Understanding and Optimizing the Performance of Heterogeneous Systems

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Engineering

by

Shuai Che

August 2012

APPROVAL SHEET

The dissertation

is submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

[illegible]

AUTHOR

The dissertation has been read and approved by the examining committee:

[illegible]

Advisor

[illegible]

[illegible]

[illegible]

[illegible]

Accepted for the School of Engineering and Applied Science:

[illegible]

Dean, School of Engineering and Applied Science

August

2012

Abstract

Heterogeneous computing with both CPUs and accelerators, such as GPUs, has become increasingly popular for general purpose computing. GPUs differ from CPUs significantly in architecture and programming models. GPUs provide high compute throughput and memory bandwidth, and offer dramatically better performance for many applications.

However, there is little previous work on understanding GPU application behaviors and how to map applications efficiently on the GPU. In addition, effective techniques are needed for both the CPU and the GPU to achieve an overall high performance. To better understand and optimize heterogeneous systems, we study the following research issues to address these concerns. These include 1) the design of the Rodinia benchmark suite for heterogeneous platforms including both the CPU and the GPU, 2) a detailed characterization of Rodinia benchmark suite, 3) the Dymaxion framework to optimize the memory access patterns of heterogeneous platforms, 4) an approach for spreading and balancing workloads across the CPU and the GPU, and 5) a methodology to predict the performance of GPU applications.

Acknowledgments

First of all, I would like to express my greatest thanks to my advisor, Prof. Kevin Skadron, for his tremendous guidance and support. His insights and constructive comments always help me improve the quality of my work and develop my research skills immensely. I am grateful to Kevin's direction and encouragement which led me overcome various challenges on my way of pursuing the Ph.D. degree.

I would like to thank my committee, Prof. Joanne Bechta Dugan, Prof. Sudhanva Gurumurthi, Prof. John Lach and Prof. Kamin Whitehouse, who spent much time reviewing my dissertation and gave me many invaluable comments about this work.

I also had an opportunity to work with Prof. Lieven Eeckhout and his group at Ghent University, Belgium in summer 2011. I enjoyed this visit a lot and am thankful to his advice on research and our IISWC'11 paper.

My dissertation work also benefits a lot from the collaboration with lava lab members: Jeremy W. Sheaffer, Michael Boyer, Lukasz Szafaryn, Jiayuan Meng and David Tarjan. I would like to thank all the contributors for the Rodinia benchmark suite.

I also thank my parents for making all this possible, and for their continuing support and understanding.

This dissertation work is supported by NSF grant nos. IIS-0612049, CNS-0916908, CNS-0615277 and CCF-1116673, SRC under task no. 1607, and grants from AMD Research, NVIDIA Research and NEC Labs America.

Contents

1 Introduction **1**

1.1 GPU Performance Optimization and Rodinia 4

1.2 GPU Workload Characterization 5

1.3 CPU-GPU Workload Spreading and Balancing 6

1.4 Memory Access Pattern Optimizations 7

1.5 GPU Performance Prediction 9

1.6 Contributions 10

2 GPU Computing and Rodinia **12**

2.1 A Short Primer for GPUs 12

2.2 Rodinia Benchmark Suite 16

3 A Characterization of Rodinia **21**

3.1 Characterization of Rodinia 22

3.2 Rodinia and Parsec 28

3.3 Analysis 31

3.4 Other Benchmark Suites 38

3.5 Conclusions and Future Work 38

4 Load Balancing **39**

4.1 Work Spreading and Balancing Framework 40

4.2 Methodology 45

4.3 Application and Experiment Results 46

4.4 Conclusions and Future Work 54

5 Dymaxion **56**

5.1 Motivation 58

5.2 Dymaxion Design and Implementation 61

5.3 Experiment Setup 69

5.4 Access Patterns and Experimental Results 70

5.5 Case Study: CPU-GPU Simultaneous Execution 82

5.6 Development Cost 84

5.7 Conclusions & Future Work 85

6 GPU Performance Prediction **87**

6.1 Motivations and Background 89

6.2 High-Level Framework 90

6.3 Methodology 94

6.4 Performance Prediction and Results 97

6.5 Other Potential Metrics 104

6.6 Related Work 105

6.7 Conclusions and Future Work 106

7 Conclusion **107**

7.1 Dissertation Summary 107

7.2 Future Research Challenges and Directions 110

Bibliography **114**

List of Figures

3.1 IPCs are measured over 8-shader and 28-shader configurations 23

3.2 Memory operation breakdown in terms of shared, texture, constant, parameter, and local or global memory instructions 24

3.3 Warp occupancies show the numbers of active threads in an issued warp over the entire runtime of the benchmark 25

3.4 Normalized kernel execution time of the GPU implementations on a GTX 280 and a GTX 480 27

3.5 A dendrogram showing the similarity between the Parsec (*P*) and Rodinia (*R*) workloads 32

3.6 The instruction mix plot using two PCA components for Parsec (dots) and Rodinia (crosses) 33

3.7 The working set plot using two PCA components for Parsec (dots) and Rodinia (crosses) 33

3.8 The sharing plot with two PCA components for Parsec ($\cdot$) and Rodinia ($+$) [17] . . 33

3.9 The miss rates of Rodinia and Parsec benchmarks under a 4 Mb cache configuration 34

3.10 The numbers of 64-byte instruction blocks touched during the program execution . 37

3.11 The number of 4 kB data blocks touched during the program execution 37

4.1 A graphical representation of parallelism within our layered approach 43

4.2 The scheduler dispatches data chunks dynamically on each device and achieves a desirable load-balanced ratio 44

4.3 Comparisons of performance of CPU-GPU communications with one chunk and multiple chunks 46
4.4 Normalized execution time of running *Kmeans* on the CPU, the GPU and the CPU/GPU simultaneously 47
4.5 Normalized execution time of running *SRAD* on the CPU, the GPU and the CPU/GPU simultaneously 49
4.6 Normalized execution time of running *Sepia* on the CPU, the GPU and the CPU/GPU simultaneously 50
4.7 The domain partitioning of Needleman Wunsch 51
4.8 Performance benefit of load balancing versus the CPU-GPU speedup gap 53
4.9 Change of execution time when we change the chunk size from 16 k to 64 k 54

5.1 The memory coalescing concept 59
5.2 The performance of Kmeans with row-major and column-major orders 59
5.3 Memory layout reorganization 62
5.4 CUDA streams are utilized to overlap chunk transfer with remapping. This example assumes that the entire work size can be evenly divided by the number of chunks . 64
5.5 Row-major to column-major transformation 71
5.6 The execution times of *k-means* for both the original and dymaxion implementations 72
5.7 Diagonal strip matrix transposition. 73
5.8 The execution times of *Needleman Wunsch* for both the original and dymaxion implementations 74
5.9 Indirect remapping for gather. 75
5.10 The execution times of *SpMV* for both the original and dymaxion implementations 75
5.11 The execution times of *NN* for both the original and dymaxion implementations . . 78
5.12 Speedups of kernels on GTX285 and GTX480 GPUs 79
5.13 The normalized throughput comparing chunking with synchronous memory transfers and chunking with asynchronous memory transfers (streaming) 80

5.14 The normalized performance of a row-major order to column-major order remapping as a function of the number of bytes (4–64 B) each thread reads and writes . . 81

5.15 The normalized performance results comparing three types of data transfers 82

5.16 The CPU and the GPU prefer different mappings. We divide the data structure into smaller chunks and schedule them onto the CPU and the GPU 84

5.17 The normalized execution time of the CPU-GPU simultaneous execution for one iteration of *k*-means 85

6.1 The framework for performance prediction 91

6.2 Changes of throughput varying the number of active threads 93

6.3 The total throughput degrades as the stride size increases 93

6.4 The predicted speedups (28 shaders v.s. 8 shaders) and measured speedups (Rodinia only) 98

6.5 The predicted speedups (28 shaders v.s. 8 shaders) and measured speedups (Rodinia, GPGPUsim and NVIDIA CUDA SDK benchmarks 98

6.6 Applications in the 3-D PCA space 99

6.7 Actual speedups (x-axis) and predicted speedups (y-axis) 100

6.8 The changes of prediction errors while increasing the number of nearest neighbors 102

6.9 The actual and predicted execution times (in cycles) of *SRAD* and *Needleman Wunsch* with different input sizes 102

List of Tables

2.1 Rodinia applications and their dwarfs and application domains 16

3.1 Rodinia applications and their input sizes 22

3.2 GPGPU-sim Configurations. 22

3.3 Incrementally optimized versions of SRAD and Leukocyte 26

3.4 Comparison between Parsec and Rodinia. 29

3.5 Parsec applications and sim-large input sizes 29

4.1 Load Balancing Applications. 45

4.2 Load Balancing Ratio of *Kmeans* 48

4.3 Load Balancing Ratio of *SRAD* 49

4.4 Load Balancing Ratio of *Sepia* 50

5.1 Fraction of total execution time devoted to PCI-E transfers 61

5.2 Total number of loads and stores of different application-input pairs reported by the CUDA profiler 79

5.3 Workload Ratios of *K-means* 84

5.4 Development cost measured by number of modified lines of code 85

6.1 Rodinia, GPGPUsim and NVIDIA CUDA SDK applications 95

6.2 Prediction errors and Spearman ranking results over different GPU configurations . 101

6.3 The three proxies and their weights for each benchmark 103

Chapter 1

Introduction

Microprocessors face increasing challenges in achieving high performance and efficiency due to severe power limits. Industry has shifted towards multi-core and many-core approaches over the past decade. One important trend is that computer systems are increasingly exposing a hybrid model consisting of accelerators -such as graphics processors (GPUs) - combined with multicore CPUs. GPUs, for instance, offer parallelism at scales not currently available with other processors and afford about an order of magnitude greater peak throughput than general-purpose, multicore CPUs, while the CPUs offer high single-thread performance. This provides the best combination of both high parallelism and high single-thread performance, to provide overall high-performance processing for computationally demanding tasks.

Heterogeneous computing overcomes the inherent limitations in homogeneous systems and provides a novel cost-effective approach to various problems, offering better compute efficiency and diverse capabilities. At the same time, it also opens up new challenges to computer architects and programmers. These include the increasing heterogeneity and consequent complexity of compute and memory structures, making performance portability a challenge. Designers will have to deal with architecture diversity and achieve a better trade-off among performance, power and cost. Furthermore, heterogeneous computing requires a friendly programming environment as well as efficient runtime and operating system support.

A vision of heterogeneous computer systems that incorporates diverse accelerators and selects the best computational unit for a particular task is widely shared among researchers and many industry analysts. To allow coordinated effective use of these compute units requires researchers to address a set of research questions first. For example, there is little previous research on understanding GPUs' application features, and how to adapt the applications efficiently onto the GPU for better performance. In addition, techniques are needed to spread workloads simultaneously across the CPU and the GPU to take advantage of the available compute resources efficiently. The CPU and the GPU may prefer different memory mappings, and how to achieve a good performance portability while minimizing the communication cost of maintaining heterogeneous layouts is also important. Furthermore, research in heterogeneous computing needs a set of well designed benchmarks, which were not previously available.

To address these concerns, this dissertation addresses two major research challenges:

1. Understanding heterogeneous systems. This includes understanding programming features, application characteristics and metrics important to performance.

2. Optimizing the performance of heterogeneous systems. High performance can be achieved if we are able to exploit the compute capabilities of each individual platform, while at the same time, ensure synergy among them for collaborative processing. We target heterogeneous systems with CPU and GPU in this work.

To optimize the performance of heterogeneous systems, this dissertation focuses on optimizing data mappings and data transfer across platforms. The main hypothesis of this work is that chunk-based dynamic load balancing and device-specific memory layout transformation improve performance for CPU-GPU systems. A second hypothesis is that performance of an arbitrary application can be accurately predicted across various GPU configurations using the performance of a set of benchmark applications, as long as essential characteristics of the application and benchmarks can be obtained.

To address these research challenges and prove our hypothesis, our approach consists of several specific research tasks:

1. Understanding the GPU programming features and investigating techniques for optimizing the GPU performance (my Master's thesis) and designing the Rodinia benchmark suite for

heterogeneous computing (partly in my Master's thesis and further extensions in this Ph.D. dissertation).

2. Performing a characterization of Rodinia and comparing it to other benchmarks.
3. Developing an approach to allow applications to run on both the CPU and the GPU simultaneously, and investigate the load-balancing issue.
4. Designing the Dymaxion framework to optimize the memory access patterns for heterogeneous systems.
5. Proposing a set of metrics and a methodology to predict the performance of GPU applications.

Each of the previous research tasks contributes an important part to the overall mission of understanding and optimizing the performance of heterogeneous systems:

To better *understand* heterogeneous systems, we first study how to take advantage of the GPU for general purpose computation and this analysis leads to the development of Rodinia (Task 1). We further conduct a characterization of Rodinia and analyze its diversity (Task 2). Part of this effort also includes studying how well the Rodinia suite is designed with a comparison to other benchmarks. We further study Task 5 for predicting the GPU performance. This can be used to evaluate how well we understand the GPU platform, if both the metrics and the methodology are effective for performance prediction.

To *optimize* the performance of heterogeneous systems and prove our hypothesis, we have pursued several directions. We take advantage of various techniques (e.g. data structure, algorithm and GPU-specific hardware features) to optimize the GPU performance (Task 1 and 4). To allow efficient CPU-GPU collaborative processing, we propose Task 3 to spread and balance workloads across the CPU and the GPU. Another important issue is that CPU and GPU may prefer different memory mappings. Task 4 studies this problem and proposes techniques to minimize the CPU-GPU communication overhead while transforming the data layout.

The following five sections demonstrate five research challenges in heterogeneous computing in details, along with their related works and our proposed approaches to address the concerns :

1.1 GPU Performance Optimization and Rodinia

In my masters thesis [13], we began to answer how to efficiently leverage the GPU platform for high-performance general purpose computation. We studied a diverse set of GPU applications with different parallel patterns [2] and explored various software and hardware strategies to optimize GPU programs. These include determining efficient algorithm mappings of applications' data structures to CUDA's domain-based model, and optimizing data-locality and memory access patterns implied by the implementation of their algorithms, in order to take advantage of a GPU memory hierarchy (e.g. specialized GPU memories). All of these findings and results were also documented in our JPDC'08 paper [15].

Before addressing all the research challenges we mentioned, researchers first need a set of well-designed workloads for their research. These workloads should span a diverse range of parallelism and compute patterns, and stress different GPU hardware components, providing researchers with various feature options to identify computer architecture bottlenecks and to fine tune hardware designs. There are many benchmark suites for parallel computing on general-purpose CPU architectures. Several multithreaded benchmark suites for multicore CPUs, including SPLASH-2 [82], Parsec [9], and SPEC OMP [71], are available. However, heterogeneous systems fall into a gap that is not covered by current benchmark suites or benchmark development. There are several GPU benchmarks available [4, 20, 75]. However, these benchmark suites do not include multithreaded CPU implementations, and were created without a formally-guided design methodology to ensure workload diversity and coverage of workload space.

Based on our experience developing and optimizing various GPU applications, we have constructed the Rodinia benchmark suite for heterogeneous computing to address these concerns (part in my Masters thesis and extensions in this dissertation). The Rodinia applications currently target both multicore CPUs and GPUs, using OpenMP, OpenCL and NVIDIA's CUDA. OpenMP is an API for multicore CPUs that allows programmers to use compiler directives to express parallelism. CUDA is NVIDIA's C extension and API for programming the GPU. OpenCL is a multi-vendor standard that targets heterogeneous systems.

Since we began tracking downloads in June 2010, Rodinia has received over 800 registered downloads, and the citations of Rodinia related publications are growing (e.g. 223 citations for the JPDC work [15], 74 citations for the SASP work [16] and 139 citations for the IISWC work [14]). The JPDC work was awarded "Top Cited Article 2005-2010" for Journal of Parallel and Distributed Computing. Rodinia is also being considered for inclusion in SPEC's new GPGPU suite [76]. The details of Rodinia will be described in Chapter 2.

1.2 GPU Workload Characterization

We endeavor to design Rodinia so that it is a useful tool for researchers to explore heterogeneous systems including the CPU and the GPU. To make effective use of it requires researchers to understand the characteristics of the suite itself, and the relationship of its program mix with other benchmark suites. However, there are some important questions yet to be answered:

- What features do the Rodinia benchmarks demonstrate in terms of important GPU and CPU metrics?
- How much do the Rodinia workloads designed for heterogeneous platforms (those with GPU accelerators) differ from those of other suites designed for multicore CPUs?
- How well do the chosen applications span and cover the workload space?

A better understanding of these issues will not only expand the knowledge of parallel benchmark construction, but could also inform decisions on workload scheduling and partitioning on different architectures and guide researchers to choose appropriate benchmarks for their research as well. Bienia *et al.* [8] compare SPLASH-2 and Parsec to determine the extent of feature overlap, and conclude that the workloads have significant differences. Many Parsec workloads have larger working sets than those in SPLASH-2, useful in the face of the scientific trend toward massive data growth. Other work compares the communication characteristics of SPLASH-2 and Parsec [81] and examines the behavior of Parsec on real hardware [7]. However, there is no previous work conducting a detailed characterization of heterogeneous workloads and comparing them to other workloads designed for the multicore CPU. Furthermore, to ensure a sufficient application coverage, a critical requirement for benchmark suite design, previous work performs studies on how well existing benchmarks cover application space, but only on single-threaded benchmarks [42, 66], while this is an open problem in heterogeneous environments.

In Chapter 3, we conduct a detailed characterization of the Rodinia benchmark suite and study its diversity from different perspectives. We also evaluate the Rodinia benchmarks on an NVIDIA GTX480, which is based on the Fermi architecture with traditional L1 and L2 caches. We subsequently perform an application space study, comparing the multithreaded CPU implementations of Rodinia with those of Parsec, and evaluate the extent to which the program selections of the two suites overlap. Finally, we present analysis and discussion of important, open research topics, including the need for an overall application space study of multithreaded workloads. We also discuss the challenges that make porting existing benchmarks (e.g. Parsec) difficult.

1.3 CPU-GPU Workload Spreading and Balancing

There has been growing research activity to explore how to efficiently utilize the available compute resources within a heterogeneous platform. One important issue is how to spread and schedule workloads simultaneously on the CPUs and the GPUs, while maintain a desirable load-balanced ratio.

GPUs differ significantly from CPUs in hardware architecture, memory hierarchy, etc. This in turns requires that the solutions to these issues be applicable in most of the practical circumstances given the diversity of the platforms. It should be also adaptable to hardware resource changes especially when multiple applications share common hardware resources. OpenCL has been released to support heterogeneous computing. However, the current release does not provide support for programmers to conduct efficient mappings and workload partitioning.

Twin Peaks [28] makes the OpenCL code originally targeted for GPUs efficiently run on CPUs. Stratton et al. [72] proposed MCUDA for mapping the NVIDIA CUDA kernels to run on multicore CPUs architectures. Saha et al. [69] develop a programming model for a heterogeneous x86 platform, consists of a general-purpose x86 CPU and Larrabee cores. Also, MapCG [34] is a MapReduce framework providing source code level portability between CPU and GPU. However, none of these works studies spreading workloads simultaneously on the CPU and the GPU. Intel's Merge framework [52] supports executing parallel compute kernels across devices. but relies on programmers to manually specify the mapping of a problem. This is not only tedious, but also not adaptable to the changes of hardware configurations or problem sizes [56]. Other works, such as the Qilin project [56], propose adaptive mappings. Qilin uses training runs to decide the amounts of works distributed onto the CPU and the GPU beforehand. However, this approach is not likely to perform well, when the available compute resources for an application change at runtime (e.g. competing resources with another application). In addition, no previous work takes into account that the CPU and the GPU may prefer different memory layouts and access patterns.

To overcome these limitations, in Chapter 4 we study an approach to execute parallel compute kernels simultaneously on the CPU and the GPU and allow efficient data partitioning and load balancing across devices. Our approach is based on hierarchical domain partitioning with chunking and work queues which keep track of the processing chunks. It has the advantage of being able to adjust to run-time compute resource changes. We also include scheduling strategies to ensure better data locality in order to save costly PCI-E transfers. Also, we apply memory remapping to achieve desirable data layouts for diverse compute devices (See Chapter 5).

1.4 Memory Access Pattern Optimizations

Memory bandwidth and latency present serious concerns that limit throughput in GPU architectures. Together with SIMT branch divergence [27, 84], they are two of the distinctive characteristics that limits the GPU performance. The memory bandwidth issue is getting relatively worse, as the number of processing elements per chip is growing much faster than bandwidth and latency are improving. Furthermore, the performance relies on effective memory bandwidth utilization [41, 77, 84].

Furthermore, an application's algorithmic behavior, as viewed by the programmer, does not necessarily lead to the most efficient memory access pattern. Today's GPU programming models require programmers to invest considerable manual effort to optimize memory accesses for high performance. For instance, GPUs' SIMD architectures require efficient memory coalescing for inter-thread data locality. Hybrid memory units—such as the GPU's *shared*, *constant*, and *texture* memories—present access patterns that are unfamiliar and unintuitive to programmers and that favor specific, specialized mappings. However, code optimized for specialized access patterns may not perform well or be portable across multiple vendors' platforms and different hardware generations. Additionally, for efficient heterogeneous computing, different architectures and multithreading models may favor different memory mappings. For example, SIMD organizations generally perform best when each thread or lane of a SIMD operation accesses adjacent data, while scalar organizations perform best when a single thread accesses adjacent data. This in turn requires heterogeneity in data layout as well as per-device optimization for simultaneous execution. A high-level abstraction to define memory mappings is needed for heterogeneous computing.

Previous work investigates how to optimize data organization for efficient memory accesses. An early report by Leung and Zahorjan [49] discusses how array elements should be laid out in memory to improve spatial locality for accesses in nested loops of CPU programs. Impulse [85] proposes application-specific optimizations through configurable physical address remapping by supporting prefetching at the memory controller. Sung et al. [77] investigated a compiler approach for layout transformation for GPU kernels, focusing merely on structured-grid applications. Jang et al. [41] use a mathematical model and associative algorithms to analyze data access patterns and target loop vectorization and GPU memory selection with different patterns. The linear, shifted, and strided access patterns [41] can be handled by either the row-major or column-major mapping. Zhang et al. [84] proposes a dynamic approach to reduce irregularities in GPU programs. However, these approaches maintain many duplicate data copies for fine-grained data reordering, and sometimes requires breaking the integrity of compute kernels. In addition, there is no previous works studying the memory remapping and related latency-hiding techniques, which are implemented during CPU-GPU data communication. Furthermore, none of the previous work evaluates the memory mapping issue when scheduling workloads simultaneously on both the CPU and the GPU, and studies the portability issue.

To address these concerns, in Chapter 5, we proposed a high-level abstraction to define memory mappings and access patterns for heterogeneous computing. We developed a framework, Dymaxion, and associated techniques to allow programmers to easily optimize the access patterns for better memory bandwidth utilizations. We further created a prototype API implementation and runtime, which optimize memory layouts accordingly and then transform subsequent memory access addresses as necessary. To minimize the layout transformation overhead across devices, we hid the latency through chunking data structures and overlapping their data transfers with layout reorganizations. With Dymaxion, applications with a variety of access patterns achieve significant speedups, while maintaining better portability. We anticipate that the techniques we proposed in Dymaxion can be further integrated into compiler frameworks for automatic memory remapping and transformation. In addition, for efficient CPU-GPU load balancing, data layouts of some applications must be transformed across devices so that memory accesses are optimized for both platforms. This is achieved by applying Dymaxion to each data chunk in the load balancing framework.

1.5 GPU Performance Prediction

It is important for researchers and users to understand and be able predict the performance of an application of interest running on the GPU. Users need to determine the platform or product achieving the best performance for their applications of interest to make appropriate purchasing decisions. Of course, the applications of users' interest are their best benchmarks. However, sometimes users need to rely on the performance of standardized benchmarks, microbenchmarks and synthetic benchmarks to estimate the performance of their applications [37], which motivates the need to evaluate the design of today's GPU benchmark suites to achieve this goal. Studying the approaches for performance prediction can also help researchers to understand first order parameters/metrics capturing the behaviors of the GPU platform.

Previous research works have explored the issues of analyzing and predicting application performance for CPUs [12, 29, 37, 70]. However, they mostly concentrate on single-thread applications. Recently, researchers have started to study performance prediction for GPUs. Some researchers build analytical models with detailed GPU hardware parameters [3, 35] as inputs. However, one challenge is the difficulty to keep up with GPUs' evolution, which invalidates highly tuned analytical models. Other works use regression methods to construct empirical models [45].

In Chapter 6, we study an approach of predicting the performance of target GPU applications by correlating with characteristics of existing workloads. We first identify a set of important GPU application characteristics and further use them to predict the GPU performance by determining the most similar benchmarks to the target application. Our prediction technique extends the prior work by Host et al. [37] to support manycore architectures. As part of our analysis, we discuss the effectiveness of using Rodinia only, as well as including other suites, to achieve the necessary accuracy. Finally, we present several imperative issues for determining mutual relationships among benchmarks and a more systematic way for future benchmark suite construction in general.

1.6 Contributions

In summary, to understand and optimize the performance of heterogeneous systems, we research the following research problems in this dissertation:

- In Chapter 2, we develop the Rodinia benchmark suite to fill the gap of benchmarking heterogeneous platforms including both the CPU and the GPU.
- In Chapter 3, we characterize the Rodinia benchmarks and show the diverse behaviors exhibited. We compare Rodinia with other benchmark suites (e.g. Parsec) and evaluate how Rodinia differ from the suites designed for multicore CPUs.
- In Chapter 4, we study an approach to allow applications to run on both the CPU and the GPU simultaneously and study the load-balancing issue.
- In Chapter 5, we design a set of APIs, layout/index transformation and latency hiding techniques to optimize memory access patterns for heterogeneous systems.
- In Chapter 6, we design and evaluate the methodology of predicting the performance of a GPU application by correlating to that of existing benchmarks.

Our research tasks and contributions are in alignment with resolving the overall dissertation. To understand heterogeneous systems, we study how to leverage GPUs for general purpose computation, characterize the Rodinia workloads and analyze their diversity, and further propose a set of metrics and a methodology for GPU performance prediction. We optimize the performance of heterogeneous systems from three perspectives: 1) exploring techniques to optimize the GPU performance, 2) developing a framework to efficiently spread and balance workloads across the CPU and the GPU, and 3) maintaining heterogeneous layouts and minimizing the cross-platform data communication overhead during layout transformation.

Chapter 2

GPU Computing and Rodinia

In this chapter, we give an overview of GPU architectures and their programming models (e.g. CUDA and OpenCL). And then we introduce the Rodinia benchmark suite developed for heterogeneous systems including multicore CPUs and GPUs.

2.1 A Short Primer for GPUs

2.1.1 NVIDIA Tesla and Fermi

NVIDIA's unified computing architecture is designed to support both graphics and general-purpose computing. The programmable processing elements share a common, very general-purpose instruction set that is used by both graphics and general-purpose computation. Each processing element is designed to be a very simple pipeline supporting multiple, time-multiplexed thread contexts. Latencies are simply tolerated by switching threads. Current GPU products support many concurrent threads. Nickolls *et al.* [62] and Lindholm *et al.* [53] provide a nice description of contemporary NVIDIA GPU architectures.

For instance, in a NVIDIA GTX 280 GPU, each streaming multiprocessor (SM) consists of 8 processing elements, called *Stream Processors* or SPs. Each SM also has 16 kb on-chip, software-managed per-block shared memory (PBSM) or scratchpad memory. To maximize the number of processing elements that can be accommodated within the GPU die, these 8 SPs operate in SIMD fashion under the control of a single instruction sequencer. The threads in a thread block are time-sliced onto these 8 SPs in groups of 32 called *warps*. Each warp of 32 threads operates in lockstep and these 32 threads are time-sliced on the 8 SPs. Multithreading is then achieved through a hardware thread scheduler in each SM. Every cycle this scheduler selects the next warp to execute. Divergent threads are handled using hardware masking until they reconverge. Different warps in a thread block need not operate in lockstep, but if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously. In the worst case, all 32 threads in a warp follow different paths without reconverging—resulting in a sequential execution of the threads across the warp. NVIDIA's GPU architecture is optimized for workloads with relatively little data locality and only very localized data reuse. As a consequence, it does not provide large hardware caches shared among multiple cores, as is the case on modern CPUs.

To support more general purpose workloads, especially those with irregular access patterns, NVIDIA introduced a new architecture code-named Fermi, integrating traditional hardware caches. For example, NVIDIA GeForce GTX 480 GPUs have 15 streaming multiprocessors with a total of 480 1.4 GHz streaming processors, and a 768 kB unified L2 cache. Each SM has a 64 kB, configurable, on-chip memory that can be configured as 48 kB shared + 16 kB L1 or as 16 kB shared + 48 kB L1. Another important feature of Fermi cards is their capability to support concurrent kernel executions and ECC memory [62].

In addition to the PBSM and hardware caches, each SM has two small, private data caches, both of which only hold read-only data: the texture cache and the constant cache. Data structures must be explicitly allocated into the PBSM, constant, and texture memory spaces. The texture cache are optimized for arbitrary access patterns to achieve the best average performance. The constant cache is optimized for broadcasting values to all PEs in an SM and performance degrades linearly if PEs request multiple addresses in a given cycle. The best performance will be achieved when all the threads within a warp touch the same cache line.

2.1.2 A Comparison with AMD GPU Architectures

The overall design concepts of NVIDIA and AMD GPUs are similar to each other. However, one feature which makes AMD GPUs differ from NVIDIA products is their adoption of very long instruction word (VLIW) processors to carry out computations in a vector-like fashion [87]. For instance, the AMD Cypress GPU (i.e. Radeon 5870) consists of 20 SIMD engines. Inside a SIMD engine, there are 16 thread processors (TP) and 32KB local memory. A SIMD engine is equivalent to a stream multiprocessor on an NVIDIA GPU, while the local memory is equivalent to the shared memory on an SM. Each SIMD engine also includes a texture unit with 8KB L1 cache. Unlike the NVIDIA GPU SPs, a thread processor within a SIMD engine is a five-way VLIW processor: each thread processor consists of five processing elements (four ALUs and a special function unit). In each cycle, independent instructions are assigned to these processing elements as a VLIW bundle and are simultaneously executed. Therefore, effective uses of AMD GPUs rely on effective compilers to generate sufficient instructions to fill the pipelines or require programmers to explicitly develop programs to achieve high packing ratios (e.g. *float4*, *int4*). Another difference between NVIDIA and AMD GPU architectures is that global memory accesses go through the L2 in NVIDIA's Fermi architectures, while only image objects and constants use the L2 in Cypress [87].

