William & Mary
CHARTERED 1693

W&M ScholarWorks

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

2021

Exploring Heterogeneous Architectures With Tools And Applications

Du Shen
William & Mary - Arts & Sciences, shendu.nju@gmail.com

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Shen, Du, "Exploring Heterogeneous Architectures With Tools And Applications" (2021). *Dissertations, Theses, and Masters Projects.* William & Mary. Paper 1616444361.
http://dx.doi.org/10.21220/s2-f8gh-0967

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Exploring Heterogeneous Architectures with Tools and Applications

Du Shen

Williamsburg, VA, USA

Master of Science, College of William & Mary, 2013

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
January 2021

© Copyright by Du Shen 2021

APPROVAL PAGE

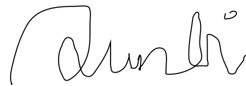
This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Du Shen

Approved by the Committee, January 2021

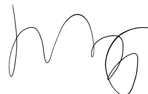
Co-Chair
Xu Liu, Courtesy Assistant Professor, Computer Science
North Carolina State University



Co-Chair
Qun Li, Professor, Computer Science
College of William & Mary

Weizhen Mao, Professor, Computer Science
College of William & Mary

Bin Ren, Assistant Professor, Computer Science
College of William & Mary



Dr. Ang Li, HPC Group
Pacific Northwest National Laboratory

ABSTRACT

Heterogeneous architectures, including GPU accelerators and memory subsystems consisting fast/slow components, have become popular due to programming flexibility and energy efficiency. Achieving high performance requires sophisticated tools and applications for heterogeneous architectures because they either lack hardware support for fast memory component, or provide complex programming model, which puts extra burdens on compilers and programmers. However, existing tools either rely on simulators or lack support across different GPU architectures, runtime or driver versions. Thus, they only provide insufficient insights.

In the first project, we develop DataPlacer, a profiling tool to provide guidance for data placement. We characterize a real heterogeneous system, the TI KeyStone II, whose memory system consists of fast and slow component, and the fast memory lacks hardware support. We develop a set of parallel benchmarks to characterize the performance and power efficiency of heterogeneous architectures. DataPlacer analyzes memory access patterns and provides high-level feedback at the source-code level for optimization. We apply the data placement optimization to our benchmarks and evaluate the effectiveness of HM in boosting performance ($11\times$ speedup) and saving energy (50% reduction in energy consumption).

In the second project, we present CUDAAdvisor, a profiling framework to guide code optimization in modern NVIDIA GPUs. General-purpose GPUs have been widely utilized to accelerate parallel applications. Given a relatively complex programming model and fast architecture evolution, producing efficient GPU code is nontrivial. CUDAAdvisor performs various fine-grained analyses based on the profiling results from GPU kernels, such as memory-level analysis (e.g., reuse distance and memory divergence), control flow analysis (e.g., branch divergence) and code-/data-centric debugging. CUDAAdvisor supports GPU profiling across different CUDA versions and architectures. We demonstrate several case studies that derive significant insights to guide GPU code optimization for performance improvement.

In the third project, we present Presponse, a GPU-based incremental graph processing framework which reduces response latency for large-scale graph queries. We first fill the gap that few incremental graph algorithms have been tailored for GPUs. Then, based on the key observation that graph evolution often follows certain patterns that can be accurately predicted, our framework speculatively conducts preprocessing on the graph during the idle period ahead of real graph update, significantly reducing response time. Experiments show that Presponse can predict over 90% of future graph updates, yielding up to a $25\times$ speedup in graph query response latency.

TABLE OF CONTENTS

Acknowledgments v

Dedication vi

List of Tables vii

List of Figures viii

1 Introduction 2

1.1 Problem Statements 3

1.2 Contributions 5

1.3 Dissertation Organization 6

2 Characterizing Emerging Heterogeneous Memory 7

2.1 Introduction 7

2.2 Related Work 10

2.3 Testbed Description and Motivation 12

2.4 Design and Implementation of HMBench 15

2.4.1 Benchmark Description 17

2.4.2 Simple Benchmark Implementation 17

Work decomposition 18

Code adaptation for the DSP compiler 18

System I/O 18

Summary: HMBench vs. Rodinia 19

2.4.3 Limitation of HMBench Implementation 19

2.5 Design and Implementation of DataPlacer 20
2.5.1 Basic Methodology of DataPlacer 20
Tracking array allocations 21
Collecting and attributing memory traces 21
Deriving metrics 22
2.5.2 Refined Methodology of DataPlacer 22
Data locality 23
Large arrays 23
Private vs. shared HM 24
Static vs. dynamic placement 25
Hybrid memory subsystem 27
2.5.3 DataPlacer Output 27
2.6 Evaluation 29
2.6.1 Optimizing HMBench on KeyStone II 29
mtrans 30
lud 30
nw 30
srad 31
Further analysis on speedups 31
2.6.2 Performance Characterization 32
2.6.3 Power Characterization 35
2.6.4 Takeaways from Experimenting KeyStone II 37
2.7 Limitations with KeyStone II 37
2.8 Chapter Summary 38

3 CUDAAdvisor: LLVM-Based Runtime Profiling for Modern GPUs 39
3.1 Introduction 39
3.2 Existing GPU Profilers and Limitations 42

3.3 CUDAAdvisor Methodology 43
3.3.1 CUDAAdvisor Instrumentation Engine 44
(I) Mandatory instrumentation 44
(II) Optional instrumentation 45
3.3.2 CUDAAdvisor Profiler 46
3.3.2.1 Code-centric Profiling 46
3.3.2.2 Data-centric Profiling 47
3.3.2.3 Profiling Outputs 48
3.3.3 CUDAAdvisor Analyzer 48
3.3.4 Limitations of CUDAAdvisor 49
3.4 Evaluation 49
3.4.1 Evaluation Methodology 50
3.4.2 Case Studies 51
3.5 Tool's Overhead Analysis 64
3.6 Related Work 65
3.7 Chapter Summary 66

4 Presponse: Accelerating Incremental Large Graph Processing on GPU via Speculative Preprocessing 67
4.1 Introduction 67
4.2 Background 70
4.2.1 Graph Embedding 70
4.2.2 GPU Acceleration 70
4.3 Methodology 71
4.3.1 Workflow Overview 71
4.3.2 Presponse Prediction Engine 72
4.3.3 Presponse Graph Processing 75
4.3.4 Graph Algorithm Implementation 76

4.3.4.1 Breadth First Search (BFS) 76
4.3.4.2 Connected Components (CC) 78
4.3.4.3 Triangle Counting (TC) 79
4.4 Evaluation 82
4.4.1 Experiment Setup 82
Evaluation Overview 82
Evaluation Platform 82
Evaluated Algorithms 82
4.4.2 Incremental Algorithms 83
4.4.3 Update Batch Prediction 87
4.4.4 Presponse Performance with Incremental Computation and
Link Prediction 88
BFS 89
CC 90
TC 91
4.5 Related Works 91
4.6 Chapter Summary 94

5 Conclusion 95

Bibliography 97

ACKNOWLEDGMENTS

This dissertation is written with the support and help from many individuals. I would like to thank all of them.

First and foremost, I would like to express my deepest appreciation to my advisor and all my committee, Dr. Xu Liu, Professor Qun Li, Professor Weizhen Mao, Dr. Bin Ren and Dr. Ang Li, for serving on my committee as well as providing insightful comments.

I enjoyed working in the office also because of my fellow students: Shasha Wen, Probir Roy, Qingsen Wang, Hao Xu, Pengfei Su, Bolun Li and Jialiang Tan.

In addition, I would like to express my gratitude to my friends here in the ’burg: Mi, Zhen, Han, Shasha, Hongyang. The time we spent together means a lot!

Last but not least, special thanks to Taro for making the world a happier place! Your companionship and comfort helped me through the hard times. Keep on wagging!

This dissertation is dedicated to my beloved parents, for their endless and selfless love and support.

LIST OF TABLES

2.1 Bandwidth comparison of MSMC and DDR with a single thread. . 15

2.2 Benchmark descriptions. 16

2.3 DataPlacer's optimization decisions based on two metrics. 27

2.4 The analysis and optimization guidance provided by DataPlacer. The speedups are measured for all benchmarks running with eight threads in KeyStone II. 34

3.1 GPU architectures for evaluation. 49

3.2 Benchmarks for showcasing CUDAAdvisor. 50

3.3 Results of Branch Divergence on Pascal. 58

4.1 Graph Datasets for Algorithm Evaluations. 83

4.2 Description of Real-world Datasets. 87

LIST OF FIGURES

2.1 The architecture and memory hierarchy of the KeyStone II. Part of L2 cache in DSP and the whole MSMC shared by DSP and ARM are configured as HM by default. 13
2.2 The functionality of DataPlacer. DataPlacer monitors program execution on x86 and generates pure software-based profiles to guide program optimization when porting the code to an HM-based architecture. 21
2.3 Creating a CCT for a program and pruning it by discarding nodes with small numbers of memory accesses. The blue nodes are internal functions, while the red nodes are leaf functions. 26
2.4 An output example of DataPlacer when monitoring `srad`. 28
2.5 Comparison of whole-system energy consumption between baseline and optimized benchmarks running with eight threads. The vertical axis indicates the energy consumption, measured in Joules. 35
2.6 Comparison of dynamic energy consumption between baseline and optimized benchmarks running with eight threads. The vertical axis indicates the energy consumption, measured in Joules. . . . 36

3.1 Workflow of CUDAAdvisor. 43
3.2 The workflow of the engine inserting instrumentation. 45
3.3 CUDAAdvisor's data-centric profiling. 48

3.4 Reuse distance analysis through CUDAAdvisor. ∞ is defined as data is never reused again during the program execution or before the next write to the address (e.g., write-evict L1 on NVIDIA GPUs). 53

3.5 Profiled memory divergence distribution of unique touched cache lines by instructions of an entire application on Kepler. X-axis represents number of unique cache lines touched (min is one and max is 32). (a) Kepler architecture with 128 Byte cache line; (b) NVIDIA Tesla P100 (Pascal) with 32 Byte cache line. 56

3.6 Normalized execution time of different applications on NVIDIA Kepler architecture when using the predicted optimal number of warps per CTA for bypassing. Baseline case is using all the warps (no bypassing). Oracle exhaustively searches the optimal solution. Prediction represents our model. 59

3.7 Normalized execution time of different applications on NVIDIA Pascal architecture when using the predicted optimal number of warps per CTA for bypassing. Baseline case is using all the warps (no bypassing). Oracle exhaustively searches the optimal solution. Prediction represents our model. 60

3.8 Code-centric view shows concatenated calling context from both host and device. 63

3.9 Data-centric view shows the interesting data objects, where it is allocated on host and device and where it is transferred. 64

3.10 Overhead of memory and control flow instrumentation, on Kepler and Pascal Architectures. 64

4.1 Comparison of three approaches to evolving graph: (a) static recomputation, (b) incremental processing, (c) proposed approach. The horizontal bottom arrow indicates time. The vertical dashed lines indicate batch emit times. The length of blue arrows indicate the processing time. 68
4.2 Workflow of Presponse. The rectangles represents components of Presponse. Given the original graph, static processing obtains initial property and prediction engine generates prediction updates. Incremental processing takes updates as input and computes intermediate properties. When an update batch is emitted, incremental processing takes correction as input and computes updated property. Orange shaded area represents procedures which run repeatedly for each batch. 72
4.3 Details of training a classifier for Presponse's prediction engine. . 74
4.4 Speedup of incremental BFS. Horizontal axes are log-scaled. Horizontal axes represent the size of update batches. Vertical axes represent incremental BFS's speedup against static BFS. 83
4.5 Relation between speedup and fraction of impacted vertices. The horizontal axis represents the fraction of non-impacted vertices in percentage. The vertical axis represents the ratio of elapsed time of incremental BFS computation to static BFS computation. The blue scattered points are data points, while the red dashed line represents linear fitting. 84
4.6 Speedup of incremental CC. Horizontal axes are log-scaled. For deletion cases, both axes are log-scaled. Horizontal axes represent the size of update batches. Vertical axes represent incremental CC's speedup against static CC. 84

4.7 Speedup of incremental TC. Both axes are log-scaled. Horizontal axes represent the size of update batches. Vertical axes represent incremental TC's speedup against static TC. 85
4.8 Relation between speedup and fraction of impacted vertices. Horizontal axis represents the percentage of impacted vertices. Vertical axis represents speedup. The blue scattered circles represent each data point, and the red dashed line represents theoretical speedup obtained via regression. 87
4.9 Prediction accuracy. Horizontal axis represents the time. Vertical axis represents prediction accuracy in percentage. 89
4.10 Prediction accuracy when re-training enabled. (a) for apple, and (b) for android. Horizontal axis represents the time. Vertical axis represents prediction accuracy, in percentage. Original setup without re-training is shown as baseline. Red triangle markers indicate points of re-training. 90

Exploring Heterogeneous Architectures with Tools and Applications

Chapter 1

Introduction

In modern computer systems, emerging parallel architectures offer heterogeneous memory systems to bridge the huge gap between processors and memory. Due to the computation efficiency and programming flexibility, heterogeneous architectures have become increasingly popular and they are widely adopted in computing domains such as accelerating scientific computing applications, deep learning and graph workloads.

A heterogeneous memory system consists of a fast memory component in completion to traditional memory systems. There are various types of heterogeneous architectures, such as Intel Xeon Phi, Texas Instruments KeyStone and Graphics Processing Units (GPUs). Their fast memory component has a higher bandwidth and lower latency. For example, Knights Mill, the latest model of Intel Xeon Phi, has an on-package memory that has $4\times$ bandwidth as the off-package memory. In addition to that, unlike traditional hardware-managed cache, fast memory component may allow software management where one can control when to hoist or evict data. With this flexibility in programming, one can explicitly control data placement and replacement to further reduce latencies and energy consumption incurred by inappropriate date movement.

However, higher flexibility comes with higher programming requirements. First, heterogeneous architectures require using certain low-level programming models, such as CUDA [1] and OpenCL [2]. Second, without deep insights of the specific architecture, one can suffer from inappropriate usage of the precious resources of fast memory, re-

sulting in higher overhead and energy consumption. For example, the cached data may be evicted without being fully utilized due to an eviction. Similarly for SIMT programming models, such as GPU, control flow divergence or memory divergence can significantly hurt performance. Thus, it is non-trivial to design efficient programs for heterogeneous architectures to fully utilize the computation power.

It is tedious, error-prone and sometimes impossible for large codebases to manually analyzing programs for heterogeneous architectures. Characterizing the performance and energy consumption is important to aid programmers to pinpoint performance bottlenecks. Existing tools like profilers may only provide coarse-grained analysis and the output can barely be helpful to aid programmers directly. On the other hand, simulators and emulators usually incur undesirable overhead and have a limited support for emerging hardware features. Therefore, sophisticated profiling tools and tailored applications are in demanded to achieve high performance on heterogeneous architectures. Thus, in this dissertation, we tackles these problems from two aspects: characterizing heterogeneous architectures with aid find-grained profiling tools and designing tailored applications.

1.1 Problem Statements

Characterizing heterogeneous memory subsystems. Nowadays, CPU employs multiple levels of caches to bridge the gap between processor and memory. Accessing memory incurs high latency. However, cache is precious resource due to the limited space. Cache is transparent to programmers and provides only hardware control. That being said, one can not explicitly conduct data placement in cache. Alternatively, emerging parallel architectures offer heterogeneous memory subsystems to complement hardware-managed caches. A typical heterogeneous memory subsystem consists fast and slow memory component and fast memory requires software management. Therefore, without guidance of a tool, inappropriate data placement and movement can

result in unsatisfactory memory performance. Prior work mainly employs simulators to study the data placement. However, it is difficult to simulate every feature of the complex architecture of heterogeneous memory subsystems. In addition, it is time-consuming to evaluate real parallel applications due to the high overhead of simulation. To tackle these two issues, we study real hardware in this dissertation.

Monitoring program execution and guiding optimization for GPUs. General-purpose GPUs are widely adopted in various application domains, including scientific computing, deep learning and graph workloads. Unlike CPUs, GPUs offer a more complex programming and architectural scenario. Thus, efficiently designing a GPU kernel is difficult. For example, designing a GPU kernel usually requires low-level programming languages, such as CUDA and OpenCL. Since GPUs employs SIMT (Single-Instruction Multiple-Threads) programming model, control flow divergence and memory divergence can hurt performance. Moreover, thousands of threads may compete for the precious cache resources. Prior work relies on simulators or emulators to perform fine-grained analysis of GPU programs. However, simulator and emulators do not support every feature of complex architectures. Some other tools may provide fine-grained analysis but have their limitations in terms of portability and coverage. In order to address all these challenges, we present a framework that provides insight of GPU program, covers the interaction between GPU and GPU, and guides optimization.

Accelerating incremental large graph processing on GPU. Large-scale graphs are pervasive nowadays. Meanwhile, enormous queries on certain graph properties impose increasing restrictions on response latency. This demands huge computation power such that GPU is frequently leveraged and incremental processing is adopted for acceleration. However, adopting GPU and incremental processing does not always provide fast responses. It is desired that we further shrink the graph query latencies. Existing works have focused on CPU or Intel Xeon Phi style architectures. However, GPUs introduces different design principles and performance concerns in the parallel execution. Moreover, existing works do not seek approaches other than optimizing algorithms. To

tackle this, we propose an approach that boosts graph processing via link prediction.

1.2 Contributions

Characterizing heterogeneous memory subsystems. We characterize the fast and slow component of heterogeneous memory subsystems and apply data placement optimization and evaluate the effectiveness in boosting performance and saving energy. Specifically, we make three contributions.

- We develop a benchmark suite, HMBench. HMBench is coded in OpenMP and runs on heterogeneous memory architectures. To the best of our knowledge, it is the first benchmark suite based on OpenMP 4.0 standard.
- We design a performance tool, DataPlacer, to guide data placement in heterogeneous memory. DataPlacer guides programmers to port their code to a heterogeneous memory system and provides rich information to intuitively present the analysis results.
- We optimize HMBench guided by DataPlacer. We use optimized benchmarks to characterize and understand the importance of HM in both performance and energy.

Monitoring program execution and guiding optimization for GPUs. We present a framework that provides insight and guide optimization for GPU program. Specifically, we make three contributions.

- CUDAAdvisor is the first fine-grained GPU profiler that supports various generations of modern NVIDIA GPU architectures and CUDA versions, to the best of our knowledge.
- CUDAAdvisor combines the code- and data-centric profiling results from both CPU and GPU, and associates performance bottlenecks with their root causes.

- We demonstrate CUDAAdvisor is able to combine different analyses and derive useful metrics and insights to guide optimizations for GPUs.

Accelerating Incremental Large Graph Processing on GPU. We develop a framework that employs link prediction to boost incremental graph process to reduce graph property query latencies. Specifically, we make four contributions.

- We develop incremental implementations of three fundamental graph algorithms, Breadth First Search, Connected Components, and Triangle Counting. The implementations are tailored for GPU execution.
- We demonstrate that updates are highly predictable for real-world evolving graphs, which can greatly benefit the incremental computation.
- We propose Presponse, a framework that leverages graph update prediction and utilizes GPU to boost incremental graph processing.
- We show that Presponse can accelerate important graph algorithms on real-world graphs with significant speedups.

1.3 Dissertation Organization

The rest of this dissertation is structured as follows. In Chapter 2, we present our study of characterizing a heterogeneous memory system and guiding data placement. In Chapter 3, we present our tool that monitors program execution and guides code optimization for GPUs. In Chapter 4, we present our framework that reduces graph response latency via link prediction and incremental processing. In Chapter 5, we conclude the dissertation.

Chapter 2

Characterizing Emerging Heterogeneous Memory

2.1 Introduction

In modern computer systems, the speed gap between processors and memory has become huge. As a result, accessing main memory incurs not only high latency but also excessive energy. To bridge such a speed gap, CPUs employ multiple levels of caches. Cache hits reduce memory access latency. Caches are precious resources due to their limited space. For a traditional memory subsystem, hardware manages caches at the granularity of cache lines. Hardware also employs built-in algorithms, e.g., least recently used (LRU), to determine which lines of data to evict. Moreover, multi-core systems employ sophisticated protocols (e.g., MESI) to guarantee the data consistency in private and shared caches, associated with different cores and sockets.

Caches are transparent to compilers and programmers. One cannot explicitly control the data placement and replacement in caches. Software usually cannot explicitly control the data locality to exploit caches. One policy does not fit all usage patterns. Hence, existing hardware-managed caches do not provide a straightforward way to achieve the optimal performance. Without the entire program profile, hardware is handicapped in making best data movement decisions. The situation is aggravated in the context of

parallel architectures, where cores compete for shared cache. For example, the cached data may be evicted without being fully utilized due to an eviction caused by another core. Contention can significantly degrade program performance. Because of the existing hardware-managed cache system, one can only use some workarounds to explicitly but indirectly interact with caches, such as non-temporal instructions [3], cache partitioning based on page coloring [4], and memory footprint reduction via loop tiling [5]. Though effective, these workarounds rely on special support from instruction set architectures (ISAs), special hardware, customized operating systems and complex source code transformations.

As an alternative, emerging parallel architectures offer heterogeneous memory (HM, also known as hybrid memory) to complement hardware-managed caches. A typical HM system consists of a fast memory component and a slow memory component. The fast component, unlike traditional caches, needs explicit software operations to hoist or evict data in or out. Memory techniques include 3D stack memory [6] and non-volatile memory [7], which together with traditional DRAM form the emerging HM systems. For example, the latest generation of Intel Xeon Phi, Knights Landing (KNL), has on/off-package memories. The on-package memory has $5\times$ the bandwidth of off-package memory [8]. The KNL's memory hierarchy is a kind of HM. Moreover, scratchpad memory [9] is widely used in accelerators, such as GPUs and digital signal processors (DSPs), and has a higher bandwidth and lower latency than DRAM. *In this chapter, we refer to a system with fast and slow memory as HM.*

HM offers flexibility in managing data. For example, programmers can partition the fast memory across different threads to avoid contention. Another advantage of HM is its power efficiency. HM does not require power hungry hardware mechanisms for cache management [10]. In the foreseeable future, HM will become more popular to complement hardware caches for its programming flexibility.

However, software-based data movement can incur much higher overhead than a hardware-based approach. Thus, inappropriate data placement and frequent data

movement in HM can significantly degrade memory performance, negating its benefit. Therefore, it is important to characterize the performance and energy consumption of HM to achieve beneficial data placement. Prior work mainly utilizes simulators to study the data placement [11, 12, 13, 14, 15]. There are two weaknesses to this approach: first, given the complex architectures, it is difficult to simulate every feature of HM and its interactions with the CPUs. Second, due to the high overhead of simulation, it is time consuming to evaluate real, long-run parallel programs. To address these two issues, we study HM in real hardware in this chapter.

Given the real hardware evaluation, we have the following questions: (1) For a real parallel program, how should we place its data in HM to achieve high performance? (2) How much can a real HM affect a program's performance and energy? In pursuit of answers to these questions, we make three contributions in this chapter:

- We develop a benchmark suite, HMBench. HMBench is coded in OpenMP and runs on HM-based machines. To the best of our knowledge, it is the first benchmark suite based on OpenMP 4.0 standard for studying the performance and energy impacts of HM.

- We design a performance tool, DataPlacer, to guide data placement in HM. This tool can help programmers when they try to port their code to a system with HM. DataPlacer provides rich information sorted by key metrics to intuitively present the analysis results.

- We optimize HMBench guided by DataPlacer. We utilize the original and optimized benchmarks to characterize the capabilities of HM-based architectures and understand the importance of HM in both performance and energy.

We use KeyStone II [10], a server-class ARM+DSP heterogeneous architecture from Texas Instruments, in our studies. KeyStone II's HM consists of an off-chip DRAM and a relatively large on-chip SRAM (scratchpad memory). We choose DRAM+scratchpad memory to study HM because stacked memory and non-volatile memory are not com-

mercially available. Moreover, the scratchpad memory on KeyStone II is much larger than the ones in existing embedded systems, which can be effectively used to emulate the emerging fast memory. In the rest of this chapter, we refer to DRAM and scratchpad memory as HM on KeyStone II.

Our experiments show that HM demands extensive attentions to obtain high performance and low energy benefits. Our DataPlacer provides rich information with reasonable overhead to successfully guide data placement in HM. This chapter also provides insights and practical evaluation of the KeyStone II.

This chapter is organized as follows. Section 2.2 reviews state-of-the-art work and distinguishes our approach from prior work. Section 2.3 describes KeyStone II, the testbed we use to evaluate HM. Section 2.4 describes the design and implementation of HMBench, which is used to characterize the performance and energy effects of HM. Section 2.5 describers DataPlacer, a tool to provide high-level guidance for optimization with HM. Section 2.6 studies HMBench in KeyStone II, including performance characterization, power measurement, and HM-based optimization. Section 2.7 discusses some limitations in studying HM with KeyStone II. Section 4.6 offers our conclusions and previews future work.

