Improving GPU Performance through Instruction Redistribution and Diversification

A Dissertation Presented

by

Xiang Gong

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

August 2018
To my family for their unconditional love and support.
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>ix</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xi</td>
</tr>
<tr>
<td>Abstract of the Dissertation</td>
<td>xii</td>
</tr>
</tbody>
</table>

1 Introduction

1.1 Toward Peak Performance on GPUs
   1.1.1 Challenge 1: A general lack of appropriate tools to perform systematic hardware/software studies on GPUs  
   1.1.2 Challenge 2: alleviating memory contention in GPUs  
   1.1.3 Challenge 3: the insufficient instruction diversity in GPUs  
1.2 Contributions of This Thesis  
1.3 Organization of Thesis  

2 Background

2.1 Overview of Parallelism
   2.1.1 The Rise of Parallelism  
   2.1.2 The Evolution of Hardware and Software  
2.2 GPU architecture
   2.2.1 Compute Unit  
   2.2.2 Memory Hierarchy  
2.3 GPU programming model
   2.3.1 OpenCL Platform Model  
   2.3.2 OpenCL Memory Model  
   2.3.3 OpenCL execution model  
   2.3.4 OpenCL Programming Model  
   2.3.5 OpenCL compilation model  
2.4 Multi2Sim
   2.4.1 Disassembler
6 Rebalanced Kernel

6.1 Motivation .............................................................. 68
6.2 Rebalanced Kernel .................................................... 70
   6.2.1 Addressing Spatial Imbalance with Kernel Fusion ............... 70
   6.2.2 Addressing Temporal Imbalance with Twin Kernels .............. 82
   6.2.3 Rebalanced Kernel ............................................... 82
6.3 Evaluation .............................................................. 83
   6.3.1 Methodology ..................................................... 83
   6.3.2 Performance Improvement: Stage 1 ................................. 84
   6.3.3 Performance Improvement: Stage 2 ................................. 86
   6.3.4 Further Discussion .............................................. 87
   6.3.5 Predictive Modeling ............................................. 89
6.4 Summary of ReBalanced Kernels ...................................... 96

7 Conclusion .................................................................. 97

7.1 Conclusion ............................................................. 97
7.2 Future Work .......................................................... 98

Bibliography ................................................................. 99
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Overview of a GPU</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Diagram of an AMD GCN Compute Unit</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Diagram of the GCN GPU memory system</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Diagram of OpenCL Platform Model</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Diagram of the OpenCL Memory Model</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Diagram of the OpenCL Kernel Execution Model</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>OpenCL Compilation Models</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>Multi2Sim software modules [36]</td>
<td>20</td>
</tr>
<tr>
<td>2.9</td>
<td>Multi2Sim Runtime and Driver Software Stack [36]</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>M2C architecture diagram</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>The Stages of the M2C front-end</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Abstract Syntax Tree of the Vector Addition kernel</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>The transition of VecAdd Kernel in Muti2C Front-end</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>Stages of the M2C Back-end pipeline</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Vector Addition: translating LLVM IR to SelectionDAG</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>Vector Addition: The DAG before and after instruction selection</td>
<td>38</td>
</tr>
<tr>
<td>4.8</td>
<td>M2V architecture</td>
<td>43</td>
</tr>
<tr>
<td>4.9</td>
<td>Breakdown of Stalls in Compute Unit</td>
<td>46</td>
</tr>
<tr>
<td>4.10</td>
<td>Breakdown of Instructions Count and Cycle</td>
<td>47</td>
</tr>
<tr>
<td>4.11</td>
<td>Utilization of Execution Unit</td>
<td>47</td>
</tr>
<tr>
<td>4.12</td>
<td>Length of Different Type of Instructions</td>
<td>47</td>
</tr>
<tr>
<td>4.13</td>
<td>Instruction Activity Comparison</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>Visualization of instruction of Matrix Multiplication kernel</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>Execution timeline for five systems</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>Workflow for the Twin Kernel Compiler</td>
<td>55</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of a Twin Kernel Binary and two regular kernel binaries</td>
<td>57</td>
</tr>
<tr>
<td>5.5</td>
<td>Assign Twin Kernels to wavefronts by setting the initial PC</td>
<td>58</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance of 5 instruction schedulers in the SIMT model, relative to the average performance of 10 instruction schedulers</td>
<td>59</td>
</tr>
<tr>
<td>5.7</td>
<td>Best performance in SIMT, relative to the baseline</td>
<td>62</td>
</tr>
<tr>
<td>5.8</td>
<td>Speedup of TKMT, relative to the baseline</td>
<td>63</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>5.9</td>
<td>Speedup of TKMT, relative to the best in SIMT</td>
<td>63</td>
</tr>
<tr>
<td>5.10</td>
<td>Potential of TKMT over SIMT</td>
<td>64</td>
</tr>
<tr>
<td>5.11</td>
<td>Evaluation of the impact of PC initialization on performance</td>
<td>65</td>
</tr>
<tr>
<td>5.12</td>
<td>Performance of TKMT while varying work-group granularity</td>
<td>66</td>
</tr>
<tr>
<td>6.1</td>
<td>ALU/MEM Utilization</td>
<td>68</td>
</tr>
<tr>
<td>6.2</td>
<td>Dynamic Analysis: ALU/MEM Instruction Ratio</td>
<td>70</td>
</tr>
<tr>
<td>6.3</td>
<td>Static Analysis: Breakdown of Instructions by Type</td>
<td>71</td>
</tr>
<tr>
<td>6.4</td>
<td>Master Dispatcher Overview</td>
<td>73</td>
</tr>
<tr>
<td>6.5</td>
<td>Master Dispatcher: Timeline</td>
<td>74</td>
</tr>
<tr>
<td>6.6</td>
<td>Work-group Partition</td>
<td>76</td>
</tr>
<tr>
<td>6.7</td>
<td>Logical Work-group Identifier Recalculation</td>
<td>76</td>
</tr>
<tr>
<td>6.8</td>
<td>Idle Work-groups</td>
<td>78</td>
</tr>
<tr>
<td>6.9</td>
<td>Eliminate Idle Work-groups</td>
<td>79</td>
</tr>
<tr>
<td>6.10</td>
<td>Work-group Identifier: Physical to Logical</td>
<td>81</td>
</tr>
<tr>
<td>6.11</td>
<td>Workflow: Kernel Fusion Enhanced by Twin Kernels</td>
<td>83</td>
</tr>
<tr>
<td>6.12</td>
<td>Share of Instructions: Single vs Fusion</td>
<td>85</td>
</tr>
<tr>
<td>6.13</td>
<td>Reasons for Improvement</td>
<td>86</td>
</tr>
<tr>
<td>6.14</td>
<td>IPC Improvement: Rebalanced Kernels</td>
<td>87</td>
</tr>
<tr>
<td>6.15</td>
<td>Breakdown of Kernel Pairs Compatibility</td>
<td>88</td>
</tr>
<tr>
<td>6.16</td>
<td>Pair-wise kernel plot of using static features only</td>
<td>92</td>
</tr>
<tr>
<td>6.17</td>
<td>Pair-wise kernel plot of using the dynamic features from CodeXL</td>
<td>93</td>
</tr>
<tr>
<td>6.18</td>
<td>Pair-wise plot of the dynamic features from CodeXL - removing Conditionally Beneficial kernel pairs</td>
<td>94</td>
</tr>
<tr>
<td>6.19</td>
<td>Prediction Accuracy</td>
<td>95</td>
</tr>
</tbody>
</table>
# List of Tables

4.1 Data Serialization Format .......................................................... 44

5.1 Pre-RA and Post-RA instruction schedulers ................................... 55

5.2 Benchmark Configuration ........................................................... 61

5.3 Model Settings for the 4 GPUs evaluated in this work ..................... 61

6.1 Details of Kernels ........................................................................... 69

6.2 Platform: GPU .............................................................................. 84

6.3 Summary of all static and dynamic features .................................... 91

6.4 Parameter for the Neural Network ................................................. 96
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interfaces</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>CKE</td>
<td>Concurrent Kernel Execution</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>CU</td>
<td>Compute Unit</td>
</tr>
<tr>
<td>DLP</td>
<td>Data Level Parallelism</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random-Access Memory</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GCN</td>
<td>Graphics Core Next</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose computing on Graphics Processing Unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphic Processing Unit</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions Per cycle</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>LDS</td>
<td>Local Data Share</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low Level Virtual Machine</td>
</tr>
<tr>
<td>MSHR</td>
<td>Miss Status Holding Register</td>
</tr>
</tbody>
</table>
MVC  Model View Controller

OpenCL  Open Computing Language

PC  Program Counter

RA  Register Allocation

SIMD  Single Instruction Multiple Data

SIMT  Single Instruction Multiple Thread

SM  Stream Multiprocessor

SOC  System On Chip

SSA  Static Single Assignment

STP  System Throughput

TKMT  Twin Kernel Multiple Threads

TLP  Thread Level Parallelism
Acknowledgments

First and foremost, I would like to thank my advisor, Dr. David Kaeli, for his constant support and excellent guidance during my Ph.D. studies. It would not be possible for me to pursue my Ph.D. without his help. He is a role model for me, both in academic and in life. I feel extremely lucky to be his student.

I would also like to thank Dr. Norman Rubin for sharing his wisdom and vision. Many questions would forever be unanswered if it was not because of him. Also, I would like to thank Dr. Gunar Schirner for his constructive comments and invaluable advice, which helped me greatly improve the quality of this dissertation.

I have had the great opportunity to work with many talented people in the Northeastern University Computer Architecture Research (NUCAR) group, including Zhongliang Chen, Amir Kavyan Ziabari, Xiangyu Li, Leiming Yu, Yifan Sun, Rafael Ubal Tena, Shi Dong, Yash Ukidave, Dana Schaa, Xun Gong, Nicolas Bohm Agostini, Charu Kalra, Fanny Nina-Paravecino, Fritz Gerald Prevolon, Julian Gutierrez, Perhaad Mistry, Trinayan Baruah, and Enqiang Sun. They have shared their thoughts and offered generous help during my studies. I enjoyed the discussions and all the fun we have had.

Last but not least, I would like to express my deepest gratitude to my parents. This dissertation would not have been possible without their encouragement, understanding, and support.
Abstract of the Dissertation

Improving GPU Performance through Instruction Redistribution and Diversification

by

Xiang Gong

Doctor of Philosophy in Computer Engineering

Northeastern University, August 2018

Professor David Kaeli, Advisor

As throughput-oriented accelerators, GPUs provide tremendous processing power by executing a massive number of threads in parallel. However, exploiting high degrees of thread-level parallelism (TLP) does not always translate to the peak performance that GPUs can offer, leaving the GPUs resources often under-utilized.

Compared to compute resources, memory resources can tolerate considerably lower levels of TLP due to hardware bottlenecks. Unfortunately, this tolerance is not effectively exploited by the Single Instruction Multiple Thread (SIMT) execution model employed by current GPU compute frameworks. Assuming a SIMT execution model, GPU applications frequently send bursts of memory requests that compete for GPU memory resources. Traditionally, hardware units, such as the wavefront scheduler, are used to manage such requests. Compute-bound threads can be scheduled to utilize compute resources while memory requests are serviced. However, the scheduler struggles when the number of memory operations dominates execution, unable to effectively hide the long latency of memory operations.

The degree of instruction diversity present in a single application may also be insufficient to fully utilize the resources on a GPU. GPU workloads tend to stress a particular hardware resource, but can leave others under-utilized. Using coarse-grained hardware resource sharing techniques, such as concurrent kernel execution, fails to guarantee that GPU hardware resources are truly shared by different kernels. Introducing additional kernels that utilize similar resources may introduce more contention to the system, especially if kernel candidates fail to use hardware resources collaboratively.

Most previous studies considered the goal of achieving GPU peak performance as a hardware issue. Extensive efforts have been made to remove hardware bottlenecks to improve efficiency. In this thesis, we argue that software plays an equal, if not more important, role. We
need to acknowledge that hardware working alone is not able to achieve peak performance in a GPU system. We propose novel compiler-centric software techniques that work with hardware. Our compiler-centric solutions improve GPU performance by redistributing and diversifying instructions at compile time, which reduces memory contention and improves utilization of hardware resources at the same time. A rebalanced GPU application can enjoy a much better performance with minimal effort from the programmer, and at no cost of hardware changes.

To support our study of these novel compiler-based optimizations, we need a complete simulation framework that can work seamlessly with a compiler toolchain. In this thesis, we develop a full compiler toolchain based on LLVM, that works seamlessly with the Multi2Sim CPU-GPU simulation framework. In addition to supporting our work, developing this compiler framework allows future researchers to explore cross-layer optimizations for GPU systems.
Chapter 1

Introduction

GPUs are specialized processors that were originally developed for accelerating 3-D graphics rendering workloads. With the growing demands from the Graphics User Interface (GUI), and the growing popularity of gaming and entertainment markets, GPUs have become an indispensable component in many systems. In mobile devices, GPUs are offered as an integrated part of the System On Chip (SOC) solution. Many laptop CPUs are equipped with integrated GPUs. A discrete GPUs can also be found in compute systems, such as gaming consoles and PCs, where more processing power is required.

GPUs are designed around a highly parallel architecture that can process videos and images efficiently. By offering massive parallelism to a wider audience, GPUs have evolved into general and programmable accelerators that empower many computationally demanding applications. GPUs are used for High Performance Computing (HPC) applications in the field of Mathematics, Physics, Chemistry, and Biology. When offloading workload to a GPU, many HPC applications have achieved 10-100x speedup over their CPU implementations. In recent years, there has also been growing use of GPUs in cloud computing, machine learning and artificial intelligence, driven by the need to interpret vast amounts of data from sources such as audio, image, and video data. GPUs are commonly used in many deep learning frameworks.

A major reason for the rapid growth in popularity of GPUs is that the hardware can provide high performance and power efficiency, at a relatively low cost. While workloads can be accelerated with traditional accelerators such as Digital Signal Processor (DSP), Field-Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC), and even though these accelerators may achieve even better performance or higher energy efficiency than a GPU, these solutions are generally too costly or difficult to program. In comparison, GPUs can offer great performance and energy
efficiency, at lost cost. GPUs are able to reap the benefits of decades of continuous optimization and massive production of graphics hardware.

Today’s GPUs are also well supported for use by application developers, in stark contrast to more traditional accelerators. General Purpose Graphics Processing Unit (GPGPU) programming models, such as OpenCL and CUDA, provide a rich set of Application Programming Interfaces (APIs) for the programmer. Equipped with these rich programming frameworks, hardware details are abstracted away so that programmers can focus on efficient algorithms. A GPU program can execute on different generations of GPUs, or even on GPUs from different vendors. GPUs support some degree of performance portability of applications. We can typically achieve better performance simply by moving to a more recent GPU platform.

1.1 Toward Peak Performance on GPUs

Today, GPUs have become the accelerator of choice for applications spanning embedded devices to supercomputers. Even with the many advantages of GPUs over other classes of accelerators, they are not free from some of the growing performance challenges longstanding in CPUs. Essentially, GPUs are multi-core processors that share a lot in common with CPUs. Both CPUs and GPUs have pipelined execution units to improve performance. Their memory systems share similar hierarchical approaches that help to reduce memory stalls. As a result, they share some common issues. Both systems struggle when handling stalls that can occur either due to control flow or memory access latencies (i.e., control and memory divergence). Both systems need better solutions when hitting the memory wall [1], where the improvements in processor speed far exceeds the rate of advancement in memory speed. As a result, most GPU applications still cannot fully utilize the compute resources and achieve peak performance, even for GPU applications tuned by experts.

The fundamental limits of hardware are usually blamed for suboptimal performance on a GPU. While it is certainly true that hardware is a major contributor, the performance of a computer system is a systematic problem that involves not only low level hardware units, but also high level software components. Software components, including the programming language, execution model, compiler, runtime system, device driver, and operating system, can affect the performance of a GPU application significantly. Some of the performance issues can be alleviated or even resolved through software redesign, without impacting hardware. However, even give the role that software can play in determining application performance, there has been little effort to consider the interactions of
CHAPTER 1. INTRODUCTION

hardware and software on a GPU, especially at the compiler level. Addressing performance issues in GPUs with compiler-centric approaches is the major focus of this thesis.

1.1.1 Challenge 1: A general lack of appropriate tools to perform systematic hardware/software studies on GPUs

One major reason why there is a lack of hardware/software research on GPUs is that the community lacks an appropriate set of tools. The system tools required to fully understand and optimize performance include benchmarks, compilers, simulators, and data analysis tools. There are specialized tools developed for GPUs that focus on identifying, measuring and analyzing a single domain. However, an open source compiler is still missing in existing frameworks to support GPU research. A complete open source toolchain is crucial for addressing system-wide performance problems. Such a framework can offer a clear view of each component, while opening the door to cross-layer studies. It is surprising that the research community has not devoted more attention to develop such tools for GPU research. The present state-of-the-art in open source GPU toolsets leave a lot to be desired.

This leads to our first challenge on achieving peak performance on GPUs. Researchers cannot systematically study GPUs because of the lack of a comprehensive toolchain that they can fully control. To address this issue, we developed a GPU compiler based on the open source LLVM framework. Our compiler works with the Multi2Sim simulation framework to provide a complete hardware/software toolchain. As part of this thesis, in addition to our compiler framework, we also improve the Multi2Sim simulator by introducing an analysis and visualization tool, which help researchers pinpoint performance issues in a GPU system in an interactive fashion. The compiler toolchain, and the enhancements for Multi2Sim, are detailed in Chapter 4.

1.1.2 Challenge 2: alleviating memory contention in GPUs

The performance of a GPU suffers when programs encounter pipeline stalls. At first glance, this issue can be resolved by adding more threads so that there are more active threads to switch to when stall occurs. However, this also means introducing substantial hardware overhead for resources, including registers and queue buffers to support more active threads. Furthermore, it has been observed that higher Thread Level Parallel (TLP) may offer no improvements at all. Kayiran has found that some GPU applications are not sensitive to TLP. Simply increasing the number of active threads does not translate to higher performance in these applications. In fact, it has been
observed that maximizing TLP can even lead to a degradation in performance for many applications. A further investigation has found that this performance saturation is often caused by memory system bottlenecks.

The memory system looms as a major bottleneck on a GPU, especially given the lack of progress in terms of improving memory technology speeds. To fulfill the massive memory requests, efforts have been made in the memory system. GPUs use high bandwidth memories such as GDDR5 and HBM, to accommodate the increased number of memory requests. However, high bandwidth alone can not effectively bridge the gap. Due to power and area restrictions, it is impossible to scale up memory hardware proportionally to keep up with the processing power of computational hardware. Furthermore, the size of the per-thread cache in GPUs is actually much smaller when compared to CPUs, which means GPUs have less chance to take advantage of temporal and spatial locality. As a result, GPU memory requests takes even longer to fulfill and the GPU memory resources can handle significant lower level of TLP than compute resources.

Unfortunately, the low TLP tolerance of memory hardware has not been a focus of GPU programmers nor GPU compiler writers. When threads cooperate, they tend to generate memory requests in a burst. If the compiler writer could spread out the number of memory requests over time, this could improve performance and utilization. While it is possible to manually coordinate a few CPU threads, it is much more challenging on GPUs due to the massive number of threads. To simplify thread management for GPU programmers, popular GPU programming models require threads to execute the same piece of code. Threads need to be identified explicitly in the code by programmers, so the runtime can coordinate on which data each thread works on. The computation logic of the program is the same for all the threads. This Single Instruction Multiple Thread (SIMT) execution model solves the massive thread management problem. However, the SIMT model does not separate memory operations from computational operations. The costs associated with these two operations are completely ignored in the SIMT model. In a sense, the SIMT model is the culprit that produces massive contention in the GPU memory system.

This leads to our second challenge in improving application performance on GPUs: the high degree of memory contention on GPUs. In Chapter 5 we will address the GPU memory contention problem in detail. We argue that this problem is not only a memory system issue, but also a problem in high-level software design. More specifically, the problem is that we have an execution model that favors simple thread management, and struggles to handle more threads. We introduce a novel compiler-based solution that improves upon the existing GPU execution model. We redistribute memory accesses by reordering instructions for different thread groups. Our solution
improves cooperation among threads, with no effort from the programmer, and only minor changes in the hardware microarchitecture.

1.1.3 Challenge 3: the insufficient instruction diversity in GPUs

The degree of instruction diversity present in a single application may also be insufficient to fully utilize the resources on a GPU. GPU workloads tend to stress a particular hardware resource, but tend to leave others under-utilized. For memory intensive GPU applications, resources (such as caches, shared memory, and DRAM) are in active use during execution, while the computational resources remain mostly idle. Compute intensive GPU applications show the opposite trend: high usage of computational resources, but low utilization of memory resources. Neither kind has sufficient instruction diversity to fully utilize the resources on a GPU. As the number of transistors continues to grow following Moore’s Law, the lack of instruction diversity will become more and more evident in the future.

