 29

TEA tiny encryption algorithm. 85

TEPS traversed edges per second. 48

WWW World Wide Web. 1

Chapter 1

Introduction

In this chapter, we first present the motivation of this thesis, and the technical challenges of developing parallel graph processing systems on the GPU. Next, we briefly introduce our solutions, key results and contributions of this work. At the end of this chapter, we give the outline of this thesis.

1.1 Motivation

Graphs are common data structures in various applications such as social networks, chemistry and the World Wide Web (WWW). Graph processing algorithms have been the fundamental tools in various fields. Users usually apply a series of operations on the graph edges and vertices to obtain the final result. The example operations can be breadth first search (BFS), PageRank [6], shortest paths and even their customized variants (for example, users may apply different application logics on top of BFS). The efficiency of graph processing is a must for high performance of the entire system. On the other hand, writing every graph processing algorithm from scratch is inefficient and involves repetitive work, since different algorithms may share the same or similar operation patterns, optimization techniques and common software components. A programming framework supporting high programmability for various graph processing applications and providing high efficiency as well can greatly improve productivity and system usability.

Recent years have witnessed increasing adoptions of General-Purpose computation on Graphics Processing Units (GPGPU) in many applications [7], such as databases [8, 9],

data mining [10, 11], network routing [12] and encoding [13], and so on. The GPU has also been used as an accelerator for various graph processing applications [2, 4, 5, 14–19]. While those GPU-based solutions have demonstrated significant performance improvement over CPU-based implementations, they are limited to specific graph operations. Users usually need to implement and optimize GPU programs from scratch for different graph processing tasks.

Writing a correct and efficient GPU program is challenging in general, and even more difficult for graph applications. First, the GPU is a many-core processor with massive thread parallelism. To fully exploit the GPU parallelism, users need to write parallel programs that scale to hundreds of cores. Moreover, compared with CPU threads, the GPU threads are lightweight, and the tasks in the parallel algorithms should be fine grained. In terms of graph processing applications, users usually have to parallelize the graph processing tasks at very fine granularity to fully utilize the GPU. Besides, users need to carefully manage task scheduling since performance of GPU is more vulnerable to load imbalance than CPU. Other issues like thread synchronization and communication can further complicate the problem. Second, the GPU has a memory hierarchy that is different from the CPU's. To harness the magnificent data access performance of the GPU, users should promote uniform data access patterns and data reuse by utilizing the scratchpad memory when designing GPU algorithms. Since graph applications usually involve irregular accesses to the graph data and has little data locality, it's difficult to enforce regular memory access patterns and take advantage of the scratchpad memory. Users need to carefully design the graph data layouts and memory accesses, which are key factors to the efficiency of GPU acceleration. Finally, since the GPU is designed as a co-processor, users have to explicitly perform memory management on the GPU, and deal with GPU specific programming details such as kernel configuration and invocation. For example, users have to manually tune the kernel configurations based on the sizes of the input graphs. All these factors make the GPU programming a difficult task.

In addition to programming challenges, we have recently observed a trend on integrating GPUs into clusters and cloud computing infrastructures. This is due to their immense

computation power, memory bandwidth and higher power efficiency. In Top500 list of June 2013, two out of the top ten supercomputers are with GPUs integrated. Amazon and Penguin have provided virtual machines with GPUs. It has been shown that with appropriate dynamic voltage/frequency scaling (DVFS), power efficiency of GPUs can even be further improved [20]. In both cluster and cloud environments, GPUs are often shared by many concurrent GPU programs (or *kernels*) (most likely submitted by multiple users). Throughput of concurrent kernel executions is an important optimization metric for efficiency and the total ownership cost of GPUs in such shared environments. First, graph processing and many other GPGPU tasks are usually throughput oriented [21]. A high throughput leads to high performance and productivity. Second, compared with CPUs, GPUs are still expensive devices. A high throughput not only means a high resource utilization but also a low total ownership cost. That might be one of the reasons that GPUs are usually deployed and shared to handle kernels from multiple users.

Previous studies [22–24] have developed various optimization techniques to improve the performance of GPU applications. These techniques mostly focus on single-kernel optimizations (e.g., new data structures [25] and GPU friendly computing patterns [26]). However, a single kernel usually severely under-utilizes the GPU. This severe underutilization is mainly due to the inherent memory and computation characteristics of a single kernel (e.g., random memory accesses and lack of instruction level parallelism). In our experiments, we have studied eight benchmark kernels (details are presented in Chapter 5). On C2050, their average *IPC* is 0.52, which is far from the optimal value (1.0). Their memory bandwidth utilization is only ranging from 0.02% to 14%. Running a wide range of applications with different memory and computation characteristics (such as graph processing tasks) in the shared GPU environment concurrently enables new opportunities for increasing the throughput.

Recent GPU architectures like NVIDIA Fermi architecture supports concurrent kernel executions, which allows multiple kernels to be executed on the GPU simultaneously if resources are allowed. In particular, Fermi adopts a form of *cooperative* kernel scheduling.

Other kernels requesting the GPU must wait until the kernel occupying the GPU voluntarily yields control. Here, we use NVIDIA CUDA’s terminology, simply because CUDA is nowadays widely adopted in GPGPU applications. A kernel consists of multiple executions of *thread blocks* with the same program on different data, where the execution order of thread blocks is not defined. This is known as Single Program Multiple Data (SPMD) execution model. On Fermi GPUs, one kernel can take the entire GPU if it has sufficient thread blocks to occupy all the multi-processors (even though it can have severely low resource utilization). Concurrent execution of two such kernels almost degrades to sequential execution of individual kernels. Recent studies [27] on optimizing concurrent kernels mainly focus on kernels with low occupancy (i.e., the thread blocks of a single kernel cannot fully utilize all GPU multiprocessors). However, the occupancy of kernels is usually high after single-kernel optimizations in practice.

In summary, while GPU has demonstrated superb performance speedup on specific graph processing tasks, designing and developing an efficient and general parallel graph processing system on the GPU is a non-trivial task, with a series of research challenges. That motivates us to develop Medusa and its optimized runtime system to address those challenges.

1.2 Our Solution

To ease the pain of leveraging the GPU in common graph computation tasks, we propose a software framework named Medusa to simplify programming graph processing algorithms on the GPU. Inspired by the bulk synchronous parallel (BSP) model [28], we develop a novel graph programming model called Edge-Message-Vertex (EMV) for fine-grained processing on vertices and edges. EMV is specifically tailored for parallel graph processing on the GPU. Like existing programming frameworks such as MapReduce [29] and its variant on the GPU [30], Medusa provides a set of sequential APIs for users to implement their applications. The APIs are oriented at the EMV programming model for fine-grained parallelism. Medusa embraces an efficient message passing based runtime. It automatically

executes user-defined APIs in parallel on one GPU or multiple GPUs on the same host, and hides the complexity of GPU programming from users. We are currently extending Medusa to support the distributed heterogeneous CPU-GPU computing environment [31]. Thus, users can write the same APIs, which automatically and transparently run on different GPU-enabled platforms.

Memory efficiency is often an important factor for the overall performance of graph applications [2, 14–16]. To improve the memory efficiency of Medusa, we have developed a series of memory optimizations. A novel graph layout is developed to exploit the *coalesced* memory feature of the GPU. A graph aware message passing mechanism is specially designed for message passing in Medusa. We also develop two multi-GPU-specific optimization techniques, including the cost model guided replication for reducing data transfer across the GPUs and overlapping between computation and data transfer.

To improve the throughput of concurrent kernel executions, we developed a throughput-oriented kernel execution runtime system for Medusa. Since individual kernels as a whole cannot realize real sharing of the GPU resources, we study whether we can slice the kernel into small pieces and then co-schedule slices from different kernels for higher GPU resource utilization. One observation is that GPU kernels conform to the SPMD execution model. A kernel execution can usually be divided into multiple *slices*, each consisting of multiple thread blocks. Slices can be viewed as low-occupancy kernels and can be executed simultaneously with slices from other kernels. The GPGPU concurrent kernel scheduling problem is thus converted to the slice scheduling problem.

With slicing, we have two more technical issues to address. The first issue is on the slicing itself: what is the suitable slice size? How to perform the slicing in a transparent manner? The smallest granularity of a slice is one thread block, which can lead to significant runtime overhead of submitting too many such small slices onto the GPU for execution. To the other extreme, the largest granularity of the slice is the entire kernel, which degrades to the non-sliced execution. The second issue is to select the slices for co-scheduling in order to maximize GPU utilization.

To address those issues in slicing-based kernel executions, we develop *Kernelet*, a runtime layer with dynamic slicing and scheduling techniques to improve the GPU utilization, where Kernelet is integrated into Medusa as an optimized runtime system. To preserve the programmability of Medusa, Kernelet transparently performs GPU memory management and data transfer between the main memory and the GPU memory. Given the kernels that are ready to execute on the GPU (i.e., the input data are available on the GPU memory), Kernelet dynamically performs slicing on the kernels, and the slices are carefully designed with tunable occupancy to allow slices from other kernels to utilize the GPU resources in a complementary way. For example, one slice utilizes the computation units and the other one utilizes the memory bandwidth. We develop a novel and effective Markov chain based performance model to guide kernel slicing and scheduling decisions. Compared with existing GPU performance models (e.g., [32, 33]) which are limited to a single kernel only, our model is designed to handle heterogeneous workloads (i.e., slices from different kernels). We further develop a greedy algorithm to always co-schedule slices from the two kernels with the highest estimated performance gain.

1.3 Results and Contributions

We have implemented Medusa in NVIDIA CUDA 5.0, and made it open-sourced at <https://code.google.com/p/medusa-gpu/>. We have evaluated the efficiency and programmability of Medusa on a machine with four NVIDIA Tesla C2050 GPUs and two Intel E5645 CPUs. To demonstrate the programmability of Medusa, we develop a set of common graph processing primitives on sparse graphs and compare Medusa-based implementations with manual implementations. The CPU-based manual implementations are based on the Multi-Threaded Graph Library (MTGL) [3], and we adopt previous GPU implementations [2, 4, 5] as GPU-based manual implementations. The data sets include the publicly available real graphs from the Stanford Large Network Dataset Collections [34] and synthetic ones generated from GTGraph generator [35].

Our experimental results show that: (1) Medusa simplifies programming GPU graph processing algorithms in terms of a significant reduction in the number of source code lines developed by users. Medusa achieves comparable or better performance than the manually tuned GPU graph operations. (2) Our optimization techniques on graph layout and message buffering significantly improve the performance of graph processing operations on the GPU. (3) Medusa executing on four GPUs is up to 1.8 and 2.6 times faster than on a single GPU for BFS and PageRank, respectively.

We have evaluated Kernelet on two latest GPU architectures (NVIDIA Tesla C2050 and GTX680). The GPU kernels under study have different memory and computational characteristics. Experimental results show that: (1) Our analytical model in Kernelet can accurately capture the performance of heterogeneous workloads on the GPU. (2) Our scheduling increases the GPU throughput by up to 31% and 23% on C2050 and GTX680, respectively. (3) Our memory command scheduler overlaps up to 99% of data transfer with kernel execution. (4) Kernelet improves performance of two concurrent Medusa applications by 29%.

In our case study, we use Medusa to implement a number of information propagation simulation algorithms in a real research project. The case study shows that Medusa significantly reduces the programming effort in building GPU graph processing algorithms by reducing the number of lines of code and almost eliminating the GPU programming learning curve. With Medusa, users without GPU programming experience can quickly implement their graph operations on the GPU, which accelerates the discovery and findings of domain specific applications. On a single C2050, Medusa based implementation is 12-16 times faster than sequential CPU implementation.

In summary, we make the following contributions in this thesis.

- We develop Medusa, a programming framework for simplifying programming graph processing applications on the GPU. Medusa allows users to build parallel GPU programs by writing a small set of sequential APIs. With various novel optimization

techniques, Medusa offers high performance and programmability for parallel graph processing on the GPU at the same time.

- We develop Kernelet, a system for transparent management of concurrent kernel executions on the GPUs. Kernelet is integrated as an optimized runtime for Medusa to improve the throughput of Medusa. Kernelet exploits concurrent execution and memory overlapping opportunities to improve the throughput of shared GPUs. We propose a probabilistic model to estimate the performance of concurrent kernels. To the best of our knowledge, it is the first performance model supporting modeling concurrent tasks on the GPU.
- We have performed extensive experiments on Medusa and Kernelet. Experimental results demonstrate the high performance of Medusa and the effectiveness of Kernelet. We also apply Medusa to a real research project to demonstrate the usability of Medusa in the real application.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we introduce the preliminaries and related work to our study. We briefly introduce the modern GPU hardware architectures. Next, we review the related work on parallel graph processing, graph processing and task scheduling on GPUs.

In Chapter 3, we give a system overview of our GPU graph processing system Medusa. We briefly present the functionality of each component in Medusa.

In Chapter 4, we present design, implementation and evaluation of Medusa on single and multiple GPUs. Specifically, we present our optimization techniques for the high performance of the system while maintaining good programmability. We have published the design and results of Medusa in our papers [36–38].

In Chapter 5, we describe our task scheduling techniques for improving GPU utilization in shared environments. Our novel slicing and scheduling techniques are presented. The main techniques and results presented in this chapter has been published in our paper [39].

We present a case study of using Medusa to study GPU based information propagation simulation in Chapter 6. We show how the optimizations and APIs of Medusa facilitated the study in terms of improving simulation performance and reducing the programming effort. Results and findings of the case study are published in [40–42].

Finally, we conclude the thesis and outline our future research directions in Chapter 7.

Chapter 2

Preliminaries and Related Work

In this chapter, we first review the applications of graph processing, as well as recent libraries, frameworks and systems for parallel graph processing. Then, we give a brief introduction of the CUDA platform and the GPU architectures.

2.1 Graph Processing

Graph processing is the core component in many classic and emerging applications. It attracts many research efforts in studying how libraries, frameworks and systems can efficiently leverage parallel architectures to accelerate graph processing algorithms and provide better programmability.

2.1.1 Graph Applications

Graph is an ideal abstraction for many data structures. Basic graph algorithms such as graph traversal, shortest path search and connected components are widely used as the building blocks of many applications. The web search engines model the WWW as a directed graph, representing web pages as vertices and hyper links as directed edges. With this graph representation, link analysis algorithms such as PageRank [6] are applied to determine the importance of web pages. The social network represents the relationship or interactions between human beings. In the real world, graph mining techniques are applied to the social networks for various applications, such as crime detection [43], viral marketing [44], and recommendation systems [45]. In the cyber world, the social networks

have grown to tremendous sizes. For example, as of September 2012, Facebook, one of the largest online social networking services (SNS) in the world, has over one billion active users [46]. Graph mining on the SNS data has led to fruitful research outcomes [44, 47–49]. Graph has also been adopted in the other fields such as bioinformatics and computational chemistry and so on [50].

2.1.2 Parallel Graph Processing

Parallel algorithms have been a classical way to improve the performance of graph processing. We mainly review the parallel graph processing algorithms on CPUs, and leave the related work on GPUs in Chapter 4.

Parallel graph processing has attracted a lot of research efforts in traditional high performance computing (HPC) environments. Dekel et al. [51] used the parallel matrix multiplication algorithms developed for cube connected and perfect shuffle computers to solve graph problems. Chin et al. [52] studied parallel algorithms for a number of graph problems on the Single Instruction Multiple Data (SIMD) model. Quinn et al. [53] conducted a comprehensive survey on algorithms and data structures developed for parallel graph algorithms.

More recently, researchers resort to the multi-core platform for graph processing acceleration. On multi-core CPUs, parallel libraries such as MTGL [3] have been developed for parallel graph algorithms. Similar to Medusa, MTGL offers a set of data structures and APIs for building graph algorithms. The MTGL API is modeled after the Boost Graph Library [54] and is optimized to leverage shared memory multithreaded machines. The SNAP framework [55] provides a set of algorithms and building blocks for graph analysis, especially for small-world graphs. Ligra [56] is an in-memory parallel graph processing framework designed and optimized for multi-core servers with a terabyte of memory. GraphCHI [57] enables processing large-scale graphs on single multi-core node by breaking the graph into small parts and processing one part at a time. Like GraphCHI, X-Stream [58] is also designed for processing external memory graphs. Compared with GraphCHI, X-Stream

eliminates the preprocessing of graph data and supports processing completely unordered edge lists loaded from storage media.

To facilitate developing distributed graph algorithms in the cluster/grid settings, software libraries such as Parallel BGL [59], CGMGraph [60] and Combinatorial BLAS [61] have been developed. Besides the traditional distributed platforms, cloud computing is becoming popular for graph applications [62]. MapReduce [63] is the de facto standard for large-scale data analysis, and MapReduce based graph processing systems such as PEGASUS [62], HADI [64], X-RIME [65] and SAHAD [66] have been introduced. PEGASUS [62] supports graph mining operations with a generalization of the matrix-vector multiplication operation. HADI [64] is designed to compute the radii and the diameter of massive graphs. Surfer [67] is a graph processing engine designed for the cloud, it extends MapReduce and provides the propagation primitive for build graph applications.

As large-scale graphs find increasing applications in data analytics of many sorts, more research efforts are taken to ease the implementation of efficient graph processing applications on the distributed and cloud environments. Pregel [28] is a system for building large-scale graph processing applications based on the BSP model. Under the BSP model, the same computation is performed independently on each vertex and carried out for many iterations with barrier synchronization at the end of each iteration. Vertices exchange data with each other by sending messages, which are only available to the recipient vertices at the beginning of the next iteration. Using Pregel, users just need to write sequential code, which is automatically executed in the distributed environment. More about the Pregel's BSP model can be found in Section 4.2.3. GraphLab [68, 69] also provides a local computation model on individual vertices, but loosens the synchronization requirement and supports asynchronous computation.

While Pregel and GraphLab adopt the random hashing-based graph partitioning mechanism to distribute the graph data on different nodmachines, Chen et al. [70] take a further step by proposing a network performance aware graph partitioning framework to reduces the network traffic for Pregel-like systems. Real world graphs usually have power-law

Figure 2.1: GPU as a coprocessor in modern machines.

degree distributions, which introduce load imbalance for the vertex-centric model. PowerGraph [71] addresses this issue by decomposing the vertex-centric model into three phases and partition the graph using balanced p -way edge-cut. The PowerLyra [72] system combines advantages of both GraphLab and PowerGraph and propose to dynamically applying different computation and partition strategies to different parts of the graph.

2.2 GPGPU

GPUs have rapidly evolved into a powerful accelerator for many applications, especially after CUDA was released by NVIDIA [73]. The top tags in <http://gpgpu.org/> show that a wide range of applications have been accelerated with GPGPU techniques, including data mining [4, 11, 74], scientific computing [75], network coding [13]. Owens et al. [7] provide a survey on general-purpose computation on the GPUs. In this section, we first introduce the GPU architecture. Then, we briefly introduce the CUDA platform [76], which is adopted for the implementation of Medusa. We also give a brief introduction on another GPU programming platform, OpenCL.

2.2.1 GPU Architecture

As shown in Figure 2.1, GPU is designed as a co-processor in modern machines. The GPU is connected with the host machine via the PCI-e bus. Compared with CPU, GPU has

Figure 2.2: The architecture of a streaming multi-processor (SM) in NVIDIA Fermi GPU [1]

Table 2.1: Comparison of hardware features of CPUs and GPUs

<i>Feature</i>	<i>CPU</i>	<i>GPU</i>
Cache	Cache size is usually large and cache performance is important for the efficiency of the CPU programs.	Cache size is small. Cache performance is less important since GPU programs usually have little data reuse.
Cores	Usually less than 16 powerful cores on one chip.	Hundreds to thousands of simple cores on one GPU.
Memory	Memory is optimized for latency and memory bandwidth is usually less than 20 GB/s.	Memory is optimized for bandwidth, which is usually hundreds of GB/s.
IO Capability	Directly interact with disk and other peripherals.	No IO capability, require CPU assistance for data transfer.
Role	Run the operating system and applications.	Run the embedded GPU code of applications.

much higher computational power and bandwidth and has been adopted in many applications for acceleration. Following previous work [7], we model the GPU as a manycore processor. The GPU consists of a number of streaming multiprocessor (SM) and a dedicated graphics memory subsystem. All the SMs share the memory bandwidth through an interconnection network. The GPU architecture is significantly different from the CPUs due to their different design goals. Table 2.1 shows the comparisons of the CPU and GPU architectures. Due to the architecture differences, porting applications from the CPU to the GPU usually involves non-trivial redesign and optimization.

Figure 2.2 shows the architecture of an SM in the NVIDIA Fermi GPU. The SM has 32 scalar processor (SP), which is assembled into two groups. Correspondingly, the SM has two sets of instruction issue and dispatch units. The SM is designed to execute hundreds of lightweight threads concurrently. Each 32 of the concurrent threads are grouped into a scheduling unit called *warp*. Threads in the same warp are executed synchronously in SIMD manner. Besides the scalar cores, there are 16 load/store (LS) units, which are responsible for executing memory load and store instructions. To increase the speed of executing transcendental instructions such as sin, cosine, reciprocal, and square root instructions, Fermi SM includes four special function unit (SFU). Each of the SP, LS, or SFU unit can

execute one instruction per thread, per clock. The two issue and dispatch units can issue instructions from two different warps concurrently. As a result, the peak IPC is 32 for SP instructions, 16 for LS instructions and 4 for SFU instructions.

Warp is the basic scheduling unit of GPU threads. When threads in the same warp execute different code paths, divergence occurs and all paths are executed sequentially. Thread divergence may significantly degrade GPU performance. As a result, it is important for the users to avoid divergence within a warp. GPU relies on large number of concurrent threads to hide the long memory latencies, which otherwise can be a major performance bottleneck. If one warp is stalled on memory instruction, the scheduler switch to another warp with zero overhead. The number of warps which can be concurrently executed depends on the resource requirements of each thread block, such as register usage. The ratio of the number of resident warps to the maximum number of resident warps is defined as the GPU *occupancy*. Larger occupancy can result in higher chances of hiding the memory latency.

The LS units connect to the GPU memory system. Each SM has a L1 cache and a scratchpad memory called shared memory. The L1 cache and shared memory share the same physical circuits and the size allocation is configurable. The shared memory is manually managed by the user. Shared memory latency is much lower than global memory and can be used to cache data for reuse. All the SMs share a L2 cache. The GPU memory is called global memory and all requests to global memory go through the L2 cache. The peak GPU memory bandwidth can be up to more than 100 GB/s. However, GPU relies on coalesced access to achieve high bandwidth utilization. The cache line size is 128 bytes and maps to a 128-byte device memory segment. A memory request issued by a warp can be *coalesced* into one memory transaction if all the memory addresses fall into the same cache line. Otherwise, the number of memory transactions equals the number of cache lines accessed. The latter access pattern is called uncoalesced access and memory bandwidth is wasted for unused data. Ensuring coalesced access is usually one of the most important GPU programming issues to achieve high performance.

Figure 2.3: CUDA threads organized in two dimensional thread blocks (right) and one dimensional grids (left).

2.2.2 CUDA and OpenCL

Compute Unified Device Architecture (CUDA) is a GPU computing platform and programming model developed by NVIDIA in 2006. The CUDA programming model is designed to leverage the enormous computational power of GPUs while avoiding long learning curves for users.

CUDA programs running on the GPU are called *kernels*, which are multithreaded and usually launched with thousands of threads. The threads in the kernel are hierarchically organized. First, threads are organized into thread blocks. The size of the thread block can be one, two or three dimensional. Second, thread blocks are further organized into one, two or three dimensional grids. Figure 2.3 shows an example of the hierarchical thread organization in CUDA. The user needs to explicitly specify the block and grid configuration. CUDA also provides block synchronization functions. In another word, users can insert barriers to synchronize all the threads in the block.

OpenCL is a similar programming model to CUDA and was introduced by the Khronos Working Group [77]. OpenCL is not only for GPUs, but also targeted at a variety of parallel architectures. OpenCL adopts the same thread hierarchy as CUDA while using

different terminology. Using the NVIDIA Toolkit, both OpenCL and CUDA programs are firstly translated into the same intermediate language before being compared into target machine code. Medusa is implemented using CUDA, and it can be easily ported to the OpenCL platform considering the great resemblance between CUDA and OpenCL.