2.2 Related Work

HM has been recognized as a key alternative or complement to caches and main memory [9]. Due to its simpler hardware design, it can provide high performance, predictability, and energy efficiency. Prior research demonstrates that HM-based systems are able to achieve higher performance than cache-based systems when the program data is carefully placed [16]. In modern accelerators, such as Keystone II[17], HM is regarded as a key enabler for high Gflops/Watt value [18], as has been demonstrated in hand-optimized programs [19]. However, using HM faces key challenges of programmability in obtaining good performance.

Recognizing the significance of HM, much work has been done to ease its management. Unfortunately, due to the difficulty in accessing real hardware platforms and applications, prior evaluation methodologies were limited to two categories: most of them are simulation without actual hardware [20, 21, 22]; some of them replay memory trace on hardware platforms, for which the memory trace is often recorded by using a simulator or software instrumentation [23]. Our efforts bridge the gap with a realistic benchmark suite targeting physical platforms.

The lack of an HM benchmark suite has been an important drawback. In much of the prior work, micro benchmarks have been used [16, 24, 25]. However, it is often difficult to map the micro benchmark outcome to that of real applications. Some work employs macro benchmarks in studying HM, however, with somewhat ad-hoc choices. The macro benchmarks employed include SPEC2006 [20], NAS [26], LULESH [22], or hand-selected apps [21]. It is unclear to what extent these benchmarks can exercise HM and therefore these benchmarks cannot quantify the benefits of HM.

Since HM was often employed in embedded and network processors, embedded systems benchmarks, e.g., MiBench, are often selected in evaluating HM proposals targeting these processors [23, 27]. However, embedded system benchmarks often feature very small working sets, from tens to hundreds of KBs. This is also the case for GPU. Since HM is a critical feature of GPU, well-known GPU benchmarks, e.g. SHOC[28], are often exploiting HM for performance and are used in evaluating software that manages GPU HM [29, 30]. However, SHOC benchmarks are explicitly tuned towards the small HM (a few hundreds of KB) on GPU. Moreover, the CUDA programming model cannot represent the general multithreading models in mainstream CPU processors. Thus, the resulting programs can hardly exercise the large HM that are emerging in new architectures, such as KeyStone II.

Beyond an effective benchmark suite, there exists no tool to guide the data placement in heterogeneous memory systems. Our previous work on memif [31] provides an efficient way to support data movement between fast and slow memory in KeyStone

II. However, memif is an OS service for data movement, providing no guidance for the data placement to an application developer. A recent work [32] describes a profiler to analyze memory access patterns to guide data placement. However, the profiler does not provide performance insights such as memory footprint metrics in the full calling contexts, which are important to understand the data structure allocations and program phases.

Unlike existing approaches, we developed HMBench, which has three unique advantages. First, HMBench is developed based on a widely used benchmark suite, Rodinia, which represents the program behaviors of different domains. Second, HMBench leverages the latest OpenMP 4.0 standard. Furthermore, we developed DataPlacer and released it along with HMBench. DataPlacer, to the best of our knowledge, is the first practical tool that provides high-level optimization guidance, such as a variety of metrics within the full calling contexts, when programmers port source code to a HM-based machine.

2.3 Testbed Description and Motivation

Texas Instruments (TI) Corporation developed KeyStone II, a heterogeneous chip that employs CPUs and DSPs. It aims to achieve high performance with low energy costs. A KeyStone II chip integrates a quad-core ARM Cortex-A15 processor as the host CPU and eight TMS320C66x DSPs as accelerators. Each ARM core has 1.4 GHz clock frequency, while each DSP has 1.2 GHz. The theoretical peak performance of the overall chip is 63 GFLOPS of double precision and 198 GFLOPS of single precision [17]. The ARM cores and DSPs are coupled with security and packet processing and Ethernet switching, which is designed for lower energy consumption, compared to multi-chip solutions. The programming models that KeyStone II supports are OpenMP [33] and OpenCL [34]. The design of KeyStone II is for embedded infrastructure applications, such as media processing, high-performance computing, transcoding, security, gam-

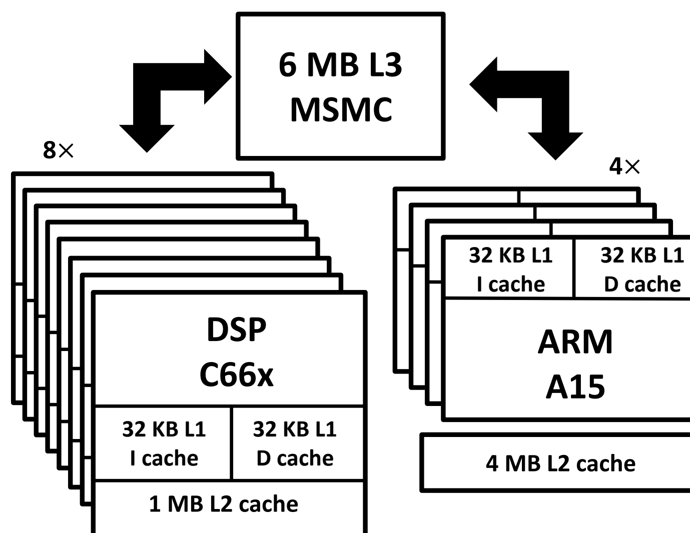


Figure 2.1: The architecture and memory hierarchy of the KeyStone II. Part of L2 cache in DSP and the whole MSMC shared by DSP and ARM are configured as HM by default.

ing, analytics, and virtual desktop.

Figure 2.1 shows the architecture of KeyStone II and its memory hierarchy. On the host side, each ARM core has a 32 KB L1 instruction cache and 32 KB data cache. All four ARM cores share a 4 MB L2 cache. Both L1 and L2 caches on the host ARM cores are managed by hardware. On the accelerator side, each DSP has a 64 KB L1 cache and a 1 MB L2 cache. By default, the L2 cache is configured as a 256 KB hardware-managed cache and a 768 KB scratchpad memory. The host and accelerators share a Multicore Shared Memory Controller (MSMC), a scratchpad memory of 6 MB. Beyond MSMC, both CPU and DSP have memory controllers to access main memory (DDR). In KeyStone II, both the scratchpad memory in L2 and MSMC are fast HM layers, which requires software to control the data placement and manage data consistency.

With the default configuration, the KeyStone II has three HM layers: L2 scratchpad memory, L3 MSMC, and DDR. Unlike other embedded systems, the scratchpad memories (L2 and L3) in the KeyStone II are large enough to emulate future HM in the mainstream CPU architectures. To explicitly place the data in each layer, TI provides APIs as shown in Listing 2.1. These APIs, like standard data allocation functions (`malloc` and

```
/* to allocate in L2 */
void __heap_init_l2 (void *ptr, int size);
/* to manage a heap on L2 */
void *__malloc_l2 (size_t size);

/* to allocate in msmc */
void __heap_init_msmc (void *ptr, int size);
/* to manage a heap on MSMC */
void *__malloc_msmc (size_t size);
void *__calloc_msmc (size_t num,size_t size);
void *__realloc_msmc  (void *ptr,size_t size);
void  __free_msmc (void *ptr);
void *__memalign_msmc (size_t alignment, size_t size);

/* to allocate in ddr */
void __heap_init_ddr(void *ptr, int size);
/* to manage a heap on DDR */
void *__malloc_ddr  (size_t size);
void *__calloc_ddr  (size_t num, size_t size);
void *__realloc_ddr (void *ptr, size_t size);
void  __free_ddr  (void *ptr);
void *__memalign_ddr  (size_t alignment, size_t size);
```

Listing 2.1: APIs for managing KeyStone II's L2 cache, MSMC, and DDR.

`free`) in `libc`, can allocate and free memory in different HM layers. The KeyStone II maps the space of different HM layers to disjoint memory segments. One can simply issue a `memcpy` to copy data from one memory segment to another.

Performance characterization of the KeyStone II's HM

To evaluate the impacts of L2, MSMC, and DDR in program performance, we developed two micro benchmarks to characterize KeyStone II: `Stream` and `Lat`. The description of the two micro benchmarks is as follows:

- Stream, a well-known benchmark [35], is used to quantify the memory bandwidth. It issues memory accesses with streaming patterns, such as array copy, scale, add, and triad. We adapt Stream to measure the bandwidth of both MSMC and DDR.
- Lat is developed for evaluating the performance impact of different data placement policies in HM-based memory hierarchies. The kernel of Lat is a sequence of random accesses to an array placed in a specific HM layer. It also moves data

Table 2.1: Bandwidth comparison of MSMC and DDR with a single thread.

HM	copy	scale	add	triad
MSMC	5.1 GB/s	4.8 GB/s	5.4 GB/s	5.3 GB/s
DDR	2.7 GB/s	2.8 GB/s	2.6 GB/s	2.8 GB/s

between different HM layers to evaluate the data movement latency.

Table 2.1 shows the experimental results of bandwidth tests in MSMC and DDR. We can see that MSMC's bandwidth is around 1.7-2$\times$ DDR's bandwidth, with respect to different access patterns. Moreover, Lat shows that placing data in MSMC obtains an 8$\times$ acceleration compared to placing data in DDR. The L2 scratchpad memory shows similar latency as MSMC. Additionally, Lat evaluates aggressive data movement policy, which always loads each word one by one from DDR to MSMC before using it. This policy significantly degrades the performance because of the nontrivial overhead incurred by the software-based memory movement. It causes a 10$\times$ slowdown compared to the original code with all accesses to DDR, and an 80$\times$ slowdown compared to the optimal code with all accesses to MSMC.

The experiments on these two micro benchmarks demonstrate the importance of data placement and data movement policies when porting code to HM-based systems. It is necessary to have a set of benchmarks and tools to characterize the performance impact of HM in the real world. The following sections describe the design and implementation of the benchmarks and tools with this purpose. *It is worth noting that the benchmarks we propose in this chapter are general for HM systems beyond the KeyStone II*. It is publicly available at `https://bitbucket.org/hmbench/hmbench.git`.

2.4 Design and Implementation of HMBench

We develop HMBench, a benchmark suite to characterize the performance impact of HM in real hardware. HMBench meets the following four criteria.

1. HMBench needs to work on heterogeneous architectures, i.e., CPU+accelerators,

Table 2.2: Benchmark descriptions.

Application	Domain	Description
`mtrans`	linear algebra	Matrix Transposition.
`mmulti`	linear algebra	Matrix Multiplication.
`bfs`	graph algorithm	Breadth-First Search one a graph.
`cfd`	fluid dynamics	Computational Fluid Dynamics solves 3-D Euler equations for compression fluid flow.
`hotspot`	physics simulation	Hotspot is a thermal simulation benchmark that assesses processor temperatures.
`kmeans`	data mining	K-means clusters points into user specified number of categories based on the distance to other points.
`lavaMD`	molecular dynamics	LavaMD computes particle potentials and relocation forces. It divides a 3D space into cubes for computation.
`lud`	linear algebra	Lud performs matrix LU decomposition.
`nn`	data mining	Nearest Neighbor is a benchmark that finds the first k nearest neighbors for a specified location.
`nw`	bioinformatics	Needleman-Wunsch is a benchmark that performs DNA sequence alignment optimization.
`particlefilter`	medical imaging	Particle Filter assesses the location of a target object with noisy measurements of the target's location.
`pathfinder`	grid traversal	Pathfinder searches a path with the lowest aggregate weights in a 2-D grid.
`srad`	image processing	Srad, with the full name Speckle Reducing Anisotropic Diffusion, is a diffusion algorithm that removes speckles from an image.

because modern HM (e.g., scratchpad memory + DRAM) is widely used in accelerators, rather than mainstream CPUs.

2. HMBench should run in parallel, as accelerators typically employ multiple threads for high thread-level parallelism, which leads to significantly different behaviors in HM from sequential execution.

3. HMBench should leverage the interfaces provided by the HM to control the data

placement and movement for evaluating different strategies.

4. HMBench should cover different kinds of applications. As the performance of HM is tightly related to memory access patterns, which differ significantly in different kinds of applications, ranging from data analytics to scientific computing. Thus, a high coverage of applications can evaluate HM thoroughly.

HMBench leverages `omp target` to support heterogeneous workloads. Thus, HMBench, with minimal adaptation, works on existing and emerging HM architectures in accelerators or co-processors, such as DSP, GPU and Xeon Phi. Moreover, it provides APIs to encapsulate memory management interfaces provided in HM-based architectures. The initial benchmark suite consists of 13 applications from different areas, including two scientific benchmarks for matrix computation and 11 benchmarks derived from Rodinia 2.2 [36]. The reason we choose to adapt Rodinia benchmarks is that Rodinia has a good coverage of application domains. It has an OpenMP implementation but no HM-aware design, which provides us an opportunity to extend its benchmarks with the OpenMP 4.0 standard and HM-friendly design. HMBench is open to enclose more benchmarks in the future. In the rest of this section, we describe different benchmarks in detail and show our design and implementation specific for HM-based accelerators.

2.4.1 Benchmark Description

Table 2.2 shows the descriptions of HMBench benchmarks. They are highly representative in their own fields according to Rodinia's specification [36]. Together with two matrix-based scientific benchmarks, HMBench has good coverage of different domains of real-world parallel applications.

2.4.2 Simple Benchmark Implementation

Porting these benchmarks to an HM-based system, such as the KeyStone II, is nontrivial. The challenges come from its uncommon architecture and system support, which

is different from general-purpose CPUs. Specifically, we need to handle work decomposition between CPU and accelerators, limited compiler support in accelerators, and lack of I/O capability in accelerators. In the rest of this section, we use the KeyStone II to illustrate the challenges and our solutions.

Work decomposition To fully leverage the computing resources in the KeyStone II, we need to split the work into the CPU part and the DSP part. We apply a simple work decomposition strategy in HMBench: we offload all OpenMP parallel regions to the DSP device and leave all non-OpenMP regions running on the CPU. As most computation is done in the OpenMP regions, this strategy can expose as much as computation for evaluating the HM on the accelerator (DSP) side.

To allow different vendor-provided compilers work on different code regions, the offload code region must be encapsulated in a subroutine and placed in a separate target source file. Thus, the CPU compiler can produce CPU binary and also cross compile the DSP binary for execution. To encapsulate each OpenMP region, we identify its inputs and outputs, which are all passed as arguments by reference to the new subroutine.

Code adaptation for the DSP compiler The DSP compiler only supports C-like syntax. Thus, the code running on the DSP cannot use C++ features, such as classes, memory allocations, and type conversions. Moreover, moving data between CPU and DSP requires the support from DSP's OpenMP compiler, which explicitly accepts array names and sizes. However, the compiler does not support multi-dimensional arrays well. In order to move multi-dimensional arrays between CPU and DSP, we need to map them to a continuous 1-D memory chunk for processing on DSP, and then map them back to the original layout for processing on CPU. Benchmarks, such as `pathfinder`, need such code transformations.

System I/O Some benchmarks, such as `bfs` and `nn`, require input files. Since the DSP software stack in the KeyStone II does not support system I/O, we need to modify the

codes so that the host CPU reads input files and map the data to the DSP for computation. DSPs process the data and move it back to the host for writing to the file system. This simple scheme works for most of our benchmarks. However, there is one exception. Benchmark `nn` repeatedly reads in 10 entries of a database for processing, until completing the whole database. To avoid frequent data movement between CPU and DSP, we perform the I/O once to read in all the entries in the input database and offload them altogether to DSP for processing.

Summary: HMBench vs. Rodinia Although HMBench shares some common programs with the Rodinia benchmark suite, it overhauls their implementations: (1) HMBench uses OpenMP 4.0 `omp target` to offload parallel regions to accelerators, where we can use the fast scratchpad memory. We identify the input and output data for `omp target` pragma to ensure the correctness of the code. Moreover, as aforementioned, we need to transpose the array layout for the data transfer between the host and accelerator. (2) HMBench fuses OpenMP regions to minimize the data transfer overhead between the host CPU and the target accelerator. (3) HMBench is extended to manage allocations of heterogeneous memory.

2.4.3 Limitation of HMBench Implementation

Our implementation of HMBench is straightforward, without taking the HM into consideration. By default, the compiler places all data in the slow memory, e.g., DDR of the KeyStone II. Benchmarks with this implementation do not achieve good performance. We need to take advantage of different HM layers to cache data for efficient accesses. However, determining which data to place in the fast HM layers is difficult, so we need a profiling tool to help make decisions. The next section describes DataPlacer for this purpose.

2.5 Design and Implementation of DataPlacer

It is challenging for programmers to port code to an HM-based system. One needs high-level guidance to place data objects in the fast HM layers to obtain high performance. In this section, we describe the design and implementation of DataPlacer, a profiler that identifies optimization opportunities of data placement in HM. Figure 2.2 shows the workflow of DataPlacer. DataPlacer works on an x86 host machine. It monitors program execution and provides optimization guidance for porting this code to an HM-based target machine. DataPlacer has the following three features to make it an effective tool.

- DataPlacer provides software metrics only. Because the host and target machines have different architectures, using the host's hardware metrics is inappropriate to guide the optimization in the target HM-based machine.
- DataPlacer provides high-level optimization guidance for programmers. The guidance can be easily used for source code transformation.
- DataPlacer can monitor parallel program execution with reasonable overhead. For private and shared HM layers between multi-cores, DataPlacer provides different optimization guidances.

In the rest of this section, we describe the basic methodology of DataPlacer and several refinements to make it practical.

2.5.1 Basic Methodology of DataPlacer

DataPlacer leverages Intel Pin [37] to instrument binary and collect memory traces. All the analyses are based on the memory traces without using any information from architecture-specific hardware performance counters. To provide high-level optimization guidance, DataPlacer performs array-centric analysis. It identifies arrays with a significant amount of accesses, which, if put into fast HM layers, can improve performance.

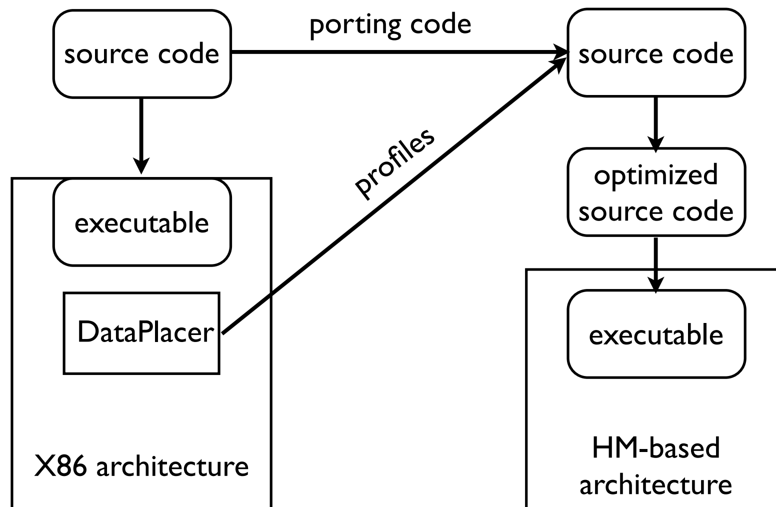


Figure 2.2: The functionality of DataPlacer. DataPlacer monitors program execution on x86 and generates pure software-based profiles to guide program optimization when porting the code to an HM-based architecture.

With this guidance, programmers can easily transform the source code for optimization. To achieve array-centric analysis, DataPlacer needs to monitor array allocations, associate memory accesses with arrays, and derive metrics for analysis.

Tracking array allocations DataPlacer leverages Pin to analyze a binary executable and monitor its execution to extract array allocations. DataPlacer monitors both static and heap arrays. On one hand, DataPlacer reads the symbol table of the binary to identify the names and memory ranges of static arrays. On the other hand, DataPlacer instruments array allocation functions, such as `malloc`, `calloc`, and `realloc`, to capture the allocated memory ranges as well as the allocation location mapped to the source code with the help of compiler debugging information. DataPlacer logs these memory ranges and IDs (names for static arrays and allocation sites in source code for heap arrays) into a map for further use.

Collecting and attributing memory traces DataPlacer utilizes Pin to instrument both memory loads and stores for their effective addresses. Upon a memory access, DataPlacer checks the map to identify the memory interval that includes the effective address

of this memory access and associates it with the array. DataPlacer counts the number of accesses attributed to each array. For multithreaded programs, accesses from multiple threads can be attributed to the same array at the same time, so DataPlacer needs to use atomic operations to ensure the correctness of accumulating the counter.

Deriving metrics From the array-centric analysis, DataPlacer obtains the number of accesses to each array. Arrays with significant accesses are candidates for being placed into the fast HM layers. To weigh the significance of arrays, we use Equation 2.1 to derive a metric $\hat{F}$ for each array, which is the average access frequency per byte. In the equation, C is the total number of memory accesses to an array. S is the number of memory bytes allocated to the array.

$$\hat{F} = \frac{C}{S} \tag{2.1}$$

DataPlacer sorts all arrays according to $\hat{F}$. With a greedy algorithm, DataPlacer recommends placing arrays with high $\hat{F}$ until the space of HM runs out.

2.5.2 Refined Methodology of DataPlacer

The basic design of DataPlacer is inadequate to be used in practice. There are five major issues.

1. Metrics $\hat{F}$ and C alone are insufficient in providing effective guidance. We need more insightful access pattern analysis to extract more features of an array, such locality, beyond the simple access quantity.

2. If the fast HM layers have limited space and the arrays used in the programs are too large to fit in, DataPlacer cannot place such large arrays. Moreover, not all elements of an array have the same number of accesses to receive the equal treatment.

3. HM may have layers that are private or shared between cores. Applying the same data placing strategy to different kinds of HM layers may hurt performance. For example, inappropriate placement can cause high overhead due to maintaining data consistency.

4. DataPlacer produces static data placement guidance. Once the data is loaded into HM, it never gets replaced. In practice, static placement preclude optimal performance because program execution can have different phases with different memory access patterns.

5. A system that integrates both traditional hardware caches and HM is difficult to optimize. DataPlacer needs to take this into consideration for HM-based data placement.

Thus, we refine DataPlacer to address all these issues.

Data locality An array with a large stride or a random access pattern does not exploit the reuse in caches. We call such array one of poor locality. An array of poor locality can significantly degrade program performance because accesses to this array are more likely to suffer from cache misses and high exposed memory latency. Therefore, DataPlacer prioritizes the placement of arrays with bad locality into fast HM. DataPlacer adopts our previous approach [38] to collect the reuse distance of memory accesses and associates them with arrays. The technology is to instrument all memory accesses and record the trace of effective addresses in a hash map for the computation of reuse distance. We report the instructions and arrays associated with long reuse distances as with poor locality. We evaluate the necessity for placing arrays of poor locality in Section 2.6.

Large arrays DataPlacer decomposes the memory intervals allocated for large arrays into small chunks with the sizes not larger than N. N is tunable by programmers; by

default, we set it as one tenth of the HM size. DataPlacer treats each chunk as a separate array and performs original array-centric analysis. With the offsets computed for chunks in the array, programmers can easily place part of the array in the HM. Besides handling large arrays that do not fit into the HM, DataPlacer's array decomposition is more appropriate for handling irregular access patterns. With irregular access patterns, elements in an array may have different access frequencies. The array decomposition provides more details in the array internals for data placement.

Private vs. shared HM HM can be private or shared in a multi-core system, e.g., the KeyStone II. For example, each DSP in the KeyStone II has a private fast layer—L2 cache— and all eight DSPs share a fast layer—L3 MSMC. Optimizations on these two kinds of fast HM layers are different. On one hand, DataPlacer recommends thread-local arrays rather than shared arrays to be placed in private HM because handling shared arrays needs to maintain data consistency. For example, if an element of a shared array is updated by one thread in the private HM, the update should be written back to the main memory. Moreover, all of the copies of this element in different private HMs have to be invalidated and reloaded from the main memory. In a traditional cache system, this data consistency is guaranteed by the hardware, which is efficient. However, HM requires software to keep the data consistency, which is expensive. Thus, DataPlacer avoids recommending shared arrays to be placed in private HM layers.

On the other hand, DataPlacer prioritizes shared arrays to be placed in shared HM layers. If there is space, DataPlacer places local arrays in the shared HM. One strength of this strategy is that no software-based data consistency is needed. Moreover, shared arrays are used by multiple threads, so loading them into shared HM can benefit many threads. In contrast, loading local arrays into shared HM only benefits a subset of threads, rather than all of them.

To provide appropriate optimization guidance, DataPlacer identifies whether an array is local or shared and adapts the array-centric analysis accordingly. When it attributes

memory accesses to arrays, it also associates the IDs of threads that perform the accesses with the array. If an array is accessed by more than one thread, DataPlacer recognizes it as a shared array. Otherwise, it is a local array.

Static vs. dynamic placement The basic implementation of DataPlacer produces the strategies of array placement in a static way: once an array is placed in the HM, it is never evicted throughout the entire execution. However, a typical program has phases. Loading an array into the fast HM layer without using it in some phases can waste the precious HM resources. Therefore, we improve DataPlacer to provide guidance for placing arrays dynamically. The main challenge of generating guidance for dynamic placement is to identify the phase changes to apply dynamic adaptation of data placement. Moreover, DataPlacer needs to provide high-level guidance that can be used by programmers to refactor their source code.