The diversity of instructions in a GPU kernel is affected by many factors. Statically, compiler optimizations can change the instruction diversity. For instance, loop unrolling can reduce branch instructions and computational instructions. Dynamically, a kernel may take different execution paths that contains different sets of instructions, as a result of input data sensitivity. However, these factors are unlikely to produce radical changes in the kernel, such as transforming a compute intensive kernel to a memory intensive one, or vice versa. Re-balancing instruction diversity requires more than small internal changes.

An effective approach to balance instructions is to consider integration of instructions from outside of the current scope. Previous studies have used coarse-grained hardware resource sharing techniques, such as concurrent kernel execution to utilize the idling hardware resources. However, it is difficult to support sharing hardware resources across different kernels. Introducing additional kernels that utilize similar resources may introduce more contention into the system, especially if the kernel candidates fail to use hardware resources collaboratively. Furthermore, it often requires major reconstruction of program, which is typically done manually by the programmer.

This leads to our third challenge in achieving the full performance benefits available on a GPU: we need to address low instruction diversity on GPUs. In Chapter 6, we identify the imbalance issue caused by insufficient instruction diversity and address it with a two-stage compiler-centric optimization. As a compiler-centric approach, we simplify and automate most part of the optimization. By improving instruction diversity and distribution, our approach produces GPU kernels that balance
the use of GPU hardware in space and time.

1.2 Contributions of This Thesis

In this thesis, we pursue compiler-centric approaches to improve GPU performance. Our approach focuses on reducing over-subscription and improving utilization of GPU hardware. The key contributions are summarized below:

1. We build an open source compiler toolchain that supports the OpenCL 1.2 standard and targets the AMD Southern Islands architecture. Our compiler can generate kernel binaries that are fully compatible with Multi2Sim. To the best of our knowledge, this is the first complete GPU toolchain that is free from vendor dependencies.

2. We develop a new analysis capability in the Multi2Sim simulator. The visualization tool that should greatly aid the GPU research community. Our visualization framework and Multi2Sim work together to provide a detailed record of GPU execution, which helps researchers to quickly pinpoint performance bottlenecks in a GPU system.

3. We perform a study using a broad set of GPU benchmarks, equipped with the enhanced simulator and our compiler toolchain. We have analyzed the impact of instruction scheduling in the traditional SIMT model and illustrated the importance of instruction schedulers in compilers.

4. We evaluate a novel Twin Kernel Multiple Thread (TKMT) execution model that improves hardware scheduling at compile time; TKMT does not require any changes in APIs, runtime libraries or the application itself. We show how we can reliably achieve better performance with our tools, using a systematic approach to tune the memory performance of GPU applications. Our evaluation on four different configurations of AMD GPUs shows that our approach can produce a 12% performance improvement on average over the baseline SIMT model.

5. We characterize a wide selection of GPU kernels on real GPU and reveal the spatial and temporal imbalance issue on GPUs. The imbalance in GPU hardware is a result of insufficient instruction diversity and poor instruction distribution in GPU kernels.

6. We propose a novel Rebalanced Kernel approach that produces GPU kernels with better instruction diversity and distribution. Our study shows that by improving instruction diversity
CHAPTER 1. INTRODUCTION

alone, the system throughput of GPUs can be improved by 24% on average. With further optimization of the instruction distribution, we can get another 4.8% IPC improvement.

7. We explore different machine learning models to identify the best kernel pairs to fuse using our Rebalanced Kernel approach. While we found that a neural network model is the most effective, we acknowledge that our model accuracy of 61% is insufficient, and that further work is needed to reduce Rebalanced Kernels to practice.

1.3 Organization of Thesis

The remainder of this thesis is arranged as follows. Chapter 2 presents background material that includes: 1) a brief overview of parallelism, 2) the architecture of a GPU, 3) the GPU programming model, and 4) the Multi2Sim CPU-GPU heterogeneous simulation framework. In Chapter 3 we review related work on tools for GPU research, including work on novel benchmark suites, compilers, and simulators. We also survey the literature of previous work in the area of memory contention and unbalanced resource utilization on GPUs. In Chapter 4 we introduce M2C and M2V, two additions to the Multi2Sim simulation framework that enable our work and future systematic studies on GPUs. In Chapter 5 we propose, implement, and evaluate our TKMT execution model. In Chapter 6 we present and evaluate the Rebalanced Kernel optimization. In Chapter 7 we summarize the contributions of this thesis, and lay out a plan for future work.
Chapter 2

Background

In this chapter, we present background discussion relevant to future chapters in this thesis. We will start with the rise of parallelism and the changes that occurred in both hardware and software. Next, we dive into details of hardware by presenting the GPU architecture of the AMD Southern Island GPU family, including the design of a compute unit and the memory subsystem. Then we discuss common programming models for GPUs, focusing on OpenCL. Lastly, we introduce Multi2sim, the simulation framework used in this thesis.

2.1 Overview of Parallelism

2.1.1 The Rise of Parallelism

In the past, performance gains in microprocessors were mainly achieved exploiting latency-oriented hardware optimizations, including higher clock frequencies, execution optimization, and memory system improvements [7]. However, it became harder and more costly to squeeze further performance improvements from a single core microprocessor. Increasing the clock frequency hit a ceiling due to the physical limits of the underlying technology and its inherent power constraints [8]. Execution optimizations such as speculative execution cause a superlinear increase in execution complexity and power consumption without linear speedup in application performance [9]. The memory system struggles to catch up with the rate of advancement in processor speed [1].

The performance of single-core CPUs has reached a plateau, so the new focus has shifted to exploiting parallelism [10]. Multiple forms of parallelism, such as instruction-level parallelism (ILP) and data-level parallelism (DLP), have been exploited in single-core microprocessors. ILP enables
concurrent execution of independent instructions, which improves the utilization of hardware [10]. DLP on a single core is enabled by instruction extensions such as SIMD [11] and AVX [12]. With DLP, a single-threaded program can operate on several data elements at the same time, while traditionally only one data element can be processed at a time.

Thread Level Parallelism (TLP) was introduced when multi-core CPUs became popular in the early 2000s [8]. Multi-core CPUs exploit TLP at small scale, where a few cores process independent tasks at the same time. As the number of transistors continues to grow, more threads are supported on modern CPUs. The latest Intel Xeon E7-8894 v4 CPU has 24 physical cores and hyper-threading, enabling 48 threads to run concurrently [13]. With many cores and hyper-threading on modern CPUs, we can easily launch a few threads in a program and gain considerable speedup.

GPUs are supercharged CPUs, as they exploit Thread Level Parallelism (TLP) at a much higher scale. While high-end CPUs can support a few dozens of threads running concurrently, a typical GPU normally can handle thousands of threads executing at the same time. The Nvidia Tesla V100 GPU contains 84 Streaming Multiprocessors (SMs), where each SMs can execute up to 2048 threads in parallel [14]. In total, this GPU can work on up to 172,032 threads at a time, which is 3584x more than the latest Intel CPU. This means GPUs need to pursue alternative approaches than latency-oriented CPUs to achieve high performance.

2.1.2 The Evolution of Hardware and Software

The pursuit of high levels of TLP has fundamentally changed the design principles of microprocessors from latency-oriented to throughput-oriented [15]. Optimizing TLP in multi-core CPUs only requires incremental changes, such as smarter caches and better memory controllers [16][17][18][19]. They are still equipped with latency hiding hardware, as found in single core CPUs. To support a much higher degree of TLP, radical changes have been made in GPU hardware. A GPU replaces latency hiding hardware with huge register files, such that context switching overhead is significantly reduced [20]. This allows GPUs to hide stalls by quickly switching to other threads, instead of worrying about the latency of a single thread, as is done on CPUs [21]. If the GPU can run a massive number of threads to keep thousands of GPU cores busy, a GPU can execute tasks much faster than on a multi-core CPU, even at the expense of increased latency of individual threads. The details of a GPU architecture is presented in Section 2.2.

As we increase the number of threads, we also need to provide proper programming support for these threads. The current programming model for a multi-core CPU is designed to explicitly
manage individual threads in a single program [22]. Programmers are responsible for the interaction of threads. While it is possible to manually coordinate a few threads on a multi-core CPU, it is challenging for the GPU program to manage the large number of threads that typically run on these devices. To simplify thread management for programmers, popular GPU programming models ask all threads to execute the same piece of code. Threads need to be identified explicitly in the code by programmer, letting the runtime coordinate which data each thread will work on. The computation logic for the program is the same for all the threads. The OpenCL programming model is described in detail in Section 2.3.

2.2 GPU architecture

Figure 2.1: Overview of a GPU.

Figure 2.1 shows a high-level overview of the elements that make up a GPU architecture. A GPU typically has two major components: a set of Compute Units(CUs) that in charge of carrying out the parallel computation, and a hierarchical memory system for managing accesses and updates to data. We will use AMD Graphics Core Next (GCN) architecture as an example to discuss details of these two components, and use this state-of-the-art architecture throughout the remainder of this thesis.
2.2.1 Compute Unit

The centerpiece and building block of a modern GPU is the Compute Unit (CU). The general architecture a Compute Unit in the AMD Southern Islands GPU is shown in Figure 2.2. The AMD Southern Islands is the first generation of the GCN architecture, which has a unique scalar/vector architecture [23]. The front-end is comprised of 4 wavefront pools, where each contains 10 sets of program counters and instruction buffers. Therefore, one CU can support a maximum of 40 wavefront in flight at a time, potentially from different work-groups or kernels [23]. The front-end is in charge of fetching instructions from instruction memory for different wavefronts, and sends them to the appropriate execution units.

The execution units in a CU include a scalar unit, a vector memory unit, a branch unit, a Local Data Share (LDS) unit, and a set of SIMD units. As GPUs are designed to deliver high throughput instead of low latency for each kernel, there are no hardware units for branch prediction and not support for out-of-order execution, features more commonly found on a CPU.

The scalar unit is a unique component in AMD GCN architecture. In the scalar unit, only one operation is performed from a group of threads, and the result is broadcast to all of the threads in the group [23]. The scalar unit is capable of integer arithmetic operations and global memory.
access operations. Common operations, such as active mask generation, base address calculation, and loading data from memory to scalar registers, are performed by the scalar unit. The scalar unit was primarily introduced by AMD to improve power efficiency of the GPU \[23\].

The vector memory unit communicates with caches and off-chip global memory. The vector memory unit reads or writes individual data operands for each thread, using the vector registers. This is in contrast to memory operations in the scalar units, where all threads in a group share a single copy of data. The branch unit is responsible for executing control flow instructions. The LDS unit processes instructions that interact with the fast on-chip scratch pad memory and vector registers. Each CU also has 4 highly-parallel SIMD units. Each SIMD unit is comprised of 16 SIMD lanes that perform vector computations across 16 different threads.

2.2.2 Memory Hierarchy

Figure 2.3: Diagram of the GCN GPU memory system.

Figure 2.3 shows a block diagram of the memory system in AMD Southern Islands GPU. The memory resources are colored gray in the diagram, where the lighter the shade means the lower the latency.

Registers are the fastest memory resource exclusive to a CU. In AMD Southern Islands GPUs, each compute unit has an 8KB scalar register file, which is accessible to the scalar unit and the SIMD units \[23\]. The 8KB scalar register file is equally divided into 512 entries and shared by the 4 SIMD units. Each SIMD unit is also allocated a 64KB vector register file. Across all SIMD units, the total is a 256KB vector register file. Each active thread needs their own set of registers. Therefore, the size of register file is a limiting factor for the number of active threads on a GPU.
The L1 and LDS memories are also allocated dedicated memory resources in a CU. Each CU is equipped with a 16KB vector L1 data cache. There is also a 16KB scalar read-only L1 data cache, which is shared by 4 CUs. Each CU also has a 64KB LDS memory of its own, which is comparable in terms of latency and bandwidth with the L1 caches. The LDS memory is shared by all of the threads executing on the same compute unit. Unlike self-managed L1 caches, the management of the LDS memory is under programmer control. A typical practice is to first load repeatedly accessed data into the LDS memory using a single thread, and then perform all successive accesses through the LDS memory. Within a thread group in the same compute unit, the LDS memory can also be used for thread synchronization and communication. When used properly, the LDS can significantly reduce the number of accesses to the slow global memory, and thus, greatly improve the performance of a GPU application.

A cluster of CUs share a single 128 KB L2 cache, which is backed by the GDDR5 DRAM memory that is capable of hosting several GBs of data. Accessing the main memory of a GPU is an expensive operation that should be avoided whenever possible. GPUs can optimize accesses to main memory when certain memory access patterns are detected. For instance, when a group of threads access a contiguous region of memory which fits into a cache line, the hardware can coalesce memory accesses, combining multiple accesses and issue a single memory request instead. Proper memory coalescing can significantly improve performance of many GPU applications, as memory bandwidth can only be effectively utilized when memory accesses are coalesced.

2.3 GPU programming model

CUDA and OpenCL are two popular programming frameworks to program a GPU system. CUDA is a parallel computing platform and programming model developed by NVIDIA for programming on their proprietary GPUs. OpenCL targets a wider range of processors, that includes CPUs, GPUs, DSPs, and FGPAs. OpenCL is supported by major vendors such as AMD, ARM, Altera, IBM, Intel and Nvidia.

OpenCL was first released in 2008 and has since been refined several times. In this thesis, we will use the OpenCL standard version 1.2, the most popular version of the standard. Version 1.2 is also the latest OpenCL standard supported in the AMD Southern Island GPU family.

OpenCL abstracts away the low-level hardware details and provides a rich Application Programming Interfaces (APIs) so that programmers can easily utilize the massive parallelism
CHAPTER 2. BACKGROUND

provided by the hardware. OpenCL is defined in terms of five models: i) platform model, ii) memory model, iii) execution model, iv) programming model, and v) compilation model.

2.3.1 OpenCL Platform Model

![Figure 2.4: Diagram of OpenCL Platform Model.](image)

Figure 2.4 summarizes the OpenCL platform model. The OpenCL platform model is defined as a host device connected to one or more compute devices. A host is typically a CPU running on a standard operating system. The compute device can be a GPU, DSP, or a multi-core CPU. A compute device is composed of a collection of one or more compute units, which is further divided into one or more processing elements.

An OpenCL application executes sequential host code on the host and execute device code, or kernel code in OpenCL term, on the OpenCL devices. The host code is responsible for submitting device kernel code from the host to OpenCL devices. An OpenCL device executes the device kernel code on the processing elements within the device.

2.3.2 OpenCL Memory Model

Following the host-device design defined in the OpenCL platform model, the memory in OpenCL is divided into two parts: the host memory and device memory. The host memory is the memory directly available to the host, defined in a host program. Device memory is the memory directly accessible to kernels that execute on OpenCL devices. There are four distinct memory regions for a OpenCL device memory: i) global memory, ii) constant memory, iii) local memory and
iv) private memory. The memory regions, and their relationship to the OpenCL platform model, are presented in Figure 2.5.

The global memory is a readable and writable memory region shared between all compute units within a given device. Reads and writes to global memory may be cached, depending on the capabilities of the device. The global memory is usually the largest, but slowest, memory in a OpenCL device. Only the host program can allocate space in global memory. The device kernel program is not allowed to allocate memory in this region.

The constant memory is a region of the global memory which remains constant during the execution of the kernel. Similar to global memory, the host program is in charge of the allocation and initialization of constant memory. The device can only read from constant memory.

The local memory is a memory region that is exclusive to a compute unit. Both the host program and the device program can allocate space in the local memory. The allocation can be done dynamically through the host program. However, the size needs to be determined statically (i.e., at compile time) for device kernel program. Although the host program can allocate local memory, it has no access to this memory region. The local memory only grants read and write access to the device kernel program. GPUs usually have a dedicated physical memory for this region. It acts as a fast scratchpad cache under the management of the programmer. When physically absent, such as in CPUs, the local memory is often mapped to the same physical memory as the global memory.

The private memory is a memory space that is accessible only to processing elements. It is
the fastest memory in an OpenCL device. The host program cannot allocate or access this region. The private memory contains all variables in the device kernel program and is mapped to registers in GPUs.

In OpenCL version 2.0, the region of global memory is extended to the memory on the host side, through a shared virtual memory (SVM) mechanism [31]. OpenCL 2.0 defines several levels of SVM in different granularity, that includes coarse-grained SVM, fine-grained buffer SVM and fine-grained system SVM. Atomics is an option in fine-grained SVMs, which enables fine-grained synchronization. With features such as shared virtual address space, regular pointer for buffer identification, and map-free data access, SVM improves the programmability of OpenCL.

2.3.3 OpenCL execution model

The OpenCL execution model is defined in terms of two distinct units of execution: a host program that executes on the host, and kernels that execute on one or more OpenCL devices. The interaction between host and device is performed through command queue. Each command-queue is associated with a single device, which by default processes all commands in a first-in-first-out (FIFO) order [30].

The host program can send three types of command to devices through command queue: 1) kernel command, 2) memory command, 3) synchronization command. A kernel command specifies a kernel to execute on a device. A memory command initiate data movement between the host and the device, between memory objects, or map and unmap memory objects from the host address space. A synchronization command is a explicit synchronization point that enforces a fixed order of execution of commands.

Figure 2.6: Diagram of the OpenCL Kernel Execution Model.
CHAPTER 2. BACKGROUND

In OpenCL, an instance of a kernel is described by an NDRange index space. Programmers need to specify the total number of threads and the size of thread groups that execute together, corresponding to the global size and the local size in OpenCL. To manage the massive number of threads running simultaneously on the device, OpenCL adopts the Single Instruction, Multiple Thread (SIMT) execution model, which is summarized in Figure 2.6.

As shown in Figure 2.6, an NDRange divides the whole threads index space hierarchically into work-groups, wavefronts and work-items. An individual thread is called a work-item in an OpenCL context. A number of work-items, typically 64 on AMD hardware, are grouped into a wavefront and execute in lock-step on a processing element [23]. The number of work-items in a wavefront is platform specific. Several wavefronts are bundled together as a work-group and are mapped to a compute unit. OpenCL guarantees that a work-group is mapped to one compute unit, so that wavefronts within a work-group can coordinate through synchronization primitives, and communicate via global and local memory. The SIMT model is an execution model that works at a wavefront level. Within a wavefront, all work-items execute the same instructions. Work-items from different wavefronts can execute different instructions.

Within a wavefront, work-items execute the same instructions and make progress in lock-step. However, they can follow different execution paths with the assistance of the execution mask. All work-items sharing a common execution that the active status of work-items changes accordingly. Work-items switch their active status and execute both execution paths when divergence is encountered. The result of inactive work-items are not committed [32] [33]. The divergence prolongs the execution as work-items are forced to execute both execution path. This can lead to significant performance loss and therefore should be avoided whenever possible [24].

2.3.4 OpenCL Programming Model

Since an OpenCL program is divided into a host program and a kernel program, these two elements needed to be programmed and compiled separately. The host program is the executable running on the host device, and is typically written in a high level language such as C or C++. OpenCL defines the interface and expected behavior of the APIs for the host program. The implementation of the APIs is provided by the vendor in the form of OpenCL runtime libraries.

An OpenCL kernel program usually consists of a set of kernels that are programmed in a dialect of the C programming language. The OpenCL kernel programming language adds certain restrictions to simplify the hardware and software design. For instance, recursive functions
and function pointers are not supported to simplify stack management \[30\]. System calls that perform memory allocation and file I/O both are absent in OpenCL kernels. These operations are the responsibility of the host program.

As a dialect of the C programming language, OpenCL introduces some extensions. It adds built-in functions to identify threads in the NDRange index space, including get_global_id() and get_local_id(). These built-in functions facilitate thread management for programmers. OpenCL supports standard scalar data types such as int, float and double, as well as more complex data structures defined by the programmer. In addition to scalar types, OpenCL also supports vector data types, including int4, float4 and double4. Furthermore, OpenCL also inherits some data types, such as image2d, image3d and sampler from the Open Graphics Language (OpenGL) \[34\]. The support of graphics data types enables inter-interoperability of general purpose computing and graphics rendering on GPUs.

Listing 2.1 shows an OpenCL 1.2 kernel program that performs vector addition. As can be seen in the code example, each work-item obtains an index from using the get_global_id(0) function and computes the addition from input vector arrays A and B, and storing the result in vector array C. The kernel parameter N is used for boundary check to avoid out-of-bounds memory accesses when the NDRange index space is larger than the array size. An out-of-bounds memory access is undefined behavior that may corrupt the kernel program on a GPU.

```c
__kernel void vecAdd(__global double *A, __global double *B,
                     __global double *C,
                     const unsigned int N) {

    // Get global thread ID
    int id = get_global_id(0);

    // Make sure we do not go out of bounds
    if (id < N) {
    }
}
```

Listing 2.1: Vector Addition kernel
2.3.5 OpenCL compilation model

OpenCL provides two different compilation models: 1) an online compilation and 2) an offline compilation. When an OpenCL program is created by the clCreateProgramWithSource API, the runtime invokes the vendor compiler to perform compilation at run time. Thus, the compiler acts like a Just-In-Time (JIT) compiler [30]. Online compilation supports a cross platform model for OpenCL kernel programs. However, this compilation path may also lead to considerable overhead when the kernel source files are large [35].