2.1.3 The CUDA Programming Model

CUDA is a C extension and API for parallel programming. CUDA represents the coprocessor as a device that can run a large number of threads. The threads are managed by representing parallel tasks as kernels mapped over a computation domain. Kernels are *scalar* and represent the work to be done at a single point in the domain. The kernel is then invoked as a thread at every point in the domain. The parallel threads share memory and synchronize using barriers.

Data is prepared for processing on the GPU by copying it to the graphics board's memory in a system with a discrete GPU. Data transfer is performed using DMA and can take place concurrently with kernel processing. AMD Fusion [1] products integrate both the CPU and the GPU on the same die, which share a single physical memory and memory channel. However, currently CPU and GPU still work on their own memory spaces and data must still be copied between different spaces.

GPU's computation domain is actually defined with a multidimensional structure, in the form of a 3D *grid* of 3D *thread blocks*. The significance of the thread block construct is that each thread block is assigned in its entirety to a single *streaming multiprocessor* and runs as a unit to completion without preemption. The number of thread blocks in a grid can greatly exceed the hardware resources, in which case fresh thread blocks are assigned to SMs as previous thread blocks retire.

In addition to global shared memory, each thread block has available a private, *per-block shared memory* that is only visible to threads within that thread block. The amount of this PBSM that will be used must be defined by the kernel but is limited to 16 KB (or 48 kb in Fermi, depending on configurations). The PBSM allows threads within a thread block to cooperate in a fine-grained fashion by sharing data among themselves with low latency.

CUDA uses a relaxed memory consistency model [33]. Synchronization *within* a thread block (i.s. syncthread()) is entirely managed in hardware. Synchronization *among* thread blocks is achieved by allowing a kernel to complete and starting a new kernel; in effect, a global barrier. It is important to note that the order in which thread blocks are assigned to SMs is arbitrary. Because order of execution among thread blocks within a grid is non-deterministic, and because thread blocks run to completion, it is important to note that thread blocks should never have a producer-consumer relationship due to the risk of deadlock. Producer-consumer relationships must be confined within thread blocks or separated across global barriers.

2.1.4 OpenCL

OpenCL [64] has been released as a unified framework designed for GPUs and other processors. CUDA and OpenCL models have much similarity in the virtual machines they define. Most techniques for optimizing applications in CUDA can be translated easily into those in OpenCL. The OpenCL platform model is based on compute devices that consist of compute units with processing elements, which are equivalent to CUDA's SM and SP units. In OpenCL, a host program launches a kernel with *work-item*s over an index space, and *work-item*s are further grouped into *work-group*s (thread blocks in CUDA). Also, the OpenCL memory model has a similar memory hierarchy as CUDA, such as the global memory space shared by all *work-group*s, the per-*work-group* local memory space, the per-*work-item* private memory space, etc. The global and constant data cache can be used for data which take advantage of the read-only texture and constant cache in CUDA.

Table 2.1: Rodinia applications and their dwarfs and application domains

Application	Dwarf	Domain
Kmeans	Dense Linear Algebra	Data Mining
Needleman-Wunsch (NW)	Dynamic Programming	Bioinformatics
HotSpot (HS)	Structured Grid	Physics Simulation
Back Propagation (BP)	Unstructured Grid	Pattern Recognition
SRAD*	Structured Grid	Image Processing
Leukocyte Tracking (LC)	Structured Grid	Medical Imaging
Breadth-First Search (BFS)	Graph Traversal	Graph Algorithms
Stream Cluster (SC)	Dense Linear Algebra	Data Mining
MUMmer (MUM)	Graph Traversal	Bioinformatics
CFD Solver (CFD)	Unstructured Grid	Fluid Dynamics
LU Decomposition (LUD)	Dense Linear Algebra	Linear Algebra
Heart Wall Tracking (HW)	Structured Grid	Medical Imaging

Finally, OpenCL adopts a "relaxed consistency" memory model similar to CUDA. Local memory consistency is ensured across *work-item*s within a *work-group* at a barrier but not guaranteed across different *work-groups*.

2.2 Rodinia Benchmark Suite

For my master thesis [13], we studied various approaches of optimizing GPU applications for higher performance including algorithmic mappings, data structures and layouts and memory access patterns (e.g. DRAM coalescing and GPU's hybrid memory spaces). Extending this work, we developed Rodinia [1] to address the issues of benchmarking heterogeneous systems including the CPU and the GPU. Several multithreaded benchmark suites for multicore CPUs, including SPLASH-2 [82], Parsec [9], and SPEC OMP [71], are available. Rodinia was developed to address the needs of benchmarking heterogeneous systems, particularly those including a GPU.

To choose applications for Rodinia, the Berkeley Dwarf taxonomy [2] was initially used as a guideline so that we do not miss any important parallel compute patterns. Table 3.1 illustrates some Rodinia applications used in this study and their corresponding domains and Dwarves. Rodinia is released open-source and maintained online at http://lava.cs.virginia.edu/wiki/rodinia.

The Rodinia benchmarks are currently implemented in OpenMP, CUDA and OpenCL [64]. OpenCL and CUDA use very similar sets of abstractions, such that CUDA is sufficient for the

[1]Rodinia is a supercontinent that existed between 1.1 billion and 750 million years ago. It was a supercontinent which combined all of Earth's continents today, so we use Rodinia to describe the concept of a benchmark suite for heterogeneous processor organizations.

characterization and diversity analysis presented in this dissertation. We expect that our reported results will transfer directly to the OpenCL ports when they are complete. Rodinia has the following important features:

- Rodinia is the first benchmark suite in academia for heterogeneous computing including implementations for both the CPU and the GPU.
- Rodinia includes applications from emerging domains such as bioinformatics, data mining, and image processing, as well as the accelerator implementations of traditional algorithms (LU decomposition and graph traversal).
- Rodinia implementations take advantage of non-traditional memory hierarchies, such as scratchpad and texture units, for general purpose computation. Cell and ClearSpeed are two examples in a trend to use other types of memories as alternatives to hardware-managed cache. This trend in turn requires benchmark development to keep up with such an evolution.
- Rodinia provides multiple versions of some applications, with successive layers of optimization, allowing designers to evaluate the impact of multiple different implementations on their architecture or compiler designs. This is important to evaluate how architectures support different programming styles and efforts.
- Rodinia's applications currently adopt an "offloading" model which assumes that accelerators use a memory space disjoint from main memory.
- We applied both algorithmic optimizations (e.g. ghost-zone technique and persist thread block) hardware-level optimizations (e.g. using GPU shared, texture and constant memories) for Rodinia. This has been well documented in our JPDC and IISWC works [14, 15].
- We also did a study by porting three Rodinia applications on FPGA [16], and discuss how efficiently a set of important compute operations execute on a GPU compared to an FPGA.

Parboil [75] and SHOC [20] are two other efforts to benchmark GPUs. However, most of the Parboil and SHOC workloads are simple kernels and no diversity analysis for their workloads is provided. Goswami et al. [22] compared NVIDIA CUDA SDK, Parboil and Rodinia with various GPGPU workload characteristics using a clustering analysis similar to Chapter 3, Section 3.2, and concluded Rodinia is the most diverse among three suites.

2.2.1 Workloads

The Rodinia benchmark suite currently includes the following benchmarks. Among these benchmarks, *HW*, *LUD*, *MUM* and *CFD* and the incremental versions for several benchmarks were introduced as an extension to Rodinia in our IISWC'10 [17] work, after its initial release in IISWC'09 [14] (part of my master work). Another major extension is the inclusion of the Rodinia OpenCL versions. Note that I only developed *SRAD*, *HS*, *BP*, *NW* and *KM*. The rest were contributed by others. We briefly describe each application we used in the experiments of this study:

Leukocyte Tracking (LC) detects and tracks rolling leukocytes (white blood cells) in video microscopy of blood vessels [11]. In the application, cells are detected in the first video frame and then tracked through subsequent frames. The major processes include computing for each pixel the maximal Gradient Inverse Coefficient of Variation (GICOV) score across a range of possible ellipses and computing, in the area surrounding each cell, a Motion Gradient Vector Flow (MGVF) matrix.

Speckle Reducing Anisotropic Diffusion (SRAD) is a diffusion algorithm based on partial differential equations and used for removing the speckles in an image without sacrificing important image features. *SRAD* is widely used in ultrasonic and radar imaging applications. The inputs to the program are ultrasound images and the value of each point in the computation domain depends on its four neighbors.

HotSpot (HS) is a thermal simulation tool [39] used for estimating processor temperature based on an architectural floor plan and simulated power measurements. Our benchmark includes the 2D transient thermal simulation kernel of *HotSpot*, which iteratively solves a series of differential equations for block temperatures. The inputs to the program are power and initial temperatures. Each output cell in the grid represents the average temperature value of the corresponding area of the chip.

Back Propagation (BP) is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The application is comprised of two phases: the Forward Phase, in which the activations are propagated from the input to the output layer, and the Backward Phase, in which the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values. Our parallelized versions are based on a CMU implementation [21].

Needleman-Wunsch (NW) is a global optimization method for DNA sequence alignment. The potential pairs of sequences are organized in a 2-D matrix. The algorithm fills the matrix with scores, which represent the value of the maximum weighted path ending at that cell. A trace-back process is used to search the optimal alignment. A parallel *Needleman-Wunsch* algorithm processes the score matrix in diagonal strips from top-left to bottom-right.

K-means (KM) is a clustering algorithm used extensively in data mining. This identifies related points by associating each data point with its nearest cluster, computing new cluster centroids, and iterating until convergence. Our OpenMP implementation is based on the Northwestern MineBench [67] implementation.

Stream Cluster (SC) solves the online clustering problem. For a stream of input points, it finds a pre-determined number of medians so that each point is assigned to its nearest center [9]. The quality of the clustering is measured by the sum of squared distances (SSQ) metric. The original code is from the Parsec Benchmark suite developed by Princeton University [9]. We ported the Parsec implementation to CUDA and OpenMP.

Breadth-First Search (BFS) traverses all the connected components in a graph. Large graphs involving millions of vertices are common in scientific and engineering applications. The CUDA version of *BFS* was contributed by IIIT [30].

LU Decomposition (LUD): LU Decomposition is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. This application has many row-wise and column-wise interdependencies and requires significant optimization to achieve good parallel performance.

Heartwall Tracking (HW): The Heart Wall [78] application tracks the changing shape of the walls of a mouse heart over a sequence of 104 ultrasound images, each with a resolution of 609×590 pixels. In its initial stage, the program performs several image processing passes—edge detection, SRAD despeckling (part of Rodinia), morphological transformation, and dilation—on the first image in the sequence in order to detect partial shapes of inner and outer heart walls. To reconstruct approximated full shapes of heart walls for tracking purposes, the application generates ellipses that are superimposed over the image and sampled to mark points on the heart walls. In its final stage, the program tracks the changing shapes of the two heart walls by detecting the movement of certain sample points throughout the sequence of images.

Computational Fluid Dynamics (CFD): The CFD solver is an unstructured-grid, finite-volume solver for the three-dimensional Euler equations for compressible flow. Effective GPU memory bandwidth is improved by reducing total global memory accesses and overlapping redundant computation, as well as by using an appropriate numbering scheme and data layout. The CFD solver is released with two versions: one with precomputed fluxes, and the other with redundant flux computations. CFD is an implementation of the work by Corrigan *et al.* [19].

MUMmerGPU (MUMmer): MUMmerGPU, developed by Schatz *et al.* [68], is an high-throughput, parallel, pairwise, local-sequence alignment program. It uses the GPU to simultaneously align multiple query sequences against a single reference sequence stored as a suffix tree encoded with 2-D textures. The tree of the reference sequence is constructed on the CPU using Ukkonen's Algorithm [80] and transferred to the GPU along with the query sequences. The query sequences are then transfered to the GPU, and are aligned with the tree on the GPU.

Chapter 3

A Characterization of Rodinia

In this chapter, we characterize the Rodinia benchmark suite on both real hardware and a simulator to better understand its program mix. Rodinia was developed to address the issues of benchmarking heterogeneous systems, particularly those including a GPU. There is growing use of the Rodinia workloads (e.g. 223 citations for the JPDC work [15], 74 citations for the SASP work [16] and 139 citations for the IISWC work [14]) , but there are some important questions yet to be answered –for instance, how much the Rodinia workloads designed for heterogeneous platforms differ from those of other suites designed for multicore CPUs, how well the chosen applications span the workload space, and how well traditional, multithreaded CPU workloads can map onto GPU platforms.

To address these concerns, this chapter makes the following contributions:

- We conduct a detailed characterization of the Rodinia GPU workloads to aid researchers in understanding the characteristics of Rodinia.
- We evaluate the Rodinia benchmarks on an NVIDIA GTX480, which is based on the Fermi architecture with traditional L1 and L2 caches.
- We perform an application space study, comparing the multithreaded CPU implementations of Rodinia with those of Parsec, and evaluate the extent to which the program selections of the two suites overlap.

Table 3.1: Rodinia applications and kernels ('*' denotes kernel).

Application	**Dwarf**	**Domain**	**Problem Sizes**
Kmeans	Dense Linear Algebra	Data Mining	204800 data points, 34 features
Needleman-Wunsch (NW)	Dynamic Programming	Bioinformatics	2048×2048 data points
HotSpot* (HS)	Structured Grid	Physics Simulation	500×500 data points
Back Propagation* (BP)	Unstructured Grid	Pattern Recognition	65536 input nodes
SRAD*	Structured Grid	Image Processing	512×512 data points
Leukocyte Tracking (LC)	Structured Grid	Medical Imaging	219×640 pixels/frame
Breadth-First Search* (BFS)	Graph Traversal	Graph Algorithms	1000000 nodes
Stream Cluster* (SC)	Dense Linear Algebra	Data Mining	65536 points, 256 dimensions
MUMmer (MUM)	Graph Traversal	Bioinformatics	50000 25-character queries
CFD Solver (CFD)	Unstructured Grid	Fluid Dynamics	97k elements
LU Decomposition* (LUD)	Dense Linear Algebra	Linear Algebra	256×256 data points
Heart Wall Tracking (HW)	Structured Grid	Medical Imaging	609×590 pixels/frame

- We present analysis and discussion of important, open research topics, including the needs for a general application space study of multithreaded workloads and for new parallel performance metrics, and we discuss the challenges that make porting existing suites difficult.

The work of this chapter has been published in IISWC 2010 [17]. Some other characterizations and diversity analysis of Rodinia (e.g. parallel speedup, execution time break down, MICA diversity [36] and power consumption) are documented in our IISWC 2009 paper (part of my master work) [14].

Table 3.2: GPGPU-sim Configurations.

Parameter	**Value**	**Parameter**	**Value**
Clock Frequency	2 GHz	No. of CTAs/Core	8
No. of SMs	28	Number of Registers/Core	16384
Warp Size	32	Shared Memory/Core	32 kB
SIMD pipeline width	32	Shared Memory Bank Conflict	True
No. of Threads/Core	1024	No. of Memory Channels	8

3.1 Characterization of Rodinia

In this section, we characterize Rodinia's applications in terms of instructions per cycle (IPC), memory instruction mix, and warp divergence. Our analysis shows that the Rodinia applications demonstrate good diversity, and the addition of new benchmarks enrich the application coverage of the previous Rodinia release [14]. We also use Rodinia to benchmark the NVIDIA GeForce GTX480 GPU (Fermi) targeting each of L1 and shared memory as preferred configurations. Table 3.1 illustrates the Rodinia applications and their input sizes we use in this study.

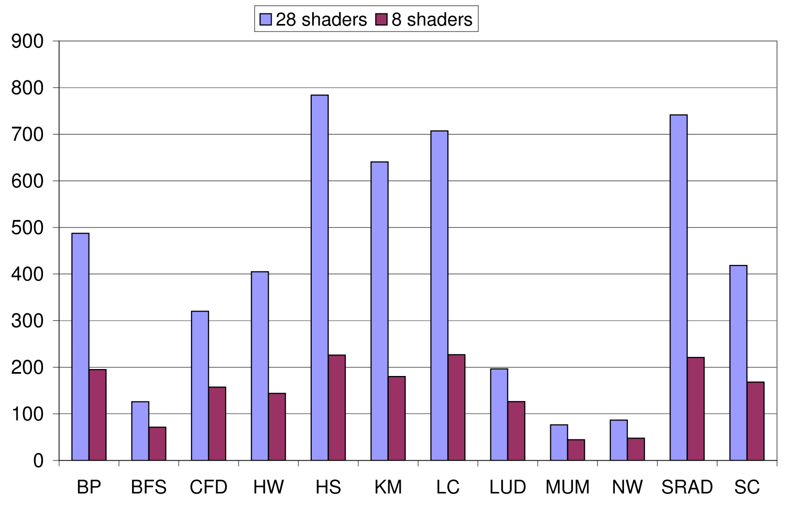

Figure 3.1: IPCs are measured over 8-shader and 28-shader configurations [17].

3.1.1 Experiment Setup

To measure program characteristics and architectural behaviors of the Rodinia GPU benchmarks, we use GPGPU-Sim [4] from the University of British Columbia. GPGPU-Sim provides a detailed simulation model of a contemporary GPU capable of running CUDA and OpenCL workloads. Table 3.2 shows the parameters we used to configure the simulator. Our GPGPU-Sim simulations did not use an L2 cache. Table 3.1 shows the input sizes of the Rodinia applications we used for simulations. To measure Rodinia on real hardware, we use an NVIDIA GeForce GTX480 with 15 streaming multiprocessors (SMs) with a total of 480 1.4 GHz streaming processors (SPs), and a 768 kB L2. Each SM has a 64 kB, configurable, on-chip memory that can be configured as 48 kB shared + 16 kB L1 or as 16 kB shared + 48 kB L1. We use NVIDIA CUDA 2.2 for the GPGPU-Sim simulations; for the GTX 480 experiments, we use CUDA version 3.0.

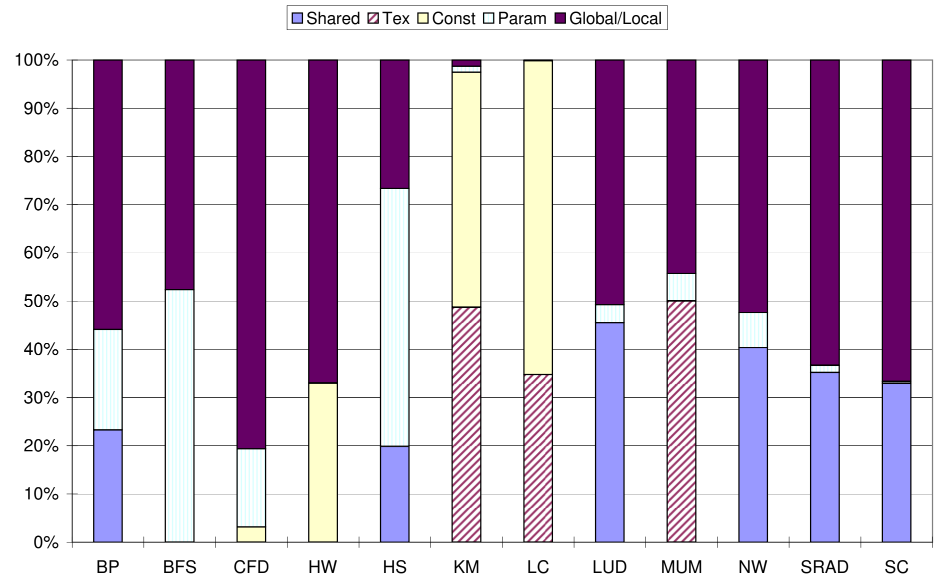

Figure 3.2: Memory operation breakdown in terms of shared, texture, constant, parameter, and local or global memory instructions [17]. "Param" memory refers to parameters passed through the GPU kernel call, which we always treat as cache hits [4].

3.1.2 GPU Benchmark Results

Figure 3.1 shows the IPCs of each of the Rodinia benchmarks measured with 28-shader—the default configuration provided by the GPGPU-Sim package [4] with a SIMD width of 32—and 8-shader configurations. The IPCs with the 28-shader configuration range from less than 100 in *MUMmer* and *Needleman-Wunsch*) to more than 700 in *SRAD*, *HotSpot*, and *Leukocyte*. The highest IPCs are usually due to massive parallelism, better usage of memory locality, and good algorithmic optimization [11,14,58,78]. Low IPC can be attributed to any of myriad faults: there is limited parallelism per iteration in *Needleman-Wunsch* due to the dependencies of processing data elements in a diagonal strip manner [15]; the overhead of the GPU's global memory accesses dominates some applications (e.g. *CFD* and *Breadth-First Search*); and other applications present many divergent branches.

Many Rodinia benchmarks take advantage of the GPU's specialized memory spaces by localizing data access patterns and inter-thread communication within thread blocks to take advantage of the SM's per-block shared memory. For read-only data structures, binding to cached constant or

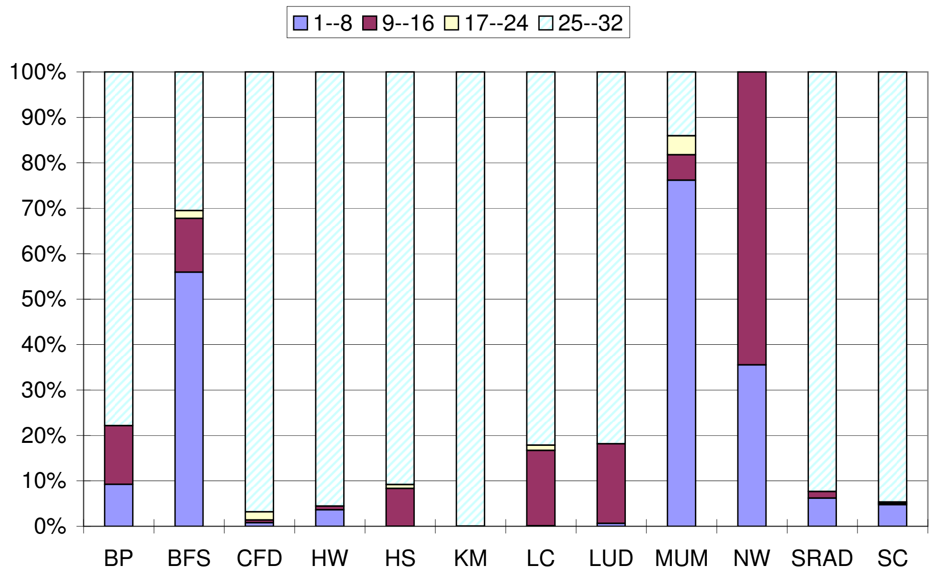

Figure 3.3: Warp occupancies show the numbers of active threads in an issued warp over the entire runtime of the benchmark [4] [17].

texture memory to reap the benefits of caching can provide significant performance improvements. Figure 3.2 shows a breakdown of different types of memory accesses. Applications such as *Back Propagation*, *HotSpot*, *Needleman-Wunsch* and *StreamCluster* make extensive use of shared memory. The performances of *Kmeans*, *Leukocyte* and *MUMmer* are improved by taking advantage of texture memory. Different from *Kmeans* and *Leukocyte*, *Heartwall* uses constant memory to store large numbers of parameters which cannot be readily fit into shared memory.

Figure 3.3 shows warp occupancies [4]—the average number of active threads over all issued warps—over the entire runtime of the benchmarks. In a SIMT model [61], the cores will achieve the best performance when the threads within a SIMT group follow the same execution path. For example, because it must determine whether or not neighboring nodes have been visited, *Breadth-First Search* contains many control flow operations; hence the high number of low occupancy warps. *SRAD* does not have much control flow; what of it there is deals with the loading and processing data elements lying on the boundaries between data blocks. *Heartwall* must determine the specific operations to execute on the various regions of the image, but this requires a relatively small portion of the calculation, and the rest of the computation executes with little control flow.

For other applications, unfilled warps are not due to branch divergence. Only some of the threads in *Back Propagation* are active, due to the parallel reduction; assuming a 16-data-element sum reduction, the number of active threads during the four iterations are 8, 4, 2 and 1. A similar situation occurs in *Needleman-Wunsch*, where, in each thread-block, the number of active threads is less than 16. *MUMmer* experiences severe performance loss in particular because more than 60% of its warps have less than 5 active threads [4].

3.1.3 Incrementally Optimized Versions

One important distinguishing characteristic of Rodinia is its support for multiple versions of individual benchmarks (*incremental versions*). Incremental versions are useful tools for architects and compiler developers because they allow analysis of the impact of hardware and software design choices on problems that are fundamentally the same but differ in certain specifics. Incremental versions can be used by programmers and compiler developers as “roadmaps” for similar problems, to aid them in getting from unoptimized to optimized or to evaluate their own optimizations.

We have released incremental code versions of *Leukocyte*, *LUD*, *Needleman-Wunsch* and *SRAD*. Table 3.3 shows sample characteristics of two different versions of *SRAD* and *Leukocyte*. We apply more shared memory optimization on the second version of SRAD, thus increasing the IPC from 404 to 748. Similarly, the performance of Leukocyte version 2 is improved by reducing the percentage of long latency global memory accesses through the use of persistent thread blocks. Boyer *et al.* provide a detailed study on the optimization of *Leukocyte* [11].

Table 3.3: Incrementally optimized versions of SRAD and Leukocyte [17].

Benchmarks	**Statistics**	
SRAD	Version 1	IPC: 404, BW Utilization: 26% Shared: 9.7%, Global: 49.3% (Mem. inst. mix)
	Version 2	IPC: 748, BW Utilization: 34% Shared: 28.9%, Global: 51.9%
Leukocyte	Version 1	IPC: 656, BW Utilization: 8% Const: 54.1%, Tex: 22.7%, Global: 7.7%
	Version 2	IPC: 707, BW utilization: 3% Const: 65.1%, Tex: 34.7%, Global: 0.0%

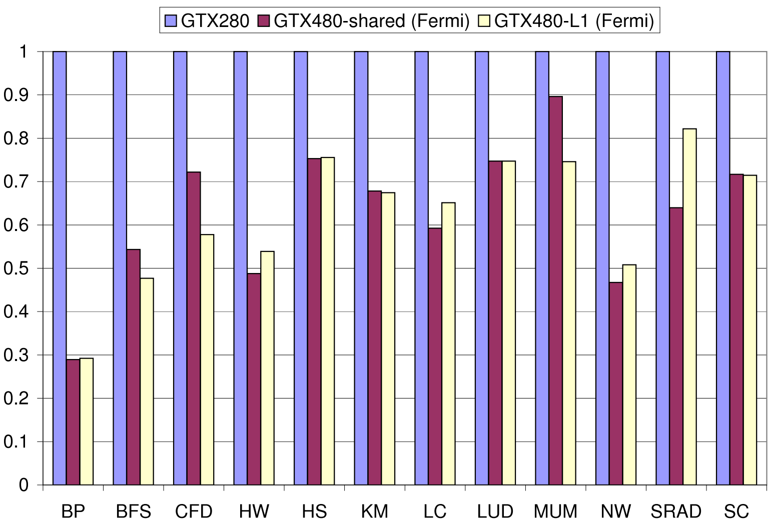

Figure 3.4: Normalized kernel execution time of the GPU implementations on a GTX 280 and a GTX 480 (Fermi) [17]. Two configurations (L1 and shared preferred) are used for the GTX 480 measurements.

3.1.4 Fermi Evaluation

In contrast to the earlier G80 and Tesla products, NVIDIA's Fermi includes traditional L1 and L2 caches. Each SM has 64 kB of on-chip memory that can be configured as 48 kB of shared memory and 16 kB of L1 (*shared bias*), the default configuration) or as 16KB of shared memory and 48 kB of L1 (*L1 bias*. CUDA provides a new API function, `cudaFuncSetCacheConfig()`, to select the desired configuration [27]. A unified L2 cache handles all memory requests for data loads and stores, as well as all texture fetches.

Figure 3.4 shows the results obtained measuring the performance of the Rodinia CUDA implementations on an NVIDIA GeForce GTX480 GPU with each memory configuration. We compare to the results on a GTX280 GPU with 240 1.3 GHz SPs and 1 GB of device memory. All the measurements are kernel execution times normalized to the GTX280. Except for applications such as *LUD* and *Leukocyte*, the total workload size is larger than the aggregate L1 capacity. The performances of *MUMmer* and *BFS*, which have large numbers of global memory accesses, improve by 11.6% and 16.7% respectively after switching the configuration from shared bias to L1 bias. Many

Rodinia applications, including *SRAD*, *Needleman-Wunsch* and *Leukocyte*, which are designed to utilize shared memory well, expectedly prefer the shared bias setting. Also, some applications (*LU Decomposition* and *StreamCluster*) show very little performance variation between the two configurations.

3.2 Rodinia and Parsec

In this section, we perform a comparison between Parsec and Rodinia benchmarks and evaluate their workload coverage. We hope that this comparison may facilitate the improvement of workload construction for multicore CPU and accelerator performance analysis.

How to perform fair comparisons between accelerator and CPU workloads running on different architectures is an open research question, and one which we cannot adequately address in this dissertation. Among the difficulties in heterogeneous, parallel benchmarking are the questions of 1. algorithm choice: How alike are the underlying algorithms of two different implementations? 2. optimization: What does it mean to compare the quantity and quality of optimization across heterogeneous platforms? 3. effort: If performance is not the sole concern, the next item on the list is probably cost or programmer effort. How difficult is an application to implement [14–16]?

We use the Rodinia OpenMP implementations to compare with the Parsec benchmarks in this study. We apply principal component analysis (PCA) to identify distinctions and also to characterize the workloads in terms of cache behavior, working set, and other metrics.

3.2.1 Comparison of Rodinia and Parsec

Parsec, a benchmark suite jointly developed by Princeton University and Intel, has been gradually gaining popularity among users of multithreaded workloads. The suite includes some workloads from emerging application domains and uses some state-of-the-art software techniques. Bienia *et al.* [8] compare SPLASH-2 and Parsec to determine the extent of feature overlap, and conclude that the workloads have significant differences. Many Parsec workloads have larger working sets than those in SPLASH-2, useful in the face of the scientific trend toward massive data growth. Other

Table 3.4: Comparison between Parsec and Rodinia.