To program in CUDA, users need to find the parallelism in the algorithm and decompose the parallel tasks according to the SPMD (Single Program Multiple Data) programming model of CUDA. In another word, users need to explicitly partition the problem to sub-problems that can be solved independently and concurrently by CUDA thread blocks. Although CUDA provides a standard C language interface, hierarchical parallelism (thread level and block level parallelism) and other co-processing features of GPU still make GPU programming a touch job.

We take the implementation of the vector addition algorithm as an example to illustrate the normal process of writing a CUDA program. The source code of this implementation can be found in Appendix A. A CUDA program usually consists of the following kinds of code.

- Main program. The functionalities of the main program includes reading the input data, building the data structure on main memory and initializing the data. This part is the same as conventional CPU programs.
- Device memory related code. This usually includes code for allocating device memory, and transferring data to and from the GPU memory. Due to the architectural differences between the main memory and GPU memory, users may need to redesign the data structure for GPU to achieve high bandwidth utilization (such as coalesced access feature).
- CUDA kernel device code. This requires the user to first parallelize the vector addition algorithm. The vector addition algorithm can be considered as data parallel since each pair of elements can be added independently and in parallel. Then, the user needs to consider how to assign the workload to CUDA threads. For efficiency, users may

need to consider various optimization techniques such as using shared memory to improve the performance of the kernels. Usually profiling [76] is required to evaluate the effectiveness of the adopted optimization techniques.

- **Kernel launches.** This requires the user to determine the best thread block configuration for the kernels.

While CUDA and other GPU platforms have greatly improved the programmability of GPU, debugging parallel GPU programs is a non-trivial task due to the complexity introduced by the massively large number of concurrent threads. Also, unlike CPU programming, debugging tools for GPU programming are very limited.

There are two more issues on CUDA platform worth discussion.

Block Scheduling. The CUDA runtime system maps thread blocks to SMs in a round-robin manner. A thread block is executed on exactly one of the SMs during its life time. If the number of thread blocks in a kernel is less than the number of SMs on the GPU, each thread block will be executed on a dedicated SM; otherwise, multiple thread blocks will be mapped to the same SM. The number of thread blocks that can be *concurrently* executed on the SM depends on their total resource (including registers and shared memory) requirements. If the kernel currently being executed on the GPU cannot fully utilize the GPU, the GPU allows to schedule thread blocks from other kernels for execution. Existing methods [27] have taken advantage of this new feature to improve the GPU utilization.

GPU Code Compilation. In the process of CUDA compilation, the compiler first compiles the CUDA C code to Parallel Thread Execution (PTX) code. PTX is a virtual machine assembly language and offers a stable programming model for evolving hardware architectures. PTX code is further compiled to native GPU instructions (SASS). The GPU can only execute SASS code. CUDA executables and libraries usually provide PTX code or SASS code or both. In shared environments, the CUDA C code is usually not available to the system back end. Thus, Kernelet should be able to work on both PTX and SASS codes.

Chapter 3

System Overview

In this chapter, we briefly describe the design and implementation of the entire system. Our system has three main design goals:

- (1) To ease GPU programming and allow users without or with little GPGPU experience to build parallel GPU graph processing applications.
- (2) To efficiently execute the user-defined applications on the underlying hardware architectures, including a single GPU and multiple GPUs.
- (3) To improve the utilization of GPUs for concurrent kernel executions in shared environments.

In the following, we first introduce the programming interface and programming workflow on how users can use Medusa, and then present the architectural design of Medusa.

3.1 Programming Interface

In this section, we describe the programming interface of the system from the users' perspective on how they use Medusa.

We present our techniques for directed graphs, and the techniques are applicable to undirected graphs. In a directed graph, we define an edge $s \rightarrow t$, where s is the head vertex and t is the tail vertex. We say the edge is *associated* with s . Each vertex in the graph has a unique ID ranging in $[0, V - 1]$, where V is the number of vertices in the graph.

Table 3.1: User-defined APIs in the EMV model

<i>API Type</i>	<i>Parameters</i>	<i>Variant</i>	<i>Description</i>
<i>ELIST</i>	Vertex v , Edge-list el	Collective	Apply to edge-list el of each vertex v
<i>EDGE</i>	Edge e	Individual	Apply to each edge e
<i>MLIST</i>	Vertex v , Message-list ml	Collective	Apply to message-list ml of each vertex v
<i>MESSAGE</i>	Message m	Individual	Apply to each message m
<i>VERTEX</i>	Vertex v	Individual	Apply to each vertex v
<i>Combiner</i>	Associative operation o	Collective	Apply an associative operation to all edge-lists or message-lists

Table 3.2: System provided APIs and parameters in Medusa

<i>API/Parameter</i>	<i>Description</i>
<i>AddEdge</i> (void* e), <i>AddVertex</i> (void* v)	Add an edge or a vertex into the graph
<i>InitMessageBuffer</i> (void* m)	Initiate the message buffer
<i>maxIteration</i>	The maximum iterations that Medusa executes ($2^{31} - 1$ by default)
<i>halt</i>	A flag indicating whether Medusa stops the iteration
<i>Medusa :: Run</i> (Func f)	Execute f iteratively according to the iteration control
<i>EMV<type> :: Run</i> (Func f')	Execute EMV API f' with $type$ on the GPU

For the set of edges associated with the same vertex, we assign a unique local ID for each edge ranging in $[0, d - 1]$, where d is the out-degree of the vertex. d_{max} is defined as the maximum value of the out-degrees in the graph.

Previous studies [28, 68, 69, 78, 79] have shown that the Bulk Synchronous Processing (BSP) model greatly simplifies the composition of graph algorithms by offering a sequential programming interface oriented on individual vertices. This model is derived from the observation of two common access patterns in various graph applications. First, the processing of vertices and edges is often localized within neighboring vertices. Second, many graph applications have multiple iterations where many edges and vertices are accessed and updated within an iteration. Most BSP-based systems provide a single vertex-based API.

The EMV model of Medusa enhances the current single vertex-based API design to support efficient and fine-grained graph processing on the GPU. In particular, Medusa offers the following two mechanisms for programmability and efficiency.

First, Medusa provides six device code APIs for users to write GPU graph processing algorithms, as shown in Table 3.1. Each API is either for processing vertices (*VERTEX*), edges (*ELIST*, *EDGE*) or messages (*MESSAGE*, *MLIST*). Using these APIs, users can define their computation on vertices, edges and messages. The vertex and edge APIs can also send messages to neighboring vertices. The idea of providing six APIs is mainly for efficiency (The details are presented in Section 4.3.1).

Second, Medusa hides the GPU-specific programming details with a small set of system provided APIs (Table 3.2). Particularly, Medusa provides *EMV < type >:: Run()* to invoke the device code API, which automatically sets up the thread block configurations and calls the corresponding user-defined EMV API function. Medusa allows users to define an *iteration* by running multiple *EMV < type >:: Run()* calls sequentially in one host function (invoked by *Medusa :: Run()*). The iteration is performed iteratively until predefined conditions are satisfied. Medusa offers a set of configuration parameters and utility functions for iteration control.

3.2 Medusa Programming Workflow

There are three steps to implement a graph algorithm based on Medusa. First, the user defines the basic data structures such as edge, message and vertex in C/C++ *structs*. Second, the user implements EMV APIs according to his/her application logic. Third, the user composes the main program, including initializing the graph structure, configuring the framework parameters and invoking the customized EMV APIs with the system provided APIs (in Table 3.2).

Many graph computation tasks require multiple iterations until convergence. To support iterations, Medusa provides two interfaces for controlling the number of iterations of the execution. Users can use both of them for a more flexible iteration control. First, the user can specify the maximum number of iterations, *maxIteration*. Medusa terminates when the number of iterations reaches the predefined limit. Second, Medusa has defined a global variable *halt*, which can be modified by the EMV APIs. By initializing *halt* as false, the

<p>Device code APIs:</p> <pre> /* ELIST API */ struct SendRank{ __device__ void operator() (EdgeList el, Vertex v){ int edge_count = v.edge_count; float msg = v.rank/edge_count; for(int i = 0; i < edge_count; i ++){ el[i].sendMsg(msg); } } /* VERTEX API */ struct UpdateRank{ __device__ void operator() (Vertex v, int super_step){ float msg_sum = v.combined_msg(); vertex.rank = 0.15 + msg_sum*0.85; } </pre> <p>Data structure definitions:</p> <pre> struct vertex{ float pg_value; int vertex_id; } struct edge{ int head_vertex_id, tail_vertex_id; } struct message{ float pg_value; } </pre>	<p>Iteration definition:</p> <pre> void PageRank() { /* Initiate message buffer to 0 */ InitMessageBuffer(0); /* Invoke the ELIST API */ EMV<ELIST>::Run(SendRank); /* Invoke the message combiner */ Combiner(); /* Invoke the VERTEX API */ EMV<VERTEX>::Run(UpdateRank); } </pre> <p>Configurations and API execution:</p> <pre> int main(int argc, char **argv) { Graph my_graph; /* Load the input graph. */ conf.combinerOpType = MEDUSA_SUM; conf.combinerDataType = MEDUSA_FLOAT; conf.gpuCount = 1; conf.maxIteration = 30; /* Setup device data structure.*/ Init_Device_DS(my_graph); Medusa::Run(PageRank); /* Retrieve results to my_graph. */ Dump_Result(my_graph); return 0; } </pre>
---	---

Figure 3.1: User-defined functions in PageRank implemented with Medusa.

framework continues the iterations until any of the API instance sets *halt* to be true. This is equivalent to all API instances needing to vote false to continue the iteration. This iteration control mechanism is similar to the one used in Pregel [28].

To demonstrate the usage of Medusa, we show an example of the PageRank implementation with Medusa, as shown in Figure 3.1. EMV APIs and data structures (e.g., *vertex*) are defined on the left side of the figure. The function *PageRank()* is composed of three user-defined EMV API function calls: an *ELIST* type API (*SendRank*), a message *Combiner* and a *VERTEX* type API (*UpdateRank*). In the main function, we configure the execution parameters such as the *Combiner* data type and operation type, the number of GPUs to use and the maximum number of iterations. *Init_Device_DS* automatically builds the graph data structures and copies them to the GPU. *Medusa::Run(PageRank)* invokes the *PageRank* function. The complete Medusa implementation of PageRank can be found on our project site <https://code.google.com/p/medusa-gpu/>.

Figure 3.2: The key modules in Medusa.

3.3 Architectural Design

Figure 3.2 shows the system architecture of Medusa. It consists of the following key modules.

Graph storage. In the storage module, Medusa allows developers to initialize the graph structure through adding vertices and edges with two system provided APIs namely *AddEdge* and *AddVertex*. The storage component of Medusa stores the graph with optimized graph layout and transfers the graph to GPU automatically. The optimized graph layout exploits the coalesced memory access feature of the GPU, whereas the classic adjacency list cannot.

Medusa code generation tool chain. This module generates the CUDA code for graph computation based on the user-defined EMV APIs. Particularly, users define the data structures (e.g., vertex, edge and messages) and implement the EMV APIs and control program according to their graph computation logic. The example code of PageRank has been given in Figure 3.1. The code generation tool chain performs the following two steps on the above-mentioned user code. First, a source-to-source transformation tool generates code for memory allocations and transforms definitions from array of structure (AOS)¹ to structure of array (SOA)². This transformation allows Medusa to leverage the coalesced

¹An AOS is an array of C/C++ structures.

²A SOA is a C/C++ structure with a set of equal-length arrays as its attributes.

memory access feature of the GPU. Programming with SOA diverts developers from the natural way of thinking about data [80]. To simplify the programming interface, Medusa allows customized data structures such as vertex and edge to be defined using C/C++ *struct*.

Second, Medusa inserts segmented-scan code [81] for the *EDGE* and *MESSAGE* APIs which are declared to be executed by *Combiner*. Many collective APIs (including *ELIST* and *MLIST*) computations are associative operations, for example, PageRank sums the values of received messages of each vertex to update rank values. This enables us to use the *Combiner* interface. Being implemented as a segmented scan operation, the *Combiner* API eliminates the load imbalance problem that each instance of the collective API processes different numbers of edges or messages.

Finally, the system generates the CUDA code for the entire graph computation by adding the code of Medusa runtime. Given the code, we allow users to investigate the efficiency of the Medusa-based program, and users can further apply specific optimizations if they want to improve the performance. Then, the program is compiled and linked with the Medusa libraries.

Medusa runtime. The runtime module is responsible for kernel execution and scheduling and memory management on the GPU. It supports multiple concurrent programs from users. The runtime system maintains a number of data structures to monitor the current state of the GPU. The data structures include (1) a command queue which consists of different kinds of commands. A command can be executing a kernel, memory allocation/deallocation, and memory transfer between the main memory and the GPU memory. (2) a queue of active memory objects (e.g., arrays and global variables) in the GPU memory. The memory management on the GPU and data transfer between the GPU memory and the main memory is managed by Medusa, which is transparent to developers.

First, the runtime prepares the kernel execution and submits the kernel for execution. Particularly, the runtime system first enqueues memory deallocation commands if there is no sufficient GPU memory available, and then enqueues memory allocation and kernel execution commands. The memory deallocation is according to the memory queue maintained

by the runtime. The victims for memory deallocation are chosen according to whether they will be referenced by the kernels in the command queue and their access patterns. Thus, the memory queue is implemented as an LRU queue with looking ahead.

Second, the scheduler schedules the commands in the command queue for execution. On the machine with N GPUs, Medusa automatically runs the EMV APIs on individual graph partitions stored in the N GPUs. There are two considerations on the hardware features of the latest GPU. First, latest GPUs can overlap kernel execution with PCI-e data transfer. We take advantage of this capability to reduce the overhead of PCI-e data transfer. Second, latest GPUs support concurrent kernel executions. We can schedule more kernels smartly for high utilization on the GPU resources. We present the details on the runtime system Kernelet in Chapter 5.

Profiler. GPU vendors have offered hardware counters on the GPU to understand the detailed performance of a GPGPU program. We develop a GUI to hide the details from vendor-specific profiler tools. As a start, we integrate NVIDIA command line profiler into our GUI and users can investigate the performance metrics on memory and execution. The main performance metrics include accesses to the global memory and the shared memory and branch divergence. These performance metrics can be shown on per kernel or per program basis. The default setting is per program. With the profiler, we are able to determine the hot region of a graph computation, and compare the profiling results for different implementations if available.

Visualization. This module visualizes a graph in a manner such that the graph structure is well laid out. Our previous study has implemented GViewer [82] to assist graph visualization and mining with GPUs. We re-implement GViewer based on Medusa, using the CUDA-OpenGL interoperability support for collaboration between computation and visualization.

Chapter 4

Medusa: Simplified Graph Processing on the GPUs

In this chapter, we first give an introduction to Medusa. Then, we present the related work on GPU graph processing. After that, we present the detailed system design of Medusa, including various optimization techniques we developed for the high performance of Medusa. At last, we present the evaluation results of Medusa.

4.1 Introduction

We propose Medusa to simplify programming GPU graph processing tasks. Medusa is provided as a software framework like MapReduce [29] and its GPU variant [30]. Inspired by the recent success of the BSP-based graph processing model (GBSP), we propose a novel graph programming model called “Edge-Message-Vertex” (EMV) for fine grained processing of vertices and edges on the GPU. EMV is specially optimized based on the hardware features of the GPU. With Medusa, user just needs to implement a small set of sequential EMV APIs instead of writing parallel GPU programs. The EMV APIs are oriented at the EMV programming model and offer fine grained parallelism for efficient GPU execution. Like the GBSP model, EMV also supports message passing among vertices. By modeling graph operations as local computations on individual graph elements combined with message passing among vertices, EMV enables users programming the GPU with sequential code. Medusa automatically executes the user-defined APIs in parallel on one or multiple GPUs. The underlying GPU programming details such as thread block configuration,

kernel launching, device memory management, multi-GPU management and so on are all hidden from the users.

Medusa not only eases GPU programming, but also ensures the high performance of GPU programs. Memory efficiency is important to the performance of graph applications due to their irregular data access patterns and poor data locality. We have developed a series of memory optimizations to improve the memory efficiency of Medusa programs. This includes redesigning the graph layout on the GPU, a graph aware message passing mechanism and optimizations for reducing communication overhead of multi-GPU execution.

4.2 Related Work

We first present the existing studies on using GPUs for graph processing. Next, we review the related work on GPU processing on multiple and distributed GPUs. At last, we review the recent developments on BSP-based parallel graph processing.

4.2.1 GPU Graph Processing

With massive parallelism, GPUs have been adopted to accelerate various applications [7, 83, 84]. Vineet et al. [15] used CUDA to accelerate the graph-cut algorithm. They parallelized the push-relabel algorithm for mincut/maxflow. Harish et al. [2] investigated the design and implementation of several commonly used graph algorithms on GPUs including BFS, single source shortest path (SSSP) and all-pair shortest path (APSP). Besides notable speedup achieved, they also reported worse performance when processing scale-free graphs than random graphs. This is because the higher irregularity in scale-free graphs and Harish’s algorithm lacks mechanism to handle load imbalance. Hong et al. [4] proposed a virtual warp-centric method and further developed optimization techniques such as deferring outliers to address the irregularity of graph data structure. Compared with Harish et al.’s work, the warp-centric method achieved notable speedup when the input graph is highly irregular. Hong et al. [18] further improved their approach by integrating CPUs and GPUs and proposed a hybrid approach. Their hybrid approach dynamically chooses the

best implementation from a set of CPU and GPU BFS implementations. Luo et al. [85] implemented BFS on GPUs with a hierarchical queue to store the frontier vertices in order to reduce the excessive accesses. Yang et al. [86] formulated a number of graph mining algorithms as sparse matrix vector multiplication problem (SpMV) and applied tiling methods to improve the SpMV operation on GPUs. Merrill et al. [5] presented an asymptotically optimal linear complexity BFS implementation on the Fermi GPUs. Merrill et al.’s work reveals that previous serial and warp-centric BFS expansion techniques significantly under-utilize the GPU for many graphs by isolating and analyzing the expansion and contraction aspects of BFS.

Most existing GPU graph processing studies focus on specific algorithms. The high performance improvement is achieved at the cost of programming complexity. Advanced optimization techniques require users having deep understanding on the GPU architecture. Writing a new graph program usually needs to revisit or reinvent those intensive optimizations again.

4.2.2 Multiple and Distributed GPU Processing

There are a few research studies on utilizing multiple GPUs within the same machine. Spafford et al. [87] studied the non-uniform memory access and contention effects in multi-GPU systems. Huynh et al. [88] proposed a framework to map streaming applications to multiple GPUs. Schaa and Kaeli [89] provided a method to allow users to predict execution time of GPU applications under different multi-GPU configurations. Kim et al. [90] proposed a multi-GPU abstraction which offers a single virtual GPU image. However, their method shows poor performance for algorithms with irregular memory accesses, such as graph processing applications.

As for GPU processing in a cluster setting, Fan et al. [91] used a cluster of commodity GPUs for airborne contaminants simulation. In Fan et al.’s later work [92], the GPU cluster is used for accelerating large-scale general-purpose computation and visualization applications. To simplify the programming of GPU clusters, they introduced the Zippy

framework which abstracts the GPU cluster programming with a two-level parallelism hierarchy and a non-uniform memory access model. Stuart and Owens [93] explored the challenges in implementing a message passing interface for GPU clusters and proposed the “DCGN” API as a GPU variant of MPI. Abdelkhalek et al. [94] used a GPU cluster to accelerate seismic modeling. Chen et al. [95] attempted to speedup FFT on GPU clusters. Since FFT is bandwidth-intensive and has little data locality, PCI and network data transfer are identified to be the major bottlenecks for GPU applications on a cluster.

4.2.3 BSP Model for Graph Processing

Previous studies [68, 78, 79] have observed that many common graph algorithms can be formulated using the GBSP model. In GBSP, local computations are performed on individual vertices. Vertices are able to exchange data with each other. The same computation and communication procedures are executed iteratively with barrier synchronization at the end of each iteration. This common algorithmic pattern is also adopted by distributed graph processing frameworks such as Pregel [28] and distributed GraphLab [69]. For example, Pregel applies a user-defined function *Compute()* on each vertex in parallel in each iteration of the GBSP execution. The communications between vertices are performed with message passing interfaces. Medusa shares the same design goal as Pregel in providing a programming framework to ease development of graph algorithms, and in hiding the complexity of the underlying runtime from users.

Medusa differs from Pregel in the following aspects. First, the design, implementation and optimization of Medusa are specific to the hardware features of GPUs. For example, our multi-GPU Medusa adopts graph partitioning to reduce data transfer on the host-device communication link (i.e., PCI-e bus), while Pregel uses random hashing by default. Second, Medusa provides more fine-grained programming interfaces than Pregel, exposing fine-grained data parallelism on edges, vertices and messages. Finally, Medusa does not have the sophisticated design for distributed systems, such as failure handling.

Table 4.1: Summary of techniques used in Medusa and their advantages

<i>Problem</i>	<i>Solution</i>	<i>Advantage</i>
Massive parallelism	EMV API	Fine grained parallelism for massive parallelism
Work efficiency	Queue-based implementation with our <i>SetActive</i> API	Allow developing more work-efficient algorithm
GPU specific programming details	Automatic GPU specific code generation	Eliminate the GPGPU learning curve
Graph layout	Novel graph representation	Better memory bandwidth utilization
Message passing efficiency	Graph-aware buffer scheme	Better memory bandwidth utilization and avoid message grouping overheads
Multi-GPU execution	Replication, memory transfer/computation overlapping	Alleviate PCI-e overheads

More recently, GraphLab2 [57, 71] further decompose the vertex-program abstraction into small pieces, which also offer fine-grained parallelism like our EMV model. Green-Marl [96] is another recent effort on easing the difficulty of optimizing GPU graph analysis algorithms, which uses domain-specific language (DSL) to provide users a high level language interface. In comparison with Medusa, Green-Marl processes all vertices with a foreach loop in the order of BFS or DFS, and does not use the message passing mechanism of the GBSP model.

4.3 System Design

This section details the design and implementation of Medusa. The Medusa runtime involves advanced and complicated mechanisms and implementations in order to improve the efficiency with the constraint of preserving high programmability. Most runtime optimizations are entirely transparent to users. Some implementations may be seemingly trivial for specific applications, but become challenging to integrate into a framework to support general graph processing operations. In particular, Medusa focuses on processing sparse graphs.

Table 4.1 presents a summary of the list of optimizations in Medusa and their respective advantages. The proposed optimizations enable Medusa to better exploit massive parallelism and memory features of the GPU while preserving the simple programming interface at the same time. For multi-GPU execution, the graph is partitioned using METIS [97].

4.3.1 Fine-Grained Graph APIs

Most GBSP model based systems provide a single vertex centered API. Users use the single vertex API to access all associated edges and messages (one typical access pattern is iterating edges/messages one by one). While the single vertex-based API design of the GBSP model has achieved good performance and programmability on distributed systems like Pregel [28], such coarse-grained designs are inefficient on GPUs due to execution divergence and irregular memory access. The vertex-based API exhibits severe divergence which makes it unsuitable for GPU execution. First, different vertices may have different numbers of edges, leading to different workloads on each API instance. Second, different number of received messages is another source of divergence. As for memory efficiency, the vertex-centric API makes the memory optimizations on edges and messages a challenging task.

To address those issues, we propose the EMV model as an extension of GBSP. It decouples the single vertex API into separate APIs which target individual vertices, edges or messages. Each GPU thread executes one instance of the user-defined API. The thread configuration such as the number of threads is tuned to maximize GPU utilization. The fine-grained data parallelism exposed by the EMV model can better exploit the massive parallelism of the GPU.

In addition, Medusa supports two variants of APIs for individual and collective operations of edges and messages associated with the same vertex. The collective APIs allow users to access the elements in each *edge-list* (the set of edges associated with the same head vertex) or a *message-list* (the set of messages sent to the same vertex) sequentially. On the other hand, the individual APIs support operations on individual edges, vertices or

messages and expose more parallelism. Medusa also provides a *Combiner* interface, with which users can apply an associative operator to all the elements of each edge-list and message-list. All these APIs require no parallel programming, and users write conventional sequential code to implement those APIs.