Offline compilation reuses the kernel binary generated in a previous online compilation and eliminates the compilation overhead. In offline mode, a kernel is pre-built and directly loaded by clCreateProgramWithBinary API. The binary only contains instructions for the specific targets. To solve possible compatibility issues, a binary file usually contains a copy of the source code. When an incompatible binary is detected, the runtime can fall back to online compilation mode and invoke the vendor compiler to generate the kernel binary.

2.4 Multi2Sim

Multi2Sim is a heterogeneous cycle-level system simulator that models CPUs and GPUs in our work. Multi2Sim provides three CPUs architectures (X86, ARM and MIPS) and two GPU architectures (AMD Southern Islands [23] and Nvidia Kepler [37]). As shown in Figure 2.8, Multi2Sim has four independent modules for each microarchitecture: a disassembler, a functional simulator, an architectural simulator and a visualization tool. These modules can work independently, or can work together in an integrated fashion. Multi2Sim also provides implementations of several
CHAPTER 2. BACKGROUND

runtime systems, that include OpenCL [30], CUDA [29] and HSA [38], to support communication between the host and device programs.

2.4.1 Disassembler

The disassembler is in charge of translating a stream of machine instructions for a selected instruction set architecture (ISA) to an internal representation (IR), that allows a straightforward interpretation of the instruction fields. These fields include the operation code, input/output operands, and immediate constants. When running independently, the disassembler produces text-based output of the internal representation.

Each microarchitecture has its own disassembler that accepts executables for a specific microprocessor. For simulations of OpenCL host programs, the X86 disassembler accepts 32-bit executables generated by compilers such as GCC [39] and LLVM [40]. For OpenCL device programs, the AMD Southern Islands disassembler decodes the instructions of kernel binaries generated by the AMD kernel compiler.

2.4.2 Functional Simulator

The decoded instructions is passed from the disassembler to the functional simulator, which reproduces the behavior of a guest program as if the program is running natively on a given microarchitecture. To keep track of the execution flow and states as in real hardware, the functional simulator implements an image of memory and the state of the register file.
CHAPTER 2. BACKGROUND

When running on its own, the functional simulator emulates the behavior of incoming instructions and updates the state of virtual memory and the register files until the program completes. The results of the simulation can be compared against the native execution for verification. The functional simulator has no concept of microarchitectural details such as pipeline stages, caches and interconnection networks.

2.4.3 Architectural Simulator

The architectural simulator, or timing simulator, models hardware structures and keeps track of their state and models detailed execution/access times. The modeled hardware includes pipeline stages, registers, instruction queues, functional units, and cache memories. The architectural simulator is structured as a main loop, with the simulation pipeline stages symmetrical to the microarchitecture of the corresponding processor. In each iteration, all pipeline stages move forward in time by one clock cycle, as they would in real hardware. The architectural simulator is responsible for the timing progress of the program. Internally, it invokes the functional simulator to handle the flow and emulation of instructions, providing functionally correct execution. This is very handy for performing programming debugging.

2.4.4 Multi2Sim OpenCL Runtime and Driver Stack

An OpenCL program has two major components: the host program and the device program. When executing on Multi2Sim, the host program and device program run on the CPU and GPU modules, respectively. The communication between the CPU modules and the GPU modules is provided by the Multi2Sim runtime and driver. The software stack, and its interaction with the hardware modules, are summarized in Figure 2.9.

Multi2Sim offers a fully functional OpenCL runtime library that needs to be linked with the OpenCL host program. It enables the interception and redirection of the OpenCL API calls. As shown in Figure 2.9 the OpenCL host program is an X86 executable linked with the Multi2Sim OpenCL runtime library. The host program is executed on the X86 modules when using Multi2Sim. All the OpenCL API calls are intercepted in the X86 modules.

To carry out communication between the runtime and the driver, Multi2Sim offers an Application Binary Interface (ABI) interface. In Multi2Sim, each ABI of a runtime-driver pair is assigned a unique system call that is not in use in Linux. The ABI call is forward to the driver, which configures the GPU modules. The driver is a Multi2Sim-specific software layer that runs
CHAPTER 2. BACKGROUND

in user mode. It is not the same as the device driver that runs in kernel mode. Since the simulator
concentrates on the execution of user-level applications, the operating system and kernel mode device
drivers are not involved in the simulation process.

![Diagram of Multi2Sim Runtime and Driver Software Stack](image)

Figure 2.9: Multi2Sim Runtime and Driver Software Stack [36]

2.5 Summary

In this chapter, we focused on the technical background of this thesis. We first reviewed
parallelism and the changes required by the hardware and software design. Then we presented
the architecture of modern GPUs. We focused on the design of an AMD Southern Islands GPU.
Next, we discussed the OpenCL parallel programming model that targets GPUs and showed how it
maps high-level software abstractions to low-level hardware. Finally, we described the Multi2Sim
framework, a heterogeneous CPU-GPU simulator used throughout our work.
Chapter 3

Related Work

In this chapter, we review prior work directly related to the goals of this thesis. We will include discussion of the various tools developed for GPU research, provide a survey of prior work on memory system optimization focused on reducing contention in GPUs, and review software and hardware approaches that improves resource utilization in GPUs.

3.1 Tools for GPU Research

In this section, we focus on prior work on toolsets for GPU research. A typical toolchain for GPU architectural research is comprised of benchmark suites, a GPU kernel compiler, and an architectural simulator.

3.1.1 GPU Benchmarks Suites

A benchmark suite is a collection of programs developed for exploring specific design spaces of a system, such as performance, reliability, and security. The measurement of benchmarks can help processor designers make decisions that often involve microarchitectural tradeoffs. Benchmarks are also used to guide optimizations in software components of a system, such as a compiler, or a runtime. There have been a number of benchmark suites specifically developed for GPU research.

One popular class of benchmarks is an application-based benchmark, one which uses real-world applications or programs with representative workloads. Most benchmark suites used in previous GPU studies fall into this category. Researchers have commonly used the SDK examples provided by the GPU vendors as benchmark suites. Rodinia [41] and Parboil [42] benchmark suites
CHAPTER 3. RELATED WORK

have OpenCL and CUDA implementations to evaluate the performance characteristic of a GPU. They include applications that stress different components of a GPU system. SHOC \[43\] focuses on testing the performance and stability of a system containing GPUs and multi-core processors. The NUPAR \[44\] benchmark suite explores features such as nested parallelism and concurrent kernel execution in modern GPU architectures. HeteroMark \[45\] is a benchmark suite targeting the Heterogeneous System Architecture (HSA) features present on GPUs, with the goal of exploring CPU-GPU cooperative computing. MachSuite \[46\] suite is a collection of 19 benchmarks for evaluating high-level synthesis tools and accelerator-centric architectures. GraphBIG \[47\] is a benchmark suite to study the behavior of graph applications on a GPU. More recent GPU benchmark suites include Chai \[48\] and DNNMark \[49\], which focus on an integrated architecture and DNN workloads, respectively.

An alternative class of workloads is synthetic benchmarks, which are programs specifically constructed to match or mimic the behavior of a large set of programs. Synthetic benchmarks usually do not fully capture the program behavior of real-world applications, and instead focus on reproducing the behavior to exercise a specific program or hardware feature. They are not suitable for characterizing an entire system. Synthetic benchmark has been used in GPU research. MINIME-GPU is an automated benchmark synthesis framework for graphics processing units (GPUs) that serves to speed up the architectural simulation of modern GPU architectures \[50\]. CLgen is a tool that synthesizes many-core OpenCL programs from a large variety of program fragments, using neural networks \[51\]. The performance of the synthesized OpenCL programs is used as training data to improve predictive models on manycore CPUs and GPUs.

3.1.2 GPU Compilers

In any hardware/software system, the compiler provides a bridge that connects the high-level software to the low-level hardware. Compilers are responsible for correctly translating programs written in high-level languages to low-level instructions for a targeted architecture. There are many open source compiler frameworks available that target CPUs. GCC \[39\], Open64 \[52\], and LLVM \[53\] are just a few of the open source compiler toolchains that have been verified, employed and extended by communities of researchers.

There has been limited research on compilers in the context of a GPU, mostly due to the proprietary nature of the GPU software stack. To date, most GPU compilers offered by hardware vendors are closed source, and not accessible to the broader research community. The LLVM project
provides an open source compiler backend for AMD GPUs, but the backend only works with their own proprietary runtime and driver stacks. Nvidia partially open-sourced their compiler infrastructure in the LLVM project. However, their open source effort stops at the NVIDIA PTX intermediate representation (IR) level, and is limited to CUDA, their own proprietary programming standard for GPU computing. This limits GPU researchers in terms of their innovations and optimizations, since they have no control beyond the IR level.

Enhancing the performance of GPU applications with compiler techniques have been pursued in previous work. Yang et al. [54] presented a GPU compiler framework to address the challenge of effective utilization of the GPU memory hierarchy and judicious management of parallelism. Based on static branch divergence analysis, Coutinho et al. [55] proposed branch fusion that merges common code from a divergent program. Han et al. [56] employed iteration delay, an optimization focused on divergent branches within a loop to improve GPU performance. Khorasani et al. [57] introduced a software technique named Collaborative Context Collection to overcome the SIMD inefficiency of divergence caused by intra-warp load imbalance or dissimilar task assignment in GPU kernels. Jablin et al. [58] revisited Trace Scheduling for GPUs, which reorganizes the instruction order to reduce the divergence time.

Another focus in research on GPU compiler is to improve code quality. Previous studies have used compiler techniques to detect bugs in GPU compilers. Compiler fuzzing is a well-established technique that identifies bugs by randomly generating test cases. CLsmith [59] is such a tool that aims to address the compiler correctness problem for many-core systems through the application of fuzz testing for OpenCL compilers.

3.1.3 GPU Simulator

A simulator is an application for reproducing the behavior of a computing device. Computer architects have used simulators for experiments of new features and exploration of architectural trade-offs in microprocessors. Depending on the degree of simulation details, a simulator can be categorized as a functional simulator or a timing simulator.

A functional simulator models the microprocessor’s architecture, or instruction set, providing functional correctness. Functional simulators can only offer limited information about the microprocessor, such as the number of instructions executed and the number of cache hits/misses. Therefore, they are often used for verification and debugging.

Ocelot is a PTX-based functional simulator that provides emulation of CUDA applications,
CHAPTER 3. RELATED WORK

producing detailed instruction traces. Ocelot supports emulation of the NVIDIA PTX ISA, a low-
level parallel thread execution ISA that captures the data-parallel SIMT execution model of CUDA
applications [60]. Ocelot also supports the translation of PTX to other backend targets, such as x86
CPUs and AMD GPUs [61]. The Ocelot framework has been used for compiler, architecture, and
system research on GPUs.

The Barra simulator supports functional simulation of the native instruction set of the Tesla
architecture [62]. It accepts CUDA executables, produced by the NVIDIA tools, as input. Barra can
generate detailed execution statistics based on CUDA emulation.

A timing simulator simulates a microprocessor at a microarchitectural level, cycle-by-cycle.
Microarchitecture details, such as the number and type of pipeline stages, the hardware scheduler, the
caches, and the branch predictor are modeled at a cycle-level in a timing simulator. Timing simulator
can provide detailed cycle-level analysis, that is indispensable in microarchitectural research.

MacSim is a heterogeneous architecture timing simulator that supports x86, ARM64 and
NVIDIA PTX instructions [63]. It can simulate homogeneous ISA multi-core simulations, as well as
heterogeneous ISA multicore simulations. Macsim supports exploration microarchitectural details
to fine tune the processing pipeline, multi-threading, memory system, interconnection network and
power budget. MacSim can be configured as either a trace-driven or execution driven cycle level
simulator.

The popular GPGPUim [64] is a GPU simulator that supports the functional and timing
simulation of Nvidia GPUs. GPGPUim provides detailed modeling of the interconnect network,
caches, DRAM memory, and work-group and wavefront schedulers. It has received contributions
such as an energy model from the GPU research community. GPGPUim only supports the dated
Fermi architecture and CUDA 4.0 software stack by default. GPGPUim is essentially a runtime
library that needs to be linked to the GPU host program at compile time. Therefore, GPGPUim does
not support simulation of the host program, which is executed natively on a CPU.

Gem5-gpu leverages two mature simulators, Gem5 and GPGPUim, to achieve simulation
of host and device programs of a GPU application. It combines the CPU and memory system models
from Gem5 [65], and the compute unit model from GPGPUim, to simulate integrated CPU-GPU
systems. Host code of a CUDA program can be simulated using the cycle-level models of x86,
Alpha, SPARC and ARM CPUs of Gem5. The CUDA kernel is simulated using the GPU model of
GPGPUim.

However, these simulators all suffer from simulation accuracy, as they only work with PTX
instructions. PTX is a high level intermediate representation. GPU kernels in the PTX IR instructions
are optimized for, and translated to, native target-architecture instructions to run on GPUs. Because of the optimization and transformation, a GPU kernel in PTX form can be considerably different from the final form. Furthermore, microarchitectural features, such as shader hints and cache prefetch flags, are not exposed in the PTX IR, which may significantly change the performance of a GPU kernel. Because of the proprietary nature of PTX, these simulators all rely on a software stack from GPU vendors to generate binaries.

3.2 Optimization of Memory Contention in GPUs

3.2.1 Wavefront Scheduling

Wavefront schedulers are custom hardware components that dynamically manage the progress of wavefronts, enabling out-of-order execution on GPUs at the wavefront level. There is no requirement that all wavefronts have to follow the same pattern. When working asynchronously, wavefronts can collaborate and access memory and compute resources in a more efficient manner. A number of improvements to wavefront schedulers have been proposed to alleviate the contention in the memory system.

Early studies in wavefront scheduling have shown that relaxing the strict order of wavefront execution can offer considerable performance improvements for GPU applications. Rogers et al. [66] proposed the cache-conscious warp scheduler (CCWS) to improve cache performance by exploiting intra-warp locality. Narasiman et al. [67] described a two-level warp scheduler to minimize the memory access latency. Jog et al. [68] proposed a coordinated CTA-aware scheduling policy to improve performance by reducing cache contention and improve latency hiding capabilities. Rhu et al. [69] designed a locality-aware memory hierarchy that adapts to fine-grained memory access patterns in irregular applications on a GPU. Sethia et al. [70] proposed an advanced memory-aware scheduler that improves the performance of memory intensive workloads.

3.2.2 Thread Block/Work-group Scheduler

Thread Block Scheduler is a hardware scheduler that dispatches thread blocks/work-groups to the compute unit of a GPU. Previous work has highlighted the problem caused by excessive TLP [64]. Kayiran et al. [2] demonstrated that memory contention can be caused by scheduling the maximum possible number of thread blocks. They proposed a mechanism that uses a dynamic thread block scheduling mechanism to reduce such contention. Lee et al. [71] exploited inter-TB locality by
scheduling consecutive thread blocks on the same SMX. Yu et al. [72] proposed a CTA scheduling optimization scheme to improve the load balance of thread blocks. Pai et al. [73] proposed the Shortest Remaining Time First (SRTF) thread block scheduler which improves performance and fairness for concurrent GPGPU kernels.

3.3 Optimization of Resource Utilization on GPUs

GPU hardware resource demands can vary significantly across GPU applications. For single kernels that cannot fully utilize GPU hardware resources, executing them concurrently can be beneficial to performance. Researchers have proposed software and hardware schemes to exploit Concurrent Kernel Execution (CKE) on GPUs.

3.3.1 Software-based CKE

To support multitasking on GPUs, CUDA and OpenCL programming models have introduced multiple Streams/Command Queues [29] [30]. Within a process, the programmer can explicitly assign GPU kernels to different Streams/Command Queues, establishing their independence from one another. For kernels selected from different processes, Nvidia provides the Multiple Process Service (MPS) that enables transparent CKE. These solutions rely on hardware support and use the left-over policy for resource allocation. With the left-over policy, a GPU assigns as many resources as possible for one kernel, then accommodates another kernel when there are spare resources. As a result, these course-grained CKE solutions have no guarantee of truly sharing GPU hardware resources [74].

A number of software approaches, that aim to deliver better resource sharing, have been proposed. Guevara et al. manually combine two kernels into a super-kernel and improve throughput for both kernels [4]. Pai et al. [75] proposed elastic kernels that dynamically adjusts kernel size to achieve efficient execution of concurrent kernels on GPUs. Zhong et al. resize a monolithic kernel to smaller kernel slices to maximize the opportunity of concurrent execution [76]. Liang et al. [77] utilized concurrent kernel execution to realize spatial-temporal multitasking on GPUs. Bauer et al. use warp specialization to achieve fine-grained CKE for kernels that have a producer-consumer relationship [78].
CHAPTER 3. RELATED WORK

3.3.2 Hardware-based CKE

Researchers also exploit hardware-based CKE to improve GPU utilization. Adriaens et al. exploit spatial multitasking on GPUs, which divides Compute Units (CUs) into multiple groups and assigns each group to a different kernel. However, because of the one-to-one mapping between CU and kernel, the execution unit within a CU may still be under-utilized. To improve the utilization within a CU, hardware approaches focus on intra-CU sharing have been proposed. With hardware modifications, work-groups from different kernels can execute on the same CU. The ratio of work-groups is the focus of these approaches. A Nvidia SMX uses the metric Dominant Resource Fairness to allocate resources of a CU among kernels [79]. Warped-Slicer uses scalability curves to determine the number of co-running work-groups from different kernels. A more recent hardware-base CKE study focused on avoiding interference of concurrent kernels to improve the overall utilization of GPU hardware [80].

3.4 Summary

In this chapter, we reviewed the various tools developed for GPU research, including work on benchmark suites, compilers, and simulators. We showed that the absence of an open source compiler hampers broader and deeper research on GPUs. We will address this issue in the next chapter. We also surveyed the literature of the prior work that alleviates memory contention on GPUs with hardware optimizations. These studies have shown the potential impact of scheduling and inspire us to explore better scheduling methods. In the end, we reviewed prior software and hardware CKE approaches that improve the utilization of GPU hardware.
Chapter 4

M2C and M2V

In this chapter, we will address the first challenge as described in Chapter 1: the lack of appropriate tools to perform systematic studies on a GPU system. We have implemented a GPU compiler and a visualization tool to complete the open source toolchain. The compiler and visualization tool are two new additions tailored to work seamlessly with the Multi2Sim simulator, but they can be extended to work with existing or future simulators. We will present the high-level architecture and low-level details of the two tools. The material presented in this chapter provides a foundation for our work. It should also benefit the GPU research community as well, supporting future GPU research.

The remainder of the chapter is organized as follows: Section 4.1 gives an overview and justifies the choices we made in the making of the compiler. The four major components in M2C compiler and the compiler test suite are detailed in Section 4.2. In Section 4.3, we introduce the M2V visualization tool, which is comprised of a data serialization layer, a back-end, and a front-end.

4.1 Overview

OpenCL is an open standard that enables parallel programming on GPUs. However, it is an open specification rather than an open implementation. It defines a common set of host-side runtime APIs and a device-side kernel language, but leaves the implementation up to the platform vendors. The proprietary nature of a vendor implementation hampers research in areas such as runtimes, compilers, and microarchitecture.

The centerpiece of the toolchain for GPU research is the simulator. As detailed in Chapter 2, Multi2Sim is a cycle-level heterogeneous simulator that supports OpenCL. There are simulators,
such as GPGPUSim and Gem5-gpu, that also support OpenCL simulation. However, only Multi2Sim
works at an ISA level. Furthermore, Multi2Sim has its own implementation of the runtime, driver
and the simulator. It only relies on the GPU vendor to provide the compilers.

Due to intellectual property concerns, most GPU vendors choose to disclose limited
information about their GPUs to the public. AMD is an exception, as they have provided low-level
details of their GCN GPU architecture in their white paper [23]. More importantly, they released the
complete ISA of their Southern Islands GPUs [81], which makes it possible to build an open-source
GPU compiler.

Building a compiler from scratch is a huge amount of work. A compiler framework that
offers basic building blocks would be a great help. The LLVM project is just such a framework
that provides a collection of modular and reusable building blocks for compiler writers. It has been
used to build compiler prototypes, as well as production quality compilers, for major microprocessor
vendors.

In addition to code committed by microprocessor vendors, the LLVM project also receives
community contributions on a variety of GPU implementations [82] [83] [84]. One of the community
GPU implementations is the AMDGCN back-end, which was initially built for compiling OpenGL
shaders for graphics rendering. However, OpenGL shader programs and OpenCL kernel programs
share the same set of instructions and run on the same hardware, which allows us to reconstruct the
project for our needs. By reusing most of the implementation and introducing a few modifications
and new modules, it significantly reduces the workload of compiler development.

4.2 M2C: An open source GPU compiler for Multi2Sim

We take advantages of the LLVM framework and build our compiler based on the
AMDGCN back-end. Our compiler is capable of translating OpenCL 1.2 kernel programs to
the AMD Southern Islands GCN1 instruction set, generating an OpenCL program binary that is fully
compatible with Multi2Sim.
CHAPTER 4. M2C AND M2V

A block diagram of M2C is presented in Figure 4.1. M2C is composed of three major modules: a front-end that translates an OpenCL kernel program to the LLVM IR, a back-end re-targets the IR program to AMD GCN1 instructions, and a finalizer that generates the kernel binary in ELF format.

