Locality and Scheduling in the Massively Multithreaded Era

by

Timothy Glenn Rogers

B. Eng, McGill University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2015

Abstract

Massively parallel processing devices, like Graphics Processing Units (GPUs), have the ability to accelerate highly parallel workloads in an energy-efficient manner. However, executing irregular or less tuned workloads poses performance and energy-efficiency challenges on contemporary GPUs. These inefficiencies come from two primary sources: ineffective management of locality and decreased functional unit utilization. To decrease these effects, GPU programmers are encouraged to restructure their code to fit the underlying hardware architecture which affects the portability of their code and complicates the GPU programming process. This dissertation proposes three novel GPU microarchitecture enhancements for mitigating both the locality and utilization problems on an important class of irregular GPU applications. The first mechanism, Cache-Conscious Warp Scheduling (CCWS), is an adaptive hardware mechanism that makes use of a novel locality detector to capture memory reference locality that is lost by other schedulers due to excessive contention for cache capacity. On cache-sensitive, irregular GPU workloads, CCWS provides a 63% speedup over previous scheduling techniques. This dissertation uses CCWS to demonstrate that improvements to the hardware thread scheduling policy in massively multithreaded systems offer a promising new design space to explore in locality management. The second mechanism, Divergence-Aware Warp Scheduling (DAWS), introduces a divergence-based cache footprint predictor to estimate how much L1 data cache capacity is needed to capture locality in loops. We demonstrate that the predictive, pre-emptive nature of DAWS can provide an additional 26% performance improvement over CCWS. This dissertation also demonstrates that DAWS can effectively shift the burden of locality management from software to hardware by increasing the performance of simpler

and more portable code on the GPU. Finally, this dissertation details a Variable Warp-Size Architecture (VWS) which improves the performance of irregular applications by 35%. VWS improves irregular code by using a smaller warp size while maintaining the performance and energy-efficiency of regular code by ganging the execution of these smaller warps together in the warp scheduler.

Preface

The following is a list of my publications that have been incorporated into this dissertation in chronological order:

[C1] Timothy G. Rogers, Mike O'Connor, Tor M. Aamodt. Cache-Conscious Wavefront Scheduling [137]. In proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO-45), pp. 72-83, December 2012.

[J1] Timothy G. Rogers, Mike O'Connor, Tor M. Aamodt. Cache-Conscious Thread Scheduling for Massively Multithreaded Processors [138]. IEEE Micro Special Issue: Micro's Top Picks from 2012 Computer Architecture Conferences, Vol. 33, No. 3, pp. 78-85, May/June 2013

[C2] Timothy G. Rogers, Mike O'Connor, Tor M. Aamodt. Divergence-Aware Warp Scheduling [139]. In proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO-46), pp. 99-110, December, 2013.

[J2] Timothy G. Rogers, Mike O'Connor, Tor M. Aamodt. Learning Your Limit: Managing Massively Multithreaded Caches Through Scheduling [140]. In Communications of the ACM, vol. 57, no. 12, December 2014.

[C3] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, Stephen W. Keckler. A Variable Warp Size Architecture [141]. In proceedings of the International Symposium on Computer Architecture (ISCA), pp. 489-501, June, 2015.

The preceding publications have been included in this thesis as follows:

Chapter 1. Some motivational elements of this chapter have been previously published in [J1].

Chapter 2. Elements of this chapter that describe the baseline GPU architecture have been taken from [C1], [C2] and [C3].

Chapter 3. A version of this material has been published as [C1], [J1] and [J2]. In [C1], [J1] and [J2], I performed the research, interpreted the data and wrote the manuscript with guidance and input from Mike O'Connor and Professor Tor M. Aamodt.

Chapter 4. A version of this material has been published as [C2]. In [C2], I performed the research, interpreted the data and wrote the manuscript with guidance and input from Mike O'Connor and Professor Tor M. Aamodt.

Chapter 6. A version of this material has been published as [C3]. In [C3], I performed the research, interpreted the data and wrote the manuscript with guidance and input from Dr. Daniel R. Johnson, Mike O'Connor and Dr. Stephen W. Keckler.

Chapter 7. Text from the related work sections of [C1], [C2] and [C3] has been incorporated into this section.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Thread
HCS	Highly Cache-Sensitive
MCS	Moderately Cache-Sensitive
CI	Cache-Insensitive
L0	Level Zero
L1	Level One
L2	Level Two
L1D	Level One Data
PKI	Per Thousand Instructions
MPKI	Misses Per Thousand Instructions
CCWS	Cache-Conscious Warp Scheduling
LLD	Lost intra-warp Locality Detector
MEMC	Memcached
IPC	Instructions Per Cycle
WIA	Warp Issue Arbiter
LRU	Least Recently Used
SWL	Static Warp Limiting
VTA	Victim Tag Array
WID	Warp ID
LLS	Lost-Locality Score

LLDS	Lost-Locality Detected Score
IC	Integrated Circuit
CPU	Central Processing Unit
CMP	Chip-MultiProcessor
SAGCS	Stand Alone GPGPU-Sim Cache Simulator
FR-FCFS	First-Ready First-Come First-Serve
CUDA	Compute Unified Device Architecture
GC	Garbage Collector
GTO	Greedy-Then-Oldest
FG	Fetch Groups
DLP	Data-Level Parallelism
TLP	Thread-Level Parallelism
ILP	Instruction-Level Parallelism
CMOS	Complementary Metal-Oxide-Semiconductor
MIMD	Multiple-Instruction, Multiple-Data
DAWS	Divergence-Aware Warp Scheduling
VWS	Variable Warp Sizing
E-VWS	Elastic Variable Warp Sizing
I-VWS	Inelastic Variable Warp Sizing
PC	Program Counter
I-Buffer	Instruction Buffer
BGTO	Big-Gang-Then-Oldest
LGTO	Little-Gang-Then-Oldest
LRR	Loose Round Robin
I-cache	Instruction-cache
SM	Streaming Multiprocessor
MSHR	Miss Status Holding Register
HPC	High Performance Computing
CoMD	Co-design Molecular Dynamics
FFT	Fast Fourier Transform
CRS	Call Return Stack
TBC	Thread Block Compaction
DWF	Dynamic Warp Formation