To address the challenge, DataPlacer makes an assumption: phase changes occur at function boundaries. In other words, DataPlacer applies the same data placement strategy inside a function. When switching to a different function, DataPlacer adapts the data placement strategy, if needed, based on the memory accesses in the new function only. However, frequent changes of data placement are costly because of the heavyweight software-based data movements. To reduce the overhead, DataPlacer adapts the data placement when coming into a new function that invokes a significant number of memory accesses. To provide high-level guidance, DataPlacer associates memory accesses with functions in their full calling contexts.

DataPlacer uses Pin [37] to instrument every function call and return instruction. It maintains a shadow stack to track function frames in the system execution stack. When calling a function, DataPlacer pushes the function frame, identified by the starting address of the function, into the shadow stack. When returning from a function, DataPlacer pops the function frame on top of the shadow stack. The calling context of any instruction under execution is in the shadow stack. DataPlacer accumulates the number of

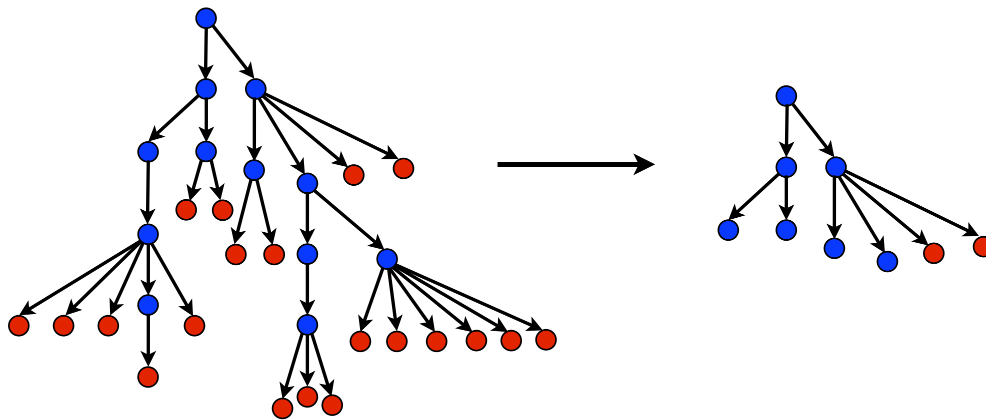


Figure 2.3: Creating a CCT for a program and pruning it by discarding nodes with small numbers of memory accesses. The blue nodes are internal functions, while the red nodes are leaf functions.

memory accesses acc to the function frame on top of the shadow stack, as exclusively to the function (not to its callers).

To efficiently maintain these per-function metrics, all the calling contexts are organized in a compact data structure, called a calling context tree (CCT) [39], by merging all common prefixes. Figure 2.3 shows a typical CCT. The root node of a CCT is the starting function, typically “main” or “thread start”; the internal nodes (in blue) are functions that have function calls inside; the leaf nodes (in red) are ones with no function call inside. To compute the inclusive metrics for each node (i.e., the aggregate metric of the function and all its callees), DataPlacer traverses the CCT from bottom to top to accumulate the inclusive acc for every node. It then prunes the CCT, leaving the nodes that account for significant proportions of memory accesses during the entire program execution, as shown in Figure 2.3. For leaf nodes in the pruned CCT, DataPlacer treats them as separate phases, in which the data placement strategy is dynamically adapted according to the array-centric metric $\hat{F}$, computed with memory accesses in the phase. As for the optimization, programmers need to flush the data in the fast HM layers at the beginning of the function and then place the data according to DataPlacer’s suggestions.

Table 2.3: DataPlacer's optimization decisions based on two metrics.

acc	fp	optimization decisions
high	high	optimization with high priority
low	high	little performance gains (low data reuse)
low/high	low	little performance gains (dataset fit in hardware cache)

Hybrid memory subsystem A system with hybrid hardware- and software-managed cache/memory is challenging for data placement. For example, each of KeyStone II's DSPs has a 256 KB L2 cache that is managed by hardware. We find that if the dataset of a program is small enough to fit into the hardware cache, placing data objects into HM does not help its performance. Therefore, DataPlacer collects memory footprints fp for each node in the pruned CCT, like acc. Memory footprint is defined as the unique memory bytes accessed in a calling context. Table 2.3 shows how DataPlacer makes optimization decisions based on these two metrics. DataPlacer suggests that optimizing data placement in contexts with high acc and fp can lead to significant performance improvement. If acc is low and fp is high, the code exposes little locality. Thus, placing data into HM does not benefit performance much. Moreover, if fp is low, the hardware caches can hold all the data, minimizing the effects of HM.

To collect fp, DataPlacer creates a hash set to maintain all unique memory bytes accessed exclusively to CCT nodes. Like acc, the hash set is merged from bottom to top in the CCT along all call paths for the inclusive footprint of a context. fp of a function is computed as the size of the inclusive hash set associated with the function.

2.5.3 DataPlacer Output

DataPlacer produces the text output once the program finishes its execution. Figure 2.4 shows an example output of DataPlacer for `srad`, a case study to be described in the next section, running with four threads. At the beginning of the output file, DataPlacer shows the total number of memory accesses in this execution. Then, DataPlacer ranks all data objects in a descending order by $\hat{F}$. In the figure, we only show one example array. DataPlacer outputs the data ID (for DataPlacer's internal usage) and the number

```
A total of 184016202 memory accesses.

Rank 0 >>>>
Dynamic Data 45703 chunk 1 accessed 33107904 times.

Data allocation call path:
45678:0x7f20c32816e0:pushq %rbp:malloc::0
  45676:0x400dda:callq 0x400bf0:main:[...]/main.c:157
    30599:0x7f20c3227b03:callq %rax:__libc_start_main::0
      29824:0x401b40:callq 0x400ba0:_start: [...]/sysdeps/x86_64/start.S:122
        1:(nil)::THREAD[0]_ROOT_CTXT::0

size = 898.109 KB contribution = 17.9918% F = 36.00

accessed by threads:
12445583 6927600 6867360 6867360

Footprint and accesses per context
Footprint is 4591168 Bytes, #accesses is 22831020
Calling contexts:
24341:0x401cf0:movsxdl  (%r14,%rax,4), %rcx:main._omp_fn.1::0
  24289:0x401590:callq  0x401c10:main::0
    14026:0x7f896debbb03:callq  %rax:__libc_start_main::0
      13647:0x401b40:callq  0x400ba0:_start:/home/abuild/rpmbuild/BUILD/
      glibc-2.19/csu/../sysdeps/x86_64/start.S:122
        1:(nil)::THREAD[0]_ROOT_CTXT::0
```

Figure 2.4: An output example of DataPlacer when monitoring `srad`.

of accesses. For static arrays, DataPlacer displays its name. For dynamic arrays, DataPlacer prints the full call path so that a programmer can associate the data object with the source code. In this example, the data object is allocated on heap by `malloc`. DataPlacer also maps the call paths to the source code for easy interpretation: the `malloc` is called at line 157 in `main.c`.

Moreover, DataPlacer computes the array size in bytes, the contribution of memory accesses (in percentage) to the whole program execution, and $\hat{F}$. DataPlacer lists the number of accesses by each thread and identifies whether the data object is shared or private. In this example, as all the four threads access this array, this array is shared by all the threads and should be placed into the shared fast memory layer.

DataPlacer also reports acc and fp, as shown in Table 2.4 for the whole program. To give dynamic optimization guidance, DataPlacer further reports acc and fp in all the functions with full calling contexts. Figure 2.4 shows one example OpenMP function. As this OpenMP function performs the most computation, DataPlacer suggests to target this function with one strategy to place the array in the fast memory, highlighted in the

allocation call path.

It is worth noting that the text output of DataPlacer can contain thousands of lines because all the allocations and functions with their full call paths are included. However, by sorting all the items (sorting arrays with $\hat{F}$ and sorting functions with acc), we can successfully shrink the searching space and focus on a few arrays and function contexts. In the future, we plan to build a graphical interface for DataPlacer for easy data interpretation.

2.6 Evaluation

We evaluate HMBench and DataPlacer on the TI KeyStone II. The configuration of KeyStone II is described in Section 2.3. The compiler on the host side is `gcc 4.7.2`, while the complier on the device side is TI's OpenMP Accelerator Model Compiler `clacc 1.1.1`. We compile all the benchmarks in HMBench with `-O3`. DataPlacer collects execution profiles of HMBench on an x86 machine, which has 16 Intel Xeon 3.2 GHz cores, with 192 GB memory. DataPlacer monitors the program executions with eight threads. The overhead is 40-60$\times$ the native execution. We average the execution time and power consumption with running each experiment five times; we find that variance is negligible.

We discuss our experiments in four aspects. In Section 2.6.1, we optimize HMBench according to the guidance of DataPlacer. In Section 2.6.2, we characterize the performance difference of HMBench due to HM in KeyStone II. In Section 2.6.3, we characterize the difference in power consumption with the utilization of HM in KeyStone II. Finally, we discuss some issues in our experiments in Section 2.6.4.

2.6.1 Optimizing HMBench on KeyStone II

With the guidance of DataPlacer, we are able to apply the optimizations to all the benchmarks in HMBench. To evaluate DataPlacer and demonstrate our optimizations, we study four benchmarks in detail. Without specific explanation, the speedups we report

```
1  /* allocation and initialization */
2  #pragma omp parallel for ...
3  for( i = 0 ; i < N ; i++)
4    for( j = 0 ; j < N ; j++)
5      B[j][i] = A[i][j];
```

Listing 2.2: Code snippet of `mtrans`: matrix `A` is transposed into matrix `B`.

are over the default execution of HMBench on KeyStone II, which does not use the fast HM.

mtrans There are two arrays in this micro benchmark, the original matrix `A` and the transposed one `B`, as shown in Listing 2.2. Both matrices are shared by all threads. DataPlacer suggests we should place matrix `B` into fast memory if there is not enough room for both, because of the bad locality in matrix `B`. We optimized the application following DataPlacer and observed an 11.51× speedup. In contrast, placing matrix `A` (of good locality) into fast memory obtains only a 5.52× speedup, or half the performance gain of placing matrix `B`.

lud There are two phases in `lud`: a file input phase and a kernel computation phase. DataPlacer identifies that there is only one significant array `m`, which contains all the matrix data for decomposition computation. As shown in Listing 2.3, m is allocated in `create_matrix_from_file` and used in `lud_omp`, the parallel kernel. Array `m` accounts for ∼18% of total memory accesses. Moreover, `m` is shared by all threads, so DataPlacer suggests placing it into fast memory. We apply the optimization according to DataPlacer's guidance and achieve a 3.95× speedup for the OpenMP parallel region when running on eight DSPs.

nw DataPlacer identifies two significant arrays used in `nw`. As shown in Listing 2.4, the two arrays `referrence` and `input_itemsets` with the same size, 2.2 MB. Both of them are used in a parallel region, shared by all threads. These two arrays account for ∼32% of total memory accesses. With this performance insights, DataPlacer recommends

```
1  float *m;
2  ...
3  /* file input */
4  create_matrix_from_file(&m,input_file,&matrix_dim);
5  ...
6  / *kernel computation */
7  lud_omp(m, matrix_dim);
```

Listing 2.3: Code snippet of `lud`. Array `m` reads the input file and then is passed for kernel computation.

placing both arrays into fast memory, which leads to a 1.5× speedup for the overall program.

srad Besides the array highlighted in Figure 2.4, DataPlacer identifies six more significant arrays, as shown in Listing 2.5, which account for ∼50% of total memory accesses in the program. Threads share six of these arrays in the following parallel region. Ideally, DataPlacer recommends placing all seven arrays in the fast memory. However, due to the limited space, the fast memory cannot hold all the arrays. With the analysis of DataPlacer, we place five arrays with the highest $\hat{F}$ to the fast memory. These arrays are `image`, `dN`, `dS`, `dW` and `c`. As for the optimization, the array `image` needs to be initialized in the host and then passed to the device. For the other four arrays, they can be initialized on the device. With this optimization, we obtain a 1.15× speedup.

Further analysis on speedups Table 2.4 summarizes the optimization to all the benchmarks in HMBench with the guidance of DataPlacer. In the table, we show the footprint, the number of accesses, and the number of arrays placed into fast memory under the guidance of DataPlacer. We set two baselines to make the comparison. Baseline $B1$ is the default program configuration without utilizing the fast memory. Baseline $B2$ utilizes scratchpad memory with a naive data placement strategy: first come, first served. From the table, we can see that eight of 13 benchmarks benefit from the HM optimization and achieve more than 1.10× speedups over $B1$. Among them, mtrans and lud obtain significant speedups. However, benchmarks like `kmeans`, `lavaMD`, `particlefilter`,

```
1  referrence = (int *)malloc( max_rows * max_cols * sizeof(int) );
2  input_itemsets = (int *)malloc( max_rows * max_cols * sizeof(int) )
       ;
3  ...
4  /* process top-left matrix */
5  #pragma omp parallel for ...
6  for( idx = 0 ; idx <= i ; idx++){
7    ...
8    input_itemsets[index]= maximum( input_itemsets[index-1-max_cols]
         + referrence[index],
9         input_itemsets[index-1] - penalty,
10        input_itemsets[index-max_cols] - penalty);
11 }
12 ...
13 /* process bottom-right matrix */
14 #pragma omp parallel for ...
15 for( idx = 0 ; idx <= i ; idx++){
16   ...
17   input_itemsets[index]= maximum( input_itemsets[index-1-max_cols]
         + referrence[index],
18        input_itemsets[index-1] - penalty,
19        input_itemsets[index-max_cols] - penalty);
20 }
```

Listing 2.4: Code snippet of `nw`. Arrays `referrence` and `input_itemsets` are frequently accessed.

and `pathfinder` obtain nearly no performance improvement. Therefore, not all kinds of benchmarks can benefit from HM. We discuss the performance impact of HM to different kinds of benchmarks in the next section. Moreover, we can see that five out of 13 benchmarks (`mtrans`, `mmulti`, `bfs`, `hotspot`, and `nw`) achieve more than $1.10\times$ speedup over $B2$. On average, DataPlacer achieves a speedup of 1.56$\times$ and 1.17$\times$ over $B1$ and $B2$ baselines, respectively.

2.6.2 Performance Characterization

In this section, we characterize the performance impact of HM and identify the workload features that can benefit from HM. We mainly focus on the performance of parallel regions in these benchmarks. Common to all the benchmarks that benefit from HM, they have three features. First, their parallel regions should be large enough to avoid parallel overhead in OpenMP from overwhelming the execution time. For example, the parallel

```
1  image_ori = (fp*)malloc(sizeof(fp)*image_ori_elem);
2  ...
3  image = (fp*)malloc(sizeof(fp) * Ne);
4  ...
5  dN = malloc(sizeof(fp)*Ne); // north direction derivative
6  dS = malloc(sizeof(fp)*Ne); // south direction derivative
7  dW = malloc(sizeof(fp)*Ne); // west direction derivative
8  dE = malloc(sizeof(fp)*Ne); // east direction derivative
9  ...
10 c  = malloc(sizeof(fp)*Ne); // diffusion coefficient
11 ...
12 resize( image_ori, image_ori_rows,...);
13 ...
14 #pragma omp parallel for ...
15 for (j=0; j<Nc; j++)
16   for (i=0; i<Nr; i++) {
17     ...
18     // divergence
19     D = cN*dN[k] + cS*dS[k] + cW*dW[k] + cE*dE[k];
20     // updates image
21     image[k] = image[k] + 0.25*lambda*D;
22     ...
23   }
24 }
```

Listing 2.5: Code snippet of `srad`. There are seven arrays with significant accesses in the OpenMP parallel region.

region in lud accounts for almost 100% of the program execution time, so our optimization shows a significant speedup. Second, parallel regions have reasonable memory footprints and accesses. Third, benchmarks have hot arrays, whose placement in the HM can benefit a large number of memory accesses. For example, `lud`, `nw`, and `srad` have hot arrays; placing them in HM can benefit 18-50% of the total memory accesses.

However, as shown in Table 2.4, there are benchmarks having little performance improvement with HM optimization. With the help of DataPlacer, we obtain the benchmark characteristics that may not benefit from HM.

- Small footprints. If the memory footprint is small, all data can be loaded into KeyStone II's L1 and L2 caches, so optimization does not help. For example, `cfd` has less than 1MB footprints that can fit into the hardware-managed L2 cache. Thus, the speedup for `cfd` is trivial.

Table 2.4: The analysis and optimization guidance provided by DataPlacer. The speedups are measured for all benchmarks running with eight threads in KeyStone II.

benchmarks	footprint (bytes)	accesses	#arrays in HM	Speedup over $B1$	Speedup over $B2$
`mtrans`	2.9E6	1.3E7	1	**11.51**×	**2.09**×
`mmulti`	3.7E8	6.4E8	1	**2.18**×	**1.85**×
`bfs`	2.0E6	1.4E8	6	**1.31**×	**1.18**×
`cfd`	6.0E5	1.2E7	2	1.05×	1.03×
`hotspot`	1.3E7	3.7E8	2	**1.17**×	**1.10**×
`kmeans`	6.7E7	1.4E10	1	1.01×	1.01×
`lavaMD`	3.3E6	3.7E6	3	1.01×	1.01×
`lud`	2.1E6	3.1E8	1	**3.95**×	1.00×
`nn`	2.1E6	2.8E9	1	**1.10**×	1.00×
nw	4.7E6	8.8E7	2	**1.51**×	**1.30**×
`particlefltr`	7.4E6	8.0E8	10	1.02×	1.00×
`pathfinder`	4.0E6	4.0E8	1	1.01×	1.06×
`srad`	1.0E7	1.2E9	5	**1.15**×	1.02×
Geo.mean	/	/	/	**1.56** ×	**1.17**×

- Streaming access patterns. If a benchmark has a streaming access pattern, loading data into HM does not benefit from many reuses. For example, `lavaMD` has a streaming access pattern ($fp \approx acc$); optimizing it shows nearly no speedup.

- Large footprint with a uniform access pattern. If all arrays in a benchmark are uniformly accessed, placing a small number of arrays in the fast HM layers does not significantly improve the performance. For example, `kmeans` has a large footprint with an uniform access pattern. Placing only a small subset of data into HM leads to nearly no speedup.

In addition to the performance, HM can improve program scalability. We evaluate strong scaling [1] of all benchmarks in HMBench. Most of the benchmarks have slightly better scalability when optimized with HM. The reason is that MSMC has much larger bandwidth than DDR in KeyStone II, so contentions in memory bandwidth can be re-

[1]Strong scaling means that the problem size is constant and the number of cores increases.

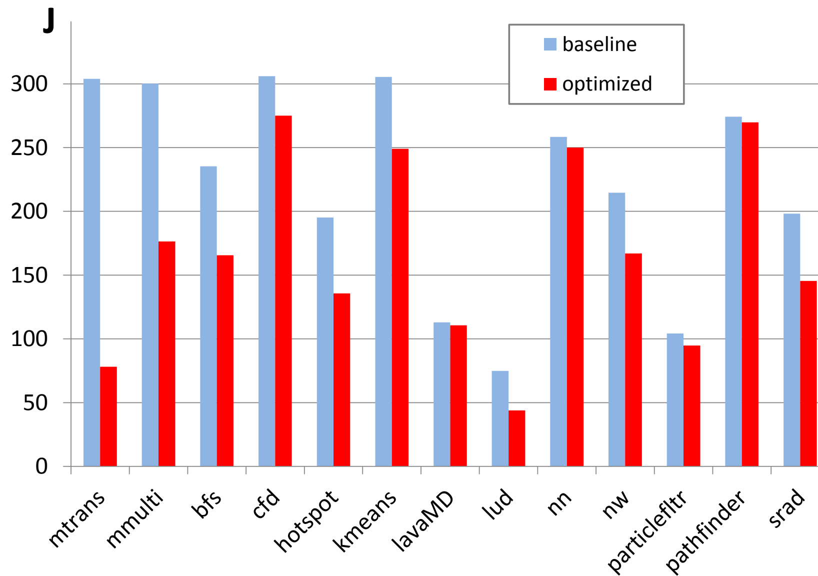


Figure 2.5: Comparison of whole-system energy consumption between baseline and optimized benchmarks running with eight threads. The vertical axis indicates the energy consumption, measured in Joules.

duced with the use of MSMC.

2.6.3 Power Characterization

We measure the power consumed by each application on a TI evaluation board that features one Keystone 66AK2H SoC, running with 1, 2, 4 and 8 DSP cores. Without a convenient way to tap into the power rails for the DSP and memory, we measure the board-level power consumption, by sampling the voltage and current with an external digital multimeter, Agilent 34450A. When the board is idle (no workload on CPU and DSP), we measure the board power as the baseline; when workload is being executed, we sample the board power. We repeatedly run each benchmark multiple times to minimize the measurement errors introduced by the system noise. We report the workload energy consumption by integrating the power over time.

Figure 2.5 compares the energy consumption of the whole system when running the original (without using MSMC) and optimized benchmarks. As shown in the figure, the

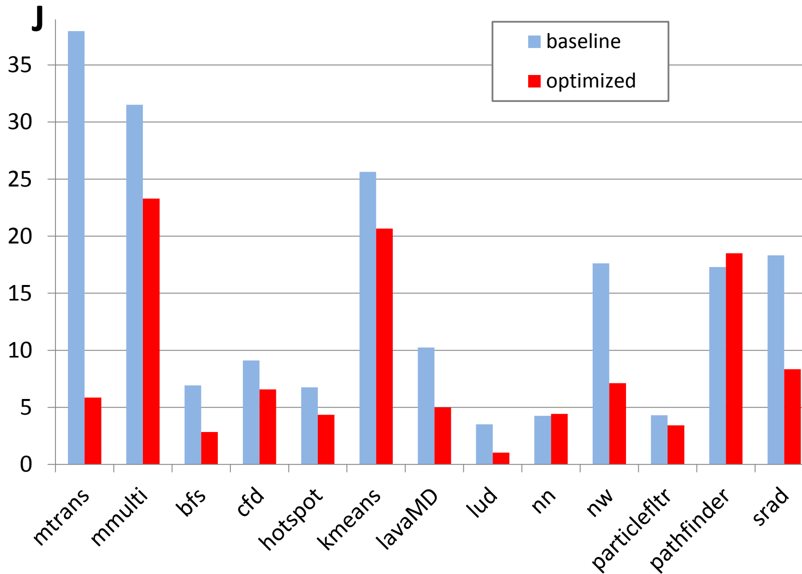


Figure 2.6: Comparison of dynamic energy consumption between baseline and optimized benchmarks running with eight threads. The vertical axis indicates the energy consumption, measured in Joules.

optimizations with HM for HMBench always reduce energy consumption. We can see that seven benchmarks have more than 20% energy reduction due to our HM optimization. It is worth noting that some benchmarks, such as `cfd`, `kmeans` and `particlefilter`, do not obtain speedups with the utilization of fast HM, but they obtain nontrivial energy reduction, 9-18%.

Due to the design limitation of the evaluation board, its static power is known to be much higher than that of a production device. To further highlight our efficiency benefit, we compare the dynamic energy consumption, which is computed as the difference between the measured energy and the baseline energy on chip. Figure 2.6 shows the measurement results: most of the benchmarks have significant reduction in energy consumption, more than $2\times$ on average. We further notice that `nn` and `pathfinder` after optimization consume more dynamic energy (Figure 2.6), but less overall energy (Figure 2.5). The reason is that the execution time reduction of these benchmarks saves a significant amount of static energy, which surpasses the dynamic energy increment.

2.6.4 Takeaways from Experimenting KeyStone II

With the evaluation of a benchmark suite running on a real system, KeyStone II, we identify that HM can benefit both performance and power consumption for many, but not all, applications. A performance tool, like DataPlacer, is necessary to guide the use of HM for the best performance.

However, we cannot further characterize HM's impact in performance and energy with hardware performance events on KeyStone II. Such hardware events include L1/L2 cache accesses/misses and MSMC accesses. The reason is that DSPs on KeyStone II lack of performance counters to record such events. Without this information, we cannot explain some phenomena. For example, we cannot directly understand DDR contention when scaling benchmarks to more DSPs. Moreover, we have no idea why nn and pathfinder consume more dynamic energy with HM optimizations. This work motivates TI to provide such support in DSPs to better understand their application behaviors.

2.7 Limitations with KeyStone II

Our study of HM based on KeyStone II has two limitations. First, the fast memory (L2 and MSMC) on KeyStone II has a small size (6 MB), so we need to tune the HMBench inputs with small sizes to make sure the fast memory can hold a sufficient portion of arrays to affect the performance. We tried large inputs of HMBench, which are difficult for us to obtain the performance gains. We expect 8-16 GB fast memory in the emerging architectures, where we foresee the benefit can be obtained from optimizing HMBench with large inputs.

Second, we lack the insights of the memory behavior in HM-based accelerators because there are no performance monitoring units (PMUs) in DSP. Thus, we cannot precisely explain why several HMBench benchmarks fail to obtain speedups with placing data in the fast memory. As the mainstream CPU architectures will employ HM in the

future, we expect to use CPU's PMU to collect rich information to understand the HM performance.