4.2.1 The Front-end

The responsibility of a compiler front-end is to translate a program written in a high-level language to an Intermediate Representation of the source code [85]. There are several steps the source code must pass through before IR is generated. The pipeline in the M2C front-end is shown in Figure 4.2.

![Figure 4.2: The Stages of the M2C front-end.](image)

4.2.1.1 Front-end Stages

The very first step in the front-end is lexical analysis, where the program source code is split into a set of words and tokens [85]. Characters, including comments, white space, and tabs are removed after lexical analysis. The original program is transformed into a front-end internal representation ready for further processing. As an example, a portion of the tokenized Vector Addition kernel is shown in Listing 4.1.

```plaintext
__kernel '__kernel' [StartOfLine] Loc=<vecadd.cl:1:1>
void 'void' [LeadingSpace] Loc=<vecadd.cl:1:10>
identifier 'vecAdd' [LeadingSpace] Loc=<vecadd.cl:1:15>
l_paren '(' Loc=<vecadd.cl:1:21>
__global '__global' Loc=<vecadd.cl:1:22>
float 'float' [LeadingSpace] Loc=<vecadd.cl:1:31>
star '*' [LeadingSpace] Loc=<vecadd.cl:1:37>
identifier 'A' Loc=<vecadd.cl:1:38>
comma ',' Loc=<vecadd.cl:1:39>
__global '__global' [LeadingSpace] Loc=<vecadd.cl:1:41>
float 'float' [LeadingSpace] Loc=<vecadd.cl:1:50>
...
```
CHAPTER 4. M2C AND M2V

Following the lexical analysis is syntactic analysis, also called parsing in compiler terms. The syntactic analysis groups the tokenized program and checks whether a group of tokens makes sense together, with respect to their physical layout [86]. Syntactical analysis takes as input a stream of tokens and produces an Abstract Syntax Tree (AST), a tree representation of the abstract syntactic structure of source code. As an example, the AST representation of the Vector Addition kernel is shown in Figure 4.3.

After syntactical analysis, semantic analysis is performed, which includes type checking and symbol table generation. Semantic analysis is also in charge of identifying semantic errors and send warnings to the programmer. A concise, yet informative, semantic error report can help programmers quickly pinpoint problems. Eventually, the AST representation is translated to the LLVM IR, a lower level representation of the source code defined in the LLVM infrastructure.

4.2.1.2 LLVM IR Generator

The LLVM IR is language independent and targets an agnostic IR for programs [53]. The IR is a simple, but well-defined, virtual instruction set. Instructions in the LLVM IR are fully typed, which can be rigorously checked for consistency [53]. LLVM supports both scalar and vector data types and provides decent coverage of operations. The LLVM IR uses a Static Single Assignment (SSA) form, which requires that each variable is assigned exactly once, and every variable is defined before it is used. The SSA design simplifies and improves the result of many compiler optimizations [87] [88] [89]. In LLVM, optimizations are implemented as pluggable passes, following the clean and modular design principles adopted by the LLVM infrastructure.
The LLVM IR provides most primitives that are specified in the OpenCL C programming language. Most data types in OpenCL C programming language have corresponding types in the LLVM IR. For example, the 32 bit integer data type in OpenCL C is mapped to an i32 in the LLVM IR. The 32 bit vector data type int4 in OpenCL is represented as $<4 \times i32>$ in the LLVM IR. Most arithmetic and logical operators can also be directly mapped to LLVM IR operations. For instance, the integer `add` operation is directly mapped to the `add` instruction in the LLVM IR.

There are certain primitives that are specific to OpenCL C. When no equivalence can be found, these OpenCL C primitives are usually expanded to a combination of primitives in the LLVM IR. For instance, the `cosine` function, a built-in math function in OpenCL C, is not supported natively in the LLVM IR. In this case, the cosine function is expanded and implemented as a combination of other primitive operations in the LLVM IR. Most built-in functions in OpenCL need to be translated to a hardware independent implementation.

Although most primitives in OpenCL C can be mapped or expanded to LLVM IR operations, there are a few exceptions. For example, the image2d data type is a basic data type in OpenCL but is not supported in the LLVM IR. To produce a legal image2d type, LLVM treats it as an extension of the basic data types to align with the target-agnostic design. Some of the OpenCL C primitives are
CHAPTER 4. M2C AND M2V

tightly coupled with the target hardware, which goes against the target independent principle of the LLVM IR. For instance, the work-item built-in function ‘get_work_item’ is target dependent, as the values are typically stored in buffers associated with the hardware. Usually, these target dependent functions are translated to LLVM intrinsic functions and pass through IR generation. The translation of these intrinsic functions is handed over to later stages, typically in the target-dependent back-end.

Our front-end is based on Clang, a C language family front-end for LLVM [90]. As a member of C language family, OpenCL is officially supported in Clang with the help of the libclc library [91]. Libclc provides an implementation of the library requirements of the OpenCL C programming language. It is responsible for the translation of the non-native primitives. We keep the majority of the Clang code untouched, except for a few modifications, such as adding a new platform identifier and a different data layout for the back-end.

Figure 4.4: The transition of VecAdd Kernel in Muti2C Front-end.

Figure 4.4 shows the transition of a Vector Addition OpenCL kernel to the LLVM IR, using our compiler front-end. As seen in the figure, the integer data type in OpenCL is mapped to the i32 data type in the LLVM IR. The built-in function ‘get_global_id(0)’ is expanded to multiple target-specific functions, which is the first 5 instructions in this example. The condition for out-of-bound checking is translated to a comparison operation and a branch operation in the LLVM IR.

4.2.2 The Back-end

The back-end of a compiler is comprised of a set of code generation, analysis, optimization, and transform passes. The back-end is responsible for converting the IR representation of a program to the target-specific assembly code [86]. For M2C, the back-end receives OpenCL programs in
the LLVM IR form as input, and outputs assembly code that contains AMD GCN1 instructions and metadata.

A program in the IR format goes through the back-end pipeline and eventually turns into instructions of a specific target. The program changes as the program progress through the back-end phases and gets closer to the actual target instructions. There are only a few different representations during the transform: LLVM IR, SelectionDAG nodes, MachineInstr, and MCInst. The overview of the stages in the M2C back-end is shown in Figure 4.5. The stages that change the representation of a program are colored dark blue in this diagram.

Before entering the back-end pipeline stages, optimizations can be applied to reduce redundancies introduced during IR translation in the front-end. It is also a place to improve the quality of the code, addressing poor programming practices. For example, loop-invariant code motion can be used to remove the loop-independent code from the body of a loop, which is easily neglected by programmers. Redundant operations can be optimized by either hoisting code into the pre-header block, or by sinking code to the exit blocks, whenever these operations are safe to apply.

The first stage in the back-end is transforming the IR to a Directed Acyclic Graph (DAG) representation, which provides an abstraction to facilitate instruction selection and low-level optimizations. In this stage, the first step is to build the initial DAG by translating the LLVM IR to a SelectionDAG representation. Each basic block is associated with a different DAG, where the nodes represent instructions or operands. The edges represent the data flow or the use-def relationship of associated operations. Figure 4.6 shows part of the SelectionDAG for the entry basic block in the Vector Addition kernel. The SelectionDAG for the LLVM IR instructions is shown in blue on the right.

At this point, the SelectionDAG is usually is incompatible with the target, as it may use some data types that are not supported by the targeted architecture. Unsupported types need to be
Figure 4.6: Vector Addition: translating LLVM IR to SelectionDAG.

transformed to supported types before entering the instruction selection stage. For the unsupported types, LLVM offers three actions: i) promote, ii) expand, and iii) customize [86]. Respectively, the unsupported operations are upscaled to a larger type, break down to a set of smaller types, and use custom transformations to build a legal SelectionDAG.

Two rounds of optimization and legalization are performed after the initial construction of the DAG. The first round is a DAG combine pass that simplifies the DAG, followed by a legalization stage to eliminate the unsupported types of the target. Then the second round of the DAG optimization is performed to clean up any residual redundancies, followed by the second round of DAG legalization to eliminate any remaining unsupported types. Finally, a third optimization takes place to remove inefficiencies introduced by the second legalization operation.

After three DAG-combine optimizations passes and two type-legalizations passes, the SelectionDAG is legal and ready for instruction selection, which transforms the program from the SelectionDAG form to a target-specific MachineDAG form. In this stage, a series of pattern matches are performed on the SelectionDAG nodes. The simple matching patterns, and their orders, are defined by a Domain Specific Language (DSL) called TableGen [93]. Eventually, the TableGen program is translated to C++ code, which generates a huge pattern matching/handling table. The back-end follows the order in this table to find a match for the nodes in the SelectionDAG. When a match is found, the SelectionDAG node is translated to a MachineDAG node, following the rules set by the pattern match.
CHAPTER 4. M2C AND M2V

Figure 4.7: Vector Addition: The DAG before and after instruction selection.

The SelectionDAG and MachineDAG representations are structurally similar. In most cases, nodes are just a direct mapping from the IR operations to target specific operations. Figure 4.7 represents the DAG of the vector addition kernel before and after instruction selection. As seen in the figure, the bottom ‘add’ node of the SelectionDAG is mapped to the ‘S_ADD_I32’ node of the MachineDAG, where the connections between nodes remain unchanged. There are cases where the connection of the nodes, or the structure of the DAG, needs to be changed, such as transforming intrinsic function nodes to a set of target-specific instructions. Due to the complexity, these transformations are not defined in TableGen. They are identified in the matching table, and then transferred to user-defined handler functions for further transformation. The instruction selection is the last stage, where the program is represented as a set of DAGs.

Following the instruction selection stage, the DAG form is flattened to a serialized representation. As the DAG presentation does not imply the ordering of independent instructions, the first round of instruction scheduling, also called Pre-Register Allocation(RA) scheduling, is performed during the serialization of the DAGs. The LLVM provides several target independent Pre-RA scheduling methods that emphasize the different aspects of code quality [86]. After completing scheduling, the program is transformed to a serialized MachineInstr representation, which is closer to the final target instructions. However, the whole program still uses SSA form and the registers are virtual registers without considering physical register limits.

Register allocation is the stage that the virtual registers are transformed into physical registers. SSA form is also destructed as part of register allocation. Since the number of physical registers is limited based on the target microarchitecture, virtual registers are assigned to memory...
locations. This assignment is also known as register spilling, which generally leads to a performance loss for latency-oriented processors such as CPUs. However, performing register spilling to reduce register usage may improve performance on GPUs as a result of a high degree of parallelism, which is inversely proportional to the number of allocated registers \[94\]. Register allocation can be abstracted as a Graph Coloring problem, an NP-complete problem that is impractical to solve, given speed and flexibility constraints \[95\]. As a compromise, LLVM offers register allocators that sacrifice quality for faster speed. The default performs a linear scan register allocator. LLVM also offers several other alternatives, such as greedy, fast, and Partitioned Boolean Quadratic Programming (PBQP) based register allocators.

After register allocation, the program is using target-specific instructions and physical registers. Then the second instance of instruction scheduling, also named Post-RA scheduling, is performed. Post-RA scheduling is generally less flexible than Pre-RA scheduling due to the constrained set by the physical registers \[86\]. However, since the actual number of physical registers is exposed at this point, the presence of extra hazards and delays associated with certain types of registers can be used to improve the instruction order.

The last step in the back-end is code emission, where the scheduled machine instructions are replaced by machine code instructions, a less informative, but lighter weight representation of the target instructions. The major responsibility of the code emission in our back-end is to generate the GCN1 assembly and other metadata, that will be useful for kernel execution. We use a simple format for the emitted code, which has four sections in plain-text form for each kernel: a global section that contains the name of the kernel, a text section with all the instructions in assembly, an argument section with argument description, and a metadata section that contains information such as register usage and buffer indices for the binary. An example is shown in Listing 4.2 which is the Vector Addition kernel emitted by the back-end.

```assembly
.global vecAdd

.text
  s_buffer_load_dword s0, s[8:11], 0x0
  s_buffer_load_dword s1, s[8:11], 0x4
  s_buffer_load_dword s13, s[8:11], 0x8
  s_buffer_load_dword s14, s[8:11], 0xc
  s_load_dwordx4 s[8:11], s[2:3], 0x60
  s_load_dwordx4 s[16:19], s[2:3], 0x58
  s_load_dwordx4 s[20:23], s[2:3], 0x50
```
4.2.3 The Finalizer

In a regular compiler, the finalizer generates object files with the output from the back-end, then hands them over to the linker. A linker connects all the object files and produces the final executable in a specific binary format, which contains instructions, as well necessary metadata, needed for execution on a microprocessor.

GPU kernels have no concept of the heap and stack memory space. The heap memory is essentially prohibited in the OpenCL C programming specification, as dynamic memory allocation
CHAPTER 4. M2C AND M2V

is not permitted [30]. There is no stack memory on a GPU. Stack memory is needed for the
bookkeeping of nested function invocations, which is a supported feature in the OpenCL C language.
However, it can be eliminated with function inlining [96]. Function inlining simplifies the memory
management and removes bookkeeping overhead. The absence of a heap and stack means individual
GPU kernels are self-contained. It also makes the linker, the component in a compiler that assembles
multiple object files or external libraries, no longer needed. In our compiler, the finalizer takes the
responsibility of generating the binary.

The finalizer in M2C is essentially a Domain Specific Language (DSL) compiler that is
comprised of three components: a lexer implemented by Flex [97], a parser developed by Bison [97],
and a binary generator. Similar to the front-end of M2C, the lexer reads the code emitted from
the back-end and splits it into tokens. These tokens are consumed by the parser to build the AST,
using a simple language. The grammar of this language is based on the format of the output from
the back-end. Following the tree structure produced in the parser, the binary generator does sanity
checking, encodes instructions, and produces the ELF-formatted binaries. The binary generator
produces one ELF binary for each OpenCL kernel function, then encapsulates them into a binary
container that is also in ELF format.

4.2.4 The Test Suite

The first priority of a compiler is to generate correct code, with later passes considering
objectives such as execution performance and binary size. For OpenCL, it is critical to avoid bugs
during the construction of the compiler. As a common practice, OpenCL kernels are usually compiled
at runtime, which means an OpenCL compiler bug cannot be easily detected and circumvented with
software workarounds.

Making a bug-free compiler is a great challenge. An OpenCL GPU compiler needs to
support the full range of OpenCL features, that includes four different memory spaces, a rich set
of scalar and vector data types and operations, a number of atomic operations and synchronization
primitives. And the use of complex data structures in OpenCL kernels brings more complexity.

The unit test is a widely used software engineering technique to find problems early in the
development cycle [98]. The essence of a unit test is to isolate each part of the program and validate
its correctness. Following the isolation principle, we developed a compiler test suite to assist the
development of our compiler. The test suite is comprised of manually and automatically generated
test cases. Each test case is a device kernel and host program pair that uses some OpenCL features.
CHAPTER 4. M2C AND M2V

Based on the number of features used in the kernel program, the test cases are categorized into three levels: basic, extended, and full.

The tests in the basic level are unit tests that verify a single feature in the OpenCL C language. The test coverage is the major concern in the basic level tests. In our test suite, these tests are generated fully automated by a template-based generator. The generator synthesizes the device kernels by exhaustively searching combinations of OpenCL primitives, such as data types, operations, and built-in functions. The associated host programs and build system are also generated automatically.

The extended level contains kernel programs that use a few features in a single kernel function. These tests are mainly designed to discover bugs introduced by optimizations in a compiler. Each test has a different testing focus, such as a branch, a loop, and a synchronization. Due to the complexity, these tests cases are manually constructed. Unlike the basic level, the result is compared to an expected value, rather than the result of a vendor implementation.

The full level is comprised of OpenCL programs from a benchmark suite, so that the verification is self-contained. These programs have CPU implementations of the OpenCL kernel so that the execution evaluated. We use a number of kernels that cover most of the features in the OpenCL C language.

4.3 M2V: A visualization infrastructure for Multi2Sim

Multi2Sim is a powerful simulation framework in terms of speed, accuracy, and feature coverage. However, it is relatively limited in terms of our ability to analyze simulation results. Multi2Sim can generate a brief and detailed report of the execution units, the network, and the memory subsystem. These reports are supposed to characterize the application so that researchers can get a better understanding of the system. It should reveal the bottlenecks in the system so that researchers optimize the hardware and software, accordingly. However, both brief and detailed reports generated by Multi2Sim fall short.

The major limitation of the profile is the lack of detail. It contains metrics for the entire program, though cannot break down statistics to the execution of individual kernels. Therefore, it is challenging to characterize a single kernel and pinpoint a performance problem. Further, this brief report is hard to interpret, since it provides a summary with a few counter metrics of each compute unit in plain text.
CHAPTER 4. M2C AND M2V

On the other hand, the detailed report is too verbose and it offers no informative statistics for the user. It stores all the events in each cycle during simulation in plain text. The events have no statistical meaning by themselves. The detailed report can be interpreted by the built-in visualization tool in Multi2Sim. However, this tool is a simple visualization of the report file that only shows the pipeline status in each cycle. Analysis and statistics are not offered in this tool. Furthermore, this tool struggles at long reports, which can involve processing several TBs of simulation results.

The desired simulation output should offer the ability to replay a simulation with adjustable resolution. It should also only provide the information of interest to the user, rather drowning the user in details. With this principle in mind, we extend the Multi2Sim framework and introduce a data serialization layer in the timing modules. The data serialization layer simplifies the interpretation of the events by transforming the cycle-based report into an instruction-based profile. The instruction-based report is interpreted by M2V, our visualization infrastructure tailored for Multi2Sim. M2V contains several analysis modules that provide rich choices of metrics. It also provides flexible resolution so that users can zoom in to check the details, or zoom out to get an overview. We also implemented filters so that users can focus on the information of interest.

M2V is tailored for Multi2Sim, but the layers in M2V are completely separated from each other. Upper layer communicates with the lower layer through structured information exchange formats. Other simulators or real system can also work with M2V, as long as they produce the report with the same format. M2V adopts the Model-View-Controller (MVC) architecture, a design pattern wildly used in programs with graphical user interfaces. The structure of M2V is shown in Figure 4.8.

![M2V architecture](image)

4.3.1 The data serialization layer

The data serialization layer defines the data format so that pipeline status can be restored and reconstructed. Due to the huge size of the trace file, the data serialization layer is built inside timing modules in Multi2Sim, instead of working as an offline parser that acts on the trace files. The field and description in the serialization data structure are summarized in Table 4.1. The serialized data is stored in a CSV file, which can be directly loaded to the database by the back-end.
CHAPTER 4. M2C AND M2V

<table>
<thead>
<tr>
<th>Field of Serialized Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
</tr>
<tr>
<td>Finish</td>
</tr>
<tr>
<td>Length</td>
</tr>
<tr>
<td>StageStart</td>
</tr>
<tr>
<td>StageEnd</td>
</tr>
<tr>
<td>StageStallWidth</td>
</tr>
<tr>
<td>StageStallBuffer</td>
</tr>
<tr>
<td>GID</td>
</tr>
<tr>
<td>ID</td>
</tr>
<tr>
<td>CU</td>
</tr>
<tr>
<td>IB</td>
</tr>
<tr>
<td>WF</td>
</tr>
<tr>
<td>WG</td>
</tr>
<tr>
<td>UOP</td>
</tr>
<tr>
<td>Assembly</td>
</tr>
<tr>
<td>ExecutionUnit</td>
</tr>
<tr>
<td>InstType</td>
</tr>
</tbody>
</table>

Table 4.1: Data Serialization Format

4.3.2 The back-end

The back-end is where the heavy tasks are performed, that includes HTTP service, requests router, database management, and data analysis module. The back-end is completely written in GO, an open source programming language developed by Google [99]. As all dependencies are statically linked in the Go language, the back-end executable has everything self-contained, which makes the deployment hassle-free.

The back-end can operate in normal mode or daemon mode. When working in normal mode, M2V is a data importer that reads the configuration file, which specifies the database configuration, then import the serialized data to the database. When no database is specified, it can launch a MySQL container instance with default settings and tables. Once the data importing job is finished, the program will terminate.

When running in daemon mode, the back-end provides all other services for the front-end. One responsibility is to act as a web server to host the front-end files for the browser, that includes some HTML pages, a JavaScript file and a few resources such as images and CSS files. It is also in charge of processing queries from user interactions. These queries are handled in the router module and forwarded to associated responders, where the data analysis takes place. Statistics are mostly...
CHAPTER 4. M2C AND M2V

dynamically calculated in database queries, with a few exceptions where in-memory analysis is performed for performance consideration.

4.3.3 The front-end

The major responsibility of the front-end is to provide the interactive interface to users. To take advantages of existing frameworks and libraries, the whole front-end is implemented by web techniques such as HTML, CSS, and JavaScript. With the chosen software stack, the user interaction is provided as a collection of web pages running in a browser. The front-end is designed to be lightweight to minimize the overhead in data visualization. Therefore, the front-end only keep the record of necessary information such as the selected metrics, cycle ranges, and other filter selections. It passes information as HTTP queries to the back-end, where data retrieving and analysis are performed.