NVRT	NVIDIA Raytracing
FCDT	Face Detection
MGST	Merge Sort
NNC	Nearest Neighbour Computation
DWS	Dynamic Warp Subdivision
CF	Control Flow
MSMD	Multiple-SIMD, Multiple Data
CTA	Cooperative Thread Array
WB	Warp Barrier
CSR	Compressed Sparse Row
BFS	Breadth First Search
SPMV	Sparse Matrix Vector Multiply
PDOM	Post Dominator
ILRP	Intra-Loop Repetition Detector
API	Application Programming Interface
TDP	Thermal Design Point
FLOPS	FLoating point Operations Per Second
FGMT	Fine-Grained MultiThreading
SMT	Simultaneous MultiThreading
APU	Accelerated Processing Unit
AMD	Advanced Micro Devices
GDDR	Graphic Dual Data Rate
DRAM	Dynamic Random Access Memory

Acknowledgments

First and foremost, I would like to thank my wife for sticking with me while I quit my job and walked out on several hundred thousand dollars for the off-chance that I would succeed as a PhD student. I could not have done any of this without Jenelle. Second, I would like to thank my children for their energy, excitement and always being there to remind me what is most important. My final personal thanks goes out to my parents and grandparents who raised me to always aspire to better myself, never give up and to fight for every inch of my goals.

Professionally, I would like to thank my advisor Professor Tor Aamodt for always pushing me pursue high quality research, giving me the freedom to explore my own ideas and helping me develop them. Tor's guidance over the last five years has been invaluable and has shaped everything I know about doing good research. A special thanks to Mike O'Connor for being a great colleague who is also a lot of fun to work with. Mike's depth of industrial knowledge has helped influence all the research I have done. From UBC, I would also like to thank Tor's previous PhD graduates Wilson Fung and Ali Bakhoda for going through this process before me and helping me with every aspect of finally getting my PhD submitted. Thanks to all of the UBC computer architecture students I have worked with: Tayler Hetherington, Dongdong Li, Ayub Gubran, Dave Evans, Jonny Kuan, Hadi Hooybar, Inderpreet Singh, Jimmy Kwa, Andrew Turner, Arun Ramamurthy, Rimon Tadros, Shadi Assadi, Ahmed ElTantawy and Andrew Boktor. From NVIDIA Research, I would like to thank Daniel Johnson and Steve Keckler for their dedicated collaboration. From AMD Reaserch, I would like to thank Brad Beckmann and Gabe Loh for their mentorship and insights on architecture research. I would also list to thank the members of my PhD qualifying and final PhD defense exams: Pro-

fessor Matei Ripeanu, Professor William Dunford, Professor Philippe Kruchten, Professor Steve Wilson, Professor Sathish Gopalakrishnan, Professor Mieszko Lis, Professor Michiel van de Panne, Professor Alan Wagner, Professor Albert Dexter and finally my external examiner, Professor Onur Mutlu for their direction and suggested improvements to my thesis.

Finally I would like to acknowledge the funding sources that made my research possible: The Alexander Graham Bell Canada Graduate Scholarship (CGS-D) provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) as well as the NVIDIA Graduate Fellowship.

To my wife Jenelle and our children, Erica and Nathan.

Chapter 1

Introduction

The dawn of Turing complete [162], digital, electronic computing in the mid twentieth century was brought about by the invention of programmable computer hardware [58, 65, 169, 170]. In the 80 years since their creation, digital computers have undergone a series of revolutions in their architecture, implementation and programming interface. These revolutions have improved the performance, energy-efficiency, cost-effectiveness and programmability of the modern computer to a point where processors are universally integrated into our society.

One of the aforementioned revolutions began in 1971 with the introduction of the Intel 4004 [4]. The 4004 was the first commercially available single-chip central processing unit (CPU) built using integrated-circuit (IC) technology. Since the introduction of the microprocessor, IC fabrication technologies have advanced at a rate which enabled the number of transistors on a cost-effective IC to double roughly once every two years. This trend, first discovered by Gordon Moore, is commonly referred to as Moore's Law [121]. In the early days of Moore's Law, the exponential increase in transistor count, primarily due to transistor gate length shrinking, was coupled with Dennard scaling [45] and enabled the clock frequency (and correspondingly the performance) of single threaded, single chip microprocessors to scale exponentially at roughly the same rate as transistor density. However, beginning around 2005, the power density guarantees provided by Dennard scaling broke down, halting the cost-effective increase in processor frequency scaling which has lead to what is commonly referred to as the Power Wall [16, 122].

In combination with the decline of frequency scaling, architectural techniques to improve increase single threaded instruction level parallelism (ILP) have reached a point of diminishing returns [43]. Today, Moore's Law continues to march forward, with chip transistor counts still exponentially increasing. However, our ability to utilize these transistors to improve single threaded performance has been severely limited. The end of single thread performance scaling has brought about the most recent revolution in computer architecture.

A relatively simple way to improve performance and energy-efficiency in the face of diminishing single thread performance is to produce chips that contain multiple processor cores, known as chip multiprocessors (CMP). CMPs improve performance by relying on software to explicitly expose parallelism. This can be done in the operating system at the process level (known as multiprocessing) where multiple programs are run on each of the chip's processors. Individual programs can also be written in a parallel format (known as multithreading) using a number of parallel application program interfaces (APIs) [1, 2, 68, 69, 87]. However, recent work exploring dark silicon [49] (which is the portion of the chip that must be powered off at any given time to stay within a power budget) suggests that multi-core scaling will not continue for future technology generations. In addition to CMP development, the break-down in single threaded energy-efficiency and performance has lead computer architects to pursue more drastic alternatives to improve computing capabilities.

One such alternative is the massively parallel acceleration provided by graphics processing units (GPUs). Originally developed for accelerating graphics applications, APIs like CUDA [6] and OpenCL [87] allow programmers to write general purpose software for GPUs. Running general purpose (or compute) workloads on a GPU is commonly referred to as general purpose GPU (GPGPU) computing.

The GPU's transition from an application specific accelerator to a general purpose device has progressed gradually over the last several product generations. However, a number of important challenges remain when accelerating general purpose software on contemporary GPUs. This dissertation proposes hardware solutions to help solve these challenges.