Features↓ \ Suite→	**Parsec**	**Rodinia**
Platform	CPU	CPU and GPU
Programming Model	Pthreads, OpenMP, and TBB	OpenMP and CUDA
Machine Model	Shared Memory	Shared Memory and Offloading
Application Domains	Scientific, Engineering, Finance, Multimedia	Scientific, Engineering, Data Mining
Application Count	3 Kernels and 9 Applications	6 Kernels and 6 Applications
Optimized for…	Multicore	Manycore and Accelerator
Incremental Versions	No	Yes
Memory Space	HW Cache	HW and SW Caches
Problem Sizes	Small–Large	Small–Large
Special SW Techniques	SW Pipelining	Ghost-zone and Persistent Thread Blocks
Synchronization	Barriers, Locks, and Conditions	Barriers

Table 3.5: Parsec applications and sim-large input sizes. [7, 8]

Application	**Application Domain**	**Problem Size**	**Description**
Blackscholes	Financial Analysis, Algebra	65,536 options	Portfolio price calculation
Bodytrack	Computer Vision	4 frames, 4,000 particles	Computer vision, tracks 3D pose of human body
Canneal	Engineering	400,000 elements	Synthetic chip design, routing
Dedup	Enterprise Storage	184 MB	Pipelined compression kernel
Facesim	Animation	1 frame, 372,126 tetrahedrons	Physics simulation, models a human face
Ferret	Similarity Search	256 queries, 34,973 images	Pipelined audio, image and video searches
Fluidanimate	Animation	5 frames, 300,000 particles	Physics simulation, animation of fluids
Freqmine	Data Mining	990,000 transactions	Data mining application
StreamCluster	Data Mining	16,384 points per block, 1 block	Kernel to solve the online clustering problem
Swaptions	Financial Analysis	64 swaptions, 20,000 simulations	Portfolio price calculations with Monte-Carlo
Vips	Media Processing	1 image, 26,625,500 pixels	Image processing, image transformations
X264	Media Processing	128 frames, 640,360 pixels	H.264 video encoder

work compares the communication characteristics of SPLASH-2 and Parsec [81] and examines the behavior of Parsec on real hardware [7].

Table 3.4 provides a high-level overview of the differing design focuses of Parsec and Rodinia, while Table 3.5 provides some more specific details on Parsec. In the previous sections, we discussed several aspects of Rodinia which distinguish it from other benchmark suites; here we provide some more discussion on the topic, specifically with respect to Parsec.

Parsec provides a rich set of features that support fine-grained parallelism (locks), languages (TBB, OpenMP, and Pthreads), and large code bases. Rodinia currently focuses only on OpenMP workloads for the CPU implementations. The use of fine-grained parallelism in Rodinia, even in CPU implementations, is restricted by our desire to maintain algorithmic congruence with the CUDA ports given the fact that CUDA supports only barrier synchronization within a thread block [53] and global synchronization at kernel exit or when using a global synchronization primitive. In the construction of the Rodinia benchmark suite, we also consider Parsec workloads. We include *StreamCluster* in Rodinia, but find that those benchmarks relying on task pipelining, like *Ferret*, do not port well unless each stage is also heavily parallelizable.

3.2.2 Methodology

To compare Rodinia and Parsec, we adopt the methodology and metrics of Bienia *et al.* [8] in their SPLASH-2 and Parsec comparison, so that the reported results are cross-comparable. The points of comparison include instruction mix (including ALU, branch, and memory instructions), working set (cache misses per memory reference), and sharing behavior (the fraction of cache lines shared, and the number of accesses to shared lines per memory reference). Our experiments use eight cache sizes, ranging from 128 kB to 16 MB, and measure the sharing and the working set behavior. We adopt a similar cache structure to that use by Bienia *et al.* as well, an 8-core processor with a single cache shared by all cores. The cache is 4-way associative with 64 byte lines. All programs are compiled with gcc 4.2.1 with OpenMP or Pthreads.

All data is obtained with Pin [55]. Pin is a dynamic, binary instrumentation tool and provides an infrastructure for writing program analysis tools, called *Pin tools*. Instruction mix is obtained using the *mix-mt* tool provided with the Pin package. A Pin tool, based on the cache tool in Pin, is developed to collect cache behavior characteristics.

3.2.3 Principal Component Analysis and Measuring Similarity

Principal components analysis (PCA) is a statistical, data analysis technique that reduces a data set's dimensionality and removes correlation from the data set while controlling the amount of information lost. PCA computes n new variables, called principal components, which are linear combinations of n original variables, such that all principal components are uncorrelated. The first of the resulting orthogonal principal components exhibits the largest variance, followed by the second, followed by the third, and so on [36]. After performing PCA, we cluster to find equivalence classes of programs with similar characteristics. PCA has been widely applied for benchmark comparison [8, 36, 42, 66] in similar contexts. However, the issue of how to perform more fair and accurate evaluation and comparison of benchmarks is an open one and beyond the scope of this dissertation.

To measure the similarity among benchmarks, we use classical hierarchical clustering analysis. Similar approaches have been used in other recent performance analysis work. Clusters are formed in such a way that data objects in the same cluster are very similar and data objects in different clusters are very distinct. We use the MATLAB [79] statistics tool box to process the data for the collected characteristic values for all the benchmarks. The algorithm involves finding the similarity or dissimilarity between every pair of data objects in the data set using distance functions and grouping the objects into a binary, hierarchical cluster tree. Dendrograms are used for result illustration.

3.3 Analysis

Here we present the results of our principal component analysis.

3.3.1 Hierarchical Clustering

Figure 3.5 shows the overlap of the two program collections. In the figure, the magnitude of the link between any two nodes (or clusters of nodes) qualtifies the measure of dissimilarity between those nodes; thus, *Leukocyte* and *Bodytrack* are fairly similar, while *Heartwall* differs significantly from all other compared benchmarks; and *MUMmer* and *Swaptions*, while spatially close in the figure, are more dissimilar than *HotSpot* and *Facesim*. From this dendrogram, it is evident that the two benchmark suites cover similar application spaces, with most clusters containing both Rodinia and Parsec applications. Also note that the new applications added to Rodinia, namely *CFD*, *LUD*, *MUMmer*, and *Heartwall*, enrich the original application set, with the latter two significantly different from others.

We also perform an analysis of some subsets of our characteristics. Figures 3.6, 3.7 and 3.8 show the instruction mix, working set and sharing behaviors of the programs as is similarly shown in the Parsec and SPASH-2 comparison [8]. In Figure 3.6, Parsec and Rodinia demonstrate disparate

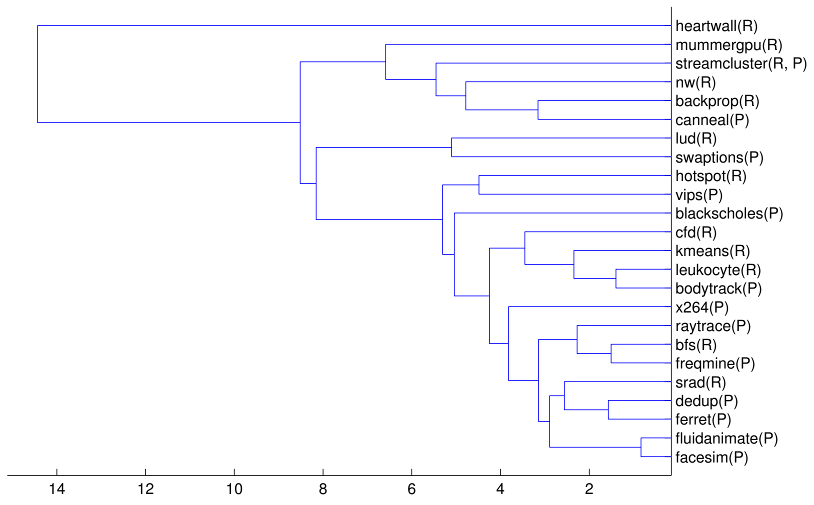

Figure 3.5: A dendrogram showing the similarity between the Parsec (*P*) and Rodinia (*R*) workloads [17]. The *x* axis represents the linkage distance in a PCA space, which has no obvious physical analog.

behavior, with *Breadth-First Search*, *Back Propagation*, and *HotSpot* from Rodinia, and *Raytrace*, *Ferret*, *Bodytrack*, and *StreamCluster* from Parsec tending to populate different areas in the space. In the working set plot of Figure 3.7, there are several Parsec and Rodinia benchmarks that are clear outliers from the main cluster; *MUMmer* is a significant outlier, which correlates with its high miss rates. The miss rates (i.e. cache misses per memory reference) of all the benchmarks under a 4 Mb cache configuration are shown in Figure 3.9. Figure 3.8 shows similar behavior—data sharing, now, rather than working set size—with *Heartwall* significantly different from the rest. Looking back at Figure 3.5, *Heartwall* and *MUMmer* are the most disparate benchmarks in the suite; something which is backed up by this series of figures.

As shown in these figures, for the characteristics we evaluated, Rodinia also provides a good workload mix for multicore CPUs. Additionally, Parsec and Rodinia demonstrate features that complement with each other, suggesting that researchers should consider both of them, possibly as well as other benchmark suites, to ensure a reasonable application coverage for their work.

With the help of the clustering tree, users can choose appropriate benchmarks to meet their needs – selecting the *N* most diverse benchmarks for any *N*. One benchmark can be chosen from each cluster by tracking down the clustering tree. If a cluster consists more than two benchmarks, the benchmark closest to the center of the cluster is chosen as a representative [66].

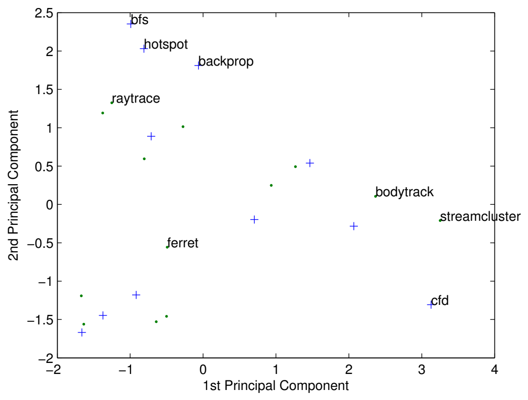

Figure 3.6: The instruction mix plot with two PCA components for Parsec (·) and Rodinia (+) [17]

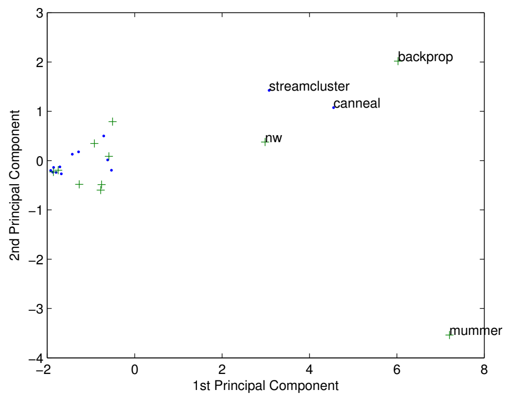

Figure 3.7: The working set plot with two PCA components for Parsec (·) and Rodinia (+) [17]

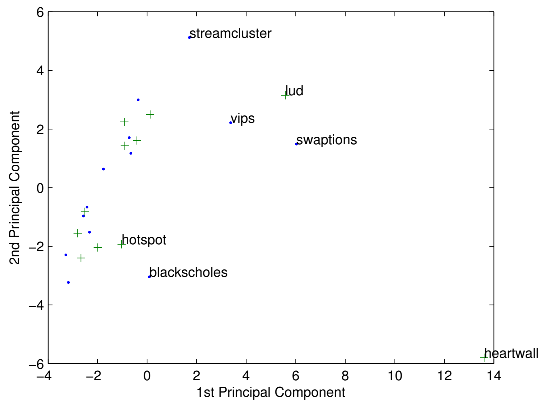

Figure 3.8: The sharing plot with two PCA components for Parsec (·) and Rodinia (+) [17]

3.3.2 Clustering Discussion

How well the application space is covered by the two suites?

— Our clustering analysis shows that Parsec and Rodinia cover similar application spaces. This does not imply that using either or both of them is sufficient for research. Consider the blank regions in the PCA spaces of Figures 3.6, 3.7, and 3.8; it is unclear whether these regions can be covered by

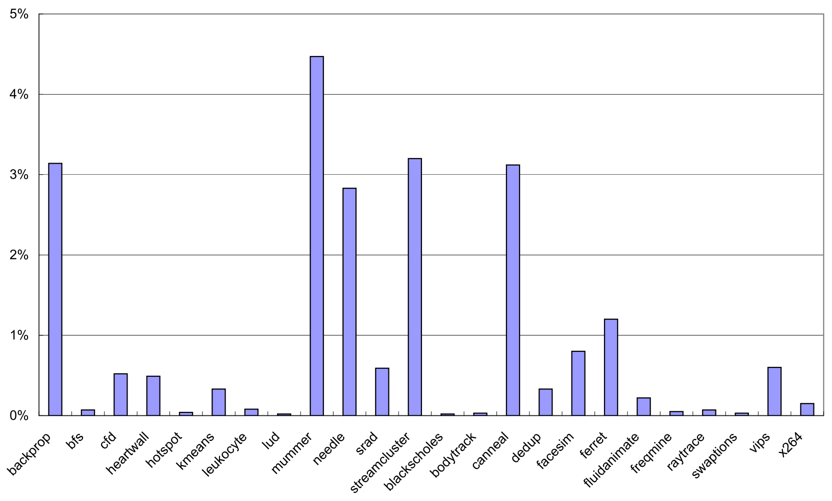

Figure 3.9: The miss rates of Rodinia and Parsec [8] benchmarks under a 4 Mb cache configuration [17].

other real-world workloads or benchmark suites. This implies that a thorough examination requires a comprehensive evaluation and comparison of all the current multithreaded benchmark suites, including SPLASH-2 and various domain-specific workloads, to establish a single set of workloads with sufficient coverage and little redundancy. Previous work performs such studies but only on single-threaded benchmarks [42,66], while this is an open problem in heterogeneous environments. In addition, how to determine whether the benchmarks in a benchmark suite are sufficiently diverse remains an open research question.

The metrics evaluated in this work are important for multithreaded program behavior [8]. On the other hand, other potentially important metrics may indicate other crucial differences between benchmarks. It is an area of ongoing research to develop a set of metrics which are able to capture most behaviors of multithreaded workloads. Host *et al.* [36] propose a set of microarchitecture-independent workload characteristics to profile single-threaded applications, and which are also useful for performance prediction [38]. A set of metrics for multithreaded workloads are needed.

Can the Parsec workloads be effectively mapped to heterogeneous platforms?

— Our clustering results indicate that, except for a few outliers, the heterogeneous workloads we developed for Rodinia are not fundamentally different from those of Parsec, developed for multicore CPUs. This, however, does not imply that since all the Rodinia benchmarks map well to the GPU platform, the same will be true of the Parsec benchmarks. We have found many challenges in the task of porting traditional, multithreaded, CPU workloads onto heterogeneous platforms. Some issues which make this less than straightforward:

- **Library Modules**: Application development depends upon libraries and reuse for productivity and maintainability. This poses a potentially large challenge in porting CPU applications to accelerators. Though it is possible to implement each library module on the GPU, for example, the cost of maintaining modularity is the possibly resultant overhead of GPU kernel call invocation and memory transfer between the CPU and the GPU. To achieve better performance for GPU applications, optimization sometimes requires cross-function algorithmic reorganization, or the division of a logical function into multiple kernels.

- **Synchronization**: Many Parsec applications heavily rely on fine-grained synchronization primitives, such as mutexes [9]. For some applications, like *StreamCluster*, it is relatively easy to reorganize for the GPU, while for others, it is non-trivial; especially in those applications using the software pipelining model (including *Dedup* and *Ferret*), which require significant algorithmic reorganization. The difficulty in supporting these primitives is directly attributable to the GPU's limited synchronization capabilities. On the GPU, synchronization within a thread block is provided, and global synchronization is achieved via a barrier. The latest CUDA versions also provide a primitive for an on-chip, global memory fence which, unfortunately, requires restructuring of applications such that thread blocks are persistent during the entire program execution. Locks across thread blocks are non-trivial to implement, and performance benefits are not guaranteed.

Are existing application classification taxonomies sufficient to differentiate application characteristics?

Several approaches have been proposed for classifying applications based on their memory access and execution patterns, including the Berkeley *Dwarves* [2] and Intel's *Recognition, Mining and Synthesis* (RMS) [51]. Rodinia and Parsec were designed with the Dwarves and RMS as guidelines, respectively. Although these taxonomies are defined at a high levels of abstraction to provide useful guiding principals and to allow users to effectively reason about program behavior, our work, often with multiple instances of a single Dwarf, suggests that the Dwarf taxonomy alone may not be sufficient to ensure adequate diversity, and that some important behaviors may not be captured by the Dwarves.

As shown in Figure 3.5, for *Structured Grid* applications, stencil-type workloads, such as *SRAD* and *Fluidanimate*, are quite similar. However, applications such as *HotSpot*, *Leukocyte*, and *Heartwall* are located in different clusters, with *Heartwall* significantly different from the others. *Back Propagation* and *CFD* are both from the *Unstructured Grid* Dwarf and show significant differences. The *Graph Traversal* applications, *MUMmer* and *Breadth-First Search*, are also very dissimilar.

Even applications from the same application domain are quite different; for example, the two fluid dynamics applications, Parsec's *Fluidanimate* and Rodinia's *CFD* differ more than *Fluidanimate* and *Facesim*, the latter members of different Dwarves. Also, two data mining benchmarks, *Kmeans* and *StreamCluster*, both of which rely on distance-based clustering, lie far apart in the binary clustering tree.

3.3.3 Instruction and Data Footprints

Figures 3.10 and 3.11 illustrate the instruction and data footprints of Parsec and Rodinia. The figures show the number of 64-byte instruction blocks and 4 kB data blocks touched during the entire program execution [42]. Figure 3.11 shows that both Parsec and Rodinia use large working sets, but, with the exception of *MUMmer*, Parsec applications tend to have larger instruction footprints, or code sizes, than Rodinia workloads.

There is a related, open question in workload characterization, that of the difference between "big" applications and "small" ones, or, in other words, between applications and kernels. Better understanding this issue requires finding the "building blocks" of the applications and a method to correlate applications with with constituent kernels. Carrington *et al.* [12] did this in the HPC domain, but some more sophisticated approaches are needed for higher prediction accuracy.

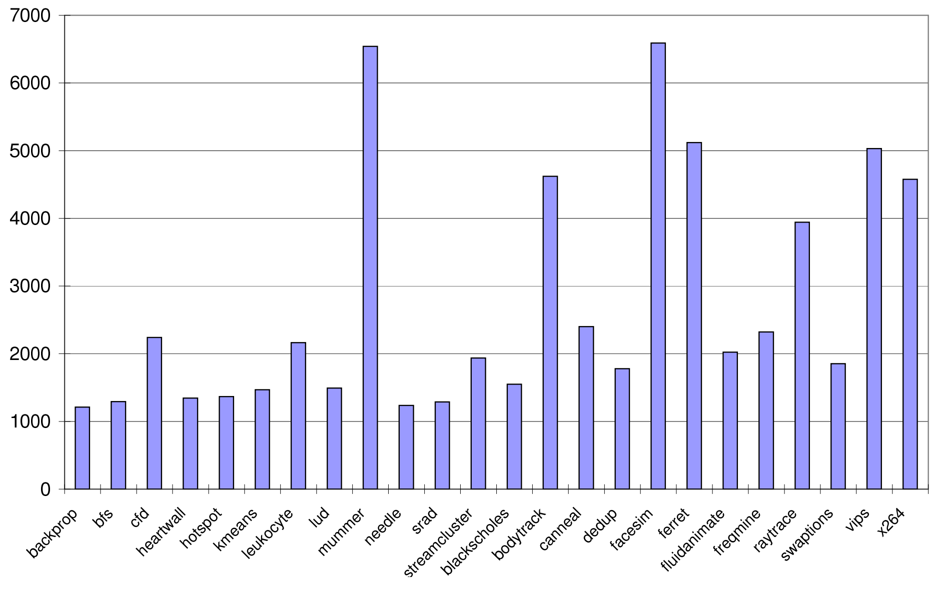

Figure 3.10: The numbers of 64-byte instruction blocks touched during the program execution [17].

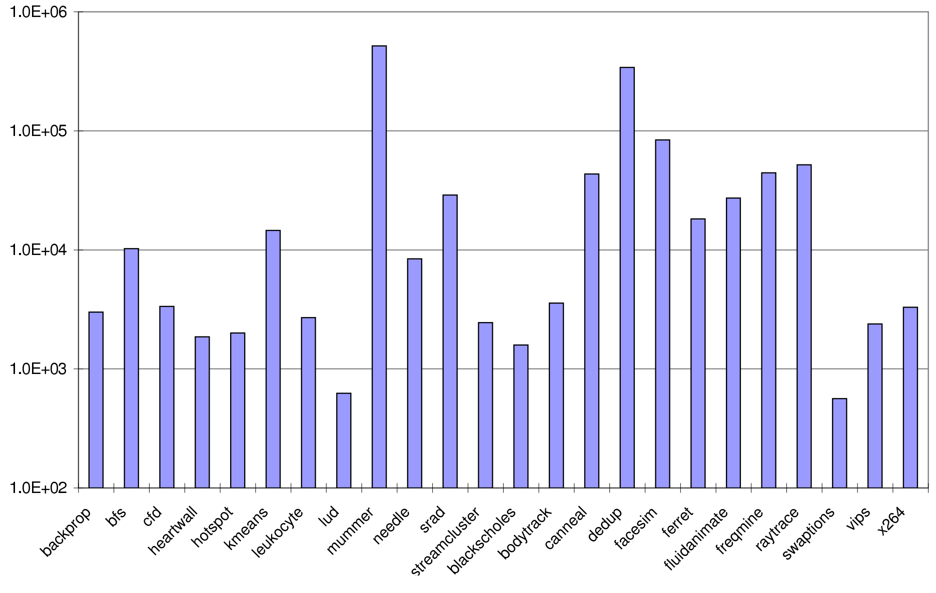

Figure 3.11: The number of 4 kB data blocks touched during the program execution [17].

3.4 Other Benchmark Suites

The Parsec benchmark suite [9] includes emerging applications from finance, multimedia, and data mining. Parsec benchmarks utilize relatively large working sets and are developed with state-of-art software techniques such as software pipelining. Some earlier benchmark suites include SPLASH-2 [82] and SPEC OMP2001 [71], consisting of general-purpose workloads focusing on science, engineering, and graphics. BioParallel [40], ALPBench [50], and MineBench [67] target specific application domains. Parboil [75] and SHOC [20] are two efforts to benchmark GPUs, but the former does not provide any diversity analysis and the latter targets systems with multiple GPU nodes. Bakhoda *et al.* developed GPGPU-Sim [4] with a set of GPU workloads to analyze various CUDA programs. Rodinia is distinct from these works primarily in that it is designed to provide implementations with diverse parallel execution patterns, optimizations, and software mappings, in addition to its ability to compare platforms, a crucial capability for tackling the design challenges of future parallel and heterogeneous systems.

3.5 Conclusions and Future Work

In this chapter, we performed a detailed characterization of Rodinia, designed to let researchers better understand this collection of benchmarks. Our experimental results show that Rodinia applications demonstrate a good mixture of diversity in application characteristics. We evaluate Rodinia on an NVIDIA's GTX 480 GPU with different L1 cache/shared memory combinations.

We also compared Rodinia with Parsec, and some important differences we observe show the importance of measuring how well existing suites span the design space and the importance of using applications from different suites together.

Directions for future work include performing an application-space coverage study of existing multithreaded workloads, and correlating program characteristics across the CPU and the GPU as well as across big applications and small kernels.

Chapter 4

Load Balancing

One trend of computer architecture development is integrating both CPUs and various accelerators to explore heterogeneity. Some representative examples of these platforms include systems with multicore CPUs + discrete GPUs (e.g. from NVIDIA and AMD), STI Cell processors [43], and AMD Fusion APUs [1] that integrate both the CPU and the GPU on a single die. On the other hand, these heterogeneous systems can be enterprise servers or machine clusters consisting of various processors, probably with diverse compute capabilities or from different hardware generations. Recently, there has been growing research to explore how to efficiently utilize the available compute resources in a heterogeneous system. Important research issues include mapping applications to their most appropriate devices and determining desirable balanced workload ratios.

This chapter studies an approach to execute parallel compute kernels simultaneously on the CPU and the GPU and aims to achieve efficient work spreading and load balancing across platforms. There are several prior research works trying to tackle a similar issue. As discussed in Section1.3, Intel's Merge framework [52] relies on programmers to specify problem mappings manually, which is not adaptable to changes of underlying hardware and problem size [56]. Qilin [56] proposes an adaptive mapping and uses training runs to decide the amount of work distributed to the CPU and the GPU. However, it will not perform well when the available resource for an application changes at runtime. Hong et al. [34] propose MapCG, a MapReduce framework to support source-code level portability between the CPU and the GPU. Twin Peaks [28] makes the applications originally developed for the GPU run efficiently on the CPU. However, these two studies target single devices without any support for workload partitioning across different processing units.

To address these concerns, we explore an approach of hierarchical domain partitioning by dividing the entire computation domain into smaller chunks and efficiently scheduling them across platforms in a on-demand way. We use task-queues to keep track of the progress of chunks.

This chapter makes the following contributions:

- We discuss our design considerations, advantages of our load balancing model and its implementation details.
- We demonstrate performance benefits of CPU-GPU load balancing by presenting four diverse applications as case studies.
- We further discuss the impact of choosing different chunk sizes and optimization techniques that improve performance.

4.1 Work Spreading and Balancing Framework

4.1.1 Load Balancing on Heterogeneous Platforms

To distribute workloads across different devices is a challenge. It is further complicated by the vastly different architectural designs of the CPU and the GPU. In our approach, we treat different compute devices as having different consumption rates, C, of computation. For instance, the rate for the CPU is C_{cpu} while the rate for the GPU is C_{gpu}. We define W as the total amount of computation. We also define p as the percentage of computation distributed on the CPU and $(1-p)$ is the percentage of computation distributed on the GPU. Therefore, the execution time of a program on a 1 CPU + 1 GPU system with this simple model will be:

$$T_{cpu} = \frac{W}{C_{cpu}} * p \tag{4.1}$$

$$T_{gpu} = \frac{W}{C_{gpu}} * (1 - p) \tag{4.2}$$

$$T = MAX(T_{cpu}, T_{gpu}) \tag{4.3}$$

Then, the goal of solving the problem becomes minimizing the overall execution time T. A similar analysis can also be found in the Qilin work [56]. The minimum T will be achieved at the cross point of the curve T_{cpu} and T_{gpu}, and thus theoretically the balanced ratio of the CPU to the GPU is determined by the following equation.

$$W_{cpu} : W_{gpu} = C_{cpu} : C_{gpu} \tag{4.4}$$

Hong et al. [34] argue that simultaneous CPU-GPU execution might not be beneficial when the GPU is significantly faster than the CPU or vice versa. However, Lee et al. [48] in their ISCA 2010 paper shows that when some applications are heavily optimized for both the CPU and the GPU, the CPU can achieve a similar or even exceed the performance of the GPU.

Our solution aims to determine workload distributions dynamically and respond to run-time resource changes. We focus on the parallel phases in an application and the following items summarize our major design:

- We implement a task-queue based scheduler which dispatches parallel works simultaneously onto the CPU and the GPU in an on-demand way.
- We divide the computation domain into data chunks, allowing the scheduler to detect computation rates of different devices at runtime and reach a desirable balanced workload ratio.
- The division of computation domain is similar to OpenCL/CUDA's hierarchical domain partitioning, suggesting that existing applications can map to our load balancing model in a straightforward manner.

4.1.2 A Hierarchical Partitioned Domain

Both OpenCL [64] and NVIDIA's CUDA [27] adopt a domain-based hierarchical approach to abstract a computation problem; the domain is conceptually described by 2D/3D matrices (i.e. grids and blocks) with each data element indexable with thread ids. We use a similar approach to partition the computation domain by introducing another layer of abstraction – *chunk* – which is like a grid in CUDA and OpenCL. Therefore, the domain contains multiple chunks, i.e. grids of blocks. Figure 4.1 shows a simple example in which a 2-D matrix is divided into 3×3 grids. Each grid is further divided into 4×4 blocks and each block consists of 8×8 work items. In addition, chunks can be multidimensional and their partitioning can be flexibly specified by programmers. The information of partitioning will be passed into the scheduler when users launch a compute kernel. This can be done in a similar way to CUDA and OpenCL (e.g. $<<< ChunkDim, GridDim, BlockDim >>>$).

Each chunk is the basic unit for scheduling and processed by one kernel call dispatched either on the CPU or the GPU. New chunks will be allocated to CPU or GPU, whenever any compute resource is available for computation. Also, the domain-based model is conceptually consistent with CUDA and OpenCL, and the extra effort spent on problem mapping is trivial, though programmers are still required to develop a compute kernel as usual.

The CUDASA programming environment [74] proposed a similar approach by partitioning the computation domain into four layers. Their work mainly uses it for purpose of mapping a problem for multi-GPU systems, and a single GPU device is usually responsible for executing a single CUDA grid. This work observes that such partitioning can also be used for CPU-GPU load balancing.

4.1.3 Chunk Scheduling

Our scheduler handles work distributions to the CPU and the GPU. When a program starts computation, it will invoke the scheduler which distributes chunks based on the partitioning information provided by programmers. The scheduler can spawn CPU worker threads and schedule GPU works by filling the task queues (e.g. OpenCL command queues). During program execution, the sched-

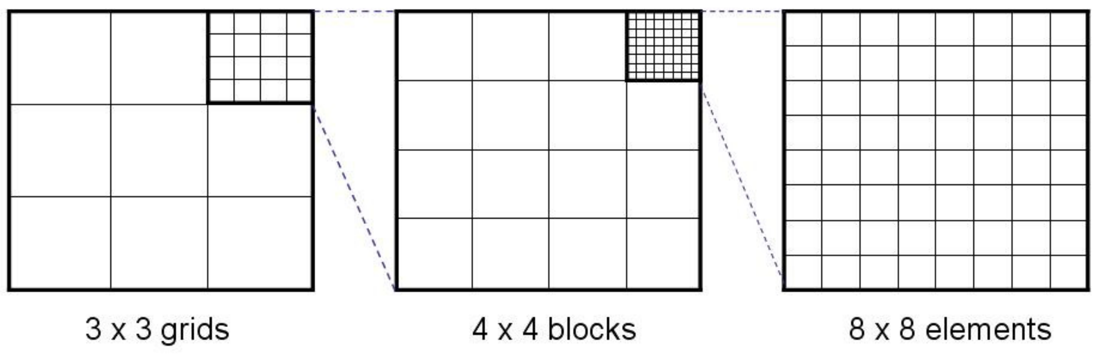

Figure 4.1: In this example, the whole application domain is partitioned into 3×3 grids (i.e. chunks). And each grid is further divided into 4×4 blocks and each block has 8×8 work items. Each grid, or chunk is the basic unit for scheduling.

uler checks the status of worker threads and in turn fill new jobs and remove the completed jobs. To implement the CPU threads, we use *pthreads* for the Linux environment.

Figure 4.2 shows a high-level view illustrating the concept of the scheduling framework. The master thread launches one worker thread for each CPU core. Each thread maintains a task queue of uncompleted jobs. The master thread distributes chunks to CPU threads by pushing chunk ids onto CPU queues. Each CPU thread pops a chunk id from the front of a task queue and subsequently launch computation for that particular chunk. Distribution of chunks to the GPU is implemented using asynchronous kernel and memory transfer calls offered by OpenCL/CUDA to minimize scheduling overhead. Determining if a chunk finishes processing on the GPU is achieved by polling the GPU command queue. The *event* is used to track the status of a kernel call. We use OpenCL *events* to ensure producer-consumer relationships among OpenCL calls, e.g. between compute kernels and memory transfers.

