CHARACTERIZING THE PERFORMANCE BOTTLENECKS OF IRREGULAR GPU KERNELS

by

Molly Anne O'Neil, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2015

Committee Members:

Martin Burtscher, Chair

Apan Qasem

Dan E. Tamir

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

To William & Miles:

For allowing me to be absent for so many bedtime stories

and

To Ron & Mary Beth:

For giving me a university campus for a childhood playground

and

To Daniel:

For believing in me

and for carrying us all through

ACKNOWLEDGEMENTS

"My heart is in the work."
– Andrew Carnegie

This work represents the dedication of many people beyond its author. First and foremost are my husband, Daniel, and our sons, William and Miles. Without their unwavering support and willing sacrifice my graduate career could never have happened.

My advisor, Dr. Martin Burtscher, has taught me by example what it means to have one's heart in the work. I hope that in my time here I have learned a tenth of his commitment to his students and his delight in scientific discovery. I am grateful for the immense freedom he allowed me in my research, for his advice and mentorship, for his humor, and for his high expectations of me; most of all, for his patience.

I am also indebted to my committee, Dr. Apan Qasem and Dr. Dan Tamir, for their time and technical insights; to Chris Shaltz, Saami Rahman, and to all of my labmates in the Efficient Computing Laboratory over the years (especially Annie Yang) for their friendship, camaraderie, and motivation, and for helping to keep me sane; to Carnegie Mellon teaching professor Greg Kesden for inspiring me as an educator and supporting me as a friend; and to Dr. Khosrow Kaikhah for getting me started.

Sreepathi Pai at The University of Texas and Joel Hestness at The University of Wisconsin provided invaluable help with LonestarGPU and GPGPU-Sim. I am very grateful to have had my work supported by the NSF Graduate Research Fellowship Program under grant #1144466 and by grants and gifts from NVIDIA Corporation.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS v

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xiii

ABSTRACT xiv

CHAPTER

1. INTRODUCTION 1

2. BACKGROUND 5

    2.1. CUDA GPUs 5
        2.1.1. Branch Divergence 8
        2.1.2. Memory Coalescing 9
        2.1.3. Cache and Memory Hierarchy 10
        2.1.4. Warp Scheduling 11

    2.2. Regular vs. Irregular Codes 13

    2.3. GPGPU-Sim 13

3. RELATED LITERATURE 16

    3.1. Simulator- and Emulator-Based Characterization 16

    3.2. Hardware-Based Characterization 18

    3.3. Hardware Modifications 19
        3.3.1. Warp Schedulers 19

4. APPLICATIONS AND INPUTS 21

4.1. Applications 21
4.1.1. Irregular Applications (LonestarGPU) 21
4.1.2. Semi-Regular Applications 24
4.1.3. Regular Applications 25

4.2. Inputs 25

5. GPGPU-SIM CONFIGURATION 28

5.1. PTX vs. PTXPlus 28

5.2. CUDA 5.5 and CUB Support 28

5.3. Bug Fixes and Additional Operations 29

5.4. Additional Performance Counters 30

5.5. GTX 480 (Base Configuration) 31

5.6. Additional Configurations 32
5.6.1. Cache and Memory Latency 32
5.6.2. Cache and Memory Bandwidth 33
5.6.3. Cache Size 34
5.6.4. Warp Scheduling Policy 34
5.6.5. No Coalescing Penalty 35

5.7. Benchmark Compilation for GPGPU-Sim 36

6. RESULTS AND ANALYSIS 38

6.1. Sources of Performance Limitation 38
6.1.1. Branch Divergence 39
6.1.2. Memory Coalescing 41
6.1.3. Cache Behavior 44

6.2. Individual Application Analysis 46

6.3. Impact of Hardware Modifications 54
6.3.1. Cache and Memory Latency 55

6.3.2. Cache and Memory Bandwidth 56
6.3.3. Cache Size.....56
6.3.4. Warp Scheduling Policy 58

6.4. Input Sensitivity.....61
6.4.1. Input Type.....62

7. SUMMARY AND CONCLUSIONS 65

7.1. Recommendations for Future Work.....66

LITERATURE CITED 67

LIST OF TABLES

Table **Page**

4.1. Application inputs 26

5.1. ROP and DRAM latency configurations (in number of shader clock cycles) 33

5.2. Interconnect (in bytes per flit) and DRAM (in bytes per DRAM chip) bandwidth configurations 33

5.3. Data cache size configurations (in kilobytes) 34

5.4. Warp scheduler configurations 35

5.5. No coalesce penalty configurations 36

LIST OF FIGURES

Figure **Page**

2.1. Relationship of threads, thread blocks, and kernel grids in CUDA 6

2.2. CUDA memory hierarchy 6

2.3. Fermi SM block-level microarchitecture 7

2.4. An illustration of branch divergence for a sample warp size of 10 threads 8

2.5. A coalesced access, i.e. all 32 threads in the warp access the same aligned 128-byte segment 9

2.6. An uncoalesced access requiring two memory transactions 10

2.7a. Fermi memory hierarchy 11

2.7b. Kepler memory hierarchy 11

2.8. An illustration of warp multithreading in a Fermi GPU 12

2.9. The architecture of a GPGPU-Sim streaming multiprocessor (i.e., SIMT core) ... 14

2.10. The GPGPU-Sim memory subsystem architecture 15

5.1. Issue stage histogram bin definitions 31

6.1. Measured instructions per cycle (IPC) of each benchmark 38

6.2. Runtime in cycles of all BFS and SSSP variants 39

6.3. Average warp occupancy of each application, both including and excluding idle/stall cycles 40

6.4. Benchmark speedup with perfect warp formation 41

6.5. Average memory access count per global or local warp load or store instruction 42

6.6. Percentage of cycles marked as coalescing stalls 43

6.7. Speedup over the default simulator configuration when removing the coalescing pipeline penalty and increasing the cache miss handling capacities 44

6.8. Data cache miss ratios (L2 ratios are local, i.e., number of misses over number of L2 accesses) 45

6.9. Cache misses per thousand warp instructions (MPKI) 45

6.10. The proportion of underused vs. fully-occupied cycles in each application (deepest pipeline stage) 48

6.11. The proportion of underused vs. fully-occupied cycles in each application (most impacted warps) 48

6.12. Speedup over the default simulator configuration when scaling the minimum L2 hit latency and DRAM latency 55

6.13. Speedup over the default simulator configuration when scaling the interconnect and DRAM bandwidths 56

6.14. Speedup over the default simulator configuration when scaling the data cache sizes 57

6.15. Speedup of the RR scheduler over the default GTO scheduler 58

6.16. Speedup of the GTLS scheduler over the default GTO scheduler 59

6.17. Speedup of the 2-level scheduler with RR inner and outer policies over the default GTO scheduler 60

6.18. Speedup of the 2-level scheduler with an RR outer and GTO inner policy over the default GTO scheduler 60

6.19. Speedup of the 2-level scheduler with GTO inner and outer policies over the default GTO scheduler 60

6.20. Instructions per cycle for each benchmark with several similar inputs 61

6.21. The proportion of underused vs. fully-occupied cycles (deepest pipeline stage) for all codes using several similar inputs 62

6.22. Instructions per cycle for all graph codes with road-network and R-MAT inputs.....63

6.23. The proportion of underused vs. fully-occupied cycles (deepest pipeline stage) for all graph codes using road-network and R-MAT inputs 63

6.24. The proportion of underused vs. fully-occupied cycles for BFS and SSSP with equally-sized inputs of different types.....64

LIST OF ABBREVIATIONS

Abbreviation	Description
API	Application program interface
BFS	Breadth-first search
BH	Barnes-Hut
CCWS	Cache-conscious wavefront scheduling
CDP	CUDA Dynamic Parallelism
CUDA	Compute Unified Device Architecture
DMR	Delaunay mesh refinement
DRAM	Dynamic random-access memory
FPC	Floating-point compression
GPU	Graphics processing unit
GTO	Greedy-then-oldest
IPC	Instructions per cycle
ISA	Instruction set architecture
LDST	Load/store
LRR	Loose round robin
MC	Monte Carlo
MPKI	Misses per thousand warp instructions
MSHR	Miss-status holding register
MST	Minimum spanning tree
NB	N-body
NCP	No coalesce penalty
PE	Processing element
R-MAT	Recursive Matrix (graph format)
ROP	Raster operations pipeline
RR	Round-robin
SDK	Software development kit
SFU	Special-function unit
SIMT	Single instruction multiple thread
SM	Streaming multiprocessor
SP	Survey propagation
SSSP	Single-source shortest paths
TSP	Traveling salesman problem

ABSTRACT

Graphics processing units (GPUs) are increasingly being used to accelerate general-purpose applications, including applications with data-dependent, irregular memory access patterns and control flow. However, relatively little is known about the behavior of irregular GPU codes, and there has been minimal effort to quantify the ways in which they differ from regular general-purpose GPU applications.

I examine the behavior of a suite of optimized irregular GPU applications written in CUDA on a cycle-level GPU simulator. I characterize the performance bottlenecks in each program and connect source code to microarchitectural performance characteristics. I also assess the performance impact of modifying hardware parameters such as the cache and DRAM bandwidths and latencies, data cache sizes, coalescing behavior, and warp scheduling policy, and I discuss the implications for future GPU architecture design.

I find that, while irregular graph codes exhibit significantly more underutilized execution cycles due to branch divergence, load imbalance, and synchronization overhead than regular programs, overall, code optimizations are often able to effectively address these performance hurdles. Insufficient bandwidth, long memory latency, and poor cache effectiveness are the biggest limiters of performance. Applications with irregular memory access patterns are more sensitive to changes in L2 latency and bandwidth than DRAM latency and bandwidth. Additionally, greedy-then-oldest scheduling is the best simple warp scheduler for irregular codes, and two-level scheduling does not significantly improve the performance of such codes.

1. INTRODUCTION

Recent years have seen a surge of interest in the use of graphics processing units (GPUs) as general-purpose computation accelerators. For programs that map well to GPU hardware, GPUs offer a substantial advantage over multicore CPUs [1] in terms of performance, performance per dollar, and performance per transistor. GPUs also outperform CPUs in energy efficiency, demonstrating improved performance per watt on codes suitable for GPU execution [2]. Due in part to these advantages, general-purpose programmable GPUs have become ubiquitous in high-performance computing [3] and are increasingly appearing as accelerators in desktops and even mobile platforms [4].

Designed to perform complex computations on blocks of pixels at high speeds and with wide parallelism, GPU architectures differ significantly from traditional CPU hardware. For example, for best performance, GPUs require large sets of threads (called *warps* or *wavefronts*) to execute the same instruction in lockstep, resulting in performance loss when threads within a warp encounter divergent control flow. This behavior is called *branch divergence*. GPUs additionally require the threads within a warp to access memory locations within the same cache line for fast, *coalesced* memory access. However, GPUs also have many more processing units than CPUs, along with wider memory buses, special operations for fast mathematical computation, and hardware barriers that enable rapid synchronization between threads without the latency of a main memory or last-level cache access.

These unique architectural features make GPUs particularly effective at accelerating programs that are highly *regular*, i.e., that operate on large vectors or matrices in statically predictable ways. Such codes often have high computational

demands, display extensive data parallelism, access memory in a streaming fashion, and require little synchronization [5]. Many important applications in scientific computing fit these criteria, including algorithms used in fields ranging from fluid dynamics [6] to computational finance [7]. There exists a broad base of knowledge on the efficient parallelization of these algorithms [8], and GPU implementations of these codes can be tens of times faster than optimized parallel CPU implementations [9].

However, many problem domains (e.g., *n*-body simulation [10], data mining [11], satisfiability problems [12], social networks [13], compilers [14], discrete-event simulation [15], and meshing [16]) employ algorithms that are *irregular* in nature: they build, traverse, and update pointer-based data structures such as trees and graphs and exhibit input-dependent control-flow and memory-access patterns. These programs are more difficult to parallelize in general and are particularly challenging to map to the unconventional architecture of GPUs. Nonetheless, the literature includes several GPU implementations of irregular algorithms that outperform their multicore CPU counterparts [17]-[20].

Due to the advantages they offer over multicore CPUs, GPUs and GPU-like architectures are likely to continue to grow in prevalence as accelerators for general-purpose computation. Despite evidence that GPUs are capable of accelerating even irregular applications, relatively little is known about the specific behaviors of irregular GPU kernels or the manner in which they interact with the GPU architecture, and there has been minimal effort to quantify the ways in which these codes differ from regular GPU applications. Identifying the most significant architectural performance limitations of irregular GPU kernels will aid software developers in accelerating these codes and is a

critical first step in enabling hardware designers to broaden the acceleration capabilities of future GPUs.

This work makes the following main contributions.

- I present a detailed, cycle-level simulation-based workload characterization focusing on irregular GPU applications.
- I develop an understanding of the impact of control-flow and memory-access irregularity on various architectural GPU performance characteristics, including memory coalescing, branch divergence, and cache effectiveness.
- I analyze programs from the LonestarGPU [21] benchmark suite of irregular codes and identify relationships between source code and microarchitectural performance behavior.
- I assess the sensitivity of these applications to hardware design parameters such as cache and DRAM latency and bandwidth, cache size, coalescing behavior, and warp scheduling policy.
- I find that memory and especially L2 latency and bandwidth and low cache effectiveness are the most significant factors limiting the performance of irregular GPGPU codes.

This thesis attempts to abstract a generalized understanding of the impact of irregular coding structures on hardware performance characteristics. It is not the goal of this work to determine the particular configuration of hardware parameters that yields the best average speedup for a particular set of codes or a particular GPU device. GPU architecture is still changing rapidly from generation to generation, and fine-tuning of microarchitectural parameters is unlikely to yield significant value at this premature

stage. Rather, this thesis identifies the major hardware bottlenecks for irregular kernels and establishes those sources of performance loss on which hardware architects should focus their efforts (versus those sources that software designers are capable of effectively addressing on their own).

The rest of this thesis is organized as follows. Chapter 2 reviews the architecture of CUDA GPUs and their main architectural sources of performance limitation, defines code regularity vs. irregularity, and describes the cycle-level simulator I employ. Chapter 3 summarizes the related literature. Chapter 4 discusses the benchmarks and inputs I study. Chapter 5 describes my simulator modifications and configurations. Chapter 6 presents and analyzes the results. Chapter 7 summarizes and concludes.

2. BACKGROUND

In this chapter, I describe the architecture of general-purpose programmable GPUs, their major sources of performance limitation, and important features of their microarchitectural design. I also review the differences between regular and irregular codes and describe the operation of the cycle-level simulator I employ.

2.1. CUDA GPUs

NVIDIA's general-purpose programmable GPUs implement the CUDA parallel programming model [22]. CUDA GPUs have a two-level compute hierarchy composed of multiple *streaming multiprocessors* (SMs), or *cores*, each composed of multiple tightly coupled *processing elements* (PEs). CUDA programs specify the behavior of *kernels*, which are launched on the GPU. A kernel is composed of parallel *threads*. As shown in Figure 2.1 [22] below, these threads are grouped by the programmer into *thread blocks* and *grids* of thread blocks. The thread blocks (or *blocks*) are dynamically assigned to SMs as SMs become available, up to a limit imposed by the available hardware resources (e.g., registers, shared memory space, block and thread tracking state) on the SM.

Each thread executes an instance of the kernel and has its own registers and local memory. The threads within a block share a software-controlled cache, called *shared memory*, as well as hardware to enable fast synchronization. Between blocks, synchronization and data exchange require accesses through slower, off-core global memory (DRAM). Figure 2.2 [22] below illustrates the CUDA memory hierarchy of per-thread local memory, per-block shared memory, and per-application global memory spaces.

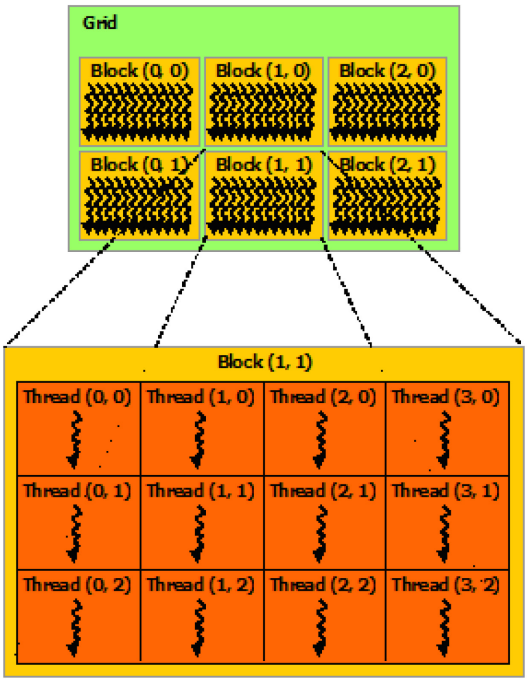


Figure 2.1: Relationship of threads, thread blocks, and kernel grids in CUDA

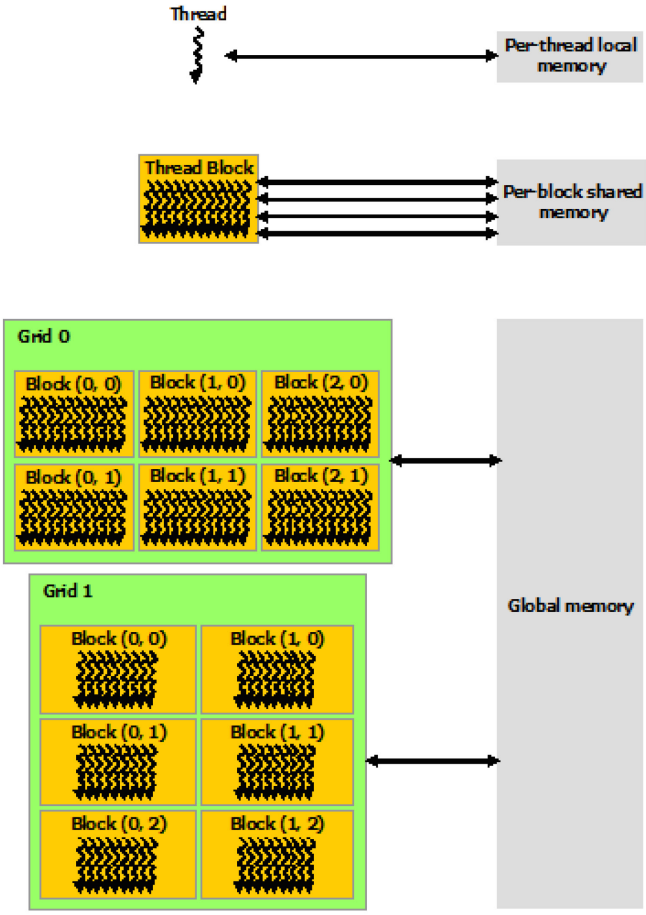


Figure 2.2: CUDA memory hierarchy

The PEs execute *warps*, which are sets of adjacent threads (32 in current CUDA GPUs) that execute on a set of PEs (either on 32 PEs or on fewer PEs split over multiple cycles) as vector instructions conditionally operating on 32 data items. The PEs are fed with warps for execution in multithreading style, allowing interleaving between thread blocks. Figure 2.3 [23] illustrates the block-level microarchitecture of one of the SMs in a compute capability 2.x (Fermi architecture [23]) GPU: the 'core' blocks are the PEs, each of which contains an integer and a floating-point pipeline. There are also load/store (LDST) units for warp instructions that access memory, as well as special-function units (SFUs) responsible for executing transcendental functions (e.g., sine, logarithm, etc.).