Operations which involve dependent computation (e.g., the computation on one edge depends on other edges in the same edge-list) can only be implemented by collective APIs. However, we have observed that many graph algorithms do not need dependent computation on the edge-lists or message-lists. Choosing individual graph elements yields better workload balance and more parallelism. Moreover, many dependent computations are associative operations, for example, PageRank sums the values of received messages of each vertex to update rank values. This enables us to use the *Combiner* interface. The *Combiner* interface is implemented as segmented scan, which has the load-balanced implementation on GPUs [81].

By default, Medusa applies the user-defined APIs on the vertices/edges of the entire graph. This may result in work-suboptimal algorithms for some applications such as BFS and SSSP. In order to allow users to implement work-efficient algorithms, we have added an additional device code API called *SetActive*(vertexID/edgeID), and users are able to indicate whether a vertex or an edge is active in the next EMV API call. The active edges/vertices are maintained in a dynamic queue. We implement the queue structure following the previous study [5], where we do not have specific order of enqueueing vertices or edges. In subsequent API invocations, users are able to apply the EMV APIs to the active vertices and edges only and thus implement more work efficient algorithms. With the *SetActive* API, we have implemented work-efficient BFS and SSSP algorithms and experimentally evaluated their performance (described in Section 4.4).

4.3.2 GPU-Transparent Programming Interface

To leverage the coalesced memory access feature, Medusa uses SOA data arrangement instead of array of structure AOS extensively. Programming with SOA diverts users from the

Figure 4.1: Device data structure definitions and CUDA kernels generated from front end for PageRank.

natural way of thinking about data [80]. To simplify the programming interface, Medusa allows customized data structures such as vertex and edge to be defined using C/C++ *struct*. Users define data structures in AOS, and Medusa automatically transforms AOS into SOA. We have developed a home-grown source-to-source transformation tool to generate SOA code for memory allocations and transforming definitions from AOS to SOA. Those transformations are straightforward but tedious. Thus, it is desirable to make this transformation automatic for ease of programming.

We use the same PageRank example to illustrate the code generation process. The top of Figure 4.1 gives the definition for vertex. A naive way of defining vertex array is to use AOS, which suffers a lot of uncoalesced accesses. The bottom of Figure 4.1 shows the generated device data structure definitions from the user-defined *struct*. Medusa copies the attributes from user code and generates pointer for the SOA definition. Also, Medusa generates a series of *get* and *set* functions for accessing the generated SOA.

Besides AOS transformation, the Medusa front end automatically generates GPU related codes including device memory management and kernel invocation. Providing sim-

plified sequential interfaces and automatic code generation reduces learning curves from users and improves the programming productivity. Although the GPGPU programming environment has been dramatically improved over the years, it still requires a steep learning curves on programming and optimizations. Medusa is designed to ease this pain and increase productivity. One example is that our colleague has successfully developed a system for simulating information propagation with Medusa [98], given that he has no GPU programming experience before. The details are given in Chapter 6.

4.3.3 Storage Layout Optimizations

We start with two basic storage layouts: vertex- and edge-oriented storage layouts. The vertex-oriented storage layout is the classic adjacency array (AA). Each vertex is represented as $\langle id, d, edge-list \rangle$, where id is the vertex ID, and d is the out-degree of the vertex. Each edge is represented as the ID of its tail vertex, and other associated data. As shown in Figure 4.2(a), AA consists of two arrays, *Edge* for storing all the edge-lists, and *Start* for the starting position of each edge-list in *Edge*. The problem with this layout is that concurrent accesses to the adjacency array are scattered in different memory segments, which induces excessive uncoalesced memory accesses.

The edge-oriented layout (ELL) attempts to arrange the edge storage to exploit the coalesced memory access feature. The edge-oriented storage layout stores all the edges with the same local ID consecutively into an array. Since we require fast accesses to all the edges for a vertex in the collective *ELIST*, we store the edges according to the ascending order of their head vertex IDs. The edge-oriented layout virtually forms a 2-D matrix, $e_{k,i}$ ($0 \leq k < d_{max}$, $0 \leq i < V$). $e_{k,i}$ stores the edge with local ID k for the head vertex with ID i . With the edge-oriented layout, memory accesses to the edges with the same local ID are coalesced accesses. However, if the out-degrees of vertices have a large variance (e.g., in power-law graphs), lots of space in the edge-oriented storage layout is wasted and memory transactions are not fully utilized.

Capturing the advantage of both basic layouts results in a hybrid layout (HY): we store the edges with local ID from 0 to t in ELL, and the remainder of the graph using the AA. t

is a tuning parameter according to the out-degree distribution of the graph. For power-law graphs, we set the suitable t such that most vertices with a small out-degree are accessed according to ELL; the vertices with large out-degrees are accessed according to AA.

The problem of the hybrid layout is that it requires tuning the threshold value on AA and ELL and it still has the uncoalesced memory access to the part stored in AA. We further develop a column-major adjacency array (CAA) to maximize the coalesced memory accesses. The basic idea is to store the edges in the adjacency array such that the memory accesses to those edges are coalesced. The CAA storage has two arrays, *Edge* for storing edges, and *Offset* for storing the offset information. In the *Edge* array, we store the edges with the same local ID consecutively in the ascending order of the head vertex IDs. The *Offset* array has the same number of entries as the *Edge* array. Suppose the local ID of edge *Edge*[i] is l ($0 \leq i < E$, E is the number of edges in the graph), *Offset*[i] stores the relative offset of the edge with local ID $(l + 1)$. If there is no edge with a local ID $(l + 1)$ with the head vertex, we set *Offset*[i] to be -1 . Otherwise, the edge is located with $(i + \text{Offset}[i])$ in the *Edge* array.

Figure 4.2 illustrates an example of CAA layout in comparison with basic adjacency array storage. The edges in CAA are stored in the order of their head vertices A, B, C and D. For example, the edges of vertex A are stored in the first and the fifth entry of the *Edge* array, in Figure 4.2(b). We also illustrate the memory access pattern for executing the collective *ELIST* when the number of threads is four. The memory accesses for AA are not consecutive among the threads, and those for CAA are consecutive and exploit the coalesced memory access feature of the GPU.

To reduce load imbalance among threads within one thread block, the vertices can be reordered such that vertices processed by the same thread block have more balanced numbers of edges. By launching a sufficiently large number of thread blocks, SMs tend to have more balanced loads.

We note that AA, ELL and HY have their counterparts in sparse matrix representations (CSR, ELLPACK and hybrid formats, respectively, in previous studies e.g., [99]). CAA is

Figure 4.2: Graph layouts: AA and CAA.

similar to JDS format of a sparse matrix developed by Saad [100]. The major difference is that CAA maintains the offset information to facilitate the accesses to all edges associated to the same vertex.

4.3.4 Graph-Aware Buffer Scheme

Messages are temporarily stored in buffers, allocated by calling the system provided API *InitMessageBuffer*. We first discuss two basic buffer schemes, array- and list-based buffer schemes with respect to the memory efficiency of sending and receiving messages. Note that we assume each message is of fixed size. It's our future work to extend our technique to support variable size messages.

The array-based buffer scheme is to allocate an array for message storage. Implementing the buffer with a fixed-sized array, this buffer scheme requires the information of the buffer size as well as the output positions for each message to avoid conflicts. Even worse, if the messages to the same vertex are not stored consecutively, Medusa needs a grouping operation in order to support message processing in collective user-defined APIs. In contrast, the list-based buffer scheme relies on dynamic memory allocation. We adopt a hash table with dynamic memory allocation [101] to store messages. This method eliminates the pre-computation of message sizes and the grouping operation in the array-based storage scheme. However, the dynamic hash table requires atomic operations and the accesses to the hash table are minimally coalesced.

Figure 4.3: Graph aware buffer scheme.

Neither of the two buffer schemes can achieve good performance on both storing and processing the messages. That motivates us to develop a buffer scheme to capture the best of both worlds. We observe that the messages are usually sent/received along the edge in the EMV model. Given the maximum number of messages that can be sent along each edge, we can compute (1) the maximum total number of messages; (2) the maximum number of messages that each vertex can receive. The awareness of the graph structure helps us to allocate the buffer, and to obtain the write positions of the messages along each edge.

To avoid the grouping operation, we ensure that the write positions of the messages sent to the same vertex are consecutive. This is achieved with the idea of “reversed edge indexed message passing.” While loading the graph, Medusa constructs a reverse graph by swapping the head and tail of each edge. The reverse graph is stored in AA format. We assign an rID (*reverse ID*) for each edge in the original graph, whereby the rID value of each edge equals the index of its reverse edge in the adjacency array. Figure 4.3(b) shows the rID value for each edge in an example graph.

The rID definition has an important property: *the rID values for the edges with the same tail vertex are consecutive integers*. For example, the $rIDs$ of the edges with the same tail vertex D in the original graph in Figure 4.3 are 4 and 5. We take advantage of this property to ensure that the write positions of the messages sent to the same vertex are consecutive.

The graph aware buffer scheme works as follows. First, a message buffer with $(E \times m)$ entries is allocated, where m is the maximum number of messages that can be sent via each edge. For example, m is equal to one in PageRank. Medusa allows users to set the m value. Second, when a message is sent along an edge and the rID of that edge is k , the start position for the message generation is $(k \times m)$ in the message buffer. Figure 4.3(c) shows an example of the graph aware buffer scheme for PageRank ($m = 1$).

When sending messages, the rID values give the write locations for the message along each edge. When receiving messages, the messages for the same vertex are already stored together. Thus, all the messages are already grouped by the tail vertex. This is because of the property of the rID values. Thus, no additional grouping operation is needed. Moreover, the message buffer uses an array, and thus the memory efficiency of message processing is much higher than that of the list-based buffer scheme, as demonstrated in our experiments.

4.3.5 Multi-GPU Execution

We first present a basic implementation of the multi-GPU extension, and then our multi-hop replication optimization to reduce the data transfer cost in the PCI-e bus. Our multi-hop replication scheme is inspired by stencil operation optimizations [102, 103]. Differently, we target at partitioned graphs in multi-GPU environments.

Replication. To accommodate multi-GPU graph processing, we divide the graph into equal-sized partitions and store each partition on one GPU. We adopt the widely used graph partitioning tool METIS [97] to partition the input graph. METIS provides high quality k-way partitioning of a graph such that the number of cross-partition edges is minimized and each partition has roughly equal number of vertices. Clearly, the quality of graph partitioning has great effect on the amount of data transfer among different GPUs. It is our future work to investigate other graph partitioning algorithms.

Figure 4.4(a) shows an example with three GPUs. A directed graph is partitioned into three parts and each part is stored on one GPU. In the design of Medusa, messages are

Figure 4.4: Graph partitioning and replication (dashed circles represent the replicas)

passed along edges. Graph partitioning introduces cross-partition edges, whose head and tail vertices are in different partitions and hence stored on different GPUs.

In order to apply EMV APIs on each graph partition, we maintain replicas of the head vertices of all cross-partition edges in the partitions where the tail vertices reside (we call it the *tail partition*). Each cross partition edge is replicated in its tail partition, as shown in Figure 4.4(b). Thus, messages are emitted directly from the replicas and every edge can access its head and tail vertices directly. The execution of EMV APIs is performed on each partition independently. After the execution, we update the replicas on each graph partition. The update requires the costly PCI-e data transfer, which can become a bottleneck for some application such as BFS. We therefore propose a multi-hop replication scheme as well as overlapping on the computation and data transfer to alleviate the overhead of PCI-e data transfer.

Multi-hop Replication Scheme. When the inter-GPU communication time is dominant in the total execution time, reducing the time cost of communication can significantly improve the application performance. The multi-hop replication scheme alleviates the overhead of inter-GPU communication by reducing the number of times of replica update.

Instead of only maintaining head vertices of cross-partition edges as replicas, we introduce the second hop replicas by replicating tail vertices of the first hop replicas. Similarly, more hops of replicas can be added to each partition. We call this approach as *multi-hop replication scheme*. Due to the message propagation nature of the EMV model, replica

Figure 4.5: Graph partitioning with multi-hop replication.

update only needs to be carried out after every n iterations if there are n hops of replicas. We call n iterations as a round and one *round* has n stages. As the stages are carried out outer hops of replicas are marked as “outdated”. That essentially uses the eventual consistency model, and the data are consistent after each round.

Figure 4.5 shows an example of the same graph as in Figure 4.4. Now *Partition 2* and *Partition 3* both maintain two-hop replication. The replicas need to be updated every two iterations, reducing the number of replica update by a half. In the first stage of each round, Medusa APIs are applied to all vertices in each partition. After that, the second hop replicas are outdated and are not processed in the second stage. After each round, the replicas are updated and a new round starts. Since messages only propagate among neighboring vertices, outdated replicas would not affect the computation results. To the user, the partitioned graph is always in a consistent state. Medusa code for single GPU can transparently run on multiple GPUs and produce exactly the same results.

As described above, increasing the number of replica hops can reduce the number of times of updating replicas. However, this scheme is not guaranteed to be beneficial compared with the basic replication scheme since more replicas and edges need to be processed. For example, maintaining multiple hops of replicas for dense graphs or small-world graphs with a small diameter can lead to explosive growth of replica vertices. However, since Medusa mainly deals with sparse graphs, multi-hop replication can be beneficial. For a given graph, we estimate the benefits of all possible hop numbers within the storage constraint and select the best one. Medusa uses a cost model to estimate the benefits of all possible hop numbers as detailed in the following paragraph.

Table 4.2: Notations in the cost model

n	The number of hops of replication.
TU_n	Time cost of updating n hops of replicas.
TE_n	Average reduced time cost of each iteration by updating n hops of replicas once a round instead of updating 1 -hop replicas once a stage.
TC_n	Computation time cost of processing n hops of replicas.
TS_n	Average time saving of each iteration using n -hop replication compared with using 1 -hop replication.

Cost model. Table 4.2 summarizes the notations used for our cost model. The total time cost of n -hop replication scheme is the sum of computation time on replicas and the replica update time. For each round of n stages, the n hops of replicas need to be updated only once. For an input graph with n -hop replication, we can first obtain the number of replicas. Once we know this number, we can get the size of data transfer for updating the replicas. We benchmark the system to obtain latency and bandwidth of PCI-e transfer, based on which we calculate TU_n . Comparing this with the time cost of updating one-hop replicas, we can get obtain TE_n .

$$TE_n = TU_1 - \frac{TU_n}{n}$$

Similarly, we can find out the number of extra calls on the EMV APIs. Next we calibrate the unit cost of running the APIs on a small graph to linearly approximate the cost in real execution and obtain the extra computation time TC_n for processing n hops of replicas. We determine n as the optimal number of hops of replication such that TS_n is maximized.

$$TS_n = TE_n - \frac{\sum_{i=1}^n TC_i}{n}$$

Also note that n should be bounded by the GPU memory size, since the required amount memory to store graph partitions increases with n .

Overlapping communication with computation. Latest GPUs can overlap kernel execution with PCI-e data transfer. We take advantage of this capability to reduce the overhead of inter-GPU communication. Since non-replicated vertices are not affected by

replication, EMV APIs can be applied to them before the replicas are updated. This allows us to overlap computation and replica update. After the replicas are updated, the same set of EMV APIs can be applied to the replicas. If the computation time on non-replicated vertices is larger than the overhead of replica updates, the overhead of replica updates can be fully masked.

4.4 Evaluation

In this section, we compare Medusa-based implementations with other existing or our homegrown implementations and evaluate the effectiveness of our optimizations.

4.4.1 Experimental Setup

We have conducted the evaluations on a workstation equipped with four NVIDIA Tesla C2050 GPUs, two Intel Xeon E5645 CPUs (totally 12 CPU cores at 2.4GHz) and 24GB RAM.

Our workloads include a set of common graph processing operations for manipulating and visualizing a graph on top of Medusa. The graph processing operations include PageRank, BFS, maximal bipartite matching (MBM), and SSSP. In order to assess the queue-based design in Medusa, we have implemented two versions of BFS: BFS-N and BFS-Q for the implementations without and with the usage of *SetActive* APIs, respectively. Similarly, we have also implemented two versions of SSSP: SSSP-N and SSSP-Q without and with the usage of *SetActive* APIs, respectively. The implementation details are presented in the following subsection. In the remainder of this section, we use “Medusa” to refer to the better-performing implementation of the two versions on BFS and SSSP, unless we specify “-N” and “-Q” explicitly.

Our experimental dataset includes two categories of sparse graphs: real-world and synthetic graphs. Table 4.3 shows their basic characteristics. We use the GTgraph graph generator [35] to generate power-law graph RMAT and Random graph. To evaluate MBM, we generate a synthetic bipartite graph (denoted as BIP), where vertex sets of two sides

Table 4.3: Characteristics of graphs used in the experiments

Graph	Vertices (10^6)	Edges (10^6)	Max d	Avg d	σ
RMAT	1.0	16.0	1742	16	32.9
Random (Rand)	1.0	16.0	38	16	4.0
BIP	4.0	16.0	40	4	5.1
WikiTalk (Wiki)	2.4	5.0	100022	2.1	99.9
RoadNet-CA (Road)	2.0	5.5	12	2.8	1.0
kkt_power (KKT)	2.1	13.0	95	6.3	7.5
coPapersCiteseer (Cite)	0.4	32.1	1188	73.9	101.3
hugebubbles-00020 (Huge)	21.2	63.6	3	3.0	0.03

have one half of the vertices and the edges are randomly generated. The real world graphs are publicly available [34, 104].

All the experiments are executed for ten runs and the average execution time is reported. The difference among runs for the same experiment is smaller than 2%. For BFS and SSSP, we randomly choose 100 source vertices and report the average execution time. We first evaluate Medusa on a single GPU, and then show performance of Medusa on multi-GPU in Section 4.4.7.

4.4.2 Graph Primitives

In this subsection, we present implementations of the four graph processing operations: PageRank, BFS, MBM, and SSSP. Those operations exhibit different graph operation patterns, which can also be used as building blocks for higher level applications such as graph-based analysis and ranking, community discovery and finding the influential nodes. We also implemented a set of visualization algorithms, which can be used to build interactive visualization assisted graph mining [105].

PageRank. As we can see from the source code of Medusa-based PageRank implementation in Chapter 3, Medusa first invokes an *ELIST* API on each vertex for sending messages to all neighbors in each iteration. Next, Medusa invokes the *Combiner* interface to calculate the sum of messages for each vertex. Finally, Medusa updates the rank of each vertex with an *VERTEX* API. Note that, we use an *ELIST* API in the first step to enable more generic PageRank implementation. Users actually could use an *EDGE* API (if appropriate) to leverage its finer-grained parallelism and more balanced execution.

BFS. BFS is a widely used graph search algorithm. The searching process starts from a predefined root vertex and iteratively expands to all the reachable vertices. In order to evaluate the queue-based design in Medusa, we have implemented two versions of BFS: BFS-N and BFS-Q for the implementations without and with the usage of *SetActive* APIs, respectively. With *SetActive* API, BFS-Q is work-efficient. In comparison, while BFS-N is work-inefficient, it is able to exploit the coalesced memory feature of the GPU.

In BFS-N, vertices are visited in levels, which correspond to the iterations in Medusa. An attribute *level* is added to the vertex structure to indicate its level. The root vertex is at the first level, and is accessed in the first iteration. In the iteration i , we invoke an *EDGE* API to access each edge and check whether the *level* attribute of its head vertex is equal to i . For each edge satisfying this condition, we set the *level* of the tail vertex to be $(i + 1)$, if the vertex has not been visited.

In BFS-Q, we invoke the *EDGE* API on the active edges only, unlike BFS on all the edges in the graph. Initially, we set the edges of the root vertex as active. In the *EDGE* API, we set edges of newly reached vertices as active. Medusa finishes execution when no edge is marked as active.

MBM. To implement the randomized maximal matching algorithm [106], two attributes are added to the vertex structure: *flag* denoting whether the vertex is in L set or in R set, and *mID* representing its matching vertex. *mID* is initialized to be *null*, meaning the vertex is unmatched.

The randomized maximal matching algorithm executes iteratively and each iteration has four phases: 1) Each unmatched L vertex sends messages to all its neighbors (which are all R vertices) and requests a match by invoking an *EDGE* API. 2) Each unmatched R vertex chooses one of its received messages randomly and sends a grant message to the sender of the chosen received message. This phase can be implemented using a *VERTEX* API. 3) If an unmatched L vertex receives any grant message, it sets its *mID* to the sender of the grant message. 4) Each L vertex, which sets its *mID* in the previous phase, notifies its matching R vertex about the newly established matching. For efficiency, we have implemented the last two phases using a single invocation of the *VERTEX* API.

SSSP. The SSSP problem is to find the shortest paths from a specified source vertex to all the other vertices. Same as the MTGL counterpart, we adopt the Bellman-Ford algorithm in our Medusa implementation. In particular, we use the individual API message *Combiner* to promote a more load-balanced implementation. The vertex structure has an attribute *distance* to denote its distance to the source vertex, and the edge structure has a *distance* attribute to indicate the distance from the head vertex to the tail vertex. Similar to BFS, we have also implemented two versions of SSSP: SSSP-N and SSSP-Q without and with the usage of *SetActive* APIs, respectively.

In the initialization of SSSP-N, the distance of the source vertex is initialized with zero and the distance of other vertices with ∞ . In each iteration, all the vertices which are updated in the previous iteration send the new distances to neighbors via outgoing edges plus the corresponding edge length. This is done by using an *EDGE* API. By applying a minimal operator on the message buffer using the *Combiner* interface, we can get the minimum message received by each vertex. Finally, a *VERTEX* API is used to update the distance of each vertex if the minimum message received is smaller than the current distance. This SSSP-N implementation is work-inefficient in two aspects. First, the *EDGE* API works on every edge, even if the head vertex of an edge is not updated in the previous iteration. Second, the *VERTEX* API works on every vertex. However, the vertices which receive no message do not require that processing.

SSSP-Q addresses the work-inefficient issues of SSSP-N with the *SetActive* API. We adopt the *SetActive* API so that the *EDGE* API is applied to the edges whose head vertex is updated in the previous iteration, and the *VERTEX* API is applied to the vertices that have received messages.

Visualization. We developed a set of visualization operations such as graph layout and drilling up/down. We implement the force direct layout algorithm [107]. The drilling up/down operation is implemented using BFS. We use OpenGL for graphics rendering and the CUDA-OpenGL interoperability support for collaboration between computation and visualization.

4.4.3 Comparison with Manual Implementations

We first compare the Medusa BFS and SSSP implementations with manual implementations of GPU graph processing: Harish’s work [2] and Hong’s work [4].

Harish’s work provides an open-source implementation of BFS and SSSP using CUDA and we tune the thread configuration and shared memory optimizations according to the C2050 Fermi architecture. We use it as the basic implementation. We implement the virtual warp-centric BFS proposed in Hong’s work [4]. The underlying difference between the Medusa implementation and the warp-centric method is that Medusa applies L threads to a vertex if that vertex has L edges, while the warp-centric method applies a virtual warp to a vertex. As a result, our method incurs more memory accesses because we check the head vertex status for every edge.

Table 4.4 shows the traversed edges per second (TEPS) comparison between the three implementations of BFS. Compared to the basic implementation, Medusa performs better on all graphs except KKT. Although Medusa incurs more memory access and runtime overhead than the highly optimized warp-centric method, Medusa outperforms warp-centric on some graphs and degrades the performance on other graphs. Despite the fact that the warp-centric method is more memory efficient than Medusa, there are two reasons causing its inferior performance on some of the input graphs: First, it’s vulnerable to outlier vertices which have very large degrees; Second, idle threads and uncoalesced access is not totally eliminated. In contrast, the Medusa-based implementation is insensitive to outliers and improves memory accesses thanks to the use of *EDGE* API. Note that the reported results of the warp-centric approach are better than those in the original paper [4], mainly because the GPU in our experiment is more powerful.