2.8 Chapter Summary

In conclusion, this chapter introduces HMBench and DataPlacer to study the impact of software-managed heterogeneous memory in a real system, the TI KeyStone II. HMBench is the first OpenMP benchmark suite that adopts OpenMP 4.0 standard and works on heterogeneous architectures. DataPlacer is a profiler to provide guidance for data placement in different layers of software-managed cache and memory. Using HMBench and DataPlacer, we observe the insight that HM plays an important role in both boosting performance and reducing energy consumption. Moreover, we leverage HMBench and DataPlacer to characterize the performance gains with HM.

Our future work is twofold. First, we will develop more benchmarks for HMBench to make it as the standard benchmark suite for evaluating HM-based systems and compilers. Second, we will extend DataPlacer to provide low-level guidance for compiler-based optimization for HM. Such low-level information includes the finer granularity of data placement on cache lines or pages, instead of arrays. We believe that optimizations on HM from both high-level source code transformation and low-level compiler-supported code generation can achieve the optimal performance.

Chapter 3

CUDAAdvisor: LLVM-Based Runtime Profiling for Modern GPUs

3.1 Introduction

General-purpose GPUs have been widely adopted in various computing domains, such as accelerating scientific computing applications, deep learning and graph workloads. From hardware perspective, 71 supercomputers in the top 500 list employ CPU+GPU heterogeneous architectures as of June 2017 [40]; among them, Piz Daint and Titan, both of which employ CPU-GPU architectures, rank third and fourth on the list, respectively. Moreover, due to the advancement of deep learning, NVIDIA released DGX-1 [41], a deep learning system consisting of eight Pascal GPUs. From software perspective, large packages, such as LAMMPS [42], TensorFlow [43] and Galois [44], have been leveraging modern GPUs to achieve superior performance.

Unlike CPUs, GPUs typically offer a relatively more complex programming and architectural scenario. For instance, they employ thousands of threads, which are divided into warps. With the *Single-Instruction Multiple-Threads* (SIMT) programming model, all the threads in one warp share the same program counter. Moreover, a warp is able to coalesce multiple memory requests to adjacent memory words into one single request, so threads can benefit from spatial locality. Caches on GPUs are often very limited in

capacity and they are shared across threads. Typically, programmers have to offload kernels to GPU (e.g., NVIDIA GPUs) to benefit from its high parallelism.

Efficiently designing a GPU kernel is difficult, especially when using low-level programming models, such as CUDA [1] and OpenCL [2]. Thus, it is not uncommon to come across performance bottlenecks that prevent the code from achieving high performance on GPUs. There are multiple unique types of challenges for GPU performance optimization. First, as GPU uses the SIMT programming model, control flow divergence may hurt parallelism. Since threads in different branch paths have the same program counter, they need to serialize between each other. Second, due to memory divergence caused by irregular or strided memory access patterns, GPU performance can be significantly degraded. Third, given the limited cache size on GPU, a large number of threads can easily compete for cache resources without efficient cache management strategies. Finally, optimizations at the level of intra- and inter-CTA [45] cannot be easily conducted without some clear guidance, especially when dynamic parallelism [46] is involved.

Manually analyzing these performance bottlenecks is tedious, error-prone and sometimes impossible for large code bases. Thus, one usually uses performance profilers to guide code optimization. For example, many CPU profilers have been proposed, such as Intel VTune [47], Oracle Solaris Studio [48], HPCToolkit [49] and gprof [50]. However, these CPU performance tools cannot directly profile GPU kernels. Existing GPU profilers, such as NVProf [51], TAU [52], and G-HPCToolkit [53], perform coarse-grained analysis for GPU kernels with relatively fixed metrics. These tools leverage CUPTI [54] interface available in NVIDIA GPUs to obtain the callbacks upon kernel launches and returns. They enable hardware performance counters on GPUs at the kernel launch point, record the occurrence of performance events during the kernel execution, and associate the events with the kernel after the kernel execution. These performance events include cache misses, memory divergence, and branch divergence. However, these coarse-grained analyses associate performance metrics with GPU kernels, lacking insights into kernel's instructions, loops, or functions. Recent NVIDIA Maxwell and

its later GPU generations support PC sampling [55], which samples instructions in a round-robin fashion and provides various stall reasons. However, PC sampling only provides sparse instruction-level insights.

To perform fine-grained analysis inside a GPU kernel, one needs to rely on GPU simulators (e.g. GPGPU-Sim [56]) or emulators (e.g., Ocelot [57]). However, simulation and emulation usually incur high overhead and are complex to develop. Moreover, they may suffer from compatibility issues related to the latest application and runtime features. Since they do not simulate or emulate every feature of the state-of-the-art GPU architectures, the optimization guidance generated may not apply to contemporary GPU hardware. Therefore, there is a high demand for fine-grained profilers that monitor kernel execution on real GPU architectures.

To support fine-grained profiling, NVIDIA released a research prototype named SASSI [58] —a tool that instruments GPU codes to support fine-grained analysis. However, as a close-source tool, SASSI has several limitations in practice, in terms of portability (i.e., not portable across CUDA runtime and architectures), expansibility (i.e., instrumentation engine is not open sourced), complexity (i.e., implementation level can be too low for common developers) and coverage (i.e., overlooks the interaction between CPU host and GPU). To address all these challenges, we present CUDAAdvisor, a fine-grained profiling framework that works on real GPU architectures across different CUDA versions. CUDAAdvisor leverages LLVM infrastructure to instrument a CUDA program for both its CPU and GPU code. Moreover, CUDAAdvisor collects performance data during CUDA program execution, associates the performance data with CPU and GPU behaviors and interactions, and derives useful metrics to guide various optimization techniques.

Contributions. In summary, we make the following contributions in CUDAAdvisor:

- CUDAAdvisor is the first fine-grained GPU profiler that works across generations of modern NVIDIA GPU architectures and CUDA versions, to the best of our knowledge.
- CUDAAdvisor combines the code- and data-centric profiling results from both CPU

and GPU and associates performance bottlenecks with their root causes.

- We also demonstrate CUDAAdvisor is able to combine different analyses and derive useful metrics and insights to guide optimizations (e.g., cache bypassing).

We evaluate CUDAAdvisor on two GPU platforms to demonstrate portability: NVIDIA Tesla K40c (Kepler architecture) with CUDA runtime 7.0 and NVIDIA Tesla P100 (Pascal architecture) with CUDA 8.0. By applying CUDAAdvisor to a number of commonly-used GPU applications, we show that CUDAAdvisor can successfully associate performance bottlenecks with program source code and understand their provenience. To showcase the optimization scenarios, we also perform software-level horizontal cache bypassing under the guidance of CUDAAdvisor, which yields speedup as high as $2\times$.

3.2 Existing GPU Profilers and Limitations

In this section, we elaborate on the most related work—SASSI, the state-of-the-art fine-grained profiling framework—and distinguish our approach. SASSI is a research prototype from NVIDIA research group, which is implemented as a pass in NVIDIA's backend compiler ptxas. SASSI selectively inserts instrumentation code to monitor the execution of CUDA kernels. SASSI can instrument instructions and functions. Moreover, SASSI is able to read the values residing in memory location and registers. As demonstrated in the chapter [58], SASSI supports to build effective fine-grained profilers. However, SASSI has several limitations in its portability, expansibility, complexity, and coverage.

- *Portability.* As SASSI is based on the CUDA backend compiler `ptxas`, it requires substantial efforts from NVIDIA to support SASSI with the rapid evolution of CUDA runtime and GPU architectures. SASSI currently does not work for CUDA runtime 8.0. In contrast, CUDAAdvisor is based on LLVM for code instrumentation, which can be generally applied to all modern NVIDIA GPUs and CUDA versions.

- *Expansibility.* As SASSI's instrumentation engine is close source, tool developers can-

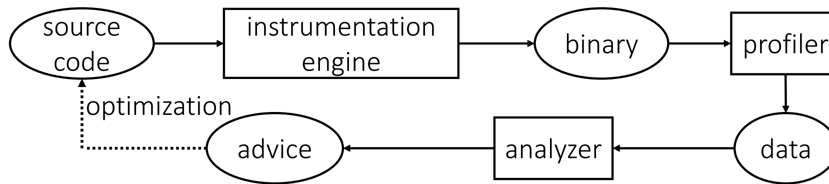


Figure 3.1: Workflow of CUDAAdvisor.

not add any new capability in SASSI. In contrast, CUDAAdvisor's instrumentation engine is based on LLVM, which is open source. Tool developers are able to extend CUDAAdvisor.

- *Complexity.* SASSI performs instrumentation during the translation from PTX code to a lower internal code representation for CUDA. Tool developers need to gain a deep knowledge of PTX to develop efficient profilers. To increase productivity, CUDAAdvisor instruments at bitcode level, which hides all the details in CUDA.

- *Coverage.* SASSI instruments GPU kernels only, which overlooks the interaction between CPU and GPU. Instead, CUDAAdvisor instruments both CPU and GPU code to analyze its interactions.

3.3 CUDAAdvisor Methodology

In this section, we introduce CUDAAdvisor's implementation methodology. Figure 4.2 shows the workflow of CUDAAdvisor, which consists of three components: instrumentation engine, profiler, and analyzer. CUDAAdvisor's instrumentation engine accepts the source code of CUDA program. It performs code transformation and leverages CUDA compiler to produce the binary code. CUDAAdvisor's profiler then collects the data that represent the behavior of the binary code during its execution on a real GPU hardware. Finally, CUDAAdvisor's analyzer analyzes the profiling data and generates optimization advice with source code attribution. Programmers can follow the advice to optimize the source code and begin another round of analysis if necessary.

In this section, we elaborate on the design and implementation of each CUDAAdvisor component. In the end, we point out some limitations of CUDAAdvisor. CUDAAdvisor is available at `https://github.com/sderek/CUDAAdvisor.git`.

3.3.1 CUDAAdvisor Instrumentation Engine

The task of the instrumentation engine is to add necessary instrumentation to CUDA code. We build CUDAAdvisor's instrumentation engine on top of LLVM framework [59] for three reasons. First, LLVM is a general compiler infrastructure that works on both CPU and GPU codes. Second, LLVM works across different GPU architectures and CUDA versions. Third, LLVM is robust, even for complex HPC programs. Figure 3.2 shows the positions of the instrumentation engine in the whole compilation workflow. As shown in the figure, LLVM frontend—`Clang` can translate the source code on both host (CPU) and device (GPU) into bitcode, the LLVM's intermediate representation. `Clang` also supports CUDA compiling (`gpucc`) [60]. Then, the instrumentation engine, implemented as an LLVM pass, works on both host and device bitcodes. After instrumentation in the device bitcode, LLVM uses specific backend [61] to translate the instrumented bitcode into the CUDA PTX intermediate code. The native CUDA assembler assembles the PTX code into a fat binary (i.e., a file with .fatbin extension), which is then inserted to the host-side CPU bitcode as a string literal. Finally, LLVM compiles the host bitcode into a binary executable.

The engine inserts mandatory instrumentation and provides an interface to add optional instrumentation. We describe these two kinds of instrumentations as follows:

(I) Mandatory instrumentation CUDAAdvisor's engine inserts mandatory instrumentations since CUDAAdvisor always reconstructs the call path and data flow in the profiling component, which will be shown in Section 3.3.2. To collect necessary performance data for the profiling, CUDAAdvisor's engine mandatorily instruments calls and returns for CPU functions, as well as for GPU kernels. Moreover, it instruments functions that

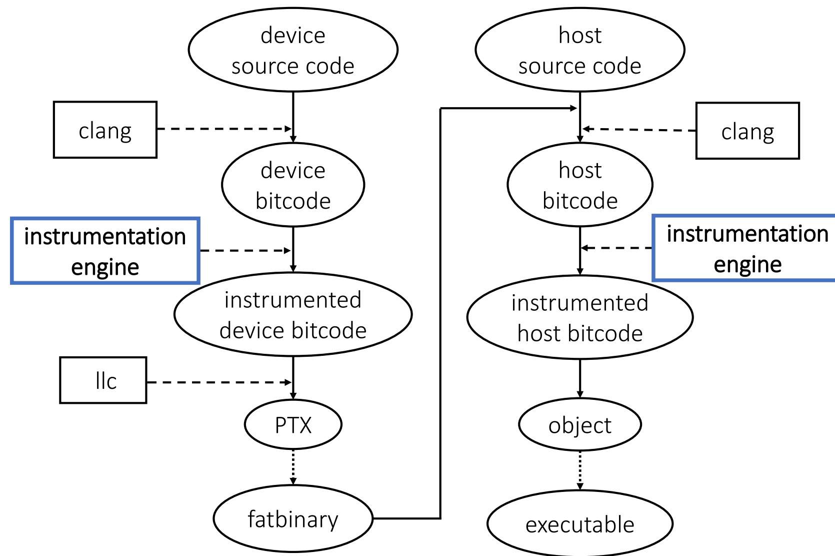


Figure 3.2: The workflow of the engine inserting instrumentation.

allocate memory in CPU code (e.g., malloc, calloc, realloc), in GPU code (e.g., cudaMalloc), and CPU-GPU data transfer functions (e.g., cudaMemcpy). At each instrumentation site, the engine inserts a function and passes the information as arguments of each function. For the memory allocations, the arguments include the starting addresses and the number of bytes allocated on CPU or GPU; for data transfers, the arguments include the starting addresses of memory ranges on both CPU and GPU, as well as the amount of bytes for the transfer.

(II) Optional instrumentation CUDAAdvisor's engine provides the capability of inserting optional instrumentations to support different analyses. Currently, the engine supports optional instrumentation in three categories.

- Memory operations. The engine can instrument every memory read and write and obtain the effective memory address accessed by this operation and its access width.
- Arithmetic operations. The engine can instrument every arithmetic computation and obtain the operator and the (symbolic) values of the operands.

- Control flow operations. The engine can instrument every control flow instruction, such as conditional or unconditional branches, and record their targets.

Common to all these mandatory and optional instrumentation functions, the engine is able to obtain the source code correlation of each instrumented operation, including the file name, line number and column number if available in the debugging information. This information is also passed to the instrumentation function as arguments.

In Section 3.3.2 and 3.4.2, we introduce the details of the profiling methods that are based on mandatory and optional instrumentation, respectively.

3.3.2 CUDAAdvisor Profiler

CUDAAdvisor's profiler divides its task into two stages: (1) the data collection during the CUDA kernel execution, and (2) the data attribution at the end of each CUDA kernel instance. Combining these two stages, CUDAAdvisor performs both code- and data-centric analyses on the fly, which provide the basic infrastructure for code optimization guidance.

3.3.2.1 Code-centric Profiling

CUDAAdvisor maintains a shadow stack to mirror the execution stack of each thread when the kernel runs on GPU. The profiler pushes the call site onto the shadow stack in the instrumented function at every call instruction, and pops the call site from the shadow stack in the instrumented function at every return instruction. By immediately querying the shadow stack, CUDAAdvisor is able to obtain the call path for each monitored instruction. For efficient analysis, CUDAAdvisor encodes each function with a unique ID number and a call stack is represented as an array of IDs. All GPU threads share the same encoding map from the number to function name and source code; the map resides in GPU global memory. To scale the analysis, each GPU thread maintains its own shadow stack; the shadow stacks are also in GPU's global memory. Upon kernel return, CUDAAdvisor copies all the data from GPU to CPU for further analysis.

On the CPU side, CUDAAdvisor maintains similar shadow stacks for CPU threads, which are used to determine the call stack for the invocation of each CUDA kernel. CUDAAdvisor concatenates this CPU call path with the ones collected inside the GPU kernel instance to give a complete path from the `main` function to each monitored CUDA instruction. We call this capability of CUDAAdvisor as code-centric profiling.

3.3.2.2 Data-centric Profiling

CUDAAdvisor's data-centric profiling reconstructs the data flow from CPU to GPU to help understand the access patterns of a data object across CPU and GPU, or even across different GPU kernels. Figure 3.3 highlights the behavior of CUDAAdvisor's profiler for data-centric analysis. The data flow of one data object starts at the beginning of its lifetime and ends in the memory accesses in GPU kernels that reference this data object. Typically, a data object is allocated on the CPU side dynamically or statically. The profiler interprets the malloc family functions (which are already instrumented by the engine) for dynamic allocation and reads the symbol table for static allocation. The profiler maintains a map that records the allocation call path for dynamic data objects and names for static data objects, and their allocated memory ranges. Similarly, CUDAAdvisor's profiler captures the data object allocation on the GPU side and keeps these data objects in another map. To correlate the two maps, CUDAAdvisor overloads the `memcpy` family functions and captures the two memory ranges involved in the memory copy.

With the two maps ready, CUDAAdvisor's profiler associates the memory accesses in GPU kernels with the data allocation. As the memory instrumentation obtains the effective address accessed by each memory read or write, the profiler utilizes this effective address to associate the memory access with data object. Since all data allocations and transfers are invoked on the CPU side, the profiler performs data-centric attribution at the end of each kernel instance after the memory access traces are copied back to the host CPU. After the data-centric profiling, CUDAAdvisor is able to construct a

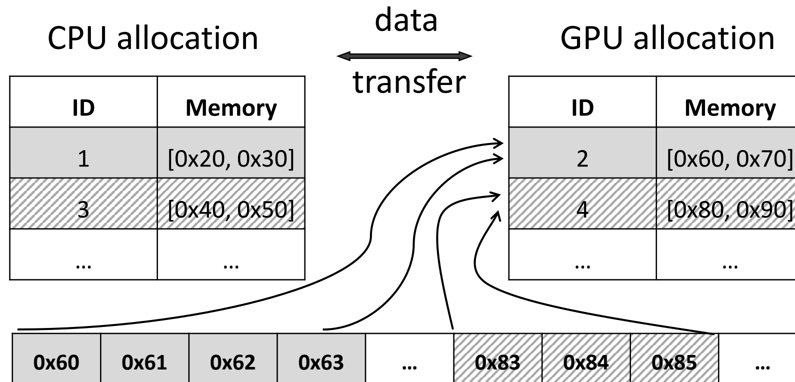


Figure 3.3: CUDAAdvisor's data-centric profiling.

complete data flow from every data allocation to its accesses, which helps observe interesting memory behaviors, such as the change of access patterns of the same data object across different GPU kernels.

3.3.2.3 Profiling Outputs

To minimize the overhead, CUDAAdvisor's profiler does not perform any analysis other than the code- and data-centric attribution. At the end of each kernel instance, the profiler copies all the performance data from GPU to CPU for further analysis. In Section 3.4, we elaborate on the implementations with case studies.

3.3.3 CUDAAdvisor Analyzer

CUDAAdvisor's analyzer has an online component that is invoked at the end of each kernel instance. It performs the analysis on the data collected by the profiler. The analyzer can be customized for different purposes. In Section 3.4, we show how we devise the analysis for reuse distance, memory divergence and branch divergence. Moreover, CUDAAdvisor's analyzer has an offline component that merges the analysis results of kernel instances in the same call path. It provides an aggregate statistical view, such as mean, min, max, and standard deviation across all these instances. Such statistical analysis demonstrates the performance variation across different instances of the same

Table 3.1: GPU architectures for evaluation.

Architecture	GPU	CC.	CUDA	Driver	Host CPU
Kepler	Tesla K40c	3.5	7.0	361.93	Intel Xeon E5-2650
Pascal	Tesla P100	6.0	8.0	375.20	Intel Xeon E5-2698

GPU kernel and provides intuitive guidance for performance optimization.

3.3.4 Limitations of CUDAAdvisor

CUDAAdvisor has a few limitations. First, similar to other runtime profilers such as SASSI, it incurs relatively high overhead (but much lower than simulators), which is caused by heavyweight fine-grained instrumentation to CPU and GPU instructions. This high overhead may disturb a program execution. However, the goal of CUDAAdvisor is not to capture the abnormal behavior of hardware-software interactions, but to formulate software metrics to identify software inefficiencies. Thus, CUDAAdvisor's overhead does not affect the accuracy of the framework. Moreover, it can capture detailed execution behavior, while no existing coarse-grained tools such as NVProf [51] and HPCToolkit [49] can. Second, CUDAAdvisor is based on LLVM, which requires the availability and recompilation of the source code of a monitored program. In the HPC community, the source code is often available and the recompilation time is much less than the execution time. Thus, the benefit of code optimization surpasses the extra work of recompilation. Third, the performance analysis is based on the code generation of LLVM, not other GPU compilers, such as `nvcc`. We will show that CUDAAdvisor's optimization guidance also works for CUDA codes with other compilers. Finally, since CUDAAdvisor is implemented at the bitcode level, it cannot profile register-related stats since NVIDIA has not released its assembly layer to the public.

3.4 Evaluation

In this section, we present use cases of CUDAAdvisor on detailed memory and control flow analysis, using a group of GPU applications. We also show two examples where

Table 3.2: Benchmarks for showcasing CUDAAdvisor.

Application	Description	warps/CTA	Input dataset	Source
backprop	Back Propagation	8	65536	Rodinia[62]
bfs	Breadth First Search	16	graph1MW_6.txt	Rodinia[62]
hotspot	Temperature Simulation	8	temp_512 power_512	Rodinia[62]
lavaMD	Molecular Dynamics	4	-boxes1d 10	Rodinia[62]
nn	Nearest Neighbor	8	filelist_4 -r 5 -lat30 -lng90	Rodinia[62]
nw	Needleman-Wunsch	1	2048-10	Rodinia[62]
srad_v2	Speckle Reducing Anisotropic Diffusion	8	2048-2048-0-127-0.5-2	Rodinia[62]
bicg	BiCGStab Linear Solver	8	1024*1024	Polybench[63]
syrk	Symmetric Rank-K Operations	8	default	Polybench[63]
syr2k	Symmetric Rank-2K Operations	8	default	Polybench[63]

CUDAAdvisor provides guidance for code optimization.

3.4.1 Evaluation Methodology

Evaluation Environment. We evaluate CUDAAdvisor on two different architectures, which are summarized in Table 3.1. Both host architectures are Intel Xeon CPUs with gcc 4.8.4 and installed with LLVM 5.0. For GPUs, we select an NVIDIA Kepler architecture because of its mainstream adaptation. Kepler's L1 cache shares the same on-chip storage with the shared memory (i.e., scratchpad memory) on each SM. Their sizes are configurable, 16/48 KB, 32/32 KB or 48/16 KB for Kepler. For demonstrating portability, we also evaluate CUDAAdvisor on the most recent NVIDIA Pascal architecture with CUDA 8.0. Pascal completely uses the entire on-chip storage for shared memory but it has a 24KB read-only L1/Texture unified cache. Note that although only Kepler and Pascal are tested in this chapter, CUDAAdvisor can be generally applied to all the other modern NVIDIA GPUs such as Fermi and Maxwell.

Benchmarks. Shown in Table 3.2, we showcase CUDAAdvisor using ten representative GPU applications, from Rodinia [62] and Polybench [63]. Both are among the most commonly-used open source CUDA benchmarks. They contain a wide range of applications that fall into various research categories. The selected applications in Table 3.2 are also used in the previous studies [45, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75].

3.4.2 Case Studies

In this subsection, we present the use cases of CUDAAdvisor on detailed memory system analysis and control flow analysis, both of which are critical performance bottlenecking factors for modern GPUs. To further showcase the application of our tool, we demonstrate how these insights extracted from CUDAAdvisor can be used to aid program debugging, steer performance optimizations and facilitate compiler/architecture research. Additionally, we provide instrumentation scenarios for these case studies to assist users.

(A) GPU Reuse Distance

Significance. Recent research [45, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75] has focused on on-chip cache locality optimization for overall performance and energy consumption since modern GPUs heavily rely on their on-chip memories (e.g., L1 and texture caches) to reduce the off-chip memory wall effects. One of the most widely applied approaches to analyze cache behavior is *reuse distance analysis* [76, 77]. At the surface level, reuse distance can generally reflect how good cache locality an application has under certain input. Combined with other detailed information such as memory divergence degree and MSHR (*miss-status holding-registers*) status, it can be used to help architects predict optimal cache design such as size and associativity. However, conducting this analysis can be quite complex due to GPU's parallel execution model and fine-grained massive multi-threading. Conventional approaches resort to either a GPU simulator such as GPGPU-Sim, which often uses trimmed down input sizes due to time-consuming execution, or a GPU emulator such as Ocelot [57] which only provides unordered lists of memory accesses rather than ordered traces that can be obtained from simulators (e.g., [76] has to enable its own assumed warp and CTA scheduler on top of Ocelot to produce usable traces). On the contrary, our CUDAAdvisor enables much faster and convenient runtime profiling of reuse distance across generations of NVIDIA GPU architectures.