1.1 Massively Parallel Computing Trends

The underlying reasons for the surge in popularity of GPGPU computing is a result of two factors, one economical and one scientific. Like all engineering fields, good computer engineering research must account for the cost-effectiveness of any solution. Since GPUs are sold in substantial volume for graphics, the cost of GPU hardware as a computing platform is kept relatively low. However, for this dissertation, the scientific reason for the success of GPGPUs is more interesting: energy-efficiency. By running many thousands of threads in parallel at a relatively low clock frequency (as opposed to the tens of threads runnable on contemporary CMPs), GPUs are able to perform a large amount of computation for relatively little energy. In 2015, the peak computational efficiency (peak performance / thermal design point (TDP)) of state-of-the-art GPUs from AMD and NVIDIA was > 18 GFLOPS/W (Floating point Operations Per Second Per Watt), whereas the most efficient CMPs from Intel performed < 6 GFLOPS/W, even through the CMPs were fabricated at a more energy-efficient technology node. In addition to the theoretical peak energy-efficiency of GPUs, it has been demonstrated that GPUs are an effective computing platform for a variety of practical workloads [64, 99]. In fact, 9 of the top 10 supercomputers on the 2014 Green 500 List [50] (which ranks global supercomputers based on their measured energy-efficiency on the Linpack Benchmark report [46]) contain a GPU.

The fundamental reason for the GPU's computational efficiency is parallelism. Traditional 3D graphics workloads, sometimes called shaders, perform thousands of independent, parallel computations on the elements that make up a rendered scene. Each of these operations helps to determine the output color of each pixel in the finally rendered scene. As a result, the 3D rendering process is very data parallel and the hardware created to accelerate it takes advantage of this parallelism to increase throughput.

Since the software running on GPUs has exposed a large amount of parallelism, GPUs traditionally rely on aggressive fine grained multithreading (FGMT) to tolerate long latency operations (such as accessing global memory). FGMT operates by issuing instructions from different threads when a single thread is stalled from executing. Given enough threads and enough memory bandwidth, the GPU's data-

path can be kept busy, increasing overall throughput at the expense of single-thread latency. In contrast, traditional CPU (and CMP) designs devote a significant portion of the chip's area and power budget to mechanisms like out of order execution and branch prediction that attempt to cover stalled execution by extracting more ILP from a single thread.

Since the exposed parallelism in traditional GPU software is data parallel, GPUs can extract energy-efficiency gains from executing instructions in a wide single instruction multiple data (SIMD) format. GPUs group the threads defined by the software into larger entities, known as warps (or wavefronts) in hardware. The threads in a warp share a program counter (PC), allowing them to execute in lock-step on the SIMD hardware and amortize the overhead of executing an instruction over the width of the warp.

Although massive FGMT and SIMD execution enable the GPUs to be highly energy-efficient, they present a number of interesting challenges surrounding programmability, efficient control flow and locality management.

1.2 GPU Programmability Challenges

Since the early days of the ENIAC [58] and the Harvard Mark I [65], the programmability of computer hardware has played a vital role in how useful that hardware ends up being. The same is true in today's CPU versus GPU debate. GPU programming models, like CUDA and OpenCL allow software developers to write code for the execution of a single scalar thread of execution. This programming model allows the developers to write multiple-instruction, multiple-data (MIMD) programs that operate on SIMD hardware. A regular program is one where each scalar thread executes a similar path through the code (control flow) and adjacent threads in the program operate on data with a similar memory address. Regular programs are well suited for GPU acceleration since they can take advantage of amortizations done in the hardware. In contrast, the GPGPU programming model also supports irregular applications, where control flow and memory accesses among thread are less uniform. These irregular applications cause performance and energy-efficiency challenges on contemporary GPUs. As a result, GPU programmers are encouraged to find ways to restructure the algorithm and data of

their applications to remove irregularity and improve performance. This process is not guaranteed to improve performance and can be both difficult and time consuming, even for experience programmers. Moreover, porting an existing piece of parallel code to an accelerator and having it run efficiently is a challenge [142]. This dissertation seeks to decrease the time and complexity involved in writing efficient GPU software by tackling some of the key challenges that stem from irregularity.

1.3 GPU Memory Locality Challenges

Since massively parallel architectures execute thousands of threads concurrently, a high bandwidth main memory system is necessary to handle the traffic generated. Additionally, GPUs employ several mechanism to exploit the memory reference locality. They use a technique known as memory coalescing [6] (described in more detail in Chapter 2) to aggregate redundant memory accesses among the threads in a warp. If there is spatial locality across the threads in a warp when a memory instruction is issued, the GPU will group the accesses for all the threads in the warp together, generating fewer main memory system requests. Regular applications often take advantage of this feature. Like CPUs, GPUs also employ on-chip data caches in an attempt to capture both spatial locality (where multiple memory references access data in nearby addresses) and temporal locality (where multiple memory references access the same address in a short period of time).

Since the appearance of the processor memory wall [173] (where the speed of the processor vastly outpaced the speed of the main memory system), exploiting memory reference locality in the caching system has been critical to improving CPU performance. As a result, the past 30 years have seen a significant amount of research and development devoted to using on-chip CPU caches more effectively. At a hardware level, cache management has typically been optimized by improvements to the hierarchy [25], replacement/insertion policy [130], coherence protocol [112] or some combination of these. Previous work on hardware caching assumes that the access stream seen by the memory system is fixed. However, massively multithreaded systems introduce another dimension to the problem. Each cycle, a fine-grained, issue level thread scheduler must choose which of a core's active threads issues next. This decision has a significant impact on the access

stream seen by the cache. In a massively multithreaded system, there may be more than one thousand threads ready to be scheduled on each cycle. This dissertation exploits this observation and uses the thread scheduler to explicitly manage the access stream seen by the memory system to maximize throughput. Although this dissertation studies the effect of thread scheduling and locality management on GPUs, it is a concern to any architecture where many hardware threads can share a cache. Some examples include Intel's Xenon Phi [70], Orcale's SPARC T4 [149], IBM's Blue Gene/Q [61].

In addition to studying the effect irregular applications have on the caching system, this dissertation also explores the effect reduced memory coalescing can have on performance and energy-efficiency when techniques aimed at mitigating control flow challenges are used.

1.4 GPU Control Flow Challenges

The GPU's scalar programming model, combined with its SIMD pipeline necessitates a unique execution model, commonly referred to as single-instruction, multiple-thread (SIMT). Since individual threads within a warp may execute different control flow paths through the code, the GPU's control hardware supports a mechanism to disable inactive SIMD lanes when threads within a warp do not execute the same control flow path (this mechanism is discussed in more detail in Chapter 2). As a result, software with irregular control flow executes inefficiently on a GPU, since many of the lanes in the wide SIMD datapath will be disabled on each executed instruction. Writing effective GPU software for irregular or data-dependent applications is challenging, as the programmer should take steps to ensure that threads in the same warp execute similar control-flow paths.