VueJS is the JavaScript framework of our choice to construct the front-end. It is a light-weight library optimized for building and rendering web interfaces. As rendering web page in a browser is a costly operation, VueJS only updates web pages when absolutely necessary. It also minimizes rendering area by only re-rendering the changes whenever possible. With VueJS, the whole web application is contained in a single web page, or a Single Page Application (SPA) in web terms. Besides the efforts made by the VueJS framework, we also limit the range of queries to the visible area, so that no excessive data is retrieved.

There are several other JavaScript libraries used in the front-end. The user interface is built with Element UI, a desktop UI library based on VueJS. It provides a wide variety of UI components, from basic icons and buttons to complex tables and navigation menus. The switch among different visualization views is realized by the Vue-router library. Retrieving data from the back-end is handled by Axios, which gets the data asynchronously without blocking tasks for page rendering. The retrieved data is visualized by ECharts, which offers the intuitive, interactive, and highly customizable charts. All libraries are managed and compiled by WebPack, which creates the bundled static assets that are acceptable by browsers.

4.3.4 Visualization Examples

A few examples produced by our visualization tool are presented in this section. These examples provide the overview as well as detailed information of the Reduction kernel simulated on Multi2Sim.
CHAPTER 4. M2C AND M2V

The bottleneck of a compute unit is often caused by stalls in execution units. A high number of stalls indicates bottlenecks in the execution units. The stalls of an execution unit can be checked in the stall view in M2V. Figure 4.9 shows the breakdown of stalls in each compute unit, categorized by pipeline stages. There are 6 different stages in Southern Islands pipeline: Frontend, Issue, Read, Decode, Execution, and Write. It is clear from this figure that we have considerable issue stalls during execution, which is the focus of many previous studies [66][67][68].

Figure 4.9: Breakdown of Stalls in Compute Unit

Figure 4.10 shows the breakdown of dynamic instructions. The count of instruction is shown on the left, which gives us an overview of the instruction diversity in the GPU kernel. The accumulated cycles spend on each instruction category is presented on the right, which reflects the workloads of execution units. As shown in the figure, the Data Share(DS) instructions take the largest proportion. Therefore, the LDS unit to execute these instructions is likely the busiest hardware unit during kernel execution, which is confirmed in the utilization view as shown in Figure 4.11, which shows the utilization of execution units in two compute unit. We can also observe memory contention in this figure, as the VMEM execution unit is mostly fully stalled when it is in use.

For a detailed characterization of each type of instruction, we can investigate the average, min and max execution time. As shown in Figure 4.12, the latency of MTBUF instructions is
CHAPTER 4. M2C AND M2V

Figure 4.10: Breakdown of Instructions Count and Cycle

Figure 4.11: Utilization of Execution Unit

Figure 4.12: Length of Different Type of Instructions
CHAPTER 4. M2C AND M2V

considerably higher than the others. MTBUF instruction is executed in the Vector Memory unit, which is usually the bottleneck of a GPU system. A more detailed comparison of two compute units is shown in Figure 4.13, which visualizes the count of active instructions in each cycle of the compute unit.

![Figure 4.13: Instruction Activity Comparison](image)

4.4 Summary

In this chapter, we complete the toolset for systematic studies on GPUs. We first introduced our OpenCL compiler, with a detailed description of the layers and the passes. We also provided a compiler test suite to assist the development of our compiler. Then, we presented M2V, an addition to Multi2Sim for simulation result analysis and visualization. M2V provides an intuitive way to analyze simulation result in our research, which can also be beneficial to other Multi2Sim users.

In the next chapter, we will use our toolchain to construct a novel compiler-based approach to address the second challenges: alleviating memory contention in GPUs.
Chapter 5

Twin Kernels

Offloading workload to accelerators, such as Graphics Processing Units (GPUs), has become a popular path to high performance due to the massive parallelism available on these devices. However, peak performance is often challenging to achieve, even for hand-tuned GPU applications optimized by experts. As mentioned in Chapter 1, one major cause is memory system bottlenecks, which often result in under-utilized compute resources.

In comparison to computational stalls, memory requests are responsible for a majority of the stalls on many applications targeting a GPU. When requests arrive at the memory system, they have to compete for limited resources, that include: cache blocks, Miss Status Holding Registers (MSHRs), and memory interconnects. This competition can impact the average response time, and can result in request queuing, cache thrashing, and network congestion. When the memory system is saturated, a new memory request has to wait until older memory requests are serviced, resulting in pipeline serialization and long queuing delays. Furthermore, as memory serialization delays an access to the data cache, the possibility of a data eviction and replacement with an address from an intervening wavefront also increases [70]. As a consequence, it may take several attempts to retrieve an address, generating repeated requests to DRAM, resulting in even more traffic in the memory system.

Despite the impact on the service times of memory requests, memory bottlenecks are further exacerbated due to the design of the GPU programming model. The SIMT execution model employed in modern GPU programming frameworks only take advantage of TLP in compute, and pay little attention to long latency memory operations. GPUs use a form of round-robin wavefront scheduling, so the SIMT model essentially ensures that a large number of memory requests will arrive at the L1 cache within a short time window [67]. This can quickly saturate the bandwidth of
CHAPTER 5. TWIN KERNE LS

the GPU’s memory system.

In this chapter, we propose the Twin Kernel Multiple Thread (TKMT) execution model, a compiler-centric solution that improves collaboration of wavefronts through instruction scheduling. By reordering instructions for selected wavefronts at compile time, TKMT better distributes memory requests. This can significantly reduce competition for bandwidth, and can help alleviate one cause of memory bottlenecks present on current GPUs. Our execution model can offer out-of-order execution capabilities without changing the hardware-based wavefront scheduler. The TKMT model is also an improvement on the SIMT model that preserves the existing memory consistency model.

The remainder of the chapter is organized as follows: In Section 5.1 we conduct a range of analyzes to assess the impact of instruction scheduling on GPU performance and present the TKMT model. Based on the work in Chapter 4 we extended the compiler in the toolchain to realize the TKMT execution model, which is detailed in Section 5.2. In Section 5.3 we show how we can reliably achieve better performance with our tools, using a systematic approach to tune the memory performance of GPU applications.

5.1 Motivation

Figure 5.1: Visualization of instruction of Matrix Multiplication kernel

The early studies presented in Chapter 3 have shown the potential impact of scheduling and inspire us to explore better scheduling methods. We note that for in-order processors, the order of instructions defines the characteristics of a program. The type of an instruction determines the hardware resources it requires, and its location in the program roughly reflects the time when the associated hardware is accessed. Therefore, we can guide the execution of a program by manipulating the order of instructions, which is determined by the instruction scheduler in the compiler back-end.

Instruction schedulers can arrange the order of instructions in a distinctive manner. Figure 5.1 shows the range of instruction sequences that were produced by 10 different instruction schedulers, all from compiling the same kernel source. Instructions executing on different hardware
units are denoted with a different color. As shown in the figure, patterns differ from each other significantly. However, as they are all generated from the same Matrix Transpose kernel, the functionality is the same. The diversity of the instruction order enables a different execution model that we will discuss in the next section.

5.1.1 Instruction Scheduling and Performance

In this section, we study the impact of instruction scheduling and present our execution model that takes advantage of the distinct distribution of instructions in a kernel. Instruction scheduling can significantly impact the performance of applications, without requiring any modifications to the hardware. Figure 5.2 presents execution timelines for our five workloads across five different system configurations.

As shown on the left, the two programs used in these workloads contain the same set of instructions. Kernel 1 starts with 2 successive load instructions, followed by 6 ALU instructions. The second load instruction is moved down to line 5 in Kernel 2. The five workloads are shown on the right. The top shows Kernel 1 running on a system with infinite resources. Second from the top shows Kernel 1 executing on a system with 3 Miss Handling Holding Registers (MSHRs). The middle shows scheduling for Kernel 2 on an ideal system. Second from the bottom shows Kernel 2 running on a system with 3 MSHRs. The bottom shows the program executing on the TKMT model.

Figure 5.2: Execution timeline for five systems.
CHAPTER 5. TWIN KERNELS

For simplicity, we assume each memory instruction takes eight cycles to execute and each arithmetic instruction takes one cycle. Only a single wavefront can issue a memory instruction or arithmetic instruction per cycle in these systems. There are 4 wavefronts executing in parallel and these systems only support in-order execution. All systems employ a round-robin policy (across ready wavefronts) to schedule wavefront. The details of the five timelines are explained as follows.

Example 1: Kernel 1 with infinite resources: Instructions in Kernel 1 are scheduled such that the first memory instruction is followed by the second, immediately. In this example, the system has infinite resources for memory requests so that memory instructions can issue at any cycle. Therefore, in wavefront 0, the second load does not need to wait for an available MSHR and can issue at $t = 5$. When the first load returns at $t = 8$, the add operation can start at $t = 9$. With unlimited resources, kernel 1 finishes execution in 32 cycles and a maximum of 8 MSHRs are used. Memory operations take 15 cycles to execute and are overlapped for 7 cycles with compute operations, resulting in an overlap ratio of 21.9%.

Example 2: Kernel 1 with three outstanding requests: Compared to the previous configuration, the number of outstanding miss requests is reduced to three. Once the number of memory requests exceeds this limit, no new requests can be issued until outstanding requests return. This limitation causes delays in some memory operations. For wavefront 3, the first load has to wait until $t = 9$, after the first load of wavefront 0 returns. The second load in wavefront 2 proceeds in a similar fashion. With limited resources, the same kernel takes 42 cycles to execute, while memory and compute operations overlap for 7 cycles. However, the memory operations take 25 cycles in this scenario, which is 10 cycles longer than the first example. As a consequence, the overlap ratio is decreased to 16.7%.

Example 3: Kernel 2 with infinite resources: In Kernel 2, the two memory instructions are separated by three arithmetic instructions. Constrained by the in-order execution design, the second load in wavefront 0 cannot issue until $t = 20$, even though there is no resource limitation. Memory operations take 22 cycles in total, which is 7 cycles more than the first example, where both have infinite resources. The kernel finishes execution in 40 cycles, while compute and memory operations overlap for 6 cycles. The overlap ratio drops to 15% as a result of the longer execution time.

Example 4: Kernel 2 with three outstanding requests: Similar to Example 2, some memory operations are delayed due to resource limitations. Consequently, memory operations take 34 cycles to finish and the total execution time increases to 41 cycles. This system exhibits poor utilization of memory resources, as we can find that only 1 MSHR is in use for 17 cycles. The
memory and compute operations overlap for 24 cycles, which is significantly better than Example 2. However, even with a much higher overlap ratio (58.5%), we cannot hide the long memory latencies. In the end, this setting is only 1 cycle faster than Example 2, where the overlap ratio was a lowly 16.7%.

**Example 5: TKMT with three outstanding requests:** Kernel 1 and kernel 2 have the same functionality, but their runtime performances are quite different. Using a round-robin wavefront scheduler, Kernel 1 stresses the memory system, but does not have enough computational intensity to hide the memory latency. Kernel 2 works better at latency hiding, but the memory system is underutilized. We need to compensate for the weaknesses in each approach by working together. However, in the traditional SIMT model, all wavefronts must execute the same kernel. We relax this restriction and allow hybrid execute of kernels for wavefronts. We refer to this execution model as the **TKMT execution model**. More details of our model are discussed later in Section 5.2. Using the TKMT model, we assign wavefronts 1 and 3 to execute kernel 1, while wavefronts 0 and 2 execute kernel 2. The MSHR configuration supports three outstanding requests, just as in examples 2 and 4.

As seen in the bottom timeline, the program finishes execution in 37 cycles when using the round-robin wavefront scheduler. Load instructions finish in 27 cycles and overlap with 15 arithmetic instructions, resulting in a 40.5% overlap ratio. We also observe that the MSHRs achieve higher utilization during those 27 cycles. Using our TKMT model, this system achieves both improved compute-memory overlap and reduced memory latency.

Previous work has highlighted the problem caused by excessive TLP [64]. We borrow the idea of reducing the maximum TLP to alleviate memory contention. However, it is achieved by producing a better distribution of instructions, versus using thread block throttling, as done in the cited previous work.

### 5.2 Implementation

SIMT is the prevailing computing model for GPU frameworks. SIMT supports only a single kernel to execute at a time. Multiple Program Multiple Data (MPMD) is an execution model that supports multiple programs to process different data, concurrently. Our proposed model is a hybrid combination of the SIMT and MPMD models: the kernels in our model are identical in terms of functionality, but can differ significantly in terms of instruction schedules. To limit complexity, we only support two concurrent kernels, and we name the individual kernels as Twin
CHAPTER 5. TWIN KERNELS

Kernels. Consequently, our execution model is referred to as a Twin Kernel, Multiple Thread (TKMT) model. In our discussion here, we will refer to the two kernels as the First and Second Twin Kernels.

Our TKMT execution model schedules workloads executing different Twin Kernels to improve the utilization of GPU hardware. The workload-kernel binding can be carried at a wavefront or work-group granularity. When applied at a wavefront granularity, a pair of Twin Kernels is assigned to the set of wavefronts. Each wavefront can choose one of the Twin Kernels to execute. Similar rules apply when working at a work-group granularity. We are not able to work at a work-item granularity since it would require a major overhaul of both hardware and software. Furthermore, executing different instructions for each work-item can reduce memory coalescing opportunities, which often lead to performance loss.

5.2.1 Memory Consistency

As an improvement to the SIMT model, the TKMT model preserves the current GPU memory consistency model, which is organized hierarchically as follows [25]:

1. Within a work-item, reads and writes to the same address are not reordered by the hardware.

2. For different work-items belonging to the same work-group, memory consistency is only guaranteed by using barrier operations.

3. Consistency is not guaranteed between different work-groups.

The TKMT model supports, and does not change, these rules. First, there is only one instruction order for a work-item, even though instructions are reordered across work-items, but TKMT does not change the in-order design of GPU hardware, so are no behavioral changes in terms of reads and writes.

Second, TKMT uses the same synchronization semantics as in the SIMT model. Memory barrier instructions may be inserted in different instruction slots. They are not removed, which can guarantee that memory consistency is supported within a work-group.

Third, we did not have to add any mechanism to TKMT in order to support consistency across work-groups. By continuing compatibility with the existing model, TKMT provides a lightweight solution in terms of software and hardware changes.
CHAPTER 5. TWIN KERNELS

![Workflow for the Twin Kernel Compiler](image)

**Table 5.1: Pre-RA and Post-RA instruction schedulers.**

<table>
<thead>
<tr>
<th></th>
<th>Pre-RA instruction scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>burr</td>
<td>Bottom-up register reduction list scheduling</td>
</tr>
<tr>
<td>source</td>
<td>Similar to list-burr but schedules in source order when possible</td>
</tr>
<tr>
<td>hybrid</td>
<td>Bottom-up register pressure aware list scheduling which tries to balance latency and register pressure</td>
</tr>
<tr>
<td>ilp</td>
<td>Bottom-up register pressure aware list scheduling which tries to balance ILP and register pressure</td>
</tr>
<tr>
<td>fast</td>
<td>Suboptimal but fast scheduler</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Post-RA instruction scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Register pressure aware</td>
</tr>
<tr>
<td>si</td>
<td>Back-ported from LLVM 3.9, designed for Southern Islands architecture</td>
</tr>
</tbody>
</table>

5.2.2 The Twin Kernel Compiler

To realize and evaluate the TKMT execution model, we implemented the Twin Kernel Compiler as an open source tool. The front-end and back-end in Twin Kernel Compiler are the same as M2C, which is detailed in Chapter 4. We extended the finalizer of M2C to support TKMT execution model, which is named the Twin Kernel Finalizer.

The Twin Kernel Compiler differs from regular compilers such as M2C. It can produce multiple binaries from a single source file, which is realized by using different instruction schedulers in the compiler backend. Instruction scheduling can take place both before and after register allocation (RA). The backend in M2C contains 5 pre-RA and 2 post-RA instruction schedulers, adopted from LLVM. The details of the instruction schedulers are presented in Table 5.1. Therefore, there can
be up to ten unique assembly files generated from a single source input. We refer to an individual assembly file as a Candidate Assembly.

Candidate Assembly files need to be converted to binary files and formatted to our target system. The binary executable files generated by our TKMT model, which we call Twin Kernel Binaries, are generated in two steps. The first step is a preprocessing performed by the Twin Kernel Finalizer. The Twin Kernel Finalizer is responsible for filtering, selecting and merging kernels, as well as performing some cleanup. To reduce redundancy, the Twin Kernel Finalizer uses permutations, rather than combinations, of the Candidate Assembly files. The Twin Kernel Combiner is specially designed for the TKMT model. If we skip the preprocessing step, the assembly files are produced for the traditional SIMT model. In the second step, the Twin Kernel Assembler takes the output files from Twin Kernel Finalizer and generates executables. These executables are in ELF format in which instructions and metadata information for our targeted system are encapsulated.

The workflow of the compiler is shown in Figure 5.3. The frontend takes a single OpenCL 1.2 kernel file as the input and works with the libclc library to generate the corresponding LLVM IR file. The libclc library is an open source library design of OpenCL that is designed to be used with Clang. The compiler backend generates a number of Candidate Assembly files using built-in instruction schedulers. Then the Candidate Assembly is processed by the Twin Kernel Finalizer. The output files are handed over to the Twin Kernel Assembler to generates a series of executable Twin Kernel Binaries.

5.2.3 The Twin Kernel Binary

GPU compute applications the use our TKMT model are no longer stuck with the monolithic kernel design of the SIMT model. To accommodate our changes without requiring any changes to the existing software stack, we adopt the structure of a standard kernel binary for our Twin Kernel Binary. The format for a Twin Kernel Binary is organized as follows:

1. The Twin Kernel Binary is encapsulated into a single ELF-formatted binary as a regular kernel binary. It has the same structure as the binaries for execution on the SIMT model.

2. An additional unconditional branch instruction is inserted before instructions of the First Twin Kernel. The patched First Twin Kernel is concatenated with the Second Twin Kernel, which is labeled as the destination of the unconditional branch instruction. This extra branch instruction is added to facilitate Twin Kernel Assignment, which is detailed later in Section 5.2.4.
3. As instruction scheduling may change the usage of registers, the candidate with heavier register usage is used for register information in the metadata section. Local memory usage is not affected.

Figure 5.4 gives an example of two Regular Binary and a Twin Kernel Binary. All three are generated from a Matrix Multiplication kernel. The two on the left are generated with different instruction schedulers. The Twin Kernel Compiler concatenates two sets of instructions and packs them into a Twin Kernel Binary, as shown on the right. In the Twin Kernel Binary, an extra branch instruction is added in line 0, which branches to the second kernel that starts at line 242. The vector and scalar register usage are set to the maximum of the two. As shown in Figure 5.4, the Twin Kernels Binary introduces no structural change to the regular kernel binaries. Therefore, there is no change in kernel loading in the software stack. The additional unconditional branch instructions introduce a small overhead, but they do not lead to dramatic changes in performance, as seen in standard thread divergence.

5.2.4 Twin Kernel Assignment

In our design, assigning different Twin Kernels to wavefronts or work-groups is done by setting the initial program counter (PC). Traditionally, a kernel begins execution with its PC set to 0, and all wavefronts/work-groups are assigned the same kernel. In our TKMT model and Twin Kernel Binaries, wavefronts/work-groups start with PC = 4 (the size of the extra branch instruction) and
executes the First Twin Kernel, while wavefronts/work-groups starting with PC = 0 are redirected to execute the Second Twin Kernel. PC initialization determines which Twin Kernel a wavefront/work-group executes. Switching to another Twin Kernel is not allowed during execution as it can cause program corruption.

We use a simple GreaterThan (GT) PC initialization strategy in our design. When TKMT is applied at wavefront granularity, the IDs of wavefronts are used to determine the initial PCs. For GT, if the ID of a wavefront is greater than a threshold, then its PC is initialized to 0, otherwise, it is set to 4. We also allow the ratio of the two PCs to be adjusted in order to provide more precise control. In addition, three other strategies are implemented for comparison purposes: LessThan (LT), Random (RD) and Round-Robin (RR). As the name suggests, LT is the opposite of GT. RD randomly initializes PCs, while RR initializes PCs to 0 or 4 in a round-robin fashion. Four examples are shown in Figure 5.5 to illustrate these strategies. From left to right, we have 4 strategies: GreaterThan (GT), LessThan (LT), Random (RD), and Round-Robin (RR). The ratio of the PCs for each assignment is adjustable, except for Round-Robin. We assume each work-group only contains one wavefront, and work-groups are scheduled to two CUs in a round-robin fashion. Eight wavefronts are assigned different PCs using the four strategies. Wavefronts, with the PC set to 4, execute the First Twin Kernel while wavefronts with the PC set to 0 execute the Second Twin Kernel. Similar mechanisms are implemented when the TKMT model is applied at a work-group granularity.

PC initialization is the only addition required to the system to support the TKMT model. For a GPU compute framework that exposes special registers to programmers, such as the %warp_id in CUDA, these strategies can be translated to a few extra instructions. The compiler can insert these instructions into the beginning of a kernel, following a similar approach that we used when adding the extra branch instruction. Instead, we choose a hardware design for the PC initialization due to architectural limitations in the modeled GPUs. As no bookkeeping is involved and the algorithm is not complex, our design adds only a very small amount of hardware real estate.
CHAPTER 5. TWIN KERNELS

5.2.5 The Twin Kernel Execution Engine

Figure 5.6: Performance of 5 instruction schedulers in the SIMT model, relative to the average performance of 10 instruction schedulers.