CUDAAdvisor Instrumentation. We instruct CUDAAdvisor to instrument at all the

```
1  virtual bool runOnBasicBlock(Function::iterator &B) {
2    for(BasicBlock::iterator BI = B->begin(), BE = B->end(); BI != BE;
        ++BI) {
3      if ( auto *op = dyn_cast<LoadInst>(&(*BI))) {
4        /* get loc info */
5        const DebugLoc &loc = BI->getDebugLoc();
6        int line =  loc.getLine();
7        int col =  loc.getCol();
8
9        /* get effective address */
10       Value* addr = op->getPointerOperand();
11       Value* ptr = builder.CreatePointerCast(addr, Type::getInt8PtrTy
            (C) );
12
13       /* get number of bits */
14       Type* tp = CI->getType();
15       int sizebits=(int)tp->getPrimitiveSizeInBits();
16
17       /* insert a function call */
18       IRBuilder<> builder(op);
19       builder.CreateCall(hook, {ptr,   builder.getInt32(sizebits),
            builder.getInt32(line), builder.getInt32(col), 1});
20         }
21     }
22 }
```

Listing 3.1: The LLVM pass to instrument memory accesses.

global memory operations. At each instrumentation site, CUDAAdvisor collects some specific information and passes them as arguments to the *analysis function*, such as effective address, number of bits accessed, source code location, etc. As discussed previously, CUDAAdvisor leverages a LLVM pass to conduct instrumentation at the bit-code level. Shown in Listing 3.1, LLVM parses the bitcode (Line 1 and 3) and instruments every global load instruction (Line 5). It then extracts the source code line and column from the debugging information (Line 7-9), obtains the effective address (Line 11-12) and the number of bits (Line 14-15). Finally, the pass creates a function call to a predefined analysis function and passes all these arguments (Line 18). Note that global stores and shared/constant/texture/read-only accesses can be profiled in a similar fashion.

Listing 3.2 shows a snippet of instrumented bitcode. Line 1 is the original code, which loads a float from address %a. Line 2 and 3 are inserted by CUDAAdvisor. Line 2 converts the pointer from float (float*) into a general pointer (i8*), while Line 3 calls

```
1 %3 = load float, float* %a, align 4,  !dbg !1127
2 %4 = bitcast float* %a to i8*, !dbg !1127
3 call void @Record(i8* %4, i32 32, i32 20, i32 13, i32 1), !dbg !1127
```

Listing 3.2: Memory access instrumented bitcode.

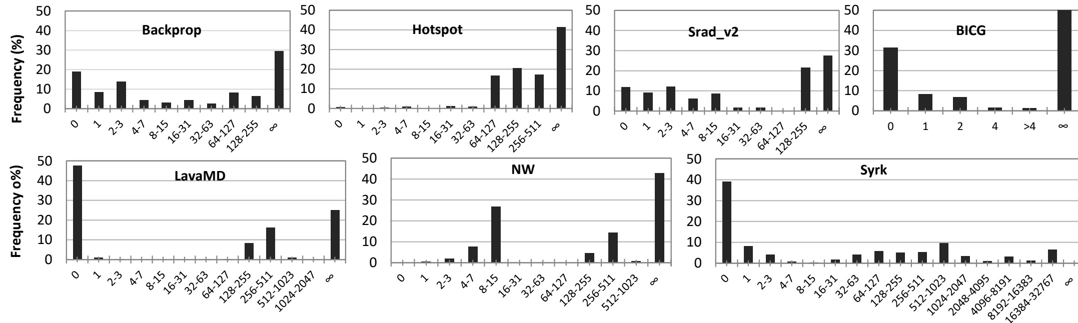


Figure 3.4: Reuse distance analysis through CUDAAdvisor. ∞ is defined as data is never reused again during the program execution or before the next write to the address (e.g., write-evict L1 on NVIDIA GPUs).

analysis function *Record()*, and passes arguments as the effective address, number of bits accessed, the source code line and column, and operation type. *Record()* packs all the arguments along with CTA ID and thread ID into one entry. Entries from all memory accesses form a trace. CUDAAdvisor stores this trace in a buffer located in GPU's global memory. This analysis function is a __device__ function so that it is callable by device. It is written in a separate CUDA source file and compiled into a separate bitcode file before being merged with the kernel bitcode by llvm-link. For data marshaling, CUDAAdvisor initiates data transfer using `cudaMemcpy` and copies collected trace from device to host (e.g., can be accomplished through Unified Memory supported by CUDA 6 and beyond). To calculate reuse distance for each CTA, the trace is first regrouped into multiple traces based on their associated CTA IDs, which is then used by CUDAAdvisor. CUDAAdvisor offers two reuse distance model: memory element based and cache line based. In addition to reuse, CUDAAdvisor also records the number of streaming accesses (i.e., accesses to memory elements that are never reused by the same CTA).

Results and Analysis. Throughout the chapter, we refer to reuse distance as the traditional definition of data reuse distance. From the view of memory and cache, a sequential execution of a program is a sequence of data access [77]. Given such a se-

quence, we define reuse distance as the number of distinctive data elements accessed between two consecutive uses of the same element. Under such definition, reuse distance directly reflects data temporal reuse. For instance, $ABCCDEFAAAB$ is a data access sequence. The reuse distance of B is 5. For better facilitating the later discussion on GPU L1 cache-level optimization, we slightly tweak its definition: once an address A is written, we will restart its reuse distance counting as another address 'A'' because GPU L1 cache follows write-no-allocate write-evict policy [68, 72]. Since reuse distance is an inherent program property and independent of cache parameters or underlying machines, we perform reuse distance analysis with CUDAAdvisor on Kepler architecture only. We evaluated ten applications from Table 3.2. Seven of them are shown in Figure 3.4. `BFS` and `NN` are excluded because they exhibit very low reuse (more than 99% of the accesses), while `Syr2k` is excluded since it resembles `Syrk`. The x-axis represents the range of reuse distance. ∞ represents no-reuse, indicating that data is never reused in the program or before the next write to it. We have the following interesting observations: (1) `BFS` and `NN` exhibit very low reuse, both suffering from branch-heavy codes with little loads during execution. For example, $Kernel$ and $Kernel2$ with few loads to $some_array$ in `BFS`, and $euclid$ with few loads to $d_locations$ in `NN`. (2) Eight out of ten applications suffer from high no-reuse accesses (except for `Syrk` and `Syr2k`), which wastes the precious resources such as cache and MSHRs. Among them, `Hotspot` exhibits both long reuse distance and very high no-reuse, making it insensitive to L1 cache level optimizations, such as capacity increment and bypassing. The other seven cases incur accesses with both decent frequency of short reuse distance and high no-reuse, e.g., `BICG` and `LavaMD`. They present some level of sensitivity to L1 level optimization schemes. (3) `Syrk` and `Syr2k` both exhibit low no-reuse and higher frequency of short reuse distance (e.g., reuse distance 0's frequency is close to 40%). However, they also exhibit accesses with very long reuse distance (e.g., around 25% beyond 512), indicating that cache capacity likely affects the effectiveness of L1 level optimization schemes. In summary, CUDAAdvisor's reuse distance analysis paints

a general picture of an application's cache locality and provides important insights for selecting potential optimizations for different software/architectures. To better steer optimizations, reuse distance analysis can be combined with other information, such as memory divergence degree, register pressure and shared memory usage.

(B) Memory Divergence Frequency and Degree

Significance. GPU memory divergence can significantly bottleneck performance, thus becomes a popular research topic in recent years [78, 74, 66, 79, 80]. It is also an important indicator on whether a program is well optimized for memory access. Because of GPU's SIMT execution model, warp instructions execute in a lock-step. All the memory requests for a given instruction must be received before this warp can proceed. At architecture level, a coalescing unit has been added into the data path before reaching to L1 cache as a "best effort" approach for combining accesses to the same cache line into a single request for reducing off-chip memory access. However, memory divergence still occurs quite often in irregular kernels that touch many unique cache lines, causing performance degradation. Effectively profiling memory divergence frequency and degree for applications is essential for optimizations. Unlike production tools (e.g., NVprof [51]) that provides coarse-grained estimation and is limited to early generations of hardware, CUDAAdvisor provides programmers with fine-grained profiling for memory divergence at instruction level. It also flexibly reports summarized results, such as memory divergence degree.

CUDAAdvisor Instrumentation. CUDAAdvisor relies on memory traces to analyze GPU kernel's memory divergence characteristics. We instruct CUDAAdvisor to instrument at all the global memory *read* and *write* operations. Similar to reuse analysis, CUDAAdvisor collects effective address, number of bits accessed and source code location at instrumentation site, and then passes them to the analysis function. CUDAAdvisor leverages the same LLVM pass of Reuse Distance in Listing 3.1 for memory divergence profiling. Runtime traces are stored on GPU's global memory and copied to host for data marshaling. Analysis metrics such as memory divergence degree is modularized

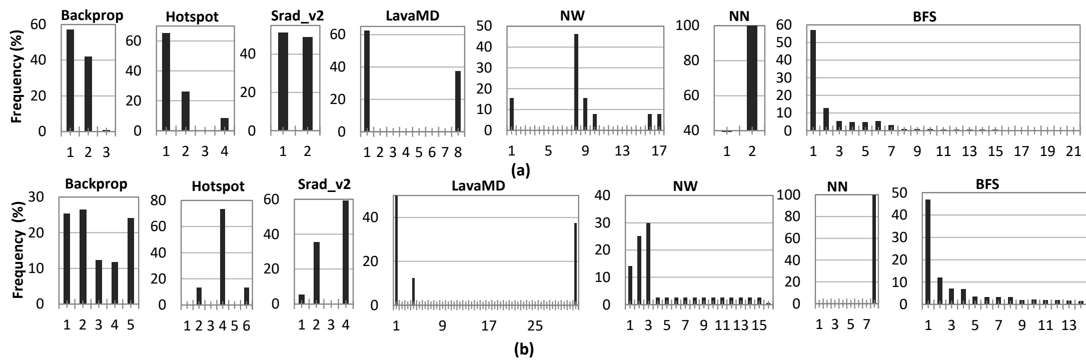


Figure 3.5: Profiled memory divergence distribution of unique touched cache lines by instructions of an entire application on Kepler. X-axis represents number of unique cache lines touched (min is one and max is 32). (a) Kepler architecture with 128 Byte cache line; (b) NVIDIA Tesla P100 (Pascal) with 32 Byte cache line.

in post-analysis.

Results and Analysis. Figure 3.5 shows the profiled memory divergence distribution for an entire application on Kepler (128 Byte cache line) and Pascal (32 Byte cache line) architectures in Table 3.1, which is calculated as the number of unique cache lines touched by each instruction. The selected applications are from Table 3.2 and the maximum range of x-axis is 32 for NVIDIA GPUs due to 32 threads/warp. Note that since the distribution figures for three applications including `BICG, Syrk and Syr2k` have mostly 1 and 32 cache lines touched, we do not show them in the figures to save space but report their numbers here: for Kepler architecture, `BICG` ($1 \Rightarrow$ 75%, $32 \Rightarrow$ 25%), `Syrk` ($1 \Rightarrow$ 50.02%, $32 \Rightarrow$ 49.98%) and Syr2k ($1 \Rightarrow$ 50%, $32 \Rightarrow$ 50%); for Pascal, `BICG` ($1 \Rightarrow$ 50%, $4 \Rightarrow$ 25%, $32 \Rightarrow$ 25%), Syrk ($1 \Rightarrow$ 49.98%, $32 \Rightarrow$ 49.98%) and `Syr2k` ($1 \Rightarrow$ 49.99%, $32 \Rightarrow$ 49.99%). Such distribution of unique cache lines touched reflects the general optimization degree on memory access for an application, or how well the memory access pattern is structured. For example, we can observe in Figure 3.5 that `Backprop, Hotspot and Srad_v2` have better code optimization for avoiding memory divergence than the others in the group. Also, architecture plays a role in deciding the distribution: the largest number of unique cache line touched in Pascal is generally larger than that on Kepler primarily due to cache line size. CUDAAdvisor also provides user interfaces for

```
1 virtual bool runOnBasicBlock(Function::iterator &B) {
2   /* construct an argument for basic block's name */
3   std::string bb_name = B->getName().str();
4   Value* str_bbname = builder.CreateGlobalStringPtr(bb_name);
5   Value* ptr_bbname = builder.CreatePointerCast(str_bbname, Type::
        getInt8PtrTy(C) );
6
7   /* fetch debug information */
8   const DebugLoc &loc = inst->getDebugLoc();
9   int ln = loc.getLine();
10  int cl = loc.getCol();
11
12  /* create a function call to analysis function */
13  builder.CreateCall(hookBB, {ptr_bbname, builder.getInt32(ln),
        builder.getInt32(cl) } );
14 }
```

Listing 3.3: Implementation of LLVM pass to instrument basic blocks.

profiling application's memory divergence degree, which is computed using the average of weighted sum of distribution for the number of unique cache lines touched. Memory divergence degree is also an important index for modeling GPU performance.

(C) Branch Divergence

Significance. In addition to the memory-level analysis capability discussed in (A) and (B), CUDAAdvisor also provides profiling for control flow analysis. Modern GPU programming models enables flexibility for programmers to add control flow in their codes. But conditional control flow can significantly affect warp efficiency and overall performance since it could cause threads in a warp to execute different instructions, which is called *branch divergence*. Branch divergence can be very harmful to GPU performance because a subset of threads in a warp will be deferred to a divergent stack until all the other threads finish executing their path. CUDAAdvisor can provide insights of the kernel's divergence, such as how many times a branch is executed, how many threads execute this branch and how often a certain branch causes a warp to diverge.

CUDAAdvisor Instrumentation. We instruct CUDAAdvisor to instrument at all basic block entries. At each instrumentation site, CUDAAdvisor collects the names of these basic blocks and extracts their source location from debug information before passing them to the *analysis function*. Similar to memory access instrumentation, CUDAAdvisor

```
1 /* string of basic block id */
2 @5 = private unnamed_addr constant [6 x i8] c"entry\\00"
3
4 /* one basic block in a certain function */
5 entry:
6   call void @passBasicBlock(i8* getelementptr inbounds ([6 x i8], [6
      x i8]* @5, i32 0, i32 0), i32 15, i32 36), !dbg !620
7   /* here starts the original instructions */
8   %0 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !dbg !625
```

Listing 3.4: Basic block instrumented bitcode.

Table 3.3: Results of Branch Divergence on Pascal.

Application	# divergent blocks	# total blocks	% divergence
backprop	26257	95011	27.64%
bfs	408420	1292788	31.59%
hotspot	1372	4197	32.69%
lavaMD	103	744	13.84%
nn	81	2001	4.05%
nw	147875	212992	69.43%
srad_v2	92643	270128	34.30%
bicg	0	1256	0.00%
syrk	0	817	0.00%
syr2k	15	393	3.82%

relies on an LLVM pass to instrument at bitcode level. As shown in Listing 3.3, the pass retrieves the name of each basic block and creates a pointer to the string where the name is stored (Line 4-6). It then fetches the source code locations from debug information (Line 9-11). Finally, it creates a function call to the analysis function and pass these arguments (Line 14). Listing 3.4 shows a snippet of instrumented bitcode. Line 5 and 8 are the original bitcodes and they reside in a certain function. Line 5 indicates a basic block named "entry" and Line 8 is the first instruction in this basic block. Line 2 and 6 are inserted by CUDAAdvisor. Line 2 is a global string which stores the basic block's name. Line 6 is a function call to the predefined analysis function $passBasicBlock()$ which takes in the basic block's name and source code location. $passBasicBlock()$ records arguments along with CTA ID and thread ID in a buffer. Similar to memory access instrumentation, $passBasicBlock()$ is written in a separate CUDA source file and merged into application's kernel at bitcode level. For data marshaling, the trace is transferred to CPU upon kernel exit.

Results and Analysis. Table 3.3 shows the profiling results on the percentage of divergent blocks in an application running on NVIDIA Pascal architecture. This result

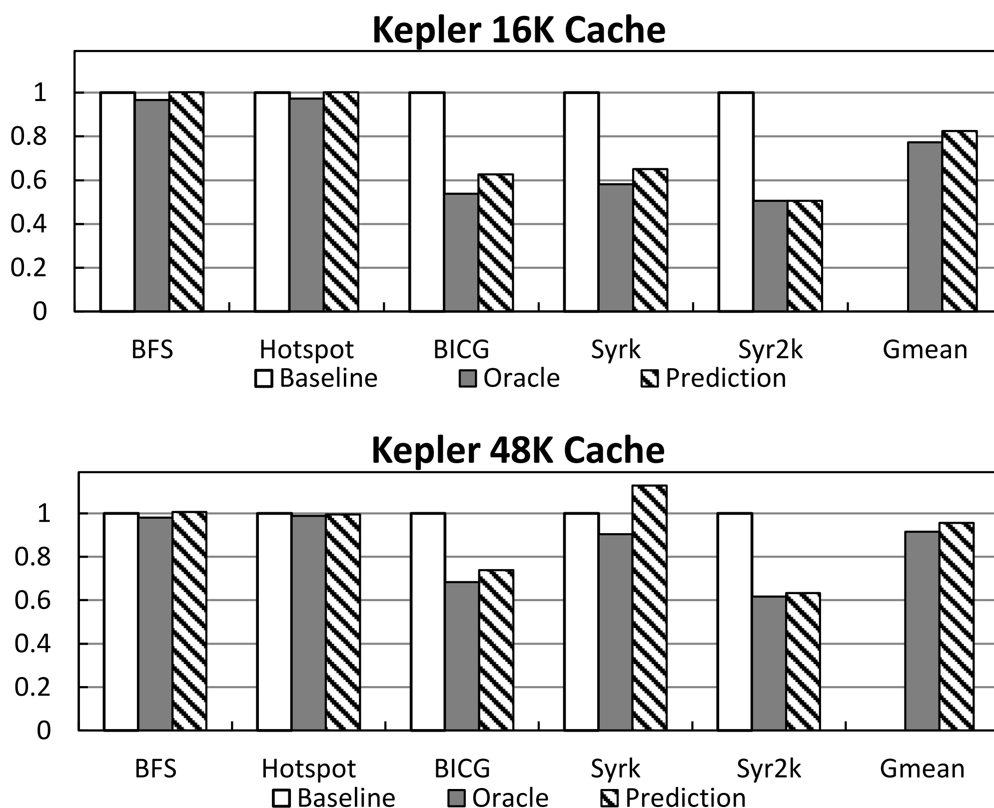


Figure 3.6: Normalized execution time of different applications on NVIDIA Kepler architecture when using the predicted optimal number of warps per CTA for bypassing. Baseline case is using all the warps (no bypassing). Oracle exhaustively searches the optimal solution. Prediction represents our model.

summary also applies to other NVIDIA GPUs since branch divergence under CUDA is independent of architectures. We can observe that `NN, BICG, Syrk and Syr2k` have very low frequency of branch divergence while the others (especially `NW`) suffer from high frequency of branch divergence. This analysis effectively helps programmers target applications that are in need of branch divergence optimizations, to which previous optimization techniques [81, 82, 83] can be applied.

(D) Optimization 1: Horizontal Cache Bypassing.

Recently, cache bypassing has become a heavily investigated research topic in GPU computing. This is because modern GPUs have very limited L1 data cache and massively threaded GPU applications often exceed the L1 capacity, causing severe thrash-

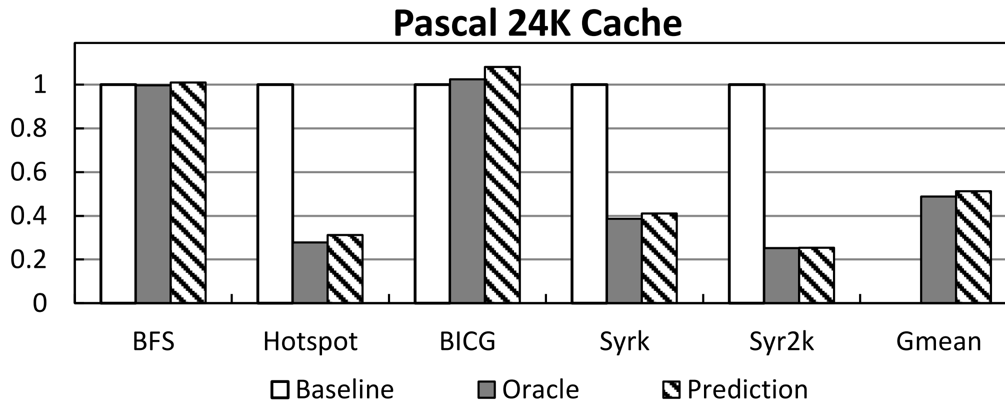


Figure 3.7: Normalized execution time of different applications on NVIDIA Pascal architecture when using the predicted optimal number of warps per CTA for bypassing. Baseline case is using all the warps (no bypassing). Oracle exhaustively searches the optimal solution. Prediction represents our model.

ing [69, 64]. Additionally, cache-level resources (e.g., MSHR entries and load/store queues) are also very limited, often causing severe resource congestion (e.g., MSHR allocation failures) [45, 74, 68]. To tackle this problem, many architecture solutions are provided, e.g., enabling bypassing threshold in tag store [68] and proposing new bypassing policy [67]. Two general types of software-level bypassing are also proposed: vertical bypassing [72] targeting on every global memory access instruction in PTX (bypassing them for every warp), and horizontal bypassing [67] which focuses on concurrency and only allows certain number of concurrent warps per CTA to access L1 cache. Each of them has certain advantages and disadvantages: vertical bypassing is more fine-grained but requires architectural and runtime information to evaluate every individual load and also bypass all the warps; horizontal bypassing is much simpler and can manage bypassing granularity better but cannot distinguish loads with little reuse. To

```
1 //=== compute warp id and set threshold ===
2 mov.u32 %r0, %tid.x; //Thread index
3 shr.u32 %r0, %r0, 5; //Warp index
4 setp.lt.s32 %p0, %r0s, $pi$; //Set Threshold
5
6 //=== for each global load ===
7 @%p0 ld.global.ca.s32 %r9, [%rd6]; //Cache
8 @!%p0 ld.global.cg.s32 %r9, [%rd6]; //Bypass
```

Listing 3.5: PTX instrumentation for horizontal bypassing.

showcase our tool's prediction capability, we choose the most recent software-level horizontal bypassing proposed in [67] to compare with, which uses a pre-execution sampling period to do exhaustive searching for the optimal number warps per CTA to access L1 on a SM and then follows such suggestion for the remaining execution. Although it can accurately identify the best warp number to access L1, it requires exhaustively searching all the options for number of warps. The actual bypassing in PTX is shown in Listing 3.5. Using the memory tracing functionality demonstrated in (A) reuse distance and (B) memory divergence from CUDAAdvisor, we can actually model the optimal number warps to access L1 along with other parameters provided by the application and architecture features, without the need of exhaustive searching. We build the optimal warp estimation model in Eq.(3.1), which can be built into post processing of CUDAAdvisor as an metric. In this model, $R.D.$ represents an application's average reuse distance, and $M.D.$ is the average memory divergence of the application; both can be calculated through the outputs of CUDAAdvisor. Note that for showcasing purpose we use the average value of $R.D.$ and $M.D.$ instead of eliminating the outliers (e.g., extreme data points) to rather conservatively estimate the optimal warp number.

$$Opt_Num_Warps = \frac{L1_Cache_Size}{R.D. * Cacheline_Size * M.D. * \#CTAs/SM} \tag{3.1}$$

We tested our model on the two GPU architectures shown in Table 3.1: Kepler with 16KB or 48KB L1 cache and Pascal with 24KB L1/Texture unified cache. We also select cache-bypassing favorable applications (based on the experience from recent simulation work [68]) from Table 3.2 to demonstrate our model's prediction accuracy for bypassing. Figure 3.6 and 3.7 show the prediction evaluation on Kepler and Pascal. The baseline case is the default scenario where cache bypassing is disabled (i.e., using cache), while oracle stands for the exhaustive horizontal bypassing in [67] and prediction represents our model. Both figures clearly show that our model achieves very good performance. It is 6.7% and 4.3% slower than the oracle scenario, for 16KB and 48KB L1 on Kepler, respectively. And for the 24KB unified cache on Pascal, our prediction

```
   // bfs.cu
57 int main( int argc, char** argv) {
     ...
63   BFSGraph( argc, argv);
     ...
   }

75 void BFSGraph( int argc, char** argv) {
     ...
172  cudaMalloc((void**) &d_graph_visited, bytes) ;
173  cudaMemcpy(d_graph_visited, h_graph_visited, bytes,
         cudaMemcpyHostToDevice) ;
     ...
217  Kernel<<<grid,threads,0>>>(..., d_graph_visited, ...);
     ...
   }

   // Kernel.cu
22 __global__ void Kernel(...,bool *g_graph_visited,...) {
     ...
33   if(!g_graph_visited[g_graph_edges[i]])
     ...
   }
```

Listing 3.6: Code Snippet of BFS.

is 5% slower. We also observe that these supposed bypassing-favorable applications actually are quite different. BFS and Hotspot are quite insensitive applications which match their streaming features discussed in Section 4.2-(A) and Figure 3.4. The other three who benefit from bypassing actually suffer greatly from capacity misses because increasing cache size from 16KB to 48KB dramatically reduces bypassing benefits (e.g., 23% to 9% for oracle on Kepler). Additionally, architecture features play an important role in bypassing. For instance, bypassing on Pascal performs better than that on Kepler because the unified cache on Pascal locates in texture processor cluster (TPC) instead of SM (i.e., multiple SMs locate on one TPC so the unified cache technically locates between SM and NoC instead of on SM). All these detailed analysis on cache bypassing optimization of real GPU hardware can be easily obtained and modeled through our CUDAAdvisor tool.

(E) Optimization 2: Code- and Data-Centric Debugging.