One potential advantage of our approach is that if an application shares compute resources with other applications in a heterogeneous system, our approach can automatically adjust the load balancing ratio based on the feedback of how fast each device processes their chunks. Furthermore, we also provide parameters (e.g. *gpu_fillsize* and *cpu_fillsize*) to interact with the scheduler, allowing manual changes of load balancing ratios for the CPU and the GPU. Currently the scheduling policy is merely based on processing speeds of different devices, i.e. the emptiness of the task queues.

Other scheduling techniques optimized for other metrics (e.g. power and energy efficiency) can be easily integrated in future.

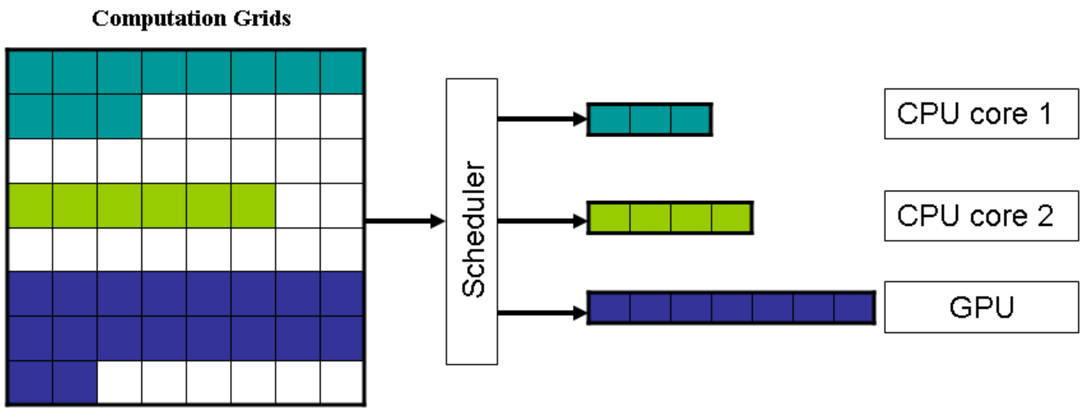

Figure 4.2: Depending computation rates of different devices, the scheduler dispatches data chunks dynamically on each device

4.1.4 Data Affinity

Systems with CPUs and discrete GPUs require data to be copied from the system memory to the GPU device memory for subsequent GPU kernel execution. Even in today's AMD Fusion APU [1] implementation in which the CPU and the GPU share the same physical memory, memory transfers are required between the CPU to the GPU memory space. For CPU-GPU load balancing, ideally only necessary data will be copied across the CPU and the GPU. Therefore, the data chunks with close data affinity and dependency relationship are desirable to schedule together on the same device. For example, for many stencil applications (e.g. *SRAD*), the calculation of a data element depends on its neighboring elements; this implies that the calculation of a chunk also depends on its neighboring chunks. Therefore, efficient domain partitioning for SRAD across the CPU and the GPU requires that we dispatch CPU chunks from one end of the domain and GPU chunks from the other end, until the two ends consume all the data and meet at the boundary. In this case, only data chunks located at the boundary need to be exchanged between two devices when data communications are needed. Otherwise, unnecessary data communications will occur with randomly

Table 4.1: Load Balancing Applications.

Application	Dwarf	Domain	Input size
SRAD	Structured Grid	Image Processing	4096×4096 matrix
K-means	Dense Linear Algebra	Data Mining	1310720 data points and 16 features
Sepia	Structured Grid	Image Processing	1024×1024 pixels
Needleman-Wunsch	Dynamic Programming	Bioinformatics	4096×4096 score matrix

scheduled chunks.

4.2 Methodology

4.2.1 Experiment Setup

Our measurement results are obtained on real hardware. We compare the execution times by running applications on the CPU and the GPU individually and simultaneously. The experiments use an NVIDIA GeForce GTX 460 with 1.35 GHz shader clock, 64 kB configurable on-chip cache per streaming multiprocessr (SM), 768 kB shared L2 cache and 1 GB device memory. The GTX 460 is designed with the Fermi architecture and allows simultaneous kernel execution. Each SM has 16 streaming processors (SPs) for a total of 480 SPs. The 64 kB cache per-SM can be set into two configurations —48 kB shared memory + 16 kB L1 or 16 kB shared memory + 48 kB L1. The CPU is a 2.66 GHz Intel Core2 Quad CPU with 3 MB L2 cache.

4.2.2 Applications

We choose several applications with diverse characteristics. Table 4.1 lists the basic information for each application. *SRAD* is a structured grid application processing 2-D array structures. The result of each element is calculated with its four adjacent neighbors. *Kmeans* presents parallelism across different rows of data elements. *Sepia* is a widely-used kernel to modify RGB value to produce artificially aged images. We choose this compute kernel (used in Qilin [56]) because of its limited performance benefit from GPU acceleration. *Needleman Wunsch* is a dynamic programming algorithm which presents limited data parallelism within each diagonal strip.

4.3 Application and Experiment Results

4.3.1 Chunking Overhead

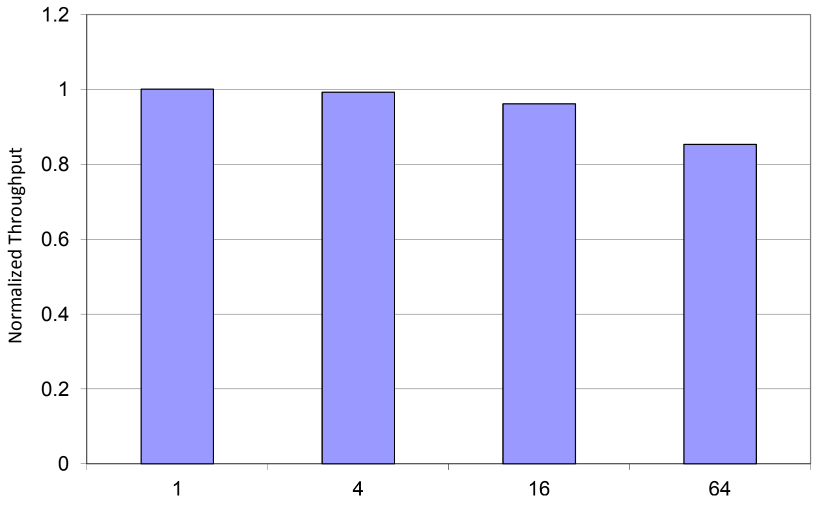

Figure 4.3: Y-axis represents the normalized throughput for CPU-GPU transfer. The baseline is to transfer the entire 64 MB data in one chunk between the CPU and the GPU. We compare it with the same amount of data transfers with multiple chunks.

In our framework, computation is partitioned into chunks to process, which means PCI-E transfers of data structures also need to be accomplished by multiple individual chunk transfers. We measure the overhead caused by such divisions. For this experiment, we developed a microbenchmark to transfer a contiguous memory region with a size of 64 MB from the CPU to the GPU. We compare the execution time of using only one OCL *clEnqueueWriteBuffer* call and that of using multiple calls to transfer the same amount of 64 MB data (See Figure 4.3). Our experimental result shows that using more chunks will incur more overhead. Part of the overhead is due to the extra time spent on launching OCL clEnqueueWriteBuffer calls. The overhead becomes significant when the number of chunks exceeds 64.

4.3.2 Kmeans

Kmeans is a clustering algorithm used extensively in data mining. In *Kmeans*, a data object is comprised of several parameters, called *features*. By dividing a set of data objects into k clusters, *Kmeans* represents all the data objects by the mean values or *centroids* of their respective clusters. In a single-device Rodinia GPU implementation, data objects are grouped into thread blocks, with each thread associated to one data object. The task of searching the nearest centroid to each data object is completely independent and can be done in parallel. To extend the *k-means* GPU implementation to a chunking implementation, its main data structure saving all the data objects is partitioned into multiple chunks and further into blocks. Because data parallelism exists across rows, 1-D chunk dimension is used $(chunkDim.x, chunkDim.y) = (\frac{NumOfElements}{ChunkSize}, 1)$. Whenever a chunk is scheduled to run on the GPU, its data objects are first transfered from the CPU to the GPU. The membership array segment for this chunk, which stores the closest centroid to each data object, will be transfered back to the CPU for new centroid calculation.

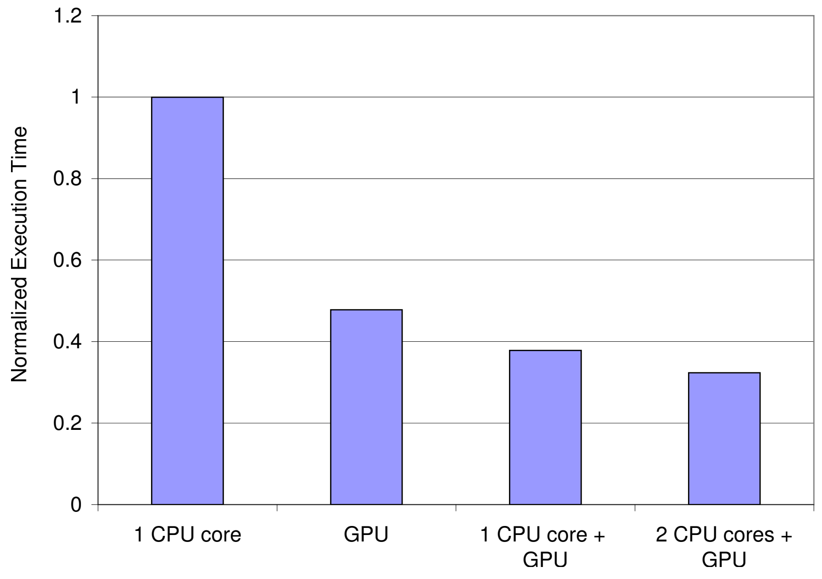

Figure 4.4: Y-axis represents the normalized execution time. The execution times are obtained with running the *Kmeans* distance kernel on the CPU, the GPU, and the CPU and the GPU simultaneously. The chunk size is 64 k of data elements.

Figure 4.4 illustrates the results for the *Kmeans* distance kernel. In this experiment, we use an input size of 1.25 M (i.e. data points) and a chunk size of 65536. The execution time is normalized

Table 4.2: Load Balancing Ratio of *Kmeans*

Core Combination	**CPU**	**GPU**
one CPU core and GPU	35%	65%
two CPU cores and GPU	50%	50%

to that of running *Kmeans* on one CPU core. Spreading the works of *Kmeans* on one CPU core and the GPU improves the performance by 20.8% against the GPU-only execution. Using two CPU cores and the GPU improve the performance further by 14.4%. As shown in Table 5.3, for the 1 CPU core + GPU configuration, the portions of the workloads mapped to the CPU and GPU are 35% and 65%, respectively. Switching to the 2 CPU cores + GPU configuration, the portions of the workloads mapped to the CPU and GPU change to 50% and 50%.

4.3.3 SRAD

SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations. It is used to remove locally correlated noise, known as speckles, without destroying important image features [83].

To adapt SRAD to our model, there are two potential ways to partition its computation domain. One way is to divide the main data structure into 2D chunks, and chunk dimensions can be denoted as $(ChunkDim.x, ChunkDim.y) = (\frac{DimX}{ChunkSize}, \frac{DimY}{ChunkSize})$ with each further divided into a number of thread blocks. This partitioning is similar to the example shown in Figure 4.1. On the other hand, we can divide the domain horizontally into strips, and chunk dimensions can be denoted as $(ChunkDim.x, ChunkDim.y) = (\frac{DimX}{ChunkSize}, 1)$. It turns out that the second way of partitioning is more efficient. The reason is that CUDA and OpenCL provide API functions to copy a linear region of memory with a start pointer and number of bytes to be transferred as inputs. For an arbitrary block in a 2-D chunk space, transferring data between the CPU and the GPU requires launching multiple *memcpy* calls with each call for one row. The overhead of API calls will be significant when data size is small. On the other hand, the second approach is more efficient because only one memory copy is needed to supply data for a specific chunk.

Figure 4.5 shows the results for the *SRAD* kernel with a 4096×4096 input matrix and a chunk size of 256 k. Spreading the chunks of *SRAD* on one CPU core and the GPU improves the per-

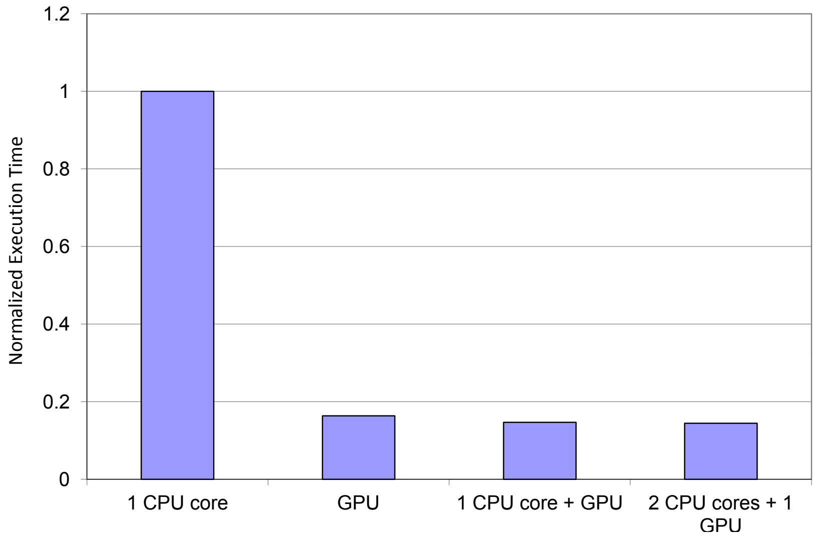

Figure 4.5: Y-axis represents the normalized execution time. The execution times are obtained with running *SRAD* on the CPU, the GPU, and the CPU and the GPU simultaneously.

Table 4.3: Load Balancing Ratio of *SRAD*

Core Combination	**CPU**	**GPU**
one CPU core and GPU	12.5%	87.5%
two CPU cores and GPU	26.6%	73.4%

formance by 10% against the GPU-only execution. Using two CPU cores and the GPU improve the performance further by 5%. As shown in Table 4.3, for the 1 CPU core + GPU configuration, the portions of workloads mapped to the CPU and the GPU are 12.5% and 87.5%, respectively. Switching to the 2 CPU cores + GPU configuration, the portions of the workloads mapped to the CPU and GPU change to 26.6% and 73.4%. Compared to *Kmeans*, *SRAD* obtains less performance benefit from CPU-GPU load balancing because the GPU execution time is much faster than the CPU execution time.

4.3.4 Sepia

Sepia is widely used in image processing to give photos with colors an old world effect (e.g. brown/grey, black/white). Each pixel in the image is represented with a 3-element vector for R, G and B values. Sepia is applied to each pixel independently with value averaging, scaling and adjustments. We choose this compute kernel (also used in Qilin [56]), because the performance benefit

Table 4.4: Load Balancing Ratio of *Sepia*

Core Combination	CPU	GPU
one CPU core and GPU	56%	44%
two CPU cores and GPU	62.5%	37.5%

of the GPU acceleration is very limited due to its memory-boundness and branch divergence [56].

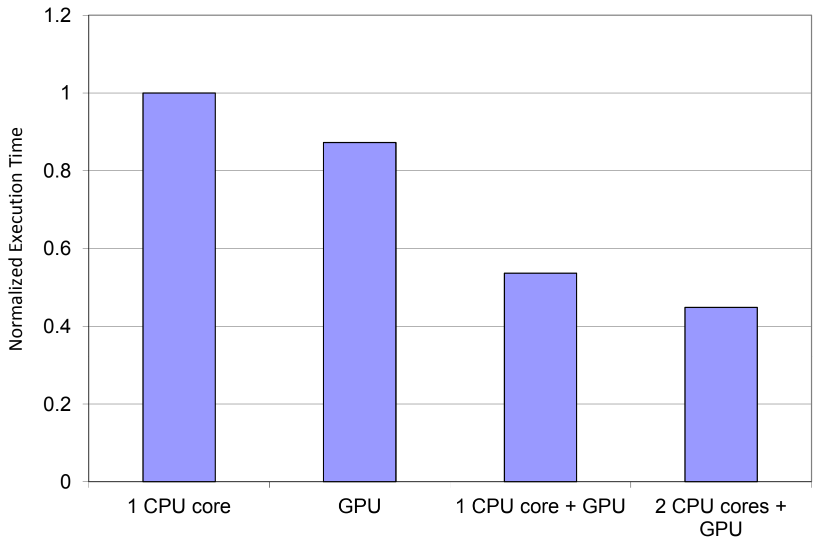

Figure 4.6: Y-axis represents the normalized execution time. The execution times are obtained with running *Sepia* on the CPU, the GPU, and the CPU and the GPU simultaneously.

Figure 4.6 shows the results of *Sepia* with an input size of 1024 $\times$ 1024 and a 65536 chunk size. Spreading Sepia on one CPU core and the GPU improves the performance by 38% against the GPU-only execution. Using two CPU cores and the GPU improve the performance further by 16%. As shown in Table 4.4, for the 1 CPU core + GPU configuration, the portions of the workloads mapped to the CPU and GPU are 56% and 44%, respectively. Switching to the 2 CPU cores + GPU configuration, the portions of the workloads mapped to the CPU and the GPU change to 62.5% and 37.5%.

4.3.5 Needleman Wunsch

Needleman-Wunsch is a global optimization method for DNA sequence alignment. Che et al. [15] optimized *Needleman Wunsch* by introducing a block-level parallelism. To make the algorithm run on the both the CPU and the GPU, we further introduce another chunk-level of parallelism.

All the chunks within a diagonal strip, each consisting of multiple blocks, can be distributed to different compute devices. Figure 4.7 shows the access pattern of *Needleman Wunsch*. In each iteration, we compute one diagonal strip of chunks. For this application, we use 2-D chunking: $(ChunkDim.x, ChunkDim.y) = (\frac{DimX}{ChunkSize}, \frac{DimY}{ChunkSize})$ (See Figure 4.7). The address of each data element can be calculated with the following equation.

```
data_element_address = cols * (chunk_size * dim.y)
+ chunk_size * dim.x + cols * block_size * blk.y
+ block_size * blk.x + cols * ty + tx;
```

When the scheduler allocates a chunk on the GPU, two strips of data elements, the north and west borders of a specific chunk will be transferred from the CPU to the GPU.

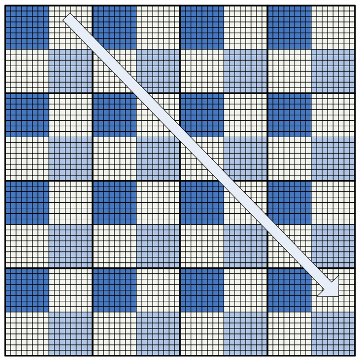

Figure 4.7: The domain partitioning of Needleman Wunsch. Parallelism exists in a diagonal strip of chunks.

We run the *Needleman Wunsch* implementation with a 4096×4096 input and a 1024×1024 chunk size. Load balancing for *Needleman Wunsch* on the CPU and the GPU does not yield any performance benefit. The best performance is achieved when scheduling a single chunk on the GPU, which outperforms the two CPU cores and GPU case by 28%.

This application is especially a challenge for the CPU and the GPU load balancing. For instance, the CPU-GPU implementation involves many small memory transfers between the CPU and the GPU (e.g. border strips within chunks). The API calls contribute significant overhead. In addition, parallelism exists across a diagonal strip of chunks and across a diagonal strip of data blocks within each chunk. Two adjacent diagonal strips must be processed in serial. Even when the input size is large enough to get close to the GPU memory capacity, the amount of data that can be processed in parallel (along the diagonal) is still small; this does not allow each chunk to fully utilize the GPU.

Needleman-Wunsch and Smith Waterman possess very similar access patterns in the first phase of filling the score matrix. Previous work [56] shows that Smith Waterman demonstrates bad performance when scheduling its works simultaneously on the CPU and the GPU, which agrees with our result.

4.3.6 Load Balancing Benefit

In this section, we discuss how much performance benefit we can achieve through load balancing across the CPU and the GPU. We assume that the CPU and GPU take T_{cpu} and T_{gpu} to process the same amount of work. Therefore, $T_{loadbalancing} = \frac{T_{cpu} \times T_{gpu}}{T_{cpu} + T_{gpu}}$ is the time spent on executing workloads on the CPU and the GPU. If using the GPU processing time as the baseline, the benefit of simultaneous CPU and GPU computation against GPU-only computation can be denoted as $\frac{T_{gpu} - T_{loadbalancing}}{T_{gpu}}$.

Figure 4.8 shows the performance benefit of load balancing versus the increasing CPU-GPU speed gap. When the CPU and GPU take equal time to process the same amount of computation. The performance of load balancing is 50% faster than CPU or GPU-only execution. As the GPU's processing speed becomes significantly faster than CPU (e.g. larger than 10$\times$), performance benefit becomes smaller (e.g. less than 10%), and scheduling and communication overheads might become significant and further reduce the benefit. Therefore, load balancing will get its most benefit when CPU and GPU processing speeds for a particular application are close to each other.

Lee et al. [48] point out that many works reported dramatic speedups obtained from GPU acceleration for many applications, however one reason is that the CPU implementations are usually less optimized leading to unfair comparisons. Their study shows that most GPU applications in their experiment, if compared against sufficiently optimized CPU codes, only achieve less than 6$\times$ speedup [48]. This suggests that many applications may potentially benefit from CPU-GPU load balancing.

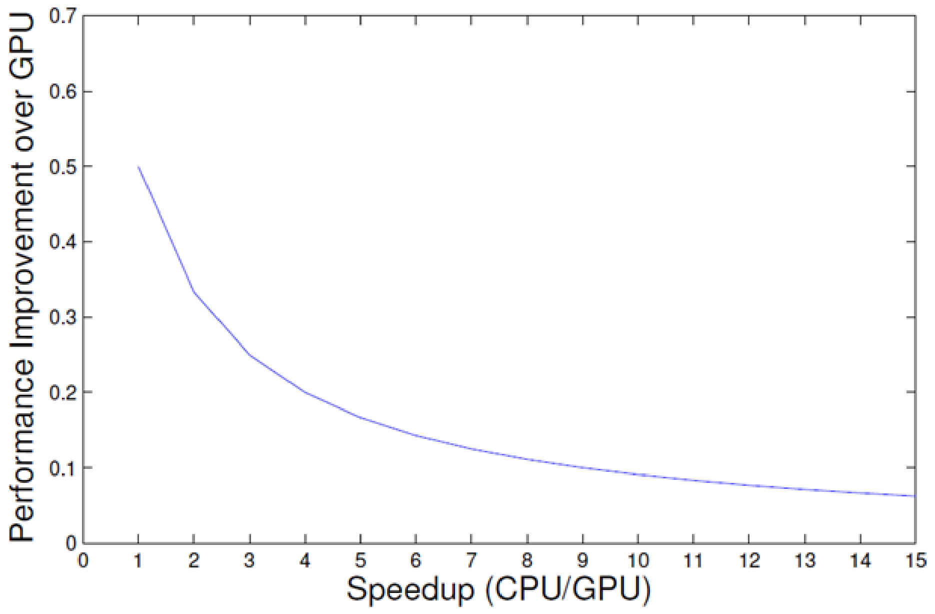

Figure 4.8: Peformance benefit of load balancing versus CPU-GPU speedup

4.3.7 The Choice of Chunk Size

One important consideration is that computation rate of the GPU can be influenced by chunk size. The rate of the CPU, C_{cpu}, is relatively constant. In another word, CPU execution time will increase linearly with increasing number of chunks. However, for many GPU applications, the speedup of the GPU against the CPU with growing input size usually increases first and then starts to reach an asymptote. This is due to the fact that small inputs can not fully utilize the GPU. Since we use each GPU kernel call to process one data chunk, the choice of chunk size is an important factor to performance.

Figure 4.9 shows the Kmeans' execution time when we change the chunk size from 16k to 64 k, given the same total input. The performance with a chunk size of 16 k is 20% slower than that with a chunk size of 64 k under a GPU-only configuration. Similarly, the performance with a chunk size of 16 k is 13% slower than that with a chunk size of 64 k in a 1 CPU + GPU configuration. All our applications use big enough chunks for the GPU.

Other techniques can be integrated into our framework to deal with the chunk size issue. Boyer et al. [10] study a scheduling technique to determine appropriate chunk sizes at runtime with profil-

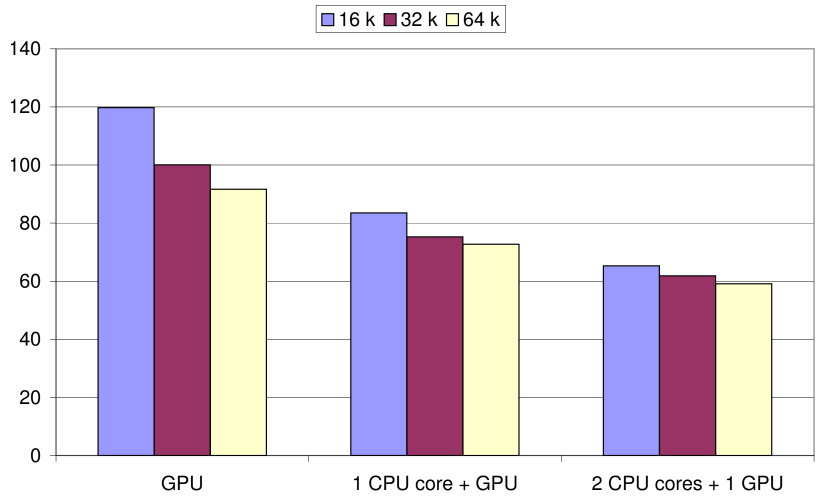

Figure 4.9: Change of execution time when we change the chunk size from 16 k to 64 k.

ing. Also, the issue will become less a concern on recent GPU implementations, such as NVIDIA's Fermi cards. Fermi GPUs support features such as issuing instructions from multiple independent *streams*. When memory transfers are not involved, multiple concurrent kernel calls can be issued from different streams to efficiently utilize GPU's hardware resources. Therefore, the execution time of one kernel call computing a big chunk with a size of $N \times S$ can be similar to that of N concurrent kernel calls, each processing a small chunk with a size of S. We plan to extend our work taking advantage of this feature in future work. When memory transfers are involved, the issue of using small chunk size can be alleviated by overlapping kernel computations and memory transfers across multiple different *streams*, which we will discuss in details in Chapter5.

4.4 Conclusions and Future Work

To take advantage of the available resources in a heterogeneous system, we study an approach to execute parallel compute kernels simultaneously on the CPU and the GPU and allow efficient data partitioning and load balancing across devices. Most previous works solve this problem manually or their approaches are not adaptable to run-time resource changes.

To overcome these limitations, our work is based on hierarchical domain partitioning with chunking and work queues. We include scheduling strategies to ensure better data locality to save costly PCI-E transfers. We use four applications as case studies– *Kmeans*, *SRAD*, *Sepia* and Needleman Wunsch and show that various performance gains are achieved due to different CPU-GPU speed gaps.

Collaborating with AMD, we are integrating our approaches into the AMD OpenCL runtime. This will potentially allow future OpenCL applications to obtain higher performance from concurrent CPU-GPU execution without any need of source-code modifications.

We restrict our study to applications that are data parallel, and computations can be easily partitioned across the CPU and the GPU. Future work includes evaluating applications with irregular parallelism and studying approaches to automatically decide desirable chunk sizes for different applications. In Chapter 5, we will present a technique of conducting memory remapping for desirable data layouts of different devices, which helps improve the load balancing performance.

Chapter 5

Dymaxion

The fast growing core counts in today's multicore and manycore arcthiectures exert significant pressure on the memory interface. Memory bandwidth and latency are improving at a slower pace and limit overal system throughput. High application performance relies on effective memory bandwidth utilization. Today's GPU programming models (e.g. CUDA and OpenCL) require programmers to spend considerable effort to optimize memory accesses for high performance. GPU's specialized memories (*shared*, *constant*, and *texture* memories) present different access patterns and require specialized mappings. GPUs' SIMD architectures require efficient memory coalescing for inter-thread data locality. Furthermore, for efficient heterogeneous computing, different architectures and multithreading models may favor different memory mappings, which brings up performance portability issues across devices.

Dymaxion addresses these concerns with a set of high-level software abstractions, APIs, and underlying mechanisms to ease programmer burden while improving memory access efficiency in unoptimized code. Dymaxion currently targets GPUs, but can be targeted to any platform. Dymaxion is also helpful for increasing the efficiency at each node for high performance computing, given the growing use of GPUs. For instance, as with any GPU cluster, an MPI process launched on each node can make use of GPUs by making CUDA calls. We find that optimizing access patterns yields substantial performance improvement by effectively hiding memory remapping latency.

This work makes the following contributions:

- We present an API framework and a data index transformation mechanism that allows users to reorganize the layout of data structures such that they are amenable to localized, contiguous access by simultaneous GPU threads. In CUDA, this implies that thread accesses can be coalesced efficiently.

- We show how to hide the overhead of layout remapping during PCI-E transfer, taking advantage of simultaneous CUDA streams. Memory layout transformation is divided into separate chunks and overlaps with PCI-E memory transfer. We also compare it to a technique which takes advantage of the *zero copy* feature on the GPU.

- We evaluate several representative access patterns common in many scientific applications and use several case studies to present the use of our framework in achieving better coupling of access patterns and actual memory layouts.

- We present a case study of spreading work simultaneously across the CPU and the GPU, in which the two platforms prefer different mappings of data layouts and access patterns respectively. We show that our framework is a clean abstraction and convenient software-level building block to ensure cross-device data coherency.

An API-based remapping mechanism has the benefit of giving programmers more control and flexibility over data mapping. Dymaxion allows hints to be provided to the system to influence memory mapping based on programmers' knowledge of the algorithms.

Another advantage of an API-based approach is portability across platforms, as an API can be optimized for different architectures. We develop Dymaxion as an extension to NVIDIA's CUDA; however, the same framework can be extended to other GPU or heterogeneous programming models, such as OpenCL [64]. Four diverse applications from the Rodinia suite are used in our evaluation [14]. Using Dymaxion on a GTX 480 GPU, an average of 3.3$\times$ speedup is achieved on compute kernels and a 20% performance improvement is achieved, including the PCI-E transfer, when compared with their original CUDA implementations. Additionally, the extra programming effort involved in using Dymaxion is trivial.

The work of this chapter has been published in Supercomputing 2011 [18].

5.1 Motivation

The impetus for Dymaxion lies in three key observations, discussed here.