Given a number of Twin Kernel Binary and an adjustable mix ratio, we choose to use a performance tuning approach, rather than static analysis, to achieve the best configuration. The major reason is that we found no instruction scheduler can perform consistently better than the others.

Figure 5.6 illustrates the performance of five instruction schedulers on seven benchmarks. The performance is relative to the average performance of all instruction schedulers in our compiler and all benchmarks executed on the SIMT model. We can observe from the figure that an instruction scheduler that works very well for one benchmark may perform poorly on another. For example, the fast instruction scheduler can produce the fastest executable for the BS and FWT benchmarks. However, it also generates the slowest kernels for the BST, DWT, FW and MM benchmarks. The best instruction scheduler can be affected by many factors, such as the source code and targeted platform.

The decision is even more challenging to make in our TKMT model, as we have more binaries to choose from. To achieve the best performance, we need to exhaustively execute all combinations of Twin Kernel Binaries and mix ratios. To accelerate this process, we analyze the exhaustive execution results and make the following observations:

1. The two kernels in the Twin Kernel Binary should use GPU resources as efficiently as possible in order to run fast in SIMT mode. It is unlikely for two slow kernels to achieve high

59
CHAPTER 5. TWIN KERNELS

performance when working together.

2. Memory instructions in two Candidate Assembly files should appear at different lines to avoid memory bottlenecks imposed by the round-robin wavefront scheduler. The difference in line numbers is referred to as the Memory Distance. If a pair of memory instructions has a small memory distance in the assembly, most wavefronts will still issue long latency memory requests roughly at the same time and saturate the memory system.

3. Interleaving the usage of hardware units, such as scalar units, SIMD units, and branch units, can improve performance when we face stalls caused by insufficient capacity in these units.

4. It is common that one kernel in the Twin Kernel Binary dominates the performance, so adjusting the mix ratio does not make any difference. We can recognize this pattern by using Trail Execution: executing on three sequential or random mix ratios. If there is no performance difference, we can safely move to other Twin Kernels, otherwise, execution is performed on the rest of the mix ratios.

5. The two kernels in the Twin Kernel Binary may reduce, rather than enhance, the performance of the other kernel. We can also use Trail Execution to detect this pattern and terminate the execution of these combinations in order to save some time.

Based on these observations, we build an execution tool to automate and accelerate the tuning process. The execution tool can filter the unlikely combinations and prioritize execution on combinations with higher potential. The execution tool works in three steps: i) profiling, ii) scheduling and iii) execution. In the first step, we profile regular kernels executing in SIMT mode. Their collaboration potential is statically evaluated based on memory distance and instruction interleaving. In the second step, the profiling and evaluation information are considered together to generate an execution schedule. The Twin Kernel Binaries with a higher probability of producing a faster program are given higher priority in this schedule. Following the schedule generated in the previous step, our execution script performs Trail Execution and updates the record when a better combination is found. With our execution tool, the TKMT model is guaranteed to offer better or equivalent performance than the traditional SIMT model.
CHAPTER 5. TWIN KERNELS

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbreviation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinarySearch</td>
<td>BS</td>
<td>2097152</td>
</tr>
<tr>
<td>BitonicSort</td>
<td>BST</td>
<td>2048</td>
</tr>
<tr>
<td>DwtHaar1D</td>
<td>DH</td>
<td>65536</td>
</tr>
<tr>
<td>FastWalshTransform</td>
<td>FWT</td>
<td>262144</td>
</tr>
<tr>
<td>FloydWarshall</td>
<td>FW</td>
<td>128</td>
</tr>
<tr>
<td>MatrixMultiplication</td>
<td>MM</td>
<td>1024</td>
</tr>
<tr>
<td>MatrixTranspose</td>
<td>MT</td>
<td>1024</td>
</tr>
<tr>
<td>QuasiRandomSequence</td>
<td>QRS</td>
<td>16384</td>
</tr>
<tr>
<td>Reduction</td>
<td>RD</td>
<td>102400</td>
</tr>
<tr>
<td>ScanLargeArrays</td>
<td>SLA</td>
<td>131072</td>
</tr>
<tr>
<td>SimpleConvolution</td>
<td>SC</td>
<td>2048</td>
</tr>
</tbody>
</table>

Table 5.2: Benchmark Configuration

<table>
<thead>
<tr>
<th>GPU model</th>
<th>7770</th>
<th>7850</th>
<th>7870</th>
<th>7970</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num of CUs</td>
<td>10</td>
<td>16</td>
<td>20</td>
<td>32</td>
</tr>
<tr>
<td>L1 Cache/CU</td>
<td>16kB, 64 Sets, 4 way, 64 B/line</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td>128kB, 128 Sets, 16 way, 64 B/line</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registers/CU</td>
<td>2048(Scalar), 65536(Vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDS/CU</td>
<td>64kB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSHR/Cache</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler</td>
<td>Round-Robin</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Model Settings for the 4 GPUs evaluated in this work.

5.3 Evaluation

5.3.1 Methodology

We select 11 benchmarks from the AMD APP SDK v2.9 to evaluate our approach. The execution parameters of the selected benchmarks are presented in Table 5.2. All kernel binaries in our experiments are generated by the Twin Kernel Compiler. The results of our experiments have been verified against CPU implementations using the self-validation feature in these benchmarks.

We model four GPUs within the AMD Southern Islands architecture family using the Multi2Sim simulator [36], a cycle-level heterogeneous CPU-GPU simulator that supports the SIMT execution model by default. Details of the GPU configurations are described in Table 5.3. We extended the timing module in Multi2Sim and add four PC initializers to support our TKMT execution model. The default settings and GPU configurations provided in Multi2Sim are used in our experiments.

Our key figure of merit is the total number of cycles that it takes to execute a kernel.
CHAPTER 5. TWIN KERNELS

with each GPU model, where a smaller number is better. In our experiments, benchmarks have two execution settings: 1) SIMT mode, where benchmarks use the traditional SIMT model and execute regular kernel binaries, 2) TKMT mode, where we use our execution tool to tune benchmark performance. By default, we choose to work at a wavefront granularity and employ the GT strategy to initialize PCs in TKMT mode.

5.3.2 Results

5.3.2.1 Performance Improvements

As mentioned in Section 5.2.5, the choice of instruction scheduler can affect the performance in the SIMT model. Figure 5.7 shows the speedup for the best schedule for the SIMT model, relative to the worst performance achieved in the SIMT model. The best instruction scheduler can offer up to a 20% performance improvement, as seen in the Binary Search (BS) benchmark. On average, the best instruction scheduler can outperform the worst scheme by 8% across the four GPU configurations, when using the SIMT model.

To fairly compare the performance benefits that TKMT can afford, we compare its performance to the worst and the best performing scheduler in SIMT mode. Figure 5.8 illustrates the speedup of TKMT over the worst performance achieved in SIMT mode. The results in this
CHAPTER 5. TWIN KERNE\L{S}

Figure 5.8: Speedup of TKMT, relative to the baseline

Figure 5.9: Speedup of TKMT, relative to the best in SIMT
comparison represent an upper bound of the speedup that TKMT can achieve when compared to the SIMT model. TKMT is able to offer up to 1.45x speedup and we observe a 12.3% average performance improvement across all experiments.

The lower bound of speedup for TKMT is presented in Figure 5.9, where the best performance of SIMT is used as the baseline. The average performance improvement varies from 3% to 7% on different GPUs. In experiments such as MM on the 7770 and MT on the 7970, TKMT offers more than a 10% speedup. For the BS benchmark on the 7970, it achieves a 1.25x speedup. We analyzed the reasons for this using the Multi2Sim visual profiler and found the average response time for memory requests has been reduced by 45%, which shows congestion in the memory system is significantly reduced. Since memory requests are distributed more evenly in time, almost all stalls previously present in vector memory unit due to queuing are eliminated.

The upper bound and lower bound of speedup define a range of improvement that TKMT can offer over SIMT, which we refer to as Potential. The Potential of TKMT is presented in Figure 5.10. The Potential is calculated as the following:

$$Potential = 100 \times \frac{Cycle_{SIMT, \text{worst}} - Cycle_{IMT, \text{best}}}{Cycle_{TKMT}}$$

As shown in Figure 5.10 the overall Potential of TKMT is 8% across all platforms. The potential of each benchmark is largely dependent on the scope of the reordering that the instruction schedulers
can apply. For the BS, BST, MM and MT workloads, the Twin Kernel Compiler produces a diverse set of unique Candidate Assembly files, diverse in the sense that memory instructions have a variety of memory distances. In contrast, for the SLA workload, the choices to produce unique Candidate Assembly are limited and the memory distance is too small to significantly change the overall performance.

### 5.3.2.2 Impact of PC Initializer

![Figure 5.11: Evaluation of the impact of PC initialization on performance.](chart)

Our previous evaluations are based on systems with GT initialization. In this section, we evaluate the three other initializers that were described in Section 5.2.4. Figure 5.11 shows the results of running TKMT execution on a 7770 GPU with four PC initializers. The performance is relative to the worst performance in SIMT mode. As seen in the figure, the choice of PC initializer has little impact on the performance in general. The performance difference is within 3% for most benchmarks. There are a few exceptions, such as BS, MT, and RD benchmarks. However, no PC initializer always outperforms the others across all benchmarks. Considering the complexity of the mechanism, we prefer less complex ones and recommend GT to be used in future implementations.

### 5.3.2.3 Choice of Granularity

As described in Section 5.2, TKMT can work at a work-group or wavefront granularity. Intuitively, wavefront granularity is preferred, as it offers more precise control. We evaluate the performance of TKMT configured for a work-group granularity on the 7770. The relative speedup
Figure 5.12: Performance of TKMT while varying work-group granularity.

of TKMT with work-group granularity is shown in Figure 5.12, where the performance of TKMT with wavefront granularity is used as the baseline. As expected, we observe a slowdown for most benchmarks when TKMT works at a work-group granularity.

5.4 Summary

In this chapter, we studied a compiler-centric approach to improve scheduling of GPU hardware. We analyzed the impact of instruction scheduling in the traditional SIMT model and illustrated the importance of instruction schedulers in compilers. We proposed a novel TKMT execution model and constructed a full toolchain to evaluate our model. Our experiments using 11 benchmarks on 4 modeled GPUs show an average improvement of 12% (25% max) performance improvement over the SIMT model.
Chapter 6

Rebalanced Kernel

As technology advances, GPUs can support a greater number of compute units and an increasing amount of memory resources. However, as GPU hardware processing power continues to grow, it becomes harder for a single GPU kernel to fully utilize the available hardware \[75\][76]. A GPU kernel tends to exhaust a specific subset of the hardware resources (e.g., ALUs, memory bandwidth) and leave others under-utilized.

Besides the utilization imbalance of different hardware resources, the imbalance is also observed within the same hardware unit. Because of the GPU’s in-order execution design, a GPU kernel only accesses specific hardware units periodically during execution. However, when the hardware unit is in use, it is often over-subscribed. Overloaded hardware often results in stalls and extra processing overhead, which can be a major performance issue in GPUs \[105\][106].

In this chapter, we propose Rebalanced Kernel, a compiler-centric approach that produces GPU kernels that can achieve a better balance of hardware utilization. Rebalanced Kernel uses a two-stage approach that leverages optimizations in the front-end and back-end of the compiler. The first stage is a source code level optimization that improves the instruction diversity in the kernel binary. It balances the use of different hardware units in space. The second stage optimization fine-tunes the distribution of instructions, which balances the usage of the same hardware unit in time.

The remainder of the chapter is organized as follows. In Section 6.1, we present a static and dynamic characterization for a set of kernels and observe utilization imbalance present in these programs. We describe our Rebalanced Kernel approach, and provide implementation details in Section 6.2. In Section 6.3, we show how we can improve the performance of a GPU using Rebalanced Kernels.
CHAPTER 6. REBALANCED KERNEL

6.1 Motivation

With continued improvements in GPU hardware, these devices can handle heavier workloads with every new hardware release. A more powerful hardware platform often comes with much better performance. However, due to insufficient instruction diversity in a single GPU kernel, we typically cannot fully utilize the hardware resources of a GPU. The kernel stresses a subset of the hardware units, while leaving others under-utilized during execution.

To illustrate the utilization imbalance of GPU hardware, we profile 35 kernels on an AMD 7970 GPU with the CodeXL profiling tool [1]. The details of the kernels are listed in Table 6.1. The ALU and MEM utilization of these kernels are shown in Figure 6.1. Kernels are sorted by ALU utilization, in ascending order. As seen in the figure, the utilization of hardware is imbalanced and low in most kernels. Only a few benchmarks, such as Convolution2D and SimpleConvolution, achieve more than 80% utilization in both ALU and MEM. The average ALU utilization and MEM utilization are 17% and 73%, respectively. As ALU instructions are processed much faster than MEM instructions, the under-utilization issue is more evident in ALU units. We term the imbalance on different hardware units as Spatial Imbalance.

Utilization of a hardware unit is generally proportional to the number of instructions executed on the unit. The ratio of ALU and MEM instructions per work-item at runtime is shown in Figure 6.2. Kernels are sorted by ALU utilization in ascending order. Kernels on the right are the workloads with higher ALU utilization. As seen in Figure 6.2, they generally have a higher percentage of ALU instructions. The lack of ALU instructions in many application is a major reason...
## Chapter 6. Rebalanced Kernel

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Kernel Name</th>
<th>Benchmark Suite</th>
<th>GlobalSize1D</th>
<th>LocalSize1D</th>
</tr>
</thead>
<tbody>
<tr>
<td>atax_1</td>
<td>atax_kernel1</td>
<td>PolyBench</td>
<td>9216</td>
<td>256</td>
</tr>
<tr>
<td>atax_2</td>
<td>atax_kernel2</td>
<td>PolyBench</td>
<td>9216</td>
<td>256</td>
</tr>
<tr>
<td>bicg_1</td>
<td>bicg_Kernel1</td>
<td>PolyBench</td>
<td>8192</td>
<td>64</td>
</tr>
<tr>
<td>bicg_2</td>
<td>bicg_Kernel2</td>
<td>PolyBench</td>
<td>8192</td>
<td>64</td>
</tr>
<tr>
<td>binSrch</td>
<td>binarySearch</td>
<td>AMD APP SDK</td>
<td>524288</td>
<td>256</td>
</tr>
<tr>
<td>bitonicSort</td>
<td>bitonicSort</td>
<td>AMD APP SDK</td>
<td>8388608</td>
<td>256</td>
</tr>
<tr>
<td>conv2d</td>
<td>Convolution2D_kernel</td>
<td>PolyBench</td>
<td>67108864</td>
<td>256</td>
</tr>
<tr>
<td>conv3d</td>
<td>Convolution3D_kernel</td>
<td>PolyBench</td>
<td>3355432</td>
<td>256</td>
</tr>
<tr>
<td>corr_corr</td>
<td>correlation_corr_kernel</td>
<td>PolyBench</td>
<td>2048</td>
<td>256</td>
</tr>
<tr>
<td>corr_mean</td>
<td>correlation_mean_kernel</td>
<td>PolyBench</td>
<td>2048</td>
<td>256</td>
</tr>
<tr>
<td>corr_rdc</td>
<td>correlation_reduce_kernel</td>
<td>PolyBench</td>
<td>4194304</td>
<td>256</td>
</tr>
<tr>
<td>corr_std</td>
<td>correlation_std_kernel</td>
<td>PolyBench</td>
<td>2048</td>
<td>256</td>
</tr>
<tr>
<td>covar_rdc</td>
<td>covariance_reduce_kernel</td>
<td>PolyBench</td>
<td>65536</td>
<td>256</td>
</tr>
<tr>
<td>dwtHaar</td>
<td>dwtHaar1D</td>
<td>AMD APP SDK</td>
<td>32768</td>
<td>256</td>
</tr>
<tr>
<td>ftdt2d_1</td>
<td>ftdt2d_kernel1</td>
<td>PolyBench</td>
<td>8388608</td>
<td>256</td>
</tr>
<tr>
<td>ftdt2d_2</td>
<td>ftdt2d_kernel2</td>
<td>PolyBench</td>
<td>8388608</td>
<td>256</td>
</tr>
<tr>
<td>ftdt2d_3</td>
<td>ftdt2d_kernel3</td>
<td>PolyBench</td>
<td>8388608</td>
<td>256</td>
</tr>
<tr>
<td>fws</td>
<td>floydWarshallPass</td>
<td>AMD APP SDK</td>
<td>16777216</td>
<td>166</td>
</tr>
<tr>
<td>fwt</td>
<td>fastWalshTransform</td>
<td>AMD APP SDK</td>
<td>4194304</td>
<td>256</td>
</tr>
<tr>
<td>gemm</td>
<td>gemm_kernel</td>
<td>PolyBench</td>
<td>131072</td>
<td>256</td>
</tr>
<tr>
<td>gesummv</td>
<td>gesummv_kernel</td>
<td>PolyBench</td>
<td>4096</td>
<td>256</td>
</tr>
<tr>
<td>grscht_2</td>
<td>gramschmidt_kernel2</td>
<td>PolyBench</td>
<td>4096</td>
<td>256</td>
</tr>
<tr>
<td>grscht_3</td>
<td>gramschmidt_kernel3</td>
<td>PolyBench</td>
<td>4096</td>
<td>256</td>
</tr>
<tr>
<td>mm2_1</td>
<td>mm2_kernel1</td>
<td>PolyBench</td>
<td>65536</td>
<td>256</td>
</tr>
<tr>
<td>mm2_2</td>
<td>mm2_kernel2</td>
<td>PolyBench</td>
<td>65536</td>
<td>256</td>
</tr>
<tr>
<td>mm3_1</td>
<td>mm3_kernel1</td>
<td>PolyBench</td>
<td>131072</td>
<td>256</td>
</tr>
<tr>
<td>mm3_2</td>
<td>mm3_kernel2</td>
<td>PolyBench</td>
<td>65536</td>
<td>256</td>
</tr>
<tr>
<td>mm3_3</td>
<td>mm3_kernel3</td>
<td>PolyBench</td>
<td>131072</td>
<td>256</td>
</tr>
<tr>
<td>mvt_1</td>
<td>mvt_kernel1</td>
<td>PolyBench</td>
<td>8192</td>
<td>256</td>
</tr>
<tr>
<td>mvt_2</td>
<td>mvt_kernel2</td>
<td>PolyBench</td>
<td>8192</td>
<td>256</td>
</tr>
<tr>
<td>reduce</td>
<td>reduce</td>
<td>AMD APP SDK</td>
<td>4194304</td>
<td>256</td>
</tr>
<tr>
<td>splConv</td>
<td>simpleConvolution</td>
<td>AMD APP SDK</td>
<td>4194304</td>
<td>256</td>
</tr>
<tr>
<td>syr2k</td>
<td>syr2k_kernel</td>
<td>PolyBench</td>
<td>12828</td>
<td>256</td>
</tr>
<tr>
<td>syrk</td>
<td>syrk_kernel</td>
<td>PolyBench</td>
<td>65536</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 6.1: Details of Kernels
CHAPTER 6. REBALANCED KERNEL

Sufficient ALU instructions are also crucial in order to achieve efficient use of the memory system. As an in-order processor, a GPU can effectively hide the long latency of MEM instructions only when sufficient ALU instructions are present. When running low on ALU instructions, MEM instructions have to be scheduled close to each other. A tight schedule often results in memory bottlenecks due to over-saturation of the memory system. Compilers and wavefront schedulers can also make non-optimal choices that lead to over-subscription. Over-subscription is a major reason for a stalled pipeline, which accounts for most performance issues in GPUs [105][70]. Essentially, over-subscription is the imbalance of accesses distribution within a hardware unit, and we term this type of imbalance as Temporal Imbalance.

6.2 Rebalanced Kernel

The gap between ALU and MEM utilization is a Spatial Imbalance issue. Within hardware units, they may suffer from over-subscription, which is a Temporal Imbalance issue. We use two compiler-centric approaches to address these issues.

6.2.1 Addressing Spatial Imbalance with Kernel Fusion

Concurrent Kernel Execution (CKE) has been proposed to address the Spatial Imbalance issue in GPUs [107][108]. The idea is to run multiple GPU application instances at the same time, in the hope that spare hardware resources can be utilized. GPU programming models such as CUDA
CHAPTER 6. REBALANCED KERNEL

and OpenCL support this type of coarse-grained resource sharing, with support for multiple CUDA streams and OpenCL command queues.

Concurrent Kernel Execution relies on the diversity of instructions in kernels to improve utilization of hardware. The breakdown of instructions in the selected kernels is shown in Figure 6.3, demonstrating how instruction types vary across different kernels. If we can identify kernels that exercise a complementary set of instructions (i.e., exercising different GPU resources), the kernels may be able to run concurrently on a GPU, improving the utilization of all GPU hardware resources [74].