1.5 Thesis Statement

This dissertation explores the massively multithreaded hardware design space surrounding memory locality and application irregularity. The dissertation proposes three novel hardware thread scheduling mechanisms that aim to improve massively parallel performance and energy-efficiency on a varied set of traditional regular and more forward-looking irregular massively parallel applications. The hardware

modifications proposed in this dissertation are aimed at reducing the amount of hardware specific knowledge required to write efficient massively parallel software.

The first mechanism proposed in this dissertation is cache-conscious warp scheduling (CCWS). CCWS is a novel warp scheduling microarchitecture that uses feedback from the caching system to make issue-level scheduling decisions in the massively parallel core. CCWS proposes a novel lost locality detector which it uses to react to the over-subscription of the processor's data cache caused by excessive multithreading. Upon the detected over-subscription, CCWS reduces the number of warps sharing the cache by preventing some of them from issuing memory instructions, effectively reducing the number of threads actively scheduled.

Divergence-aware warp scheduling (DAWS) is the second mechanism proposed by this dissertation. It expands on the core insights developed in CCWS and proposes a pre-emptive mechanism to curb the issue-level scheduling of warps based on an online characterization of the program's loops. Using this information, in tandem with runtime information about the amount of control flow divergence being experienced by each warp, DAWS makes a pre-emptive prediction about each warp's cache footprint and curbs the scheduling of warps such that the capacity of the system's data cache is not exceeded.

This dissertation then studies the effect issue-level thread scheduling in hardware can have on the programmability of GPUs. A case study is performed on two implementation of sparse matrix-vector multiply: one GPU-optimized, one not. The dissertation then studies the performance and memory system impacts of both implementations using CCWS and DAWS to demonstrate that issue-level thread scheduling in the hardware can come within 4% of the performance of tuned GPU code.

Finally, this dissertation explores a variable warp-size architecture (VWS). VWS seeks to solve the control flow divergence problem in irregular or less optimized GPU programs by proposing a machine that is capable of running with a more narrow warp size. VWS demonstrates that operating at a small warp size is detrimental to the horizontal locality (described in Chapter 2) present in a large set of existing GPU applications. VWS proposes a ganged scheduling mechanism that enforces wide, lock-step execution when appropriate while still allowing narrow

execution in the presence of divergence.

1.6 Contributions

This thesis makes the following contributions:

1. It identifies intra and inter warp locality and quantifies the trade-off between maximizing intra-warp locality and concurrent multithreading.

2. It proposes a novel cache-conscious warp scheduling (CCWS) mechanism which can be implemented with no changes to the cache replacement policy. CCWS uses a novel lost intra-warp locality detector (LLD) to update an adaptive locality scoring system and improves the performance of highly cache-sensitive workloads by 63% over existing warp schedulers and increases the total chip area by only 0.17%.

3. It demonstrates that CCWS reduces L1D cache misses more than the Belady-optimal replacement scheme.

4. It demonstrates that CCWS can be tuned to trade-off power and performance. A power-tuned configuration of CCWS reduces energy-expensive L1D cache misses an additional 18% above the performance tuned configuration while still achieving a 49% increase in performance on highly cache-sensitive workloads.

5. It quantifies the relationship between data locality, branch divergence and memory divergence in GPUs on a set of economically important, highly cache-sensitive workloads.

6. It demonstrates that code regions can be classified by both data locality and memory divergence.

7. It demonstrates, with an example, that DAWS enables unoptimized GPU code written in a scalar fashion to attain 96% of the performance of optimized code that has been re-written for GPU acceleration.

8. It proposes a novel divergence-aware warp scheduling (DAWS) mechanism which classifies static load instructions based on their memory usage. It uses

this information, in combination with the control flow mask, to appropriately limit the number of scalar threads executing code regions with data locality. DAWS achieves a harmonic mean 26% speedup over CCWS [137] and 5% improvement over a profile-based warp limiting solution [137] with negligible area increase over CCWS.

9. It characterizes the performance, control flow/memory divergence, and fetch/decode effects of different warp sizes on a large number of graphics and compute GPU workloads.

10. It demonstrates that reducing the warp size of modern GPUs does not provide a universal performance advantage due to interference in the memory system and an increase in detrimental scheduling effects.

11. It explores the design space uncovered by enabling a dynamic, variable warp size. It quantifies the effects of scheduling and gang combination techniques when the machine has the flexibility to issue from multiple control flow paths concurrently.

12. It proposes a novel warp ganging microarchitecture that makes use of a hierarchical warp scheduler, enabling divergent applications to execute multiple control flow paths while forcing convergent ones to operate in lock-step.

1.7 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 details the background GPU architecture used in this dissertation.
- Chapter 3 introduces cache-conscious warp scheduling (CCWS), a novel microarchitecture that detects locality in cache-sensitive GPU applications and reactively curbs the warp scheduling algorithm to improve performance and energy consumption.
- Chapter 4 details divergence-aware warp scheduling (DAWS), a GPU microarchitectural innovation that pre-emptively curbs the warp scheduling al-

gorithm to further increase cache-sensitive application performance and improve the programmability of GPUs.

- Chapter 5 performs a case study on a GPGPU application written in two different ways: one GPU-optimized and the other more general. It examines the performance effect of running CCWS and DAWS on the less optimized, more general code and demonstrates that DAWS is able bring the performance of the unoptimized code withing 4% of the optimized code.

- Chapter 6 presents a variable warp-size architecture, a GPU microarchitecture and thread scheduling mechanism that improves the performance of irregular applications, while maintaining contemporary GPU performance and energy-efficiency on regular codes.

- Chapter 7 discusses related work.

- Chapter 8 concludes the dissertation and discusses directions for potential future work.

Chapter 2

Background

This chapter first describes the GPU programming model, then details the architecture of a contemporary GPU, which is used as a baseline throughout this dissertation. The chapter then details the memory system of a GPU, how locality manifests itself and defines what is meant by memory irregularity on a GPU. The chapter then details a contemporary mechanism for dealing with control flow on GPUs and defines what is meant by control flow irregularity on GPUs.