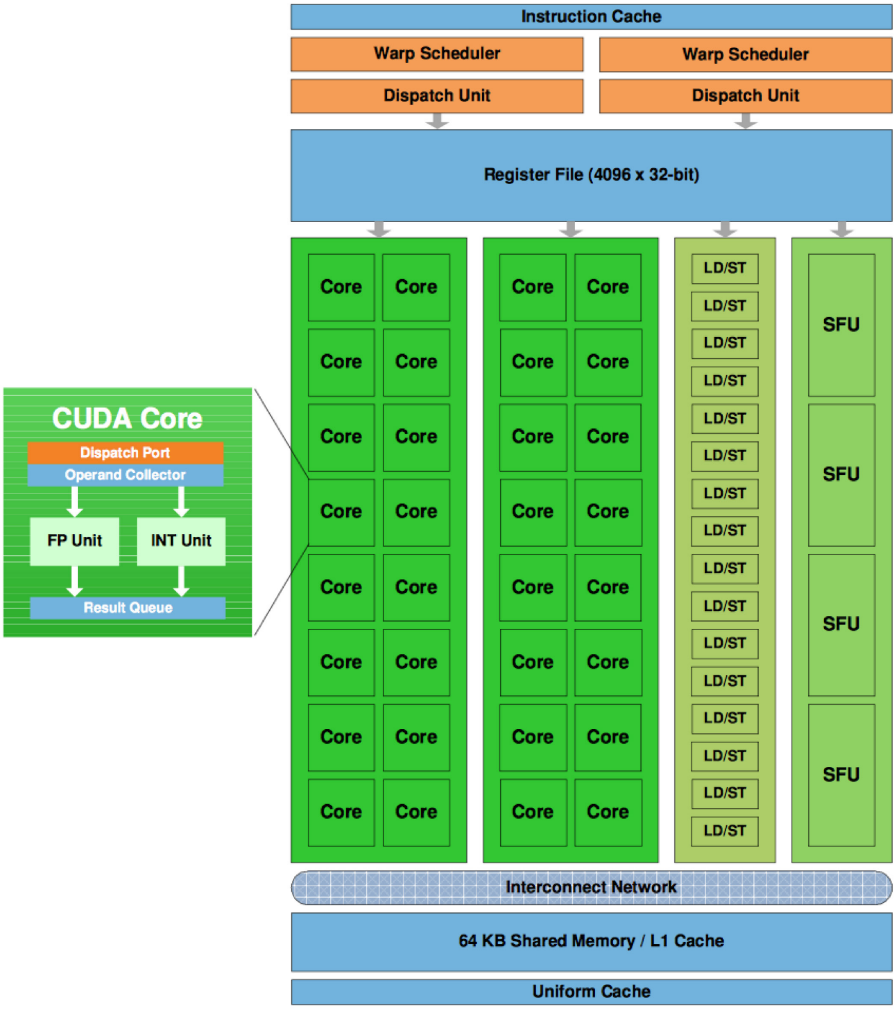


Figure 2.3: Fermi SM block-level microarchitecture

GPU pipeline microarchitectures are generally simple, without result forwarding, branch speculation, or out-of-order execution, and instead hide latency by arbitrarily interleaving warps. Good performance therefore requires a large number of warps in flight. In general, GPUs require large amounts of parallelism and minimal global synchronization. For best performance, they require threads in a warp to execute the same control-flow path and access nearby memory addresses.

2.1.1. Branch Divergence

To execute in parallel, the threads in a warp must share identical control flow. When they do not (i.e., when a conditional branch is encountered and a subset of threads in the warp evaluate the condition differently from the others), execution is automatically subdivided by the hardware into sets of threads such that all threads in the set execute the same instruction. These sets of threads are then executed serially until they re-converge.

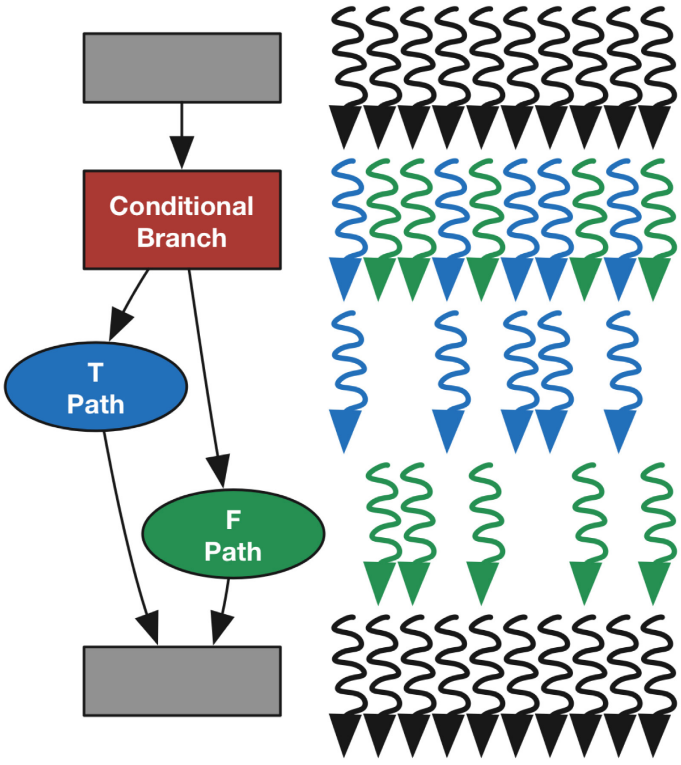


Figure 2.4: An illustration of branch divergence for a sample warp size of 10 threads

Figure 2.4 above illustrates an example of the utilization penalty associated with this serialization. In this example, the warp size is 10 threads. The threads execute in parallel until they reach a conditional branch and only 5 of the threads evaluate the condition as true. This results in 5 threads following one execution path and 5 threads following another. Until the execution paths reconverge, only half of the warp will be active in any cycle, lowering hardware utilization and increasing the number of cycles required to execute all threads. As such, good performance requires codes to have minimal *branch divergence*, i.e., for the threads in a warp to follow the same control-flow path most of the time.

2.1.2. Memory Coalescing

The GPU's memory subsystem is optimized for warp-based processing. If the threads in a warp simultaneously access words in global memory that fall into the same aligned 128-byte segment, the hardware merges the 32 reads or writes into a single, *coalesced* memory access, as shown in Figure 2.5 [24]. In the figure, the threads access memory locations in order by thread ID; however, the hardware would still coalesce the accesses into a single transaction if the arrows in the figure crossed over one another, as long as they all fell within the highlighted memory segment from byte 128 to byte 255.

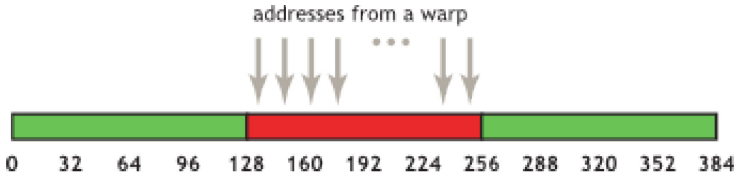


Figure 2.5: A coalesced access, i.e., all 32 threads in the warp access the same aligned 128-byte segment

When a warp instruction touches multiple 128-byte segments, the hardware must perform a memory transaction for each segment. This can occur even when the accesses have a single-word stride and fall within 128 consecutive bytes if the addresses cross a

128-byte boundary. Figure 2.6 [24] illustrates an *uncoalesced* access of this variety, requiring two separate memory transactions of 128 bytes each.

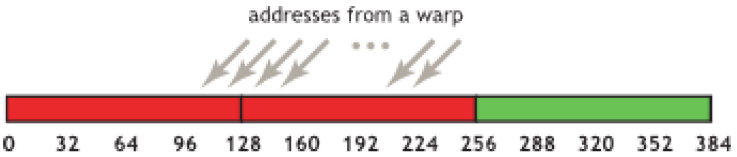


Figure 2.6: An uncoalesced access requiring two memory transactions

Since there are 32 threads in a warp, it is possible for a single warp instruction to touch addresses scattered across 32 separate 128-byte segments. Thirty-two separate memory transactions result, all of which may require data from global memory (DRAM). Even if all 32 accesses hit in a cache, the warp instruction will replay down the load/store pipeline 32 times, once for each transaction. Thus, coalesced accesses are crucial for good performance.

2.1.3. Cache and Memory Hierarchy

The GPU's main memory (DRAM) is separate from the CPU's, and explicit library calls are required to transfer blocks of data to or from GPU memory over the PCI-Express bus. Figure 2.7 below illustrates the memory and cache hierarchy of CUDA GPUs. All SMs share the DRAM as well as a unified L2 cache. Each SM has a software-managed scratch pad memory space called shared memory to allow fast data sharing within a block. In the Fermi architecture, shown in Figure 2.7(a) [23], each SM also has an on-chip, incoherent L1 data cache. This cache stores local data (including register spills) as well as global data. It is therefore necessary to declare global data that may be written and read by multiple blocks as *volatile* to prevent it from being cached in the L1.

In compute capability 3.X (Kepler architecture [25]) devices, the memory hierarchy of which is illustrated in Figure 2.7(b) [25], the L1 cache is used for local

memory only. Global data may be read into a separate read-only data cache only by explicit programmer direction (via the *__ldg()* intrinsic or the *const __restrict__* keywords). By default, global reads are non-cached and can therefore use smaller minimum bus transaction widths (because the full cache line is not required).

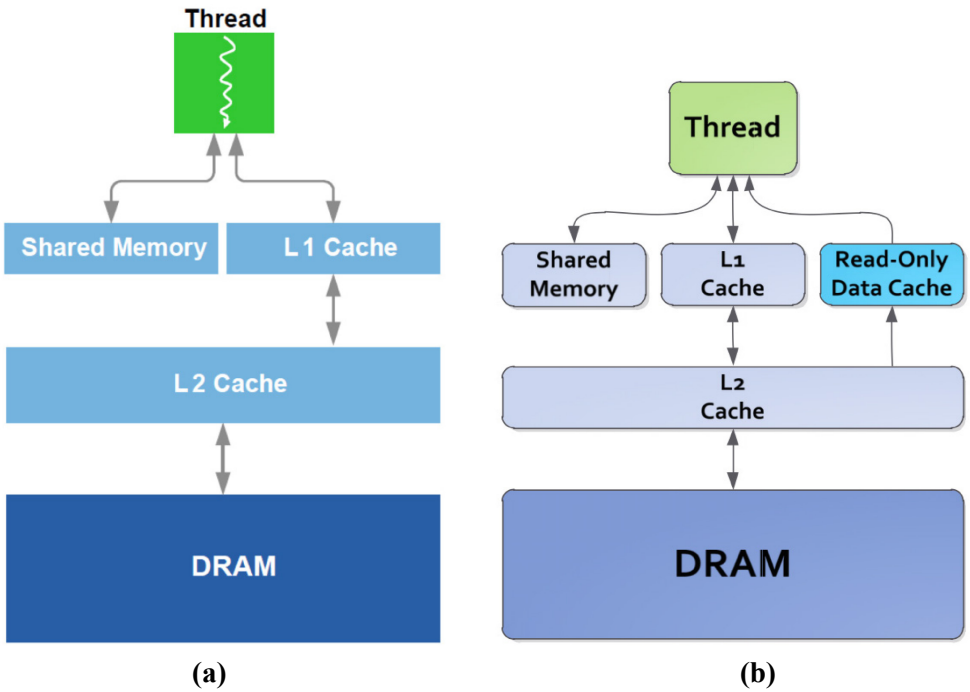


Figure 2.7: (a) Fermi memory hierarchy; **(b)** Kepler memory hierarchy

The shared memory and L1 cache of each SM share 64KB of on-chip storage. The shared memory size can be software-configured to either 16KB or 48KB (or 32KB in Kepler devices); the L1 cache occupies the remainder of the 64KB. These sizes may be configured separately on a per-kernel basis.

2.1.4. Warp Scheduling

GPUs exploit thread-level parallelism in part by multi-threading many warps on a single core (SM). When a warp encounters a long-latency operation (e.g., a memory access) or stall (e.g., a branch, a register dependency, etc.), the core can rapidly switch to another in-flight warp, thus interleaving instructions from many warps on a cycle-by-

cycle basis as shown in Figure 2.8 [23]. When a stall or long-latency operation is encountered and no other warp can issue its next instruction (either because there are insufficient warps in flight or because all other warps are also stalled), underutilization results.

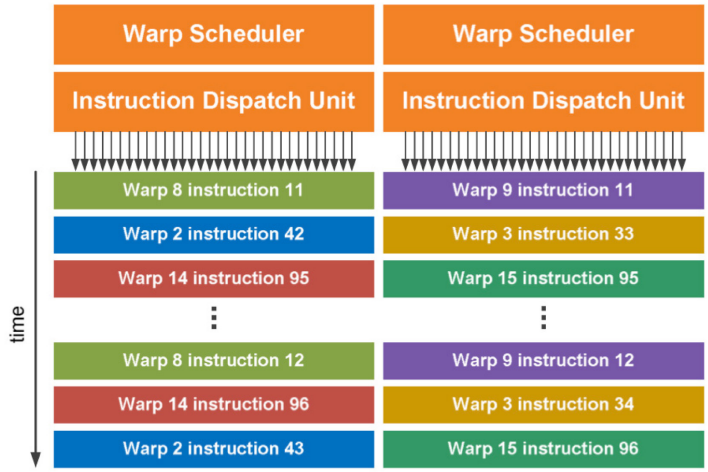


Figure 2.8: An illustration of warp multithreading in a Fermi GPU

When multiple warps are available to issue an instruction in a given cycle, the scheduling policy used to select the next warp can significantly affect the GPU's ability to hide latencies [26]. One simple scheduling policy, *round robin*, is illustrated in Figure 2.8 above: warps are ordered arbitrarily and each cycle, the next warp in order is considered for issue. A *loose round robin* (LRR) scheduler allows warps that are not able to issue to be skipped in the round robin ordering. An alternative scheduler is *greedy-then-oldest* (GTO) [27], which prioritizes issue from a selected warp until that warp reaches a long-latency operation, then prioritizes the oldest alternative in-flight warp. Round-robin schedulers give equal priority to each warp and tend to result in all warps arriving at long-latency operations in close time proximity (leaving no warp that can issue); however, schedulers (like GTO) that allow warps to become progressively out-of-sync can destroy memory-access locality and lead to starvation [26].

2.2. Regular vs. Irregular Codes

Parallel algorithms are often discussed in terms of regular vs. irregular behavior. *Regular* code refers to programs displaying neither data-dependent control flow nor data-dependent memory-access patterns. For example, the dynamic behavior (i.e., the conditional branch decisions and memory reference stream on an in-order processor) of a straightforward matrix multiply program can be statically determined knowing only the input size and the data-structure starting locations, but without knowing any of the matrix values.

Irregular code, in contrast, refers to programs in which the runtime behavior is a function of the input values. Both control-flow and memory-access patterns may differ for different inputs. Irregular code usually arises from the use of dynamic data structures such as trees and graphs. For example, an application that builds a binary search tree from a set of input data will follow a different control-flow path, build a tree of different shape, and access a different sequence of memory addresses depending on the input values and the order in which they are processed.

Compared to regular codes, irregular algorithms are more difficult to parallelize in general and more challenging to map to GPUs in particular. Their data-dependent dynamic behavior makes it difficult to assign work to threads in a manner that ensures identical control flow, coalesced memory accesses, or load balance.

2.3. GPGPU-Sim

In order to better understand the performance of irregular codes on GPU hardware, I study the behavior of a benchmark suite of irregular codes using GPGPU-Sim, a cycle-level microarchitectural model of an NVIDIA-like GPU for general-purpose

computation [28]. GPGPU-Sim performs functional and timing simulation of PTX assembly instructions extracted from a CUDA executable. In PTXPlus mode (cf. Chapter 5.1), it has been correlated against an NVIDIA GT 200 GPU with an IPC correlation of 97.6% on a subset of the CUDA SDK and against Fermi hardware with an IPC correlation of 97.3% on the Rodinia benchmark suite with reduced problem sizes [29]. GPGPU-Sim models the SMs (or SIMT cores), L1 caches, shared memory, interconnect, memory partition (including the L2 cache), and off-chip DRAM of a GPU.

Figure 2.9 [29] illustrates the block-level architecture of a GPGPU-Sim SIMT core. In addition to the caches and shared memory, each SIMT core models an instruction cache, fetch, decode, a warp scheduler, the SIMT stacks used to resolve branch divergence and reconvergence, a scoreboard and operand collector for register file access, as well as ALU, LDST, and SFU pipelines. The shared memory model simulates bank conflicts and associated stalls, and coalescing stalls are modeled based on the coalescing logic of compute capability 1.3 devices. The L1 data cache model includes *miss-status holding registers* (MSHRs) for tracking outstanding cache misses.

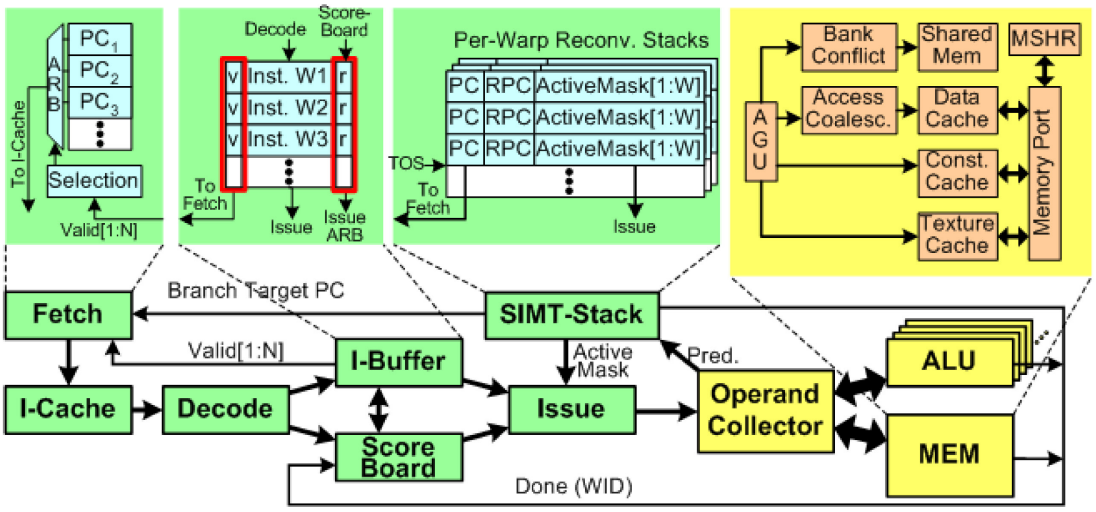


Figure 2.9: The architecture of a GPGPU-Sim streaming multiprocessor (i.e., SIMT core)

The unified L2 cache is modeled in the memory partition, separated from the SIMT cores by an interconnect network model. As shown in Figure 2.10 [29] below, the memory partition model includes a unit to resolve atomic operations, the L2 cache, a DRAM scheduler, the bus to off-chip DRAM, and a configurable timing model for the DRAM. L2 hit latency is modeled via the *raster operations pipeline* (ROP) queue, which sits between the interconnect and the L2 cache. DRAM hit latency is modeled via a queue between the L2 and the DRAM scheduler. Both of these queues are of configurable length, and the interconnect bandwidth and DRAM bus width are also configurable.