Figure 4.6 shows the performance comparison between Medusa and basic implementation of SSSP. Medusa provides comparable performance with the basic implementation except on Road and Huge. On large-diameter graphs such as Road and Huge, the performance of Medusa-based SSSP is notably worse than that of the basic implementation.

Table 4.4: Traversed edge per second (10^6 TEPS) comparison with manual implementations [2, 4].

	Basic	Warp-centric	Medusa
Wiki	61.4	152.9	1091.1
Road	26.2	45.7	63.5
RMAT	593.2	971.1	895.8
Rand	648.6	844.95	765.8
Huge	5.7	1.3	68.1
KKT	480.7	175.7	351.5
Cite	1460.4	1503.1	2686.7

Figure 4.6: Performance comparison between Medusa and existing GPU implementation of SSSP [2].

This is because the *Combiner* API invocation in SSSP takes a large part of its execution time and that overhead is almost fixed for every iteration.

Programmability is difficult for a quantitative comparison. As a start, we show the programmability comparisons on some major implementation issues of GPU programs in Table 4.5. Medusa simplifies GPU programming for graph processing, by significantly reducing the number of GPU-related source code lines written by users. This is because Medusa hides the GPU programming complexity by offering a small set of user-defined APIs. For example, users only need to write 7 and 11 lines of source code for defining the APIs in BFS-Q and SSSP-Q, respectively, whereas the basic implementation [2] has 56 and 59 lines of GPU-related code. Moreover, compared to manual implementations, Medusa requires no parallel or GPU specific programming.

Table 4.5: Coding complexity of Medusa implementation and manual implementations.

	Baseline	Warp-centric	Medusa (N/Q)
GPU code lines (BFS)	56	76	9/7
GPU code lines (SSSP)	59	N.A.	13/11
GPU memory management	Yes	Yes	No
Kernel configuration	Yes	Yes	No
Parallel programming	Thread	Thread+Warp	No

Table 4.6: Traversed edge per second (10^9 TEPS) comparison with Merrill et al.’s paper [5].

	Medusa	Contract-Expand [5]	Hybrid [5]
Huge	0.1	0.4	0.4
KKT	0.4	0.7	1.1
Cite	2.7	1.3	3.0

Overall, Medusa offers reasonable performance in comparison with manual implementations. With different design goals, Medusa is to offer good programmability with reasonable performance, whereas manual implementations usually do not consider programmability. Some techniques that are applicable to manual implementations may not be applicable to Medusa, if they hurt programmability.

We present more experimental results on BFS. Table 4.6 shows the comparison on BFS between Medusa-based implementation and the Contract-Expand and Hybrid approaches in Merrill et al.’s paper [5]. The Hybrid approach is more optimized than the Contract-Expand approach. The design and implementation of Medusa-based BFS is similar to the Contract-Expand approach, but targets at general graph processing. Overall, Medusa-based implementation can be slower than the Contract-Expand approach on some graphs such as Huge and KKT, and can be faster on other graphs such as Cite. On the other hand, Medusa-based implementation is slower than the Hybrid approach on all the three graphs. Compared with various specific optimizations for BFS, Medusa involves considerable runtime overhead in supporting general graph processing, for example, message passing-based mechanisms.

Figure 4.7: Performance impact of different graph layouts on PageRank using RMAT as input.

4.4.4 Impact of individual optimizations

In order to study the impact of a certain optimization, we conduct the experiment by disabling/enabling the optimization while other optimizations are enabled. In particular, we first evaluate the impact of different graph layouts, followed by our graph-aware buffer scheme.

Since we observed similar results on different operations, we present the results for PageRank.

Figure 4.7(a) compares the elapsed time for PageRank with Medusa using different graph storage layouts on RMAT. The edge-oriented storage layout results in “out-of-memory” problems, and thus its results are not included. The hybrid layout chooses the threshold value to be the average out-degree. Among different graph layouts, CAA is the best, the hybrid layout comes the second, and AA is the last. The ELIST API of PageRank with CAA is 29.6% faster than that with AA. This is correspondent with the number of memory requests to the device memory. We have performed profiling on the L2 cache with CUDA command line profiler. As shown in Figure 4.7(b), CAA has the smallest number of read misses on the L2 data cache, and the smallest number of read requests to the L2 data cache. This indicates that CAA is optimized with the coalesced memory accesses. The bandwidth utilization is improved, reaching over 50%.

We study the impact of different buffer schemes in Medusa with other optimizations enabled. We compare the elapsed time for PageRank of Medusa using different buffer

Figure 4.8: Graph aware buffer scheme.

schemes [101] on RMAT and Rand. It was shown that list-based buffer scheme is superior to the array-based scheme [101], thus we only compare with the list-based buffer scheme. Figure 4.8 shows that the overall performance improvement of graph aware scheme over the list-based scheme is 53% to 57%. The awareness of the graph structure significantly improves the efficiency of Medusa.

4.4.5 Comparison with CPU-based Execution

Overall comparisons. We implement the graph processing operations with MTGL [3], as the baseline for graph processing on multi-core CPUs.

The BFS and PageRank implementations are offered by MTGL and we implement the Bellman-Ford algorithm for single source shortest paths and a randomized maximal matching algorithm [106] using the MTGL APIs. We tuned the number of threads in MTGL and report the best result obtained when the number of threads was 12 on our machine. MTGL running on 12 cores is on average 3.4 times faster than that running on one core. Due to the memory intensive nature of graph algorithms, the scalability of MTGL is limited by the memory bandwidth.

Figure 4.9 shows the speedup for Medusa over MTGL running on 12 cores. The *speedup* is defined as the ratio between the elapsed time of the CPU-based execution and that of Medusa-based execution. PageRank is executed with 100 iterations. Medusa is significantly faster than MTGL on most comparisons and delivers a performance speedup of 1.0–19.6

Figure 4.9: Performance speedup of Medusa running on the GPU over MTGL [3] running on 12 cores.

with an average of 5.5 (we report the better results of the two implementations of BFS and SSSP, respectively). On some graphs such as Road, BFS-N is notably slower than MTGL-based BFS, because the work-inefficient issue of BFS-N is exaggerated on the graphs with large diameter. We show the performance speedup of comparing Medusa with MTGL with a single thread in Figure 4.10. The performance speedup over MTGL on a CPU core is 2.1–67.7, with an average of 15.2. Compared with the single thread MTGL, Medusa-based BFS-N is also slower on the Road graph because the large searching depth of the graph.

The work-efficient BFS and SSSP algorithms (BFS-Q and SSSP-Q) achieve better performance on the graphs with large diameters, and can degrade the performance in some cases (e.g., Rand, Wiki and KKT) due to the computation and memory overhead in maintaining the queue structure. This is consistent with the previous studies [4]. Currently, we leave the decision on whether to use the *SetActive* API to the users. In the future work, we consider whether this decision can be made automatically in Medusa.

4.4.6 Efficiency of Visualization Operations

The visualization experiment is conducted on a graphics workstation with one NVIDIA Quadro 5000 GPU and one Intel Xeon W3565 processor with 4 GB memory. We use the DBLP graph to evaluate visualization assisted graph mining. The DBLP graph is extracted

Figure 4.10: Performance speedup of Medusa running on the GPU over MTGL [3] running on one core.

from DBLP (<http://dblp.uni-trier.de/xml/>): each author as a vertex, and an edge meaning co-authorship between the two corresponding authors.

With Medusa, users can compose a variety of visualized graph discovery tasks. Additionally, combining computation and visualization into a single GPU eliminates the PCI-e data transfer for each invocation of graphics rendering.

Figure 4.11(a) shows the visualization result of the DBLP graph. On the Quadro 5000 GPU, Medusa takes only 120 seconds to get the overview, while the multi-threaded CPU implementation on the Intel Quad-core CPU takes over 1000 seconds. Figure 4.11(b) shows the two-hop neighbors of a selected author Jiawei Han obtained by a drill-down operation. Medusa greatly improves the responsiveness of graph visualization tasks.

4.4.7 Results on Multi-GPU Extension

We have evaluated the common graph processing algorithms, and present the results on BFS-N and PageRank. BFS-N exhibits a wave-front computing pattern and involves little computation, while PageRank has relatively heavy computation.

One-hop Replication with Execution/Memory Overlapping. Table 4.7 shows the performance of BFS-N and PageRank on the Road graph with the number of GPUs

Figure 4.11: Visualization results on DBLP co-authorship graph.

Table 4.7: Performance comparison of varying the number of GPUs with/without overlapping.

	#GPUs	PageRank (ms)	BFS-N (ms)
Basic	1	319.2	261.6
	2	203.2	347.3
	4	146.1	444.7
Overlap	2	200.2	197.1
	4	140.4	189.2

varied. Execution/memory overlapping improves the overall performance for both BFS-N and PageRank. Execution/memory overlapping brings more benefits to BFS-N. This is because the computation of BFS-N is lightweight and the inter-GPU communication time dominates the total execution time. On the other hand, communication takes a smaller portion of PageRank execution time. When overlapping is enabled, the speedup of BFS-N and PageRank compared with single-GPU implementation is 1.3–1.4 and 1.6–2.3, respectively.

Multi-hop Replication. Figure 4.12 shows the measured performance of BFS-N and PageRank with the number of hops in replication varied on four GPUs. Multi-hop replication effectively increases the performance of BFS-N. However, there is little benefit using this approach for PageRank. Since BFS-N carries a very small amount of computation

Figure 4.12: Execution time with increasing hops of replication.

Figure 4.13: Comparison between the predicted and actual measured TS .

and the communication overhead is the performance bottleneck, reducing the amount of data transfer can greatly reduce the overall execution time of BFS-N. In contrast, PageRank is computation bounded and communication cost takes a much smaller share of the total execution time. With multi-hop replication, the speedup of BFS-N and PageRank on four GPUs is increased up to 1.8 and 2.6, respectively.

We also evaluate the effectiveness of the suitable hop number estimation. Our estimation of TC_n and TE_n is plotted in Figure 4.13, together with the measured decreased time for one stage. Experiments on From the figure, we can see the prediction for BFS is quite

close to the actual saved time. The prediction for PageRank is less accurate and the prediction error is 30 % to 80 %. This is because PageRank includes more APIs and the combiner interface execution time is actually not in linear with the number of messages. Both prediction and actual measurement show positive effects on increasing the number of replication hops for BFS and PageRank.

We also evaluate the effectiveness of the suitable hop number estimation. We find that our cost estimation is sufficiently accurate to guide our decision on the suitable number of hops in replications. Take BFS-N and PageRank as an example. Figure 4.13 shows the predicted and measured TS for the two applications. The prediction on the decreased time with the increasing number of hops for BFS-N is quite close to the actual decreased time. In comparison, the prediction for PageRank is less accurate. This is because PageRank includes more APIs and the execution time of the *Combiner* interface does not conform to our linear cost assumption. Nevertheless, both the prediction and actual measurement show positive effects on increasing the number of replication hops for BFS-N and PageRank.

Replication for Power Law Graphs. Power-law graphs such as social networks and the web graph usually have very small diameters. Replication may replicate a large portion of the vertices for power-law graphs, and our cost model decides not to perform replication. We study the number of replicated vertices and execution time of Medusa on four GPUs with the number of hops in replication varied on RMAT. We find that one-hop replication leads to a significant portion of replicated vertices (over 96% of the vertices in RMAT) and the execution time is almost the same across different hops of replications (even with slight degradation sometimes).

4.5 Summary

In this chapter, we present the Medusa programming framework to address the efficiency and programmability of GPU-based parallel graph processing. We design Medusa to harness the magnificent power of GPUs for graph processing and simplify GPU programming as well. Medusa embraces a runtime with various optimizations specific to features of the

GPU architecture and characteristics of graph applications. Experimental results show that Medusa not only notably reduces the programming effort, but also delivers comparable or better performance over manually tuned GPU implementations and significant speedup over parallel CPU implementations.

Chapter 5

Kernelet: High-Throughput Concurrent Kernel Execution

In this chapter, we first briefly introduce the key techniques in Kernelet. Next, we review the related work on task scheduling on Simultaneous Multi-Threading (SMT) CPUs and concurrent kernel scheduling on GPUs. Next, we provide the definition of our problem, followed by the overview and detailed design of Kernelet. At the end of this chapter, we present the evaluation results of Kernelet.

5.1 Introduction

In shared environments like clusters and cloud, the GPU usually needs to execute kernels submitted by multiple applications at the same time. The submitted kernels may cover a wide range of computation and memory intensiveness. For graph processing, some applications like BFS are memory intensive and others like PageRank are more computation intensive. Thus, it is possible to better utilize GPU resources by scheduling kernels with different memory and computation characteristics. Also, with concurrent applications requesting to transfer data between the main memory and the GPU memory, it is possible to overlap the data transfer with kernel execution. The goal of Kernelet is to exploit the opportunities of concurrent kernel execution and data transfer overlapping to improve throughput of shared GPUs. To achieve this goal, we need to solve the following three problems. In the following, we present these problems and our proposed solutions.

First, due to the limitation of the current CUDA platform, concurrent executions of multiple optimized kernels almost degrade to sequential executions of individual kernels. This is because optimized kernels are usually configured with high occupancy to make the most use of the GPU. A high occupancy kernel excludes other kernels from executing on the GPU as long as it has sufficient number of unfinished blocks to fully occupy the GPU. To improve the utilization of concurrent kernel executions, we propose to slice the kernels into low occupancy small pieces and co-schedule slices from multiple kernels to realize real sharing of the GPU. Each slice contains a subset of thread blocks from the original kernel. Kernelet dynamically performs slicing on the kernels, and the slices are carefully designed with tunable occupancy to allow slices from other kernels to utilize the GPU resources in a complementary way. With automated kernel slicing tools on the compiled code, Kernelet does not require the source code of the kernels, and is completely transparent from the users. With kernel slicing, we convert the GPGPU concurrent kernel scheduling problem into the slice scheduling problem.

Second, we need a slice co-scheduling method which not only improves the throughput of concurrent executing kernels, but also incurs little overhead for online scheduling. The scheduling method should decide how to slice the kernels properly and how to select slices for co-scheduling. We propose to use performance models to estimate the co-scheduling performance of concurrent slices and determine the slicing and scheduling plans based on the estimated performance gain. However, existing GPU performance models (e.g., [32,33]) are limited to a single kernel only. We develop a novel and effective Markov chain based performance model to predict the performance of concurrent kernels. Experiment results show that our model offers sufficient accuracy to support the scheduling method.

Third, with multiple applications making requests to the GPU (including allocating GPU memory, executing kernels on the GPU, and transferring data between the main memory and GPU memory), we need a mechanism to schedule the applications to further increase the overall utilization of the GPUs and to avoid resource allocation deadlocks. For this purpose, we introduce the two-level scheduling mechanism of Kernelet including

the slice scheduler and the memory command scheduler. The memory command scheduler manages the device memory related commands in the user applications transparently. The memory command scheduler proactively exploits opportunities of overlapping data transfer with kernel execution. This increases the number of concurrent kernels that are ready to execute on the GPU and provides more opportunities for concurrent kernel execution.

5.2 Related Work

In this section, we first present the related work on concurrent task scheduling on SMT CPUs. Afterwards, we review related work on supporting and improving concurrent kernel executions on the GPU.

5.2.1 Scheduling on SMT CPUs

SMT has been an effective hardware technology to better utilize CPU resources. SMT allows instructions from multiple threads to be executed on the instruction pipeline at the same time. Various SMT aware scheduling techniques have been proposed to increase the CPU utilization [108–113]. The core idea of SMT aware scheduling is to co-schedule threads with complementary resource requirements. Performance models including the Markov chain-based ones have also been adopted for concurrent tasks modeling on the CPU [114–116]. Due to the architectural difference between the CPU and the GPU, their models are not applicable to the GPU.

Several mechanisms have been proposed to select the threads to be executed on the same SMT core, such as hardware counters feedback [108, 111], pre-execution of all combinations [109] and probabilistic job symbiosis model [113]. Serrano et al. [114, 115] developed a model to estimate the instruction throughput of super-scalar processors executing multiple instruction streams. Chen et al. [116] proposed an analytical model for estimating throughput of multi-threaded CPUs.

Despite the fruitful research results on SMT scheduling, they are not applicable to GPUs due to the architecture differences. First, main performance affecting issues are

different for CPU and GPU applications. L2 data cache is a key consideration issue for SMT-aware thread scheduling, whereas thread parallelism is usually more important for the performance of GPGPU programs [32, 33, 117]. Second, scheduling on GPUs is not as flexible as that on CPUs. Current GPUs do not support task preemption. Third, unlike CPUs supporting the concurrent execution of a relatively small number of threads, each GPGPU kernel launches thousands of threads. Additionally, the maximum number of co-scheduling threads equals the number of hardware context on the CPU, while the number of active warps on GPUs is dynamic, depending on the resource usage of the thread blocks. The slicing, scheduling and performance models in Kernelet are specifically designed for GPUs, taking those issues into consideration.

5.2.2 Concurrent Kernel Scheduling on GPUs

GPU architectures have undergone significant and rapid improvements for GPGPU support. Due to lack of concurrent kernel support in early GPU architectures, researchers initially proposed to merge two kernels at the source code level [118]. In those methods, two kernels are combined into a single kernel with if-else branches on different granularities (e.g., thread blocks). They have three major disadvantages compared with our approach. First, they will increase the resource usage of each thread block, leading to lower SM occupancy and performance degradation. Second, those approaches require source code, which is usually unavailable in shared environments. Third, it requires two kernels with different block sizes avoiding using barriers within the thread block, otherwise deadlocks may occur.

New-generation GPUs like NVIDIA Fermi GPUs support concurrent kernel executions. Taking advantage of this new capability, a number of multi-kernel optimization techniques [119–121] have been developed to improve the utilization of GPUs. Ravi et al. [27] proposed kernel consolidation to enable space sharing (different kernels run on different SMs) and time sharing (multiple kernels reside on the same SM) on GPUs. Space sharing happens when the total number of thread blocks of all kernels does not exceed the number of SMs and each block can be executed on a dedicated SM. If the total number

of thread blocks is larger than the number of SMs, while SMs have sufficient resources to accommodate more thread blocks from different kernels, time sharing happens. However, kernel consolidation does not have space sharing and has little time sharing when the launched kernels have sufficient thread blocks to fully occupy the GPU. Furthermore, they determined the kernel to be consolidated with heuristics based on the number of thread blocks. In contrast, Kernelet utilizes slicing to create more opportunities for time sharing, and develops a performance model to guide the scheduling decision. Wende et al. [122] exploited space sharing for concurrent execution of kernels with a small number of thread blocks. Peters et al. [123] used a persistently running kernel to handle requests from multiple applications. GPU virtualization has also been investigated [120,121]. Pai et al. [124] proposed *elastic Kernels* to allow concurrent execution of thread blocks from different kernels. However, their scheduling policies have not considered the co-scheduling opportunities from different memory/computation characteristics. Compared with elastic kernel transformation, kernel slicing not only incurs ignorable register overhead, but also requires no costly division or modulus operations in the kernel. Additionally, to enable sharing GPUs remotely, a number of software frameworks such as rCUDA [125] and V-GPU [126] have been developed.

Recent studies also address the problem of GPU scheduling when multiple users share one machine, e.g., RGEM [127] and PTask [119] manage the GPU at the operating system level. All those scheduling methods do not consider how to schedule concurrent kernels in order to fully utilize the GPU resources. Automatic memory and data transfer management has been studied [128]. Most studies rely on advanced programming analysis techniques to automate and optimize the data transfer between the GPU memory and the main memory. Those techniques are orthogonal to Kernelet. Pegasus [129] coordinates computing resources including accelerators and CPUs and provides a uniform resource usage model. Timegraph [130] and PTask [119] manage the GPU at the operating system level. Kato et al. [127] introduced the responsive GPGPU execution model (RGEM).

In addition to software techniques, Adriaens et al. [131] took a hardware-based approach, which partitions the GPU SMs into subsets and each subset is exclusively allocated to one

Figure 5.1: Application scenarios of concurrent kernel executions on the GPUs.

GPU task. Kernelet is complementary to this hardware approach by optimizing the resource utilization per SM.

As for performance models on GPUs, Hong [32] and Sim [33] proposed analytical models based on the round-robin warp scheduling assumption. All those models are designed for a single kernel. Moreover, they usually require extensive hardware profiling and/or simulation processes. In contrast, our performance model is designed for concurrent kernel executions on the GPU.

5.3 Problem Definition

In this subsection, we first introduce the application scenario, followed by the terminology and problem definitions.

Application scenario. We consider two typical application scenarios in shared environments as shown in Figure 5.1. One is sharing the GPUs among multiple tenants in the virtualized environment (e.g., cloud). As illustrated in Figure 5.1a, there is usually a GPU virtualization layer integrated with the hypervisor. Figure 5.1b shows the other scenario, in which GPU servers offer API reception softwares (like rCUDA [125]) to support local/remote CUDA kernel launches. In both scenarios, the GPU faces multiple pending kernel launch requests. Kernelet can be applied to schedule those pending kernels.

Our study mainly targets at the concurrent kernel executions on a single shared GPU. Kernelet can be extended to multiple GPUs with a workload dispatcher distributing tasks

Figure 5.2: Kernel state transitions.

to individual GPUs.

We make two assumptions. First, we assume kernels are usually throughput oriented, with flexibility on the response time for scheduling. Still, we do not assume *a priori* knowledge of the order of the kernel arrival. Second, we assume that thread blocks in a kernel are independent with each other. This assumption is mostly true for the GPGPU kernels due to SPMD programming model. Most kernels in NVIDIA SDK and benchmarks like Parboil [132] do not have dependency among thread blocks in the same kernel. This assumption ensures that our slicing technique on a given kernel is safe.

We formally define the terminology in Kernelet.

Kernel. A kernel $\mathcal{K}$ consists of k thread blocks with IDs, 0, 1, 2, ..., $(k - 1)$.

Kernel state. A general flow for a GPGPU program consists of three steps. First, the host code allocates GPU memory for input and output data, and copies input data from the main memory to the GPU memory. Second, the host code starts the kernel on the GPU. The kernel performs the task on the GPU. Third, when the kernel execution is done, the host code copies results from the GPU memory to the main memory. Thus, we define the following states for a kernel (as illustrated in Figure 5.2): (1) *waiting*: the kernel is in the waiting state, if its input data is not totally available on the GPU. Initially, the kernel is in the waiting state. (2) *ready*: all the input data for the kernel execution are available on the GPU. (3) *running*: the kernel is sliced and its slices are being scheduled for execution. (4) *termination*: all slices of the kernel finish execution and its memory resources can be

reclaimed.

Slice. A slice k is a subset of the thread blocks of a launched kernel. Block IDs of a slice is continuous in the block index space. The size of a slice s is defined as the number of thread blocks contained in the slice.

Slicing plan. Given a kernel $\mathcal{K}$, a slicing plan $\mathcal{S}(\mathcal{K})$ is a scheme slicing $\mathcal{K}$ into a sequence of n slices ($s_0, s_1, s_2, \dots, s_{n-1}$). We denote the slicing plan to be $\mathcal{K}=s_0, s_1, s_2, \dots, s_{n-1}$.

Co-schedule. Co-schedule cs defines concurrent execution of n ($n \geq 1$) slices, denoted as $s_0, \dots, s_{n-1}$. All the n slices are active on the GPU.

Scheduling plan. Given a set of n kernels $\mathcal{K}_0, \mathcal{K}_1, \dots, \mathcal{K}_n$, a scheduling plan $\mathcal{C}$ ($cs_0, cs_1, \dots, cs_{n-1}$) determines a sequence of co-schedules in their order of execution. cs_i is launched before cs_j if $i < j$. All thread blocks of the n kernels occur in one of the co-schedules once and only once. A scheduling plan embodies a slicing plan for each kernel.

Co-scheduling profit. We define the performance benefit of co-scheduling n kernels to be the *co-scheduling profit* ($CP = 1 - \frac{1}{\sum_{i=0}^{n-1} \frac{cIPC_i}{IPC_i}}$), where IPC_i and $cIPC_i$ are IPC for sequential execution and concurrent execution of kernel i , respectively. Our definition is similar to those in the previous studies on CPU multi-threaded co-scheduling [109, 133].