2.1 GPU Programming Model

In CUDA [6] and OpenCL [87], the programmer writes the code for a GPU program (or kernel) from the perspective of a scalar thread. When this kernel is launched for execution on the GPU, it is divided into cooperative thread arrays (CTAs) or workgroups. Each CTA is assigned to a GPU streaming multiprocessor (SM). SMs are known as compute units in AMD terminology. Threads within a CTA are able to share memory through an on-chip scratchpad (local data store in AMD) and can perform fast synchronization amongst themselves using barriers. The number of threads in the CTA is determined by the programmer and is bounded by resource constraints such as the number of registers and amount of scratchpad memory used. Applications begin execution on a host CPU, which spawns coarse grained kernels to the GPU for execution. Traditionally, the GPU and CPU operate on different memory spaces requiring the transfer of data from the host memory

to the device memory. Recent chips such as AMD's accelerated processing units (APUs), have the ability to share a memory space between the CPU and GPU [14].

2.2 Contemporary GPU Architecture

Figure 2.1 depicts our model of a modern GPU. The GPU consists of several SMs connected to the main memory system via an interconnection network. Each SM in the GPU is responsible for the execution of multiple CTAs, the aggregate of which can consist of more than one thousand scalar threads. To mitigate the instruction fetch, decode and scheduling overhead of so many threads on a single processor, GPUs group scalar threads into scheduling entities known as warps (or wavefronts, in AMD terminology). Threads in a warp execute the same instructions in lockstep. The warp size of contemporary GPUs is fixed at 32 threads for NVIDIA GPUs and 64 threads for AMD.

Our pipeline decouples the fetch/decode stages from the issue logic and execution stage by storing decoded instructions in per-warp instruction buffers. Each warp can store multiple decoded instructions in its instruction buffer. Each instruction entry in the buffer also contains a valid bit, which is set when an instruction is filled into the buffer, and a ready bit, which is set when the in-order scoreboard indicates the instruction is able to execute.

The front-end of each SM includes an L1 instruction cache which is probed once per cycle by the warp fetch scheduler. The warp fetch scheduler determines which empty entry in the instruction buffer is to be filled. On the execution side, a warp issue scheduler selects one decoded, ready instruction to issue each cycle.

2.3 Memory on GPUs

Contemporary GPUs have four memory spaces visible to the programmer: texture memory (image object in AMD machines), shared scratchpad memory (local data store), constant memory, local memory (private memory) and global memory. Texture memory is a read-only memory space (used primary by graphics applications), is addressable by special texture fetch instructions and is cached on-chip with a specialized hardware texture cache [60]. The shared scratchpad memory addresses a high bandwidth, on-chip data store and is used by GPGPU applications

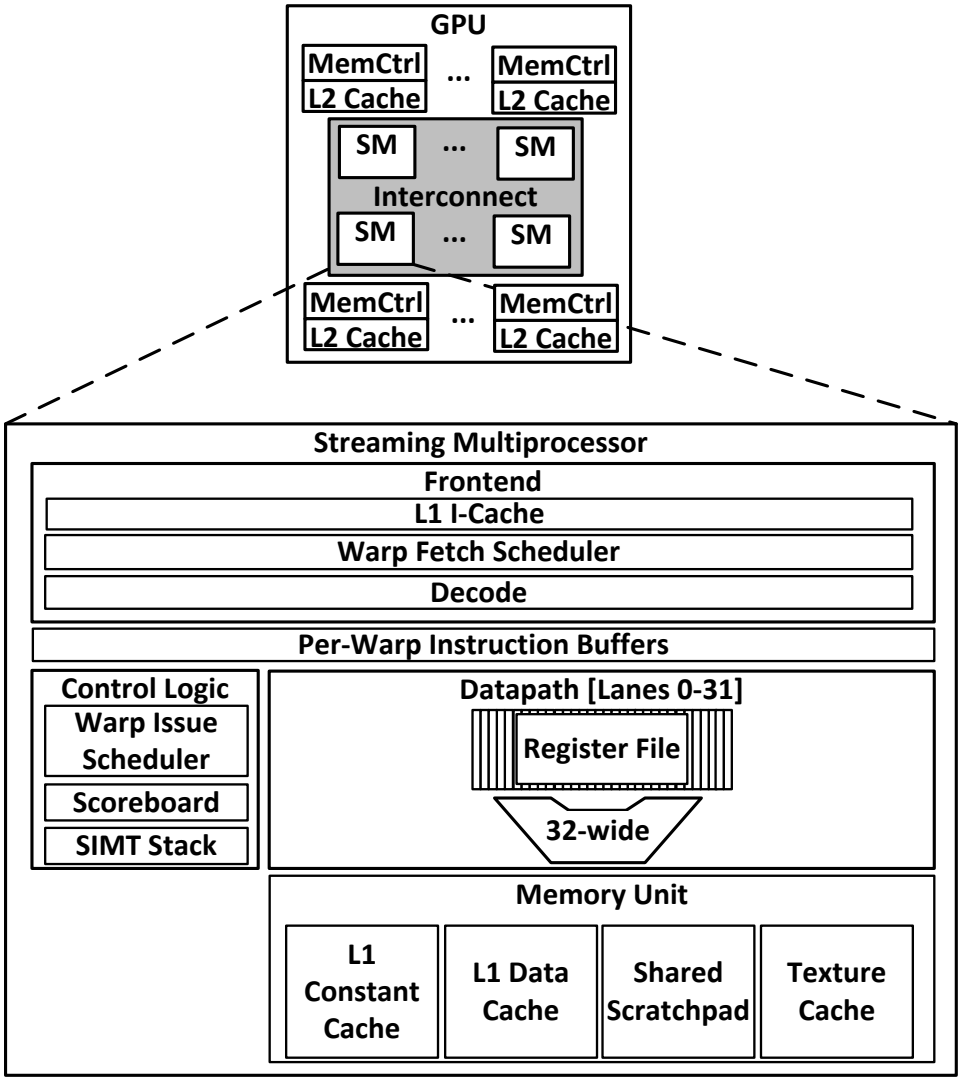


Figure 2.1: Contemporary GPU architecture. MemCtl=memory controller. SIMT Stack=single-instruction, multiple-data stack. I-cache=instruction cache.

for sharing data between the threads in a CTA. This memory space is explicitly managed by the GPU programmer. The constant memory space is used for read only data such as program constants and kernel input parameters and has a small on-chip cache. Local memory is private to each scalar thread and is cached in the L1 data cache. Finally, global memory is the most general form of GPU memory, used for all other general read/write data. Global memory is cached in the on-chip L1 data cache. Much of this dissertation focuses on capturing locality in the global memory space. Data from all the memory spaces is cached in the shared L2 cache, which is strided across the GPUs memory controllers.