5.1.1 CUDA Coalescing

One important performance optimization for GPUs (supported on NVIDIA hardware starting with the GT200 generation) is the coalescing of global memory accesses generated by streaming multiprocessors (SMs). The SMs schedule and execute threads in lock-step groups of 32 threads called *warps*. Global memory accesses within a half-warp will be coalesced into the minimum number of memory transactions [27]. Figure 5.1 shows a simple example: if the k^{th} thread accesses the k^{th} word in a segment, a single 64-byte transaction is required. Different scenarios and requirements for memory coalescing are documented in detail in the NVIDIA technical guides [25, 27].

5.1.2 Memory Locality of Inter-thread Accesses

The following code segment shows two simple examples of CUDA code that loop over the data elements of a 2-D array and assigns their values to another array. In each iteration, a strip of data elements is accessed concurrently.

```
/* The CUDA implementation               */
int bx = blockIdx.x;  /* thread block ID */
int tx = threadIdx.x; /* thread ID       */
int tid = BLOCK_SIZE * bx + tx;
//Example 1: access different rows (row-major)
for (i = 0; i < N; i++) {
        des[cols * tid + i] = src[cols * tid + i];
}
//Example 2: access contiguous data elements (column-major)
for (i = 0; i < N; i++) {
        des[cols * i + tid] = src[cols * i + tid];
}
```

In this implementation, the accesses are parallelized, each thread responsible for processing one element. One important observation is that if the thread id, `tid`, is used as the lowest dimension of the index to access an array, as in `array[cols * i + tid]` (See Example 2), multiple simultaneous threads will access contiguous memory locations. Thus, the memory accesses of the second loop manifest better inter-thread spatial locality than those of the first. In fact, the need for both types of accesses comes up in many applications (e.g. matrix multiplication).

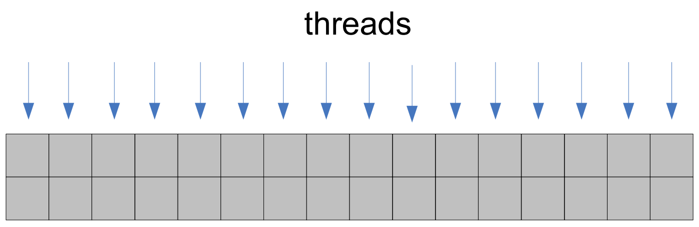

Figure 5.1: The memory coalescing concept. If threads access contiguous data elements, multiple thread accesses can be coalesced. Each element is 4 bytes in this example.

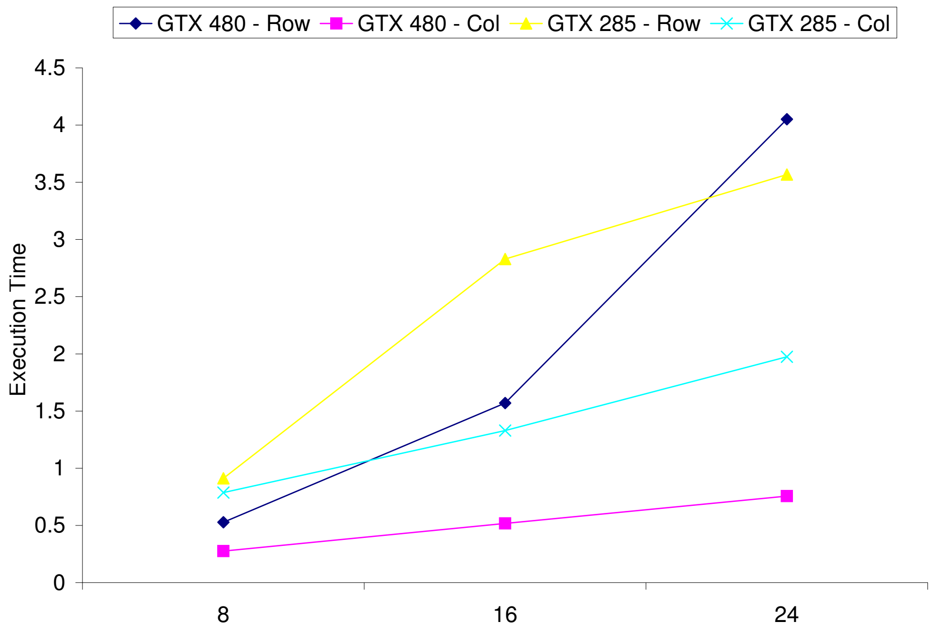

Figure 5.2: The *x*-axis represents feature size while the *y*-axis represents execution time [18]. Execution time is one iteration of the *k-means* distance kernel with an input of 64 k data objects. More features mean more uncoalesced memory accesses.

An example shows how poor locality of concurrent memory accesses leads to poor performance and scalability of GPU applications. Figure 5.2 shows the performance of two versions of a *k-means*

GPU implementation, which we will discuss in Section 5.4.1 in more details. One version assigns each thread to compute a row of the main data structure, which is row-major. In this version, each row represents a data element while each column represents a feature; different data elements can be processed in parallel. This organization results in suboptimal memory coalescing for threads within a warp. In contrast, the other implementation uses a column-major layout, in which threads within a warp access adjacent data elements and achieve better inter-thread locality. We vary the number of features in the main data structure and measure execution times on NVIDIA GeForce GTX 480 and 285 GPUs.

The column-major organization achieves better performance on both platforms. But when the number of features surpasses 16, the 480, a more powerful GPU running a row-major based *k-means*, actually yields poorer performance than the 285; this application is memory-bound and benefits from coalescing, which requires a column-major organization so that warps access contiguous data. This example illustrates how much impact memory access patterns play in GPU performance. To solve this issue, Dymaxion enables programmers to match memory access patterns and data layouts automatically through a simple programming interface that declares the access pattern in terms of the original data structure. This high-level knowledge then permits transparent memory layout remapping to optimize bandwidth (assuming that all accesses to the data structure are mediated by the API).

On the other hand, CPUs and GPUs may prefer different data layouts for certain applications. This is due to the fact that better cache locality is needed for contiguous memory accesses issued by individual CPU threads, while an efficient GPU memory transaction is desirable to feed data to multiple simultaneous SIMD threads. For instance, for the same *k-means* problem, CPUs, in contrast, favor a row-major layout, as we discuss in Section 5.5. Furthermore, in contrast to the CPU, GPU has a distinct memory hierarchy with specialized memories, each of which prefers a different mapping between data layout and access pattern. For example, a typical texture unit design adopts a Morton-curve access pattern. An efficient use of constant memory requires simultaneous thread accesses from a single warp to touch the same cache lines; therefore, hand-optimizing memory mappings for different platforms is not only tedious, but also the relevant code may need rewritten

Table 5.1: Fraction of total execution time devoted to PCI-E transfers [18]

Applications	**PCI-E Transfer**	**GPU Kernel**
K-means	51%	49%
Needleman-Wunsch (NW)	32%	68%
SpMV	77%	23%
Nearest Neighbor (NN)	70%	30%

for good performance and portability across platforms. To resolve these issues, we need a high-level abstraction to define memory mappings.

5.1.3 Making Data Ready on the GPU During PCI-E transfer

Often GPU applications expend significant time on data transfer between system and GPU device memory [23,27]. Because PCI-E transfers have less available bandwidth than DRAM accesses, and also because of the device call overhead associated with each transfer, an efficient implementation should minimize data transfer, both instances and volume.

Table 5.1 shows the fractions of total execution time dedicated to CPU-GPU memory transfer and GPU kernel execution. PCI-E transfers consume a large fraction of execution time in all four applications. In order to both leverage and reduce this overhead, we propose that additional functionality, such as memory remapping, be implemented during PCI-E communication in order to increase memory locality for subsequent GPU computation. Such functionality can be implemented either in software (the driver), hardware (the DMA mechanism), or both. Ideally, these operations would be programmable to maximize their generality. Because we do not have access to proprietary GPU drivers, our prototype Dymaxion implementation takes advantage of CUDA stream functionality to aid data reorganization. Furthermore, the data structure is also broken into chunks to hide the latency of memory remapping, described further in the next section.

5.2 Dymaxion Design and Implementation

In this section we describe the design and implementation of Dymaxion.

5.2.1 CPU-GPU Data Transfer and Remapping

In our framework, programmers start by calling remapping functions on the target data structures. This launches a series of operations to transfer data between the CPU and the GPU in a remapped order that yields efficient data access for the GPU compute kernel (see Figure 5.3). The memory layout transformation brings new overhead, which we attempt to minimize while trading it off for improvements in GPU data locality. If the reorganization overhead is less than the time required for PCI-E transfer, most of it can be hidden through pipelining. We found that CPU exhibits lower bandwidth than the PCI-E transfers, so we decided to take advantage of the high bandwidth and deep multithreading features of the GPU for layout reorganization.

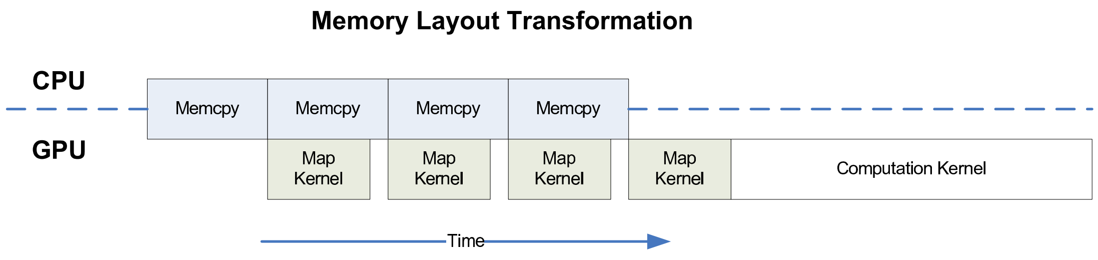

Figure 5.3: Memory layout reorganization. The entire data structure is broken into small chunks and transfered from the CPU and the GPU chunk by chunk. After each chunk completes transfer, layout reorganization is applied to that particular chunk on the GPU. We assume *memcpy* is executed sequentially.

Currently, the remapping flow can be broken down into two major components:

1. Break the data into small chunks and transfer each chunk asynchronously from the CPU to the GPU one by one.

2. Immediately after data transfer of each chunk, launch remapping kernels to reorganize data layout; each thread is responsible for relocating one data element.

Figure 5.3 illustrates the idea of overlapping PCI-E transfer and layout transformation on the GPU. In this example, we assume the PCI-E implementation is serial. In reality, it can be implemented more efficiently using parallelism. However, we are missing implementation details necessary to take advantage of that organization. Because there is a one-to-one mapping between the original and remapped locations, and because the remappings do not overlap, the remapping of data are independent; however, the latter invariant introduces storage overhead, mitigated by chunking.

Figure 5.4 shows sample code illustrating one possible implementation with CUDA streams. CUDA applications often manage concurrency through streams [27]. A stream, in CUDA, is a sequence of commands that execute in order. Distinct streams are only partially ordered [27]. To use streams, we allocate our host memories with `cudaMallocHost()`. This example is similar to the stream example in the CUDA programming guide [27], the difference being that, for each CUDA stream, `stream[i]`, we first copy a chunk and then execute a specific kernel to perform layout remapping for that chunk. The chunk index `i` and chunk size are used to determine which data to reorganize. In Section 5.4.7, we compare this approach with an alternative one using the *zero-copy* feature.

Note that data reorganization is one of the implementation options for memory remapping in Dymaxion, designed as a high-level abstraction. Other possible approaches include physical-address-to-physical-address translations and associated latency hiding techniques [85]. Another possibility is to leverage MMU for layout transformation, avoiding the extra CPU-GPU copy, but that doing the transformation across the PCI-E hub may not be as efficient as a bulk copy onto the GPU card, from which the GPU's massive bandwidth and parallelism can be leveraged to speed up the transformation.

5.2.2 Index Transformation

Following the layout transformation, a GPU device memory pointer for the reorganized data structure is returned for the user to pass to the compute kernel. Because of the change in layout, the indices of future accesses must also be transformed. For each type of layout transformation, we provide a corresponding index transform function to achieve this functionality. For example, indexing a specific data element, `array[index]`, is achieved after the layout transformation with `array[index_transform(index)]`. This is the only change required to apply to the original GPU kernel code.

```
/* divide the work into chunks */
int chunk_size = size / num_kernels;

/* create CUDA streams */
cudaStream_t *stream = (cudaStream_t *) malloc(num_kernels
                        * sizeof (cudaStream_t));

for (i = 0; i < num_kernels; i++)
  cudaStreamCreate(&stream[i]);

/* launch the asynchronous memory copies and map kernels */
for (i = 0; i < num_kernels; i++)
  cudaMemcpyAsync(array_d + i * chunk_size,
                  array_h + i * chunk_size,
                  sizeof (float) * chunk_size,
                  cudaMemcpyHostToDevice,
                  stream[i]);

for (i = 0; i < num_kernels; i++)
  map_kernel<<<grid, block, 0, stream[i]>>>
            (array_d_map, /* remapping destination    */
             array_d,     /* input array              */
             i,           /* chunk index              */
             chunk_size,  /* chunk size               */
             num_kernels  /* num simultaneous kernels */);
```

Figure 5.4: CUDA streams are utilized to overlap chunk transfer with remapping. This example assumes that the entire work size can be evenly divided by the number of chunks

```
//Original Version
__global__ kmeans_distance(float *feature_d, ...){
  //feature_d is the original array
  int tid = BLOCK_SIZE * blockIdx.x + threadIdx.x;
  /* ... */
  for (int l = 0; l < nclusters; l++) {
    index = tid * nfeatures + l;
    ...feature[index]...
  }
}
//Dymaxion Version
__global__ kmeans_distance(float *feature_remap, ...){
  //feature_remap is the remapped array
  int tid = BLOCK_SIZE * blockIdx.x + threadIdx.x;
  /* ... */
  for (int l = 0; l < nclusters; l++) {
    index = tid * nfeatures + l;
    ...feature_remap[transform_row2col(index,
                                      npoints,
                                      nfeatures)]...
  }
}
//Manually Mapped Version
__global__ kmeans_distance(float *feature_remap, ...){
  //feature_remap is the remapped array
  int tid = BLOCK_SIZE * blockIdx.x + threadIdx.x;
  /* ... */
  for (int l = 0; l < nclusters; l++) {
     index = l * npoints + tid;
    ...feature_remap[index]...
  }
}
```

The above code snipplets shows an example of the necessary changes to the *k-means* distance kernel and its associated index transformation function (a row-major to column-major transformation). We apply the layout transformation on the `feature` array, which is the primary data storage structure in this implementation. To access an element, the general form `feature_remap[transform_row2col(index, npoints, nfeatures)]` is used in place of the basic `feature[index]` lookup. Programmers can choose to manually modify the index without using a Dymaxion index transformation function (e.g. swapping the loop index), only if they know exactly how a specific layout is optimized by the remapping function on a particular platform (e.g., DRAM parallelism and memory alignment). Because different platforms may prefer different layouts, the index transform function is preferable, as it maintains code portability across platforms without any need of manual effort. It is also a convenient tool to help programmers transform complicated index term and will be needed when implementation details are hidden from programmers.

5.2.3 Dymaxion API Design

Dymaxion currently optimizes single-dimension linear memory accesses (e.g. `array[index]`) to GPU global memory for various access patterns. The framework can be extended to support other memories, for example texture memory. There are several important design goals we used to guide our API development:

- Dymaxion should provide abstractions for specifying various access patterns, and the implementation of Dymaxion should rely on and can be optimized for different architectural details.
- Programmers should not be required to program with Dymaxion, which is primarily for optimization.
- The implementation should *be an API*, with a small set of extensions to existing languages, not an entirely new language.

Map Functions:

```
void map_row2col(  void       *dst,
                   const void *src,
                   unsigned   height,
                   unsigned   width
                   type_t     type);

void map_diagnal(  void       *dst,
                   const void *src,
                   unsigned   dim,
                   type_t     type);

void map_indirect( void       *dst,
                   const void *src,
                   const void *index,
                   unsigned   size,
                   type_t     type);

void map_arrstruct(void       *dst,
                   const void *src,
                   unsigned   argc,
                   arg_list   *list);
```

Index Transform Functions:

```
unsigned transform_row2rcol(unsigned index,
                            unsigned height,
                            unsigned width,
                            type_t   type);

unsigned transform_diagonal(unsigned index
                            unsigned height,
                            unsigned width,
                            type_t   type);

void     *transform_struct( void     *array,
                            unsigned tid,
                            unsigned num_mem,
                            unsigned num_nodes,
                            unsigned mem_offset,
                            type_t   type);
```

Dymaxion is not restricted to GPU use, but also applicable to other heterogeneous platforms. Compared with compiler-based tools, Dymaxion gives programmers more control while saving them significant optimization effort. The Dymaxion framework consists of two major parts: 1. A set of remapping functions to direct data remappings, and 2. associated index transformation functions.

The function list shows the current API functions implemented in Dymaxion. So far we have implemented our API for *row-major order to column-major order*, *diagonal-strip*, *indirect*, and *array-of-struct* transformations, which cover the access patterns common in many scientific applications. Note that API functions such as `cudaMemcpy()` in CUDA and `clEnqueueMapBuffer()` in OpenCL are special cases of memory mappings which map a linear region of memory space in the host to a region on the device. We do not think this is an complete list of API functions; Dymaxion is extensible to other access patterns (e.g. Morton and other space-filling curves, graph traversal), a task we leave for future work.

At the same time, there is a way in Dyamxion for programmers to define their own memory mapping that is not supplied by the API. For instance, programmers can write a GPU remapping function (equivalent to *map_kernel* in Figure 5.4), following the declaration rules of user-defined function in Dymaxion. As shown in the following example, the function pointer to this GPU remapping function will be used to pass to a Dymaxion `map_user` function, which automatically handles the overlapping of memory transfer and remapping.

```
void     map_user(void         *dst,
                  void         *src,
                  unsigned int height,
                  unsigned int width,
                  type_t       type,
                  usr_func_ptr map_kernel)
```

Our study is restricted to loop-based algorithms, which possess a single major access pattern that dominates computation and thus typically requires only one copy for each remapped data structure. A challenge arises when applications present multiple access patterns in accessing a single data structure. Sometimes it is beneficial to keep separate mappings for each pattern, and special care is needed to maintain consistency among the copies. But even in the presence of multiple access patterns, one desirable layout for the "major" access pattern may still generate better overall performance. Determining the performance benefits of one or many remappings depends on the degree of reuse of different access patterns in each particular application. We leave this for future work.

5.3 Experiment Setup

Our results are based on execution on NVIDIA GeForce GTX 285 and 480 GPUs. The 285 has 240 cores with a 1.48 GHz shader clock, 16 kB shared memory and 1 GB device memory. The 480 has 480 cores, a 1.4 GHz shader clock, 64 kB configurable on-chip cache (shared memory + hardware cache), 768 kB shared L2 cache and 1.6 GB device memory. We use CUDA 3.1 and GCC

4.2.4 with the -O3 flag to compile our programs. To demonstrate the benefits of our framework, we also report the number of global memory loads and stores before and after using Dymaxion. This is measured by using the CUDA profiler with `CUDA_PROFILE=1` on the 285. The CPU we use is an Intel Core2 Quad CPU with a clock of 2.66GHz and a 3MB L2 cache. The results are timed on the main computational loops of the applications and include PCI-E transfer and GPU kernel execution. Also, this study is restricted to cases in which the combined memory spaces consumed by an applications working set and memory remapping does not surpass the capacity of GPU device memory.

5.4 Access Patterns and Experimental Results

In this section, we report the performance improvements of Dymaxion for different memory patterns, each with a widely used, representative application.

5.4.1 Row-Major Order to Column-Major Order Remapping

Figure 5.5 shows a conceptual view of a row-major to column-major transformation. Essentially, a 2-D array with column-major order is a 90-degree transpose of a row-major version of the same data. From an algorithmic perspective, programmers see no difference in the two layouts (assuming of course that the program accesses the data to match the layout!); however, for regular row-wise or column-wise accesses, these two organizations are crucial to memory locality and performance. Switching from row-major order to column major order, the relationship between the new and old array index is described with

```
new_index = height * (old_index % width) +
                      (old_index / width)
```

where all operations are integer and implemented by the index transform function `transform_row2col()` in the API list.

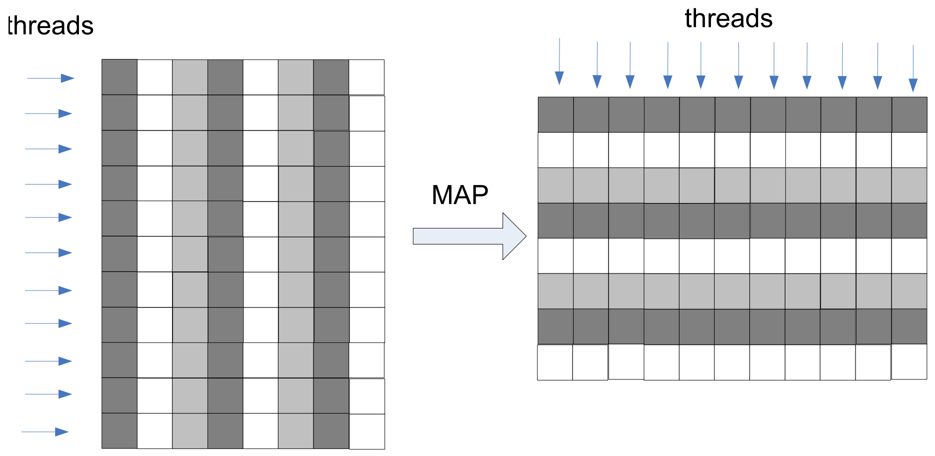

Figure 5.5: Row-major to column-major transformation.

5.4.1.1 *K*-means

In the single-device Rodinia [14] GPU implementation, data are partitioned according to thread blocks, with each thread associated with one data element. The task of searching for the nearest centroid to a given element is independent of all others. We discuss the *k*-means implementation in detail in an earlier work [14, 15].

Programmers often prefer to store data in a 2-D arrays with each row representing a data object and each column representing a feature. Such a layout tends to be inefficient on the GPU; for instance, when threads calculate the distance of individual elements to centroids, they access whole array rows, which are often spread among multiple memory transactions. This is shown on the left in Figure 5.5. On the other hand, inter-thread locality is improved through coalescing after remapping the array into column-major order, shown on the right in Figure 5.5. This example presents a mismatch between the data affinity relationships inherent in the algorithm and the locality characteristics imposed by the mapping of SIMD operations to the DRAM organization.

We applied Dymaxion to the naïve *k-means* GPU implementation from Rodinia. Figure 5.6 shows the performance we obtained for the naïve implementation and the one using Dymaxion. This figure also gives the breakdown of execution time in terms of PCI-E transfer, remapping, and computation. We vary input sizes from 64 k to 256 k elements. In our experiments, the performance of the new version always outperformed the original implementation. On the GTX 480, the

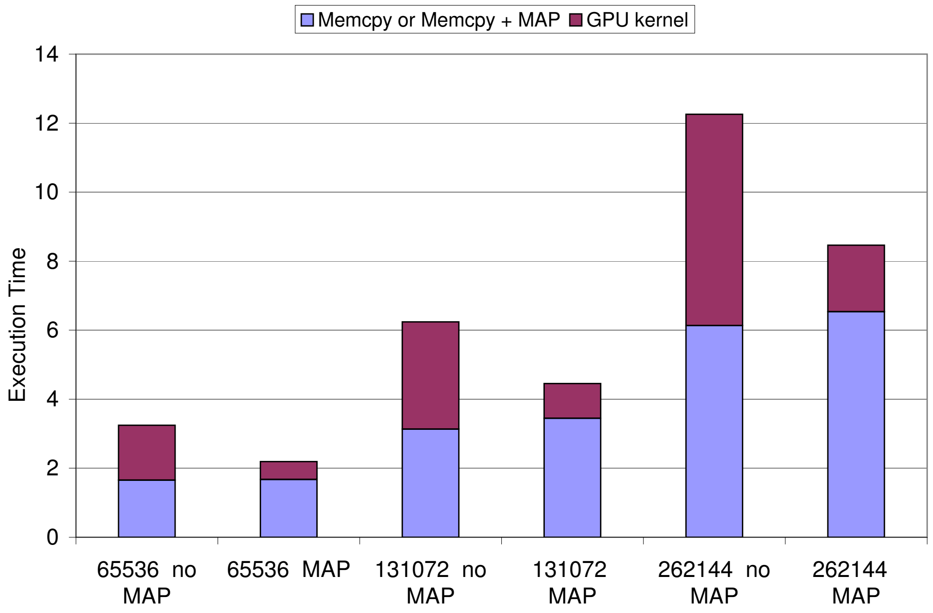

Figure 5.6: The y-axis represents the execution time of one iteration of the *k-means* distance calculation [18]. Execution time is measured for both the original implementation and the port to Dymaxion. *Memcpy+MAP* represents the total amount of time due to layout remapping and data transfer. Because these two operations overlap and CUDA only provides timing for the completion of a whole stream, we measure the end-to-end time and compare it against the original data transfer (i.e. *Memcpy*)

performance of the GPU kernel improves by an average of $3.11\times$ due to better coalesced memory accesses. Considering layout remapping and PCI-E overheads, the overall performance improves an average of 30.6%. The combined PCI-E transfer plus layout transformation incurs only an average of 5.8% overhead when compared with the PCI-E transfer of the original implementation.

5.4.2 Diagonal-Strip Remapping

Often in dense linear algebra and in dynamic programming algorithms, loops manifest memory access patterns other than regular row- or column-wise traversals; however, their access patterns *are* well defined. For example, some applications traverse arrays with constant, non-unity strides. One example is a diagonal strip traversal, which is the result of a constant stride with size $(columns - 1)$. Such patterns tend to have very poor data locality. A diagonal strip is a special case of a strided access.

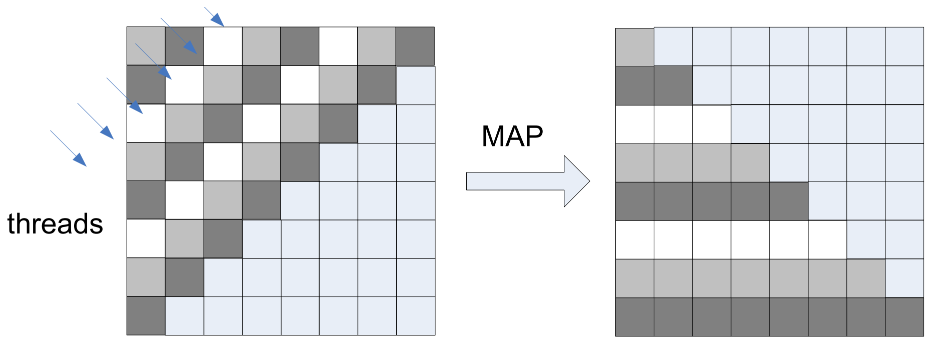

Figure 5.7: Diagonal strip matrix transposition.

5.4.2.1 Needleman-Wunsch

Needleman-Wunsch is a global optimization method for DNA sequence alignment. Potential sequence pairs are organized in a 2-D matrix. The algorithm has two major phases: 1. the algorithm fills the matrix with scores in parallel, which represent the value of the maximum weighted path ending at that cell; and 2. a traceback process is used to find the optimal alignment for the given sequences [15]. Our implementation [15] takes advantage of the GPU's on-chip shared memory to improve program locality and reduce memory latencies; block-level parallelism within *Needleman-Wunsch* is also exploited. We focus on optimizing *Needleman-Wunsch* through efficient memory coalescing starting from a Rodinia version with only global memory accesses.

Figure 5.7 illustrates the memory access patterns of *Needleman Wunsch*. This figure shows the upper-left triangular region of the 2-D matrix and its associated transformation under Dymaxion. For this particular access pattern, the relationship between the new and old array index can be described by the equation

```
new_index = dim * ((old_index % dim)  +
                   (old_index / dim)) +
                    old_index / dim
```

This transformation is achieved via the `transform_diagonal()` function in the API list. Prior to the layout transformation, parallelism exists within each diagnal strip, and each thread is assigned to compute one data element. Using our API function, programmers can make a 45 degree transposition of the matrix. The resulting layout allows threads to concurrently access data elements within the same row. In *Needleman-Wunsch*, the result of the GPU computation must be copied back to the CPU for the serial trace back; therefore, at the end of the GPU kernel, a reverse transposition is applied.

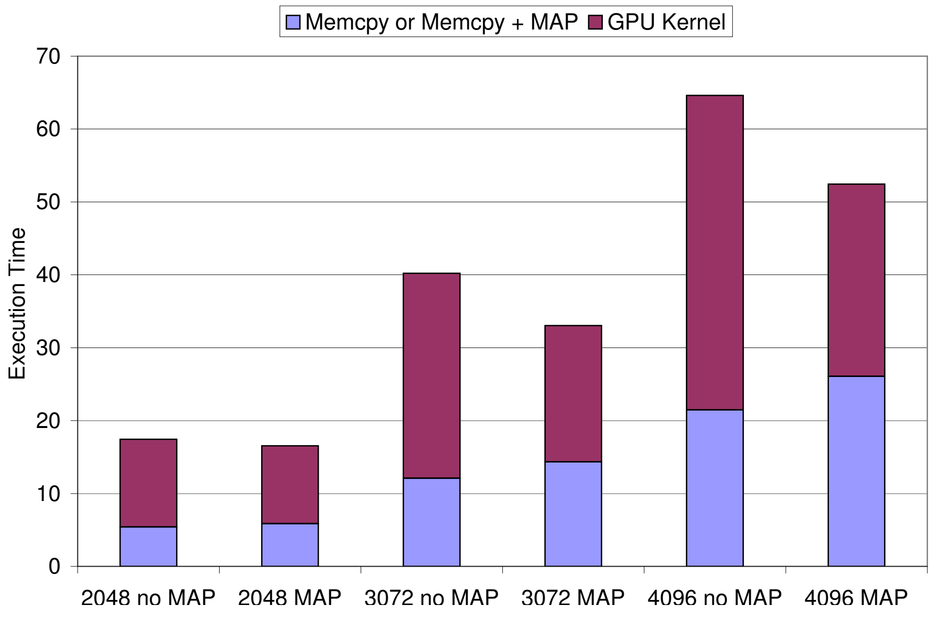

Figure 5.8: The y-axis represents the execution time of the *Needleman-Wunsch* kernel [18]. Execution time is measured for both the original and the Dymaxion implementations.

We applied Dymaxion on the original, naïve *Needleman-Wunsch* GPU implementation. Figure 5.8 shows a performance comparison between the original implementation and the one using Dymaxion. We varied the input sizes from 2048^2 to 4096^2 data elements. On the GTX 480, the kernel performance improves by an average of 42.2%. The overall improvement averages 14.0% after accounting for PCI-E transfer and layout reorganization. The combined PCI-E transfer plus layout transformation incurs an average of 16.1% overhead when compared with the PCI-E transfer of the original implementation. Also, the best-performing *Needleman-Wunsch* version in Rodinia is 25% faster than the current Dymaxion version, because it uses the GPU shared memory, which Dymaxion does not support now, and which we leave for future work.