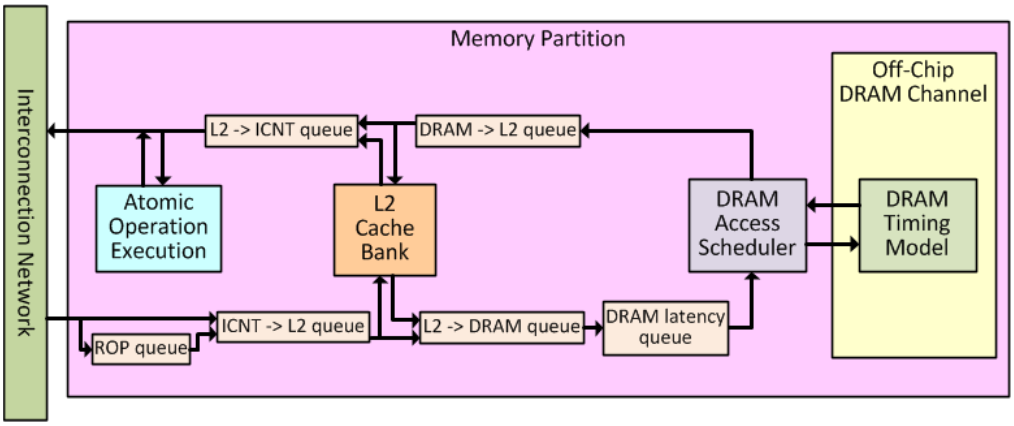


Figure 2.10: The GPGPU-Sim memory subsystem architecture

At present, GPGPU-Sim models GPUs similar to NVIDIA's compute capability 1.x and 2.x (Fermi) devices. It does not provide configurations for Kepler devices, nor does it model the memory hierarchy changes implemented in the Kepler architecture.

3. RELATED LITERATURE

This chapter summarizes existing literature related to understanding the behavior of GPU applications and/or irregular codes. It also describes the warp schedulers whose impact on irregular codes I examine in this thesis.

3.1. Simulator- and Emulator-Based Characterization

Previous simulator-based studies characterizing GPU applications focus on the mostly regular codes found in the CUDA SDK [30], Rodinia [31], and Parboil [32] benchmark suites. Bakhoda et al. [33] present GPGPU-Sim and study twelve (mostly regular) CUDA applications demonstrating various levels of GPU performance. They characterize the performance impact of several microarchitectural design choices including interconnect topology, memory controller design, and workload distribution and analyze the instruction mix, DRAM locality, and warp occupancy of each benchmark. Unlike this thesis, they do not investigate stall cycle distribution or warp scheduling policy. Goswami et al. [34] propose a set of microarchitecture-agnostic general-purpose GPU workload characterization metrics (e.g. instruction mix, DRAM row locality, branch divergence density) and use these metrics to study the benchmarks in the CUDA SDK, Rodinia, and Parboil suites using GPGPU-Sim. Their work focuses on identifying workload characteristics that are not sensitive to microarchitectural configuration, whereas this thesis quantifies benchmark sensitivity to various design tradeoffs. Blem et al. [35] propose a set of challenge benchmarks (selected from the GPGPU-Sim benchmarks, Rodinia, and a handful of naïve ports of Parsec applications) where the achieved instructions per cycle (IPC) is less than 40% of peak. They use GPGPU-Sim to present a characterization of the benchmarks' key architectural

bottlenecks and apply an analytic model to predict the performance impact of mitigating each bottleneck, whereas this work measures the performance impact of various hardware modifications in simulation. Che et al. [36] evaluate the Rodinia benchmark suite on a GTX 480 and in GPGPU-Sim, including warp occupancy and the performance impact of scaling DRAM bus width. However, their primary focus is an assessment of the Rodinia benchmark suite in comparison to traditional multithreaded CPU workloads. Lee and Wu [37] use GPGPU-Sim to examine program behavior in terms of stall cycle distribution, which is also a component of my work. They focus on the Rodinia benchmark suite and do not attempt to delineate the impact of irregularity. Hestness et al. [38] compare the memory system behavior of the Rodinia suite on the CPU versus the GPU in a heterogeneous system simulated with gem5-gpu [39], which integrates the GPU core model from GPGPU-Sim. Their study focuses mostly on regular codes and is limited to memory behavior.

General-purpose GPU application performance has also been studied using PTX emulators that do not provide cycle-accurate simulation. Kerr et al. [40] propose a set of metrics for GPU workloads and analyze these metrics on over fifty mostly regular applications, including the SDK and Parboil, via the GPU Ocelot emulator [41]. They investigate the impact of optimizations such as various branch re-convergence mechanisms and memory read coalescing. Wu et al. [42] study several benchmarks (including the SDK, Rodinia, and Parboil) using GPU Ocelot and identify sources of control-flow irregularity.

Two publications from the same conference in late 2014 focus specifically on the behavior of irregular applications in simulation. Xu et al. [43] study the behavior of a

collection of CUDA graph codes collected from multiple authors using hardware performance counters and GPGPU-Sim. These codes include BFS, MST, SP, and SSSP; BH and DMR are not represented, nor are the multiple implementations of BFS and SSSP included in LonestarGPU. The authors examine pipe stall cause, cache effectiveness, SIMD lane utilization, and thread block distribution. They also study the performance impact of cache size scaling and a handful of different warp scheduling schemes (LRR, GTO, and a single configuration of the two-level scheduling described in Chapter 3.3.1 below), which they find to yield insignificant performance variation. Unlike this thesis, they do not examine coalescing behavior or the impact of scaled cache and memory latencies and bandwidths. O'Neil and Burtscher [44] detail an intermediate selection of the work presented here.

3.2. Hardware-Based Characterization

Most of the existing studies that concentrate on irregular codes characterize application performance via hardware counters and sensors. Burtscher et al. [45] previously characterized the control-flow and memory-access irregularity in the LonestarGPU 1.0 benchmark suite of irregular GPU codes. Their study relied on hardware performance counters for issued and executed instructions, divergent branches, and instructions replayed for coalescing or bank conflicts. Coplin and Burtscher [46] characterized the power behavior of the LonestarGPU suite and the impact of program irregularity on power consumption. Their study profiles power characteristics via the built-in power sensor on NVIDIA K20 GPUs. Che et al. [47] also describe a hardware performance counter-based characterization of a suite of irregular GPU graph applications. Wang and Yalamanchili [48] implement eight irregular, data-intensive

applications using CUDA dynamic parallelism (CDP) and analyze the control-flow and memory behavior of each implementation versus its non-CDP version using hardware performance counters.

However, many of the event counts of interest in a study of irregular codes are not available through hardware counters, and simulation is necessary to provide a more complete picture of application behavior as well as to allow the evaluation of proposed hardware improvements.

3.3. Hardware Modifications

Many previous publications (e.g., Brunie et al. [49], Singh et al. [50], Xiang et al. [51]) have proposed GPU architecture modifications and several have used GPGPU-Sim to reason about the likely performance impact of their proposed changes. Some of these works include microarchitectural performance characterization similar to this thesis but applied specifically to the proposed architectural enhancement. For example, Meng et al. [52] propose a method of dynamic warp subdivision to hide branch and memory latency divergence and characterize the performance impact of additional variations in cache miss latency and other microarchitectural parameters.

3.3.1. Warp Schedulers

Several publications have proposed novel GPU warp schedulers. Narasiman et al. [26] present a two-level warp scheduler that splits concurrently executing warps into fetch groups and prioritizes issuing warps from a single fetch group until that group stalls. Such prioritization between fetch groups attempts to mitigate the tendency with a round-robin scheduler for all active warps to reach long latency operations together, while still preserving DRAM row-buffer locality within a single fetch group. The authors

present experimental results for a fetch group size of eight with round-robin selection for both the inner policy, i.e., within each fetch group, and the outer policy, i.e., when switching to a new fetch group. However, other schedulers could be employed at either of the two levels.

Rogers et al. (2012) [27] propose cache-conscious wavefront scheduling (CCWS), which detects when intra-wavefront locality is lost because the working sets of inflight warps oversubscribe the capacity of the L1 data cache. The scheduler then throttles back the number of warps eligible to issue. On the authors' selected highly cache-sensitive benchmarks, CCWS yields a 63% performance improvement over previous schedulers. Rogers et al. compare their CCWS scheduler to a two-level scheduler similar to that described by Narasiman et al. but using greedy-then-optimal inner and outer scheduling policies with a fetch group of size two, which proved the best two-level configuration for their codes.

4. APPLICATIONS AND INPUTS

In this work, I present a workload characterization of several irregular GPGPU applications and compare the codes' microarchitectural performance behavior to that of more regular applications. This chapter describes the applications and inputs I profile.

4.1. Applications

I focus primarily on irregular codes. However, I also characterize several more regular programs in order to provide a basis of comparison for the irregular codes.

4.1.1. Irregular Applications (LonestarGPU)

My irregular programs are selected from version 2.0 of the LonestarGPU benchmark suite [21]. LonestarGPU is a collection of hand-optimized CUDA implementations of real-world irregular applications. It includes the following algorithms, some of which have multiple implementations.

- *Breadth-First Search (BFS)*

This graph algorithm labels each node in an unweighted graph with the node's minimum level (i.e., number of edges) from a specified source node, which is defined to be level zero [53]. It is a key kernel in many applications (e.g., mesh partitioning). The GPU kernel considers each node in the graph and updates the level of any of its neighbor nodes for which a new minimum edge count from the source has been found. The kernel is invoked iteratively from the host until no minimum levels change. The LonestarGPU benchmark suite includes several implementations of this algorithm:

BFS: This is a *topology-driven* implementation, meaning that it processes all nodes in the graph in each iteration as long as at least one node is active. It assigns one node per thread.

BFS-unroll: This is a topology-driven implementation that processes multiple frontiers per iteration by maintaining a local (shared memory) worklist of changed destinations. It assigns one node per thread.

BFS-wlw: This is a *data-driven* implementation, meaning that it processes work items from a *worklist* of active nodes until the worklist is empty. Only nodes which require work, i.e. which have become active, are added to the worklist. This implementation assigns one node per thread.

BFS-wlc: This is a data-driven implementation that assigns one edge per thread using the strategy described by Merrill et al. [19]. It cooperatively tracks edge and vertex frontier lists in global memory in parallel.

- *Barnes-Hut (BH)*

This *n*-body algorithm simulates the effect of gravity on a star cluster [10] through several time steps. In each step, it hierarchically decomposes the space around the stars and records the structure in an octree. Special octree traversals allow for the quick approximation of forces. This code includes several GPU-specific optimizations, including sorting the bodies so that nearby threads perform similar work. Its force calculation kernel, which dominates application runtime, has been implemented in an explicitly warp-based manner by expanding octree prefixes so that all warp threads perform the same prefix traversal.

- *Delaunay Mesh Refinement (DMR)*

This is a mesh refinement algorithm from the field of computational geometry [54] that iteratively transforms those triangles in an input mesh that fail to conform to a quality criterion, i.e., 'bad' triangles (containing angles less than 30 degrees), into 'good'

triangles by retriangulating the cavity around each bad triangle. The GPU algorithm distributes triangles to threads. At each iteration, the kernel checks if the triangle assigned to it is bad, collects the triangles in the surrounding cavity, and re-triangulates the cavity. The host repeatedly invokes the kernel until no bad triangles remain in the mesh. In order to avoid expensive locks on the cavity surrounding a bad triangle, the code implements a barrier-based race and resolve prioritization scheme to avoid conflicts.

- *Minimum Spanning Tree (MST)*

Borvuka's MST algorithm [55] computes a minimal spanning tree of an undirected, weighted graph through successive application of minimum weight edge contractions. This process is repeated until the graph consists of just a single node, at which point the contracted edges form the spanning tree. In order to avoid modifying the graph, the GPU implementation performs edge contraction indirectly by tracking merged edges, or components. The computation is split across several kernels, which find the minimum weight edge out of each node, find the minimum weight edge out of each component, identify the edge to contract (i.e., the two nodes or components to merge), and perform the merge. These kernels are called repeatedly from the host until a single component remains or the number of components stops changing.

- *Survey Propagation (SP)*

This benchmark is a heuristic SAT-solver based on Bayesian inference [12] which represents a Boolean formula as a bipartite graph with variables on one side and clauses on the other. The algorithm iteratively updates each variable with the likelihood that it should be assigned a true or false value. Each iteration of the GPU code has two phases: the first kernel updates the survey on each edge; the next kernel fixes highly biased

variables to a specific value and removes them from the graph. These kernels are called repeatedly from the host until only trivial surveys exist or no progress has been made for several iterations.

- *Single-Source Shortest Paths (SSSP)*

This classic graph problem computes the shortest (i.e., minimum weight) path to each node in a directed, weighted graph from a designated source node using a modification of the Bellman-Ford algorithm [53]. The GPU code is similar in operation to BFS. In each iteration, the kernel considers each edge and updates the shortest distance to each destination node to which a shorter path has been found. The kernel is invoked repeatedly until it converges, i.e., until it reaches an iteration where no distances change. The LonestarGPU benchmark suite includes several implementations of this algorithm:

SSSP: This is a topology-driven implementation that assigns one node per thread.

SSSP-wln: This is a data-driven implementation that assigns one node per thread.

SSSP-wlc: This is a data-driven implementation that assigns one edge per thread using the strategy presented in Merrill et al. [19] adapted to SSSP.

4.1.2. Semi-Regular Applications

In addition to the LonestarGPU codes, I examine the behavior of two CUDA programs that display some regular and some irregular behavior.

- *Floating-Point Compression (FPC)*

This program implements a lossless data compression and decompression algorithm for double-precision floating-point values [56]. The code processes chunks of input in parallel. It displays streaming memory-access behavior but irregular control flow dependent on how well each word can be compressed.

- *Traveling Salesman Problem (TSP)*

This is a classic combinatorial optimization problem involving finding the minimal Hamiltonian tour in a complete, undirected, weighted graph. The GPU code implements an iterative hill climbing approach with random restarts to find an approximate solution [57] and stores a distance matrix in software-controlled shared memory. It has largely regular control flow but a data-dependent memory-access pattern.

4.1.3. Regular Applications

Lastly, I also characterize the behavior of two fully regular GPGPU programs.

- *N-Body (NB)*

Similarly to BH, this is an *n*-body application that simulates the motion of stars. Unlike BH, it performs precise all-to-all force calculations, making both its control flow and memory-access pattern regular. It is an in-house implementation from Texas State University's Efficient Computing Laboratory and outperforms the corresponding CUDA SDK code.

- *Monte Carlo (MC)*

This program evaluates the fair call price for a set of European options using the Monte Carlo method. It is a highly regular, array-based floating-point code from the CUDA SDK version 4.2 [30].

4.2. Inputs

For each application, I examine performance behavior using realistic inputs large enough to keep the simulated hardware busy but small enough to result in reasonable simulator runtimes (less than 48 hours where possible, but up to several weeks for some

applications). Table 4.1 lists the inputs I use with the primary input listed first for each benchmark.

Table 4.1: Application inputs

Application	Input			
	Name	**Description**	**Working Set Size**	**L2 Size Multiplier**
BFS, SSSP	USA_road_d.NY	NY roads (264K nodes, 734K edges)	3899 kB	5.08
	USA_road_d.BAY	SF Bay Area roads (321K nodes, 800K edges)	4380 kB	5.70
	rmat200k-1600k	R-MAT (200K nodes, 1600K edges)	7031 kB	9.16
	rmat264k-734k	R-MAT (264K nodes, 734K edges)	3898 kB	5.08
BH	494,000 1 (seed=7)	494K bodies, 1 timestep	7718 kB	10.05
	494,000 1 (seed=1)	494K bodies, 1 timestep	7718 kB	10.05
DMR	massive.2	100.3K triangles, maxfactor=10	7840 kB	10.21
	30k	60K triangles, maxfactor=10	4688 kB	6.10
	25k	50K triangles, maxfactor=10	3906 kB	5.09
MST	USA_road_d.NY	NY roads (264K nodes, 734K edges)	3898 kB	5.08
	USA_road_d.BAY	SF Bay Area roads (321K nodes, 800K edges)	4380 kB	5.70
	rmat30k-250k	R-MAT (30K nodes, 250K edges)	1093 kB	1.42
SP	random-4200-1000-3-seed23.cnf	4.2K clauses, 1K literals, 3 literals/clause	414 kB	0.54
	random-4200-1000-3-seed27.cnf	4.2K clauses, 1K literals, 3 literals/clause	414 kB	0.54
	random-4200-1000-3-seed71.cnf	4.2K clauses, 1K literals, 3 literals/clause	414 kB	0.54
FPC	obs_error	60 MB dataset, 30 blocks, 24 warps/block, dimensionality=24	60 MB	78.12
	num_plasma	34 MB dataset, 30 blocks, 24 warps/block, dimensionality=2	34 MB	44.27
	msg_lu	X MB dataset, 30 blocks, 24 warps/block, dimensionality=5	186 MB	242.19
TSP	att48.tsp 15,000	48 cities, 15K restarts	9 kB	0.01
	eil51.tsp 15,000	51 cities, 15K restarts	10 kB	0.01
	pr76.tsp 20,000	76 cities, 20K restarts	23 kB	0.03
NB	23,040 1 (seed=7)	23,040 bodies, 1 timestep	360 kB	0.47
	23,040 1 (seed=19)	23,040 bodies, 1 timestep	360 kB	0.47
	23,040 1 (seed=43)	23,040 bodies, 1 timestep	360 kB	0.47
MC	(default)	SDK input w/ 262,144 paths	1024 kB	1.33

By definition, irregular codes display input-dependent behavior. I profile each application using a primary input across all simulator configurations; additional inputs

are examined on the default configuration in order to quantify the input sensitivity of the results. The graph codes are studied using both road network inputs from the 9th DIMACS Implementation Challenge [58] as well as R-MAT [59] graphs, which are denser and have higher and more varied out-degree per node.

For each code, Table 4.1 also includes the size of the data structures read by the dominant kernel's inner loop (i.e., the kernel's *working set*) compared to the simulated GPU's L2 cache size. In order to characterize realistic cache behavior, benchmarks are tested with a working set at least five times the size of the default L2 cache configuration where possible, limited by practical simulation runtime and input availability. (Note that several of the more regular applications are streaming or limited by shared memory size and therefore do not have meaningful working sets).

5. GPGPU-SIM CONFIGURATION

I study each benchmark's cycle-level behavior using a modified version of GPGPU-Sim release 3.2.1 [28]. I modified the simulator to support CUDA 5.5 binaries, to implement a handful of additional operations, and to resolve some known bugs. I also added simulator configurations and instrumentation for additional performance counters. This chapter describes my simulator modifications and configurations as well as the benchmark changes required to support execution on GPGPU-Sim.

5.1. PTX vs. PTXPlus

GPGPU-Sim performs functional and timing simulation of PTX assembly instructions extracted from a CUDA executable. Because PTX is a virtual ISA and not the actual code that runs on the hardware, GPGPU-Sim also supports PTXPlus, an extended version of PTX that adds addressing modes, condition codes, and instructions similar to those in SASS, NVIDIA's native hardware instruction set. PTXPlus simulation is likely to result in more accurate correlation to real hardware; however, GPGPU-Sim does not fully support it for all programs supported in PTX. Most of my studied benchmarks do not run in PTXPlus mode, and I do not present PTXPlus results.

PTX assumes an infinite register set and therefore simulation results do not capture the impact of register spill code. Of my studied codes, DMR, SP, and NB cause a handful of register spills in their dominant kernels.

5.2. CUDA 5.5 and CUB Support

The LonestarGPU 2.0 benchmark suite must be compiled with the CUDA 5.0+ tool suite, because the benchmark codes rely on functionality from the CUB library [60]

version 1.1.1. CUB provides various PTX intrinsics, device management functions, and reduction primitives for CUDA and requires a modern C++ compiler, which has been incorporated into CUDA 5.0+. GPGPU-Sim 3.2.1 natively supports CUDA versions up to only CUDA 4.2. Therefore, I had to modify the simulator using patches provided by the LonestarGPU suite authors [61]. These patches shim the *cuobjdump* and *ptxas* utilities so the simulator's parsers are able to parse binaries compiled with CUDA 5.5, update the naming of some texture lookup functions, and add functionality to the simulator's implementation of the CUDA library functions *cudaGetDeviceProperties* and *cudaFuncGetAttributes*.