When understanding or debugging a large project, it can be cumbersome and error-

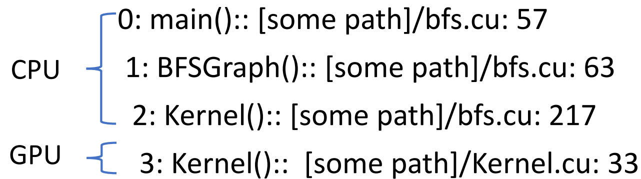


Figure 3.8: Code-centric view shows concatenated calling context from both host and device.

prone to read the code and track all data objects. As discussed in Section 3.3.2, a major contribution of CUDAAdvisor is to provide code-centric and data-centric profiling to show insights of the program and guide debugging. To showcase these features, we take `BFS` as an example. A code snippet is shown in Listing 3.6. Since `BFS`'s kernel uses data types of bool and float, a warp can ideally touch only one cache line on Kepler (128 Byte cache line) and up to four cache lines on Pascal (32 Byte cache line), assuming the program has no memory divergence. However, `BFS` has a portion of memory accesses that touch more than the limits, as previously shown in Figure 3.5.

If a programmer is interested to know which memory accesses suffer from memory divergence, CUDAAdvisor can show not only the source code location, but also the calling context. Figure 3.8 is an illustrative example. Each rows lists the index, the function name, the source file and line number. This example shows that Line 33 of Kernel.cu has significant memory divergence. As can be seen in the figure, CUDAAdvisor concatenates the call path from both host and device to show the calling context starting from main function on host all the way to the suspicious site on device, to better guide programmers to understand the program's behavior. Note that CUDAAdvisor is able to capture and display function calls in CUDA kernel as well.

CUDAAdvisor also detects which data object is associated with memory divergence. An illustrative output is shown Figure 3.9. CUDAAdvisor shows the calling context to malloc(), cudaMalloc() and cudaMemcpy(). It shows programmer that an array of bool `d_graph_visited` allocated at Line 172 of *bfs.cu* suffers from memory divergence, and that its counterpart on host is `h_graph_visited` allocated at Line 113 of *bfs.cu*.

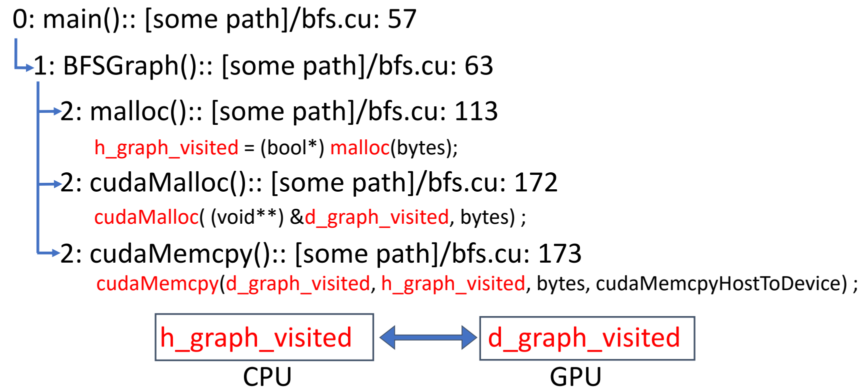


Figure 3.9: Data-centric view shows the interesting data objects, where it is allocated on host and device and where it is transferred.

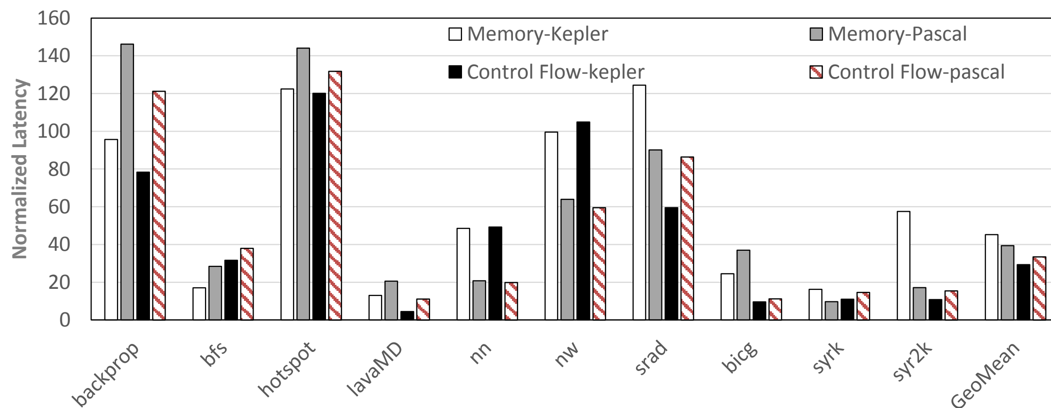


Figure 3.10: Overhead of memory and control flow instrumentation, on Kepler and Pascal Architectures.

These features of CUDAAdvisor can significantly reduce debugging time for a fairly large project.

3.5 Tool's Overhead Analysis

Figure 3.10 shows the runtime overheads of running GPU kernels for each benchmark listed in Table 3.2. The baseline used is the original benchmark compiled by Clang 5.0. We perform overhead analysis on Kepler and Pascal architectures, with detailed information in Table 3.1. The experimental results are averaged on five runs. As shown

in Figure 3.10, the runtime overhead mostly ranges from $10\times$ to $120\times$. It is much faster than simulators such as GPGPU-Sim that usually incurs 1-10 millions of slow down to the native execution [58].

With further analysis, the overhead mostly comes from three sources. First, CUDAAdvisor utilizes atomic operations to serialize memory access or control flow events. Second, CUDAAdvisor inserts a function call to each instrumentation site. In the future, we will explore a more efficient way to insert instructions rather than heavyweight function calls. Third, the buffer on the device side lives in GPU's global memory. It competes the GPU kernel with shared resources such as cache and MSHR.

3.6 Related Work

We have distinguished CUDAAdvisor from the most related work—SASSI [58] in Section 3.2. In this section, we review other profilers that work on GPUs.

NVIDIA provides its own tools to support profiling CUDA code, such as Visual Profiler (NVP) [16], nvprof [51], and NSight [84]. These profilers collect performance data via hardware performance counters and lightweight binary instrumentation. They are able to identify inefficient CPU-GPU interactions and pinpoint performance bottlenecks in CUDA code. However, these production-quality tools are not open sourced and very inflexible for novel research exploration and various analysis since they only provide very limited pre-selected metrics. For example, unlike CUDAAdvisor, these tools do not provide intuitive optimization guidance with cache bypassing, detailed reuse distance analysis, and memory divergence distribution frequency.

In the HPC community, several profilers that can support coarse-grained GPU-level analysis have been proposed, including Vampire [85], TAU [52], Scalasca [86], G-HPCToolkit [53] and [87]. They collect data via CUPTI tool [54] and hardware performance counters available on GPUs and associate these data with GPU kernels and timestamps. These tools usually incur small overhead and provide insights into prob-

lematic CPU-GPU interactions with profiles and traces. However, they fail to identify detailed performance insights into GPU kernels and lack the root cause analysis on performance bottlenecks inside CUDA kernels.

In the architecture community, GPGPU-sim [56] is a widely-used simulator to perform fine-grained simulation tracing. However, compared to runtime profilers, simulation is very slow, preventing it from working on real inputs. Moreover, simulation may not simulate all the features of each hardware component, so the analysis result may not reflect the real execution on newer GPU architectures (i.e., GPGPU-sim only models Fermi architecture).

3.7 Chapter Summary

This chapter presents CUDAAdvisor, a general framework that supports fine-grained analysis to identify performance bottlenecks of CUDA code. CUDAAdvisor is built atop LLVM to instrument CUDA code running across different GPU architectures and CUDA runtimes. Moreover, CUDAAdvisor supports novel code- and data-centric profiling to provide intuitive guidance for understanding abnormal behaviors in CUDA code. Finally, CUDAAdvisor supports various analysis techniques. We have shown three different analyses for reuse distance, memory and branch divergence. Based on these analyses, CUDAAdvisor is able to further devise new metrics to guide GPU optimization of cache bypassing, which achieves up to 2$\times$ speedup. It also provides unique code-centric and data-centric debugging capability.

Chapter 4

Presponse: Accelerating Incremental Large Graph Processing on GPU via Speculative Preprocessing

4.1 Introduction

Processing large graphs is always a difficult yet highly-crucial task for warehouse companies and high performance computing as it demands substantial computation power and high memory bandwidth. As a result, GPUs are often adopted to accelerate graph processing [88, 89, 90] because of the scalability and energy efficiency.

In the meantime, real-world graphs are not statically conserved but continuously evolving. Queries regarding to the graph properties often involve computation upon a large portion of the graph. When a graph evolves, it is desired that properties are up-to-date and the latest changes to the graph are reflected in query responses. However, fast processing for graph updates is computation intensive and crucial. One workaround is to aggregate updates in batches and emit the batches periodically. The period be-

tween batches depends on the update rate and batch size. Query responses reflect the changes in the latest batch and remain valid until the next batch is emitted and processed.

To process an update batch, a naive approach is to update the graph and recompute the properties of the entire evolved graph, marked as `static` in Fig 4.1a. Based on the observation that updates usually apply to only a small fraction of a large graph, existing approaches employ incremental computation to achieve efficient processing for evolving graphs [91, 92, 89, 93, 94]. Unlike static re-computation, incremental processing conducts minimal computation only for the evolved fraction. Thus, the computation and response latency can be substantially reduced, as shown in Fig 4.1b.

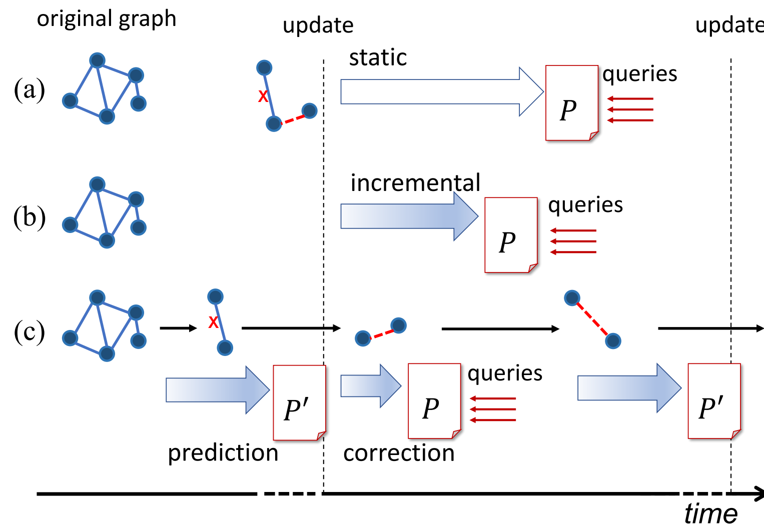


Figure 4.1: Comparison of three approaches to evolving graph: (a) static re-computation, (b) incremental processing, (c) proposed approach. The horizontal bottom arrow indicates time. The vertical dashed lines indicate batch emit times. The length of blue arrows indicate the processing time.

We can further accelerate the incremental processing by speculatively computing the possible updates based on our observation that *graph updates are highly predictable*. Large graphs in the real world, such as social networks, road maps, paper citations and community question answering graphs, often reflect real relations between vertices. Thus, real-world graphs are not random but semantically meaningful. For example, in a

social network, a vertex stands for a user, and an edge between two vertices represents a friend relation between the two users. New edges are more likely to appear between users who have a lot of mutual friends, or share common interests. Based on this, changes to real-world graphs follow certain patterns, making them predictable. Our experiments show that a well-trained classifier can reach over 90% accuracy.

Based on this observation, we propose Presponse, a prediction-based incremental graph processing framework. As shown in Fig 4.1c, during the idle interval between batches, Presponse predicts the updates in the next batch, and speculatively computes the intermediate properties. When the real update batch is emitted, our framework computes the updated properties by processing some correction changes. The falsely predicted changes are rolled back, and missed changes are patched. Given a sufficiently high prediction accuracy, the processing time is substantially reduced. It is worth noting that the update pattern may also evolve over time. To take that into consideration, Presponse monitors the changes to the graph and keeps learning the new patterns, which guarantees that our framework continuously follows the pattern changes and delivers reliable predictions.

In summary, we present Presponse, a framework that boosts graph processing by predicting graph updates. We propose three incremental graph algorithms tailored for GPU. We empirically show that prediction accuracy can reach over 90%, and delivers speedups up to $25\times$ for query responses.

Contributions. In summary, we make the following contributions in this chapter:

- We develop incremental implementations of three fundamental graph algorithms, Breadth First Search, Connected Components, and Triangle Counting. The implementations are tailored for GPU execution.
- We demonstrate that updates are highly predictable for real-world evolving graphs, which can greatly benefit the incremental computation.
- We propose Presponse, a framework that leverages graph update prediction and

utilizes GPU to boost incremental graph processing.

- We show that Presponse can accelerate important graph algorithms on real-world graphs with significant speedups.

In the rest of this chapter, Section 4.2 discusses some background knowledge used in this chapter. Section 4.3 elaborates on the methodology used by Presponse. Section 4.4 shows our evaluation and analysis on Presponse. Section 4.5 discusses some related work and distinguishes Presponse. Section 4.6 summarizes this chapter.

4.2 Background

In this section, we discuss the two most related topics: graph feature embedding and GPU support for processing evolving graphs.

4.2.1 Graph Embedding

Graph embedding extracts global properties from a graph by mapping to a multi-dimensional feature space. A trending technique to extract features is to apply random walks [95, 96, 97]. Multiple paths are sampled from a graph, and each path is regarded as a word sequence. Further feature learning is conducted based on Skip-gram, which is originally developed in the context of NLP (Natural Language Processing). The approach `node2vec` [97] is the state-of-the-art framework for learning feature vectors for vertices in a graph. It explores diverse neighborhood of a vertex by conducting a biased random walk.

4.2.2 GPU Acceleration

Adopting GPU for evolving graph processing requires storage structure that supports efficient graph updates, and fine-tuned implementations of algorithms. We concentrate on the storage structure and the algorithms will be discussed in the following section. There

are two types of graph representation: adjacent matrix [98] and compressed sparse row (CSR) [99]. Adjacent matrix supports efficient edge insertion/deletions but consumes substantial space; CSR is efficient in terms of space consumption but is not ideal for evolving graphs. Because adding or deleting one single edge requires resizing arrays and extensive memory copies. As a result, the latest work utilizes a hybrid storage structure [100] to support evolving graphs. The key of hybrid storage is to adopt a linked list of edge blocks. Each block is continuous on memory and block size is a configurable parameter. Upon insertion, a new edge is appended to the last block if there is a vacancy. If the last block is fully occupied, a new block is allocated. Upon deletion, the edge is swapped to the last edge and mark the last edge as deleted. Thus, efficient updates to evolving graphs are supported.

4.3 Methodology

This section introduces the workflow of Presponse, defines the problem that Presponse solves, discusses graph update prediction, and elaborates on incremental algorithms.

4.3.1 Workflow Overview

Presponse consists of two major components: prediction engine and graph processing. The prediction engine accepts the original graph as input, predicts changes for next batch and generates a list of prediction updates. The graph processing component processes a graph either statically or incrementally and outputs graph properties for query response.

Figure 4.2 overviews the workflow of Presponse. For initialization, Presponse takes the original graph and generates initial properties via static processing. In the next step, Presponse's prediction engine generates prediction updates based on the characteristics of the original graph. We discuss the details of the prediction engine in the following section. Given the prediction updates and the initial properties, incremental processing

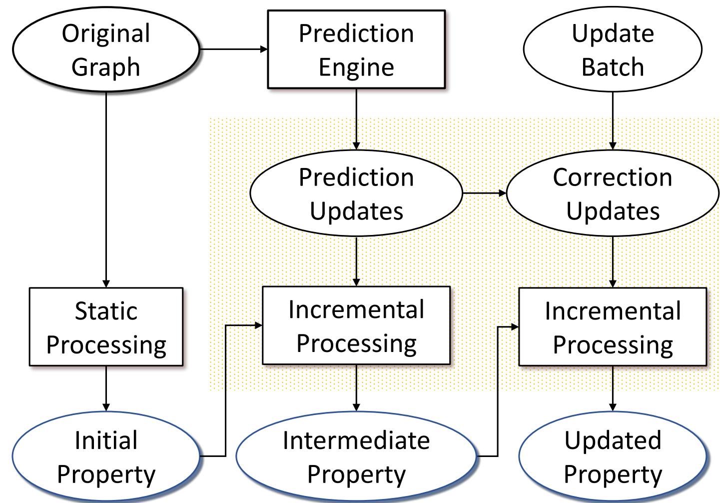


Figure 4.2: Workflow of Presponse. The rectangles represents components of Presponse. Given the original graph, static processing obtains initial property and prediction engine generates prediction updates. Incremental processing takes updates as input and computes intermediate properties. When an update batch is emitted, incremental processing takes correction as input and computes updated property. Orange shaded area represents procedures which run repeatedly for each batch.

component generates intermediate properties. It is worth noting that the intermediate properties are not necessarily for the real graph and are not used to respond queries. When an update batch is emitted, Presponse compares prediction updates with the batch, computes correction updates, and applies another round of incremental processing to obtain updated properties. These updated properties are obtained from the updated graph and are used to respond queries. Since updates are grouped into batches, the prediction and correction processes are repeated for each batch.

4.3.2 Presponse Prediction Engine

The task of prediction engine is to predict updates in the future. As mentioned in Section 4.3.1, Presponse launches the prediction engine when the static original graph is available. The engine periodically makes predictions about future changes during the interval between the arrival of two batches. The prediction engine works asynchronously. That being said when the system predicts graph changes, the graph processing component

can still respond to queries with the current properties at the same time.

We implement Presponse's prediction engine based on a well-trained binary classifier. Figure 4.3 shows the diagram of training the classifier. Starting from the static graph G_0, Presponse first learns the characteristics of the graph by generating a feature vector for each vertex. Second, the vertex features are converted into edge features. Third, the prediction engine generates *True* and *False* labels by sampling G_0. Lastly, a classifier is trained using aforementioned edge features and labels.

(I) Feature Embedding. Presponse relies on `node2vec` [97] to extract graph features. node2vec is an algorithmic framework which learns features of vertices of a graph and outputs a map of vertices to a multi-dimensional feature space. The feature learning problem is formulated as an optimization problem. To implement that, `node2vec` extends skip-gram to graphs. Skip-gram architecture is originally developed in the context of Natural Language Processing [101]. In `node2vec`, the objective function maximizes the probability of observing a certain vertex's network neighborhood, conditioned on its feature. Moreover, `node2vec` defines a flexible network neighborhood of a vertex by generating biased random walk. This flexible notion is key to richer feature representations. In summary, Presponse feeds a static graph to `node2vec` and obtains features of each vertex.

The next step is to convert vertex to edge features. The feature of an edge (u, v) is generated by taking the Hadamard product [102] of the two vertices u and v. Hadamard product is a binary operation that takes two vectors of the same dimensions and produces a new vector whose element i is the product of elements i from the original two vectors: $(A \circ B)_i = A_i \cdot B_i$. Note that Presponse only generates features for those edges, which will be used for classifier training.

(II) Training Classifier. The edge features will be used to train a binary classifier. The *True* labels represent positive training set of existing edges in G_0 and *False* training set represents negative training set of edges that are not present in G_0. To build positive set, edges in G_0 are randomly sampled. To construct a *False* label, two vertices are randomly

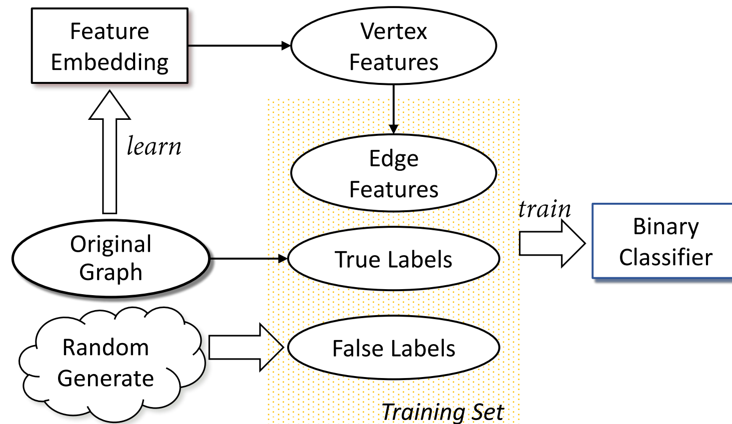


Figure 4.3: Details of training a classifier for Presponse's prediction engine.

selected. If these two vertices are not directly connected by an edge, this virtual edge is tagged *False*. The entire training set is used to train an SVM binary classifier [103]. When given an edge of interest, the classifier will indicate the edge will be present or absent in the future. Thus, Presponse generates a list of edges that will be inserted and deleted.

Thus, Presponse's prediction engine is able to capture the characteristics of a graph and make predictions. In some cases, the prediction results may suffer from low accuracy because the update pattern of a graph is dynamically evolving. For example, in a social network, a user's interest may shift. As another example in a community question answering graph, one user may start to explore new domains. In order to follow these changes of an evolving graph, we propose a *refined model* of the prediction engine. The refined model monitors the prediction accuracy for each batch. If the characteristics of a graph has changed after a few batches, the prediction accuracy is going to decrease. When the accuracy drops below a threshold, the prediction engine will re-train the classifier using the current graph features, which can increase the prediction accuracy.

4.3.3 Presponse Graph Processing

The graph processing component consists of static and incremental implementations. *Static* processing takes a static graph as input and computes certain properties of the graph. When the graph evolves, static processing would apply the changes and recompute for the entire new graph. This approach is time-consuming. Therefore, Presponse utilizes static processing merely to obtain initial properties, and relies on incremental processing for graph updates. *Incremental* processing takes previous property and graph updates (known as batches) as input, and efficiently computes the property for the updated graph.

Presponse relies on GPU to efficiently process graph and updates. GPUs have been widely utilized in graph workloads due to multi-threading and high computation power. Here we introduce the storage format and load balance strategy on GPU. Note that these implementation choices are independent of any concrete graph algorithms, which will be covered in the next section.

We employ a hybrid data storage format introduced by cuStinger [100]. This hybrid format stores vertices in arrays and each vertex has a linked list of blocks to maintain its edges, where a block is a continuous memory space, or equivalently, an array. We employ this format rather than adjacent matrix format and compressed sparse row format. Adjacent matrix format consumes considerable space, making it undesirable to store a large graph on memory-constrained architectures, e.g., GPU. Compressed sparse row format is compact in space, but inserting or deleting edges requires data reallocation and movements, which incurs high overhead. As a good tradeoff, the linked list of blocks only consumes reasonable space yet provides high efficiency for applying updates, which is especially suitable for storing evolving graphs on GPU.

In order to fully utilize the computation power of GPU, graph workload should be distributed to GPU blocks and threads. Depending on different graph algorithms, the kernel may need to traverse either vertices or edges. For vertex traversal, Presponse simply assigns the same amount of vertices to each block. When even distribution is

not possible, the last block will be assigned fewer vertices. On the other hand, edge traversal requires more efforts to ensure balancing. The amount of neighbouring vertices (i.e., degree) is taken into account. Each block will be assigned various amounts of vertices but the sum of the degrees of these vertices are similar. Thus, workload is evenly distributed to each block. It is worth noting that strict even distribution is often impossible for edge traversal; approximate even distribution is sufficient [100].

4.3.4 Graph Algorithm Implementation

Presponse provides static and incremental implementation of three graph algorithms: Breadth First Search (BFS), Connected Components (CC), and Triangle Counting (TC). We select these algorithms because they are fundamental to complex graph algorithms that are widely utilized.

4.3.4.1 Breadth First Search (BFS)

BFS starts from the *root* and calculates *depth* of all vertices. The depth is defined as the number of hops between certain vertex and the root. For the vertices that are not connected to the root, depth is infinity.

Static Computation: Initially, each vertex is assigned with an infinity depth. A root is randomly picked and pushed into a task queue. A global depth counter is initialized to 0. In each iteration, Presponse traverses every vertex in the task queue. If the the vertex has a depth smaller than infinity, it means this vertex has been previously visited by another thread. In this case, no action is needed on this vertex. If a vertex still has a depth of infinity, it takes the value of global depth counter and pushes all its neighbors into the task queue for computation in the next iteration. After each iteration, the global depth counter increments by 1. The entire computation halts when the task queue is empty.

Incremental Computation: Presponse scans all edges in the update batch and identifies all unique vertices. Presponse finds the vertex that is closest to the root. Such

Algorithm 1: BreadthFirstSearch

```
1  StaticBFS(G_0, frontier)
   Input: G_0:static input graph, frontier: vertices to initialize the queue
   Output: an array D of depth of each vertex
2  add each vertex in frontier into queue
3  D ← inf
4  depth ← 0
5  while queue do
6      initialize next_queue
7      for each u in queue do
8          if D[u] is inf then
9              D[u] ← depth
10             add all neighbors of u into next_queue
11     queue ← next_queue
12     depth ← depth + 1
13 return D
14
15 IncrementalBFS(G_0, B, D_0)
   Input: G_0:static input graph, B:update batch, D_0: pre-computed depth of each
          vertex in G_0
   Output: an array D_1 of depth of each vertex
16 threshold ← inf
17 for each vertex u in B do
18     threshold ← min (threshold, D_0[u])
19 frontier ← vertices depth ≥ threshold
20 D_1 ← StaticBFS(G_0,frontier)
21 return D_1
```

vertex has the lowest depth value, which we call the *threshold* depth. The global depth counter is set to the *threshold* depth. For all vertices of depth greater than or equal to this threshold, their depths are reset to infinity and pushed into a task queue. Finally, Presponse reverts to the aforementioned static computation.