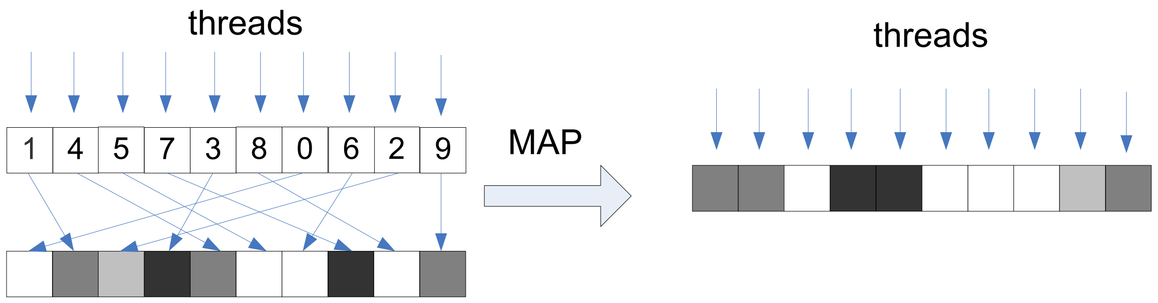

Figure 5.9: Indirect remapping for gather.

5.4.3 Indirect Remapping

Scatter and gather are two fundamental operations in many scientific and enterprise computing applications. They are very common in sparse matrix, sorting and hashing algorithms [31]. Accessing randomly-distributed memory locations makes poor use of GPU memory bandwidth. We evaluate a sparse matrix-vector multiplication (*SpMV*) to demonstrate Dymaxion's support for gather operations. A similar approach can be applied to scatter operations as well.

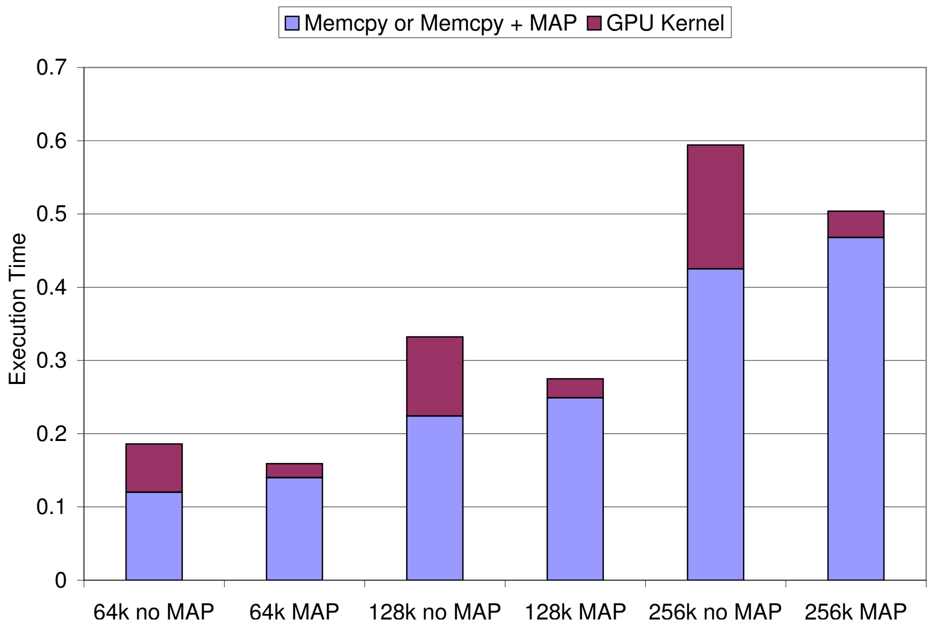

Figure 5.10: The y-axis represents the execution time of *SpMV*, not including reduction [18].

5.4.3.1 Sparse Matrix-Vector Multiplication

Our implementation adopts the compressed row format (*CSR*) to represent the sparse matrix [5,31]. A 2-D sparse matrix, `M`, is encoded using three arrays: `DATA`, `ROWS`, and `COLUMN`. The non-zero elements of `M` are stored in the compressed data array `DATA`. Data in `ROWS[i]` indicate where the `i`th row begins in `DATA`. `COLUMN[i]` indicates the column of `M` from which the element stored in `DATA[i]` comes [85].

We use a similar algorithm to those described in previous work [31, 84], which computes the multiplication `W = M * V`. The algorithm computes `W` in two steps: 1. compute the partial multiplication results and store them in array `R` where `R[i] = (DATA[i] * V[COLUMN[i]])`; 2. perform a reduction stage on the partial results in `R` [31]. In the first stage, `V` is first transfered to the GPU and then we perform the indirect transformation on `COLUMN`, which is chunked and transfered to the GPU, overlapping with a gather from `V` to `V'`; this stage is handled by the `map_indirect()` function. On the kernel side, after remapping, users can directly access `V'[i]` with continuously gathered data, instead of `V[COLUMN[i]]`.

Figure 5.10 shows the performance improvements and execution time breakdowns for the sparse matrix-vector multiply. We varied the input sizes from 64 k to 256 k data elements. Again, the performance of the implementation with Dymaxion outperforms the original implementation for all inputs. The new GPU kernel, benefiting from coalesced memory accesses, improves 4.1× from the original GPU kernel on the GTX 480. The overall performance, including the PCI transfer and layout remapping, improves by an average of 15.6%. The combined PCI-E transfer plus layout transformation incurs an average of 10.2% overhead when compared with the PCI-E transfer of the original implementation.

5.4.4 Struct-Array Transformation

A record or structure (`struct` in C) is an aggregate type which can store multiple data members grouped together under one name. The code on the left above shows an array of structures of length `NUM_ELEM`, each of which contains two floating point and two integer members. In algorithms where the elements are independent, each can be assigned individually to a thread for computation. But

```
#define NUM_ELEM 256
struct my_struct_t {
  float a;
  float b;
  int   c;
  int   d;
} mystruct[NUM_ELEM];
```

```
#define NUM_ELEM 256
struct my_struct_t {
  float a[NUM_ELEM];
  float b[NUM_ELEM];
  int c[NUM_ELEM];
  int d[NUM_ELEM];
} my_struct;
```

because the structure members were laid out contiguously, multiple thread accesses to the same member of different structs may exhibit poor data locality.

Organizing data as a structure of arrays is often preferable to an array of structures for memory access efficiency. We provide a simple API which facilitates this transformation for GPU computation. Currently, Dymaxion only supports structures that contain non-aggregate data members, because C provides limited capability to determine the type of variables at runtime. We created an enumerated type which numbers various commonly used built-in types. Users are asked to provide structure details by passing a list of members and their types to the API function. The transformation is achieved by moving data from the array of structures to a single linear memory region, saving all the raw data. For the GPU kernel, we provide index transform functions to access data elements with information such as number of nodes and member offset. The kernel returns a pointer to the location of the resultant value.

5.4.4.1 Nearest Neighbor

Nearest neighbor (*NN*) is an algorithm to find the *k*-nearest neighbors of a data element in an abstract space. Our parallel NN implementation has two major phases: 1. the parallel phase calculates the Euclidean distances from all the data to specified data; and 2. the reduction phase sorts data in order of ascending distance. We are interested in the first phase. Its distance calculation is similar to that of *k*-means, differing primarily in representation, as NN uses an array of structures.

Figure 5.11 shows the performance we obtained for the naïve implementation and the one with reorganized structures. The input sizes are varied from 64 k to 256 k elements. For all inputs, the performance of the optimized version outperforms the original implementation. On the GTX 480, the GPU kernel was able to achieve 4.4$\times$ speedup over the original version, and the performance,

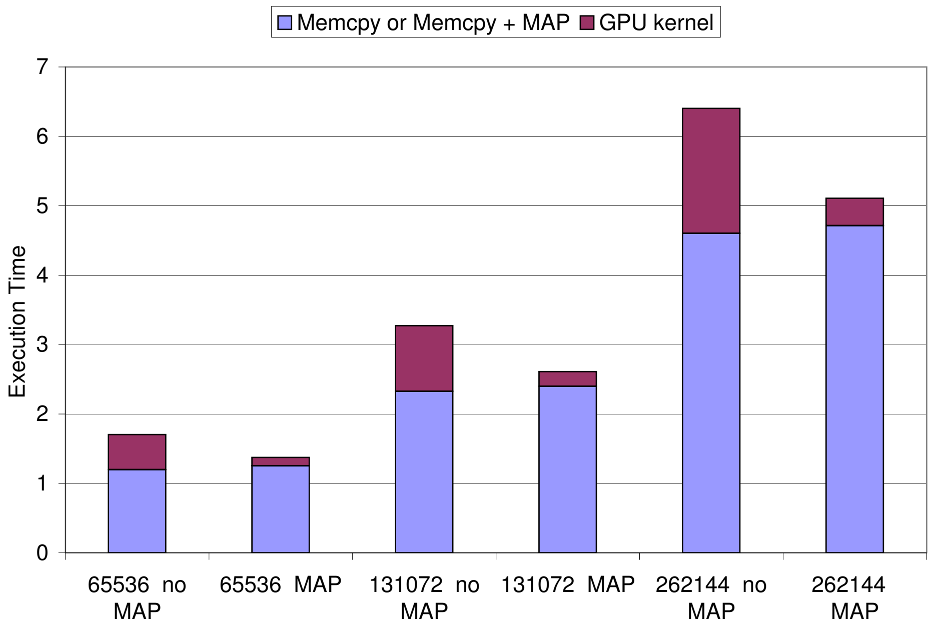

Figure 5.11: The *y*-axis represents the execution time of the first phase of the *NN* implementation [18].

including remapping and the data transfer, increases by 20%. The combined PCI-E transfer plus layout transformation incurs only an average of 3.4% overhead when compared with the PCI-E transfer of the original implementation.

5.4.5 The Benefits of Memory Remapping

To further evaluate the benefits of Dymaxion, we also perform the same set of experiments on an NVIDIA GTX 285 GPU. Figure 5.12 shows the speedups of the GPU kernels with Dymaxion against their original implementations for our applications on both the GTX 480 and GTX 285. The speedups range from 1.4× to 4.4× on the 480 and from 2.1× to 3.0× on the 285. The performance benefits are due to a better match between the memory access patterns and layouts of data structures after applying Dymaxion, which leads to improved memory coalescing in CUDA. We also use NVIDIA's CUDA profiler to characterize the GPU kernels. As shown in Table 5.2, the number of global memory loads and stores is reduced significantly using Dymaxion, with approximate reductions of 4.2× for *k*-means and 3.6× for SpMV.

Table 5.2: Total number of loads and stores of different application-input pairs reported by the CUDA profiler [18]

Application	Input Size	Without Remapping Opt.	With Remapping Opt.
k-means	64 k	289600	68224
	128 k	590784	133824
	256 k	1193152	267648
NW	2048	1536	325
	3072	2432	497
	4096	3200	625
SpMV	64 k	9042	2496
	128 k	19084	4896
	256 k	35846	9792
NN	64 k	188032	60800
	128 k	376064	124132
	256 k	744896	250496

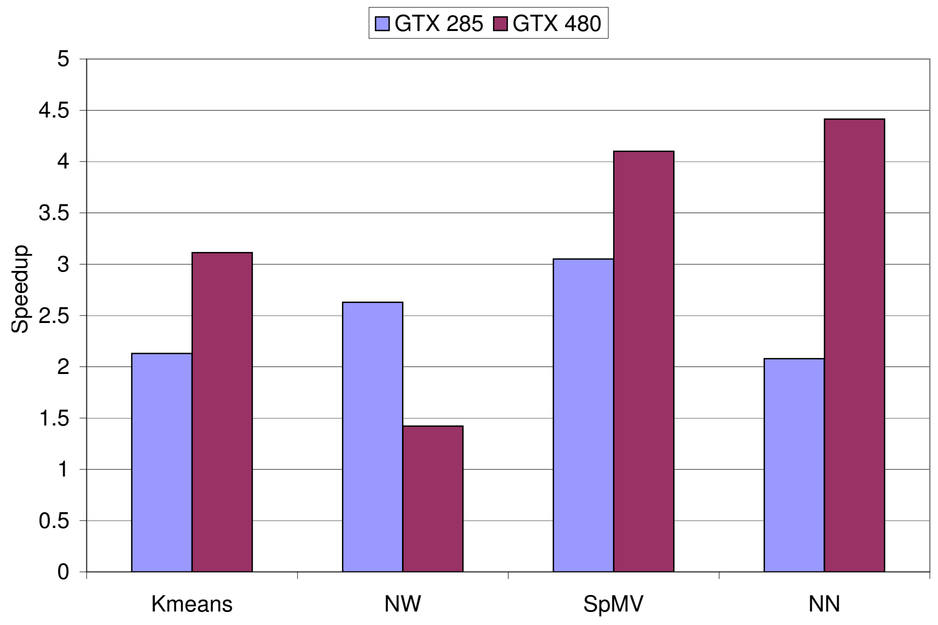

Figure 5.12: The *y*-axis represents the speedup of the GPU kernels with Dymaxion against the original implementation [18].

5.4.6 Chunking Overhead and Parameters for Remapping

Using a microbenchmark, we measured the overhead of conducting PCI-E transfers by breaking up the data into chunks. We consider two scenarios: one in which chunks are transferred with synchronous `cudaMemcpy()` versus another combining streaming and `cudaMemcpyAsync()`. We repeatedly iterate over the loop, transferring one chunk per iteration. Figure 5.13 shows the normalized throughputs measured while transferring a total of 16 MB of contiguous data. We varied the number of chunks from 1 to 512. Conducting smaller data transfers incurs more performance overhead, and the throughputs of both scenarios begins to degrade significantly at about 16 chunks. The latter scenario, streaming + `cudaMemcopyAsync()`, achieves better throughput with an average of 16.7% improvement over the case using synchronous transfers. The benefit is due to multiple streams of chunks and reduced overhead due to the queuing of asynchronous memory calls.

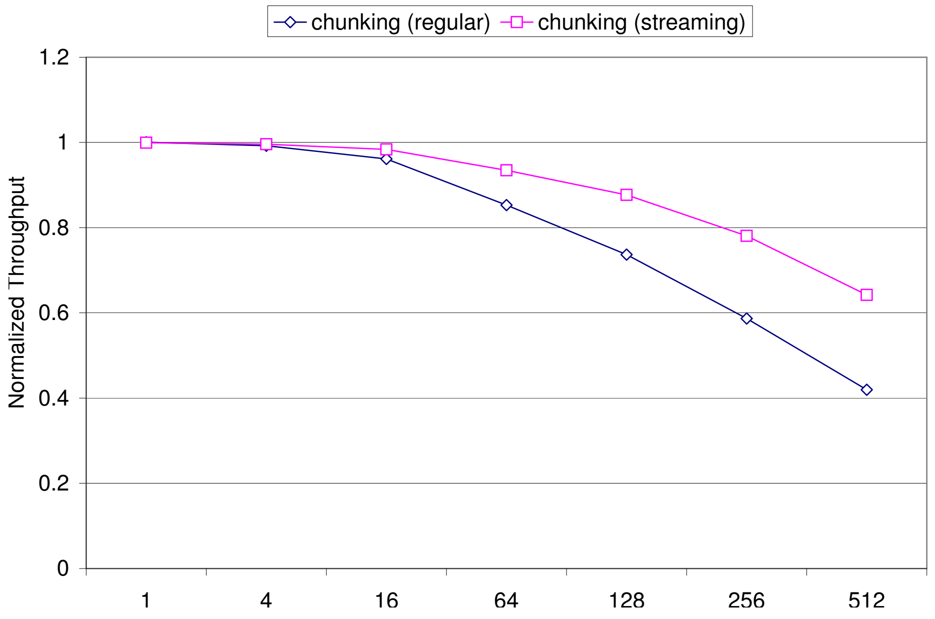

Figure 5.13: The *y*-axis represents the normalized throughput comparing chunking with synchronous memory transfers and chunking with asynchronous memory transfers (streaming) [18]

We also investigate an approach to optimize the performance of remapping, which maps the locations of data elements from one memory space to another. In the GPU implementation of Dymaxion, the remapping is achieved through a call to a GPU kernel. Though these kernels are simple, the performance can vary considerably depending on the amount of work done per thread. Merrill et al. [60] use a set of techniques to optimize GPU-to-GPU data movements. Figure 5.14 shows the normalized performance of a row-major order to column-major order remapping as a function of the number of bytes (4–64 B) each thread reads and writes with a thread-block size of 256. The performance differences between best and worst performing points can be as large as

24%.

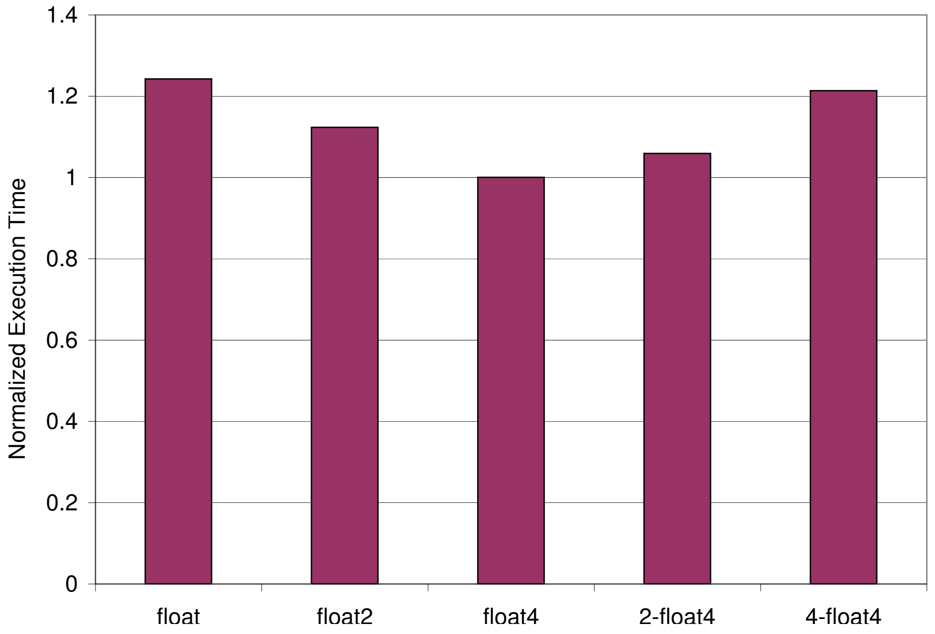

Figure 5.14: The vertical axis represents the performance of remapping (row2col) as a function of the number of bytes (4–64 B) each thread reads and writes [18]. The execution times are normalized to the best performance point. The input is a 16 k$\times$16 float matrix.

5.4.7 Gathering Data through Zero-Copy

In Section 5.2.1, we discussed the approach of hiding remapping latency by overlapping PCI-E data transfer and remapping each chunk, which can be applied in general circumstances. In this section, we use a CUDA-specific feature, *zero copy*, to achieve the same goal of gathering data into contiguous segments. *Zero copy* allows GPU threads to directly access host memory which are page-locked [25]. In the former approach, we transfer each chunk from system memory to an intermediate staging buffer on the GPU and subsequently perform a remapping by saving the final remapped data to a destination buffer.

Using *zero copy*, we launch the remapping kernel with threads sourcing data directly from system memory and then storing the data into the destination buffer in GPU device memory. This saves the memory duplication overhead on the GPU and also obviates the need for each data element to be read and written twice; however, the drawback is that many smaller transactions are

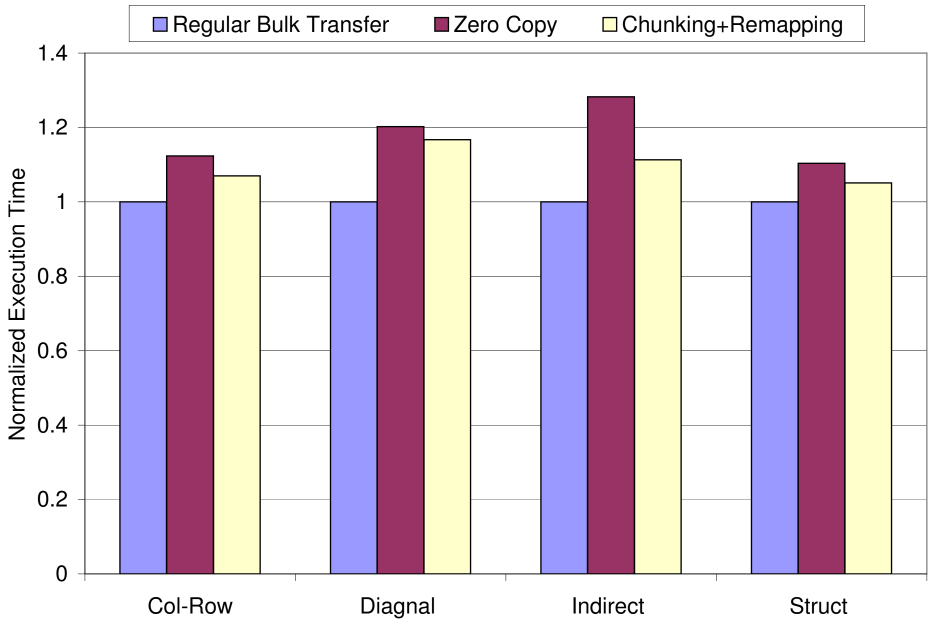

Figure 5.15: The *y*-axis represents the normalized execution time of three types of data transfers [18]. The baseline is the time consumed by the bulk data transfer in the original implementation

needed. Figure 5.15 shows normalized performance results comparing three types of data transfers. *Zero copy* gathers data from non-contiguous memory regions and incurs an average of 7% performance degradation when compared with chunking + remapping; however, all of our applications still get an average speedup of 16.2% on the overall performance due to improved GPU kernel execution time. In other words, our proposed remapping approach has an advantage–despite the extra copy step–because the GPU, working from GPU memory, can achieve higher throughput on data layout transformation than attempting to perform transformation as part of the PCI-E transfer. This is thanks to the GPU's higher memory bandwidth and parallelism, coupled with the chunking approach's ability to hide this latency.

5.5 Case Study: CPU-GPU Simultaneous Execution

In the previous section, we show the performance improvements of offloading work to the GPU and applying our Dymaxion framework. In this section, we present a case study using *k*-means, spreading the workload simultaneously across the CPU and the GPU, which is desirable for two reasons: some CPUs are capable enough to contribute meaningfully to overall performance, and using the CPU will reduce the amount of data that needs to be transferred to the GPU. Some programming models, such as OpenCL [64], support heterogeneous systems, allowing programmers to write one piece of code that is portable to multiple platforms. Unfortunately, one implementation of the compute kernel is usually developed assuming a single data layout, suggesting that it would not work well across diverse platforms.

The Dymaxion framework is useful in this regard, maintaining data coherence while optimizing access patterns across the CPU and GPU. For instance, in a multithreaded *k*-means CPU implementation, each CPU thread is responsible for processing one region of data. Each thread processes one data element and proceeds to process the next element and so on within its own region. Therefore, on the CPU, *k*-means favors a row-major array organization, and the features of a single data element can reside contiguously in cache lines to generate better data locality for distance calculations. This is quite different from the GPU's preference, as discussed previously, for a column-major ordering. Our tests show that a column-major layout degrades CPU performance approximately $2\times$ compared with row-major order, while a row-major layout degrades GPU performance approximately 50% compared with column-major order.

Previous work, including Qilin [56] and Merge [52], presents work-spreading across the CPU and the GPU, but we are unaware of any previous work evaluating the performance impact of different memory mappings when concurrently scheduling workloads on heterogeneous compute resources.

Figure 5.16 illustrates this concept with the *k*-means implementation. To ease the computational domain partitioning over multiple devices, we divide the main data structure into smaller chunks and schedule them on different devices. Load balancing across devices is an interesting research issue in itself and is not the focus of this work. In our experiment, the baseline is a CPU implementation whose data structure is stored in a row-major order; for each chunk dispatched onto the GPU, Dymaxion is applied to remap the chunk into column-major order for efficient GPU execution.

Figure 5.17 shows the normalized execution time for the *k*-means distance kernel with 1.25 M data points and 16 features. When both the CPU and GPU use row-major order, the simultaneous CPU-GPU execution improves the performance by 20% over GPU-only execution. After applying Dymaxion to obtain column-major layout, the GPU-only execution obtains 15% performance improvement over simultaneous CPU-GPU execution with row-major-only order layout. If the CPU uses the row-major layout and GPU uses column-major layout, scheduling *k*-means on the CPU and the GPU further improves the performance by 18% over the GPU-only, column-major layout. As shown in Table 5.3, for the CPU + GPU (row-major order) configuration, the portions of the workloads mapped to the CPU and GPU are 35% and 65%, respectively. Switching to the CPU (row-major order) + GPU (column-major order) configuration, the portions of the workloads mapped to the CPU and GPU change to 25% and 75%.

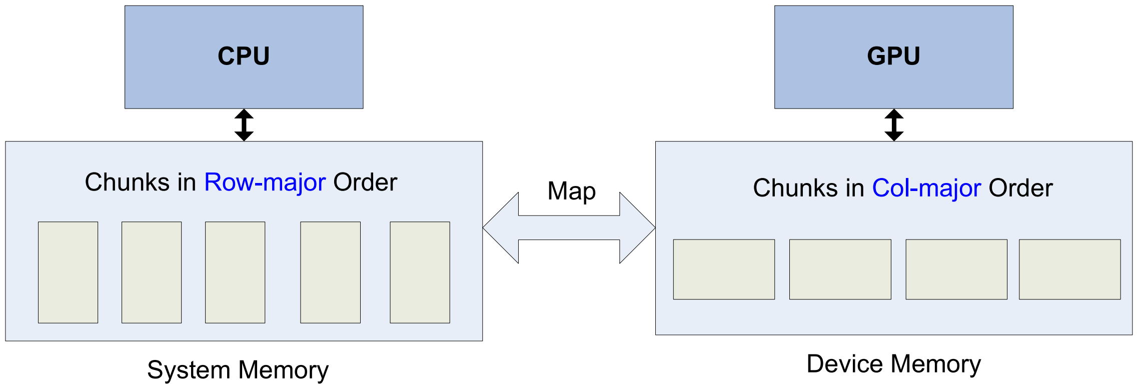

Figure 5.16: The CPU and the GPU prefer different mappings. We divide the data structure into smaller chunks and schedule them onto the CPU and the GPU

Table 5.3: Workload Ratios of *K-means* [18]

Core Combination	**CPU**	**GPU**
CPU + GPU (Row-major Order)	35%	65%
CPU + GPU (Column-major Order)	25%	75%

5.6 Development Cost

The goal of Dymaxion development is to improve the productivity of programmers in optimizing memory accesses. Programming effort must be taken into account in the evaluation of our API's utility. Because it is difficult to get accurate development-time statistics for coding applications, we use *Lines-of-Code* (LOC) as our metric to estimate programming effort. Table 5.4 shows the number

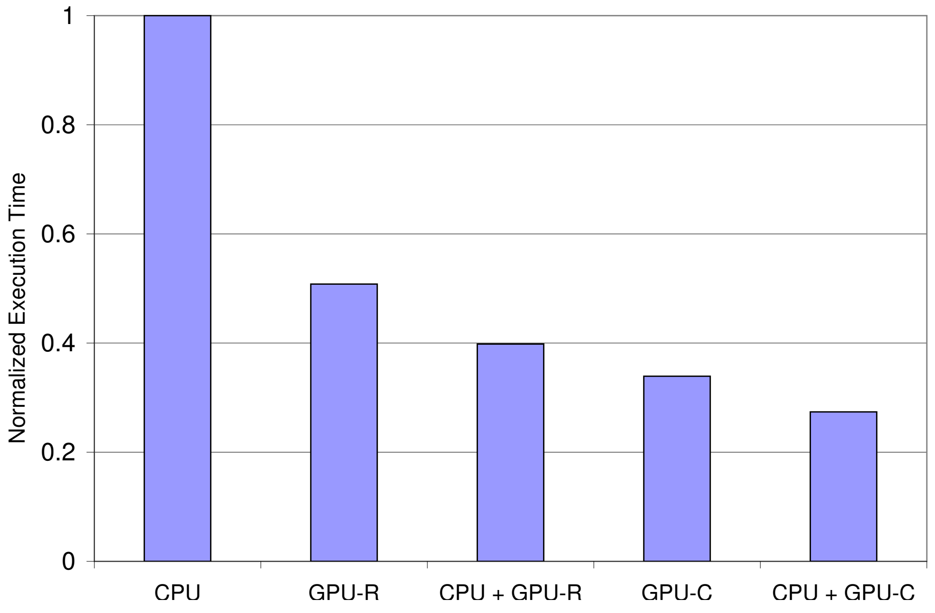

Figure 5.17: The normalized execution time of the CPU-GPU simultaneous execution for one iteration of *k*-means [18]. In all the cases, the CPU uses row-major order. For the GPU implementation, we use two layouts: *R* represents row-major order while *C* represents column-major order

of changed lines of code for the four applications used in this study. For all of the applications, Dymaxion required only 6-20 lines of changes. This suggests the programmer effort of applying our API is trivial compared with the performance gains.

Applications	Kmeans	NW	NN	SpMV
LOC	7	18	20	6

Table 5.4: Development cost measured by number of modified lines of code [18].

5.7 Conclusions & Future Work

In this chapter, we propose the Dymaxion framework to optimize the efficiency of DRAM accesses through memory layout remapping and index transformation. We hide the overhead of remapping through data structure chunking and by overlapping with the CPU-GPU PCI-E data transfer. Usage of our API requires only minimal changes to the original implementations. The four applications we evaluate, each with a unique access pattern, achieve an average of 3.3× speedup on the compute kernels and 20% overall performance improvement, including the PCI-E transfer, on an NVIDIA GTX 480 GPU when compared with their original implementations. The overall benefit is limited by PCI-E overhead, so the benefit will improve as the PCI-E protocol improves or the GPU becomes a peer with the CPU. Also, Dymaxion is a convenient building block to ensure data coherence between the CPU and the GPU for heterogeneous computing; a remapping is needed when writing data to the GPU while a reverse-remapping is needed when reading data from the GPU. We plan to extend Dymaxion to support the transformation of multidimensional arrays and special memories such as texture and shared memory. Dymaxion is available online at http://lava.cs.virginia.edu/dymaxion.

Today's GPU programming models require programmers to manually optimize memory access patterns. Commercial GPU compilers do not yet support Dymaxion-like memory-remappings. We anticipate that the techniques used in this work can be further integrated into compiler frameworks for automated memory remapping. For instance, an OpenMP-like directive can be used to specify the preferred data structure organization for the CPU or the GPU. The compiler can then automatically insert the Dymaxion-like remapping and transformation.