5.3. Bug Fixes and Additional Operations

Several modifications to the simulator were required to support my selected benchmarks.

GPGPU-Sim incorrectly implements the REM instruction, which is used to perform a modulo operation. The correct hardware behavior is to mask the operation to the size specified by the <type> field; however, the simulator always performs the modulo operation with 64-bit operands and stores a 64-bit result. I modified the functional simulator to make the REM instruction size aware.

GPGPU-Sim's functional simulator does not sign-extend integer literals when used as operands for 64-bit instructions. I modified the implementation of the MOV.u64 and MOV.s64 instructions to sign-extend a source operand literal to 64 bits.

The timing simulator includes performance counters for various stall causes. In GPGPU-Sim 3.2.1, an L1 data cache reservation fail incorrectly increments the

coalescing stall performance counter. I modified the simulator's shader core implementation to prevent the incorrect increment.

The CUB library reads the *laneID* register, which returns the position of a thread within its warp. GPGPU-Sim does not implement this register. I modified the functional simulator to return the correct data for reads of the *laneID* register.

The SP benchmark (via the CUB library) also uses the BFE instruction, which extracts a bit field of specified starting position and length from a source operand. GPGPU-Sim does not implement this PTX instruction. I modified the functional simulator to correctly implement the bit-field extract operation.

5.4. Additional Performance Counters

By default, GPGPU-Sim includes performance counters in the warp issue stage. In each cycle, every SM increments a counter with the active instruction count of the warp issued in that cycle. In cycles where no issue can be made, a counter is incremented for the cause of the issue stall instead. I supplemented these counters to create a histogram of fully occupied issue cycles as well as cycles where the active instruction count is less than the full warp size (i.e., divergence cycles), and to differentiate several causes of a stall or idle (i.e., no issue) cycle. In the case of issue stalls due to a full functional pipe, I added instrumentation to collect additional information on the functional unit responsible for the stall.

To ensure that only one bin is updated per scheduler per cycle, it is necessary to define a priority between the idle and stall conditions, since warps ineligible for issue in that cycle may be stalled for multiple reasons. Figure 5.1 illustrates the priority definitions of the histogram bins. The deeper in the pipeline a stall cause is determined,

the higher the priority of that stall. For example, if there are 32 warps available on the SM, 31 of them are invalid due to control hazards, and one warp has a valid instruction but cannot issue because it requires a functional unit that is already full, the cycle will be marked as a pipeline stall (because functional unit backup is determined later in the pipeline than control-flow hazards).

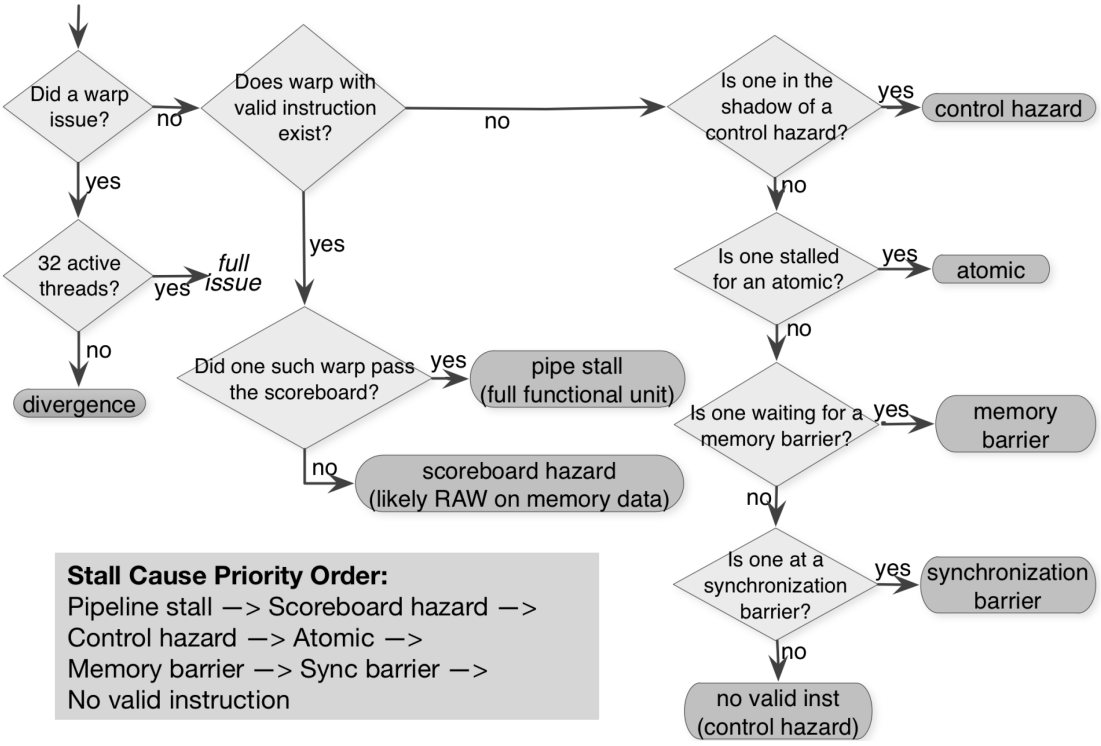


Figure 5.1: Issue stage histogram bin definitions

I also instrumented the simulator with an additional histogram that increments the cause per no-issue cycle that is responsible for stalling the *most* warps eligible for issue that cycle.

5.5. GTX 480 (Base Configuration)

GPGPU-Sim 3.2.1 includes a configuration for the GTX 480, an NVIDIA Fermi GPU. The GTX 480 has 15 SMs. Each SM includes two warp schedulers and two dispatch units, allowing warps to dual-issue. The 32 threads in a warp are issued over two

cycles, 16 threads at a time. The SMs can support a maximum of 1536 threads and 8 thread blocks. Each SM has 32,768 registers. The SMs have two ALU pipelines, one LDST pipeline, and one SFU pipeline each. Each SM has 64kB of data cache space, which can be configured to 16 kB of software-controlled shared memory and 48 kB of hardware-controlled L1 cache or vice versa. A cache line is 128 bytes. The L2 cache is 768 kB.

In addition to the specifications above, the simulator's GTX 480 configuration specifies the cache and memory topology, latencies, and bandwidths based on the results of a microbenchmarking study of the GT200 [62]. The GTX 480 configuration uses a 16-way L2 cache in each of its 6 memory partitions. When the L1 is configured to 16 kB, it is 4-way set associative; it becomes 6-way set associative when configured to 48 kB. Because register allocation is not done in PTX, the simulator determines the register and shared memory usage of each kernel and uses that information and the applied configuration to determine the maximum number of thread blocks that can run concurrently on an SM.

I use the GTX 480 configuration as the default configuration in my study.

5.6. Additional Configurations

In order to assess the performance impact of various microarchitectural parameters and design decisions, I modify the default GTX 480 simulator configuration. This subchapter details these additional configurations.

5.6.1. Cache and Memory Latency

GPGPU-Sim models the minimum L2 hit latency via the raster operations pipeline (ROP) latency, which determines the minimum latency between when a memory

request arrives at the memory partition and when it accesses the L2 cache. Additionally, the simulator allows configuration of the DRAM latency, the minimum latency between when a memory request accesses the L2 cache and when it is pushed to the DRAM scheduler. Table 5.1 displays the ROP and DRAM latency configurations I examine.

Table 5.1: ROP and DRAM latency configurations (in number of shader clock cycles)

Configuration Name	ROP Latency	DRAM Latency
Default	240	200
1/2x ROP Latency	120	200
2x ROP Latency	480	200
1/2x DRAM Latency	240	100
2x DRAM Latency	240	400
No Latency	0	0

GPGPU-Sim models the minimum L1 hit latency as a single cycle, and I did not modify this behavior. However, a recent study [63] suggests a significantly longer hit latency, and this is an area for future investigation.

5.6.2. Cache and Memory Bandwidth

GPGPU-Sim enables configuration of the interconnect bus width between the memory partitions (which include the L2 cache) and the core, as well as the DRAM bus width. Scaling these bus widths is equivalent to scaling the L2 and memory bandwidths. Table 5.2 displays the interconnect and DRAM bandwidth configurations I study.

Table 5.2: Interconnect (in bytes per flit) and DRAM (in bytes per DRAM chip) bandwidth configurations

Configuration Name	Interconnect Flit Size	DRAM Bus Width
Default	32	4
1/2x Interconnect B/W	16	4
2x Interconnect B/W	64	4
1/2x DRAM B/W	32	2
2x DRAM B/W	32	8
1/2x Interconnect + 1/2x DRAM B/W	16	2
2x Interconnect + 2x DRAM B/W	64	8

5.6.3. Cache Size

Table 5.3 details the data cache size configurations I employ. Note that regardless of the configuration selected from this table, the size of the L1 data cache always depends on the cache configuration being simulated: at the default configuration, 48 kB of shared memory and 16 kB of L1 data cache (termed *PreferShared*), or 16 kB of shared memory and 48 kB of L1 data cache (termed *PreferL1*).

All cache size configurations scale the data cache size by modifying the number of sets; the block size and associativity remain at their default values.

Table 5.3: Data cache size configurations (in kilobytes)

Configuration Name	L1 Data Cache Size		L2 Cache Size
	PreferShared	***PreferL1***	
Default	16	48	768
1/2x L1D	8	24	768
2x L1D	32	96	768
1/2x L2 Cache	16	48	384
2x L2 Cache	16	48	1536

5.6.4. Warp Scheduling Policy

I simulate my benchmarks using four warp-scheduling policies, some of which have several configuration options. The first is the simulator default, a greedy-then-oldest (GTO) warp scheduler intended to prevent available warps from reaching long-latency operations in close time proximity. GPGPU-Sim also includes a loose round-robin (RR) warp scheduler, which exploits inter-warp locality by keeping warp execution roughly synchronized, and an implementation of the two-level scheduler described in Chapter 3.3.1. By default, the two-level scheduler uses RR selection for both the inner and outer policies. In addition to this configuration, I implemented GTO scheduling policies for both inner and outer warp selection. Lastly, I implemented a greedy-then-least-stalled (GTLS) warp scheduler that tracks the number of times warps eligible for issue have been

non-issued for a long operation (i.e., a synchronization barrier, memory fence, atomic, or RAW hazard on load data) and, when a warp stalls, prioritizes issue from the warp that has been stalled for the fewest number of cycles. The tracking of stall counts is cleared every clear_count cycles (where clear_count is a configuration-defined parameter). Table 5.4 lists the warp scheduler configurations I assess.

Table 5.4: Warp scheduler configurations

Configuration Name	Clear Count	Outer Policy	Inner Policy	Fetch Group Size
GTO (Default)	N/A	N/A	N/A	N/A
LRR	N/A	N/A	N/A	N/A
GTLS_50	50	N/A	N/A	N/A
GTLS_100	100	N/A	N/A	N/A
GTLS_500	500	N/A	N/A	N/A
2 Level: RR_RR_2	N/A	RR	RR	2
2 Level: RR_RR_4	N/A	RR	RR	4
2 Level: RR_RR_8	N/A	RR	RR	8
2 Level: RR_RR_16	N/A	RR	RR	16
2 Level: RR_GTO_2	N/A	RR	GTO	2
2 Level: RR_GTO_4	N/A	RR	GTO	4
2 Level: RR_GTO_8	N/A	RR	GTO	8
2 Level: RR_GTO_16	N/A	RR	GTO	16
2 Level: GTO_GTO_2	N/A	GTO	GTO	2
2 Level: GTO_GTO_4	N/A	GTO	GTO	4
2 Level: GTO_GTO_8	N/A	GTO	GTO	8
2 Level: GTO_GTO_16	N/A	GTO	GTO	16

5.6.5. No Coalescing Penalty

In order to study the relationship between coalescing behavior and performance, I added a configuration option to GPGPU-Sim that removes the pipeline stall penalty associated with non-coalesced accesses. This configuration allows an SM to issue a warp instruction requiring multiple accesses in a single cycle. However, it does not further improve the memory pipeline to handle the increased memory traffic.

This configuration is not intended to model a realistic hardware improvement. Rather, it provides an understanding of the portion of the coalescing-related performance penalty that results from the ensuing pipeline stalls or replays versus the additional

memory accesses. I apply the no-coalesce-penalty configuration both by itself and in combination with increased-capacity cache miss queues and MSHRs. Table 5.5 details these configurations.

Table 5.5: No coalesce penalty configurations

Configuration Name	Coalesce Penalty ?	L1 Data Cache			L2 Cache		
		Miss Queue Entries	MSHR Entries	Max. MSHR Merges	Miss Queue Entries	MSHR Entries	Max. MSHR Merges
Default	Y	8	32	8	4	32	4
No Coalesce Penalty (NCP)	N	8	32	8	4	32	4
NCP + Improved L1 Miss Handling	N	16	64	16	4	32	4
NCP + Improved L1+L2 Miss Handling	N	16	64	16	8	64	8

5.7. Benchmark Compilation for GPGPU-Sim

Several of the studied benchmarks utilize the CUDA API routine *cudaFuncSetCacheConfig*, which sets (on a kernel basis) the L1 cache configuration to either 48 kB software-controlled shared memory and 16 kB hardware-managed cache or vice-versa. GPGPU-Sim 3.2.1 does not support this API call. Hence, I modified the benchmark codes to remove these calls and simulated each benchmark with the shared memory and L1 data cache sizes set according to the cache configuration used by the program's dominant kernels. In each case, I confirmed on real hardware that these changes result in negligible runtime impact.

The BFS benchmark includes a call to *cudaSetDeviceFlags* to enable pinned memory allocation by the host. This CUDA library function is not supported in GPGPU-Sim, and the benchmark does not actually allocate pinned memory. Therefore, I removed this library call from the BFS code.

The DMR benchmark calls the CUDA 5.5 library cosine function, which includes an inlined PTX multiply-accumulate instruction format that the simulator's parser does not support. I modified DMR to revert to the CUDA 4.2 implementation of cosine.

The CUB library's *DeviceRadixSort* implementation (which is used by the SP benchmark) calls the CUDA runtime API functions *cudaDeviceGetAttribute* and *cudaDeviceGetSharedMemConfig* in order to allow a change in the shared memory vs. L1 data cache configuration during the radix sort. GPGPU-Sim 3.2.1 does not support these API calls, nor does it support changing the L1 cache configuration mid-simulation. I modified *DeviceRadixSort* to remove its dependence on the unsupported API calls.

Two of the data-driven graph implementations in LonestarGPU, *BFS-wlc* and *SSSP-wlc*, rely on texture memory. These codes cause GPGPU-Sim to print a warning message that the implementation of cudaRegisterTexture() is incomplete. It is unknown how the simulator's current texture implementation affects the results for these benchmarks.

All benchmarks except BH were compiled using *nvcc* from CUDA 5.5 with the '-cudart shared -O3 -arch=sm_20' flags. BH was compiled using *nvcc* from CUDA 4.2 with the '-O3 -arch=sm_20' flags and simulated on a version of GPGPU-Sim omitting the changes for CUDA 5.5 described in Chapter 5.2 above.

6. RESULTS AND ANALYSIS

This chapter studies the impact on the selected benchmark codes of several common sources of GPU performance limitation. Next, I examine each application individually and assess the dominant performance bottlenecks of each. I then evaluate the performance impact of several hardware modifications. Lastly, I discuss how input selection and type impact the results.

6.1. Sources of Performance Limitation

The LonestarGPU suite includes applications across a wide range of performance points. Figure 6.1 illustrates the instructions per cycle (IPC) of each studied application. The theoretical peak performance of the GTX 480 is 480 IPC.

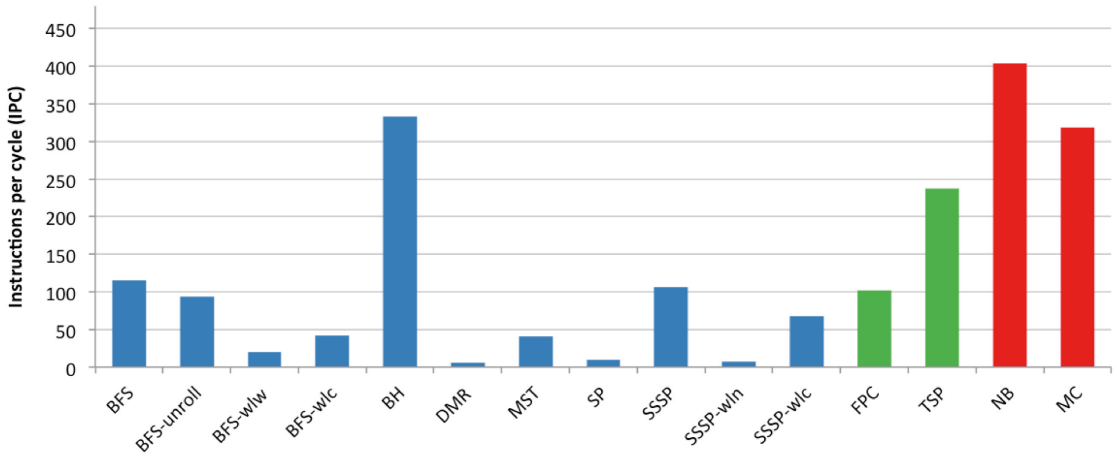


Figure 6.1: Measured instructions per cycle (IPC) of each benchmark

As expected, the regular programs NB and MC perform very well. More surprisingly, the irregular code BH also reaches a high IPC, though not in all of its kernels. The remaining irregular codes match or underperform FPC, the worst of the (semi-)regular codes. Overall, there is a clear tendency towards much lower IPCs for irregular codes and none of them come close to achieving peak performance. However, it

is also clear that there is no simple or fixed delineation between the performance of codes operating on irregular data structures and more regular codes.

It should be noted that IPC provides a useful picture of an application's utilization of the hardware but cannot be used to directly compare the runtime performance of different applications and implementations. Figure 6.2 displays the runtime in number of cycles of the several different implementations of BFS and SSSP included in the LonestarGPU suite. For both applications and across all tested inputs, the 'wlc' variant is significantly faster in reaching a solution than the default topological variant, even though the default implementation reaches higher IPC.

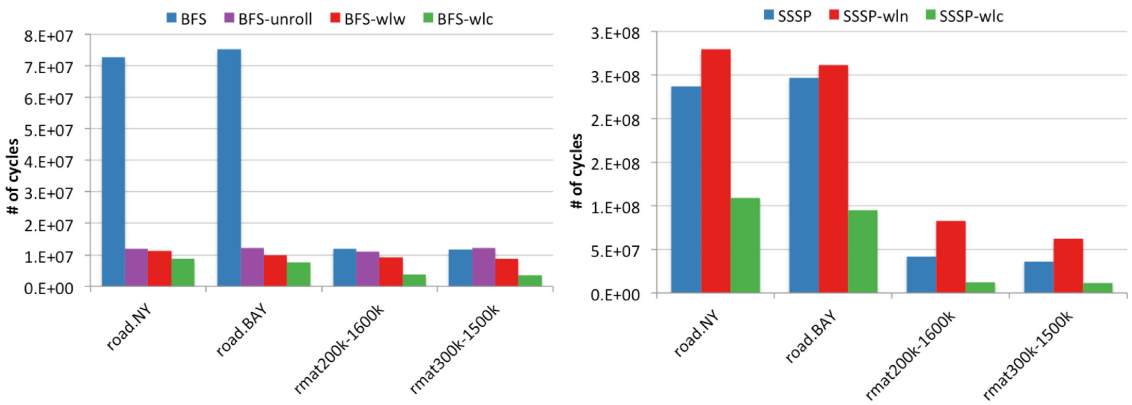


Figure 6.2: Runtime in cycles of all BFS and SSSP variants

GPU performance is heavily impacted by the presence of branch divergence within a warp and memory accesses that cannot be coalesced and/or miss in the cache. In this subchapter, I study the effect of these factors on each code.