Our memory system is similar to that used in modern GPUs [5, 7]. Each SM has a software-managed scratchpad, an L1 data cache, L1 constant cache and a texture unit. Access to the memory unit is shared by all lanes. To reduce the number of memory accesses generated from each warp, GPUs coalesce memory requests into cache line sized chunks when there is spatial locality across the warp. A single instruction that touches only one cache line will generate one transaction that services all 32 lanes. From a memory access perspective, regular programs are those that generate few memory accesses from each SIMD warp instruction. Memory irregularity occurs when adjacent threads (in space) access different cache lines, generating several memory accesses from one SIMD warp instruction. This memory irregularity across the same warp instruction is called memory divergence. Memory regularity occurs when threads with adjacent thread identifiers in the program access adjacent (or redundant) memory addresses. We call this type of locality *horizontal locality* since it occurs horizontally in time across threads in the same instruction. In contrast, *vertical locality* occurs when an individual warps re-reference cache lines on different dynamic instructions. Memory regular programs have horizontal locality, while vertical locality can occur in both regular and irregular applications. In massively multithreaded systems, exploiting locality in memory references is critical to achieving high performance and energy-efficiency. Every global memory instruction from every thread in a GPU program requires a response from the global memory system. The arrangement of these accesses in both space and time is critical to reducing off-chip memory traffic. GPUs have several mechanism to exploit memory reference locality. This dissertation focuses on exploiting two of them: coalescing and caching. Our main memory system in-

cludes a variable latency, fixed bandwidth graphics dual data rate (GDDR) dynamic random access memory (DRAM) model.

2.4 Control flow on GPUs

The programming model for GPUs is from the perspective of a scalar thread. Each scalar thread can have conditional statements that affect the control flow of that particular thread. However, the underlying datapath and execution model of a GPU groups the scalar threads into warps, where only one PC (or assembly instruction) is executed for all the threads in the warp. As a result, some of the lanes in the GPU's SIMD pipeline must be disabled when some of the threads in the same warp execute do not execute a particular instruction. A structure to bookkeep the threads active in a warp in each basic block, as well a mechanism to re-active lanes in a warp when threads return from conditional code is required. Contemporary GPUs uses what is know as a SIMT (or call return) stack [55] to keep track of which threads are active in each basic block. The re-activation of lanes in a warp occurs when threads in the warp that were executing different control flow paths converge at a common basic block. These reconvergence points typically occur at basic block post-dominations, and GPU compilers insert special instructions to inform the hardware when they occur. When threads within the same warp take different control flow paths, resulting in the execution of SIMD instructions with inactive lanes, we call this control flow divergence. Applications with little control flow divergence (i.e. most of the SIMD instructions execute with most of the lanes active) are considered regular from a control flow perspective. In contrast, control flow irregular applications are those that loose significant function unit utilization (also called SIMD efficiency) because threads within the same warp do not execute the same control flow path.

Chapter 3

Cache-Conscious Warp Scheduling

This chapter studies the effects of hardware thread scheduling on cache management in GPUs. We propose Cache-Conscious Warp Scheduling (CCWS), an adaptive hardware mechanism that makes use of a novel intra-warp locality detector to capture locality that is lost by other schedulers due to excessive contention for cache capacity. In contrast to improvements in the replacement policy that can better tolerate difficult access patterns, CCWS shapes the access pattern to avoid thrashing the shared L1. We show that CCWS can outperform any replacement scheme by evaluating against the Belady-optimal policy. Our evaluation demonstrates that cache efficiency and preservation of intra-warp locality become more important as GPU computing expands beyond use in high performance computing. At an estimated cost of 0.17% total chip area, CCWS reduces the number of threads actively issued on a core when appropriate. This leads to an average 25% fewer L1 data cache misses which results in a harmonic mean 24% performance improvement over previously proposed scheduling policies across a diverse selection of cache-sensitive workloads.

Each GPU cycle, a hardware warp scheduler must decide which of the multiple active warps execute next. Our work focuses on this decision. The goal of a warp scheduler is to ensure the execution pipeline is kept active in the presence of long latency operations. The inclusion of caches on GPUs [5] can reduce the latency of

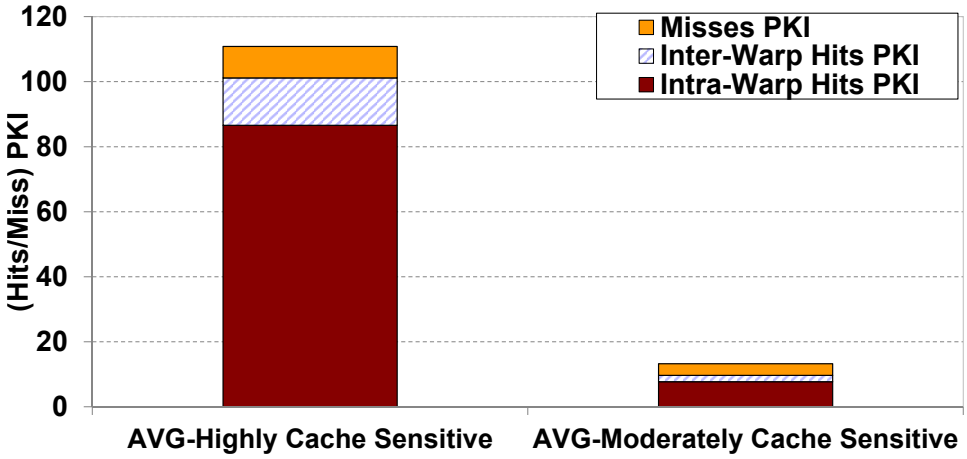


Figure 3.1: Average hits and misses *per thousand instructions* (PKI) using an unbounded L1 data cache (with 128B lines) on cache-sensitive benchmarks.

memory operations and act as a bandwidth filter, provided there is some locality in the access stream. Figure 3.1 presents the average number of hits and misses *per thousand instructions* (PKI) of *highly cache-sensitive* (HCS) and *moderately cache-sensitive* (MCS) benchmark access streams using an unbounded *level one data* (L1D) cache. The figure separates hits into two classes. We classify locality that occurs when data is initially referenced and re-referenced from the same warp as *intra-warp locality*. Locality resulting from data that is initially referenced by one warp and re-referenced by another is classified as *inter-warp locality*. Intra-warp locality is a combination of intra-thread locality [96] (where data is private to a single scalar thread) and inter-thread locality where data is shared among scalar threads in the same warp. Figure 3.1 illustrates that the majority of data reuse observed in our HCS benchmarks comes from intra-warp locality.