However, CKE is a coarse-grained solution that cannot guarantee improved concurrent execution of GPU kernels. Coarse-grained approaches such as CKE uses the left-over policy for resource allocation [109]. Under this policy, a GPU first fulfills the hardware resource requirement of one kernel, then allocates the spare hardware resources to schedule the others. Therefore, a GPU kernel can only overlap execution with another kernel at the end of its execution, when individual threads completed, and execution resources are no longer in use [74]. As a result, GPU hardware is occupied by a single kernel most of the time.

Kernel slicing can achieve better resource utilization than CKE [76]. It takes advantage of a property of CKE: when workloads from kernels are small enough to fit in a single together GPU together, they can execute concurrently and share the GPU. Kernel slicing divides a workload of a monolithic kernel into smaller pieces so that smaller slices from different kernels can run on the GPU at the same time. However, kernel slicing also makes no promises in terms of increasing GPU utilization. The dispatching order of slices is an undefined behavior [110]. Therefore, slices from
CHAPTER 6. REBALANCED KERNEL

different kernels may be misaligned and serialized.

As kernel slices are intended to consume only partial resources of a GPU, serialization of slices can result in an even poorer GPU utilization, as compared to the back-to-back single kernel execution. Besides, a single kernel execution requires multiple invocations of the slices, which not only introduces extra runtime overhead, but also makes it harder to align the execution.

Relying on hardware-based CKE does not give enough control to the programmer and cannot guarantee that we achieve a good mapping of kernels, and can lead to reduced performance. Implementing CKE in software can overcome most of these limitations. One approach that guarantees true resource sharing is kernel fusion [5]. Kernel Fusion is a static method that merges separate kernels into a single kernel. Kernel fusion can be performed vertically or horizontally.

Vertical Kernel Fusion prefers a producer-consumer relationship between the kernel pairs. Producer-consumer is a common pattern found in workloads such as HPC applications and Deep Neural Networks, where the output of one stage/layer is the input to the next stage/layer, Vertical Kernel Fusion is beneficial as the extra data movement and runtime overhead are eliminated [111][112]. However, vertical kernel fusion does not solve the Spatial Imbalance issue in GPU hardware, as kernel execution are essentially serialized in the fused kernel.

Horizontal Kernel Fusion works with independent kernels. It partitions running threads and assigns them to different kernels, providing spatial partitioning of GPU hardware. Within the same context, kernels with a producer-consumer relationship cannot be horizontally fused. However, horizontal fusion is useful when kernels are from different instances. When using the GPU hardware collaboratively, horizontal Kernel Fusion can effectively reduce Spatial Imbalance in GPUs.

Kernel Fusion often requires extensive reconstruction of the host programs and device kernels. To reduce programmer effort, we use different approaches to construct host and device programs. The host programs are not fused by Kernel Fusion. Instead, they are loaded separately by the Master Dispatcher. The construction of the kernel program is fully automated by the Kernel Merger.

6.2.1.1 Master Dispatcher

Figure[6.4] presents an overview of the Master Dispatcher, which is composed of a master thread and two worker threads. The master thread creates the two worker threads by loading the host programs from dynamic libraries. In our example in the figure, the master thread creates a worker thread 0 with app0.so, and a worker thread 1 with app1.so. The two worker threads are also loaded
CHAPTER 6. REBALANCED KERNEL

The master thread and worker threads have different responsibilities. The master thread is in charge of initializing shared resources, such as OpenCL platform/device/context/command queues. It is also in charge of all OpenCL program creation and kernel launching. The two worker threads do not execute any GPU kernel, they prepare data and kernel launch metadata, then forward the information to the master thread. They perform host-side tasks such as host/device buffer management, kernel argument configuration, and NDRange setup.

To avoid major reconstruction of the host program, most logical functions of the host programs remain unchanged. As the master thread takes care of the OpenCL initialization, these OpenCL API calls are removed from the host programs. Certain OpenCL function calls are replaced by the *Interception APIs*, that include `clBufferCreate`, `clSetKernelArg` and `clEnqueueNDRange`. Interception APIs capture the parameters of the corresponding OpenCL APIs and forward them to the master thread. For example, the OpenCL API function `clSetKernelArg` is replaced by the `interceptKernelArgs` function. The argument index, size, and pointer to the OpenCL memory object are captured and forwarded. As most changes involve deleting code or replacing APIs with equivalent calls, the reconstructed host program contains less code when compared to the original. The only addition to the host program is one line of code that sends an all-work-done notification, which needs to be explicitly added at the end of the host program for proper synchronization. Synchronization is essential in this master-worker design, which is detailed in the following example.

As an example, the co-execution of the Atax and Bicg benchmarks is shown in Figure 6.5. Atax and Bicg benchmark each have two GPU kernels. Timeline A is the baseline model, where all kernels are executed serially in a single process - no two tasks are executed at the same time.

The master thread launches two worker threads, which load GPU applications separately. The master thread then waits for ready signals from the worker threads. In this example, thread 0 and thread 1 are created by the master thread, which executes the Atax and Bicg applications,
respectively. The two GPU applications work independently in their own thread until they are ready for kernel launch. As seen from the timeline B, the atax_0 host program and the bicg_0 host program execute concurrently on their own at a different pace. At the end of host program execution, a ready signal is sent by the host program to the master thread. Then the worker thread polls for a ready signal from the master.

After receiving two ready signals, the master thread can choose one of the three execution patterns: 1) launch the fused kernel, 2) launch the two kernels in serial, 3) run one kernel and leave the other on hold. As seen in the timeline B, kernel atax_0 and kernel bicg_0 are fused to kernel atax_0_and_bicg_0, which is executed by the master thread. The host programs for atax_1 and bicg_1 are run in parallel, but kernel atax_1 and kernel bicg_1 are executed serially. In timeline C, kernel atax_1 and kernel bicg_0 are fused. As atax_1 has a prior dependency atax_0, the bicg benchmark has to wait until the host program of atax_0 and atax_1 both finish execution.

### 6.2.1.2 Kernel Merger

Combining kernels can be done at a work-item, wavefront, and work-group granularity [5]. At the work-item level, the fused kernel is constructed by concatenating the logical functionality of the kernel candidates. Essentially, it is a vertical Kernel Fusion. As a work-item executes both kernels, this method prefers the same work size for the two kernels. Otherwise, the execution of the kernels needs to be determined by conditional branches using work-item IDs. Conditional branches lead to wavefront divergence, which will result in a performance loss on a GPU.

Wavefront-level Kernel Fusion is a form of horizontal Kernel Fusion. Wavefront level Kernel Fusion does not suffer from additional divergence, as it associates all work-items in a
CHAPTER 6. REBALANCED KERNEL

wavefront with one of the kernels. All work-items within a wavefront can only execute one of the kernels. However, this method only works when kernels have no synchronization requirements. Otherwise, synchronization of partial wavefronts is needed, which is not supported by OpenCL.

Work-group level Kernel Fusion is free of these issues. In this method, wavefronts are associated with one of the kernels. Wavefronts can also synchronize and communicate with each other through local memory. Because of the simplicity and flexibility, we choose to combine kernels at a work-group granularity. As an example, an overview of work-group level Kernel Fusion is shown in Listing 6.1. Besides handling parameters for the two kernels, there are additional parameters in the merged kernel functions. These parameters are necessary additions for workload partitioning, identifier recalculation, and idle work-group elimination.

```c
__kernel void merged_kernel(orginal_kernel_0_params,
                         additional_kernel_0_params,
                         orginal_kernel_1_params,
                         additional_kernel_1_params,
                         additional_misc_params) {
  if(condition(work-group id))
    // Sub-kernel 0
    kernel_0_source_code
  else
    // Sub-kernel 1
    kernel_1_source_code
}
```

Listing 6.1: Workgroup Level Kernel Fusion

**Work-group Partition**  Work-groups are dispatched to GPU compute units in a prioritized round-robin order [113], until the hardware is fully occupied. Therefore, by controlling the number of running work-groups from each of the two kernels, we can achieve effective resource sharing on a GPU. This is a true sharing of hardware, as the work-groups are guaranteed to overlap their execution by the execution model.

The overview for work-group partition is shown in Listing 6.2. To control the partition ratio, additional parameters are needed. Parameter ap_M is the number of compute units of the GPU device, which can be obtained using OpenCL APIs. Parameter ap_N is the number of compute units allocated for one kernel, which is configurable at run time. These two parameters are the additional_misc_parameters in Listing 6.1.
CHAPTER 6. REBALANCED KERNEL

Listing 6.2: Work-group Partition

<table>
<thead>
<tr>
<th>K_0: 3/8</th>
<th>K_1: 5/8</th>
<th>K_0: 4/8</th>
<th>K_1: 4/8</th>
<th>K_0: 1/8</th>
<th>K_1: 7/8</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>id</td>
<td>num</td>
<td>id</td>
<td>num</td>
<td>id</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

(a) 3:5 Partition  (b) 4:4 Partition  (c) 1:7 Partition

Figure 6.6: Work-group Partition

Figure 6.6 shows how partitioning works. Assuming a GPU with 8 compute units and a round-robin work-group dispatching policy, when the parameter ap_N is 3 and ap_M is 8, the work-groups are partitioned using a ratio of 3:5, as shown in Figure 6.6a. When setting ap_N to 4 and 1, the work-groups are partitioned with a ratio of 4:4 and 1:7, respectively, as shown in Figure 6.6b and Figure 6.6c. Similar to the setting of the kernel parameters, ap_N and ap_M can be easily adjusted through OpenCL APIs.

Logical Identifier Recalculation

By setting kernel parameters, the work-groups are split into two parts. Each kernel is assigned their own set of work-groups. However, the identifiers for the
work-groups in each kernel are no longer valid after partitioning. On the left of Figure 6.7, the work-groups for the two kernels have identifiers assigned by the hardware. These identifiers are clearly different from the identifiers expected by the two kernels, which are shown on the right. In this work, the identifier assigned by the hardware is named the physical identifier. The identifier expected by the kernel is named the logical identifier. We use the approach from Wen et al. to rebuild the mapping [110].

In standard single kernel execution, the physical identifier is the same as the logical identifier. This one-to-one mapping is broken because of the merged identifier space. The pseudo code for the fix is shown in Listing 6.3:

```
// Current level
int currLevel = get_group_id(0) / ap_M

// Work-group ID recalculation
int k0_logical_wgID = get_group_id(0) - currLevel * (ap_M - ap_N)
int k1Logical_wgID = get_group_id(0) - (currLevel + 1) * ap_N
```

Listing 6.3: Identifier Recalculation: Logical Work-group

We will also need to adjust parameters for built-in OpenCL functions. Within a workgroup, work-items can obtain identifiers through OpenCL built-in functions that include get_global_id, get_group_id, and get_local_id. Similar to work-group id, these physical identifiers are no longer valid after merging kernels. These OpenCL built-in functions need to be replaced in all sub-kernels by the recalculated values.

Recalculation of these identifiers requires the logical work-group identifier, as well as the global/local size of the two kernels when running separately. The global/local sizes of the two kernels are passed in additional parameters, as shown in Listing 6.1. The pseudo code for the recalculation of these identifiers is described in Listing 6.4:

```
// Variable starts with 'ap' are additional parameters in kernel function
int logical_gid = recalculated_wgID * get_local_size(0) + get_local_id(0)
if(logical_gid < ap_global_size_0 * ap_global_size_1) {
    // linearized identifiers in logical grid
    int group_id = logical_gid / (ap_local_size_0 * ap_local_size_1)
    int local_id = logical_gid % (ap_local_size_0 * ap_local_size_1)
}

// logical work-group identifier
int get_group_id_0 = group_id % (ap_global_size_0 / ap_local_size_0)
```
CHAPTER 6. REBALANCED KERNEL

```c
int get_group_id_1 = group_id / (ap_global_size_0 / ap_local_size_0)

// logical local work-item identifier
int get_local_id_0 = local_id % ap_local_size_0
int get_local_id_1 = local_id / ap_local_size_0

// logical global work-item identifier
int get_global_id_0 = get_group_id_0 * ap_local_size_0 + get_local_id_0
int get_global_id_1 = get_group_id_1 * ap_local_size_1 + get_local_id_1
```

// Original kernel source code, with OpenCL built-in function replacement
...
```
Listing 6.4: Identifier Recalculation: OpenCL built-in functions

Figure 6.8: Idle Work-groups

(a) Partition Imbalance  
(b) Workload Imbalance  
(c) Both Imbalance

Idle Work-group Elimination  Work-group partitioning may result in a very large number of idle work-groups, which not only introduces additional overhead, but also leads to a low hardware utilization. Restricted by the maximum work-items supported by a GPU, the idle work-groups might result in an out of resource kernel launching failure. As seen in Figure 6.8a, a large number of work-groups in kernel 0 are idle because of partition imbalance, even though the two kernels have similar workload demands. When the partition is equal, workload imbalance can also lead to idle work-groups, which is shown in Figure 6.8b. An extreme case is shown in Figure 6.8c, where both the partitioning and workload are out of balance. As a result, the majority of the work-groups are idle during kernel execution.
CHAPTER 6. REBALANCED KERNEL

Figure 6.9: Eliminate Idle Work-groups

Idle work-groups can be fully eliminated when they are re-assigned to executing the other kernel. As seen in Figure 6.9, the 14 work-groups belonging to kernel 1 are idle. After migrating some of the workload from kernel 0, the idle work-groups in kernel 1 no longer exist. The level/row of the migration can be obtained with the algorithm described in Listing 6.5. In the example in Figure 6.9 without re-assignment, kernel 0 needs 6 rows of work-groups, while kernel 1 needs 4 rows. Therefore, the migration takes place at the 4th level/row.

Listing 6.5: Level of Migration

```
//Maximum level needed, when there is no re-assign
int k0_maxLevel = (k0_countWGs + ap_N - 1) / ap_N - 1
int k1_maxLevel = (k1_countWGs + ap_M - ap_N - 1) / (ap_M - ap_N) - 1

//Level of potential work-group migration
int levelMigration = min(k0_maxLevel, k1_maxLevel)