Problem definition. Given a set of kernels for execution, the problem is to determine the optimal scheduling plan (and slicing) so that the total execution time for those kernels is minimized. That corresponds to the maximized throughput. Given a set of n kernels $\mathcal{K}_0, \mathcal{K}_1, \dots, \mathcal{K}_{n-1}$, we aim at finding the optimal scheduling plan $\mathcal{C}$ for a minimized total execution time of $\mathcal{C}(\mathcal{S}_0(\mathcal{K}_0), \mathcal{S}_1(\mathcal{K}_1), \dots, \mathcal{S}_{n-1}(\mathcal{K}_{n-1}))$. Note, in the shared GPU environment, the arrival of new kernels may trigger re-optimization.

5.4 System Overview

In this section, we present the rationales on the Kernelet design, followed by a system overview.

5.4.1 Design Rationales

Since kernels are submitted in an ad-hoc manner, the scheduling decision has to be made in real time. The optimization process should take the newly arrived kernels into consideration. Moreover, our runtime system of slicing and scheduling should be designed with light overhead. The overhead of slicing and scheduling should be small compared with their performance gain.

Unfortunately, finding the optimal slicing and scheduling plans is a challenging task. The solution space for such candidate plans is huge. Due to the limited PCI-e bandwidth and GPU memory capacity, different memory management and data transfer orders result in different opportunities for co-scheduling optimizations. For slicing a kernel, we have the factors including the number of slices as well as the slice sizes. For scheduling a set of slices, we can generate different scheduling plans with co-scheduling slices from different kernels. All those factors are added up into a huge solution space. Considering newly arrived kernels makes the huge scheduling space even larger.

Due to real-time decision making and light-weight requirements, it is impossible to search the entire space to get a globally optimal solution. The classic Monte Carlo simulation methods are not feasible because they usually exceed our budget on the runtime overhead and violate the real-time decision making requirement. There must be a more elegant compromise between the optimality and runtime efficiency. Particularly, we make the following considerations.

First, inspired by the classical multi-level design on process scheduling in operating systems [134], we decouple the scheduling decisions into two independent levels: *memory command scheduler* on handling memory management and data transfer commands and *kernel scheduler* on performing co-scheduling on the *ready* GPU kernels. With the two-level scheduler design, the first level aims at fully utilizing the PCI-e bus bandwidth, and the second level aims at fully utilizing the GPU computation resources.

Second, the scheduling considers two kernels only. Previous study [133] on the CPU has shown, when there are more than two co-running jobs, finding the optimal scheduling plan

is an NP-complete problem. Following previous studies [27, 133], we make our scheduling decision on co-scheduling two kernels only.

Third, once we choose two kernels to schedule, their slice sizes keep unchanged until either kernel finishes.

5.4.2 Design Overview

We develop Kernelet as a runtime back end to generate the slicing and scheduling plans for optimized throughput. Kernelet also automatically manages the device memory to facilitate sharing GPU among multiple applications and exploits opportunities of overlapping PCI-e data transfer with kernel execution.

Figure 5.3 shows an overview of Kernelet. Kernelet maintains two queue structures: *waiting queue* and *ready queue*. The waiting queue stores kernels in waiting state, and the ready queue stores the kernels in ready state. Initially, kernels are submitted and are temporarily buffered in the waiting queue. The memory command scheduler automatically schedules execution of GPU memory commands. If a kernel has all its input data available on the GPU (i.e, its preceding memory commands in the program are all executed), the kernel is migrated from the waiting queue to the ready queue. The kernel scheduler performs co-scheduling on ready kernels based on our performance model, which estimates the performance of co-scheduling slices from two different kernels in a probabilistic manner. Usually, a kernel is submitted in the form of SASS or PTX code. Once the scheduling plan is obtained, the kernel slicer transforms the kernel code into slices. Then, the slices from the two kernels are co-scheduled and executed on the GPU until either kernel finishes.

5.5 Kernelet Methodology

In this section, we first briefly introduce our two-level scheduling mechanism. Next, we present the technical details of kernel slicing. Then, we introduce the scheduling algorithm of Kernelet, and the co-scheduling space pruning algorithm for reducing scheduling overhead. At last, we introduce our novel performance model, which provides profitability estimation for the Kernelet scheduler.

Figure 5.3: Design overview of Kernelet.

5.5.1 Two-level Scheduling

Our two-level scheduler design is inspired by the classic multi-level process scheduler design in operating systems [134]. Particularly, in operating systems, the memory and the CPU are two major resources and they are managed by two levels of schedulers: one for selecting the processes to allocate in the main memory (after memory allocations, processes are considered to be *ready*), and one for selecting the ready process to execute on the CPU. Similarly, a kernel also has states, depending on whether the input data are available on the GPU. Thus, we develop two levels of scheduling: memory command scheduler and kernel scheduler. We present details of the memory command scheduler in the following.

The memory command scheduler is responsible for handling memory commands from applications, preparing the input data for the kernel as well as copying the output data to the main memory when appropriate. All data transfers between the host and the GPU are managed by the memory command scheduler of Kernelet. The memory command scheduler maintains the memory commands for the same kernel into one queue. When all the memory commands of the kernel are available in the queue, the memory command scheduler schedules those commands. If the commands from more than one kernel are ready for execution, we adopt the first come, first serve (FCFS) algorithm for simplicity.

There are several implementation issues worth further discussions. First, we need to change the synchronous function calls for memory commands to asynchronous ones. As a result, those function calls simply put the commands into the queue of the corresponding kernel, and return immediately without really executing the memory commands. Second,

we perform the memory commands of the same kernel in a batch so that we can avoid the device memory allocation deadlock. Third, Kernelet prefers to allocate page-locked host memory to exploit the GPU’s capability of overlapping kernel execution with PCI-e data transfer. If the page-locked memory reaches the system limit, unlocked memory is allocated. In our experiments, the system limit for page-locked memory is 2.5 GB.

5.5.2 Kernel Slicing

The purpose of kernel slicing is to divide a (ready) kernel into multiple slices so that the finer granularity of each slice as a kernel can create more opportunities for time sharing. Moreover, we need to determine the minimum slice size for keeping low slicing overhead (i.e., the difference between the total execution time of all slices and the original kernel execution time). Particularly, we experimentally determine the the minimum slice with slicing overhead not greater than $p\%$ of the kernel execution time. In this study, $p\%$ is set to be 2% by default. We focus on the implementation of slicing on PTX or SASS code. Note that, we choose thread blocks as the units for slicing, instead of warps, because warps within the same thread block usually have data dependency with each other, e.g., with the usage of shared memory.

Kernelet performs kernel slicing by transforming a single kernel invocation into a set of invocations. We propose the *index rectification* process to ensure the output is exactly the same as the original kernel invocation. The transformation takes the PTX/SASS code of the kernel as input, and does not require the source code.

We describe the slicing algorithm with an example. Figure 5.4 illustrates the sliced execution of *MatrixAdd* with pseudo code. In the example, *MatrixAdd* is a GPU kernel to add two 256×256 matrices and each thread adds elements in the same position of the input matrices. Based on the matrix size, *MatrixAdd* is configured to launch with 16×16 thread blocks in total and each block has 16×16 threads. Figures 5.4a and 5.4b illustrate the definition and launch of the original kernel, respectively. In comparison, the sliced version of the kernel launches a slice with 8 thread blocks each time. The codes for the original

<pre> void MatrixAdd(float *MatrixA, float *MatrixB, int width){ int row = blockID_X*blockDim_X+threadID_X; int col = blockID_Y*blockDim_Y+threadID_Y; /* Each thread process one element */ A[row+col*width] += B[row+col*width]; } </pre>	<pre> /* Grid dimension: 16x16 */ dim3 gridConf(16,16); /* Block dimension: 16x16 */ dim3 blockConf(16,16); MatrixAdd<<<gridConf, blockConf>>>(...); </pre>
(a) Each thread access the corresponding matrix element using block and thread indices.	(b) Launch the same number threads as the number of matrix elements.
<pre> void MatrixAdd(float *MatrixA, float *MatrixB, int width, dim3 blockOffset, dim3 gridConf){ /* Compute rectified indices */ int rBlockID_X = blockID_X+blockOffset.x; int rBlockID_Y = blockID_Y+blockOffset.y; /* Process carry-over */ if (rBlockID_X > gridConf.x){ rBlockID_X -= gridConf.x; rBlockID_Y ++; } /* Replace all subsequent access to blockID_X (blockID_Y) with rBlockID_X (rBlockID_Y) */ ... } </pre>	<pre> /* Sliced grid dimension: 8x1 */ dim3 sGridConf(8,1); dim3 blockOffset = (0,0); while(blockOffset.x < gridConf.x && \ blockOffset.y < gridConf.y){ MatrixAdd<<<sGridConf, blockConf>>>(..., blockOffset, gridConf); blockOffset.x += sliceGrid.x; while (blockOffset.x > gridConf.x){ blockOffset.x -= gridConf.x; blockOffset.y ++; } } </pre>
(c) Sliced kernel with rectified thread block indices.	(d) Launch all slices of a kernel in a loop.

Figure 5.4: An example of kernel slicing.

and sliced kernels are mostly the same. The difference lies on how the built-in thread block indices are referenced. The block indices (denoted as *blockID_X* and *blockID_Y*) of the sliced kernel are in a smaller index space ($\{(x, 0) | 0 \leq x < 8\}$) compared with the original index space ($\{(x, y) | 0 \leq x < 16 \text{ and } 0 \leq y < 16\}$). Index rectification ensures that the combined output of all the slices is same as that of a single kernel. As shown in Figure 5.4c, we add an offset value to each thread block index and obtain the rectified index value. The rectified indices are used to replace all subsequent accesses to the built-in indices. On the CPU side, we launch the slices in a loop and adjust the offset values for each slice launch (Figure 5.4d). Note, this Figure 5.4 is simply for illustration purposes. The index rectification is performed on PTX or SASS code as described in the next paragraph.

We formally define index rectification as a function $R(B)$, where B is the set of block indices for kernel $\mathcal{K}$. $R(B)$ transforms B into a set of block indices with same size as B . For a three dimensional grid, B is equal to $\{(x_i, y_j, z_k) | x_i \in X, y_j \in Y, z_k \in Z\}$. Note that one and two dimensional grids can also be viewed as three dimensional grids since any

dimension can have just one value (for example, for two dimensional grids we have $Z = \{0\}$). Similarly, we define b_i as the set of block indices for slice k_i of kernel $\mathcal{K}$ ($0 \leq i < n$). We have the following condition hold.

$$\bigcup_{i=0}^{n-1} R(b_i) = B \quad (5.1)$$

Kernelet automatically implements index rectification without any user intervention. Taking PTX/SASS code as input, Kernelet does not require CUDA C source code. This makes Kernelet viable as a transparent back end. Kernelet interprets and modifies the PTX/SASS code at runtime. The resulting PTX code is compiled to GPU executables by the GPU driver, and the SASS code is assembled using the open source Fermi assembler Asfermi [135]. Figure 5.5 shows an example of index rectification on PTX code of the *MatrixAdd* kernel. To save space, we only show the code for setting up the matrix indices. Figure 5.5a shows the original input code for setting x (*blockID_X*) index and y (*blockID_Y*) index, which corresponds to the first two lines of code in the kernel function shown in Figure 5.4a. Figure 5.5b shows the resulting code after index rectification, which corresponds to the first two lines of code in the kernel function shown in Figure 5.4c. Kernelet stores the rectified x and y index in register *rect_ctaid_x* and *rect_ctaid_y*, respectively. All references to the built-in block indices are then replaced with the corresponding registers¹.

¹Details of PTX programming can be found in [136]

```

1  .entry MatrixAdd(
2      .param .u64 MatrixAdd_param_0,
3      .param .u64 MatrixAdd_param_1,
4      .param .u32 MatrixAdd_param_2
5  )
6  {
7      .....
8      mov.u32    %r2, %ntid.x;
9      //Load X index.
10     mov.u32    %r3, %ctaid.x;
11     mov.u32    %r4, %tid.x;
12     mad.lo.s32    %r5, %r2, %r3, %r4;
13     mov.u32    %r6, %ntid.y;
14     //Load Y index.
15     mov.u32    %r7, %ctaid.y;
16     mov.u32    %r8, %tid.y;
17     mad.lo.s32    %r9, %r6, %r7, %r8;
18     .....
19     exit;
20 }

```

(a) PTX code of *MatrixAdd* kernel generated by CUDA compiler.

```

1  .visible .entry MatrixAdd(
2      .param .u32 cta_id_x_offset,
3      .param .u32 cta_x_dim,
4      .param .u32 cta_id_y_offset,
5      .param .u64 MatrixAdd_param_0,
6      .param .u64 MatrixAdd_param_1,
7      .param .u32 MatrixAdd_param_2
8  )
9  {
10     .....
11     /* Declare virtual registers for computation
12        and storage of rectified thread block indices. */
13     .reg .u32 %rect_ctaid_x; //rectified blockID_X
14     .reg .u32 %rect_ctaid_y; //rectified blockID_Y
15     .reg .u32 %rect_ntid_x;  //x dimension of the grid
16     .reg .pred p_reg;        //predicate register
17     .reg .u32 %temp_reg;     //temporary register
18     /* Compute rect_ctaid_x. */
19     /* Load X offset. */
20     ld.param.u32    %temp_reg, [cta_id_x_offset];
21     mov.u32    %rect_ctaid_x, %ctaid.x;
22     /* Add X index with X offset and process carry-over. */
23     add.u32    %rect_ctaid_x, %rectaidx, %temp_reg;
24     ld.param.u32 %rect_ntid_x, [cta_x_dim];
25     setp.ge.u32 p_reg, %rectaidx, %rect_ntid_x;
26     @p_reg sub.u32 %rectaidx, %rectaidx, %rect_ntid_x;
27     /* Compute rect_ctaid_y. */
28     /* Load Y offset. */
29     ld.param.u32    %temp_reg, [cta_id_y_offset];
30     mov.u32    %rect_ctaid_y, %ctaid.y;
31     /* Add Y index with Y offset. */
32     add.u32    %rect_ctaid_y, %rect_ctaid_y, %temp_reg;
33     /* Process carry-over from rect_ctaid_x. */
34     @p_reg add.u32 %rect_ctaid_y, %rect_ctaid_y, 1;
35     .....
36 }

```

(b) Rectified PTX code of *MatrixAdd* kernel generated by Kernelet. The first three parameters are added to pass the index offset values and block dimension of the unsliced kernel.

Figure 5.5: Index rectification of *MatrixAdd* kernel based on PTX code.

This rectification process may introduce extra register usage. We briefly describe our optimization on reducing the register usage in this paragraph. If the input is PTX code, Kernelet simply allocates new virtual registers and relies on the CUDA compiler to optimize the kernel register usage in the native code generation stage. If the input is SASS code, we adopt the classic register usage optimizations in compiler [137]. We try to reuse allocated registers with the beginning points of live intervals coming after references to the built-in block indices. If there are no such registers, new registers have to be allocated. Note, kernel slicing only requires a single scan on the input code and the runtime overhead is negligible. In our experiments, we find that register usage after slicing keeps unchanged in most of our test cases, because of two major observations. First, block indices are usually referenced at the beginning of the kernel to get the input data for the thread. In those cases, their live intervals do not overlap with registers allocated for the actual computation. Second, the number of index registers is small (usually ≤ 3 in CUDA).

5.5.3 Kernel Co-Scheduling

According to our design rationales, our scheduling decision is made on the basis of two ready kernels, to avoid the complexity of scheduling three or more kernels as a whole. Thus, we develop a greedy scheduling algorithm, as shown in Algorithm 1. The scheduling algorithm considers new arrival kernels in Lines 2–4 in the main algorithm. The main procedure calls the procedure *FindCoSchedule* to obtain the optimal co-schedule in Line 6. The co-schedule is represented in four parameters $\langle \mathcal{K}_1, \mathcal{K}_2, size_1, size_2 \rangle$, where $\mathcal{K}_1$ and $\mathcal{K}_2$ denote the two selected kernels with slice sizes $size_1$ and $size_2$, respectively.

In Procedure *FindCoSchedule*, we first consider the entire candidate space consisting of co-schedules on pair-wise kernel combinations. Because the space may consist of C_n^2 co-schedules (n is the number of kernels for consideration), it is desirable to reduce the search space. Therefore, we perform pruning according to the computation and memory characteristics of input kernels as detailed in the next subsection. After pruning, we apply the performance model (Section 5.5.5) to estimate the CP for all co-schedules, and pick the one with the maximized CP for execution on the GPU.

Procedure. 1 Scheduling algorithm of Kernelet

```

1: Denote  $\mathcal{R}$  to be the set of kernels pending for executions;
2: if A new kernel call  $\mathcal{K}$  comes then
3:   Add  $\mathcal{K}$  into  $\mathcal{R}$ ;
4:   Perform profiling if the kernel has not been submitted before;
5: end if
6: while  $\mathcal{R} \neq \text{null}$  do
7:    $\langle \mathcal{K}_1, \mathcal{K}_2, size_1, size_2 \rangle = \text{FindCoSchedule}(\mathcal{R})$ ;
8:   Denote the co-schedule to be  $c$ ;
9:   Execute  $c$  on the GPU;
10:  while  $\mathcal{R}$  does not change, and  $\mathcal{K}_1$  and  $\mathcal{K}_2$  both still have thread blocks do
11:    Generate co-schedule according to  $c$  and execute it on the GPU;
12:  end while
13: end while

```

Proc. *FindCoSchedule*($\mathcal{R}$)**Function:** generate the optimal co-schedule from $\mathcal{R}$.

```

1: Generate the candidate space for co-schedules  $\mathcal{C}$ ;
2: Perform pruning on  $\mathcal{C}$  according to the computation and memory characteristics of input kernels;
3: Apply the performance model (Section 5.5.5) to compute  $CP$  for all the co-schedule in  $\mathcal{C}$ ;
4: Obtain the optimal co-schedule with the maximized  $CP$ ;
5: Return the result co-schedule;

```

5.5.4 Co-scheduling Space Pruning

Given a set of co-schedules as input, we aim at developing pruning techniques which removes the co-schedules that are not “promising” to deliver high performance gains. Our pruning is based on the key performance factors of a single kernel that affect the throughput of concurrent kernel executions on the GPU.

There are many factors affecting the GPU performance. According to the CUDA profiler, there are around 100 profiler counters and statistics for the performance tuning. For an effective scheduling algorithm, there is no way of considering all these factors. Following a previous study [111], we use the regression model to explore the correlation between the above mentioned factors and CP . Through detailed performance studies, we find that instruction throughput and memory bandwidth utilization are the most correlated performance factors. We define *Pipeline Utilization Ratio (PUR)* and *Memory-bandwidth Utilization Ratio (MUR)* to characterize user submitted kernels. High PUR means that the

 Figure 5.6: Correlation between MUR/PUR and CP .

instruction pipeline is highly utilized and there is little room for performance improvement. High MUR means that a large number of memory requests are processed and memory latency is high. PUR and MUR is calculated as follows,

$$PUR = \frac{Instruction_Executed}{Time \times Frequency \times Peak_IPC}$$

$$MUR = \frac{Dram_Reads + Dram_Writes}{Time \times Frequency \times Peak_MPC}$$

$Peak_IPC$ and $Peak_MPC$ represent the peak numbers of instructions and memory requests per cycle of the GPU, respectively. $Instruction_Executed$ is the total number of instructions executed. $Dram_Reads$ and $Dram_Writes$ are the numbers of read and write requests to DRAM, respectively.

We build a set of testing kernels to demonstrate the correlation between PUR/MUR and CP . A testing kernel is a mixture of memory and computation instructions. We tune the respective instruction ratios to obtain kernels with different memory and computational characteristics. The single kernel execution PURs and MURs of the testing kernels are in the range of $[0.26, 0.83]$ and $[0.07, 0.84]$, respectively. Figure 5.6 shows the strong correlation between MUR/PUR and CP . The observation conforms to our expectation on co-scheduling. First, if one kernel has high PUR while the other kernel has low PUR,

the former kernel is able to utilize the idle cycles exposed by the latter kernel. Second, co-scheduling kernels with complementary memory requirements (one kernel has low MUR and the other kernel has high MUR) will alleviate memory contention and reduce idle cycles exposed by long latency memory operations.

In summary, our pruning rule is to remove the co-schedules where the two kernels have close PUR or MUR values. We set two threshold values α_p and α_m for PUR and MUR, respectively. That means, we prune the co-schedule if the two kernels have PUR difference lower than α_p , or have MUR difference lower than α_m . Note, if all the co-schedules are pruned, we need to increase α_p or α_m . We experimentally evaluate their impact in Section 5.6.

5.5.5 Performance Model

We need a performance model for two purposes: firstly, to select the two kernels for co-scheduling; secondly, to determine the number of thread blocks for each kernel in the co-schedule (i.e., the slice size). Previous performance models on the GPU [32, 33, 138] assume a single kernel on the GPU, and are not applicable to concurrent kernel executions. They generally assume that the thread blocks execute the same instruction in a round-robin manner on an SM. However, this is no longer true on concurrent kernel executions. The thread blocks from different kernels have interleaving executions, which cause non-determinism on the instruction execution flow. It is not feasible to statically predict the interleaving execution patterns for warps from multiple kernels. To capture the non-determinism, we develop a probabilistic performance model to estimate the performance of co-schedule. Our performance model has very low runtime overhead, because it uses a series of simple parameters as input and leverages the Markov chain theory to get the performance of concurrent kernel executions.

Table 5.1 summarizes the notations used for our performance model.

Since the GPU adopts SPMD model, we use the performance estimation of one SM to represent the aggregate performance of all SMs on the GPU. We model the process of

Table 5.1: Notations in the performance model

Para.	Description
W	Maximum number of active warps
$Round$	A warp scheduling cycle that all ready warps are served by the warp scheduler.
D_i	Round duration when there are i idle warps
R_m	Memory instruction ratio
$P_{r \rightarrow r}$	Probability that a ready warp remains ready
$P_{r \rightarrow i}$	Probability that a ready warp transits to idle
$P_{i \rightarrow r}$	Probability that an idle warp transits to ready
$P_{i \rightarrow i}$	Probability that an idle warp remains in idle
$N_{r \rightarrow r}$	Number of ready warps that remain ready
$N_{r \rightarrow i}$	Number of ready warps that transit to idle
$N_{i \rightarrow r}$	Number of idle warps that transit to ready
$N_{i \rightarrow i}$	Number of idle warps that remain idle
L	Average memory latency (cycle)
S_i	S_i corresponds the state where i warps are idle on the SM ($i = 0, 1, \dots, W$).
P_{ij}	P is the Markov chain transit matrix. Entry P_{ij} of P represents the probability of transiting from state S_i to state S_j .

kernel instruction issue as a stochastic process and devise a set of states for an SM during execution. Based on the state transition probabilities of the SM, we develop our Markov chain-based model for single-kernel executions (homogeneous workloads). We then extend the model for concurrent kernel executions (heterogeneous workloads).

For presentation clarity, we begin with our description on the model with the following assumptions, and relax those assumptions at the end of this section. First, we assume that all the memory requests are coalesced. This is the best case for memory performance. We will relax this assumption by considering both coalesced and uncoalesced memory accesses. Second, we assume that the GPU has a single warp scheduler. We will extend it to the GPU with multiple warp schedulers.

Homogeneous Workloads. We first investigate the performance of a single kernel executed on the GPU and each SM accommodates W active warps at most.

An active warp can be in either of two states: idle or ready. An idle warp is stalled by memory access, while a ready warp has at least one instruction ready for execution. Its

Figure 5.7: Warp state transition diagram.

state transition is illustrated in Figure 5.7. When a warp is currently in the ready state, we have two cases for state transitions by definition:

- remaining in the ready state with the probability of $P_{r \rightarrow r} = 1 - R_m$.
- transiting to the idle state with the probability of $P_{r \rightarrow i} = R_m$.

When a warp is currently in the idle state, we also have two cases for state transitions:

- transiting to the ready state with the probability of $P_{i \rightarrow r} = \frac{1}{\frac{L}{W-I}} = \frac{W-I}{L}$, where I is the number of idle warps on the SM. ($W - I$ warps will be executed in a round, and $W - I$ is also the time duration for the execution of one round.)
- remaining in the idle state with the probability of $P_{i \rightarrow i} = 1 - P_{i \rightarrow r} = \frac{L-W+I}{L}$.