Correctness Proof: The incremental computation merely decides the threshold depth and conducts re-computation for those vertices of higher depth than threshold. This is essentially static computation without vertices above (of lower depth than) threshold. This approach generates correct results because vertices above the threshold are not impacted by the updates. Moreover, the execution time of incremental computation

```
Algorithm 2: ConnectedComponents
1  StaticCC(G_0, frontier)
   Input: G_0:static input graph, frontier: the entire graph for static implementation
   Output: an array componentID of all vertices
2  initialize componentID with vertex_id
3  flag ← true
4  while flag do
5  |   flag ← false
6  |   for each edge (u, v) in frontier do
7  |   |   if componentID[u] ≠ componentID[v] then
8  |   |   |   id ← min(componentID[u], componentID[v])
9  |   |   |   componentID[u] ← id
10 |   |   |   componentID[v] ← id
11 |   |   |   flag ← true
12 return componentID
13
14 IncrementalCC(G_0, B, componentID)
   Input: G_0:static input graph, B:update batch, componentID: pre-computed
          component ID of all vertices in G_0
   Output: componentID_1 of component ID of all vertices
15 initialize frontier for each edge (u,v) in B do
16 |   if componentID[u] ≠ componentID[v] then
17 |   |   add u into frontier
18 |   |   add v into frontier
19 componentID_1 ← StaticCC(G_0,frontier)
20 return componentID_1
```

is dependent on the threshold rather than batch size. In section 4.4.2, we will empirically show that the speedup is dependent on the threshold, or approximately proportional to non-impacted vertices.

4.3.4.2 Connected Components (CC)

A connected component means there is always a path connecting any two vertices within the component. Our implementation of CC counts the number of connected components in a graph and outputs the components for any vertex.

Static Computation. Presponse uses a classical working set algorithm for static CC computation. Initially, each vertex has a *component ID* that is equal to its vertex ID,

which means that each vertex itself is a component. A global boolean flag is defined to control the iterative computation. When an iteration starts, the global flag is first set to False. In each iteration, Presponse traverses all edges in the graph and compares the component IDs of the two ending vertices of each edge. If the component IDs are different, two actions are required. First, the framework takes the smaller one of these two IDs and assigns it to both vertices. Thus, these vertices are grouped into one component. Second, the global flag is set to True and more following iterations are needed. When the global flag remains False at the end of an iteration, the computation converges and the execution halts.

Incremental Computation. Presponse traverses the update batch. If the two ending vertices of an edge (to be inserted or deleted) fall into different components, these two components are marked and further actions are required. After parsing the batch, all vertices in the marked components are pushed into the task queue. Presponse resets the component IDs of all vertices in the task queue and reverts to the static computation algorithm. It is worth noting that if the two ending vertices of a new edge fall into the same component, inserting this new edge will not change the component ID of any vertex. Thus, Presponse does not need further computation on this kind of edges.

Correctness Proof The incremental computation generates correct results because it conducts re-computation for all vertices in the impacted components, which is a subset of the entire graph. It is straightforward that the speedup of incremental computation relies on the amount of components.

4.3.4.3 Triangle Counting (TC)

A triangle in a graph means that there are three edges (u, v), (u, w), and (v, w) connecting three vertices u, v, and w. The TC algorithm counts the number of triangles that any given vertex is involved.

Static Computation. Presponse maintains a counter for each vertex to record the number of triangles that this vertex belongs to. Initially, the counter is set to 0. For

Algorithm 3: TriangleCounting

```
1  StaticTC(G_0)
   Input: G_0:static input graph
   Output: number_tri: of each vertex
2  number_tri ← 0
3  for each vertex u in G_0 do
4  |   for each v in u's adjacent list do
5  |   |   number_tri[u] ← number of common neighbors
6  for each vertex u in G_0 do
7  |   number_tri[u] ← number_tri[u]/3
8  return number_tri
9
10 IncrementalTC(G_0, B, number_tri)
   Input: G_0:static input graph, B:update batch, number_tri: pre-computed triangle
          counts of all vertices in G_0
   Output: an array number_tri_1 of depth of each vertex
11 initialize queue with unique vertices in B
12 for each vertex u in queue do
13 |   number_tri[u] ← 0
14 for each vertex u in queue do
15 |   for each v in u's adjacent list do
16 |   |   for each w as a common neighbor do
17 |   |   |   number_tri[u] ← number_tri[u] + 1 if w not in queue then
18 |   |   |   |   number_tri[w] ← number_tri[w] − 1
19 for each vertex u in G_0 do
20 |   number_tri[u] ← number_tri[u]/3
21 return number_tri
```

each vertex u, Presponse examines each vertex v in its adjacent list. For each pair of (u, v), Presponse counts the number of common neighbors of u and v by taking the intersection of their adjacent lists. This number of common neighbors is accumulated to the counter of vertex u. After all vertices finalize their counters, Presponse divides the counter values by three to obtain the total number of triangles for each vertex, because the algorithm repeatedly counts each triangle three times.

Incremental Computation. The incremental algorithm handles edge insertion and deletion separately. Similar to BFS and CC algorithms, the incremental TC for edge

insertion works on the updated graph with the update batch. It consists of two steps. In the first step, we traverse the update batch, identify all unique vertices, and push them into a task queue, resetting their corresponding counters to 0. In the second step, for each vertex u in the queue, Presponse examines each vertex v in its adjacent list. For each pair of (u, v), Presponse counts the number of common neighbors of u and v by taking the intersection of their adjacent lists. This number of common neighbors is added to the counter of vertex u. In addition, for each common neighbor w for pair (u, v), the counter of w is incremented by 1 if w itself is not in the task queue. This way, Presponse obtains the number of triangles of each vertex. The second step is similar to the static implementation with two following differences. 1) The incremental algorithm only traverses vertices in the task queue while the static implementation traverses all vertices in the graph. 2) The incremental algorithm needs to increment the counter of the common neighbors by 1 if that common neighbor itself is not in the task queue.

The incremental TC of edge deletion works on the original graph without the batch update. Presponse first traverses the update batch and tags all edges from the batch in the original graph with a *delete marker*. Presponse then performs incremental TC of batch insertion but skip any edge that is marked with a *delete marker*. After the incremental TC is finished, we apply the batch to the original graph and obtain the evolved graph. Because of the fact that we only mark edges to be deleted without actually removing them, incremental TC of batch deletion spends time accessing those edges on memory. This procedure downgrades the performance. However, this is harmful only when batch size is comparable to the graph. We will confirm with experiments and discuss the results in Section 4.4.

Correctness Proof: If an vertex is impacted by the updates, incremental computation resets its triangle counter and re-computes this vertex via static computation. The only exception is that the counter of a common neighbour also needs to increment when the common neighbour itself is not in the updates. The rationale is changing one edge may introduce one triangle, impacting three vertices. When the common neighbor is not

present in the updates, it will be not be reset and re-computed. Therefore, incremental computation has to increment its counter in order to guarantee correctness. It is easy to note that the execution time is dependent on the batch size. To be more specific, it is proportional to the amount of unique vertices in the batch (see Section 4.4.2).

4.4 Evaluation

In this section, we evaluate the speedup of aforementioned incremental graph algorithms, analyze the effectiveness of prediction, and assess Presponse using real-world graph datasets.

4.4.1 Experiment Setup

Evaluation Overview Our empirical studies consist of two aspects. First, we compare the static re-computation against incremental processing, and demonstrate that incremental processing provides performance gains. Second, we evaluate the effectiveness of Presponse and demonstrate that combining link prediction and incremental processing can yield further speedups.

Evaluation Platform We evaluate Presponse on two platforms. One platform employs 48 Intel Xeon E-2650 processors running at 2.2 GHz and an NVIDIA K40c (Kepler) GPU with driver 384.111. The other platform employs 24 Intel Xeon E5-2643 processors running at 3.4 GHz, hosting an NVIDIA GTX 1080 (Pascal) GPU with driver 390.77. For both platforms, the host compiler is `gcc 4.8.5`, and CUDA version is 8.0.

Evaluated Algorithms We evaluated three widely used graph algorithms: BFS, CC and TC. BFS takes either directed or undirected graph as input, while CC and TC accept undirected graph only.

Table 4.1: Graph Datasets for Algorithm Evaluations.

Dataset	Name	Type	#Vertices	#Edges
delaunay_n15	n15	synthetic	32769	196548
delaunay_n18	n18	synthetic	262145	1572792
delaunay_n20	n20	synthetic	1048577	6291372
ca-HepPh	hep	real world	12008	237010
af_shell9	shell	real world	504856	17084020
thermal12	thermal	real world	1227088	7352268

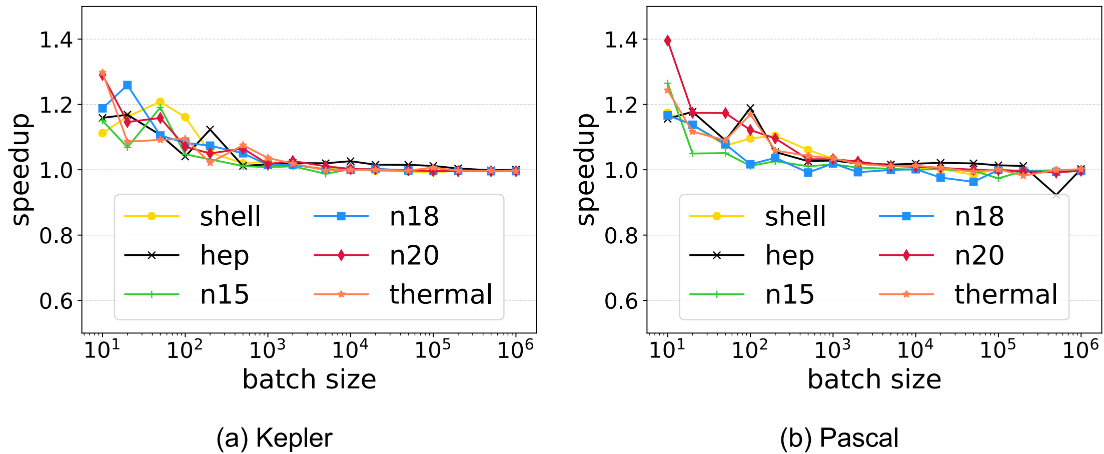


Figure 4.4: Speedup of incremental BFS. Horizontal axes are log-scaled. Horizontal axes represent the size of update batches. Vertical axes represent incremental BFS's speedup against static BFS.

4.4.2 Incremental Algorithms

We evaluate the performance of incremental algorithms against the original static implementations. Given a graph and its updates, we compare the execution time of incremental processing against static re-computation. Updates are randomly generated and are rendered in batches. Batch sizes range from 10 to $1,000,000$ edges. Table 4.1 summarizes the datasets used for evaluation; these datasets are representative ones widely used in previous work [104, 105].

BFS: Figure 4.4 shows the speedup of incremental BFS against static re-computation. We can see that on both platforms, incremental algorithm runs faster with smaller update batches (<100 edges). However, the speedup decreases as batch size

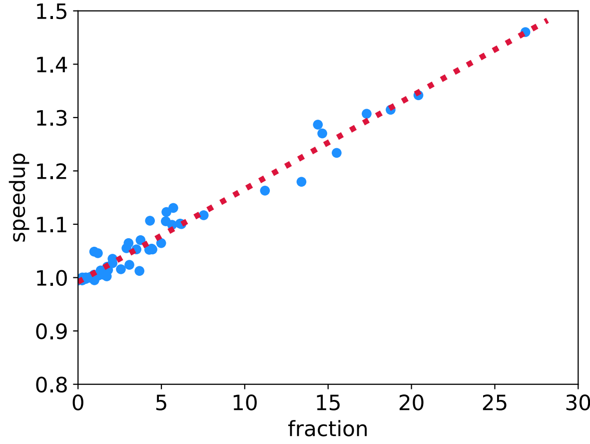
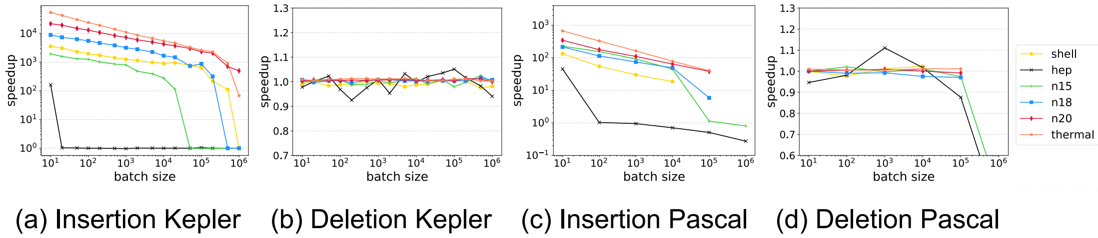


Figure 4.5: Relation between speedup and fraction of impacted vertices. The horizontal axis represents the fraction of non-impacted vertices in percentage. The vertical axis represents the ratio of elapsed time of incremental BFS computation to static BFS computation. The blue scattered points are data points, while the red dashed line represents linear fitting.

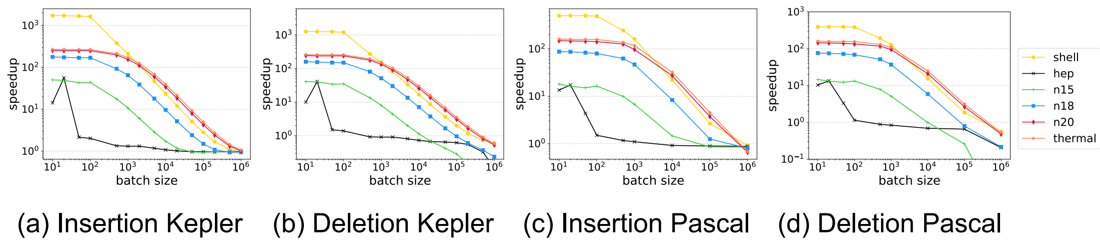


(a) Insertion Kepler (b) Deletion Kepler (c) Insertion Pascal (d) Deletion Pascal

Figure 4.6: Speedup of incremental CC. Horizontal axes are log-scaled. For deletion cases, both axes are log-scaled. Horizontal axes represent the size of update batches. Vertical axes represent incremental CC's speedup against static CC.

increases. The speedup starts around $1.4\times$ and drops to $1\times$. Such trends are similar across different datasets.

To understand why speedup drops as update batch size grows, we conduct further investigation. We find that speedup is highly related to the fraction of vertices that are not *impacted* by the update batch. We take `n20` as an example and show its results in Figure 4.5. The horizontal axis represents the *non-impact* fraction (in percentage), and vertical axis is the speedup. The figure clearly shows the fewer vertices are impacted by the update batch, the higher speedup we can achieve. To be more specific, speedup is almost proportional to non-impact fraction. The root cause is straightforward. If an



(a) Insertion Kepler (b) Deletion Kepler (c) Insertion Pascal (d) Deletion Pascal

Figure 4.7: Speedup of incremental TC. Both axes are log-scaled. Horizontal axes represent the size of update batches. Vertical axes represent incremental TC's speedup against static TC.

update inserts or deletes an edge that is very close to the root, it is straightforward to foresee that vertices that are deeper or equal to that vertex need re-computing, as described in Section 4.3.4.1; Thus, its execution time is close to the static re-computation. On the other hand, if the update batch only impacts vertices are that far away to the root, it takes much less time to incrementally compute the updated results only.

CC: Figure 4.6 shows the speedup of incremental CC computation against static recomputation. For incremental CC computation with edge insertion (Figure 4.6a and 4.6c), the speedup is significantly greater than $1\times$ for smaller batches, but drops as batch size increases. It is worth noting that the speedup always starts from a large value and quickly dimishes at a so-called *inflection point*. It is due to the property of graphs used in our experiments. To be more specific, a few components consist most vertices, i.e., dominant components, but other components only contains few vertices. When we randomly generate updates, it is likely to connect vertices in the same component. As described in Section 4.3.4.2, such edges requires no extra computation, resulting in high speedup. We are more likely to see this case for smaller batches. In contrast, when the batch size increases, the probability of inserting a new edge across different components increases dramatically. In this case, more vertices are put into the task queue for recomputation, resulting in no speedups. It is worth noting that the speedup can be smaller than $1\times$ due to the overhead incurred from the update batch traversal.

As for incremental CC computation with edge deletion (Figure 4.6b and 4.6d, the speedup is always around 1 except for the very large batch on Pascal. The reason

is that the datasets used in the experiments consist only a few components where a certain component owns most vertices. As long as one edge from the batch touches this component, all vertices of this component are put into the task queue for recomputation. Because little computation can be saved, the speedup is around $1\times$.

TC: Figure 4.7 shows the speedup of incremental TC computation against static recomputation, with edge insertions and deletions, respectively. In both figures, the speedup is significantly greater than 1 for small update batches across all input graphs. As the batch size increases, the speedup drops. In the edge insertion mode, for most input graphs, the speedup drops to $1\times$ when batch size is large. The reason is that the number of vertices in the task queue for recomputation cannot exceed the total number of vertices in the graph. Thus, the incremental computation at worst is the same as the static recomputation. Some datasets such as `shell`, `n18` suffer from slowdown because of the overhead incurred in update batch traversal and task queue operations.

In edge deletion mode, the incremental computation can incur a large slowdown given a large enough update batch with the following reason. When the size of the update batch is comparable to the original graph G_0, the task queue for incremental TC computation includes most vertices in the graph. When traversing the adjacent list of a certain vertex, we need go through every edge and only act on those that are not marked as delete (see Section 4.3.4.3). Thus, the computation time of incremental algorithm in deletion mode is comparable to the static TC on G_0. On the other hand, when the update batch is large enough, the evolved graph can be a small subgraph of G_0, so the static algorithm needs little time for recomputation. As an extreme example, when the update batch deletes all vertices and edges in G_0, the static recomputation requires zero runtime but the incremental computation still yields significant runtime.

To further understand the speedup, we examine the relation between the speedup and the number of impacted vertices. We take shell as an example and show the results in Figure 4.8. From the figure we can see that the speedup is inversely proportional to the number of impacted vertices in the graph during the evolution.

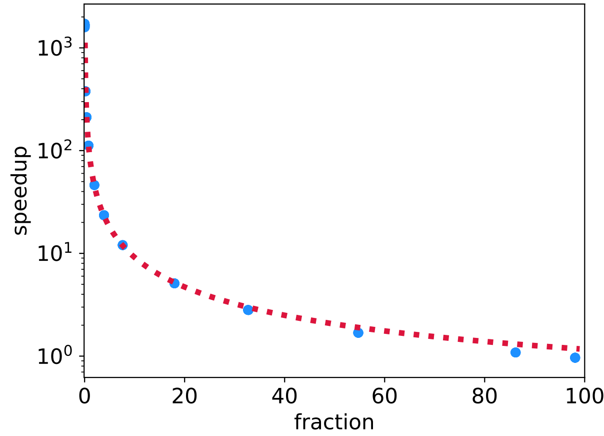


Figure 4.8: Relation between speedup and fraction of impacted vertices. Horizontal axis represents the percentage of impacted vertices. Vertical axis represents speedup. The blue scattered circles represent each data point, and the red dashed line represents theoretical speedup obtained via regression.

Table 4.2: Description of Real-world Datasets.

Dataset	#Vertices	#Edges
android	36418	51441
apple	64613	117730
gaming	52170	112829
tex	47992	149103
unix	69143	171146

4.4.3 Update Batch Prediction

In this section, we discuss the accuracy of link prediction. We use real-world graph generated from the data from *stackexchange.com* for evaluation. The website *stackexchange* is for community question answering (CQA) and offers raw data for download. The raw data are categorized into many domains and we select the following domains for our evaluation: android, apple, gaming, tex, and unix. Table 4.2 elaborates on the details of these graphs. In these graphs, each vertex represents one user, who asks and/or answerers a question; each edge (u,v) denotes that user v answers a question asked by user u.

We construct the initial graph using the data till Jan 1, 2018. We create updates using new data collected later and divide them into batches on a monthly basis. It is worth noting that there are only insertions in a CQA graph, because existing questions and answers are never deleted.

We use *accuracy* to assess Presponse's classifier. By definition, accuracy is the ratio of correct predictions (true positive plus true negative) to all the test cases; 100% means the prediction is always correct, while 0% means the prediction is always wrong. Figure 4.9 quantifies the accuracy. For tex and gaming, the accuracy is around 90% and is stable over the time. For unix, apple and android, the accuracy is below 80%, indicating that these three topics are difficult to predict. Among all five domains, android and apple show a clear trend that the classifier's accuracy decreases over time. This implies that the hot topics and interests change fast in these two domains. Thus, the classifier is not highly reliable on predicting future changes.

In order to address this low accuracy issue, we retrain the classifier when the accuracy is low. In our experiment, we set a threshold of 70% and show the results of re-training in Figure 4.10. The red line represents the accuracy without re-training, and the blue line represents accuracy with re-training. In Figure 4.10a, apple's accuracy starts above 70% and drops below the threshold in Mar. When re-training is enabled, the accuracy goes higher. More importantly, the prediction accuracy on the following batches also improves over the baseline. Similarly, in Figure 4.10b, the classifier is re-trained multiple times to improve the accuracy. We obtain a satisfactory accuracy in Jun, which is higher than 70%. In contrast, the accuracy drops below 60% if re-training is disabled.

4.4.4 Presponse Performance with Incremental Computation and Link Prediction

In this section, we demonstrate the benefits that graph update prediction can provide on top of incremental processing. We use standalone incremental processing (without the

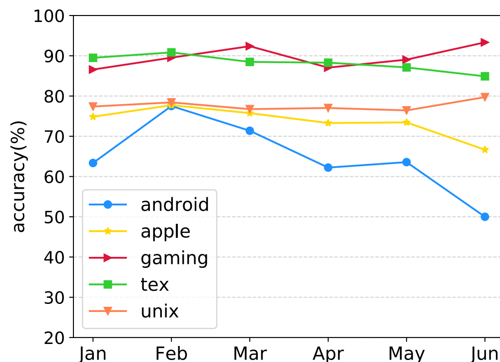
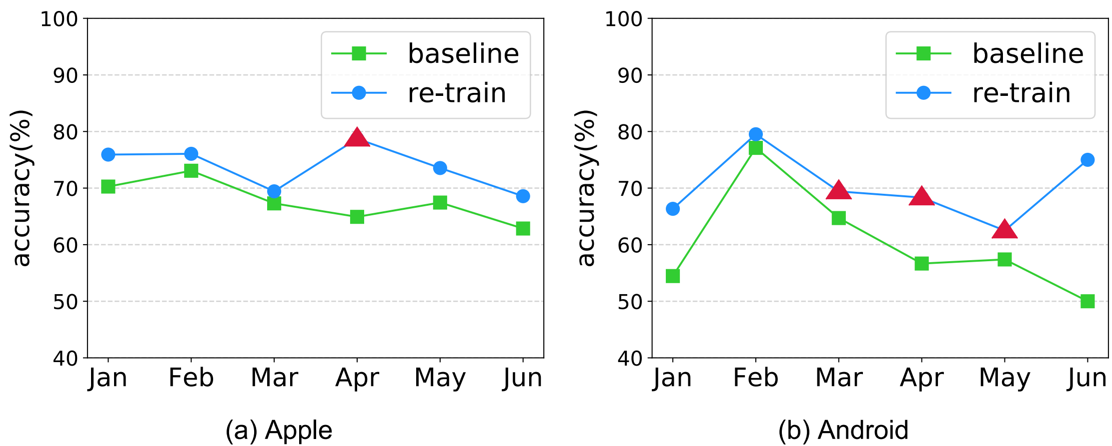


Figure 4.9: Prediction accuracy. Horizontal axis represents the time. Vertical axis represents prediction accuracy in percentage.

link prediction) as the baseline to show the speedup of Presponse, which combines link prediction and incremental processing. Presponse predicts future changes, generates a *prediction* batch, and incrementally processes it. When the real batch is emitted, Presponse generates a *correction* batch by comparing the *prediction* and the *real* batch. The final results can be obtained by incrementally processing the graph with the *correction* batch. As the inputs, we use the same real-world graphs in Section 4.4.3.