6.1.1. Branch Divergence

Figure 6.3 plots the average warp occupancy based on the active mask of instructions with a warp at the issue stage in each scheduler. The first bar for each benchmark represents the average warp occupancy in only those cycles where a warp instruction was issued to the core pipeline. The second bar represents the average warp

occupancy across all cycles of the simulation, including cycles in which a scheduler did not issue any warp due to a stall or idle condition.

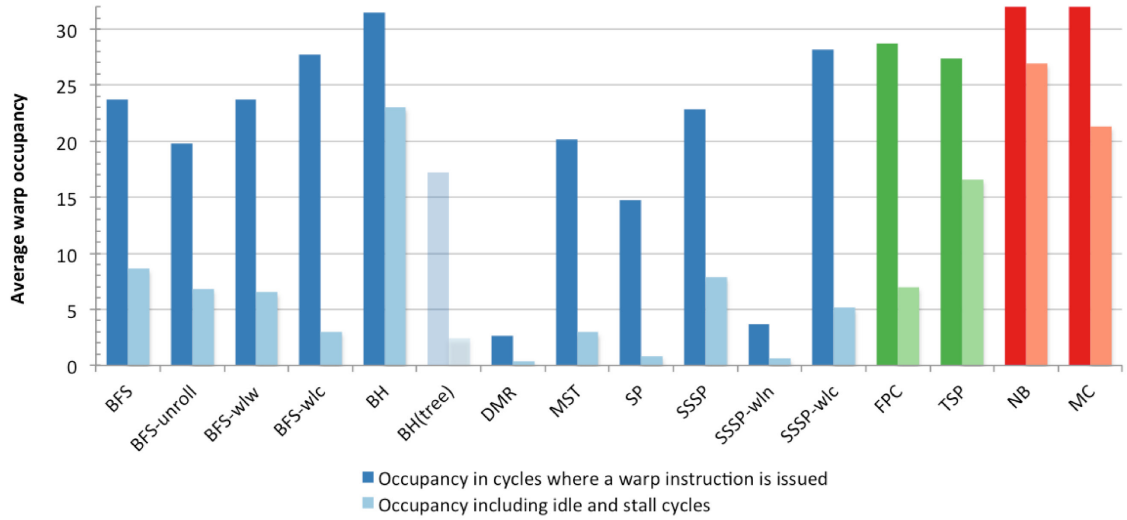


Figure 6.3: Average warp occupancy of each application, both including and excluding idle/stall cycles

The left bar for each application provides a graphical representation of the amount of branch divergence. A program with no branch divergence would reach an average warp occupancy (not including idle and stall cycles) of 32. Indeed, the two highly regular codes I study have average warp occupancies very close to 32, implying they do not suffer from branch divergence. As expected from codes operating on an irregular data structure, the irregular codes tend to display lower warp occupancies. However, only DMR and SSSP-wln fall below around half-occupied. Of the irregular codes, only BH has very little branch divergence because its force calculation kernel, which accounts for 95% of its runtime, has been implemented in a warp-based manner to improve performance. In contrast, the tree-building kernel, denoted in Figure 6.3 as ‘BH(tree)’, exhibits significantly more control irregularity.

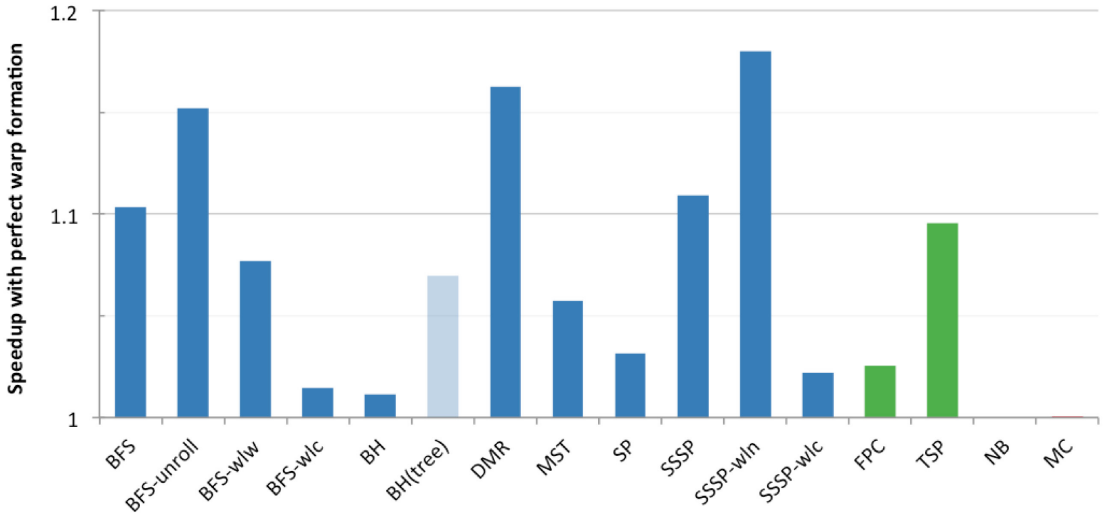


Figure 6.4: Benchmark speedup with perfect warp formation

Figure 6.4 illustrates the speedup that each benchmark would achieve given perfect warp formation, i.e., if each cycle in which an issue could be made issued 32 instructions per scheduler. Overall, with the exception of BFS-unroll, DMR, and SSSP-wln, branch divergence is somewhat less severe a penalty than expected from irregular codes. DMR displays severe load imbalance between threads as the refinement process runs out of work, leading to the high divergence. SSSP-wln includes control flow around a section of code that executes only if a thread finds a new shortest path to a node. It is unlikely that more than a handful of threads in any warp will do so in an iteration.

The right bar of each application in Figure 6.3 illustrates the impact on warp occupancy of issue stalls, particularly due to uncoalesced accesses and memory latency. The next subchapters investigate these factors in more detail.

6.1.2. Memory Coalescing

Figure 6.5 plots the average number of memory accesses performed by each global or local load or store instruction. A bar height above one illustrates the presence of uncoalesced memory accesses.

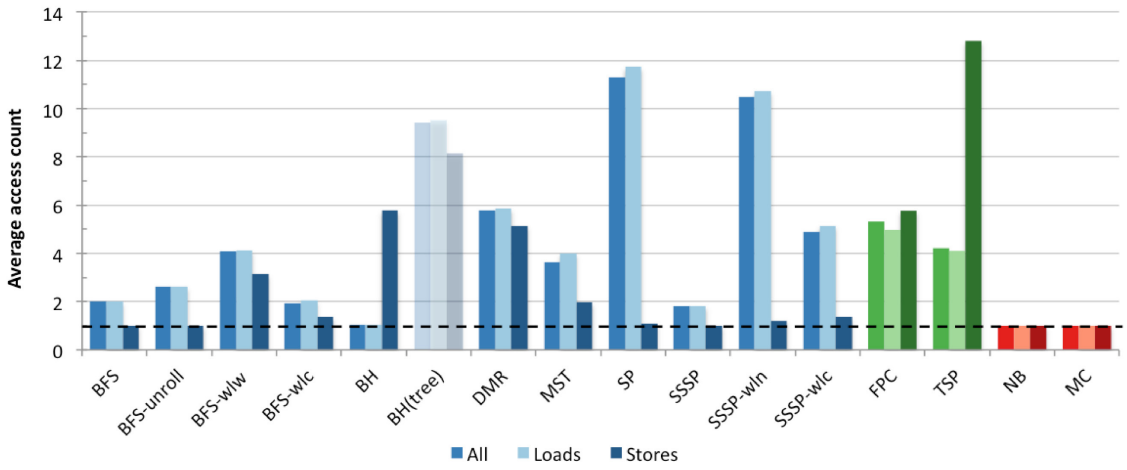


Figure 6.5: Average memory access count per global or local warp load or store instruction

The highly regular applications NB and MC have average access counts of 1, meaning that essentially all loads and stores are fully coalesced, i.e., each access by a warp results in a single memory transaction. BH and TSP perform very few but highly uncoalesced stores. On average, BH's loads are almost all coalesced, but this is again due to the dominant regularized force calculation kernel. The BH tree-building kernel, in contrast, exhibits one of the highest load access counts of the studied codes. TSP possesses a data-dependent and byte-granular memory-access pattern and thus exhibits highly uncoalesced accesses. FPC suffers from a high average access count resulting from its data-dependent byte-granular memory accesses, since stores to two bytes in the same word are serialized by the hardware. SP and SSSP-wln both have very high average load-access counts. SP makes many scattered accesses around a randomly-connected graph, leading to the high uncoalescing. SSSP-wln operates on nodes popped from a worklist of those nodes whose values changed in the last iteration and then reads destination node data. It is unlikely that nearby nodes on the worklist (and therefore nearby threads) will traverse similar segments of the graph, leading to many scattered and uncoalesced reads.

BFS, BFS-unroll, and the SSSP variants have fully coalesced stores, and the topological BFS and SSSP implementations display only slightly increased load access counts. However, their high load instruction counts result in significant slowdown from coalescing, as seen in Figure 6.6, which displays the percentage of simulation cycles in each benchmark that the simulator marks as stalls due to coalescing.

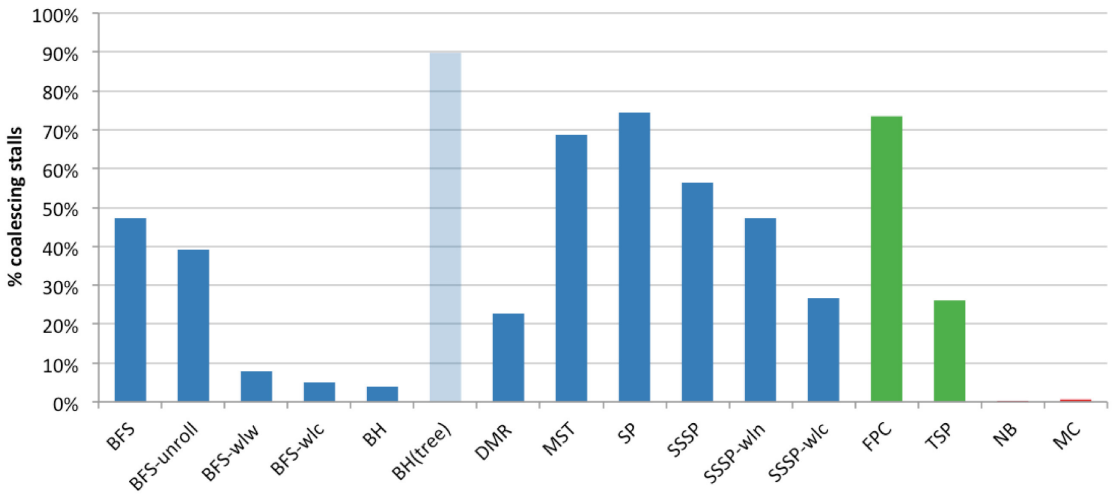
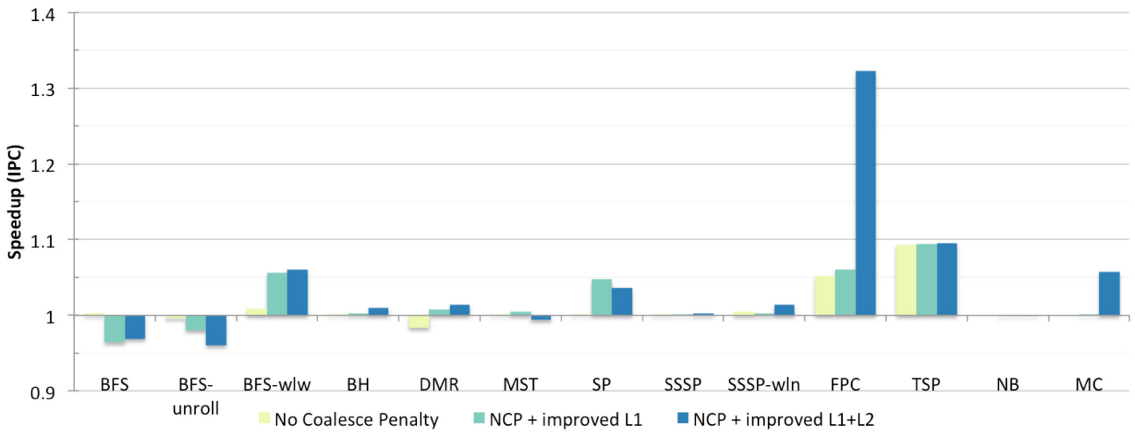


Figure 6.6: Percentage of cycles marked as coalescing stalls

This figure provides a visualization of the theoretical speedup that would result from somehow entirely removing the coalescing requirements. To further study the relationship between coalescing and performance in these benchmarks, I added a configuration to GPGPU-Sim that removes the pipeline stall penalty associated with non-coalesced accesses. This configuration allows an SM to issue a warp instruction requiring multiple accesses in a single cycle. However, it does not further improve the memory pipeline to handle the increased memory traffic. It is not intended to model a realistic hardware improvement, but it provides some visualization of the amount of coalescing performance penalty that comes from increased memory traffic versus pipeline stalls.

I studied each code with the no-coalesce-penalty (NCP) configuration applied by itself and in combination with increased-capacity cache miss queues and MSHRs. Figure 6.7 plots the speedups over the default setting for each of the NCP configurations.



(Note: Does not include data for BFS-wlc and SSSP-wlc due to assertions in texture memory simulator code)

Figure 6.7: Speedup over the default simulator configuration when removing the coalescing pipeline penalty and increasing the cache miss handling capacities

As intuitively expected, simply removing the pipeline penalty associated with coalescing stalls has little impact on performance (and is in some cases harmful) due to a corresponding increase in cache reservation stalls and interconnect backup. More surprisingly, for most of the benchmarks, improving the miss-handling capability of the caches does little to improve the performance impact of removing the coalescing stall penalty. (FPC is an outlier due to its many serialized byte-granular stores, which are counted as coalescing stalls). This suggests that hardware improvements aimed at reducing the coalescing pipeline penalty are likely to be ineffective on irregular codes unless they are combined with increased memory bandwidth.

6.1.3. Cache Behavior

Lastly, I observe the data cache miss ratio (in Figure 6.8) and miss rate (in Figure 6.9) in misses per thousand warp instructions (MPKI) for each program. Most of the

applications, including the highly regular codes, have L1 cache miss ratios above 50%, which would be considered extremely high for CPU applications. Note that CPU and GPU architectures have L1 data caches for different reasons: in GPUs, they mostly provide coalescing support rather than exploit temporal locality, because there cannot be an expectation of the cache holding data for a significant period of time due to the high number of active threads.

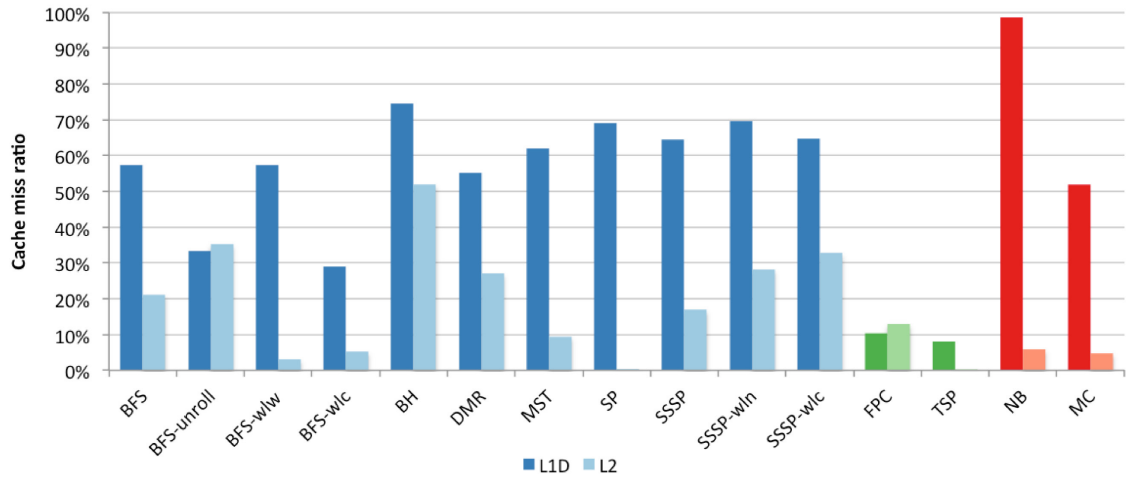


Figure 6.8: Data cache miss ratios (L2 ratios are local, i.e., number of misses over number of L2 accesses)

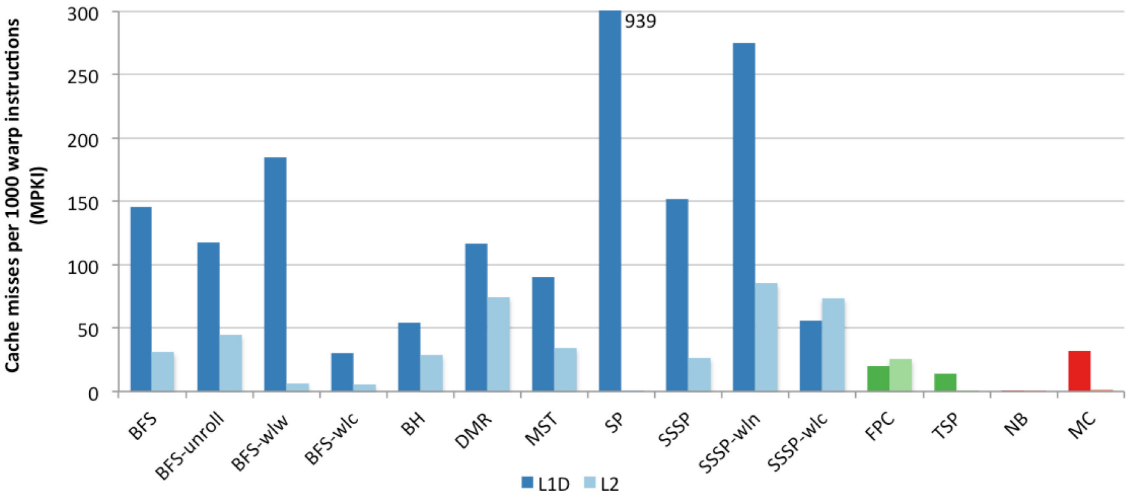


Figure 6.9: Cache misses per thousand warp instructions (MPKI)

However, most of the irregular codes have markedly higher MPKI rates than the regular codes. NB, which tiles its data into shared memory and is compute-bound, has MPKI rates near zero for both data caches. TSP is composed mostly of shared memory accesses and many of its local memory accesses are strided, which is why this program has one of the lowest observed miss rates. FPC is a streaming code and therefore also exhibits a relatively low miss rate. SSSP-wln's high average load access count results in a high MPKI and contributes to the longer runtime of this data-driven SSSP variant compared to the topological implementation. BFS-wlc's miss rate is low, presumably because much of the worklist fits within the L1 cache. SP (whose L1 miss rate exceeds the Figure 6.9 axis by more than a multiple of three) has the highest average access count of the studied codes (i.e., it is the most poorly coalesced), leading to an extremely high miss rate even at my relatively small input size. The small SP input size (limited by simulation time) results in an L2 miss rate near zero. In general, the irregular codes perform a significant number of pointer-chasing operations and are not able to exploit much spatial locality.