We specifically define the time step and state transition of the Markov chain model to capture the GPU architectural features. Although lack of official documentation, we assume the GPU adopts a round-robin style warp scheduling policy to minimize block synchronization overhead. This assumption is also adopted in previous studies [32, 139]. In each *round*, the warp scheduler polls each warp to issue its ready instructions so all ready warps can make progress. We model the SM state based on the number of idle warps. We denote S_i to be the SM state where i warps are idle on the SM ($i = 0, 1, \dots, W$). Thus, we consider the state change of the SM in one round and use round as time step in our Markov chain-based model. In each round, every ready warp has an equal chance to issue instructions. In contrast, models for the CPU usually assume that the CPU will keep executing one thread until this thread is suspended and model each cycle as a step.

We use IPC to represent the throughput of the SM. Thus, the number of idle warps on the SM is a key parameter for IPC . More outstanding memory requests usually lead to higher latency because of memory contention [139]. We adopt a linear memory latency model to account for the memory contention effects. We calculate L as $L = a_0 \cdot x + b_0$, where x is the number of outstanding memory requests, and a_0 and b_0 are the constant parameters in the linear model. We follow the previous micro benchmarks on varying the number of outstanding memory requests [139] to obtain a_0 and b_0 .

For homogeneous workload, the probabilities of state transitions are the same for all ready warps in a round. We assume when SM transits from S_i to S_j , $N_{i \rightarrow r}$ idle warps transit to the ready state and $N_{r \rightarrow i}$ ready warps transit to idle state. The following conditions hold by definition.

$$\begin{cases} 0 \leq N_{i \rightarrow r} \leq S_i \\ 0 \leq N_{r \rightarrow i} \leq W - S_i \\ N_{r \rightarrow i} - N_{i \rightarrow r} = S_j - S_i \end{cases} \quad (5.2)$$

With those constraints, there are multiple possible transitions from S_i to S_j (the transition probability is denoted as P_{ij}). Since the possible transitions are mutually exclusive events, P_{ij} is calculated as the sum of the probabilities of all possible transitions. With all entries of the transition matrix P obtained, we can calculate the steady-state vector of the Markov chain. This is done by finding the eigenvector π corresponding to the eigenvalue one for matrix P [140].

$$\pi = (\gamma_0, \gamma_1, \dots, \gamma_W) \quad (5.3)$$

In Equation 5.3, γ_i is the probabilities that the SM is in state S_i in each round, i.e., the probability there are i idle warps in one round. The duration of the time step is $(W - i)$ cycles. In the case $i = W$, the round duration is one, indicating no warp is ready and the SM experiences an idle cycle. Hence, the estimated IPC is the ratio of non-idle cycles given

in equation 5.4, where $\sum_{i=0}^{W-1} \gamma_i(W-i)$ is the total non-idle cycles and γ_W is the total idle cycles.

$$IPC_K = \frac{\sum_{i=0}^{W-1} \gamma_i D_i}{\sum_{i=0}^W \gamma_i D_i} \quad (5.4)$$

Heterogeneous Workloads. When there are multiple kernels running concurrently, the model needs to keep track of the state of each workload. Although we only consider two concurrent kernels ($\mathcal{K}_1$ and $\mathcal{K}_2$) in scheduling, our model can be used to handle more than two kernels.

Suppose there are two kernels $\mathcal{K}_1$ and $\mathcal{K}_2$, and each has w_1 and w_2 active warps, respectively ($w_1 + w_2 = W$). The number of possible states of the SM will be $(w_1 + 1) \times (w_2 + 1)$. The state space is represented as a value pair (p, q) with $0 \leq p \leq w_1$ and $0 \leq q \leq w_2$, where p and q are the numbers of idle warps of $\mathcal{K}_1$ and $\mathcal{K}_2$, respectively. We first consider individual workload state transition probabilities based the single kernel model. State transitions of the two kernels are independent with each other, since they are scheduled independently. Thus, the probability that the SM transits from state $(p_i, q_{i'})$ to state $(p_j, q_{j'})$ is the product of the individual transition probabilities.

The steady state vector of our two kernel Markov chain-based model is denoted as $\pi = \{\gamma_{(0,0)}, \gamma_{(0,1)}, \dots, \gamma_{(w_1, w_2)}\}$. With π , we calculate the *IPC* of each workload using the same method for homogeneous workloads, except the parameters are defined and calculated in the context of two kernels. For example, the round duration is equal to the total number of ready warps of both kernels. Individual IPCs of $\mathcal{K}_1$ and $\mathcal{K}_2$ is calculated as the ratio of non-idle cycles for each workload, as shown in Eq. (5.5) and (5.6), respectively. The concurrent IPC is the sum of individual IPCs (Eq. (5.7)).

$$IPC_{\mathcal{K}_1} = \frac{\sum_{i=0}^{w_1-1} \sum_{i'=0}^{w_2} \gamma_{(i,i')} \times (w_1 - i)}{\sum_{i=0}^{w_1} \sum_{i'=0}^{w_2} \gamma_{(i,i')} \times D_{(i,i')}} \quad (5.5)$$

$$IPC_{\mathcal{K}_2} = \frac{\sum_{i=0}^{w_1} \sum_{i'=0}^{w_2-1} \gamma_{(i,i')} \times (w_2 - i')}{\sum_{i=0}^{w_1} \sum_{i'=0}^{w_2} \gamma_{(i,i')} \times D_{(i,i')}} \quad (5.6)$$

$$C = IPC_{\mathcal{K}_1} + IPC_{\mathcal{K}_2} \quad (5.7)$$

With the estimated IPC, we now discuss how to estimate the optimal slice size ratio for two kernels. We define the slice ratio which minimizes the execution time difference of co-scheduled slices as the *balanced slice ratio*. By minimizing the execution time difference, the kernel-level parallelism is maximized. The execution time difference is calculated as ΔT in Eq. (5.8).

$$\Delta T = \left| \frac{1}{IPC_{\mathcal{K}_1}} \times I_{\mathcal{K}_1} \times P_{\mathcal{K}_1} - \frac{1}{IPC_{\mathcal{K}_2}} \times I_{\mathcal{K}_2} \times P_{\mathcal{K}_2} \right| \quad (5.8)$$

$I_{\mathcal{K}_i}$ and $P_{\mathcal{K}_i}$ represent the number of instructions per block and the slice size of kernel $\mathcal{K}_i$ ($i = 1, 2$). Since $P_{\mathcal{K}_i}$ is less than the maximal number of active thread blocks, only a limited number of slice ratios need to be evaluated to get the balanced ratio.

Uncoalesced Access. So far, we assume that all memory accesses are coalesced and each memory instruction results in the same number of memory requests. However, due to the different address patterns, memory instructions may result in different amounts of memory requests. On Fermi GPUs, one memory instruction can generate 1 to 32 memory requests. Here we consider the two most common access patterns: fully coalesced access, and fully uncoalesced access. We extend our model to handle both coalesced and uncoalesced accesses by defining three states for a warp: ready, stalled on coalesced access (coalesced idle), and stalled on uncoalesced access (uncoalesced idle). The memory operation latency depends on the memory access type. Since uncoalesced access generates more memory traffic, its latency is higher than that of coalesced access. We also use the linear model to estimate the latency. By identifying the ratio of coalesced and uncoalesced memory instructions, we can easily extend the two-state model to handle three states and their state transitions. Distinguishing between coalesced and uncoalesced accesses increases the accuracy of our model.

Adaptation to GPUs with multiple warp schedulers. Our model assumes there is only one warp scheduler. New-generation GPUs can support more than one warp schedulers. The latest Kepler GPU features four warp schedulers per SMX (SMX is the Kepler

terminology for SM). We extend our model to handle this case by deriving a single pipeline virtual SM based on the parameters of the SMX. The virtual SM has one warp scheduler, and its parameters such as active thread blocks and memory bandwidth are obtained by dividing the corresponding parameters of the SMX by the number of warp schedulers. This virtual SM can still capture the memory and computation features of a kernel running on the SMX. Experimental results in Section 5.6.3 show that performance modeling on the virtual SM provides a good estimation on the Kepler architecture.

There are two more issues that are worthwhile to discuss.

The first issue is on the efficiency of executing our model at runtime. We have developed mechanisms to make our model more efficient without significantly sacrificing the model accuracy. The $O(N^3)$ complexity of calculating the steady state in Markov chain makes it hard to meet the online requirement (N is the dimension of the transition matrix). To reduce the computational complexity, we consider the thread block as a scheduling unit, instead of considering individual warps. In this way, the computational complexity is significantly reduced, and time cost of our model is negligible to the GPU kernel execution time.

The second issue is on getting the input for the model. We perform profiling for single kernel execution to get the values for the key factors to the performance of concurrent kernel execution. We adopt an online and light-weight approach. Particularly, we simply run one kernel slice without co-scheduling and obtain the profiling results on PUR and MUR. Since a kernel usually has many slices, the profiling overhead is small. Because of the SPMD execution model, co-scheduling decisions are applicable to other slices in the kernel and the future submission of the same kernel. We can also obtain the number of memory requests issued and the total number of instructions executed, and calculate Rm as their ratio. Note that the profiling process is transparent to users.

In summary, our probabilistic model has captured the inherent non-determinism in concurrent kernel executions. First, it simply requires only a small set of profiling results on the memory and computation characteristics of individual kernels. Second, with careful

Table 5.2: GPU configurations.

	C2050	GTX680
Architecture	Fermi GF110	Kepler GK104
Number of SMs	14	8
Number of cores per SM	32	192
Core frequency (MHz)	1147	706
Global memory size (MB)	3072	2048
Global memory bandwidth (GB/s)	144	192

probabilistic modeling, we develop a performance model that is sufficiently accurate to guide our scheduling decision. The effectiveness of our model will be evaluated in the experiments (Section 5.6).

5.6 Evaluation

In this section, we first evaluate Kernelet as an independent system on latest GPU architectures with micro benchmarks. At the end, we present the experimental results on improving throughput of Medusa applications using Kernelet.

5.6.1 Experimental Setup

We have conducted experiments on a workstation equipped with one NVIDIA Tesla C2050 GPU, one NVIDIA GTX680 GPU, two Intel Xeon E5645 CPUs and 24GB RAM. We note that C2050 and GTX680 are based on Fermi and Kepler architectures, respectively. One C2050 SM has two warp schedulers, and each can serve half a warp per cycle (with a theoretical *IPC* of one). In contrast, one GTX680 SMX features four warp schedulers and each warp scheduler can serve one warp per cycle (with a theoretical *IPC* of eight considering its dual-issue capability). Our implementation is based on GCC 4.6.2 and NVIDIA CUDA toolkit 5.0.

GPU. Table 5.2 shows some architectural features of C2050 and GTX680.

Workloads. We choose eight benchmark applications with different memory and computation intensivenesses. Sources of the benchmarks include the CUDA SDK, Parboil

Table 5.3: Specification of benchmark applications and thread configuration (#threads per thread block $\times$ #thread blocks).

Name	Description	Input settings	Thread configuration on C2050
Pointer Chasing (PC)	Traversing an array randomly	Index values for 40 million accesses	256×16384
Sum of Absolute Differences (SAD)	An operation used in MPEG encoding	Image with 1920×1072 pixels	32×8048
Sparse Matrix Vector Multiplication (SPMV)	Multiplying a sparse matrix with a dense vector.	A 131072×81200 matrix with 16 non-zero elements per row on average	256×16384
Stencil (ST)	Stencil operation on a regular 3-D grid	3D grid with 134217728 points	128×16384
Matrix Multiplication (MM)	Multiplying two dense matrices	One 8192×2048 matrix, the other 2048×2048	256×16384
Magnetic Resonance Imaging - Q (MRIQ)	A matrix operation in magnetic resonance imaging	2097152 elements	256×8192
Black Scholes (BS)	Black-Scholes Option Pricing	40 million	128×16384
Tiny Encryption Algorithm (TEA)	A thread block cipher	20971520 elements	128×16384

Benchmark [132], CUSP [141] and our home grown applications. The benchmark applications include: pointer chasing (PC), sum of absolute differences (SAD), SPMV, matrix multiplication (MM), magnetic resonance imaging-Q (MRIQ), Black Scholes (BS), and tiny encryption algorithm (TEA).

Table 5.3 describes the details of each application, including the default input setting and thread configuration on C2050 of the most time-consuming kernel.

Table 5.4 shows the memory and computation characteristics of the most time-consuming kernel of each application on both C2050 and GTX680. We observed that the PUR/MUR values are stable as we vary the input sizes (as long as the input size is sufficiently large to achieve the kernels' highest occupancy).

To evaluate the two-level scheduler design in Kernelet, we define a *scale factor* (sf) for

Table 5.4: Memory and computational characteristics of benchmark applications.

Benchmarks	C2050			GTX680		
	PUR	MUR	Occupancy	PUR	MUR	Occupancy
PC	0.01	0.14	100%	0.01	0.17	100%
SAD	0.15	0.11	17%	0.11	0.14	25%
SPMV	0.35	0.003	100%	0.30	0.004	100%
ST	0.36	0.12	68%	0.20	0.12	100%
MM	0.58	0.02	68%	0.53	0.06	100%
MRIQ	0.85	0.0002	83%	1.68	0.0007	100%
BS	0.86	0.06	68%	1.20	0.13	100%
TEA	1.00	0.02	68%	1.14	0.04	100%

each application. We define scale factor equals one, when input data size of the application equals the default size. Given a scale factor of sf , we scale the input data size of the application to be sf times as that with the scale factor of one. We vary the scale factor to evaluate different cases for computation/memory characteristics. Particularly, we are interested in two cases. In *Case I*, all the scale factors equal one ($sf = 1$). The transfer of input data is done once at the beginning and repeatedly accessed by serials of kernel invocations. This case represents the applications where many kernels continuously access the input data that can fit into the GPU memory (e.g., GPU-based query processing in databases [25]). In this case, the data transfer overhead is ignorable (the input data transfer is amortized among many kernel executions and the output transfer is overlapped with the kernel execution). Thus, we can study the sole impact of kernel slicing and co-scheduling. In *Case II*, we evaluate the impact of different scale factors. Particularly, we evaluate the effectiveness of both kernel co-scheduling and the overlapping on PCI-e data transfer and the kernel execution. The GPU memory may not be large enough to accommodate all working sets and the PCI-e data transfers are more frequent than Case I.

To assess the impact of kernel scheduling under different mixes of kernels, we create four groups of kernels namely CI, MI, MIX and ALL (as shown in Table 5.5). CI represents the computation-intensive workloads, whereas MI represents workloads with intensive memory accesses. MIX and ALL include a mix of CI and MI kernels. ALL has all the eight kernels. In each workload, we assume the application arrival conforms to Poisson distribution.

Table 5.5: Workload configurations.

Workload	Applications
CI	BS, MM, TEA, MRIQ
MI	PC, SPMV, ST, SAD
MIX	PC, BS, TEA, SAD
ALL	PC, SPMV, ST, BS, MM, TE, MRIQ, SAD

The parameter λ in the Poisson distribution affects the weight of the application in the workload. For simplicity, we assume that all application has the same λ . We also assume λ is sufficiently large so that at least two kernels are pending for execution at any time for a high utilization of the GPU. In the experiments, two instances of each kernel are submitted per second on average ($\lambda = 2$ instances per second). Thus, we fix the total number of instances of kernel executions for all scheduling algorithms and compare their total execution times (from submitting the first kernel till the completing the last kernel).

Comparisons. To evaluate the effectiveness of kernel scheduling in Kernelet, we have implemented the following scheduling techniques:

- **Kernel Consolidation (Base):** the kernel consolidation approach of concurrent kernel execution [27].
- **Oracle (OPT):** OPT uses the same scheduling algorithm as Kernelet, except that it pre-executes all possible slice ratios for all combinations to obtain the CP and then determines the best slice ratio and kernel combination. In another word, OPT is an offline algorithm and provides the best throughput for the greedy scheduling algorithm.
- **Monte Carlo-based co-schedule (MC):** We adopt the Monte Carlo method to generate the performance distribution of random co-scheduling plans in the solution space. In each Monte Carlo simulation, we randomly pick kernel pairs for co-scheduling with random slice ratios. We denote the result of MC to be $MC(s)$, where s is the number of Monte Carlo runs.

Figure 5.8: Sliced execution overhead with varying slice size on both C2050 and GTX680.

5.6.2 Results on Kernel Slicing

We first evaluate the overhead of sliced execution, which is defined as $\frac{T_s}{T_{ns}} - 1$, where T_s and T_{ns} are the execution time with and without slicing, respectively. Figure 5.8 shows the overhead for executing individual kernels with varying slice sizes on C2050 and GTX680. Slice sizes are set to multiples of the number of SMs on the GPU and ranges from $|SM|$ to the maximum number under the occupancy limit. Overall, as the slice size increases, the slicing overhead decreases. However, we observe quite different performance behaviors on C2050 and GTX680, due to their architectural differences. On C2050, when the size is small, the slicing overhead is very high (up to 67% for SAD). When the slice is larger than or equal to 42 (three thread blocks per SM), the overhead is ignorable for most kernels. The overhead on GTX680 is much smaller than on C2050. Almost all slice sizes lead to overhead less than 2% on GTX680. On both architectures, the ignorable overhead of kernel slicing allows us to exploit kernel slicing for co-scheduling slices from different kernels with little additional cost.

5.6.3 Results on Model Prediction

We evaluate the accuracy of our performance model in different aspects, including the estimation of *IPC*s for single kernel and concurrent kernel executions, and *CP* prediction for concurrent kernel executions.

Single Kernel Performance Prediction. Figure 5.9 compares the measured and estimated *IPC* values for the eight benchmark applications on C2050 and GTX680. Most

Figure 5.9: Comparison between predicted and measured single kernel execution IPCs on two GPUs.

results are within the 20% difference scope. We further define the absolute error to be $|e - e'|$, where e and e' are the measured and estimated *IPC* values, respectively. The average absolute error for the eight benchmark applications is 0.08 and 0.21 on C2050 and GTX680, respectively. Note that the prediction error of GTX680 is larger than that of C2050. This is introduced by the adaptation of virtual SM. Nevertheless, our probabilistic model has achieved a reasonable accuracy in estimating the performance of single-kernel executions on both GPUs.

Concurrent Kernel Performance Prediction. For the eight benchmark applications, we run every possible combination of kernel pairs and measure the *IPC* for each combination. Figure 5.10 compares the measured and predicted *IPCs* with the suitable slice ratio given by our model. For all the figures comparing the predicted and measured *IPC* and *CP* in this thesis, we also show the two lines ($y = x \pm 0.2$ for C2050 and $y = x \pm 1.6$ for GTX680) to highlight the scope where difference between prediction and measurement is within $\pm 20\%$ of the peak *IPC*. Note, the theoretical *IPCs* for C2050 and GTX680 are one and eight, respectively. If the result falls in this scope, we consider the estimation well captures the trend of the measurement. Figure 5.10 shows most of the predictions are with this scope. Figure 5.11 compares the measured and predicted *IPCs* with fixed *ratio* = 1 : 1. For different kernel combinations and slicing ratios, our model is able to well capture the trend of concurrent executions for both dynamic and static slice

Figure 5.10: Comparison between predicted and measured concurrent kernel execution IPCs on two GPUs with optimal slice ratio.

Figure 5.11: Comparison between predicted and measured concurrent kernel execution IPCs on two GPUs with fixed one-to-one slice ratio.

ratios.

Model Optimizations. We evaluate the impact of incorporating coalesced/uncoalesced memory accesses and the number of warp schedulers on the GPU. Only two applications (PC and SPMV) in our benchmark have uncoalesced memory accesses. We conduct the single kernel execution prediction experiments by (wrongly) assuming those two kernels with coalesced memory accesses only. The results are shown in Figure 5.12. Without considering uncoalesced access, the predicted IPC values are much larger than measurements since the assumption of coalesced access only underestimates the memory contention effects.

Figure 5.13 shows the results of concurrent execution IPC prediction on GTX680 with-

Figure 5.12: Comparison between predicted and measured concurrent kernel execution IPCs with/without considering uncoalesced access on C2050.

Figure 5.13: Comparison between predicted and measured concurrent kernel execution IPCs without considering multiple warp schedulers on GTX680.

out considering the multiple warp schedulers. The estimation without considering the number of warp schedulers severely underestimates the IPC on GTX680, in comparison with the results in Figure 5.11b and Figure 5.10b.

CP Prediction. We evaluate the accuracy of *CP* prediction. Figure 5.14 shows the comparison between measured and predicted *CP* on C2050. We observe similar results on GTX680. The prediction is close to the measurement. With accurate prediction on *IPC*, the *CP* difference between prediction and measurement is small. The results are sufficiently good to guide the scheduling decision.

Figure 5.14: Comparison between predicted and measured CP on C2050.

5.6.4 Results on Kernel Scheduling

In this section, we evaluate the effectiveness of our kernel scheduling algorithm by comparing with Base and OPT. To simulate the continuous kernel submission process, we initiate 1000 instances for each kernel and submit them for execution according to Poisson distributions. Different scheduling algorithms are applied and the total kernel execution time is reported.

Results for *Case I*. We first study Case I, where PCI-e data transfer overhead is small. Figure 5.15 shows the total execution time of executing those kernels on C2050 and GTX680. We also include the performance of sequential execution for reference. On all the four workloads with different memory and computation characteristics, Kernelet outperforms Base (with the improvement 5–31% for C2050 and 7–23% for GTX680). Kernelet achieves similar performance to OPT (with the difference 1–3% for C2050 and 4–15% for GTX680). The performance improvement of Kernelet over Base is more significant on MIX and ALL, because Kernelet have more chances to select kernel pairs with complementary resource usage. Base only slightly improves the sequential execution since each individual kernel has sufficient thread blocks to entirely occupy the GPU.

We also study the execution time distribution of the scheduling candidate space. Figure 5.16 shows the cumulative distribution function (CDF) of the execution time of the MC(1000) (1000 Monte Carlo simulations) on ALL workload. None of the random schedules is better than Kernelet. It demonstrates that random co-schedules hurt the performance

Figure 5.15: Comparison between different scheduling methods on both C2050 and GTX680.

Figure 5.16: CDF (cumulative distribution function) of execution time of MC(1000).

with a high probability due to the huge space of scheduling plans.

We finally study the number of kernel combinations pruned during kernel scheduling with varying pruning parameters. Table 5.6 shows the results on C2050. For simplicity, we

Table 5.6: Number of kernels pruned with varying α_p and α_m on C2050.

$\alpha_m \backslash \alpha_p$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.015	0	0	2	2	3	4	4	4	4	4
0.03	0	2	5	6	7	8	9	9	9	9
0.045	0	3	7	8	9	10	11	11	11	11
0.06	1	4	8	9	12	13	15	15	15	15
0.075	1	4	8	9	12	13	15	15	15	15
0.09	1	4	8	9	12	13	15	15	16	16
0.105	1	4	9	10	14	15	18	18	20	20
0.12	2	6	11	12	17	18	21	22	24	24
0.135	2	6	11	12	17	19	22	23	25	26
0.15	2	6	11	13	18	20	23	24	27	28

Figure 5.17: Execution time comparison of different Kernelet variants on C2050 ($sf = 1$).

study the pruning results on all the co-schedules from the eight kernels in our study. There are 28 (C_8^2) co-schedules in total. Increasing α_p and α_m leads to more kernel combinations being pruned. Similar pruning result is observed for GTX680. Varying those two parameters can affect the pruning power and also the optimization opportunities. Thus, we choose the default values for α_p and α_m as 0.4 and 0.1 on C2050, and 0.4 and 0.105 on GTX680, respectively, as a tradeoff between pruning power and optimization opportunities.