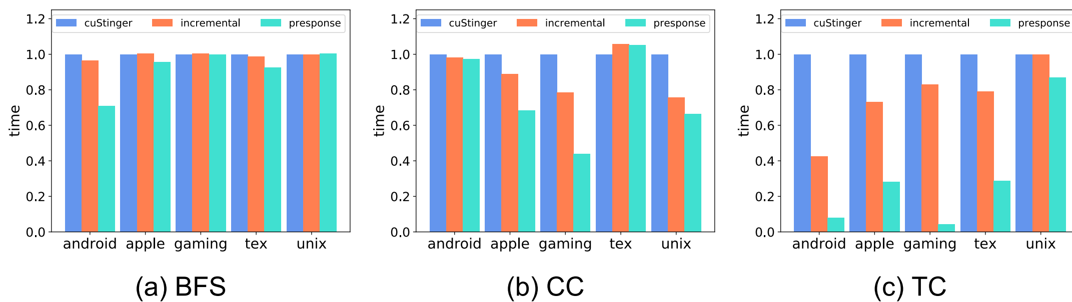
Figure **??** shows the normalized execution time. Since training classifier introduces randomness, we repeat our experiments for five times. Each bar in the figure represents average execution time. We can see that Presponse is always faster. Take TC running on gaming as an example, Presponse is $25\times$ faster. We discuss each algorithm individually in the following paragraphs.

BFS The results of BFS are shown in Figure 4.11a. Our first observation is that Presponse has speedup on gaming and tex, which falls align with the fact that gaming and tex are highly predictable (as shown in Figure 4.9). For other datasets, the execution time of Presponse is generally shorter than baseline on both architectures. As we explained in Section 4.4.2, the performance of incremental BFS is highly dependent on the depth from the root to the impacted level, instead of batch sizes. Therefore, although Presponse makes correct prediction on 85% changes in the *Real* batch, it only provides



(a) Apple (b) Android

Figure 4.10: Prediction accuracy when re-training enabled. (a) for apple, and (b) for android. Horizontal axis represents the time. Vertical axis represents prediction accuracy, in percentage. Original setup without re-training is shown as baseline. Red triangle markers indicate points of re-training.



(a) BFS (b) CC (c) TC

limited performance speedup because the impacted level is close to the root. Thus, Presponse has to be conservative about making predictions to vertices close to the root. Incorrect predictions (both false positive and false negative) on such vertices result in a higher penalty, which can surpass the benefits of prediction.

CC The results of CC are shown in Figure 4.11b. On both architectures, Presponse is typically faster than the baseline. Among all datasets, gaming obtains the highest speedup because of the high prediction accuracy. It is worth noting that Presponse incurs slowdown for tex. As explained in Section 4.4.2, the performance of incremental CC computation is highly dependent on impacted vertex fraction. In a graph where there is a large component, we are more likely to see limited speedup. Even if the change

impacts only one vertex in the dominant component, all vertices in this component have to be recomputed. Thus, the speedup is squandered due to the overhead of parsing batches.

TC The results of TC are shown in Figure 4.11c. We observe significantly speedups of Presponse for all datasets on both architectures. The highest speedup can be $25\times$ on the gaming dataset, which falls align with the fact that gaming has the highest prediction accuracy (in Figure 4.9). It is worth noting that Presponse on unix has 80% prediction accuracy but the execution time is close to baseline. It is because unix is the largest among the five datasets. It has the largest vertex and edge counts, and the highest average degree. As a result, traversing the adjacent lists costs considerable computation, surpasses the prediction benefit.

4.5 Related Works

Graph Embedding It has been recently shown that many popular random walk based approaches, such as DeepWalk [95], LINE [96], and node2vec [97], can be unified into the matrix factorization framework with closed forms [106]. All of the aforementioned models are based on the Skip-Gram model introduced by Mikolov *et al.* [107]. For example, DeepWalk samples multiple paths from the graph, each of which is regarded as a word sequence. Node2vec offers a flexible sampling strategy, with two parameters controlling the shape of the sampled paths. For each vertex in the sequence, they predict the nearby vertices in both direction, and update the vector according to the Skip-Gram model. However, these methods ignore the asymmetric nature of the path sampling procedure and train the model symmetrically, which restricts their applications. Since node pairs from two hops away will be regarded as negative labels, LINE can only preserve symmetric second-order proximity when applied to directed graphs [108].

Line introduces the second order proximity between a pair of vertices, which encodes the similarity measured by their local neighborhood. However, Line can only preserve

symmetric second-order proximity when applied to directed graphs [109]. In addition, it cannot preserve the higher-order similarities, since node pairs from two hop away will be regarded as negative labels [108].

Recently, one has seen a surge of interest in developing methods for learning representations for directed graphs. For example, ATP [110] incorporates graph hierarchy and reachability information naturally by relying on a nonlinear transformation that operates on the core reachability and implicit hierarchy within such directed graphs to preserve the asymmetric transitivity in the embedding space. Sun *et al.* [111] proposed to solve the directed graph embedding problem via a two-stage approach: in the first stage, the graph is symmetrized in one of several possible ways, and in the second stage, the so-obtained symmetrized graph is embedded using any state-of-the-art (undirected) graph embedding algorithm such as node2vec [97].

Link Prediction. Wang *et al.* [112] categorize link prediction techniques into the following groups: node-based, topology-based and social theory -based. Node-based approaches leverage the attributes of nodes to mine the similarity between them, for example interests of users and keywords in profiles [113, 114, 115]. Topology-based approaches relies on the structural features of the graph when attributes are absent. The structural features may come from common neighbors, paths or random walks [116, 117, 118, 119]. The third category includes a lot of social theory metrics such as community, strong and weak ties, homophily, *etc*. These techniques can capture additional social interaction information [120, 121, 122, 123, 124, 125].

Evolving graph processing. There are two categories: offline and online. Offline processing [126, 127, 92] involves graph generation, storing and analysis of snapshots of evolving graph. GraphScore [126] is shared-memory CPU based solution. It conducts community discovery and anomaly detection by processing graph encodings. Chronos [92] supports temporal graph processing. It puts together graph vertex data from different versions to achieve good cache locality. TEG [127] partitions evolving graph across graph vertices and enables subgraph querying. Online processing provides continuous

query. STINGER [128] defines a layout of storing graph vertices and edges, and supports fast efficient insertions/deletions to the graph. EvoGraph [104] applies a hybrid structure to enable compressed storage for static computation and incremental computation on edge lists. Ji *et al.* [129] experimentally demonstrate that initial values of the computation on a new graph snapshot have a critical impact one the convergence of iterative graph algorithms, such as PageRank and Kmeans.

GPU for graphs. Existing works focus on CPU or Intel Xeon Phi style architectures, such as Stinger [130], Kineograph [131], CellIQ [132], GraphTau [133] [134] and SLFE [135]. However, GPUs introduces different design principles and performance concerns in the parallel execution [105]. King *et al.* [136] explores the direction of using GPUs to process real-time analytics on dynamic graphs. However, it only supports insertions and lacks an efficient indexing mechanism. cuStinger [100] defines a format for graph processing on GPU. cuStinger is a framework that implements several graph premises for GPU. It support various input format and employ a hybrid data storage to reach a good balance on performance and space consumption. Graphie [137] is a systemt that supports traversing large-scale graph on single GPU. It relies on two renaming algorithm to efficiently utilize shared memory and reduce global memory accesses. Sha *et al.* [105] proposed two parallel update algorithms (GPMA and GPMA+) to support efficient stream updates so that the maintained graph is immediately available for high-speed analytic processing on GPUs. GPMA explores a lock-based approach which works efficiently for the case where few concurrent updates conflict, *e.g.*, small-size update batches with random updating edges in each batch. Regarding to the scenarios where massive conflicts occur, GPMA+, a lock-free approach, was proposed in a bottom-up way by prioritizing updates that occur in similar positions. GPMA+ is able to maximize coalesced memory access and achieve linear performance scaling w.r.t the number of computation units on GPUs, regardless of the update patterns.

4.6 Chapter Summary

In this chapter, we propose incremental algorithms of three fundamental graph applications (Breadth-First-Search, Connected Components, and Triangle Counting), tailored for GPU. Moreover, this chapter presents Presponse, a framework that boosts incremental graph processing. Presponse adopts link prediction that predicts update batches for evolving graph to reduce the delay of the batch processing. Thus, queries regarding to graph properties can be responded much faster. We demonstrate that prediction accuracy can as high as 90% and query responses can be up to $25\times$ faster. For future work, we will further improve the incremental algorithms such that the performance is only dependent on the batch size. We plan to generalize current implementation to multi-GPU platforms for better scalability.

Chapter 5

Conclusion

In this dissertation, we propose solutions to enhance the performance of programs for heterogeneous architectures. Specifically, we work on the following three projects:

First, we present DataPlacer, a framework that studies the impact of software-managed heterogeneous memory in a real system, the TI KeyStone II. We also develop HMBench, the first OpenMP benchmark suite that adopts OpenMP 4.0 standard and works on heterogeneous architectures. DataPlacer is a profiler to provide guidance for data placement in different layers of software-managed cache and memory. Using HMBench and DataPlacer, we observe the insight that HM plays an important role in both boosting performance and reducing energy consumption. Moreover, we leverage HMBench and DataPlacer to characterize the performance gains with HM.

Our future work is twofold. First, we will develop more benchmarks for HMBench to make it as the standard benchmark suite for evaluating HM-based systems and compilers. Second, we will extend DataPlacer to provide low-level guidance for compiler-based optimization for HM. Such low-level information includes the finer granularity of data placement on cache lines or pages, instead of arrays. We believe that optimizations on HM from both high-level source code transformation and low-level compiler-supported code generation can achieve the optimal performance.

Second, we present CUDAAdvisor, a general framework that supports fine-grained analysis to identify performance bottlenecks of CUDA code. CUDAAdvisor is built atop

LLVM to instrument CUDA code running across different GPU architectures and CUDA runtimes. Moreover, CUDAAdvisor supports novel code- and data-centric profiling to provide intuitive guidance for understanding abnormal behaviors in CUDA code. Finally, CUDAAdvisor supports various analysis techniques. We have shown three different analyses for reuse distance, memory and branch divergence. Based on these analyses, CUDAAdvisor is able to further devise new metrics to guide GPU optimization of cache bypassing, which achieves up to 2$\times$ speedup. It also provides unique code-centric and data-centric debugging capability.

Third, we propose Presponse, a framework that boosts incremental graph processing. Presponse adopts link prediction that predicts update batches for evolving graph to boost batch processing. Thus, queries regarding to graph properties can be responded much faster. Moreover, we develop incremental algorithms of three fundamental graph applications (Breadth-First-Search, Connected Components, and Triangle Counting), tailored for GPU. We demonstrate that prediction accuracy can as high as 90% and query responses can be up to $25\times$ faster. For future work, we will further improve the incremental algorithms such that the performance is only dependent on the batch size. We plan to generalize current implementation to multi-GPU platforms for better scalability.

Bibliography

[1] NVIDIA. CUDA Programming Guide, 2015.

[2] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[3] Andreas Sandberg, David Eklöv, and Erik Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[4] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Feb 2008.

[5] Bin Bao and Chen Ding. Defensive loop tiling for shared cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[6] Jayesh Gaur, Raghuram Srinivasan, Sreenivas Subramoney, and Mainak Chaud-

huri. Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 395–407, New York, NY, USA, 2013. ACM.

[7] Mary Baker, Satoshi Asami, Etienne Deprit, John Ouseterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 10–22, 1992.

[8] Intel. Knights Landing, the Next Generation of Intel Xeon Phi. http://www.enterprisetech.com/2014/11/17/enterprises-get-xeon-phi-roadmap/. Last accessed: Dec. 08, 2014.

[9] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[10] Texas Instruments. DSP products website. http://www.ti.com/lsds/ti/dsp/overview.page. Last accessed: Dec. 08, 2014.

[11] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen. Exploiting program semantics to place data in hybrid memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 163–173, Oct 2015.

[12] Niladrish Chatterjee, Manjunath Shevgoor, Rajeev Balasubramonian, Al Davis, Zhen Fang, Ramesh Illikkal, and Ravi Iyer. Leveraging heterogeneity in dram main memories to accelerate critical word access. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 13–24, 2012.

[13] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, 2011.

[14] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Software-managed energy-efficient hybrid dram/nvm main memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 23:1–23:8, New York, NY, USA, 2015. ACM.

[15] Dong Li, Jeffrey S. Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 945–956, Washington, DC, USA, 2012. IEEE Computer Society.

[16] Javed Absar and Francky Catthoor. Analysis of scratch-pad and data-cache performance using statistical methods. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6–pp. IEEE, 2006.

[17] Texas Instruments. 66ak2hx keystone multicore dsp+arm system-on-chips. http://www.ti.com/lit/ml/sprt651a/sprt651a.pdf.

[18] Lina J Karam, Ismail AlKamal, Alan Gatherer, Gene A Frantz, David V Anderson, and Brian L Evans. Trends in multicore dsp platforms. *Signal Processing Magazine, IEEE*, 26(6):38–49, 2009.

[19] Yang Gao, Fan Zhang, and Jason D Bakos. Sparse matrix-vector multiply on the keystone ii digital signal processor. In *IEEE HPEC*, 2014.

[20] Jason Cong, Hui Huang, Chunyue Liu, and Yi Zou. A reuse-aware prefetching scheme for scratchpad memory. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 960–965, New York, NY, USA, 2011. ACM.

[21] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, July 2000.

[22] Mitesh R Meswani, Gabriel H Loh, Sergey Blagodurov, David Roberts, John Slice, and Mike Ignatowski. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 9–16. IEEE, 2014.

[23] Ross McIlroy, Peter Dickman, and Joe Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM ’08, pages 31–40, New York, NY, USA, 2008. ACM.

[24] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 409–415. IEEE, 2002.

[25] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT ’06, pages 321–330, New York, NY, USA, 2006.

[26] Tong Chen, Tao Zhang, Zehra Sura, and Mar Gonzales Tallada. Prefetching irregular references for software cache on cell. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 155–164, New York, NY, USA, 2008.

[27] Ke Bai and Aviral Shrivastava. Automatic and efficient heap data management for limited local memory multicore architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 593–598. IEEE, 2013.

[28] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[29] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 88–100, Dec 2014.

[30] C. Li, Y. Yang, Z. Lin, and H. Zhou. Automatic data placement into GPU on-chip memory resources. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '15, New York, NY, USA, 2015. ACM.

[31] Felix Xiaozhu Lin and Xu Liu. Memif: Towards programming heterogeneous memory asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 369–383, 2016.

[32] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 15:1–15:16, New York, NY, USA, 2016. ACM.

[33] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013.

[34] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[35] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream.

[36] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the 2009 IEEE Intl. Symp. on Workload Characterization (IISWC)*, pages 44–54, Washington, DC, USA, 2009.

[37] Chi-Keung Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005.

[38] Xu Liu and John Mellor-Crummey. Pinpointing data locality bottlenecks with low overheads. In *Proc. of the 2013 IEEE Intl. Symp. on Performance Analysis of Systems and Software*, Austin, TX, USA, April 21-23, 2013.

[39] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[40] Top500 supercomputer sites. https://www.top500.org/lists/2017/06, Jun. 2017.

[41] NVIDIA Group. NVIDIA DGX-1 AI Supercomputer. http://www.nvidia.com/object/deep-learning-system.html, 2017.

[42] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117(1):1–19, March 1995.

[43] Google Inc. TensorFlow: An open-source software library for Machine Intelligence. https://www.tensorflow.org, 2017.

[44] Keshav Pingal. Galois. http://iss.ices.utexas.edu/?p=projects/galois, 2014.

[45] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, and Henk Corporaal. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XXII, New York,NY, USA, 2017. ACM.

[46] Guoyang Chen and Xipeng Shen. Free launch: optimizing gpu dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 407–419. ACM, 2015.

[47] Intel VTune Amplifier XE 2017. http://software.intel.com/en-us/intel-vtune-amplifier-xe, April 2017.

[48] Oracle. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html, 2012.

[49] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22:685–701, 2010.

[50] Dominic A. Varley. Practical experience of the limitations of gprof. *Software: Practice and Experience*, 23(4):461–463, 1993.

[51] NVIDIA Visual Profiler. NVIDIA, 2017.

[52] Allen D. Malony, Scott Biersdorff, Sameer Shende, Heike Jagode, Stanimire Tomov, Guido Juckeland, Robert Dietrich, Duncan Poole, and Christopher Lamb. Parallel performance measurement of heterogeneous parallel systems with gpus. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society.

[53] Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. Effective sampling-driven performance tools for gpu-accelerated supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 43:1–43:12, New York, NY, USA, 2013. ACM.

[54] NVIDIA Corp. CUDA Tools SDK CUPTI User's Guide DA-05679-001_v01. https://developer.nvidia.com/nvidia-visual-profiler, October 2011.

[55] NVIDIA. CUDA 7.5: Pinpoint Performance Problems with Instruction-Level Profiling. https://devblogs.nvidia.com/parallelforall/cuda-7-5-pinpoint-performance-problems-instruction-level-profiling, 2015.

[56] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[57] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[58] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. Flexible software profiling of gpu architectures. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 185–197, New York, NY, USA, 2015. ACM.

[59] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Sympo-*

sium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[60] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. Gpucc - an open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 105–116, New York, NY, 2016.

[61] LLVM Group. LLVM: User Guide for NVPTX Back-end. `http://llvm.org/docs/NVPTXUsage.html`, 2016.

[62] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[63] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)*. IEEE, 2012.

[64] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[65] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither more nor less: optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 157–166. IEEE Press, 2013.

[66] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware Warp Scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Sympo-*

sium on Microarchitecture, MICRO-46, pages 99–110, New York, NY, USA, 2013. ACM.

[67] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 17. ACM, 2015.

[68] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic gpu cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS'15, pages 67–77. ACM, 2015.

[69] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–283, Feb 2014.

[70] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 343–355, Washington, DC, USA, 2014. IEEE Computer Society.

[71] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 516–523, Piscataway, NJ, USA, 2013. IEEE Press.

[72] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for GPUs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88, Feb 2015.

[73] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 25–38, Oct 2015.

[74] Lingda Li, Ari B Hayes, Shuaiwen Leon Song, and Eddy Z Zhang. Tag-Split Cache for Efficient GPGPU Cache Utilization. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS'17, page 43. ACM, 2016.

[75] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 86–98, New York, NY, USA, 2013. ACM.

[76] C. Nugteren, G. J. van den Braak, H. Corporaal, and H. Bal. A detailed gpu cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 37–48, Feb 2014.

[77] Chen Ding and Zhong Yuntao. Reuse Distance Analysis. In *Computer Science at University of Rochester Tech report UR-CS-TR-741*. U of Rochester, 2001.

[78] Jingweijia Tan, Shuaiwen Leon Song, Kaige Yan, Xin Fu, Andres Marquez, and Darren Kerbyson. Combating the reliability challenge of gpu register file at low supply voltage. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 3–15, New York, NY, USA, 2016. ACM.

[79] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, New York, NY, USA, 2010. ACM.

[80] P. Xiang, Y. Yang, and H. Zhou. Warp-level divergence in gpus: Characterization, impact, and mitigation. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 284–295, Feb 2014.

[81] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanović. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, Feb 2013.

[82] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. *SIGPLAN Not.*, 46(3):369–380, March 2011.

[83] Zheng Cui, Yun Liang, Kyle Rupnow, and Deming Chen. An accurate gpu performance model for effective control flow divergence optimization. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 83–94. IEEE, 2012.

[84] NVIDIA Corp. NVIDIA Nsight. http://www.nvidia.com/object/nsight.html, 2017.

[85] Daniel Hackenberg, Guido Juckeland, and Holger Brunst. Performance analysis of multi-level parallelism: inter-node, intra-node and hardware accelerators. *Concurrency and Computation: Practice and Experience*, 24(1):62–72, 2012.

[86] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. Identifying the root causes of wait states in large-scale parallel applications. *ACM Trans. Parallel Comput.*, 3(2):11:1–11:24, July 2016.

[87] Shuaiwen Leon Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 673–686. IEEE, 2013.

[88] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[89] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[90] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 51, page 11. ACM, 2016.

[91] Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John A Miller. Towards efficient query processing on massive time-evolving graphs. In *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 567–574. IEEE, 2012.

[92] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM, 2014.

[93] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[94] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs.

In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[95] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *KDD*, 2014.

[96] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *WWW*, 2015.

[97] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[98] Norman Biggs, Norman Linstead Biggs, and Biggs Norman. *Algebraic graph theory*, volume 67. Cambridge university press, 1993.

[99] GH Golub and CF Van Loan. Matrix computations. *The Johns Hopkins*, 1996.

[100] Oded Green and David A Bader. custinger: Supporting dynamic graph algorithms for gpus. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–6. IEEE, 2016.

[101] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[102] Chandler Davis. The norm of the schur product operation. *Numerische Mathematik*, 4(1):343–344, 1962.

[103] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[104] Dipanjan Sengupta and Shuaiwen Leon Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *International Supercomputing Conference*, pages 97–119. Springer, 2017.

[105] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. volume 11, pages 107–120. VLDB Endowment, September 2017.

[106] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In *WSDM*, 2018.

[107] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[108] Chang Zhou, Yuqiong Liu, Xiaofei Liu, Zhongyi Liu, and Jun Gao. Scalable graph embedding for asymmetric proximity. In *AAAI*, 2017.

[109] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *KDD*, 2016.

[110] Jiankai Sun, Bortik Bandyopadhyay, Armin Bashizade, Jiongqian Liang, P. Sadayappan, and Srinivasan Parthasarathy. Atp: Directed graph embedding with asymmetric transitivity preservation. In *AAAI*, 2019.

[111] Jiankai Sun and Srinivasan Parthasarathy. Symmetrization for embedding directed graphs. In *AAAI*, 2019.

[112] Peng Wang, BaoWen Xu, YuRong Wu, and XiaoYu Zhou. Link prediction in social networks: the state-of-the-art. *Science China Information Sciences*, 58(1):1–38, 2015.

[113] Ashton Anderson, Daniel Huttenlocher, Jon Kleinberg, and Jure Leskovec. Effects

of user similarity in social media. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 703–712, 2012.

[114] Prantik Bhattacharyya, Ankush Garg, and Shyhtsun Felix Wu. Analysis of user keyword similarity in online social networks. *Social network analysis and mining*, 1(3):143–158, 2011.

[115] Cuneyt Gurcan Akcora, Barbara Carminati, and Elena Ferrari. User similarities on social networks. *Social Network Analysis and Mining*, 3(3):475–495, 2013.

[116] Panagiotis Symeonidis and Nikolaos Mantas. Spectral clustering for link prediction in social networks with positive and negative links. *Social Network Analysis and Mining*, 3(4):1433–1447, 2013.

[117] Purnamrita Sarkar, Deepayan Chakrabarti, and Andrew W Moore. Theoretical justification of popular link prediction heuristics. In *IJCAI proceedings-international joint conference on artificial intelligence*, volume 22, page 2722. Citeseer, 2011.

[118] Mark EJ Newman. Clustering and preferential attachment in growing networks. *Physical review E*, 64(2):025102, 2001.

[119] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[120] Jorge Valverde-Rebaza and Alneu de Andrade Lopes. Exploiting behaviors of communities of twitter users for link prediction. *Social Network Analysis and Mining*, 3(4):1063–1074, 2013.

[121] Haifeng Liu, Zheng Hu, Hamed Haddadi, and Hui Tian. Hidden link prediction based on node centrality and weak ties. *EPL (Europhysics Letters)*, 101(1):18004, 2013.

[122] Rong-Hua Li, Jeffrey Xu Yu, and Jianquan Liu. Link prediction: the power of maximal entropy random walk. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1147–1156, 2011.

[123] Baojun Qiu, Kristinka Ivanova, John Yen, and Peng Liu. Behavior evolution and event-driven growth dynamics in social networks. In *2010 IEEE Second International Conference on Social Computing*, pages 217–224. IEEE, 2010.

[124] Baojun Qiu, Qi He, and John Yen. Evolution of node behavior in link prediction. In *AAAI*. Citeseer, 2011.

[125] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, pages 5165–5175, 2018.

[126] Jimeng Sun, Christos Faloutsos, Christos Faloutsos, Spiros Papadimitriou, and Philip S Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 687–696. ACM, 2007.

[127] Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John A Miller. Towards efficient query processing on massive time-evolving graphs. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, pages 567–574. IEEE, 2012.

[128] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[129] Shuo Ji, Yinliang Zhao, and Xiaomei Zhao. A low-latency computing framework for time-evolving graphs. *The Journal of Supercomputing*, 75(7):3673–3692, 2019.

[130] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, Sep. 2012.

[131] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[132] Anand Iyer, Li Erran Li, and Ion Stoica. Celliq : Real-time cellular network analytics at scale. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 309–322, Oakland, CA, 2015. USENIX Association.

[133] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 5:1–5:6, New York, NY, USA, 2016. ACM.

[134] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, page 9. ACM, 2018.

[135] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K John. Start late or finish early: a distributed graph processing system with redundancy reduction. *Proceedings of the VLDB Endowment*, 12(2):154–168, 2018.

[136] James King, Thomas Gilray, Robert M. Kirby, and Matthew Might. Dynamic sparse-matrix allocation on gpus. In *High Performance Computing*, Julian M. Kunkel, Pavan Balaji, and Jack Dongarra, editors, pages 61–80, Cham, 2016. Springer International Publishing.

[137] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th International Confer-*

ence on Parallel Architectures and Compilation Techniques (PACT), pages 233–245. IEEE, 2017.