6.2. Individual Application Analysis

As described in Chapter 5.4, I supplemented GPGPU-Sim's warp issue metrics with additional stall type counters. Based on the resulting warp occupancy histogram and stall distributions, I calculated (on a per-application basis) the number of issue cycles with underused thread occupancy due to branch divergence, control flow, memory and synchronization barriers, atomics, scoreboard hazards, functional unit stalls, and work imbalance between blocks, as well as cycles in which the GPU issued at full occupancy (*busy* cycles). Scoreboard hazards include both read-after-write (RAW) and write-after-

write (WAW) hazards but are dominated by RAW hazards in all of the codes. Because outstanding loads cause the majority of these RAW hazards, the scoreboard hazards metric provides a rough estimate of the impact of memory latency. In addition, load/store unit (LSU) pipeline stalls reflect both coalescing penalty and cache reservation fails; the latter is also an indication of memory-latency-associated slowdown.

Figure 6.10 below displays the breakdown of underused and busy cycles based on the prioritized-stall-cause tracking illustrated in Figure 5.1, which in each cycle increments a counter for the stall cause associated with the warp whose no-issue decision was resolved in the deepest pipeline stage. Figure 6.11 below displays the breakdown of underused and busy cycles based on a histogram formed by, in each cycle, incrementing the bin associated with the cause responsible for the greatest number of available warps that could not issue.

I discuss each code in detail below and then draw some general conclusions.

- *Breadth-First Search—Topological (BFS)*

This code suffers from a high number of LSU stalls (full LSU pipeline) and RAW hazards (operations waiting on load data), both of which are indicative of memory-latency-related slowdown resulting from the data-dependent nature of BFS's memory accesses (based on the connectivity of the input graph). The BFS implementation further displays some control-flow irregularity due to graph nodes having different numbers of edges, which results in branch divergence.

- *Breadth-First Search—Unroll (BFS-unroll)*

The BFS-unroll code processes multiple frontiers per iteration by maintaining a worklist of changed destination nodes in shared memory. Its runtime is significantly

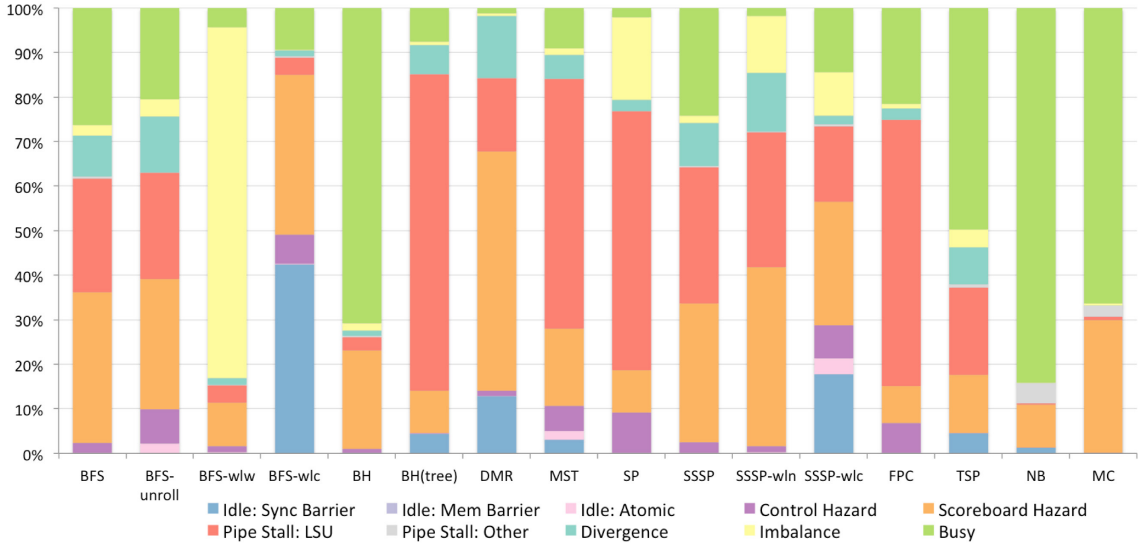
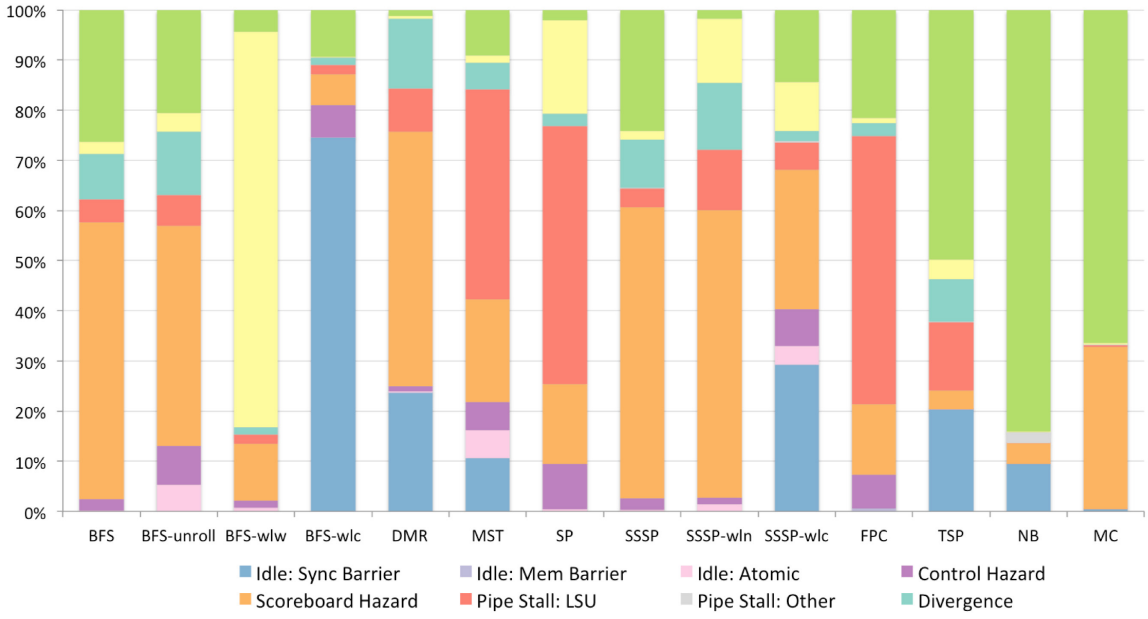


Figure 6.10: The proportion of underused vs. fully-occupied cycles in each application (deepest pipeline stage)



(Note: BH is omitted because this histogram instrumentation was not included in my CUDA 4.2 simulator version)

Figure 6.11: The proportion of underused vs. fully-occupied cycles in each application (most impacted warps)

lower on the same inputs as the default BFS implementation. Compared to the default implementation, it displays less memory penalty because of the data-driven nature of the unrolled segments of frontier processing, exposing more of the atomics required to update node levels. The reduction in memory-related slowdown also exposes cycles that suffer from divergence and control-flow hazards due to the relatively small loop trip count over outgoing edges.

- *Breadth-First Search—Data-driven (BFS-wlw)*

This code implements BFS in a data-driven manner, processing nodes from a worklist of active nodes and assigning one node per thread. It suffers from severe interblock imbalance because of the low parallelism in frontier nodes available during much of its execution.

- *Breadth-First Search—Merrill's strategy (BFS-wlc)*

The BFS-wlc code operates on edge and vertex frontiers stored in global memory and therefore relies on global barriers (implemented by having blocks check in and then spin-wait for all check-ins via global memory and requiring several synchronization barriers). It also uses the CUB library's block scan primitive, which relies on intra-block synchronization. Of my studied codes, this implementation displays by far the largest penalty from synchronization, with stalls caused by a majority of warps available for issue stuck at a barrier in around 75% of its cycles. This benchmark also results in the most significant difference between the two methods of counting stall causes illustrated in Figures 6.10 and 6.11. In many of the stall/idle cycles of the simulation, there were available warps that made it past the barrier stage and still could not issue because of RAW hazards on load data.

- *Barnes-Hut (BH)*

The BH code, while tree-based and irregular in nature, is dominated by its force calculation kernel, which has been optimized to eliminate almost all divergence and to ensure that most of its main memory accesses are coalesced. As a result, BH spends a larger percentage of execution time at full occupancy than the other irregular codes. It should be noted that BH's warp threads perform some unnecessary computation to minimize branch divergence, and these unnecessary cycles are denoted as busy cycles rather than divergence. However, the unnecessary work improves both the performance and the accuracy of the algorithm. Figure 6.10 above also includes metrics for the BH tree-building kernel. This kernel exhibits significantly more irregularity, suffering from both memory-access stalls and branch divergence. It also exhibits a noticeable performance penalty due to the synchronization barriers necessary to build the tree in parallel. There is a small amount of work imbalance as it is a priori unknown how deep the various branches of the octree will be.

- *Delaunay Mesh Refinement (DMR)*

DMR has the lowest IPC of all the codes I examine and is one of the most irregular as well. It suffers from a large amount of memory-access stalls, divergence, and one of the larger fractions of synchronization stalls of the studied codes. For each bad triangle, DMR's refinement kernel builds a cavity whose size and shape are data dependent, checks whether there is overlap with another cavity, employs a priority-based back-off mechanism in case of overlap, and finally refines the cavity if the thread has the highest priority of all the triangles in the cavity. The synchronization stalls stem from global barriers separating these phases, the divergence is the result of load imbalance

between threads, and the memory stalls are likely unavoidable when processing an irregular graph whose shape changes at runtime.

- *Minimum Spanning Tree (MST)*

MST spends the majority of its cycles waiting for uncoalesced load data due to the irregular nature of its accesses to the merged graph nodes, or components. The innermost loop contains a set of nested *if* statements to unify the minimum-weight components, resulting in both significant divergence and control-flow penalties. MST displays one of the higher fractions of slowdown associated with atomics of the codes I study due to the atomic operations necessary to merge components; however, it is still a minor source of performance loss compared to memory-related stalls.

- *Survey Propagation (SP)*

SP traverses and operates on a random 3-SAT input graph, resulting in randomly scattered memory accesses and a large number of coalescing stalls (visible in my metrics as a large amount of LSU pipeline stalls). It suffers from control hazard idle cycles due to the very short loop body and tiny loop count (over the number of literals per clause) of its innermost loop. The code exhibits imbalance due to its statically unknown amount of work that depends on how fast the algorithm converges as well as divergence due to separate code paths for positive and negative edges. Overall, this application exhibits very few fully occupied execution cycles.

- *Single-Source Shortest Paths—Topological (SSSP)*

This algorithm is similar to the topological BFS implementation except that SSSP processes a directed, weighted graph. Similarly to BFS, its performance is limited by LSU and RAW hazard stalls resulting from uncoalesced memory accesses and

insufficient memory bandwidth as well as branch divergence due to control-flow irregularity when processing the input graph.

- *Single-Source Shortest Paths—Data-driven (SSSP-wln)*

The SSSP-wln code is a data-driven implementation where threads process nodes from a worklist similarly to the BFS-wlw code. There is a small amount of slowdown due to the atomic operations necessary to update the newly-found shortest distances within destination nodes as well as significant interblock imbalance as threads run out of work and divergence due to both imbalance and differing control flow for adjacent threads processing nodes in worklist order. The worklist variant of SSSP displays a much larger memory-related penalty than the similar BFS code due to RAW hazards from the additional load for the atomic compare operation, whose return value is immediately used to resolve a branch condition. The scattered and uncoalesced nature of these accesses, based on the order of nodes in the worklist, make SSSP-wln's slower than the topological SSSP variant.

- *Single-Source Shortest Paths—Merrill's strategy (SSSP-wlc)*

This code is similar to the BFS-wlc implementation, and it likewise suffers from a very high synchronization penalty. Like SSSP-wln, however, SSSP-wlc suffers from both atomics penalty and additional RAW hazards due to the atomic operations that update new shortest paths. BFS-wlc also suffers from some interblock imbalance as threads run out of work.

- *Floating-Point Compression (FPC)*

This application spends most of its time stalled due to a full load/store unit pipeline. These stalls stem from the coalescing behavior of its double-precision memory

accesses as well as warp threads reading and writing data-dependent byte locations in global memory. FPC also exhibits a large percentage of control-flow-hazard stalls due to the short (maximum iteration count of 8) data-dependent loops that read and write the compressed bytes corresponding to an uncompressed double. FPC further suffers from some branch divergence due to imbalance in the number of bytes processed by each warp thread, as well as a code section in which only every other thread has work.

- *Traveling Salesman Problem (TSP)*

The TSP code is semi-regular, possessing relatively regular control flow but data-dependent memory accesses, and it spends the majority of its execution time on computation. Its memory accesses are mostly to shared memory, but it does access main memory when performing the 2-opt city ordering swaps, resulting in uncoalesced accesses and LSU stalls. TSP also exhibits a large fraction of idle time associated with synchronization barriers. These barrier idle cycles occur as threads in a block finish computing their locally best solution but have to wait for the slowest thread in the block before the global solution can be updated. The branch divergence stems from an instance of control-flow irregularity where some threads have reached a local minimum and want to move on to a new tour while other threads are still searching for a minimum.

- *N-Body (NB)*

This code is highly regular and computation-bound. Additionally, I chose an input size to exactly fill the resident blocks to provide a basis of comparison for the irregular benchmarks. NB demonstrates the highest busy ratio of the studied programs. Its lost cycles are due mostly to RAW hazards on the small amount of data not accessed via

shared memory as well as full computation pipelines and the synchronization barriers necessary between transferring data to shared memory and the computation phase.

- *Monte Carlo (MC)*

The MC application is embarrassingly parallel, highly regular, and dominated by computation. Its largest source of slowdown is scoreboard hazards, both due to cache misses and long computation operations. It also displays some imbalance between blocks as well as pipeline stalls in the special function unit (SFU) pipeline due to its extensive use of square root operations in the quasi-random sequence generation kernel.

Many irregular codes rely on synchronization and memory barriers and atomic operations. With the exception of the synchronization penalty in BFS-wlc and SSSP-wlc codes, however, these primitives contribute a smaller fraction of program slowdown than expected. Memory fences appear to contribute very little to program slowdown for these codes. The performance loss associated with imbalance and branch divergence was also somewhat less severe than expected of codes operating on irregular data structures. This suggests that these benchmark codes have been successfully optimized to minimize the performance impact of these aspects [64][65]. In line with expectation, memory bandwidth appears to be the most significant performance bottleneck for highly optimized irregular GPU applications, suggesting this bottleneck is more difficult to address with source-code optimizations.

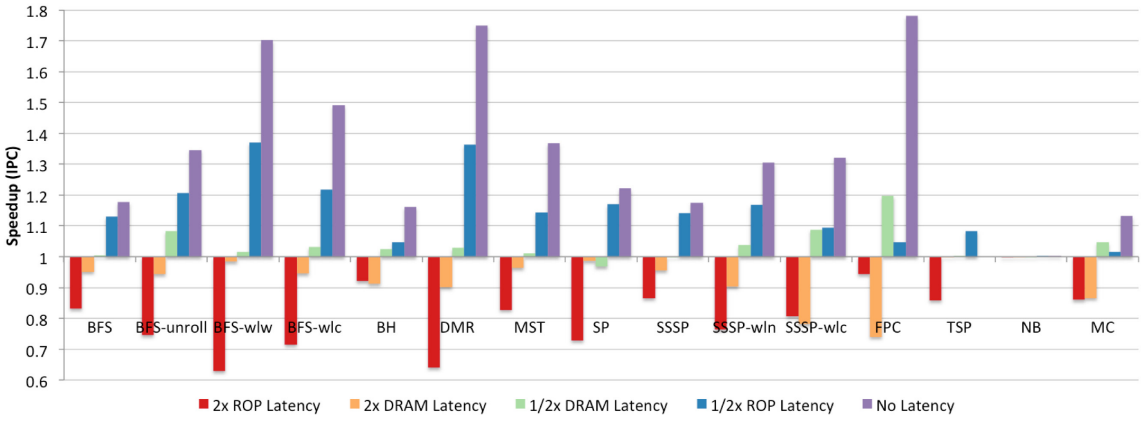
6.3. Impact of Hardware Modifications

Next I examine the performance impact of several hardware modifications in order to better understand the behavior of irregular codes and their sensitivity to various hardware parameters. The hardware configurations discussed in this subchapter are

described in Chapter 5.6. All speedups in this chapter are calculated using IPC. Several of these codes iterate until convergence is reached, making direct cycle count (i.e., runtime) comparisons meaningless because even minor changes in microarchitectural timing can alter the number of iterations and therefore the number of instructions executed. IPC comparisons normalize for these fluctuations.

6.3.1. Cache and Memory Latency

I begin by scaling the L2 hit latency (ROP latency) and the DRAM access latency. I examine the runtime (in cycles) of each benchmark with both latencies configured to zero as well as with the latencies doubled and halved. Figure 6.12 plots the speedup relative to the default setting for each latency configuration.



(Note: TSP does not include data for the No Latency configuration due to a simulator deadlock)

Figure 6.12: Speedup over the default simulator configuration when scaling the minimum L2 hit latency and DRAM latency

Interestingly, nearly all of the studied benchmarks are more sensitive to the L2 latency than the DRAM latency, even in the presence of working set sizes several times larger than the L2 capacity. The exception is FPC, which accesses data in a streaming manner and displays high spatial locality. The regular NB code uses tiling to read all its data into shared memory and is largely compute bound; it is thus insensitive to memory

latencies. At least for these inputs, L2 latency appears to be more important than DRAM latency for the performance of GPGPU codes, especially irregular ones.

6.3.2. Cache and Memory Bandwidth

Next I scale the interconnect bandwidth between the memory partitions (including the L2 cache) and the core, as well as the DRAM bandwidth. Figure 6.13 illustrates the performance impact on each benchmark of halving and doubling the interconnect and DRAM bandwidths, both individually and in combination with one another.

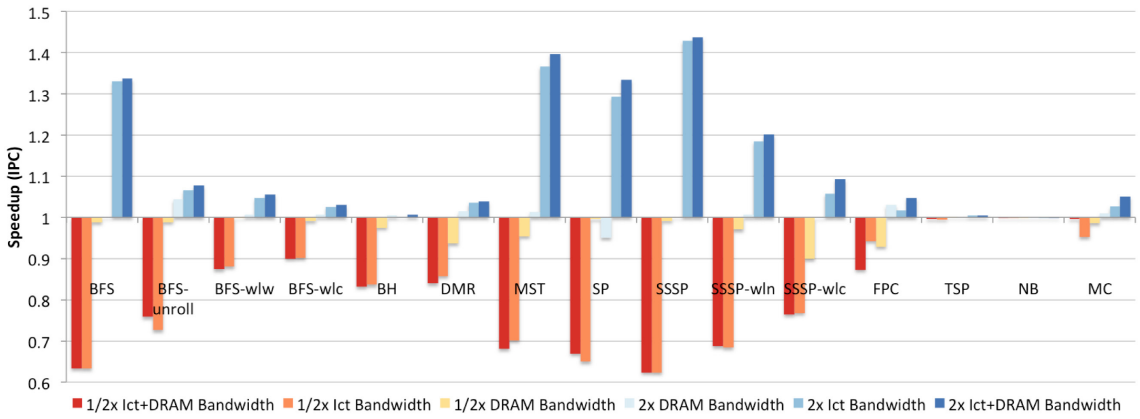


Figure 6.13: Speedup over the default simulator configuration when scaling the interconnect and DRAM bandwidths

Similarly to the L2 and DRAM latency behavior, most of the studied applications are significantly more sensitive to interconnect bandwidth than to DRAM bandwidth. It seems that for these applications and input sizes, the L2 is large enough that sufficient L2 bandwidth keeps enough warps able to execute. The regular codes are helped very little by additional memory bandwidth.

6.3.3. Cache Size

Figure 6.14 plots the performance impact on each benchmark of halving and doubling the L1D and L2 cache sizes over the default configuration.