There are also several other directions of future work we plan to explore. This work focuses on a single machine node. Although MPI can launch the same CUDA operations (including Dymaxion remappings) on each node in a cluster, a global, cross-cluster approach may allow further optimizations, especially when cross-node data transfers and system-level interconnect are considered. Additionally, the dynamic detection of application access patterns is a very promising research direction. Currently, Dymaxion requires programmers to manually choose the appropriate API calls for data rearrangement; fortunately, Dyamxion allows programmers to easily roll back to the original version whenever the performance is not satisfactory. Also, because remapping of very large data arrays may introduce additional power overhead, future work will explore the energy efficiency. We also wish to explore opportunities for remapping among different levels of the memory hierarchy, especially when heterogeneous processors share memory (e.g. in AMD Fusion [1]) or even a last-level cache.

Chapter 6

GPU Performance Prediction

With increasing use of GPGPU, research challenges include understanding GPUs' application behaviors, identifying first-order metrics capturing their performance, and designing methodologies for predicting performance. A better understanding of these issues is not only useful for analyzing and comparing different hardware platforms, but also can guide users to choose the best platforms that serve their computation needs.

An important way of helping understand GPUs' performance behaviors is through benchmarking. Because users' applications may not exist in standard benchmark suites, one important goal of benchmark design is allowing users to predict their own applications' performance based on the benchmarks' performance on different platforms. However, predicting performance of arbitrary applications using benchmarks remains an open problem. For instance, in an extreme case, suppose that application A and B present opposite scaling trends; when users intend to purchase a system to improve B's performance, they will make an incorrect decision if they directly use A's benchmark scores.

This work strives to analyze GPU performance on arbitrary user applications by taking advantage of data available for existing benchmarks. Our hypothesis is that, with a set of key metrics and a well-designed benchmark repository, effective performance prediction is possible by sampling and interpolating the benchmark space. Studying this approach also has its practical usefulness. For example, due to limited benchmark mix offered by vendors, customers may sometimes need vendors to port and time applications before customers make large-scale purchases [29], which is a costly procedure. Furthermore, for individual users, they may not have access to hardware before committing the purchase.

To address these concerns, our framework determines the most similar GPU benchmarks as the proxies to an application of interest based on their mutual similarity in the workload space. The predicted performance is expressed as a linear combination of the performance of the proxy benchmarks through interpolation. Our work focuses on manycore architectures e.g. GPUs, and uses a similar approach to the study by Hoste et al. [37] for single-thread CPU applications. We also examine how well the applications included in today's GPU benchmark suites can represent the characteristics of real workloads, which we believe will facilitate a more scientific way for future benchmark suite construction.

Our work makes the following contributions:

- We identify a set of simple, first-order application characteristics for the GPU platform, and analyze their impacts on performance. We then demonstrate the entire flow of the performance prediction framework based on program similarity.
- We use the Rodinia benchmark suite and workloads from other benchmark suites for performance prediction.
- We evaluate the effectiveness of our prediction approach using different processor configurations, program inputs and numbers of nearest neighbors.
- We finally discuss important directions for future benchmark construction. We point out that future benchmark design should adopt a collaborative approach to improve overall feature coverage. One metric to evaluate how well a suite is designed can be its workload space coverage and how effective it can be used for performance projection.

The framework can accurately predict GPUs' application performance except for the applications that are isolated in the workload space. Our experiment result shows an arithmetic mean of 21.6% prediction error across different GPU configurations. In addition, the predicted performance shows a strong correlation with the actual performance from the Spearman Ranking analysis.

6.1 Motivations and Background

6.1.1 Motivations

Users' applications are their best benchmarks. However, because most of these applications are not included in standard benchmark suites, users sometimes need to predict the performance of their applications by referring to benchmark scores of standard benchmarks [37]. Therefore, researching mutual relationships among applications becomes important. One challenge is that it is almost impossible for users to run their applications on all the systems available in the market [29] due to accessibility and costs. Users sometimes have to rely on hardware specifications or white papers to roughly estimate the performance of a platform, which tends to be less accurate than well-designed performance prediction models. Furthermore, most users do not have experience or skills to configure and change architectural simulators to model hardware of interest. Also, simulators are time-consuming and prone to their own inaccuracies.

These issues motivate the need for researching methodologies to correlate the performance of a particular application with that of existing benchmarks to accurately predict the performance of the target application. Gustafson et al. [29] did an early study of performance correlations among benchmarks. Hoste et al. [37,42] pioneered the research of benchmark suite coverage and used standardized benchmarks to conduct performance prediction for single-thread CPU workloads. Carrington et al. [12] predicted application performance using single, simple synthetic metric (i.e. diverse compute kernels) and a linear combination of these simple metrics. For all of these works, one common requirement is to first build a benchmark repository covering diverse application behaviors. As far as we know, there is no previous work studying this issue for manycore architectures such as GPUs.

6.1.2 Prediction for GPU Kernels

A classic debate for CPU benchmarks is whether today's benchmarks can represent the real workloads being used. This issue is well documented in the literature [29] for CPU benchmarking. One major reason is that benchmark designers usually pick relatively smaller problems (e.g. small applications and small inputs) so that the benchmarks can fit all kinds of machines and be generally adopted. However, simple problems sometimes lose important characteristics of real applications [29]. This in turn brings the challenge of predicting the performance of “big” applications with “small” ones, or, in other words, real applications with kernels [12].

However, this is less an issue for benchmarking and predicting GPUs. Today’s GPU programming models like CUDA and OpenCL require programmers to explicitly map algorithms and data structures to their domain-based compute models. GPU applications themselves are a set of well-defined, small compute kernels, accelerating compute-intensive loops. In addition, developers sometimes have to divide a large logical function into constituent smaller kernels due to the global synchronization requirement. Furthermore, the scaling behavior relative to input size, unlike CPU’s input-vs-cache behavior, is predictable once the input is sufficiently large to fill the GPU; processing of the entire dataset will be divided into batches of thread blocks, distributed onto GPU’s processing elements. Therefore, all these features make accurate performance prediction for GPU kernels possible and practical.

6.2 High-Level Framework

In this section, we discuss the overall flow of performance prediction and the metrics we use to determine program similarity. We also use microbenchmarks to study sensitivities of GPU performance to uncoalesed memory accesses and unfilled warps.

6.2.1 The Flow of Performance Prediction

Figure 6.1 illustrates our high-level prediction framework. This approach is in dual with the framework proposed by Hoste et al. [37] for CPU performance prediction. This work differs from their work by targeting manycore architectures and identifying a set of first-order program metrics for GPU applications.

As shown in the diagram, performance prediction is achieved through correlating the characteristics of the application of interest with those of existing benchmarks in the workload space,

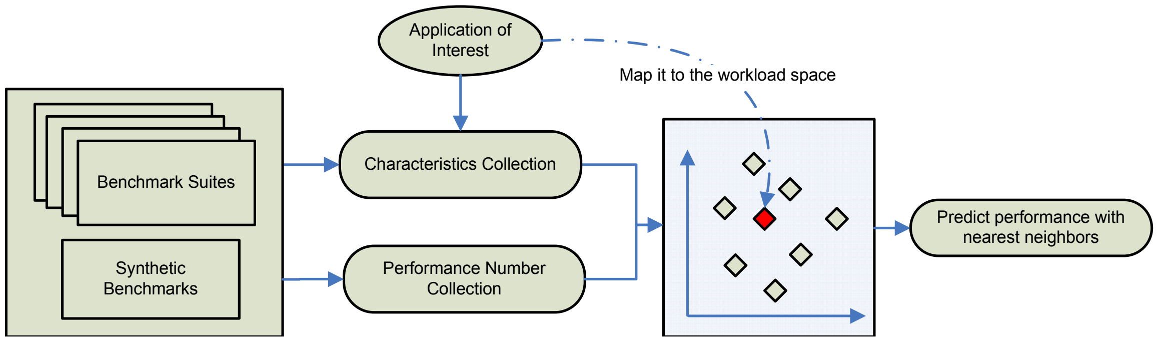
Application of Interest
Map it to the workload space
Benchmark Suites
Synthetic Benchmarks
Characteristics Collection
Performance Number Collection
Predict performance with nearest neighbors

Figure 6.1: Benchmark statistics are first collected in order to construct a workload space. An application of interest is profiled with exactly the same set of metrics and mapped into the workload space. Prediction is achieved with the nearest proxy benchmarks to the application of interest [37].

whose performance scores are known before prediction. Program statistics are collected in order to construct a workload space and calculate pair-wise similarity values across different applications. Given an application of interest, we profile it with exactly the same metrics, map the application into the workload space and use the similarity information to determine its nearest proxy benchmarks from the benchmark repository [65]. Finally, the performance of the target application is predicted by interpolating the performance of proxy benchmarks. For example, many computer system vendors report performance scores of their systems by instrumenting the SPEC benchmark suite [71]; these scores can be used for prediction and reference purposes for CPUs. The SPEC HPG group has been developing a standard benchmark suite for GPGPU [76], and we anticipate that vendors may use them similarly to report benchmark scores for their GPU platforms in near future.

Two issues are of particular importance to accurate prediction in our approach:

- How effectively can the chosen metrics capture the major behaviors of the GPU and represent similarity?
- How diverse are the benchmarks included in today's benchmark suites? How well do they cover the workload space?

The first issue is important for accurately determining if two applications are similar, while the second issue is important to ensure the existence of appropriate proxy benchmarks to the application of interest. The rest of this chapter endeavors to address these two problems.

6.2.2 Application Profiling and Metrics

To determine the similarity among different applications, we use a set of metrics which are fundamental to the GPU performance. Our choice of metrics are based on the observations of previous experiences of benchmarking GPUs [4,22,44]. These metrics have proved to be capable of capturing the program behaviors of both NVIDIA and AMD GPUs effectively. However, this does not preclude other important metrics, which can make the performance prediction more accurate. The following list illustrates the metrics used in this study. They represent degrees of compute intensity, memory locality, branch divergence, etc.

- **Instruction Throughput** This metric demonstrates the aggregate throughput of an application. We use instruction per cycle (IPC) in this work. Note that different applications can achieve a similar level of IPC but quite different scaling behaviors, which suggests additional metrics are needed to present other aspects of application behaviors.
- **Computation/Memory Access Ratio** A higher ratio implies that an application stresses more arithmetic units, while a low ratio suggests more stresses on memory interface. This metric is a widely-used factor to determine if an application is compute-bound or memory-bound.
- **Memory Instruction Mix** A GPU application may take advantage of multiple memory spaces on the GPU. Significant global memory accesses lead to poor performance and scalability. A high ratio of memory accesses to GPU caches (scratchpad, constant and texture) generally means a better usage of data locality, fewer off-chip accesses and better program scalability.
- **Memory Efficiency** One important performance optimization for GPUs is the coalescing of global memory accesses [27]. A high ratio of uncoalesced memory accesses suggests a waste of effective memory bandwidth and thus degrades the overall application performance.
- **Warp Occupancy** Warp occupancy captures the average number of active threads over all issued warps over the entire runtime of the benchmarks [4]. We classify this metric into four

buckets, namely $[1-8]$, $[9-16]$,$[17-24]$ and $[25-32]$. A higher warp occupancy means a better utilization of GPUs' computation resources.

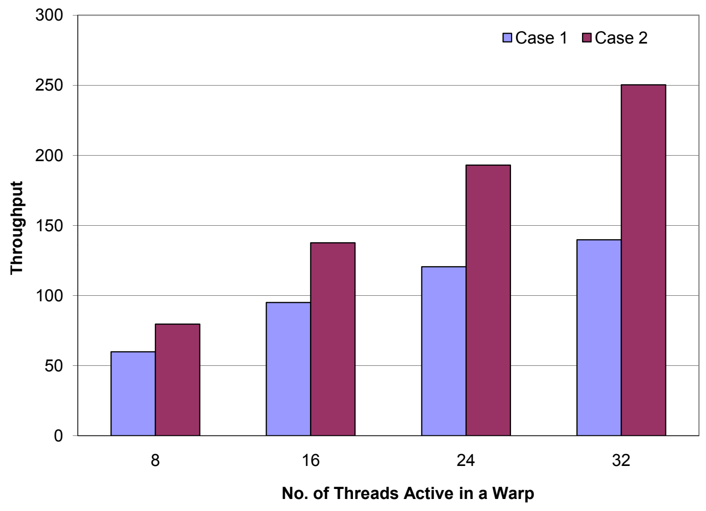

Figure 6.2: The y-axis represents throughput (instructions per cycle) while x-axis represents the number of threads active within a warp. Unfilled warps lead to lower instruction throughput.

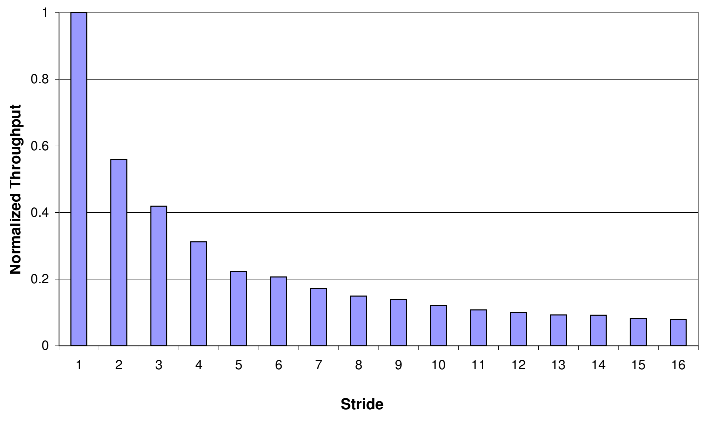

Figure 6.3: The y-axis represents normalized throughput to the stride-1 case. The total throughput degrades as the stride size increases. The results are reported with GPGPUsim.

As discussed in previous chapters, among these metrics, warp occupancy and memory coalescing are two distinctive features limiting GPU's throughput. Low warp occupancy means a majority of threads are inactive, leading to low instruction throughput. The reasons for unfilled warps can be due to different reasons, such as branch divergence and parallel reduction [17]. We measure a microbenchmark by controlling the number of active threads within a warp. Figure 6.2 shows instruction throughput we collect with the GPGPUsim simulator varying the number of active threads from 8 to 32. The total throughput increases linearly with increasing numbers of active threads.

We also measure the impact of uncoalesced memory accesses on total throughput. For instance, many applications present access patterns to array structures with non-unit strides, i.e. array[stride * tid + offset]. Such access patterns will lead to undesirable memory coalescing behaviors on the GPU, because adjacent threads are accessing non-contiguous data elements [27]. We use a microbenchmark to measure the variations of total throughput versus the changes of the stride size. Figure 6.3 shows the normalized throughput varying the size of stride from 1 to 16. The performance degrades significantly with increasing number of uncoalesced memory accesses. This behavior is also well documented in NVIDIA's best practice guide [26].

6.3 Methodology

In this section, we discuss our methodology, including the experiment environment, the workloads used in this work and how we calculate similarity among benchmarks and further use it for performance prediction.

6.3.1 Experiment Setup

To measure program characteristics, we use GPGPU-Sim [4] from the University of British Columbia. GPGPU-Sim provides a detailed simulation model of contemporary GPUs. We use GPGPUsim to report program statistics such as IPC, instruction mix, warp occupancy, uncoalesced memory accesses, etc., which will be used for subsequent performance predictions. We also use GPGPUsim to simulate the performance of GPU systems with 4, 8, 16, and 28 shader cores.

A GPU application usually consists of CPU parts, CPU-GPU PCI-E transfers and GPU kernels. We restrict our performance prediction work to GPU kernels and leave CPU execution time as a constant variable. In addition, time spent on PCI-E transfers is proportional to the data size transferred across the CPU and the GPU. Therefore, the transfer time can be easily predicted by dividing data size with the PCI-E bandwidth.

Table 6.1: Rodinia, GPGPUsim and NVIDIA CUDA SDK applications

Application	Abbrev	Suite	Input Size
Back Propagation	BP	Rodinia	65536 input nodes
CFD Solver	CFD	Rodinia	97k elements
Heart Wall Tracking	HW	Rodinia	609×590 pixels/frame
HotSpot	HS	Rodinia	500×500 grid
LU Decomposition	LUD	Rodinia	256×256 matrix
Needleman-Wunsch	NW	Rodinia	2048×2048 data points
SRAD	SRAD	Rodinia	512×512 data points
Stream Cluster	SC	Rodinia	2048×2048 data points
LIBOR Monte Carlo	LIB	GPGPUsim	4096 paths, 15 options
Neural Network	NN	GPGPUsim	28 digits
NQueen Solver	NQU	GPGPUsim	10×10 grid
Ray Tracer	RAY	GPGPUsim	256×256 image
Weather Prediction	WP	GPGPUsim	10 timesteps
BlackScholes	BLK	NVIDIA SDK	4M options
DXTC	DXTC	NVIDIA SDK	512×512 image
Matrix Multiply	MM	NVIDIA SDK	80×48, 48×128
FastWalshTransform	FWT	NVIDIA SDK	32 k data points
MersenneTwister	MT	NVIDIA SDK	24000000 samples

6.3.2 Workloads

Accurate performance prediction requires the workload repository to include a sufficient number of benchmarks to construct a training set. These benchmarks also need to be diverse in application characteristics and ideally distributed evenly in the workload space. We use real workloads with diverse characteristics for training. For this study, we first evaluate the effectiveness of using only Rodinia [14] for performance prediction, and then we will take other workloads into account, which provides a richer workload space.

The input sizes we choose for all the benchmarks can make full use of the GPU. We also restrict our study to applications which do not take advantage of GPU's texture memory space. CUDA and OpenCL allows programmers to bind big read-only data structures to texture memory, which is cached and optimized for arbitrary memory access patterns. Texture units present a unique access pattern, which we leave for future work.

6.3.3 Similarity and Proxy Benchmarks

We define similarity between two benchmarks as their mutual euclidean distance in the n-dimensional workload space. Each benchmark is represented by a data vector with multiple dimensions; each dimension of the workload space represents one characteristic. We calculate pairwise distances for all the benchmarks. For each benchmark, we search the entire workload space to find k $(k \geq 1)$ closest benchmarks for the interpolation purpose —we use the *k-nearest* neighbor algorithm for searching [65]. K benchmarks will be used as the proxy benchmarks for the application of interest. The values of different metrics vary in the range. Therefore, when calculating the Euclidean distances among benchmarks, normalization is first applied to raw data for all characteristics across all benchmarks.

The choice of the value k is also important to prediction accuracy, which we will discuss in details in section 6.4.4. We use MATLAB [79] to process data for collected characteristics, calculate similarities among benchmarks (i.e. *pdist*) and search nearest neighbors for individual benchmarks.

6.3.4 Scaling Prediction

$$Weight(i) = \frac{\frac{1}{dist(i)}}{\sum_{i=1}^{n} \frac{1}{dist(i)}} \tag{6.1}$$

$$SpeedupPred = \sum_{i=1}^{n} Weight(i) * Speedup_i \tag{6.2}$$

$$Error = \frac{|SpeedupPred - SpeedupReal|}{SpeedupReal} * 100\% \tag{6.3}$$

After we determine the most similar benchmarks to an application of interest, we predict its relative performance: the speedup of running an application on one platform against another platform. The predicted speedup is calculated with the above equations (1, 2). The predicted speedup of the application of interest can be represented by a linear combination of its nearest neighbors, each contributing a component to the overall predicted value. The weights of the neighboring applications are assigned to be inversely proportional to the distances to the application of interest [37]. This means that the more similar a benchmark is to the application of interest, the more weight should it be given for prediction. If we can obtain the run times on one platform, we can also predict the run times on another platform by multiplying the speedups. To evaluate how accurate the prediction is, we compare the predicted speedup with the real speedup achieved. The absolute prediction error can be calculated with equation (3). In addition, we choose the arithmetic mean of absolute prediction errors because we are reporting results that are samples from the overall population of application behaviors.

6.4 Performance Prediction and Results

In this section, we report performance prediction results with different number of benchmarks and across multiple processor configurations and multiple application inputs. We also study the impact of the number of proxy benchmarks and verify statistical significance of prediction accuracy.

6.4.1 Performance Prediction with Existing GPGPU Benchmarks

In real-world practice, users may rely on standard benchmarks with widely-accessible information and performance scores. There are several benchmark suites released for GPGPU [4, 14, 20, 75]. However, how representatively they can be used as references has not been sufficiently understood by prior works.

In the first experiment, we consider eight benchmarks from the Rodinia benchmark suite. To predict the performance of a particular application, the rest of applications are deemed as the training set. We use a "leave-one-out" approach for each application [37, 65], and calculate its pair-wise distances with the rest of Rodinia applications. We subsequently calculate the nearest neighbors of the target application and predict its scaling behavior with determined proxies. Figure 6.4 shows the predicted and actual speedups for eight Rodinia applications, which tends to be inaccurate. This is attributed to the small application space covered by these benchmarks both in number and feature. In addition, the experiment result suggests that in order to evaluate the prediction framework, we should consider more benchmarks enriching the application coverage.

In the second experiment, we take some other applications from the GPGPUsim suite and NVIDIA's CUDA SDK into consideration. We report the results for the benchmarks we are able to simulate successfully and exclude those making use of the texture memory. We also did not consider simple kernels such as vector add and matrix transpose from the CUDA SDK. Figure 6.5

shows the prediction results comparing the predicted speedups versus the actual speedups. The prediction accuracy is significantly improved by including new applications. The results show an average of 21.9% absolute errors. In particular, *NQU* (50.3%) and *LUD* (133.0%) are poorly predicted. The following PCA and workload space study also shows that they are two of the outlier applications.

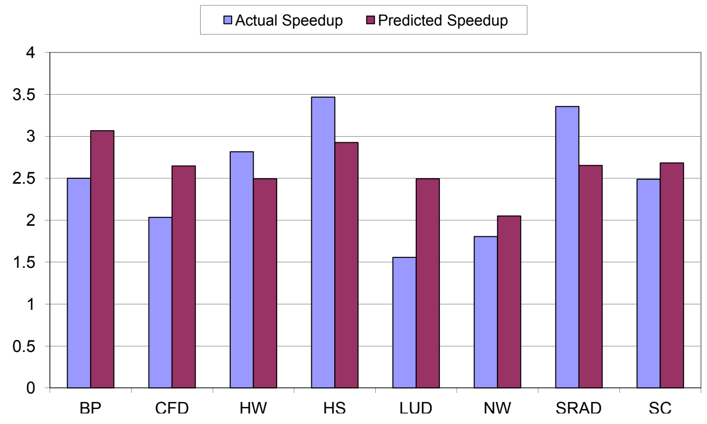

Figure 6.4: The predicted speedups (28 shaders v.s. 8 shaders) and measured speedups. We use only Rodinia benchmarks in this experiment.

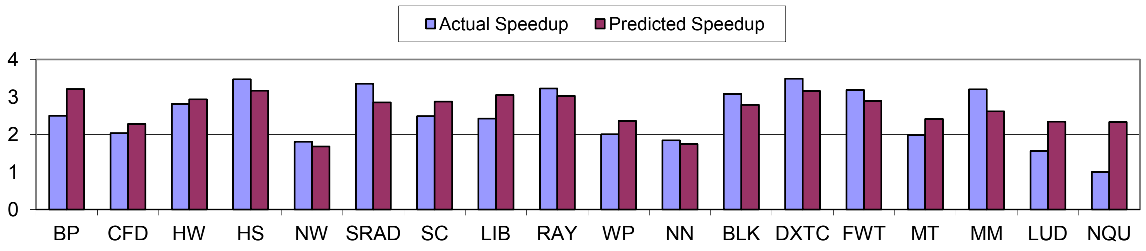

Figure 6.5: The predicted speedups (28 shaders v.s. 8 shaders) v.s. measured speedups for all the applications. Prediction is conducted with real workloads from Rodinia, GPGPUsim and NVIDIA CUDA SDK.

To better illustrate similarity/dissimilarity among benchmarks, we conduct a principal component analysis (PCA) across all benchmarks for all characteristics. PCA transforms a number of possibly correlated variables (or dimensions) into a set of uncorrelated variables, called principal components [42]. PCA has the ability to describe a big data set along a limited number of dimensions while still capturing the essence of the entire data set. We plot all the benchmarks in the 3-D PCA space representing a 78% of total variance. It is interesting to notice that some outlier benchmarks (e.g. *NN* and *NW*) in the workload space are not necessarily badly predicted. Similar to their scaling behaviors, the nearest neighbors determined by the framework (See Table 6.3) also exhibit relatively poor scalings, making prediction relatively accurate. From our experiment, the worst predicted benchmarks (*LUD* and *NQU*), though also belonging to the category of outlier benchmarks, are located at the "transition" region in the workload space; the nearest neighbors to *LUD* and *NQU* both include benchmarks which scale well (i.e. over prediction).

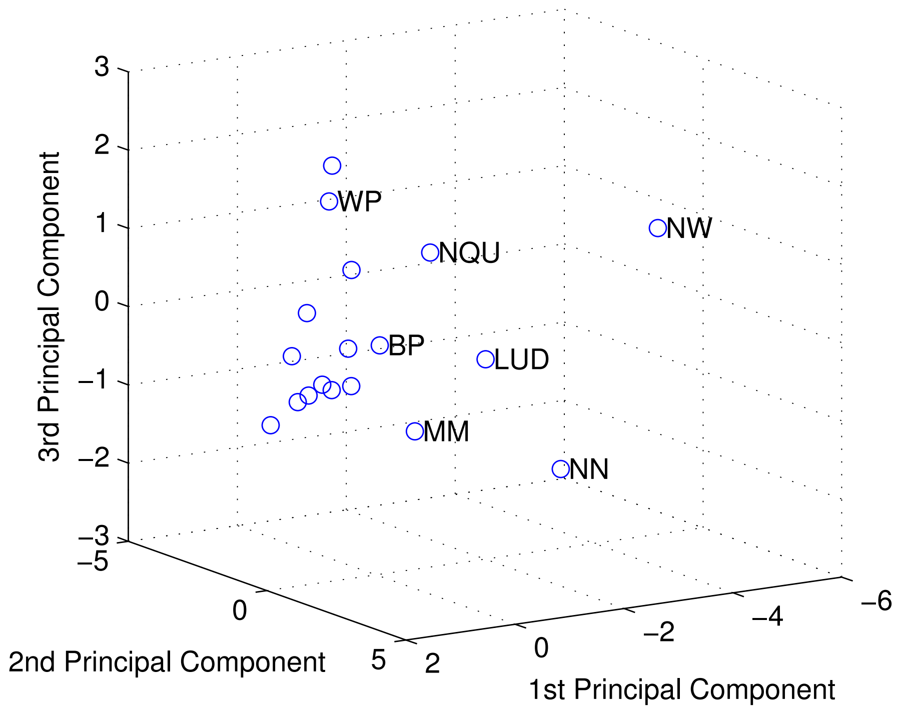

Figure 6.6: Applications in the 3-D PCA space

6.4.2 Spearman's Ranking

Using the metric of absolute error alone is not sufficient to justify accurate prediction; the evaluation of correlation between predicted and real values is needed. We use Spearman's Rank Correlation [37], a non-parametric statistical measure of mutual relationship and dependence between two variables. What's more, it does not require that samples meet any specific distribution requirements, e.g. normal distribution. Figure 6.7 illustrates 2-D views with the x-axis representing the predicted speedup while the y-axis represents the actual speedup. The predicted speedups show a strong correlation with the actual speedups across all applications. Ideally, all the points should be on the $y = x$ line (i.e. perfect prediction). For instance, to evaluate the significance of prediction for the experiment shown in Figure 6.5, we rank all the predicted and actual speedups and calculate Spearman's Rank Correlation Coefficient, which results in a high ρ value of 0.738. And then we determine the "critical value". We use the common significance level of $\alpha = .05$. The corresponding significance value is 0.429 (0.582 when $\alpha = .01$). Our calculated correlation efficient, 0.738, is much higher than this value, which means that our approach is reasonably accurate.

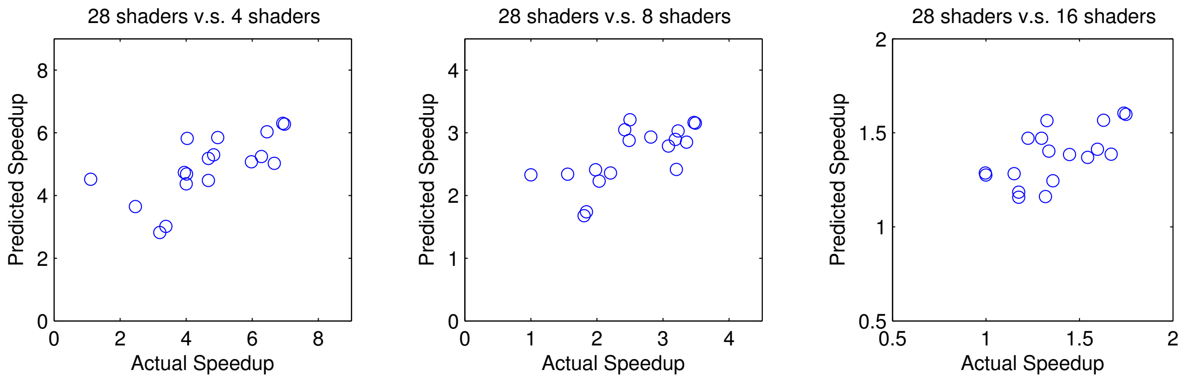

Figure 6.7: Actual speedups (x-axis) and predicted speedups (y-axis). Each dot in the graph represents one benchmark.

6.4.3 Prediction with Different GPU Configurations

We also predict the scalings of applications under different GPU configurations. Table 6.2 shows the prediction results for the speedups of a 28-shader GPU against GPUs with 4, 8 and 16 shaders. The arithmetic mean of the absolute prediction errors ranges from 15.8% to 27.3% and the ρ values range from 0.620 to 0.774. In particular, the four-shader case (27%) shows relatively higher average prediction errors. The 16-shader case shows a relatively lower average absolute error as well as a lower ρ. Figure 6.7 shows the prediction results for three GPU configurations. Each dot in the graph represents one benchmark.

GPUs	4 shaders	8 shaders	16 shaders
Arith. Mean Pred. Errors	27.3%	21.9%	15.8%
Significance: ρ	0.774	0.738	0.620

Table 6.2: Prediction errors and Spearman ranking results over different GPU configurations

6.4.4 Number of Proxies

The number of proxies is an important parameter critical to accurate prediction. We compute the speedup for the application of interest using a number of proxies. The actual proxies, as discussed in Section 6.3.3, are determined by the k-nearest neighbors algorithm. It is not feasible for the sample space to include all the points (i.e. essentially all the theoretical applications), therefore our assumption of the framework is that we will need more than one nearest neighbor ($k > 1$) for interpolation purpose. On the other hand, if k is too large, it will adversely degrade the accuracy of prediction; using a large k risks that some applications dissimilar to the application of interest may be treated as proxies.