Results for Case II. In Case II, we can evaluate the impact of memory transfer/computation overlapping and kernel co-scheduling. Figure 5.17 shows the total execution time including all memory transfers and kernel executions comparison with different algorithm variants “ O_1, O_2 ” for $sf = 1$ on C2050. We observed similar results on GTX680. While the total input data size of all kernels is roughly equal to the GPU memory size, each kernel execution accesses different input data. Thus, PCI-e data transfer is necessary for each kernel execution, unlike Case I. Considering many kernel executions, the total working set size of all kernel executions is much larger than the GPU memory size. In our notation, $O_1 = Y, N$ represents the algorithm with and without memory/computation overlapping, respectively, and $O_2 = Y, N$ represents the algorithm with kernel co-scheduling in Kernelet and the default execution method, respectively. Thus, $Y - Y$ represents our fully optimized Kernelet system. Overall, Kernelet is up to 9.2%, 13.0%, 16.8%, and 17.5% faster than the other three variants on CI, MI, MIX and ALL, respectively. It is the combined improvement of kernel scheduling and memory transfer/computation overlapping.

Figure 5.18: Impact of kernel co-scheduling for different sf on C2050.

Let us analyze the impact of individual techniques in more details. Comparing Y, Y and N, Y , we find that the memory transfer/computation overlapping improves the performance by 9%, thanks to the automated memory management techniques. We divide the data transfer time on PCI-e into two categories, depending on whether the memory transfer is overlapped with GPU kernel execution. In Kernelet, the ratios of the memory transfer that overlaps with GPU kernel execution are 99%, 95%, 97%, and 97% on CI, MI, MIX and ALL for Y, Y , respectively. Most of PCI-e data transfer overhead is hidden by the kernel execution.

Comparing Y, Y and Y, N , we find that kernel co-scheduling achieves a smaller performance improvement compared with those in Case I. The improvement of kernel co-scheduling is 5–12% on the four workloads. Figure 5.18 shows the performance for different algorithm variants for MIX when $sf = 1$ and $sf = 2.5$. When $sf = 2.5$, the total input data size of all kernels exceeds the GPU memory size. Kernelet consistently improves the performance for varying scale factors. We also note that, when $sf = 2.5$, 95% of the PCI-e data transfer is overlapped with kernel execution in MIX for Y, Y .

5.6.5 Results on Medusa Workload Scheduling

In this subsection, we evaluate the throughput improvement of concurrent Medusa applications with the Kernelet back end. We choose the PageRank and BFS primitives of Medusa for the evaluation. In particular, we choose the VERTEX API of PageRank and

Figure 5.19: Sequential and concurrent execution of Medusa PageRank and Medusa BFS on Random.2MV.64ME

Figure 5.20: Sequential and concurrent execution of Medusa PageRank and Medusa BFS on RMAT.2MV.64ME

the EDGE API of BFS-N. PageRank and BFS both usually involve many iterations. In addition, PageRank is more computational intensive than BFS, while BFS involves more uncoalesced accesses. The complementary resource requirement of PageRank and BFS makes them good candidates for co-scheduling.

Figure 5.19a and Figure 5.20a show the execution time per iteration of PageRank and BFS in both sequential and concurrent execution mode with two different input graphs. The input graphs include a Random graph (Random.2MV.64ME) and a RMAT graph (RMAT.2MV.64ME). Both of the two graphs have 2 million vertices and 64 million edges. For concurrent execution, we apply both the Base and Kernelet co-scheduling method. We select the vertex with highest degree as the root for BFS. We instruct PageRank to execute the same number of iterations as BFS.

From Figure 5.19a and Figure 5.20a, we can see that the execution time of PageRank is stable in different iterations. In contrast, execution time of BFS depends on the size of the frontier (in number of vertices) in each iteration (shown in Figure 5.19b and Figure 5.20b).

We have the following observations from this experiment.

- (1) Performance of Kernelet scheduling is always similar to or better than that of Base scheduling in all the iterations. Execution time of Base is similar to the total time of sequential execution. This is consistent with our earlier analysis and experiments.
- (2) In the early and late iterations (Iterations 1, 2, 3 in Figure 5.19 and Iterations 1, 2, 5 in Figure 5.20), execution time of BFS is small because the frontier size is small. In those iterations, Kernelet scheduling yields little performance improvement.
- (3) In iterations where BFS frontier size is large, Kernelet scheduling has prominent performance improvement (up to 29%).

5.7 Summary

In this chapter, we further optimize the Medusa runtime by improving throughput of concurrent kernel executions on the GPU. We formally define the problem and propose a Markov chain-based model to predict performance of concurrent kernel executions. We support transparent concurrent kernel execution on current GPU platforms by developing the kernel slicing technique. With our two-level scheduling and co-scheduling space pruning algorithms, we effectively improve the throughput of concurrent kernel execution by exploiting the complementary resource requirements of concurrent kernels and overlapping kernel execution with data transfer. Experimental results show that our performance model accurately predicts the concurrent execution performance, and the throughput of concurrent kernel execution is improved by 5-31% and 7-23% on C2050 and GTX680, respectively.

Chapter 6

Case Study: Medusa-Based Information Propagation Simulations

In this chapter, we present a case study on applying Medusa to a real research project. We collaborate with our colleagues to study the feasibility of using GPUs to accelerate information propagation simulations over social networks. Information propagation simulations provide a flexible and valuable method to study the behaviors over complex social networks. With the increased computing resource usage in large-scale network based simulations, it is therefore attractive to apply GPUs to improve the simulation performance. In this project, we have developed a set of novel simulation strategies which take advantages of Medusa to harness the power of GPUs.

In the rest of this chapter, we first introduce the background of this case study. Next, we present the main techniques that we developed for information propagation simulation on the GPU and some findings from the simulation results. At last, we discuss the users' experiences in adopting Medusa to their applications.

6.1 Background

A social network refers to a specific social structure which consists of entities (such as individuals or organizations) and the patterns/implications of the relationships between these entities [142]. Social network approaches are useful for modeling and explaining many social phenomena. A social network can be employed to model the structure of a social

group, or can be used to explain how this social structure influences the behaviors among the entities over time. In conventional social network studies using mathematical models, the individual is essentially ignored [143, 144]. However, many different types of individual relationships or attribute combinations usually form a complex network structure. It is therefore necessary to apply advanced modeling and simulation methods to analyze these behaviors. Examples include the spread of infectious disease over complex social networks.

Large-scale network-based simulations involving information propagation often require a large amount of computing resources. It is therefore necessary to develop performance-oriented simulation techniques and to map those optimized simulation methods onto high performance computing (HPC) platforms such as GPUs. For complex networks, the network structures are highly irregular due to the complicated relationships among the nodes. Such irregularity of the data structure may exhibit very poor memory access locality. Thus, the underlying network layout in the memory needs to be carefully designed and optimized.

The irregularity of the network data structure poses great challenges on efficient GPU acceleration. We also note that the memory space of a single GPU device is not enough to contain very large networks. The space limit of a single GPU memory may become a limitation that constrains the scalability of GPU simulations. It is therefore necessary to design a simulation paradigm on multiple GPU devices for large-scale network simulation. However, the multi-GPU system introduces synchronization and communication overhead that may reduce the parallel performance. It is therefore important to investigate techniques and optimizations to alleviate these overheads of the multi-GPU system.

When the research project starts, we carefully considered two options to develop the simulation system. First, we can directly implement the project using the popular GPGPU programming platforms like CUDA and OpenCL. Second, we can adopt Medusa and implement simulations with Medusa APIs. We choose using Medusa to implement the experimental system based on the following considerations:

- While implementing the project from scratch seems the most flexible method, developing network-based information propagation is still a tough job as we introduced in

Chapter 1. What's worse, the core members of the research team has little GPU programming experience and the learning curve can be long. Medusa requires no GPU programming knowledge. With Medusa, we can save significant amount of time in the knowledge acquiring phase of this project. All members of this project are already familiar with C/C++ programming and can easily use Medusa to build prototypes for fast evaluation of preliminary ideas.

- Medusa provides flexible APIs. The EMV APIs offered by Medusa are sufficient to build all of the algorithms for the simulations in the project.
- The system provided APIs in Medusa can save a lot of engineering work in building the experimental system.

6.2 Techniques and Findings

In this section, we describe the main techniques we developed for this project and some findings from the simulation results.

6.2.1 Models of Information Propagation

The simulation of information propagation is to investigate the interactive behaviors between *Active* nodes and *Inactive* nodes within a given network. Currently, the Independent Cascade Model (ICM) [145] and the Linear Threshold Model (LTM) [146] are widely used in studying the behaviors of information propagation over networks. In the ICM, we define an initial set of active nodes A_0 at step 0. The information propagation unfolds in discrete time steps: at step t , the newly active node v_i has a single chance to activate its inactive neighbor u with an independent probability $p_{(v_i, u)} \in [0, 1]$. If v_i succeeds in activating u , u will transit its status from inactive to active at step $t+1$ and remain active afterwards [145]. Such a process continues until no more new activations are made in a step. In the LTM, each node on the network is assigned with a random threshold $T_u \in [0, 1]$. At step t , each inactive node is influenced by its active neighbors (a set A_t , where A_t is $\emptyset$ if no active

neighbor exists). The influence weight between the active node v_i and the inactive node u can be expressed as a probability $b_{(v_i, u)}$. Thus, node u 's influence weight from its active neighbors can be calculated and represented by $\sum_{i=1}^l b(v_i, u)$, where l denotes the number of active neighbors and v_i is the i th active neighbor of u [146]. If the transition probability $\sum_{i=1}^l b(v_i, u)$ is larger than the predefined threshold value, u 's status will transit from inactive to active at step $t + 1$ and remain afterwards.

6.2.2 Algorithm Optimizations

According to the model definitions [145, 146], we first introduce two types of simulation algorithms named as C-Loop and T-Loop.

- C-Loop: Starting from the active nodes in the network, each active node goes through its neighbors at each simulation step and tests whether it can propagate the information to the inactive neighbors with a specific probability. If the inactive nodes receive the information, they will change status to be active at the next step.
- T-Loop: In contrast to the C-Loop, the T-Loop starts from the inactive nodes and traces the neighbors' status at each step. The inactive node can be activated at the next step by any active neighbor if the transmission probability is satisfied.

Both C-Loop and T-Loop can be easily implemented using the *ELIST* API provided by Medusa. Since the process of neighbors in C-Loop and T-Loop imposes no edge order constraint, we also propose to use *EDGE* API to implement similar functionalities as C-Loop and T-Loop. This is inspired by the analysis from Medusa that *EDGE* API alleviates the load imbalance problem of *ELIST* API and exhibits better data access performance. We name the *EDGE* API based algorithm as E-Loop.

- E-Loop: Different from the vertex-oriented approach (C-Loop and T-Loop), the E-Loop starts from each edge element and checks the status of the connected pair of nodes. If the two connected nodes have different status such as Active-Inactive, the information can be propagated from active to inactive with the given probability.

Figure 6.1: Execution time per simulation step.

Figure 6.1 shows the execution time per simulation step on C2050. The dataset is a random graph generated by GTGraph [35] with 1 million vertices and 8 million edges. Due to different memory access patterns, the above three algorithms can exhibit different costs in each step of the simulation. For example, the cost of a C-Loop step is initially small due to the small number of active vertices. As the number of active vertices increases, the cost of a C-Loop step increases. The cost of T-Loop iterations is opposite to that of C-Loop. In contrast, the cost of a E-Loop stays stable in different iterations. Compared to the CPU serial simulation performance, the C2050 GPU based simulation shows 12.5x, 13.1x, 15.6x speedup with CLoop, T-Loop, and E-Loop, respectively [40].

Medusa allows fast implementation of these three algorithms. Considering the different characteristics of the algorithms, we propose an adaptive technique to further increase the performance of the simulation. Our paper [40] presents more details on this adaptation. In summary, the adaptation chooses the C-Loop at the beginning of the simulation, and switches to the T-Loop at a proper time step. The swapping time step is determined by computational complexity analysis.

6.2.3 Simulation on Multiple GPUs

Using multiple GPUs for the simulation is a fast way to increase the memory and computation capacities of the system. We use the default graph partitioning method provided

by Medusa to partition the graph. Medusa automatically builds the replicas for each partition and handles the update of the replicas. Thus, with Medusa, the network-based information propagation is enable on multiple GPUs with minimal effort. Our paper [41] comprehensively evaluates the multi-GPU simulation performance on large scalable networks. Experimental results show that the number of simulation steps, computation time, synchronization time, and data transfer time affect the overall simulation performance. The simulation performance is improved by 2.7 times on four GPUs compared with on one GPU.

6.2.4 Findings in Viral Advertisement

We apply our proposed simulation techniques to study the propagation behaviors during the process of viral advertisement diffusion. We examine the spread of viral advertisements over a realistic social network using multiple GPUs. We also investigate the effect of different initial nodes selection policies in maximizing the performance of advertisement diffusion.

We have published this study in our paper [42] and here we briefly discuss our findings. According to our simulation studies on viral advertisement diffusion, we observe that the number of initial selected nodes is important to the diffusion behaviors. However, we also note that the initial selection policy plays a limited role in the final result of viral advertisement diffusion. Therefore, we propose to improve viral advertising strategies by using mass marketing first to increase the willingness of accepting a product and apply viral marketing to facilitate the maximization of advertisement diffusion.

6.3 Experience on Using Medusa

In this section, we summarize the user experience in using Medusa as the base infrastructure of building network based information propagation simulations.

Learning Curve. Medusa requires no knowledge on GPU programming. Users just need to have C/C++ programming knowledge. To learn Medusa, users need to understand

the Medusa programming model and to get familiar with the Medusa APIs. Thanks to the simplicity of the EMV model and the small number of APIs, users usually can start building simple Medusa-based applications in a few hours. With Medusa, our collaborator took only three hours to write the first simple graph processing example. As a matter of fact, CUDA requires a much longer learning curve, particularly on the CUDA programming model and tedious performance optimizations. It usually take several weeks or even months to master CUDA.

Advantages of Medusa. Throughout the implementation of this project, Medusa greatly simplifies our work on utilizing GPUs for information propagation simulation. First, the programming model of Medusa fits well with the real information propagation process. Information propagation over social networks usually happens among neighboring vertices. Similarly, Medusa allows users to formulate their algorithm with the granularity of individual vertices or edges. Second, the individual and collective APIs of Medusa allow us develop different approaches (i.e., vertex-oriented and edge-oriented) for the simulation, leading to more opportunities for improving the overall performance of the simulation. Third, despite the fact that Medusa provides an abstract data model and hides the GPU related implementation details from users, experienced users can still easily access the underlying data structures. User can customize Medusa based on their specific requirements, such as integrating Medusa with the Curand library [147]. Forth, a Medusa program can transparently run on multiple GPUs. This feature of Medusa allows the user to enjoy the benefits of multiple GPUs (more memory space and computation power) with little extra implementation effort.

Expectations on Future Improvements. This project has also triggered some new ideas on further improvement to Medusa. First, Medusa only supports processing static graphs, while some information propagation strategies may need to change the links between nodes at runtime. It is our future work to investigate the feasibility of supporting dynamic graph processing in Medusa. Second, Medusa does not support dynamically changing the number of GPUs used for simulation at runtime. More GPUs do not always

mean better performance due to the inter-GPU communication cost. For example, for C-Loop algorithm it may be more beneficial to run the simulation on one GPU in the early steps, and to increase the number of GPUs used as the number of active nodes becomes larger.

Chapter 7

Conclusion and Future Work

In this chapter, we conclude this thesis and outline our future work.

7.1 Conclusions

Graph is the de facto data structure in many classic and emerging applications, and the efficiency of graph processing algorithms is a must for the overall system performance. On the other hand, designed as many-core co-processors, GPUs have been adopted in accelerating various applications and shown extraordinary performance improvement. However, the irregularity of the graph data structure and the complexity of graph processing algorithms make efficient parallel graph processing still a challenging research problem. The fact that GPUs are more and more being used as shared devices in clusters and cloud environments introduces new challenges as well as opportunities in improving throughput of parallel graph processing on the GPU.

In this thesis, we address the efficiency and programmability of GPU-based parallel graph processing by developing a general graph processing system on the GPU named Medusa. Medusa embraces a programming framework to hide the programming complexity of implementing parallel graph computation tasks for GPUs. Developers only need to write sequential programs to implement a small set of APIs. Medusa can automatically scale to multiple GPUs in the same machine. On an NVIDIA Tesla C2050 GPU, Medusa-based implementations are 5.5 times on average faster than the parallel MTGL based implementations on two Intel six-core CPUs. Then, we propose an optimized runtime

system Kernelet to improve the throughput of concurrent kernel executions in Medusa. Kernelet implements transparent memory management and data transfer techniques on the GPU, creates more sharing opportunities with kernel slicing, and uses a probabilistic performance model to capture the non-deterministic performance features of multiple-kernel executions. We evaluate Kernelet on two NVIDIA GPUs, Tesla C2050 and GTX680, with Fermi and Kepler architectures, respectively. Our experiments demonstrate the accuracy of our performance model, and the effectiveness of Kernelet by improving the concurrent kernel executions by 5-31% and 7-23% on C2050 and GTX680, respectively. Experiments show that Kernelet improves the performance of concurrent Medusa applications by 29%.

7.2 Future Work

Through the designing and implementing Medusa, we have identified the following open problems as our future work.

Dynamic Graph Processing. Real world graphs, such as the social networks, are usually evolving in different speeds. Also, some graph algorithms require changing the graph structure during runtime such as the Delaunay mesh refinement algorithm. However, there has been little work on GPU-based dynamic graph processing. Narse et al. [19] presented implementation of five graph algorithms which morph structure of the input graph in different ways. They provided a number of basic methods for adding and deleting subgraphs and required users to make their choices based on the application characteristics and the scale of the problem. Their methods are application dependent and require non-trivial programming efforts to implement. With the idea of simplifying dynamic graph processing on the GPU, we consider offering dynamic graph processing support in Medusa. Advancements in GPU architectures, such as the improved memory bandwidth and reduced cost of atomic operations, have opened new opportunities for support dynamic graph processing.

External Memory Graph Processing. Existing studies on GPU graph processing mainly deal with in-memory processing of graphs [4, 5, 17–19, 37]. The size of the largest input graph is limited by the main memory capacity. External memory processing have

been widely adopted by many algorithms, including graph processing algorithms [57], for processing data larger than main memory. Introducing external memory algorithms to Medusa is challenging due to the larger gap between GPU memory bandwidth and disk bandwidth, as well as the overhead of PCI-e data transfer. One promising direction is to study data compression algorithms to reduce the overhead of data transfer for external GPU graph processing.

Distributed Graph Processing. GPUs have already been deployed in clusters, clouds and even super computers. Extending Medusa to large-scale systems could significantly increase the graph size and accelerate the processing speed with GPUs. One of the distinct issues is that the great computational power of GPUs further widens the speed gap between network and processors. Communication operations need to be carefully examined to ensure scalability and performance. We propose to extend Medusa to the distributed environment and optimize the communication operations based on the characteristics of graph applications. We have published some preliminary results in this direction [31].

Simplified Graph Processing on New Architectures. New architectures are becoming available to both PCs and servers. The Accelerated Processing Unit (APU) [148] plays the same role as the conventional CPU, with an integrated GPU on chip for GPGPU applications. By eliminating the PCI-e bus on the discrete CPU/GPU architecture, the APU has lower communication overhead between the CPU and GPU cores. That opens up new opportunities for fine-grained co-processing [149]. The Intel MIC (Many Integrated Core Architecture, or Intel Xeon Phi) [150] processor is also a many-core processor, with the same X86 instruction set as the CPU. With the support of OpenCL, it is trivial to port Medusa to these new platforms. More importantly, the architectural differences of those new platforms pose new challenges and opportunities in performance optimization or even architectural redesign of Medusa.

Publications

Conference

- [1] **J. Zhong**, B. He “Towards GPU-Accelerated Large-Scale Graph Processing in the Cloud.” *5th International Conference on Cloud Computing Technology and Science*, accepted, 2013.
- [2] **J. Zhong**, B. He “Parallel Graph Processing on Graphics Processors Made Easy.” In *Proceedings of the VLDB Endowment*, Volume 6, No. 12, pp. 1270-1273, 2013. (demo paper)
- [3] J. Jin, S. J. Turner, B. Lee, **J. Zhong**, B. He, “Simulation of Information Propagation over Complex Networks: Performance Studies on Multi-GPU.” *The 17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, accepted, 2013.
- [4] J. Jin, S. J. Turner, B. Lee, **J. Zhong**, B. He, “Simulation Studies of Viral Advertisement Diffusion on Multi-GPU.” *Winter Simulation Conference*, accepted, 2013.
- [5] **J. Zhong**, B. He “An Overview of Medusa: Simplified Graph Processing on GPUs.” In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 283-284, 2012. (poster)
- [6] Y. Gong, B. He, **J. Zhong**, “An Overview of CMPI: Network Performance Aware MPI in the Cloud”. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 297-298, 2012. (poster)

- [7] J. Jin, S. J. Turner, B. Lee, **J. Zhong**, B. He, “HPC Simulations of Information Propagation over Social Networks.” *Procedia Computer Science*, 9 (2012): pp. 292-301.
- [8] **J. Zhong**, B. He, “GViewer: GPU-Accelerated Graph Visualization and Mining.” *Social Informatics, Lecture Notes in Computer Science*, Volume 6984/2011, pp. 304-307. (poster)
- [9] **J. Zhong**, B. He, “GPU-Assisted Buffer Management.” In *Proceedings of the International Conference on Computational Science*, pp. 362-371, 2011.

Journal

- [1] **J. Zhong**, B. He, “Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling,” *IEEE Transactions on Parallel and Distributed Systems*, accepted, 2013.
- [2] **J. Zhong**, B. He, “Medusa: simplified graph processing on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, PrePrint, 2013.
- [3] Y. Gong, B. He, **J. Zhong**, “Network Performance Aware MPI Collective Communication Operations in the Cloud,” *IEEE Transactions on Parallel and Distributed Systems*, PrePrint, 2013.

Appendix A

CUDA Implementation of Vector Addition

Vector addition takes two vectors (stored as two arrays) as input, computes the addition of the two vectors and output the result to a new vector. Figure A.1 - A.5 shows the CUDA implementation of vector addition from the CUDA Toolkit [151]. We have highlighted the GPU related code and comments in bold.

```
/**
 * Copyright 1993-2012 NVIDIA Corporation. All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/**
 * Vector addition: C = A + B.
 *
 * This sample is a very basic sample that implements element by element
 * vector addition. It is the same as the sample illustrating Chapter 2
 * of the programming guide with some additions like error checking.
 */

#include <stdio.h>
#include <cuda_runtime.h>
```

Figure A.1: CUDA implementation of vector addition (Part I).

```
/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C. The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}

/**
 * The main program
 */
int main(void)
{
    // Error code to check return values for CUDA calls
    cudaError_t err = cudaSuccess;

    // Print the vector length to be used, and compute its size
    int numElements = 50000;
    size_t size = numElements * sizeof(float);
    printf("[Vector addition of %d elements]\n", numElements);

    // Allocate the host input vector A
    float *h_A = (float *)malloc(size);

    // Allocate the host input vector B
    float *h_B = (float *)malloc(size);

    // Allocate the host output vector C
    float *h_C = (float *)malloc(size);

    // Verify that allocations succeeded
    if (h_A == NULL || h_B == NULL || h_C == NULL)
    {
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
    }
}
```

Figure A.2: CUDA implementation of vector addition (Part II).

```
// Initialize the host input vectors
for (int i = 0; i < numElements; ++i)
{
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
}

// Allocate the device input vector A
float *d_A = NULL;
err = cudaMalloc((void **)&d_A, size);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Allocate the device input vector B
float *d_B = NULL;
err = cudaMalloc((void **)&d_B, size);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Allocate the device output vector C
float *d_C = NULL;
err = cudaMalloc((void **)&d_C, size);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Copy the host input vectors A and B in host memory to the device input vectors in
// device memory
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
```

Figure A.3: CUDA implementation of vector addition (Part III).

```
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector A from host to device (error code %s)!\n",
        cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector B from host to device (error code %s)!\n",
        cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Configure and launch the CUDA kernel
int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid, threadsPerBlock);
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
err = cudaGetLastError();

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n",
        cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Copy the device result vector in device memory to the host result vector
// in host memory.
printf("Copy output data from the CUDA device to the host memory\n");
err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector C from device to host (error code %s)!\n",
        cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Verify that the result vector is correct
for (int i = 0; i < numElements; ++i)
```

Figure A.4: CUDA implementation of vector addition (Part IV).

```
{
    if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
    {
        fprintf(stderr, "Result verification failed at element %d!\n", i);
        exit(EXIT_FAILURE);
    }
}

// Free device global memory
err = cudaFree(d_A);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to free device vector A (error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
err = cudaFree(d_B);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to free device vector B (error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
err = cudaFree(d_C);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to free device vector C (error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

// Free host memory
free(h_A);
free(h_B);
free(h_C);

// Reset the device and exit
err = cudaDeviceReset();

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to deinitialize the device! error=%s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

printf("Done\n");
return 0;
}
```

Figure A.5: CUDA implementation of vector addition (Part V).