In general, those benchmarks that are significantly sensitive to interconnect bandwidth also benefit most from increased L1 data cache size. One exception is the irregular BH tree-construction kernel. This kernel traverses tree prefixes beginning with the root of the tree. The top of the tree is therefore likely to hit in the L1, but after the top portion of the tree there is insufficient locality to leverage even a larger L1. Increased bandwidth to the L2, however, improves performance because the L2 allows for significant exploitation of locality in the traversals.

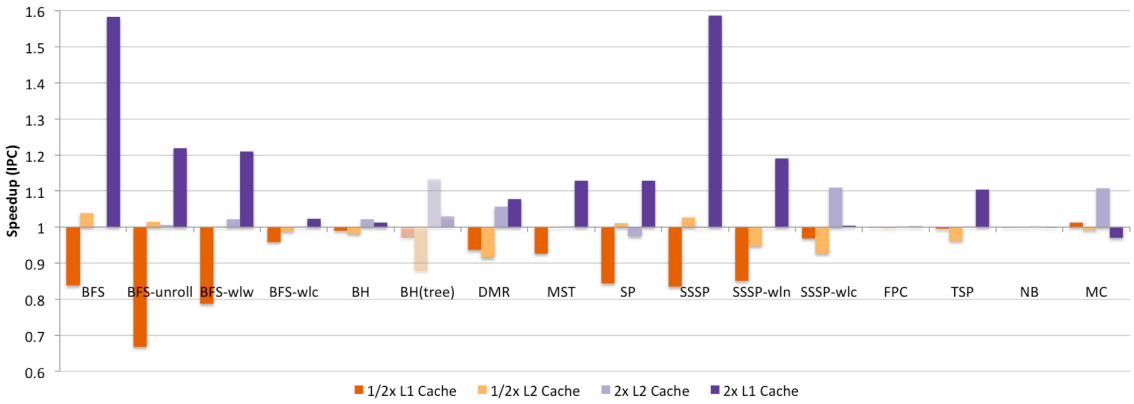


Figure 6.14: Speedup over the default simulator configuration when scaling the data cache sizes

Most of the irregular codes are hurt more by decreased L1 capacity than by decreased L2 capacity. The regular codes, on the other hand, are impacted more by decreased L2 capacity, but the impact is relatively minor in comparison to the impact of the L1 cache size on irregular codes. For these inputs, BFS, SP, and SSSP all display an unexpected speedup with a reduced L2 size. These applications iterate until they converge, and microarchitectural changes can alter the timing behavior, which in turn can affect the number of iterations and the order in which tasks are processed, potentially altering the code's memory locality and other performance factors and leading to counterintuitive performance changes.

6.3.4. Warp Scheduling Policy

By default, GPGPU-Sim implements a greedy-then-oldest (GTO) warp scheduler, which prioritizes a particular warp until that warp stalls, then prioritizes the oldest other warp eligible for issue. Figure 6.15 shows the performance impact of using a round-robin (RR) scheduler instead.

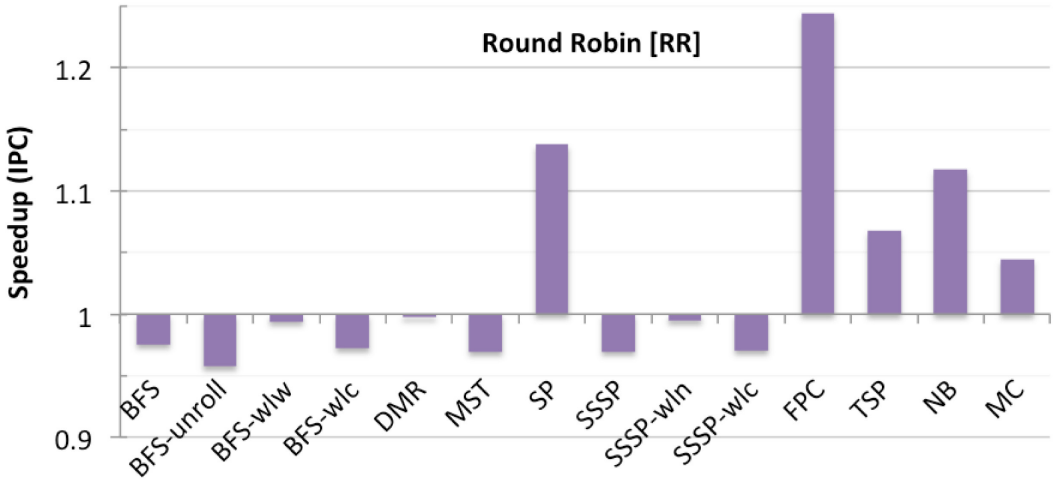


Figure 6.15: Speedup of the RR scheduler over the default GTO scheduler

With the exception of SP, the RR scheduler degrades the performance of the irregular codes. RR scheduling preserves locality between warps by keeping warps roughly synchronized. This has a significantly positive performance impact on the more regular codes, whose data-independent access patterns tend to assign nearby data locations to nearby warp instructions. RR scheduling improves the performance of FPC, for example, by nearly 25% because of a reduction in RAW hazards on load data.

While good for preserving inter-warp locality, the downside to RR scheduling is that warps tend to encounter long operations (i.e., stalls) at the same time. GTO scheduling, on the other hand, sacrifices inter-warp locality in order to address the stall issue. It appears that GTO warp scheduling is superior for irregular codes, which often possess little inter-warp locality.

Figure 6.16 displays the performance impact of my greedy-then-least-stalled (GTLS) warp scheduler with several configurations of clear_count (cf. Chapter 5.6.4). This scheduler trivially helps a handful of the irregular applications and in general hurts the irregular codes less than the RR scheduler while preserving some of the benefit to the highly regular codes. However, because the GTLS scheduler is greedy in nature, it still sacrifices some inter-warp synchronization and therefore locality, and its benefit to the regular codes cannot match the RR scheduler.

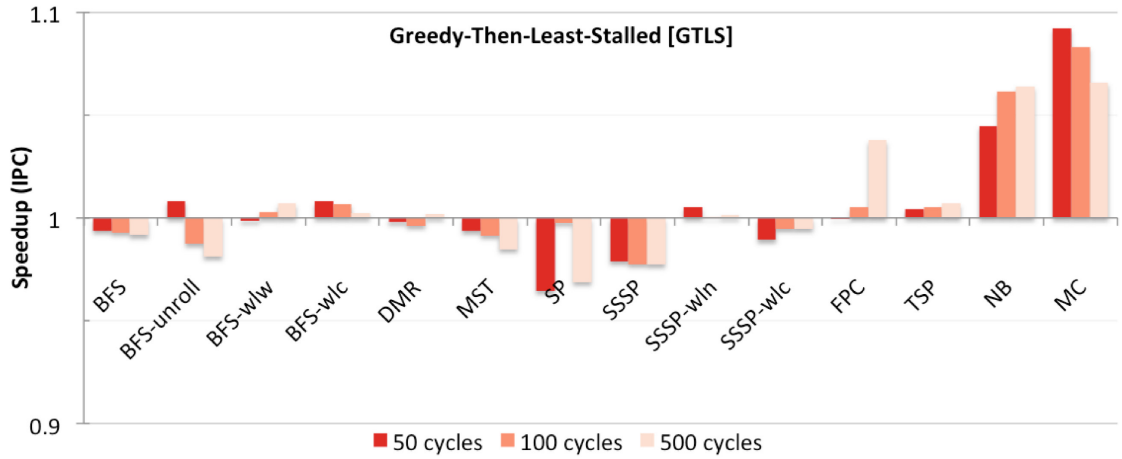


Figure 6.16: Speedup of the GTLS scheduler over the default GTO scheduler

Lastly I examine the performance impact on irregular codes of the two-level scheduler proposed by Narasiman et al. [26], using several configurations of inner and outer scheduling policies and fetch group size. Figures 6.17 – 6.19 below show the impact on the performance of each benchmark.

Overall, two-level warp scheduling appears detrimental to irregular codes. Small fetch group sizes are always harmful. Two-level scheduling benefits the more regular codes when employed with a round-robin selection policy; however, only two codes (FPC and MC) perform better with some configuration of the two-level scheduler than they did with the concrete RR scheduler.

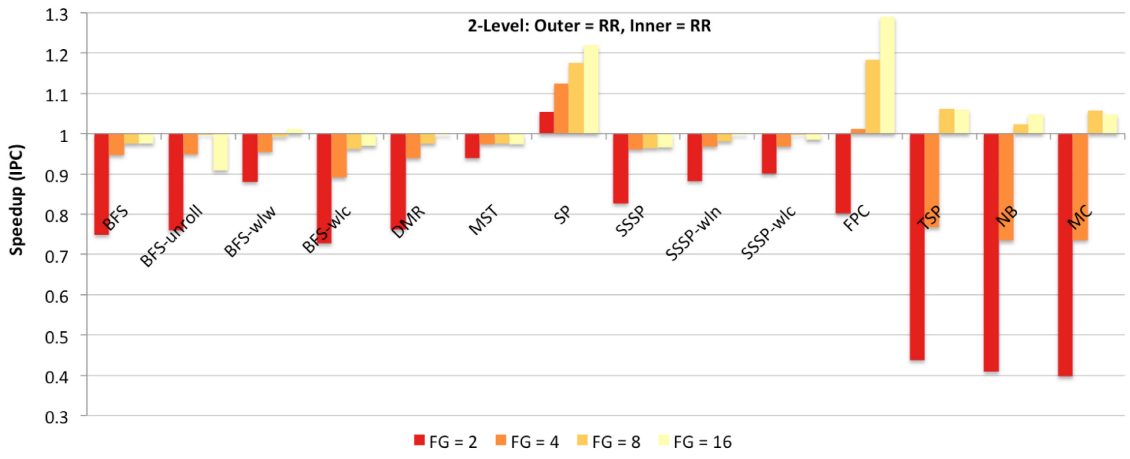


Figure 6.17: Speedup of the 2-level scheduler with RR inner and outer policies over the default GTO scheduler

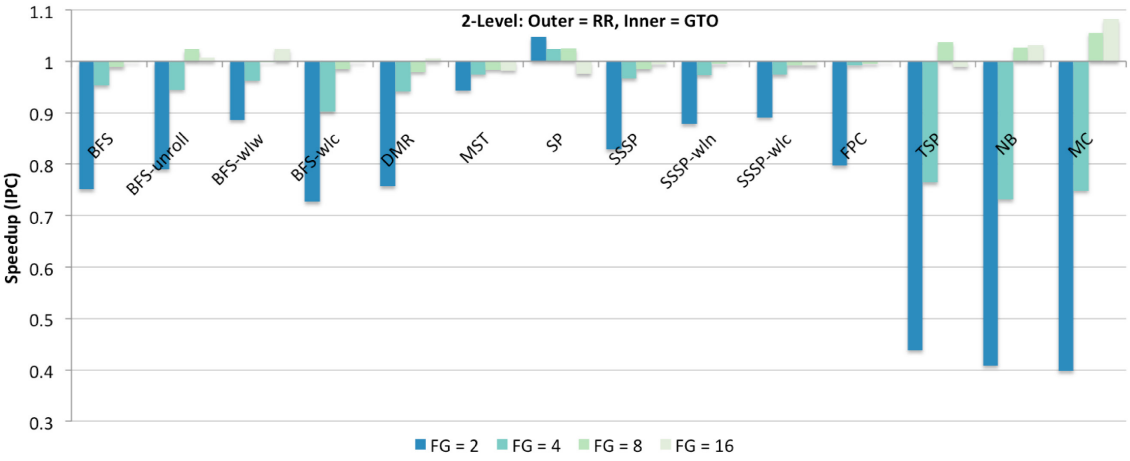


Figure 6.18: Speedup of the 2-level scheduler with an RR outer and GTO inner policy over the default GTO scheduler

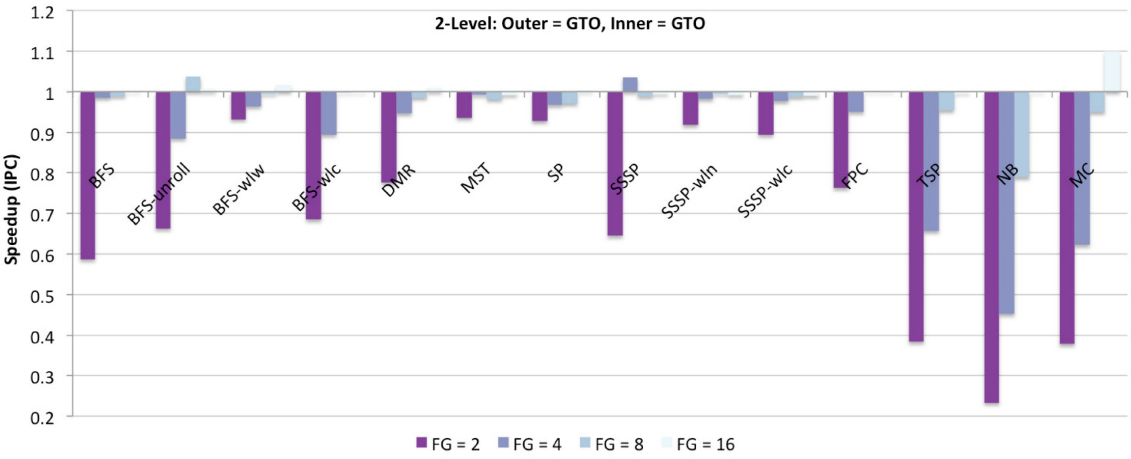


Figure 6.19: Speedup of the 2-level scheduler with GTO inner and outer policies over the default GTO scheduler

Of the simple warp schedulers proposed in the literature, GTO appears to be the best choice for supporting GPGPU execution of irregular codes. For these applications, further performance improvements from scheduling will likely require more complicated schemes for identifying and responding to sources of slowdown (e.g., cache contention).

6.4. Input Sensitivity

By definition, irregular codes display behavior that varies depending on the input data. In order to understand the impact of input variation on my results, I simulated each benchmark with several inputs (cf. Chapter 4.2) and compared the results.

Despite the input-sensitive nature of the codes, different inputs of similar size result in similar microarchitectural performance behavior. Figure 6.20 displays the IPC of each benchmark across multiple inputs of similar size and type (road networks in the case of all graph codes). As expected, the fully regular code, NB, maintains the same IPC regardless of input data. The irregular codes do display some IPC variation; however, the difference is minimal across all benchmarks.

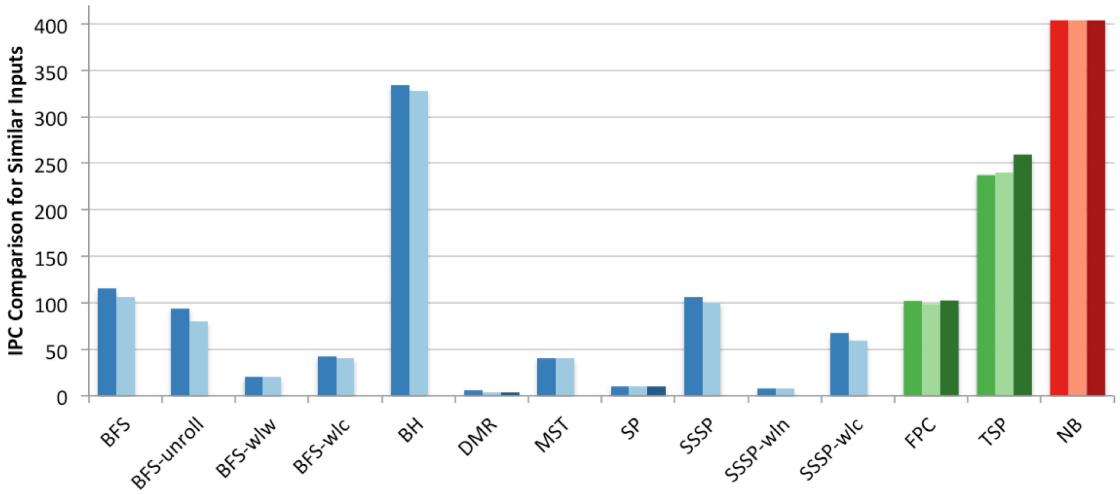


Figure 6.20: Instructions per cycle for each benchmark with several similar inputs

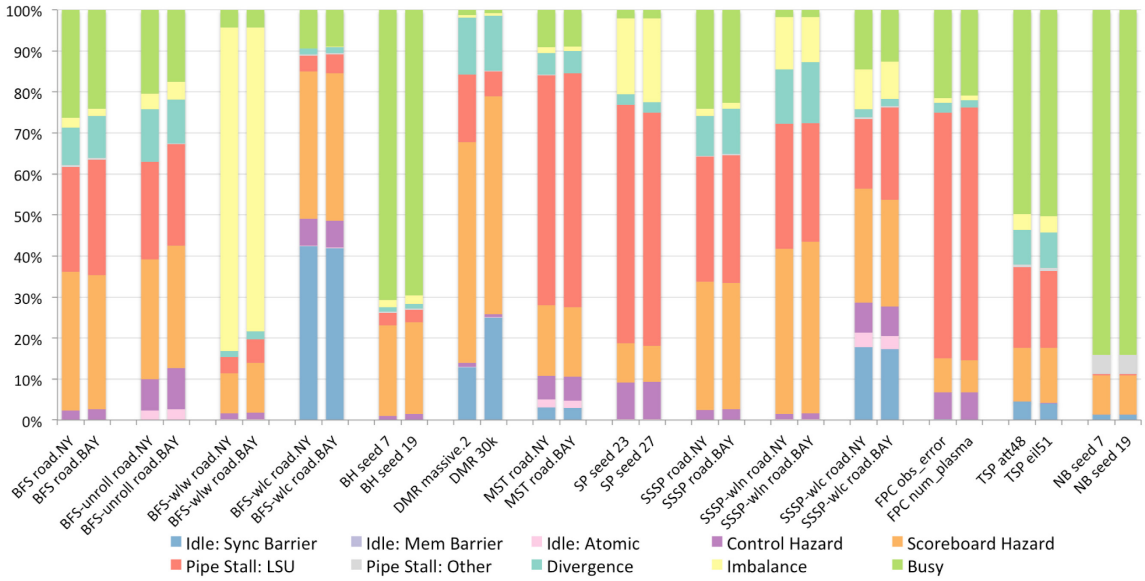


Figure 6.21: The proportion of underused vs. fully-occupied cycles (deepest pipeline stage) for all codes using several similar inputs

Figure 6.21 illustrates the breakdown of underused cycles in each benchmark using the same inputs as in Figure 6.20 above, based on the deepest-pipeline-stage-prioritized version of the histogram detailed in Chapter 5.4. Despite the input-dependent behavior of the code, the microarchitectural performance behavior of a given application remains largely consistent across inputs of similar size and structure.

6.4.1. Input Type

In order to observe their sensitivity to input type, I compared each graph benchmark (i.e., BFS, MST, and SSSP) with both its primary road-network input and an R-MAT input. Figure 6.22 plots the IPC of each graph code using the primary input of each type. Figure 6.23 displays the breakdown of underused cycles for each code and input, once again based on the deepest-pipeline-stage-prioritized histogram.