Figure 6.8 quantifies the sum prediction error as a function of number of proxies. The prediction error improves when increasing the number of proxies and then starts to degrade with larger numbers, which proves our hypothesis. The best predictions happen when we use three proxies in our experiment.

6.4.5 Different Input Sizes

In this section, we predict performance using different application input sizes. The GPU schedules thread execution by dispatching thread blocks on multiple streaming multiprocessors. As discussed in Section 6.1.2, when the number of thread blocks are sufficiently large to fill the GPU, further increasing the input size will not influence the scaling behavior much (i.e. approximately linear relationship between run time and input). We show two applications, *SRAD* and *Needleman Wunsch*, as examples and vary their input sizes from 512×512 to 2048×2048 and from 1024×1024 to 2048×2048 respectively. We measure the execution time in cycles on a 8-shader GPU and calculate the execution time of different inputs on a 28-shader GPU with the predicted speedup. Figure 6.9 shows the results comparing the actual cycles versus the predicted cycles. SRAD achieves an av-

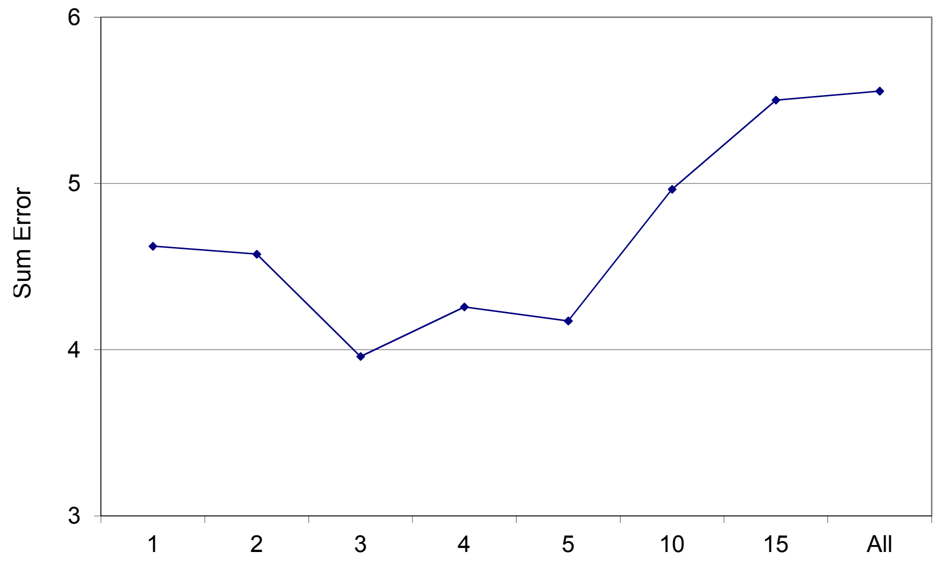

Figure 6.8: The changes of prediction errors while increasing the number of nearest neighbors (i.e. proxies).

erage of 19.3% absolute prediction error while *Needleman Wusnch* achieves an average of 11.6% error. Of course, this is only restricted to those applications whose behaviors are not data dependent.

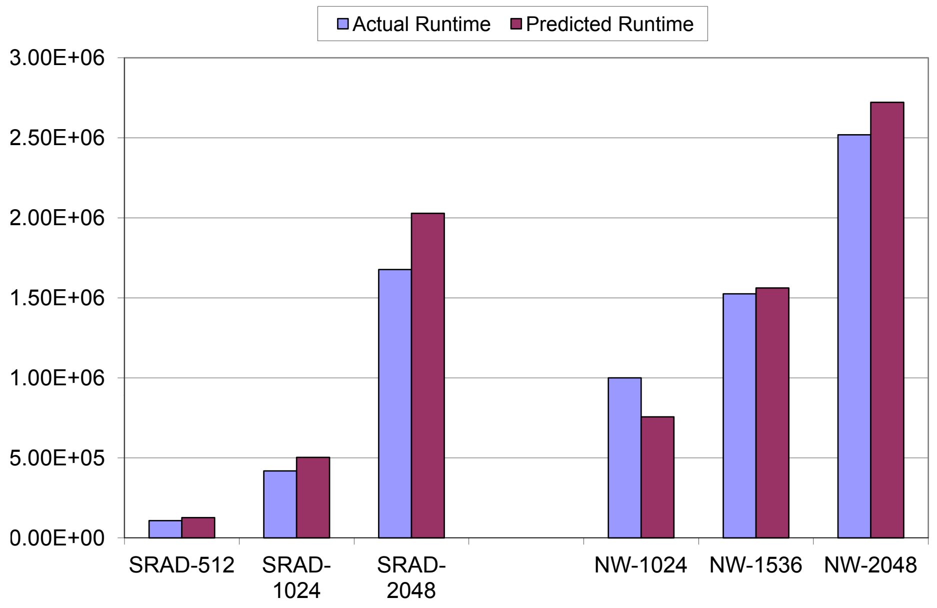

Figure 6.9: The actual and predicted execution times (in cycles) of *SRAD* and *Needleman Wunsch* with different input sizes.

Table 6.3: The three proxies and their weights for each benchmark

	first proxy		**second proxy**		**third proxy**	
	benchmark	weight	benchmark	weight	benchmark	weight
Needleman Wunsch	LU Decomposition	0.3423	N-Queen Solver	0.3330	Backpropagtion	0.3247
HotSpot	DXTC	0.4522	Backpropagation	0.2925	SRAD	0.2553
Backpropagation	SRAD	0.3755	Heartwall	0.3417	DXTC	0.2828
SRAD	Heartwall	0.4004	fastWalshTransform	0.3572	Monte Carlo	0.2424
Streamcluster	Heartwall	0.3477	SRAD	0.3429	Monte Carlo	0.3094
CFD Solver	MersenneTwister	0.4733	Monte Carlo	0.2735	StreamCluster	0.2532
LU Decomposition	Matrix Multiply	0.3916	StreamCluster	0.3208	NQueen Solver	0.2876
Heartwall	Monte Carlo	0.4023	SRAD	0.3317	fastWalshTransform	0.2660
Monte Carlo	Heartwall	0.4769	fastWalshTransform	0.2849	SRAD	0.2382
Neuro Network	NQueen Solver	0.3549	Needleman Wunsch	0.3265	Backpropagation	0.3186
Ray Tracer	HotSpot	0.3921	CFD Solver	0.3096	DXTC	0.2983
Weather Prediction	CFD	0.3682	MersenneTwister	0.3440	Ray Tracer	0.2878
BlackScholes	Monte Carlo	0.3757	fastWalshTransform	0.3326	Heartwall	0.2917
DXTC	HotSpot	0.3794	SRAD	0.3362	Backpropagation	0.2844
NQueen Solver	CFD Solver	0.3574	Streamcluster	0.3311	Backpropagation	0.3115
fastWalshTransform	SRAD	0.3689	Heartwall	0.3315	Monte Carlo	0.2996
MersenneTwister	CFD Solver	0.4790	BlackScholes	0.2661	Monte Carlo	0.2549
Matrix Multiply	LU Decomposition	0.3915	Streamcluster	0.3147	DXTC	0.2938

6.4.6 Discussion

Our analysis show that one challenge of this performance prediction approach is its difficulty to predict an application of interest that is isolated in the workload space (as discussed in Section 6.4.1). The same observation has been made by Hoste et al. [37]. For these applications, their proxy benchmarks determined by the framework are relatively farther away from the application of interest than those applications in “richer” areas of workload space, which means the benchmarks might not accurately represent the application behavior of the target application. We have shown that predictions can be poor with inappropriately determined nearest neighbors. Therefore, constructing a comprehensive benchmark repository for training is critical to accurate prediction.

This in turn raises an important question of benchmark suite design in general: what programs we should select for inclusion in a standard benchmark suite [37] and how many benchmarks are sufficient in terms of both feature coverage and cost. Several prior works have examined similar issues for today’s research workloads. Phansalkar et al. [66] make the argument that the SPEC CPU benchmark suite only covers a subset of application behaviors and its workloads exhibit redundant behaviors. Hoste et al. [36] use microarchitecture independent characteristics and PCA to characterize single-threaded workloads. Heirman et al. [32] use a cycle stack approach to compare multithreaded CPU workloads including SPLASH2 [82], Parsec [9] and Rodinia. Goswami et al. [22] and Che et al. [17] analyze GPU workloads and their coverage. Their findings are all in agreement with our observations: a thorough examination for benchmark construction requires a comprehensive evaluation and comparison of all the current benchmark suites to establish a single set of workloads with sufficient coverage and little redundancy. The benefit of this framework is that it can be used as a method to test how well the benchmark suite is constructed by following the prediction process discussed in this work. The size and workload coverage of a suite can be determined by trading off between prediction accuracy and instrumentation cost. Another interesting open research question is whether some real-world applications indeed exist to cover those "desert" areas of application space. This issue might be partially resolved by developing synthetic benchmarks to mimic diverse application behaviors.

6.5 Other Potential Metrics

Our work has shown promising trends of predicting GPUs' application performance with existing benchmarks. We hope to address some limitations in future work. In this work, we only consider a few first-order characteristics which impact GPU performance. Other factors might make performance prediction more accurate. We plan to study metrics capturing memory access efficiency of other GPU caches (e.g. texture and Fermi's hardware caches). We also plan to consider synchronization overhead within each thread block (e.g. syncthread()) and across thread blocks (i.e. global synchronization). On the other hand, in certain cases, some of these factors might not influence prediction results a lot (e.g. first-order metrics). For instance, when *syncthread()*s are used at the points of exchanging data between global memory and shared memory, interleaved warp executions may hide most synchronization overheads. Global synchronization overhead is caused by different finishing times of thread blocks dispatched to multiple SMs. If there are a sufficiently large number of thread blocks evenly distributed to SMs, synchronization overhead is negligible – the execution time of one or few thread blocks. Finally, part of the program characteristics can be directly collected through GPU hardware counters; we leave instrumentation and analysis of GPU hardware for future work.

6.6 Related Work

Gustafson and Todi [29] performed an early work in correlating the performance of benchmarks with others. Hoste et al. conducted a benchmark suite coverage study [42] and they used standardized benchmarks to conduct performance prediction [37]. The framework they proposed for prediction is based on principal component analysis (PCA) and genetic algorithms. Snavely et al. [70] studied an approach to predict parallel application performance on HPC systems. They collected machine profiles and application signatures, and combined them for prediction with a convolution method. Carrington et. al. [12] predicted application performance using single, simple synthetic metric (i.e. compute kernel) and a linear combination of these simple metrics as well. The individual "metrics" are namely small synthetic benchmarks such as LINPACK [54], STREAM [73] and HPC Challenge Benchmarks [6], etc. All of these prior works mainly focused on CPU workloads.

Hong et al. [35] proposed an analytical model that estimates the execution time of massively parallel programs on GPUs. Similarly, Baghsorkhi et al. [3] proposed a performance model capturing performance effects of major GPU microarchitecture features using an approach based on the program dependence graph (PDG). Kerr et al. evaluated a set of metrics for GPU workloads [44] and used them to analyze the behavior of GPU programs. They further used PCA and regression modeling to predict GPU performance [45].

Meng et al. [57] proposed GROPHECY, a GPU performance projection framework that can estimate the performance benefit of GPU acceleration without actual GPU programming or hardware. Users only need to skeletonize pieces of CPU code as targets for GPU acceleration, which are further transformed in various ways to tune GPU code in the optimization space. The code characteristics are further used by an analytical model to project GPU performance. Zhang et al. [86] developed a microbenchmark-based performance model which allows programmers and architects to identify GPU program bottlenecks and predict the benefits of potential program optimizations and architectural improvements. Different from these works, our approach predicts GPUs' application performance by taking advantage of the characteristics and performance numbers of existing benchmarks.

6.7 Conclusions and Future Work

In this chapter, we study an approach of using existing benchmarks to predict performance of arbitrary GPU applications. For instance, users can take advantage of performance scores of a variety of standard GPU benchmarks provided by GPU vendors (e.g. CPU vendors report performance scores for standardized CPU benchmarks such as SPEC [71]). Given a target application, prediction is conducted by collecting a set of important GPU characteristics for all the benchmarks in the repository, identifying the most similar proxy benchmarks in the workload space, and using the performance of proxy benchmarks to predict that of the target application. We predict performance speedups of various applications across different GPU configurations. The predicted value for a particular benchmark is represented with a weighted sum of the speedups of its proxy benchmarks. We allocate the contribution of each benchmark to be inversely proportional to its distance to the target application. We consider workloads from Rodinia, GPGPUsim and NVIDIA CUDA SDK to construct a richer benchmark space. The experimental results show that accurate performance prediction is possible for the proposed metrics and the methodology based on nearest neighbors. We are able to achieve an arithmetic mean of 21.6% prediction error across different GPU configurations. Much of the error is due to outlier applications in the workload space. In addition, the predicted performance shows a strong correlation with the actual performance from the Spearman Ranking analysis.

This approach requires a well-constructed GPU benchmark suite with sufficient diversity and feature coverage, which requires a holistic approach for benchmark suite construction. Furthermore, studying mutual relationships among benchmarks allows users to focus on understanding and analyzing the most important and relevant proxy benchmarks, which helps them make appropriate design and purchasing decisions. In addition, the entire flow of our framework, including program profiling, data collection and processing (e.g. pair-wise distance and nearest neighbors) and performance prediction, can be automated as an integrated tool for easy use. Understanding the robustness of our prediction approach with different programming styles and architectures is an open research question, which we leave for future work.

Chapter 7

Conclusion

7.1 Dissertation Summary

Heterogeneous computing with accelerators (e.g. GPUs) opens up new challenges and opportunities for application performance improvement. These accelerators are significantly different from CPUs in architecture and programming models, which requires researchers to leverage their unique features to achieve diverse computation needs.

How to program and adapt various applications to run efficiently on a heterogeneous platform with these emerging devices has not been sufficiently studied by prior works. First, understanding and performance engineering, e.g. benchmarking and performance modeling, for parallel programs are essential to achieve optimization goals. Secondly, high performance of a heterogeneous system requires per-device optimizations as well as efficient cross-platform collaborative processing. Particularly, challenges include mapping tasks to their most appropriate devices and spreading workloads proportionally to individual devices with regard to their computation rates. Additionally, CPU and GPU may prefer different memory layouts, which raises the important performance portability issue.

To address these concerns, this dissertation endeavors to understand and optimize the performance of heterogeneous system using GPU as a case study. However, major principles in program analysis and performance optimizations can be adapted to other platforms as well. A summary of this dissertation and its contributions are as follows:

- To address the research challenges in heterogeneous computing, researchers first need a set of well-designed workloads for their research. We developed Rodinia, a benchmark suite for heterogeneous computing. The Rodinia benchmark suite is designed to target multiple platforms, e.g. multicore CPUs and GPUs, and takes advantage of different languages - OpenMP, CUDA and OpenCL. We also did a preliminary study comparing GPU and FPGA efficiency for diverse application characteristics [16]. Rodinia includes the state-of-art implementations of 17 applications from different application domains and leverages various optimizations techniques from our research. It spans a diverse range of parallelism and compute patterns and stresses different hardware components. The initial version of Rodinia was part of my masters thesis work. My Ph.D. work extended it with diverse applications [17] as well as support for OpenCL.

- Part of the Rodinia research is to study formal analysis approaches and methodologies for workload characterization and construction of parallel benchmark suite with sufficient diversity. In addition, as Rodinia sees higher levels of acceptance, it becomes important that researchers understand this new set of benchmarks, especially how they differ from previous works. We characterize the Rodinia benchmarks and show the diverse behaviors exhibited (e.g. IPC, memory instruction mix, warp occupancy, etc). The analysis also include analyzing performance results on an NVIDIA GeForce GTX480 with configurable first-level caches and characteristics for different incremental optimized versions. Furthermore, we also compare and contrast Rodinia with Parsec to gain insights into the similarities and differences of the two benchmark collections; we apply principal component analysis to analyze the application space coverage of the two suites. Our analysis shows that many of the workloads in Rodinia and Parsec are complementary. We concluded that benchmark construction should consider various workloads from all suites and create a final set of benchmarks with sufficient feature coverage and minimum redundancy.

- To take advantage of all the available resources in a heterogeneous system, we study an approach to execute parallel compute kernels simultaneously on the CPU and the GPU and allow efficient data partitioning and load balancing across devices. There has been growing research efforts to investigate this issue for heterogeneous platforms. However, most previous works solve this problem manually or their approaches are not adaptable to run-time resource changes. To address these concerns, our framework adopts a method of hierarchical domain partitioning with chunking and work queues. The computation domain is divided into chunks with hierarchical domain partitioning. The scheduler is responsible for dispatching chunks on the CPU and the GPU by checking the progress of each compute device at runtime. Our solution has the advantage of adapting to runtime resource changes and reaching a desirable balanced workload ratio. The performance improvement of CPU-GPU load balancing over GPU-only execution varies across applications, e.g. SRAD (11.7%) and Kmeans (29%), depending on their CPU-GPU speed gaps.

- GPUs offer a large number of parallel cores and have access to high memory bandwidth; however, data structure layouts in GPU memory often lead to sub-optimal performance for programs designed with a CPU memory interface. This implies that application performance is highly sensitive irregularity in memory access patterns. This issue is all the more important due to the growing disparity between core and DRAM clocks; memory interfaces have increasingly become bottlenecks in computer systems. To address these concerns, we proposed a high-level abstraction to define memory mappings and access patterns for heterogeneous computing. We developed a framework, Dymaxion, and associated techniques, to allow programmers to easily optimize the access patterns for better memory bandwidth utilizations in a heterogeneous system with CPUs and GPUs. Memory layout reorganizations are abstracted in a high-level API extending the CUDA programming model. The runtime system is responsible for optimizing memory layouts accordingly and then transforming subsequent memory access addresses as necessary. To minimize the layout transformation overhead across devices, we hid the latency through chunking data structures and overlapping their data transfers with layout reorganizations. With Dymaxion, four applications with a variety of access patterns achieve an average of $3.3\times$ speedup on GPU kernel computations and 20% performance improvement when taking PCI-E transfers and layout reorganization into account. Furthermore, the codes with Dymaxion maintain portability across CPUs and GPUs.

- GPUs present significantly different architectures from CPUs and require specific mappings and optimizations to achieve high performance. This makes GPU workloads demonstrate application characteristics different from those of CPU workloads. Therefore, it is critical for researchers to understand the first-order metrics that influence GPU performance and scalability most. Furthermore, methodologies and associated tools are needed to analyze and predict the performance of GPU applications and will help users to make proper purchase decisions.

  In this work, we study an approach of predicting the performance of GPU applications by correlating them to existing workloads. One tenet of benchmark design, also a motivation of this paper, is that users should be given capabilities of leveraging standard workloads to infer the performance of applications of their interest – an important issue neglected by most of today's benchmark suite designers. We first identify a set of important GPU application characteristics and then use them to predict performance by determining the most similar proxy benchmarks to the target application. We demonstrate the prediction methodology and conduct prediction with benchmarks from different suites to achieve a better workload coverage. The experiment results show that we are able to achieve satisfactory performance predictions except for certain outlier applications. The arithmetic mean of prediction error is 21.6% across different GPU configurations. Finally, we discuss several important considerations for systematically constructing future benchmark suites in general.

7.2 Future Research Challenges and Directions

One challenge in mapping traditional CPU applications efficiently onto the GPU is that traditional data structures, e.g. list, tree and graph, do not directly fit to the GPU. For many data structures, e.g. stack and queue, the operations are inherently sequential, which leads to significant efforts in rewriting the applications and redesigning the algorithms to achieve parallelism. To tackle these challenges, researchers need to revisit various traditional CPU applications, especially those with complicated and irregular data structures, and explore opportunities and techniques to perform easy mappings to GPU. Interesting research directions include studying alternative parallel solutions to operations on these data structures, discovering efficient parallel data structures and algorithms for common problems, and further developing transformation routines and software building blocks to ease programming [59].

In this dissertation, applications require the programmers to decide mappings of tasks to different platforms: which regions of code to run on the CPU and which to run on the GPU and to actually write the CUDA or OpenCL kernels for offloading to the GPU. On the other hand, as a future research direction, existing legacy codes can be taken as inputs for compiler analysis to generate task partitionings across platforms. This requires profiling the types of heterogeneous parallelism inherent in a variety of applications and possibly performing affinity analysis on different code types and their favorable architectures. Other important factors that need taking into account also include input data sizes and communication overheads (e.g. PCI-E transfers) which are important for making correct offloading decisions. Some related works in this regard include PGI Accelerator [24], OpenACC [63], OpenMPC [47] programming models and the work by Szafaryn et al. [46], which all require programmers to make offload decisions by adding pragmas to particular regions of codes by specifying parallelism and data mappings.

Furthermore, a comprehensive analysis on the sources of poor cross-platform performance portability is also needed. Our Dymaxion work tackles this issue from the memory layout perspective. An API and automatic layout and index transformation are applied to applications to achieve optimized memory accesses for CPU and GPU, when they prefer different layouts. In addition, certain optimization techniques appropriate for the CPU may yield bad performance on the GPU, or vice versa, which suggests each device has its own strategies for optimizations. Thus, a single code implementation (e.g. OpenCL) cannot guarantee satisfactory performance across all the platforms. Researchers need to evaluate the performance portability issue from other perspectives, such as data structures and algorithms, compiler techniques and different multithreaded models, which is an interesting future research.

In this work, we use systems with multicore CPUs and discrete GPUs as case studies. However, some techniques can apply to other heterogeneous organizations (e.g. AMD Fusion) as well. For instance, our load balancing approach can be used in platforms with heterogeneous devices in general. The approach based on chunking and work queues is able to respond to diverse computation rates of different devices to achieve load balancing. In addition, for dymaxion, integrating programmability into memory management units may enable Dymaxion-like transformation across different memory hierarchies. Dymaxion-like memory remapping can be enabled in memory/cache controllers, with data prefetching mechanisms for latency hiding.

My Ph.D. research has focused on understanding the unique features of emerging devices, especially GPUs, exploring their optimization strategies and leveraging heterogeneity for cross-platform performance improvement. In addition, our study was restricted to a single system node, however, techniques and principles discussed in this dissertation can be adapted to larger scale systems (e.g. heterogeneous compute clusters) as well:

- Future work can extend the Rodinia benchmark suite to support heterogeneous compute clusters. Interesting research issues include explorations of additional node-level parallelism, per-node data locality, cross-node load balancing and approaches to minimize cross-node data communications.

- Supporting multiple compute nodes implies new requirements and metrics for workload characterization. Especially, characterizing runtime data flows (between CPU and GPU within one node and among different nodes), dynamically changing resource demands, causes for load imbalance as well as performance bottlenecks will be critical to optimize overall system performances.

- Studying how to program heterogeneous clusters is also an interesting question. To efficiently leverage such systems, programmers will need to reason about multi-level parallelism, data locality and task mappings. Our load balancing framework uses CUDA/OpenCL-like hierarchical domain partitioings. A similar idea can be extended to support multiple nodes by introducing another layer of hierarchy with efficient work scheduling and load balancing support.

- The Dymaxion approach is not limited to one compute node. For example, MPI can launch memory remappings on each node in a cluster. Also, a global, cross-cluster approach may allow further optimizations, especially when cross-node data transfers and system-level interconnect are considered.

In summary, to understand and optimize the performance of heterogeneous systems, we explore the following research problems in this dissertation:

- We develop the Rodinia benchmark suite to fill the gap of benchmarking heterogeneous platforms including both the CPU and the GPU.
- We characterize the Rodinia benchmarks and show the diverse behaviors exhibited. We compare Rodinia with other benchmark suites (e.g. Parsec) and evaluate how Rodinia differ from the suites designed for multicore CPUs.
- We study an approach to allow applications to run on both the CPU and the GPU simultaneously and study the load-balancing issue.
- We design a set of APIs, layout/index transformation and latency hiding techniques to optimize memory access patterns for heterogeneous systems.
- We design and evaluate the methodology of predicting the performance of a GPU application by correlating to that of existing benchmarks.

Bibliography

[1] AMD Fusion APU. Web resource. `fusion.amd.com/`.

[2] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jan 2010.

[4] A. Bakhoda, G. L. Yuan, W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009.

[5] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec 2008.

[6] HPC Challenge Benchmarks. Web resource. `http://icl.cs.utk.edu/hpcc/`.

[7] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct 2009.

[8] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Sep 2008.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct 2008.

[10] M. Boyer, S. Che, K. Skadron, and N. Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of TECHCON*, Sept 2012.

[11] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating Leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, May 2009.

[12] L. Carrington, M. Laurenzano, A. Snavely, R. Campbell, and L. Davis. How well can simple metrics represent the performance of HPC applications? In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Nov 2005.

[13] S. Che. Benchmarking GPUs for general purpose applications. In *MS thesis, Univ. of Virginia School of Engineering and Applied Science*, Dec 2009.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct 2009.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[16] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute intensive applications with GPUs and FPGAs. In *Proceedings of the 6th IEEE Symposium on Application Specific Processors*, June 2008.

[17] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Dec 2010.

[18] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011.

[19] A. Corrigan, F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference*, June 2009.

[20] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable HeterOgeneous computing (SHOC) benchmark suite. In *Proceedings of Third Workshop on General-Purpose Computation on Graphics Processing Units*, Mar 2010.

[21] Neural Networks for Face Recognition. Web resource. `http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html`.

[22] N. Goswami, R. Shankar, M. Joshi, and T. Li. Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implication. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Dec 2010.

[23] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011.

[24] Portland Group. PGI Fortran and C Accelerator programming model. `http://www.pgroup.com/lit/pgi_whitepaper_accpre.pdf`.

[25] CUDA C Programming Best Practices Guide. Web resource. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf`.

[26] NVIDIA CUDA C Best Practice Guide. Web resource. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf`.

[27] NVIDIA CUDA Programming Guide. Web resource. `http://developer.nvidia.com/object/gpucomputing.html`.

[28] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2010.

[29] J. Gustafson and R. Todi. Conventional benchmarks as a sample of the performance spectrum. In *Proceedings of the Thirty-first Hawaii International Conference on System Sciences*, Jan 1998.

[30] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 2007 International Conference on High Performance Computing*, Dec 2007.

[31] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2007.

[32] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Nov 2011.

[33] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[34] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2010.

[35] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 152–163, Jun 2009.

[36] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[37] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2006.

[38] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2006.

[39] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on VLSI Systems*, 14(5):501–513, 2006.

[40] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006.

[41] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22:105–118, 2010.

[42] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.

[43] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev, 49(4/5):589-604*, 2005.

[44] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of PTX kernels. In *Proceedings of the 2009 International Symposium on Workload Characterization*, Oct 2009.

[45] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU workloads and systems. In *Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units*, April 2010.

[46] B. R. de Supinski L. G. Szafaryn, T. Gamblin and K. Skadron. Experiences with achieving portability across heterogeneous architectures. In *Proceedings of the Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, May 2011.

[47] S. Lee and R. Eigenmann. OpenMPC: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE conference on Supercomputing*, Nov 2010.

[48] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th International Symposium on Computer Architecture*, Jun 2010.

[49] S. T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95-09-01, University of Washington, Sept 1995.

[50] M. Li, R. Sasanka, S. V. Adve, Y. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, Oct 2005.

[51] B. Liang and P. Dubey. Recognition, mining and synthesis moves computers to the era of Tera. *Technology@Intel Magazine*, Feb 2005.

[52] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2008.

[53] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[54] LINPACK. Web resource. `http://www.netlib.org/linpack/`.

[55] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[56] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd International Symposium on Microarchitecture*, 2009.

[57] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. GROPHECY: GPU performance projection from CPU code skeletons. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011.

[58] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd Annual ACM International Conference on Supercomputing*, June 2009.

[59] D. Merrill. *Allocation-oriented Algorithm Design with Application to GPU Computing*. PhD thesis, University of Virginia, Charlottesville, VA, USA, 2011.

[60] D. Merrill and A. Grimshaw. Parallel scan for stream architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, Dec 2009.

[61] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[62] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, 2007.

[63] OpenACC. Web resource. `http://www.openacc-standard.org/`.

[64] OpenCL. Web resource. `http://www.khronos.org/opencl/`.

[65] A. Phansalkar. *Measuring program similarity for efficient benchmarking and performance analysis of computer systems*. PhD thesis, University of Texas at Austin, Austin, TX, USA, 2007.

[66] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th International Symposium on Computer Architecture*, June 2007.

[67] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical Report CUCIS-2005-08-01, Department of Electrical and Computer Engineering, Northwestern University, Aug 2005.

[68] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[69] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):18:1–18:15, 2008.

[70] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Nov 2002.

[71] The Standard Performance Evaluation Corporation (SPEC). Web resource. `http://www.spec.org`.

[72] J. Stratton, S. Stone, and W. Hwu. MCUDA: CUDA compilation techniques for multi-core CPU architectures. In *Proceedings of the 21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC 2008)*, Aug. 2008.

[73] STREAM. Web resource. `http://www.cs.virginia.edu/stream/`.

[74] M. Strengert, C. Muller, C. Dachsbacher, and T. Ertl. CUDASA: Compute unified device and systems architecture. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, April 2008.

[75] Parboil Benchmark suite. Web resource. `http://impact.crhc.illinois.edu/parboil.php`.

[76] SPEC Accelerator Benchmark Suite. Web resource. `http://www.spec.org/hpg/press/acceleratorbmk.html`.

[77] I-J Sung, J. A. Stratton, and W-M W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept 2010.

[78] L. G. Szafaryn, K. Skadron, and J. J. Saucerman. Experiences accelerating MATLAB systems biology applications. In *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits*, June 2009.

[79] MATLAB Statistics Toolbox. Web resource. `http://www.mathworks.com`.

[80] E. Ukkonen. On-line construction of suffix trees, 1995.

[81] N. B. Williams, C. Fensch, and S. Moore. A communication characterization of SPLASH-2 and PARSEC. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct 2009.

[82] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[83] Y. Yu and S. Acton. Speckle reducing anisotropic diffusion. *IEEE Transactions on Image Processing*, 11(11):1260–1270, 2002.

[84] E. Z. Zhang, Z. Guo Y. Jiang, K. Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceeding of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2011.

[85] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The Impulse memory controller. *IEEE Transations on Computers*, 50(11):1117–1132, 2001.

[86] Y. Zhang and J. D. Owens. A quantitative performance analysis model for gpu architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*, February 2011.

[87] Y. Zhang, L. Peng, B. Li, J.K. Peir, and J. Chen. Architecture comparisons between NVIDIA and ATI GPUs: Computation parallelism and data communications. In *Proceedings of the 2011 International Symposium on Workload Characterization*, Dec 201.