To exploit this type of locality in HCS benchmarks, we introduce Cache-Conscious Warp Scheduling (CCWS). CCWS uses a novel *lost intra-warp locality detector* (LLD) that alerts the scheduler if its decisions are destroying intra-warp locality. Based on this feedback, the scheduler assigns intra-warp locality scores to each warp and ensures that those warps losing intra-warp locality are given more exclusive access to the L1D cache.

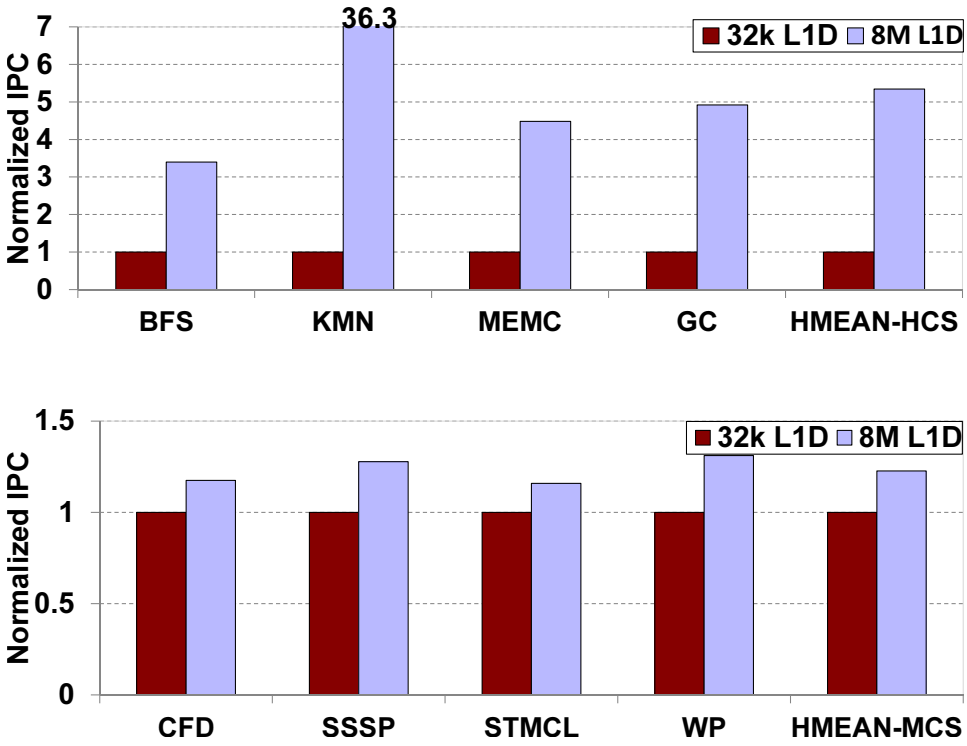


Figure 3.2: Performance using a loose round-robin scheduler at various L1D cache sizes for highly cache-sensitive (top) and moderately cache-sensitive benchmarks (bottom), normalized to a cache size of 32k. All caches are 8-way set-associative with 128B cache lines.

Simple warp scheduling policies such as round-robin are oblivious to their effect on intra-warp locality, potentially touching data from enough threads to cause thrashing in the L1D. A two level scheduler such as that proposed by Narasiman et al. [123] exploits inter-warp locality while ensuring warps reach long latency operations at different times by scheduling groups of warps together. However, Figure 3.1 demonstrates that the HCS benchmarks we studied will benefit more from exploiting intra-warp locality than inter-warp locality. Existing schedulers do not take into account the effect issuing more warps has on the intra-warp locality of those warps that were previously scheduled. In the face of L1D thrashing, the round-robin nature of their techniques will cause the destruction of older warp's intra-warp locality.

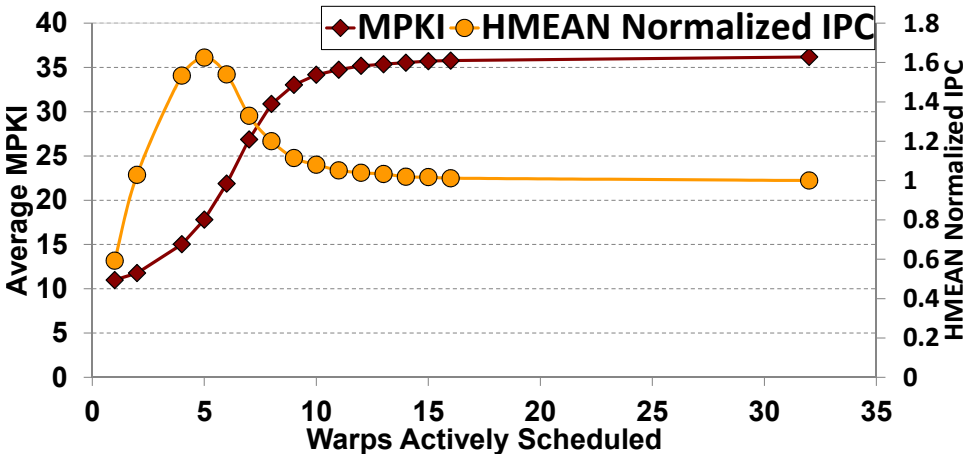


Figure 3.3: Average *misses per thousand instructions* (MPKI) and *harmonic mean* (HMEAN) performance improvement of HCS benchmarks with different levels of multithreading. *Instructions per cycle* (IPC) is normalized to 32 warps.

Figure 3.2 illustrates the cache size sensitivity of our benchmarks (described in Section 3.3) when using a round-robin scheduler and the baseline system described in Section 3.3. Although all of these benchmarks are somewhat cache-sensitive, the HCS benchmarks plotted on the left in Figure 3.2 see 3× or more performance improvement with a much larger L1 data cache.

For GPU-like architectures to effectively address a wider range of workloads, it is critical that their performance on irregular workloads is improved. Recent work on the highly cache-sensitive Memcached (MEMC) [62] and BFS [119] has shown promising results running these commercially relevant irregular parallel workloads on GPUs. However, since current GPUs face many performance challenges running irregular applications, there are relatively few of them written. In this work we evaluate a set of irregular GPU applications and demonstrate their performance can be highly sensitive to the GPU's warp scheduling policy.

Figure 3.3 highlights the impact warp scheduling can have on preserving intra-warp locality. It shows the effect of statically limiting the number of warps actively scheduled on a core. Peak throughput occurs at a multithreading value less than maximum concurrency, but greater than the peak cache performance point (which

limits concurrency to a single warp). Although it may seem counterintuitive to limit the amount of multithreading in a GPU, our data demonstrates a trade-off between hiding long latency operations and creating more of them by destroying intra-warp locality.