Bibliography

- [1] NVIDIA, *NVIDIAs next generation CUDA compute architecture: Fermi*, 2009.
- [2] P. Harish and P. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” in *High performance computing–HiPC 2007*, pp. 197–208, Springer, 2007.
- [3] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, “Software and algorithms for graph queries on multithreaded architectures,” in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–14, IEEE, 2007.
- [4] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating CUDA graph algorithms at maximum warp,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pp. 267–276, ACM, 2011.
- [5] D. Merrill, M. Garland, and A. Grimshaw, “Scalable GPU graph traversal,” in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP ’12, pp. 117–128, 2012.
- [6] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web,” technical report, 1999.
- [7] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” in *Computer graphics forum*, vol. 26, pp. 80–113, Wiley Online Library, 2007.
- [8] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, “Relational query coprocessing on graphics processors,” *ACM Transactions on Database Systems*, vol. 34, pp. 21:1–21:39, Dec. 2009.
- [9] B. He and J. X. Yu, “High-throughput transaction executions on graphics processors,” *Proceedings of the VLDB Endowment*, vol. 4, pp. 314–325, Feb. 2011.
- [10] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, “Frequent itemset mining on graphics processors,” in *Proceedings of the fifth international workshop on data management on new hardware*, pp. 34–42, ACM, 2009.

- [11] Y. Li, K. Zhao, X. Chu, and J. Liu, “Speeding up k-means algorithm by GPUs,” *Journal of Computer and System Sciences*, vol. 79, no. 2, pp. 216–229, 2013.
- [12] S. Han, K. Jang, K. Park, and S. Moon, “PacketShader: a GPU-accelerated software router,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 195–206, 2010.
- [13] X. Chu, K. Zhao, and M. Wang, “Massively parallel network coding on GPUs,” in *Performance, Computing and Communications Conference, 2008. IPCCC 2008. IEEE International*, pp. 144–151, IEEE, 2008.
- [14] G. He, H. Feng, C. Li, and H. Chen, “Parallel SimRank computation on large graphs with iterative aggregation,” in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 543–552, ACM, 2010.
- [15] V. Vineet and P. Narayanan, “CUDA cuts: Fast graph cuts on the GPU,” in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW’08. IEEE Computer Society Conference on*, pp. 1–8, IEEE, 2008.
- [16] G. J. Katz and J. T. Kider, Jr, “All-pairs shortest-paths for large graphs on the GPU,” in *the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pp. 47–55, 2008.
- [17] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using cuda,” in *Proceedings of the 14th international conference on High performance computing*, HiPC’07, (Berlin, Heidelberg), pp. 197–208, Springer-Verlag, 2007.
- [18] S. Hong, T. Oguntebi, and K. Olukotun, “Efficient parallel graph exploration on multi-core CPU and GPU,” in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 78–88, IEEE, 2011.
- [19] R. Nasre, M. Burtcher, and K. Pingali, “Morph algorithms on GPUs,” in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP ’13, pp. 147–156, 2013.
- [20] X. Mei, L. S. Yung, K. Zhao, and X. Chu, “A measurement study of GPU DVFS on energy conservation,” in *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower ’13, (New York, NY, USA), pp. 10:1–10:5, ACM, 2013.
- [21] M. Garland and D. B. Kirk, “Understanding throughput-oriented architectures,” *Communications of the ACM*, vol. 53, no. 11, pp. 58–66, 2010.

- [22] S. Ryoo, C. Rodrigues, S. Stone, S. Bagsorkhi, S.-Z. Ueng, and W.-m. W. Hwu, “Program optimization study on a 128-core GPU,” in *The First Workshop on General Purpose Processing on Graphics Processing Units*, pp. 30–39, Citeseer, 2007.
- [23] S. Ryoo, C. I. Rodrigues, S. S. Bagsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, “Optimization principles and application performance evaluation of a multi-threaded GPU using CUDA,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73–82, ACM, 2008.
- [24] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, and A. W. Toga, “CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms,” *Computer methods and programs in biomedicine*, vol. 106, no. 3, pp. 175–187, 2012.
- [25] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, “Relational query coprocessing on graphics processors,” *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, p. 21, 2009.
- [26] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, “High performance discrete fourier transforms on graphics processors,” in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 2, IEEE Press, 2008.
- [27] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, “Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework,” in *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC ’11, pp. 217–228, 2011.
- [28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD ’10, (New York, NY, USA), pp. 135–146, ACM, 2010.
- [29] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [30] 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*, pp. 260–269, ACM, 2008.
- [31] J. Zhong and B. He, “Towards gpu-accelerated large-scale graph processing in the cloud,” in *5th International Conference on Cloud Computing Technology and Science*, no. Accepted, 2013.

- [32] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, (New York, NY, USA), pp. 152–163, ACM, 2009.
- [33] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, “A performance analysis framework for identifying potential benefits in GPGPU applications,” in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, (New York, NY, USA), pp. 11–22, ACM, 2012.
- [34] Stanford Large Network Dataset Collections, <http://snap.stanford.edu/data/index.html>, accessed on Mar 13th, 2014.
- [35] GTGraph Generator, <http://www.cse.psu.edu/~madduri/software/GTgraph/>, accessed on Mar 13th, 2014.
- [36] J. Zhong and B. He, “An overview of medusa: simplified graph processing on GPUs,” in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '12, (New York, NY, USA), pp. 283–284, ACM, 2012.
- [37] J. Zhong and B. He, “Medusa: Simplified graph processing on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, 2013.
- [38] J. Zhong and B. He, “Parallel graph processing on graphics processors made easy,” *Proc. VLDB Endow.*, vol. 6, pp. 1270–1273, Aug. 2013.
- [39] J. Zhong and B. He, “Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, p. 1, 2013.
- [40] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He, “HPC simulations of information propagation over social networks,” *Procedia Computer Science*, vol. 9, pp. 292–301, 2012.
- [41] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He, “Simulation of information propagation over complex networks: Performance studies on multi-GPU,” in *Distributed Simulation and Real Time Applications (DS-RT), 2013 IEEE 17th International Symposium on*, 2013.
- [42] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He, “Simulation studies of viral advertisement diffusion on multi-GPU,” in *Winter Simulation Conference (WSC), 2013*, 2013.

- [43] H. Chen, W. Chung, Y. Qin, M. Chau, J. J. Xu, G. Wang, R. Zheng, and H. Atabakhsh, “Crime data mining: an overview and case studies,” in *Proceedings of the 2003 annual national conference on Digital government research*, pp. 1–5, Digital Government Society of North America, 2003.
- [44] P. Domingos, “Mining social networks for viral marketing,” *IEEE Intelligent Systems*, vol. 20, no. 1, pp. 80–82, 2005.
- [45] M. Jamali and M. Ester, “Mining social networks for recommendation,” *Tutorial of ICDM*, vol. 11, 2011.
- [46] G. A. FOWLER, “Facebook tops billion-user mark,” *The Wall Street Journal (New York)*, October 4 2012.
- [47] 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.
- [48] J. Kunegis, A. Lommatzsch, and C. Bauckhage, “The slashdot zoo: mining a social network with negative edges,” in *Proceedings of the 18th international conference on World wide web*, pp. 741–750, ACM, 2009.
- [49] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, “Mining email social networks,” in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 137–143, ACM, 2006.
- [50] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa, “Efficient processing of graph similarity queries with edit distance constraints,” *The VLDB Journal*, pp. 1–26, 2013.
- [51] E. Dekel, D. Nassimi, and S. Sahni, “Parallel matrix and graph algorithms,” *SIAM Journal on computing*, vol. 10, no. 4, pp. 657–675, 1981.
- [52] F. Y. Chin, J. Lam, and I.-N. Chen, “Efficient parallel algorithms for some graph problems,” *Commun. ACM*, vol. 25, pp. 659–665, Sept. 1982.
- [53] M. J. Quinn and N. Deo, “Parallel graph algorithms,” *ACM Computing Surveys (CSUR)*, vol. 16, no. 3, pp. 319–348, 1984.
- [54] J. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

- [55] D. Bader and K. Madduri, “SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, (Miami, FL), pp. 1–12, IEEE, April 2008.
- [56] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 135–146, ACM, 2013.
- [57] A. Kyrola, G. Blelloch, and C. Guestrin, “GraphChi: large-scale graph computation on just a PC,” in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI’12, (Berkeley, CA, USA), pp. 31–46, USENIX Association, 2012.
- [58] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: Edge-centric graph processing using streaming partitions,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, (New York, NY, USA), pp. 472–488, ACM, 2013.
- [59] D. Gregor and A. Lumsdaine, “The parallel BGL: A generic library for distributed graph computations,” in *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.
- [60] A. Chan and F. Dehne, “CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 117–125, 2003.
- [61] A. Buluç and J. R. Gilbert, “The Combinatorial BLAS: Design, implementation, and applications,” *International Journal of High Performance Computing Applications*, vol. 25, Nov. 2011.
- [62] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “PEGASUS: A peta-scale graph mining system implementation and observations,” in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM ’09, (Washington, DC, USA), pp. 229–238, IEEE Computer Society, 2009.
- [63] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI’04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

- [64] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, “HADI: Fast diameter estimation and mining in massive graphs with Hadoop,” Tech. Rep. CMU-ML-08-117, Carnegie Mellon University, 2008.
- [65] K. Shuang, Y. Yang, B. Cai, and Z. Xiang, “X-RIME: Hadoop-based large-scale social network analysis,” in *3rd IEEE International Conference on Broadband Network and Multimedia Technology*, pp. 901–906, 2010.
- [66] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. Kumar, and M. V. Marathe, “Sahad: Subgraph analysis in massive networks using hadoop,” in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 390–401, IEEE, 2012.
- [67] R. Chen, X. Weng, B. He, and M. Yang, “Large graph processing in the cloud,” in *Proceedings of the 2010 international conference on Management of data*, SIGMOD ’10, (New York, NY, USA), pp. 1123–1126, ACM, 2010.
- [68] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, “GraphLab: A new parallel framework for machine learning,” in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.
- [69] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: a framework for machine learning and data mining in the cloud,” *Proceedings of the VLDB Endowment*, vol. 5, pp. 716–727, Apr. 2012.
- [70] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, “Improving large graph processing on partitioned graphs in the cloud,” in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC ’12, (New York, NY, USA), pp. 3:1–3:13, ACM, 2012.
- [71] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: distributed graph-parallel computation on natural graphs,” in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI’12, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2012.
- [72] R. Chen, J. Shi, Y. Chen, H. Guan, B. Zang, and H. Chen, “Powerlyra: Differentiated graph computation and partitioning on skewed graphs,” technical report ipadstr-2013-001, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, 2013.
- [73] NVIDIA CUDA, <http://developer.nvidia.com/object/cuda.html>, access on Mar 13th, 2014.

- [74] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, “Fast and approximate stream mining of quantiles and frequencies using graphics processors,” in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD ’05, (New York, NY, USA), pp. 611–622, ACM, 2005.
- [75] V. Volkov and J. W. Demmel, “Benchmarking GPUs to tune dense linear algebra,” in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC ’08, (Piscataway, NJ, USA), pp. 31:1–31:11, IEEE Press, 2008.
- [76] NVIDIA, *NVIDIA CUDA C Programming Guide 5.5*, 2013.
- [77] Khronos Working Group, <http://www.khronos.org/>, accessed on Mar 13th, 2014.
- [78] M. Delorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, and A. Dehon, “GraphStep: A system architecture for sparse-graph algorithms,” in *the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 143–151, 2006.
- [79] J. Lin and M. Schatz, “Design patterns for efficient graph algorithms in MapReduce,” in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG ’10, (New York, NY, USA), pp. 78–85, ACM, 2010.
- [80] M. C. Shebanow, “Pervasive massively multithreaded GPU processors,” in *Proceedings of the 6th ACM conference on Computing frontiers*, CF ’09, (New York, NY, USA), pp. 227–227, ACM, 2009.
- [81] S. Sengupta, M. Harris, and M. Garland, “Efficient parallel scan algorithms for GPUs. NVIDIA,” tech. rep., 2008.
- [82] J. Zhong and B. He, “Gviewer: GPU-accelerated graph visualization and mining,” in *Social Informatics*, pp. 304–307, Springer, 2011.
- [83] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, “GPGPU: general-purpose computation on graphics hardware,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.
- [84] T. M. Aamodt, “Architecting graphics processors for non-graphics compute acceleration,” in *Communications, Computers and Signal Processing, 2009, IEEE Pacific Rim Conference on*, pp. 963–968, IEEE, 2009.

- [85] L. Luo, M. Wong, and W.-m. Hwu, “An effective GPU implementation of breadth-first search,” in *Proceedings of the 47th Design Automation Conference, DAC '10*, (New York, NY, USA), pp. 52–55, ACM, 2010.
- [86] X. Yang, S. Parthasarathy, and P. Sadayappan, “Fast sparse matrix-vector multiplication on gpus: implications for graph mining,” *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 231–242, 2011.
- [87] K. Spafford, J. S. Meredith, and J. S. Vetter, “Quantifying NUMA and contention effects in multi-GPU systems,” in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, p. 11, ACM, 2011.
- [88] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, “Scalable framework for mapping streaming applications onto multi-GPU systems,” in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPOPP '12*, (New York, NY, USA), pp. 1–10, ACM, 2012.
- [89] D. Schaa and D. Kaeli, “Exploring the multiple-GPU design space,” in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IPDPS '09*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.
- [90] J. Kim, H. Kim, J. H. Lee, and J. Lee, “Achieving a single compute device image in OpenCL for multiple GPUs,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPOPP '11*, (New York, NY, USA), pp. 277–288, ACM, 2011.
- [91] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, “GPU cluster for high performance computing,” in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing, SC '04*, (Washington, DC, USA), pp. 47–, IEEE Computer Society, 2004.
- [92] Z. Fan, F. Qiu, and A. E. Kaufman, “Zippy: A framework for computation and visualization on a GPU cluster,” *Computer Graphics Forum*, vol. 27, no. 2, pp. 341–350, 2008.
- [93] J. C. Phillips, J. E. Stone, and K. Schulten, “Adapting a message-driven parallel application to GPU-accelerated clusters,” in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08*, pp. 8:1–8:9, IEEE Press, 2008.
- [94] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, “Fast seismic modeling and reverse time migration on a GPU cluster,” *2009 International Conference on High Performance Computing Simulation*, pp. 36–43, 2009.

- [95] Y. Chen, X. Cui, and H. Mei, “Large-scale FFT on GPU clusters,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS ’10, (New York, NY, USA), pp. 315–324, ACM, 2010.
- [96] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, “Green-Marl: a DSL for easy and efficient graph analysis,” in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 349–362, ACM, 2012.
- [97] Metis, <http://glaros.dtc.umn.edu/gkhome/views/metis/>, accessed on Mar 13th, 2014.
- [98] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He, “HPC simulations of information propagation over social networks,” *Procedia Computer Science*, vol. 9, pp. 292 – 301, 2012.
- [99] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, p. 18, ACM, 2009.
- [100] Y. Saad, “Krylov subspace methods on supercomputers,” *SIAM Journal on Scientific Computing (SISC)*, vol. 10, pp. 1200–1232, Nov. 1989.
- [101] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, “MapCG: writing parallel program portable between CPU and GPU,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT ’10, (New York, NY, USA), pp. 217–226, ACM, 2010.
- [102] F. Basseti, K. Davis, and D. J. Quinlan, “Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures,” in *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.
- [103] M. Frigo and V. Strumpen, “Cache oblivious stencil computations,” in *Proceedings of the 19th annual international conference on Supercomputing*, ICS ’05, (New York, NY, USA), pp. 361–366, ACM, 2005.
- [104] “10th DIMACS implementation challenge,” <http://www.cc.gatech.edu/dimacs10/index.shtml>, accessed on Mar 13th, 2014.
- [105] F. van Ham and J. J. van Wijk, “Interactive visualization of small world graphs,” in *Proceedings of the IEEE Symposium on Information Visualization*, (Washington, DC), pp. 199–206, IEEE Computer Society, 2004.

- [106] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, “High-speed switch scheduling for local-area networks,” *ACM Transactions on Computer Systems*, vol. 11, pp. 319–352, November 1993.
- [107] T. M. J. Fruchterman and E. M. Reingold, “Graph drawing by force-directed placement,” *Software: Practice and Experience*, vol. 21, pp. 1129–1164, 1991.
- [108] S. E. Sujay Parekh and H. Levy, “Thread-sensitive scheduling for SMT processors,” technical report 2000-04-02, University of Washington.
- [109] A. Snavely and D. M. Tullsen, “Symbiotic jobscheduling for a simultaneous multithreaded processor,” in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS IX, (New York, NY, USA), pp. 234–244, ACM, 2000.
- [110] A. Snavely, D. M. Tullsen, and G. Voelker, “Symbiotic jobscheduling with priorities for a simultaneous multithreading processor,” in *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 66–76, 2002.
- [111] T. Moseley, D. Grunwald, J. L. Kihm, and D. A. Connors, “Methods for modeling resource contention on simultaneous multithreading processors,” in *Proceedings of the 2005 International Conference on Computer Design*, ICCD ’05, (Washington, DC, USA), pp. 373–380, IEEE Computer Society, 2005.
- [112] D. Tam, R. Azimi, and M. Stumm, “Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors,” in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys ’07, (New York, NY, USA), pp. 47–58, ACM, 2007.
- [113] S. Eyerman and L. Eeckhout, “Probabilistic job symbiosis modeling for SMT processor scheduling,” in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS ’10, pp. 91–102, 2010.
- [114] M. Serrano, W. Yamamoto, R. Wood, and M. Nemirovsky, “A model for performance estimation in a multistreamed superscalar processor,” in *Computer Performance Evaluation Modelling Techniques and Tools*, vol. 794 of *LNCS*, pp. 213–230, 1994.

- [115] M. Serrano, “Performance estimation in a simultaneous multithreading processor,” in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1996. Proceedings of the Fourth International Workshop on*, MASCOTS ’96, pp. 97–101, IEEE, 1996.
- [116] X. Chen and T. Aamodt, “A first-order fine-grained multithreaded throughput model,” in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 329–340, 2009.
- [117] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, “An adaptive performance modeling tool for GPU architectures,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’10, (New York, NY, USA), pp. 105–114, ACM, 2010.
- [118] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, “Fine-grained resource sharing for concurrent GPGPU kernels,” in *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, pp. 10–10, USENIX Association, 2012.
- [119] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, “PTask: operating system abstractions to manage GPUs as compute devices,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP ’11, pp. 233–248, ACM, 2011.
- [120] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, “Gvim: GPU-accelerated virtual machines,” in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt ’09, (New York, NY, USA), pp. 17–24, ACM, 2009.
- [121] T. Li, V. Narayana, E. El-Araby, and T. El-Ghazawi, “GPU resource sharing and virtualization on high performance computing systems,” in *Parallel Processing (ICPP), 2011 International Conference on*, 2011.
- [122] F. Wende, F. Cordes, and T. Steinke, “On improving the performance of multithreaded CUDA applications with concurrent kernel execution by kernel reordering,” in *Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing*, SAAHPC ’12, (Washington, DC, USA), pp. 74–83, IEEE Computer Society, 2012.
- [123] H. Peters, M. Koper, and N. Luttenberger, “Efficiently using a CUDA-enabled GPU as shared resource,” in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT ’10, (Washington, DC, USA), pp. 1122–1127, IEEE Computer Society, 2010.

- [124] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, “Improving GPGPU concurrency with elastic kernels,” in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pp. 407–418, ACM, 2013.
- [125] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, “rCUDA: Reducing the number of GPU-based accelerators in high performance clusters,” pp. 224–231, 2010.
- [126] Zillians, “V-GPU: GPU virtualization,” <http://www.zillians.com/products/vgpu-gpu-virtualization/>, accessed on Mar 13th, 2014.
- [127] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. R. Rajkumar, “RGEM: A responsive GPGPU execution model for runtime engines,” in *RTSS, RTSS ’11*, (Washington, DC, USA), pp. 57–66, IEEE Computer Society, 2011.
- [128] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, “Automatic CPU-GPU communication management and optimization,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’11, (New York, NY, USA), pp. 142–151, ACM, 2011.
- [129] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, “Pegasus: coordinated scheduling for virtualized accelerator-based systems,” in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2011.
- [130] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, “TimeGraph: GPU scheduling for real-time multi-tasking environments,” in *USENIXATC*, 2011.
- [131] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, “The case for GPGPU spatial multitasking,” in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*.
- [132] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” IMPACT technical report, IMPACT-12-01, UIUC, 2012.
- [133] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, “Analysis and approximation of optimal co-scheduling on chip multiprocessors,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT ’08, (New York, NY, USA), pp. 220–229, ACM, 2008.

- [134] W. Stallings, *Operating Systems: Internals and Design Principles*, 6/E. Pearson Education India, 2009.
- [135] “asfermi: An assembler for the NVIDIA Fermi instruction set,” <http://code.google.com/p/asfermi/>, accessed on Mar 13th, 2014.
- [136] NVIDIA, *Parallel Thread Execution ISA Version 3.2*, 2013.
- [137] M. Poletto and V. Sarkar, “Linear scan register allocation,” *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895–913, 1999.
- [138] Y. Zhang and J. D. Owens, “A quantitative performance analysis model for GPU architectures,” in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 382–393, IEEE, 2011.
- [139] S. S. Bagsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu, “Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors,” in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP ’12, (New York, NY, USA), pp. 23–34, ACM, 2012.
- [140] A. Papoulis and R. Probability, *Stochastic processes*, vol. 3. McGraw-hill New York, 1991.
- [141] N. Bell and M. Garland, “Cusp: Generic parallel algorithms for sparse matrix and graph computations version 0.3.0,” <http://cusplibrary.github.io/>, accessed on Mar 13th, 2013.
- [142] S. Wasserman, *Social network analysis: Methods and applications*, vol. 8. Cambridge university press, 1994.
- [143] J. Scott, *Social Network Analysis: A Handbook (2nd edition)*. Thousand Oaks, CA: Sage Publications, 2000, 2000.
- [144] S. H. Strogatz, “Exploring complex networks,” *Nature*, vol. 410, no. 6825, pp. 268–276, 2001.
- [145] J. Goldenberg, B. Libai, and E. Muller, “Talk of the network: A complex systems look at the underlying process of word-of-mouth,” *Marketing letters*, vol. 12, no. 3, pp. 211–223, 2001.
- [146] M. Granovetter, “Threshold models of collective behavior,” *American journal of sociology*, pp. 1420–1443, 1978.

- [147] NVIDIA, “Curand library,” 2010.
- [148] N. Brookwood, “AMD fusion family of apus: Enabling a superior, immersive PC experience,” *Insight*, vol. 64, no. 1, pp. 1–8, 2010.
- [149] J. He, M. Lu, and B. He, “Revisiting co-processing for hash joins on the coupled cpu-gpu architecture,” *Proceedings of the VLDB Endowment*, vol. 6, pp. 889–900, Aug. 2013.
- [150] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, *et al.*, “Larrabee: a many-core x86 architecture for visual computing,” in *ACM Transactions on Graphics (TOG)*, vol. 27, p. 18, ACM, 2008.
- [151] NVIDIA, “CUDA toolkit,” <https://developer.nvidia.com/cuda-toolkit>, accessed on Mar 13th, 2014.