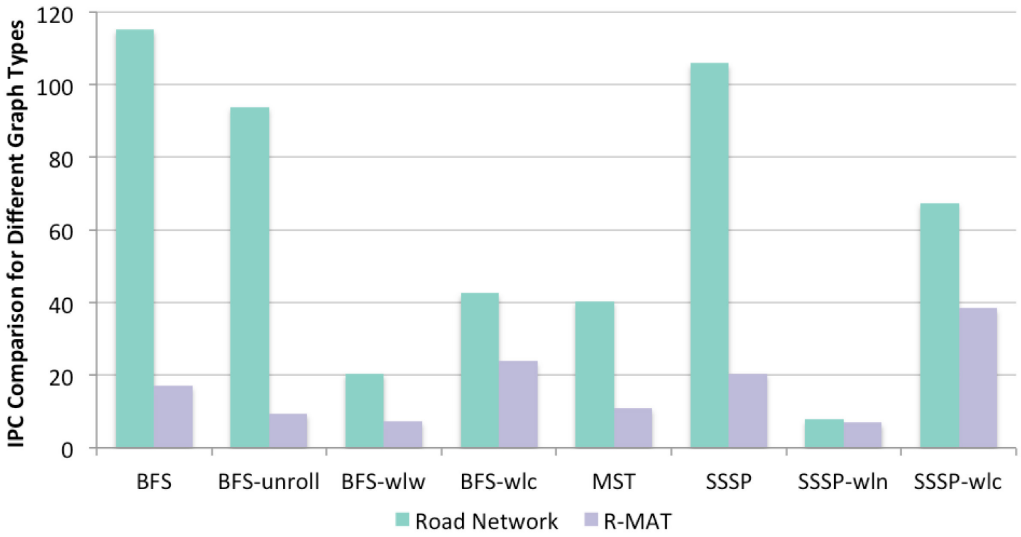


Figure 6.22: Instructions per cycle for all graph codes with road-network and R-MAT inputs

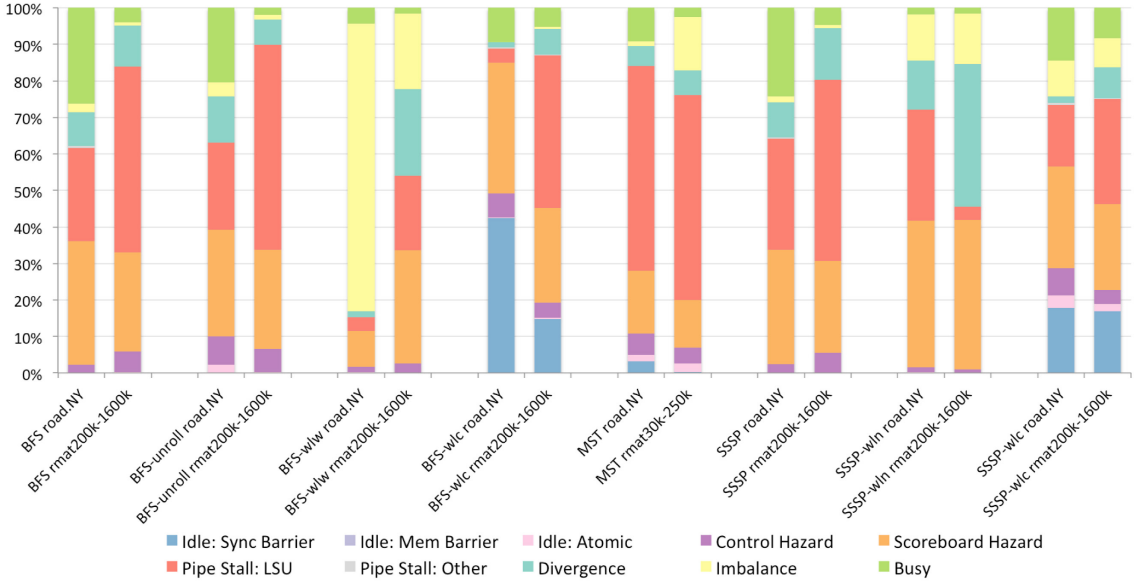


Figure 6.23: The proportion of underused vs. fully-occupied cycles (deepest pipeline stage) for all graph codes using road-network and R-MAT inputs

Variance in input type results in greater variation in performance characteristics. All of the graph codes perform at lower IPCs on R-MAT inputs. R-MAT graphs are denser and have higher and more varied out-degree per node than road networks, and this leads to a higher divergence penalty in most of the benchmarks. For the BFS and SSSP

implementations, the tested R-MAT graph is larger than the New York City road map input, which reduces inter-block imbalance in BFS-wlw and results in higher memory latency penalty in several of the codes. Due to the limitations of simulation time, the R-MAT input for MST is significantly smaller than the road network. However, MST's performance still degrades on the R-MAT graph due to the different graph structure.

Figure 6.24 shows the underutilized and busy cycles of the BFS and SSSP variants on a road network input and an R-MAT input of equal size. In general, even equally sized R-MAT graphs result in more divergence and more memory penalty than road networks.

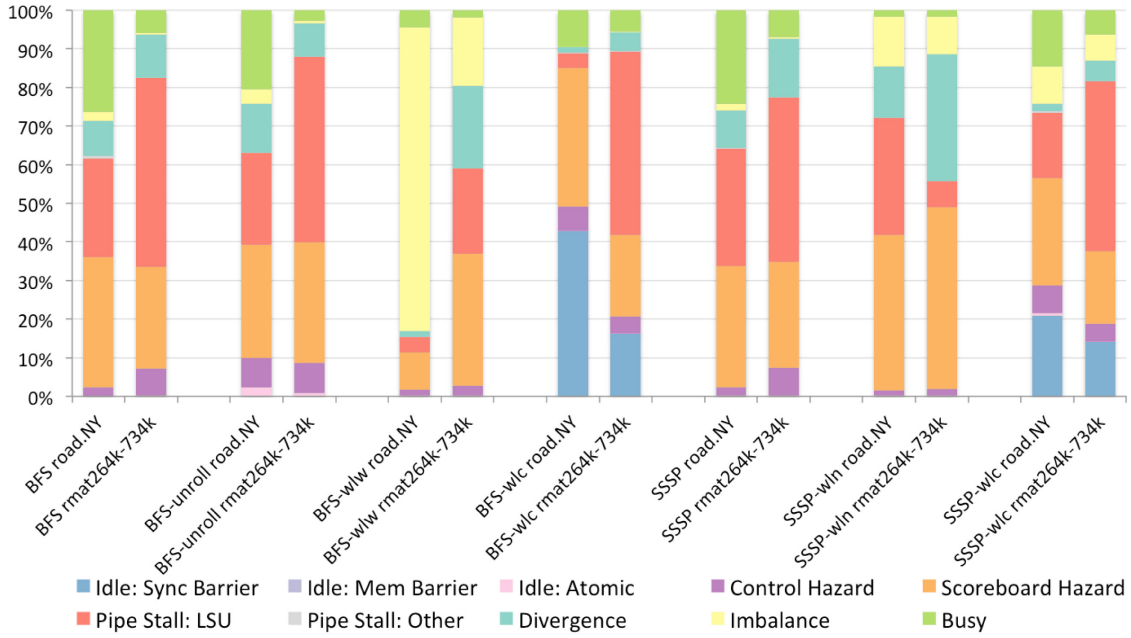


Figure 6.24: The proportion of underused vs. fully-occupied cycles for BFS and SSSP with equally-sized inputs of different types

7. SUMMARY AND CONCLUSIONS

This thesis presents a microarchitectural workload characterization focusing on irregular GPU codes. I study the impact of control-flow and memory-access irregularity on several performance aspects, analyze how this behavior differs from that of regular GPGPU programs, and characterize the sensitivity of irregular codes to changes in cache and DRAM latency and bandwidth, cache size, coalescing behavior, and warp scheduling policy. Additionally, I connect source code to particular microarchitectural performance characteristics.

As expected, even extensively hand-optimized graph and tree codes running on GPUs tend to achieve lower IPCs than regular codes. In general, they exhibit greater performance loss due to load imbalance, branch divergence, and uncoalesced memory accesses resulting from the unpredictable nature of their control-flow and memory-access patterns. This general trend is not always true, however. BH, for example, builds and operates on an irregular data structure (an octree) but, due to targeted code optimizations, displays less memory irregularity than FPC, which does not operate on a dynamic data structure.

Interestingly, for the most part, load imbalance and branch divergence limited the performance of these (presumably comprehensively hand optimized) irregular codes less than expected. For many of the studied benchmarks, this was also true of the performance limitation of synchronization and atomic operations. Memory-related slowdown appears to be the single biggest factor limiting the performance of irregular applications, even those that have been demonstrated to possess ample parallelism [66], because irregular memory-access patterns appear to be difficult to regularize and coalesce.

I find that, for my sizeable tested inputs, improving the L2 latency and bandwidth is more important than improving the DRAM latency and bandwidth to boost the performance of programs with irregular memory accesses. Hardware strategies designed to reduce the penalty associated with uncoalesced memory accesses, including in-core miss-handling resources, are unlikely to have a significant performance effect without corresponding improvements in memory bandwidth and latency. Two-level warp scheduling appears to have minimal or negative impact on irregular codes, which perform better with a greedy-then-oldest scheduler than with other simple policies.

7.1. Recommendations for Future Work

Based on these results, future work aimed at improving the suitability of GPGPUs for irregular codes should focus on enhancing cache effectiveness and/or improving memory and especially last-level-cache and memory latencies and bandwidths. Simple warp schedulers proposed in the literature for enhancing the performance of regular codes, e.g., two-level active scheduling, seem ineffective for irregular codes. For these applications, addressing sources of slowdown via warp scheduling will likely require more complicated schemes aimed at increasing cache effectiveness and maximizing memory bandwidth.

LITERATURE CITED

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, pp. 80-113, Mar. 2007.

[2] S. Huang, S. Xiao, and W. Feng, "On the Energy Efficiency of Graphics Processing Units for Scientific Computing," in *Proc. 23rd IEEE Int. Symp. Parallel and Distributed Processing (IPDPS)*, Rome, 2009, pp. 1-8.

[3] (2014). *The Green 500 List* [Online]. Available: http://www.green500.org/news/green500-list-june-2014

[4] P. McGuinness. (2014). *What's Next for Mobile? Heterogeneous Processing Evolves. Embedded Computing Design* [Online]. Available: http://embedded-computing.com/articles/whats-next-for-mobile-heterogeneous-processing-evolves/

[5] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39-55, Mar.-Apr. 2008.

[6] M.J. Harris, "Fast Fluid Dynamics Simulation on the GPU," in *GPU Gems*, R. Fernando, Ed. Boston: Addison-Wesley, 2004, pp. 637-665.

[7] C. Kolb and M. Pharr, "Options Pricing on the GPU," in *GPU Gems 2*, M. Pharr, Ed. Boston: Addison-Wesley, 2005, pp. 719-731.

[8] G.H. Golub and C.F. Van Loan, *Matrix Computations*. Baltimore: Johns Hopkins Press, 1996.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron, "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA," *J. Parallel Distributed Computing*, vol. 68, no. 10, pp. 1370-1380, Oct. 2008.

[10] J. Barnes and P. Hut, "A Hierarchical O(n log n) Force-Calculation Algorithm," *Nature*, vol. 324, pp. 446-449, Dec. 1986.

[11] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Boston: Addison-Wesley, 2005.

[12] A. Braunstein, M. Mézard, and R. Zecchina, "Survey Propagation: An Algorithm for Satisfiability," *Random Structures and Algorithms*, vol. 27, no. 2, pp. 201-226, Sept. 2005.

[13] K. Hildrum and P.S. Yu, "Focused Community Discovery," in *Proc. 5th Int. Conf. Data Mining*, 2005, pp. 27-30.

[14] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston: Addison-Wesley, 1986.

[15] J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Survey*, vol. 18, no. 1, pp. 39-65, Mar. 1986.

[16] L.P. Chew, "Guaranteed-Quality Mesh Generation for Curved Surfaces," in *Proc. 9th ACM Symp. Computational Geometry (SOGC)*, San Diego, CA, 1993, pp. 274-280.

[17] M. Burtscher and K. Pingali, "An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm," in *GPU Computing Gems Emerald Edition*, W.-M.W. Hwu, Ed. Burlington: Morgan Kaufman, 2011, pp. 75-92.

[18] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU Implementation of Inclusion-Based Points-To-Analysis," in *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP)*, New Orleans, LA, 2012, pp. 107-116.

[19] D.G. Merrill, M. Garland, and A.S. Grimshaw, "Scalable GPU Graph Traversal," *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP)*, New Orleans, LA, 2012, pp. 117-128.

[20] A. McLaughlin and D.A. Bader, "Scalable and High Performance Betweenness Centrality on the GPU," in *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, New Orleans, LA, 2014, pp. 572-583.

[21] *LonestarGPU v. 2.0* [Online]. Available: http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

[22] *CUDA C Programming Guide v. 6.5* [Online]. NVIDIA. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[23] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, white paper, NVIDIA, 2009.

[24] *CUDA C Best Practice Guide v. 6.5* [Online]. NVIDIA. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide

[25] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, white paper, NVIDIA, 2012.

[26] V. Narasiman, M. Shebanow, C.J. Lee, R. Miftakhutdinov, O. Mutlu, and Y.N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proc. 44th Ann. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Porto Alegre, Brazil, 2011, pp. 308-317.

[27] T.G. Rogers, M. O'Connor, T.M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proc. 45th Ann. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Vancouver, Canada, 2012, pp. 72-83.

[28] *GPGPU-Sim* [Online]. Available: http://www.gpgpu-sim.org

[29] T.M. Aamodt and W.W.L. Fung, Eds., *GPGPU-Sim 3.x Manual rev. 1.2* [Online]. Available: http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual

[30] *CUDA Samples* [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples/index.html

[31] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Austin, TX, 2009, pp. 44-54.

[32] J.A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G.D. Liu, and W.-M.W. Hwu, *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*, technical report IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[33] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. 2009 IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS)*, Boston, MA, 2009, pp. 163-174.

[34] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications," *in Proc. 2010 IEEE Int. Symp. Workload Characterization (IISWC)*, Atlanta, GA, 2010, pp. 1-10.

[35] E. Blem, M. Sinclair, K. Sankaralingam, "Challenge Benchmarks That Must Be Conquered to Sustain the GPU Revolution," in *Proc. 4th Ann. Workshop Emerging Applications and Many-Core Architecture (EAMA)*, San Jose, CA, 2011.

[36] S. Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proc. 2010 IEEE Int. Symp. Workload Characterization (IISWC)*, Atlanta, GA, 2010, pp. 1-11.

[37] S.-Y. Lee and C.-J. Wu, "Characterizing the Latency Hiding Ability of GPUs," in *Proc. 2014 IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, 2014, pp. 145-146.

[38] J. Hestness, S.W. Keckler, and D.A. Wood, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior," in *Proc. 2014 IEEE Int. Symp. Workload Characterization (IISWC)*, Raleigh, NC, 2014, pp. 150-160.

[39] *Gem5-GPU* [Online]. Available: https://gem5-gpu.cs.wisc.edu/wiki/

[40] A. Kerr, G. Diamos, and S. Yalamanchili, "A Characterization and Analysis of PTX Kernels," in *Proc. 2009 IEEE Int. Symp. Workload Characterization (IISWC)*, Austin, TX, 2009, pp. 3-12.

[41] *GPUOcelot: A Dynamic Compilation Framework for GPU Computing* [Online]. Available: http://gpuocelot.gatech.edu

[42] H. Wu, G. Diamos, S. Li, and S. Yalamanchili, "Characterization and Transformation of Unstructured Control Flow in GPU Applications," in *Proc. 1st Int. Workshop Characterizing Applications for Heterogeneous Exascale Systems (CACHES)*, Tucson, AZ, 2011.

[43] Q. Xu, H. Jeon, and M. Annavaram, "Graph Processing on GPUs: Where are the Bottlenecks?" in *Proc. 2014 IEEE Int. Symp. Workload Characterization (IISWC)*, Raleigh, NC, 2014, pp. 140-149.

[44] M.A. O'Neil and M. Burtscher, "Microarchitectural Performance Characterization of Irregular GPU Kernels," in *Proc. 2014 IEEE Int. Symp. Workload Characterization (IISWC)*, Raleigh, NC, 2014, pp. 130-139.

[45] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *Proc. 2012 IEEE Int. Symp. Workload Characterization (IISWC)*, San Diego, CA, 2012, pp. 141-151.

[46] J. Coplin and M. Burtscher, "Power Characteristics of Irregular GPGPU Programs," in *Proc. 2014 Int. Workshop Green Programming, Computing, and Data Processing (GPCDP)*, Dallas, TX, 2014.

[47] S. Che, B.M. Beckmann, S.K. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *Proc. 2013 IEEE Int. Symp. Workload Characterization (IISWC)*, Portland, OR, 2013, pp. 185-195.

[48] J. Wang and S. Yalamanchili, "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," in *Proc. 2014 IEEE Int. Symp. Workload Characterization (IISWC)*, Raleigh, NC, 2014, pp. 51-60.

[49] N. Brunie, S. Collange, and G. Diamos, "Simultaneous Branch and Warp Interleaving for Sustained GPU Performance," in *Proc. 39th Int. Symp. Computer Architecture (ISCA)*, Portland, OR, 2012, pp. 49-60.

[50] I. Singh, A. Shriraman, W.W.L. Fung, M. O'Connor, T.M. Aamodt, "Cache Coherence for GPU Architectures," in *Proc. 19th IEEE Int. Symp. High-Performance Computer Architecture (HPCA)*, Shenzhen, China, 2013, pp. 578-590.

[51] P. Xiang, Y. Yang, and H. Zhou, "Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation," in *Proc. 20th Int. Symp. High-Performance Computer Architecture (HPCA)*, Orlando, FL, 2014.

[52] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proc. 37th Int. Symp. Computer Architecture (ISCA)*, Saint-Malo, France, 2010, pp. 235-246.

[53] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, New York: McGraw Hill, 2001.

[54] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P. Chew, "Optimistic Parallelism Requires Abstractions," in *Proc. ACM Conf. Programming Languages Design and Implementation (PLDI)*, San Diego, CA, 2007, pp. 211-222.

[55] J. Nešetřil, E. Milková, H. Nešetřilová, "Otakar Borůvka on Minimum Spanning Tree Problem: Translation of Both the 1926 Papers, Comments, and History," *Discrete Mathematics*, vol. 233, pp. 3-36, April 2001.

[56] M.A. O'Neil and M. Burtscher, "Floating-Point Data Compression at 75 Gb/s on a GPU," in *Proc. 4th Workshop General-Purpose Processing on Graphics Processing Units (GPGPU)*, Newport Beach, CA, 2011, no. 7.

[57] M.A. O'Neil, D. Tamir, and M. Burtscher, "A Parallel GPU Version of the Traveling Salesman Problem," in *Proc. 2011 Int. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, NV, 2011.

[58] *9th DIMACS Implementation Challenge – Shortest Paths* [Online]. Available: http://www.dis.uniroma1.it/challenge9/download.shtml

[59] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proc. 4th SIAM Int. Conf. Data Mining*, Lake Buena Vista, FL, 2004, pp. 442-446.

[60] *CUB* [Online]. Available: http://nvlabs.github.io/cub/

[61] *Running LonestarGPU 2.0 on GPGPU-Sim 3.2.1* [Online]. Available: http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu/gpgpusim

[62] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture through Microbenchmarking," in *Proc. 2010 IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS)*, White Plains, NY, 2010, pp. 235-246.

[63] R. Meltzer, C. Zeng, and C. Cecka, "Micro-benchmarking the C2070," at *GPU Technology Conference* [poster], San Jose, CA, 2013.

[64] R. Nasre, M. Burtscher, and K. Pingali, "Morph Algorithms on GPUs," in *Proc. 18th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP)*, Shenzhen, China, 2013, pp. 147-156.

[65] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free Irregular Computations on GPUs," in *Proc. 6th Workshop General Purpose Processing on Graphics Processing Units (GPGPU)*, Houston, TX, 2013, pp. 96-107.

[66] M. Kulkarni, M. Burtscher, R. Inkulu, and K. Pingali, "How Much Parallelism is There in Irregular Applications?" in *Proc. 14th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Raleigh, NC, 2009, pp. 3-14.