//Potential work-group migration
if (currLevel <= levelMigration) {
  //Work-group Migration Fix
} else {
  //Work-group Below Migration Fix
}
```

With work-group migration, a work-group can have two roles: 1) executing the assigned kernel, 2) change their role to execute the other kernel. The role of the work-group changes when its logical identifier surpasses the maximum number of work-groups in its associated kernel. Pseudocode for the algorithm to switch roles is shown in Listing 6.6.
// Flattened local size in merged kernel
int localSize1D = get_local_size(0) * get_local_size(1)

// Count of work-groups, using additional kernel parameters
int k0_count_wgs = (ap_k0_globalSize_0 * ap_k0_globalSize_1) / localSize1D

// Role changes when the logical work-group id surpasses the maximum
if(k0_logical_wgID < k0_count_wgs) {
    // kernel 0 with identifier fixes
} else {
    // Calculation to get new_group_id
    // kernel 1 with identifier fixes
}

Listing 6.6: Work-group Migration

When the role of a work-group changes, a recalculation of its logical identifier is required. The new logical work-group identifier is calculated differently for the two kernels. The kernel in the left partition obtains the new identifier using equation 6.1, while the kernel on the right uses equation 6.2. The updated logical work-group identifier is used to calculate the logical global identifier(\text{logical\_gid}) in Listing 6.4.

\begin{align}
    k0\_new\_group\_id &= k0\_wg\_id - k0\_countWGs + (curr\_Level + 1) \times (ap\_M - ap\_N) \\
    k1\_new\_group\_id &= k1\_wg\_ID - k1\_countWGs + (curr\_Level + 1) \times ap\_N
\end{align}

Work-groups below the migration level are also affected. With work-group migration, these work-groups execute the remaining work-groups of the kernel assigned the higher workload. In the example on the right of Figure 6.9, the work-groups below the migration row have logical identifiers 16 and 17. If no migration occurs, they should be assigned 12 and 13, as shown on the left of this figure. As work-groups 12 through 15 have been migrated to the 4th row, the logical identifiers of these work-groups need to be shifted. Listing 6.7 provides pseudocode to produce the new work-group identifier.

if (k0_maxLevel > k1_maxLevel) {
    // Execute the rest of kernel 0
    int group_id_offset = get_group_id(0) % ap_M
    int group_id_base = currLevel * ap_M - k1_countWGs
    int new_group_id = group_id_base + group_id_offset
}

80
CHAPTER 6. REBALANCED KERNEL

// Kernel 0 with identifier fixes follows
} else {
  // Execute the rest of kernel 1
  int group_id_offset = get_group_id(0) % ap_M
  int group_id_base = currLevel * ap_M - k0_countWGs
  int new_group_id = group_id_base + group_id_offset

  // Kernel 1 with identifier fixes follows
}

Listing 6.7: Fix Work-group Identifier below Migration Level

Summary  In summary, we need to rebuild the logical identifier of work-groups based on their role in the fused kernel. For work-groups above the migration level, when they execute the assigned kernel, their logical identifiers are recalculated with the mechanism described in Listing 6.3. This applies to work-groups 0 through 27 in Figure 6.10a.

For work-groups that are switching roles at the migration level, their identifiers are recalculated. Work-groups belonging to Kernel 0 use equation 6.1, and work-groups belonging to Kernel 1 use equation 6.2. This applies to work-groups 28 through 31 in Figure 6.10a. These work-groups are initially assigned to execute Kernel 1. As there is no more work to do for Kernel 1, they switch roles and work on Kernel 0. They are on the right side of the partition, so their logical identifiers are recalculated using equation 6.2.

Figure 6.10: Work-group Identifier: Physical to Logical

81
CHAPTER 6. REBALANCED KERNEL

Below the migration level, all work-groups execute the same kernel. For these work-groups, even if there is no role changing, their identifiers are affected by the migration above, and thus need adjustment. Their identifiers are recalculated following Listing 6.7 which applies to work-groups 32 and 33 in Figure 6.10a.

The rectified work-group identifier is used to recalculate the logical global identifier and the replacement for the OpenCL built-in functions, as described in Listing 6.4. Because of the elimination of idle work-groups, we only need to consider the sum of work-items in the two individual kernels when launching the fused kernel.

The work-group partitioning and ID recalculation add overhead when compared to the single kernel execution. However, the benefits achieved from co-running kernels outweighs this overhead, and we achieve an overall system performance improvement. To minimize the impact of the overhead, we optimize the execution of the fused kernel with Twin Kernels.

6.2.2 Addressing Temporal Imbalance with Twin Kernels

Kernel Fusion generates a single kernel that contains the logical functionality of the two kernels. It balances instructions of different types to reduce Spatial Imbalance. As a by-product of the diversified instructions, Temporal Imbalance can be mitigated at the same time. However, just as in other single kernels, the fused kernel may still suffer from over-subscription of GPU hardware from time to time.

Our work in Chapter 5 provides a detailed discussion on the over-subscription issue and offers the Twin Kernels solution. Performed at different stages of compilation, Twin Kernels can work seamlessly with Kernel Fusion. Kernel Fusion works at the front-end, while Twin Kernels works at the back-end. They solve GPU performance problems from a different perspective. Kernel Fusion remedies instruction imbalance and addresses Spatial Imbalance, while Twin Kernels remedies the periodical over-subscription problem and addresses Temporal Imbalance.

6.2.3 Rebalanced Kernel

Producing a rebalanced Kernel is a two-stage optimization that combines Kernel Fusion and Twin Kernels. The workflow is shown in Figure 6.11. The source code of the two kernels is consumed by the Kernel Merger to produce the fused kernel. The fused kernel can run on real GPUs and Multi2Sim with the Master Dispatcher. Then the fused kernel is compiled by the Twin Kernel
Compiler, which produces a set of Twin Kernel Binaries. These binaries can run on Multi2Sim using the Master Dispatcher.

Although the final binaries are statically generated, Kernel Fusion and Twin Kernels both require parameter tuning at runtime. For Kernel Fusion, the work-group partition ratio is tuned at execution time. Multiple invocations of the fused kernel are required to get the best partition balance. For Twin Kernels, the auto-tuning engine finds the best Twin Kernel Binary and the best mix ratio. Both parameters are crucial if we want to achieve optimized performance on GPUs [110][114].

### 6.3 Evaluation

#### 6.3.1 Methodology

##### 6.3.1.1 Benchmarks

We select 20 benchmarks from AMD APP SDK and PolyBench, which contain 35 kernels in total. The details of the kernels are listed in Table 6.1. The workload of kernels is configured to fully occupy the GPU when running alone.

##### 6.3.1.2 Platforms

The first stage of Rebalanced Kernel is evaluated on a system with an AMD FX-8320 CPU and an AMD 7970 GPU. The details of the GPU is shown in Table 6.2. Due to architectural limitations, experiments in stage 2 are carried out on the Multi2Sim simulator, a cycle-level CPU-GPU heterogeneous microarchitecture simulator. The simulator is configured to model the same AMD 7970 GPU used in stage 1.
CHAPTER 6. REBALANCED KERNEL

| AMD 7970 |
|-----------------|----------------|
| Compute Unit    | Count: 32 |
|                 | Clock: 1000 MHz |
| Registers / CU  | Scalar: 2048 |
|                 | Vector: 65536 |
| Cache           | L1 / CU: 16 kB, 64 Sets, 4 way, 64 B/line |
|                 | L2: 128 kB, 128 Sets, 16 way, 64 B/line |
| Local Memory    | LDS / CU: 64 kB |
| Global Memory   | Memory Size: 3072 MB |
|                 | Memory Clock: 1500 MHz |
|                 | Bus type: GDDR5 |
|                 | Bus Width: 384-bit |
|                 | Bandwidth: 288 GB/s |

Table 6.2: Platform: GPU

6.3.1.3 Metrics

The evaluation metric for the first stage is System Throughput (STP) \[115\], which is also known as the Weighted Speedup. In our results, we substitute kernel execution time for cycles in the equations for STP. The kernel execution time is measured by the OpenCL event. STP is a higher-is-better metric.

\[
STP = \frac{\text{ExecTime}(\text{Kernel}_0 + \text{Kernel}_1)}{\text{ExecTime(\text{Kernel}_\text{Fused})}}
\]

The evaluation metric for the second stage is Instructions Per Cycle (IPC), which is reported by the Multi2Sim simulator. In our experiments, each simulation runs for 400K cycles, using IPC of this 400K cycles for evaluation. IPC is also a higher-is-better metric.

6.3.2 Performance Improvement: Stage 1

In this section, we evaluate Kernel Fusion, the first stage of the Rebalanced Kernel. We identify kernel pairs that that benefit from Kernel Fusion, which account for 58% of all the kernel pairs.

Instruction Diversity In Section 6.1 we have shown that presence of insufficient ALU instructions in a kernel is a major contributor to the low-ALU utilization. The static analysis shows that merging kernels can effectively improve the ratio of ALU instructions. Figure 6.12 provides a static analysis of the changes in the ratio of instruction types. As seen in the figure, the ratio of SIMD instructions
increases to 76%, which is 15% more than the average ratio in the single kernels. As a result, the ratio of VMEM instructions drops from 6% to 2%. A lower ratio of MEM instructions is generally beneficial to performance, as it means a smaller chance of a constrained instruction schedule.

Spatial Imbalance The increased share of ALU instruction translates to a more balanced use of hardware units. We use the Utilization Gap (UG) metric to represent the degree of Spatial Imbalance. UG is a lower-is-better metric, which is calculated as follows:

\[ UG = |Utilization_{ALU} - Utilization_{MEM}| \]

We compare the UG of the fused kernel pairs to the maximum UG of the two kernels. The average reduction of UG is 17.8%, meaning the gap between ALU and MEM utilization is reduced by 17.8% on average. A reduced UG is a strong indicator of reduced Spatial Imbalance on GPUs.

STP Improvement Achieving more balanced resource utilization should improve GPU performance. The performance of the GPU is evaluated using STP, as described in Section 6.3.1.3. Overall, the average STP for the kernel pairs is 1.24, where the maximum STP obtained is 2.17.

Figure 6.13 presents an analysis of the reasons resulting in better performance. ALU Util, MEM Util, and Cache Hits are higher-is-better metrics, while MEM Stalls and Write Stalls are lower-is-better metrics. Kernel performance improves due to one or more of these reasons. When averaging values for the two kernels as the baseline, 43% of the kernel pairs benefit from improved ALU utilization, and 52% benefit from a better MEM utilization.
CHAPTER 6. REBALANCED KERNEL

Figure 6.13: Reasons for Improvement

Kernel Fusion not only reduces Spatial Imbalance, but also decreases Temporal Imbalance. When taking a closer look at the memory system, reduced stalls in the MEM and Write Unit are observed in 39% and 44% of the kernel pairs, respectively. We also find that 66% of the kernel pairs benefit from the improved cache hit rates. These improvements suggest a memory system that is less congested and more efficient.

Kernel Fusion improves the performance of the GPU from a system perspective. It does not necessarily translate to an improvement of these metrics when compared to the best performance in single kernel execution. We find that 27% of fused kernel out-perform the single kernel execution in terms of ALU utilization. 13% of the fused kernel experiment has better MEM utilization, and 23% achieve better cache performance. Fewer MEM stalls and Write stalls are observed in 9% and 1% of the kernel pairs, respectively.

6.3.3 Performance Improvement: Stage 2

In this section, we evaluate both stages of running a Rebalanced Kernel, where Kernel Fusion is applied first, followed by Twin Kernels. Experiments in stage 2 are designed to explore the performance upper-bound that the hardware can offer. Constrained by the support and speed of the simulator, experiments in this stage are also less extensive than in stage 1. The kernel pairs are narrowed down to a subset of the kernel pairs. We also limit experiments to the kernel pairs that achieved the most STP during stage 1 optimization.

86
Although Stage 1 optimization mainly focused on reducing the Spatial Imbalance, the improved instruction diversity also helps to alleviate the over-subscription issue. The temporal improvements generated by stage 1 optimization leaves a limited room for further reduction of Spatial Imbalance. However, stage 2 optimization still results in improvements for some kernel pairs. For a fair comparison, we compare the IPC to the best and worst IPC achieved in the baseline SIMT execution model, following our evaluation methodology outlined in Chapter 5. When compared to the worst-case SIMT execution model, we can achieve up to a 17.5% improvement. When compared to the best SIMT execution model, there is a smaller 2.5% improvement. On average, there is a 1.2% to 4.8% IPC improvement on selected kernels pairs. The details of these kernel pairs are shown in Figure 6.14.

To evaluate the effectiveness of our approach and consider how close our implementation is to an ideal system, we compare the IPC achieved to the theoretical upper-bound. The theoretical IPC upper-bound is obtained on an ideal system, where the memory latency is configured to be a single cycle on Multi2Sim. On average, we find we can achieve up to 92% of the theoretical IPC when leveraging Rebalanced Kernels.

6.3.4 Further Discussion

The focus of our work is to achieve peak performance on GPU systems by a more balanced use of GPU hardware. The most performance benefit is obtained from Kernel Fusion, the first stage of our Balanced Kernel methodology. However, kernel pairs do not always benefit from Kernel Fusion. Co-running selected kernel pairs always results in STP improvement, despite the work-group
partition ratio. However, the choice of this ratio is crucial for the other pairs. These kernel pairs only benefit from co-execution when we find the right balance of work-groups. In our experiments, 50% of kernel pairs fell into this category. We classify the former as *Unconditional Beneficial Pairs* and the latter as *Conditionally Beneficial Pairs*. Kernel pairs cannot benefit from co-execution is termed as *Non-Beneficial Pairs*, which account for 42% in our experiment. The breakdown of the kernel pairs is summarized in Figure 6.15.

The selection of the kernel pairs and the partition ratio are crucial to achieve improved performance. For new sets of kernels, it is helpful to have a performance model for selecting kernel pairs and their partition ratio. A predictive model requires a set of representative features to make an accurate prediction. Features are usually selected from dynamic profiling or static analysis. As performance on half of the kernel pairs is highly sensitive to runtime choices, the static information is not rich enough for such decision making. Prior work based on CKE made attempts on building a performance model. Dynamic profiling information obtained from profiling tools [76][74] is used in their work. However, constrained by tools, the metrics for building this same kind of detailed performance model on our platform is limited. Without enough representative information, our predictive model only achieves a 61% prediction accuracy. We plan to enhance the feature set to improve the performance model in the future.
CHAPTER 6. REBALANCED KERNEL

6.3.5 Predictive Modeling

In our experiments, we found that 58% of kernel pairs could potentially benefit from Kernel Fusion. To find the best pairs we performed an exhaustive search through all kernel pairs; developing a predictive model to select kernel pairs is highly desirable in order to reduce Kernel Fusion to practice. Essentially, the selection of kernel pairs is a binary classification problem, as we only have two choices for a given pair of kernels. To develop this model, we have explored supervised learning algorithms to build an accurate predictive model. Supervised learning algorithms require training data to tune the model, which is composed of label and features.

6.3.5.1 Training Data

6.3.5.2 Label

To find candidate kernel pairs using supervised machine learning, we need to provide accurate training data with labels. We label kernel pairs based on the results of our exhaustive search: if merging the two kernels resulted in an improved STP, the label for this kernel pair is 1; otherwise, the label is 0.

6.3.5.3 Features

To capture the execution characteristics of the kernels, static and dynamic features are used as features. The static features are extracted by analyzing the assembly of the kernels, while the dynamic features are obtained by profiling the kernels using the CodeXL profiling tools [116] and Multi2Sim [36].

The static features obtained are based on instruction counts in the AMD GCN1 assembly code. In AMD GCN1 ISA, there are 6 different types of instructions:

1. Scalar ALU
2. Scalar MEM
3. LDS
4. Branch
5. SIMD
6. Vector MEM
CHAPTER 6. REBALANCED KERNEL

Therefore, we use 6 instruction counters to capture static features for each kernel binary. We also consider a coarser classification, dividing these classes into ALU versus MEM instruction. In this case, each kernel has 2 static features.

Dynamic profiling metrics provided with CodeXL and Multi2Sim are used as dynamic features in our classification model. CodeXL can provide coarse-grained dynamic statistics, that include ALU utilization, MEM utilization, MEM stall ratio, and cache hit ratio. To capture additional details, we also use Multi2Sim, a cycle-based execution simulator, to capture stall counters and utilization of each execution unit. We also collect information such as global memory usage, local memory usage, and global work size for each kernel using the runtime APIs.

To make our static and dynamic features appropriate for training, we first normalized the values. The features have a wide range of values, especially the counter-based features. These counter-based features are grouped and scaled down. The details of the static and dynamic features collected are summarized in Table 6.3. We generate this set of features for each kernel and combine them as a feature vector.

6.3.5.4 Quality of the Features

A predictive model relies on leveraging representative features, in order to produce an accurate prediction. For binary classification problems, an ideal feature should appear as two disjoint groups of data points. To better understand the quality of the selected features, we plot them in pairs in Figure 6.16. In this figure, the diagonal axis shows the distribution of the two classes for each feature.

Figure 6.16 shows the static features used in our model, that includes the ratio of ALU and MEM instruction in individual kernels. The kernel pairs that benefit from Kernel Fusion are colored green, while the kernel pairs experiencing a decrease in performance are colored blue. As seen in the figure, it is hard to separate the kernel pairs with the static features selected. The two classes of kernel pairs exhibit nearly the same distribution.

One logical approach to improve performance is to combine a compute-intensive kernel with a memory-intensive kernel. To implement this heuristic, we should be able to use the ALU and MEM statistics to identify the best kernel pairs. However, previous studies have shown that the compute-memory pairing strategy can result in performance degradation in many cases \[110\]. A more recent study has shown that the performance for a merged compute-intensive kernel with a memory-intensive kernel can be even lower than when combining 2 memory-intensive kernels \[80\].
**CHAPTER 6. REBALANCED KERNEL**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>bench_ldsh_stall</td>
<td>LDS stall counter(normalized)</td>
<td>Multi2Sim</td>
</tr>
<tr>
<td>bench_sclr_stall</td>
<td>Scalar stall counter(normalized)</td>
<td>Multi2Sim</td>
</tr>
<tr>
<td>bench simd_stall</td>
<td>SIMD stall counter(normalized)</td>
<td>Multi2Sim</td>
</tr>
<tr>
<td>bench_vmem_stall</td>
<td>VMEM stall counter(normalized)</td>
<td>Multi2Sim</td>
</tr>
<tr>
<td>bench_vmem_dvg</td>
<td>VMEM divergence counter(normalized)</td>
<td>Multi2Sim</td>
</tr>
<tr>
<td>bench_ipc</td>
<td>IPC(normalized)</td>
<td>Multi2Sim</td>
</tr>
<tr>
<td>bench_alu_insts</td>
<td>dynamic ALU instruction counter(normalized)</td>
<td>CodeXL</td>
</tr>
<tr>
<td>bench_mem_insts</td>
<td>dynamic MEM instruction counter(normalized)</td>
<td>CodeXL</td>
</tr>
<tr>
<td>bench_valu_util</td>
<td>VALU utilization</td>
<td>CodeXL</td>
</tr>
<tr>
<td>bench_salu_util</td>
<td>SALU utilization</td>
<td>CodeXL</td>
</tr>
<tr>
<td>bench_cache_hit</td>
<td>cache hit ratio</td>
<td>CodeXL</td>
</tr>
<tr>
<td>bench_mem_util</td>
<td>memory utilization</td>
<td>CodeXL</td>
</tr>
<tr>
<td>bench_mem_hit_ratio</td>
<td>memory stall ratio</td>
<td>CodeXL</td>
</tr>
<tr>
<td>bench_glbsz_ratio</td>
<td>individual global work size / combined global work size</td>
<td>Runtime API</td>
</tr>
<tr>
<td>bench_glb_datasz_ratio</td>
<td>individual global memory / combined global memory</td>
<td>Runtime API</td>
</tr>
<tr>
<td>bench_loc_datasz_ratio</td>
<td>individual local memory / combined local memory</td>
<td>Runtime API</td>
</tr>
<tr>
<td>bench_salu_ratio</td>
<td>SALU instruction / all instructions</td>
<td>Assembly</td>
</tr>
<tr>
<td>bench_smem_ratio</td>
<td>SMEM instruction / all instructions</td>
<td>Assembly</td>
</tr>
<tr>
<td>bench_brch_ratio</td>
<td>Branch instruction / all instructions</td>
<td>Assembly</td>
</tr>
<tr>
<td>bench_valu_ratio</td>
<td>VALU instruction / all instructions</td>
<td>Assembly</td>
</tr>
<tr>
<td>bench_lds_ratio</td>
<td>LDS instruction / all instructions</td>
<td>Assembly</td>
</tr>
<tr>
<td>bench_vmem_ratio</td>
<td>VMEM instruction / all instructions</td>
<td>Assembly</td>
</tr>
<tr>
<td>bench_mem_ratio</td>
<td>MEM instruction / all instructions</td>
<td>Assembly</td>
</tr>
<tr>
<td>bench_alu_ratio</td>
<td>ALU instruction / all instructions</td>
<td>Assembly</td>
</tr>
</tbody>
</table>

Table 6.3: Summary of all static and dynamic features.
Figure 6.16: Pair-wise kernel plot of using static features only.
Figure 6.17: Pair-wise kernel plot of using the dynamic features from CodeXL.
CHAPTER 6. REBALANCED KERNEL

Due to the interference, the memory-intensive kernel may dominate the usage of the memory pipeline, which can cause significant delays in memory requests required in the compute-intensive kernels, stalling the compute operations. It is not always beneficial to merge compute-intensive and memory-intensive kernels. Therefore, ALU and MEM utilization of the individual kernels cannot fully capture if it will be beneficial to fuse two kernels.

Figure 6.17 shows the dynamic features obtained with CodeXL, which include ALU and MEM utilization of the kernels when running alone. Dynamic features offer better separation than just using the static features. As seen in the scatter plots, there is less overlap. However, the two classes still have a similar distribution, which makes it difficult to separate them.

Figure 6.18: Pair-wise plot of the dynamic features from CodeXL - removing Conditionally Beneficial kernel pairs.
CHAPTER 6. REBALANCED KERNEL

One contributor to the poor separation using ALU and MEM utilization is the work-group partition optimization in our Kernel Fusion approach. In our experiments, more than half of kernels pairs require fine-tuning of their work-group assignments (i.e., the portion of the compute units assigned to a specific kernel) in order to produce performance improvements. If we remove these Conditionally Beneficial kernel pairs from the training data, the ALU and MEM utilization shows better separation. As seen in Figure 6.18, the distributions are no longer fully overlapped, and there is some separation in the scatter plot figures. However, the distributions overlap if the Conditionally Beneficial kernel pairs are added. Similar patterns are observed in other features obtained from CodeXL and Multi2Sim.

6.3.5.5 Algorithms and Results

We use several classifiers in the scikit-learn machine learning libraries to drive our binary prediction model [117]. The choices include Nearest Neighbor, Linear Support Vector Machines (SVMs), Radial Basis Function (RBF) SVMs, Decision Trees, and Neural Networks.

In our experiments, we use 5-fold cross-validation during the training. To find the best algorithm for identifying kernel pairs to fuse, we iterate over these algorithms and train with our training data. The default parameters in the scikit-learn library are used for these algorithms.

Predicting the performance of fused kernel candidates is a non-trivial task. As seen in our feature analysis, there is no clear boundary in most figures, which makes it difficult for any of the candidate algorithms to produce an accurate classification. A detailed comparison is shown in Figure 6.19. In our experiments, a Neural Network model proved to be the most accurate predictor.
CHAPTER 6. REBALANCED KERNEL

The parameters of the Neural Network are provided in Table 6.4. However, the Neural Network based predictive model only achieves a 61% prediction accuracy.

<table>
<thead>
<tr>
<th>Param Name</th>
<th>Value</th>
<th>Param Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hidden_layer_sizes</td>
<td>100, 100, 50, 30, 10</td>
<td>learning_rate</td>
<td>constant</td>
</tr>
<tr>
<td>activation</td>
<td>relu</td>
<td>learning_rate_init</td>
<td>0.001</td>
</tr>
<tr>
<td>solver</td>
<td>adam</td>
<td>power_t</td>
<td>0.5</td>
</tr>
<tr>
<td>alpha</td>
<td>1.1</td>
<td>max_iter</td>
<td>200</td>
</tr>
<tr>
<td>batch_size</td>
<td>auto</td>
<td>shuffle</td>
<td>False</td>
</tr>
<tr>
<td>random_state</td>
<td>None</td>
<td>nesterovs_momentum</td>
<td>True</td>
</tr>
<tr>
<td>tol</td>
<td>0.0001</td>
<td>early_stopping</td>
<td>False</td>
</tr>
<tr>
<td>verbose</td>
<td>False</td>
<td>validation_fraction</td>
<td>0.1</td>
</tr>
<tr>
<td>warm_start</td>
<td>False</td>
<td>beta_1</td>
<td>0.9</td>
</tr>
<tr>
<td>momentum</td>
<td>0.9</td>
<td>beta_2</td>
<td>0.999</td>
</tr>
</tbody>
</table>

Table 6.4: Parameter for the Neural Network

6.4 Summary of ReBalanced Kernels

In this chapter, we addressed spatial imbalance and temporal imbalance present in a single kernel. We proposed a two-stage compiler-centric optimization that effectively reduces the spatial and temporal imbalance on a GPU. Our experiments using 35 kernels show a systematic throughput improvement of 24% (117% max) with Kernel Fusion, and a further 1.2% to 4.8% of IPC improvement using Kernel Fusion with Twin Kernels. We attempted to find an ML-enabled method to guide the selection of kernels to apply Kernel Fusion, though only achieved a 61% prediction accuracy with the best model.
Chapter 7

Conclusion

7.1 Conclusion

In this thesis, we focus on addressing GPU performance issues with compiler-centric approaches. In summary, we have made three novel contributions.

First, we developed M2C and M2V, two new tools to complete the toolchain for GPU research. M2C is a GPU compiler based on LLVM, which translates GPU kernels written in OpenCL 1.2 to kernel binaries that are fully compatible with the Multi2Sim simulator. We also enhanced the statistics support in the Multi2Sim framework with M2V, which analyzes simulation and generates visualization results. With M2V, we can replay a simulation with adjustable resolution and visualize the metrics of interest. M2C and M2V provide a foundation for the work in this thesis, and enable future cross-layer (i.e., hardware and software) studies targeting GPU systems.

Second, we studied the over-subscription issue of GPU memory system and proposed the Twin Kernels execution model to solve this problem. Twin Kernels optimizes the schedule of instructions in a single kernel in order to reduce contention in the memory system. Twin Kernels adds instruction scheduling passes in the back-end of M2C compiler to achieve instruction redistribution at compile time. Twin Kernels outperforms the prevailing SIMT model without any changes from the programmer.

Third, we examined the utilization issue in GPUs and offered the Rebalanced Kernel, a two-stage compiler-centric optimization that addresses the Spatial Imbalance and Temporal Imbalance utilization issues in GPUs. The first stage optimization improves instruction diversity and balances the utilization of hardware through Kernel Fusion. We also explored developing a predictive model for kernel selection in Kernel Fusion. The second stage optimization improves instruction distribution
and reduces over-subscription through Twin Kernels. Rebalanced Kernel compilation improves system throughput without major reconstruction of GPU applications.

7.2 Future Work

Our focus on a compiler-centric solution has enabled cross-layer optimization opportunities for future GPU researches. In our Twin Kernels work, the speedup potential is limited by the inherent reordering opportunities present in the code. Further opportunities can be exploited with a wavefront scheduler that works more efficiently with our work. In addition, given the static nature of our implementation, we rely on our execution tool to tune performance. Profile feedback using hardware counters can provide us with further guidance to accelerate this process.

Software and hardware optimization can work together to get closer to peak performance on GPUs. Our Rebalanced Kernels work gives some control to the programmer, but it is still opportunistic. For example, we can only control the ratio of the running work-groups, but there is no control over how the work-groups are dispatched. A software-hardware co-design can reduce the indeterministic and improve the overall performance of GPUs even further. Besides, we still lack an accurate predictive model, which would guide the selection of kernels to fuse. We have completed initial work in this direction, but found that there was no clear winner in terms of the classification model. Finally, we have run our experiments with fairly simple GPU kernels taken from standard benchmark suites. Emerging GPU workloads, including machine learning and deep learning, may benefit from merged kernel execution.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


107
BIBLIOGRAPHY


108