Our work draws inspiration from cache replacement and insertion policies in that it attempts to predict when cache lines will be reused. However, cache way-management policies decisions are made among a small set of blocks. A thread scheduler effectively chooses which blocks get inserted into the cache from a pool of potential memory accesses that can be much larger than the caches associativity. Similar to how cache replacement policies effectively *predict* each line's re-reference interval [73], our proposed scheduler effectively *changes* the re-reference interval to reduce the number of interfering references between repeated accesses to high locality data. Unlike scheduling approaches for managing contention implemented in the operating system [181], our technique exploits fine-grained information available to the low-level hardware scheduler.

3.1 Effect of Shaping the Access Pattern

To illustrate the effect an issue-level thread scheduler can have on the cache system, consider Figures 3.4 and 3.5. They present the access pattern created by two different thread schedulers in terms of cache lines touched. GPUs group threads into warps (or warps) for execution, issuing the same static instruction from multiple threads in a single dynamic instruction. This means one memory instruction can generate up to M data cache accesses where M is the warp width. In this example, we assume each instruction generates four memory requests and we are using a fully associative, four entry cache with a *least recently used* (LRU) replacement policy. The access stream in Figure 3.4 will always miss in the cache. However, the stream in Figure 3.5 will hit 12 times, capturing every redundant accesses. The access patterns in these two examples are created by the GPU's warp scheduler. In Figure 3.4 the scheduler chose to issue instructions from warps in a round-robin fashion without considering the effect of the resulting access stream on cache performance. The scheduler in Figure 3.5 prioritized accesses from the same warp together (indicated by the red boxes), creating a stream of cache accesses where

Figure 3.4: Example access pattern (represented as cache lines touched) resulting from a throughput oriented round-robin scheduler. The letters (A,B,C,...) represent cache lines accessed. W_i indicates which warp generated this set of accesses. For example, the first four accesses to cache lines A,B,C and D are generated by warp 0.

Figure 3.5: Example Access pattern resulting from a scheduler aware the effect scheduling has on the caching system. The red boxes indicate that the issue-level scheduler has re-arranged the order warp's accesses are issued in.

locality is captured by our example cache.

3.2 Warp Scheduling to Preserve Locality

This section describes our scheduling techniques. First, Section 3.2.1 analyzes warp scheduling for locality preservation in an example workload with intra-warp locality. Next, Section 3.2.2 introduces *static warp limiting* (SWL) which gives high-level language programmers an interface to tune the level of multithreading. Finally, Section 3.2.3 describes *Cache-Conscious Warp Scheduling* (CCWS), an adaptive hardware scheduler that uses fine-grained memory system feedback to capture intra-warp locality.

3.2.1 A Code Example

Consider the inner loop of a graph processing workload presented in Example 1. The problem has been partitioned by having each scalar thread operate on all the

edges of a single node. The adjoining edges of each node are stored sequentially in memory. This type of storage is common in many graph data structures including the highly space efficient compressed sparse rows [22] representation. This workload contains *intra-warp locality* resulting from *intra-thread locality* (data's initial reference and subsequent re-references come from the same scalar thread).

The inner loop of each scalar thread strides through attributes of its assigned node's edges sequentially. This sequential access stream has significant spatial locality that can be captured by a GPU's large cache line size (e.g. 128B). If the GPU was limited to just a single thread per SM, the memory loads inside the loop would hit in the L1D cache often. In realistic workloads, more than one thousand threads executing this loop will share the same L1D cache.

Example 1 Example graph algorithm kernel run by each scalar thread.

```
int node_degree = nodes[thread_id].degree;
int thread_first_edge = nodes[thread_id].starting_edge;
for ( int i = 0; i < node_degree; i++ ) {
    edge_attribtes = edges[thread_first_edge + i];
    int neighbour_node_id = edge_attributes.node;
    int edge_weight = edge_attributes.weight;
    ...
}
```

We find that if the working set of all the threads is small enough to be captured by the L1D, optimizing both cache efficiency and overall throughput is largely independent of the scheduler choice. In the other extreme, if only one warp's working set fits in the cache, optimizing misses would have each warp run to completion before starting another. Optimizing performance when the L1D is not large enough to capture all of the locality requires the warp scheduler to intelligently trade-off preserving intra-warp locality with concurrent multithreading.

If the scheduler had oracle information about the nature of the workload, it could limit the number of warps actively scheduled to maximize performance. This observation motivates the introduction of *static warp limiting* (SWL) which allows a high-level programmer to specify a limit on the number of warps actively scheduled per SM at kernel launch.

3.2.2 Static Warp Limiting (SWL)

Figure 3.3 shows the effect limiting the number of warps actively scheduled on a SM has on cache performance and system throughput. Current programming API's such as CUDA and OpenCL allow the programmer to specify CTA size. However, they allow as many warps to run on each SM as shared core resources (e.g., registers, shared scratchpad memory) permit. Consequently, even if the programmer specifies small CTAs, multiple CTAs will run on the same SM if resources permit. As a result, the number of warps/warps running at once may still be too large a working set for the L1D. For this reason, we propose *static warp limiting* (SWL) which is implemented as a minor extension to the warp scheduling logic where a register is used to determine how many warps are actively issued, independent of CTA size.

In SWL, the programmer must specify a limit on the number of warps when launching the kernel. This technique is useful if the user knows the optimal number of warps prior to launching the kernel, which could be determined by profiling.

In benchmarks that make use of work group level synchronization, SWL limits the number of warps running until a barrier, allows the rest of the work-group to reach the barrier, then continues with the same multithreading constraints.

In Section 3.4 we demonstrate that the optimal number of warps is different for different benchmarks. Moreover, we find this number changes in each benchmark when its input data is changed. This dependence on benchmark and input data makes an adaptive CCWS system desirable.

3.2.3 Cache-Conscious Warp Scheduling (CCWS)

This subsection first defines the goal and high level implementation of CCWS in Section 3.2.3. Next, Section 3.2.3 details how CCWS is applied to the baseline scheduling logic. Section 3.2.3 explains the *lost intra-warp locality detector* (LLD), followed by Section 3.2.3 which explains how our locality scoring system makes use of LLD information to determine which warps can issue. Finally, Section 3.2.3 describes the locality score value assigned to a warp when lost locality is detected.

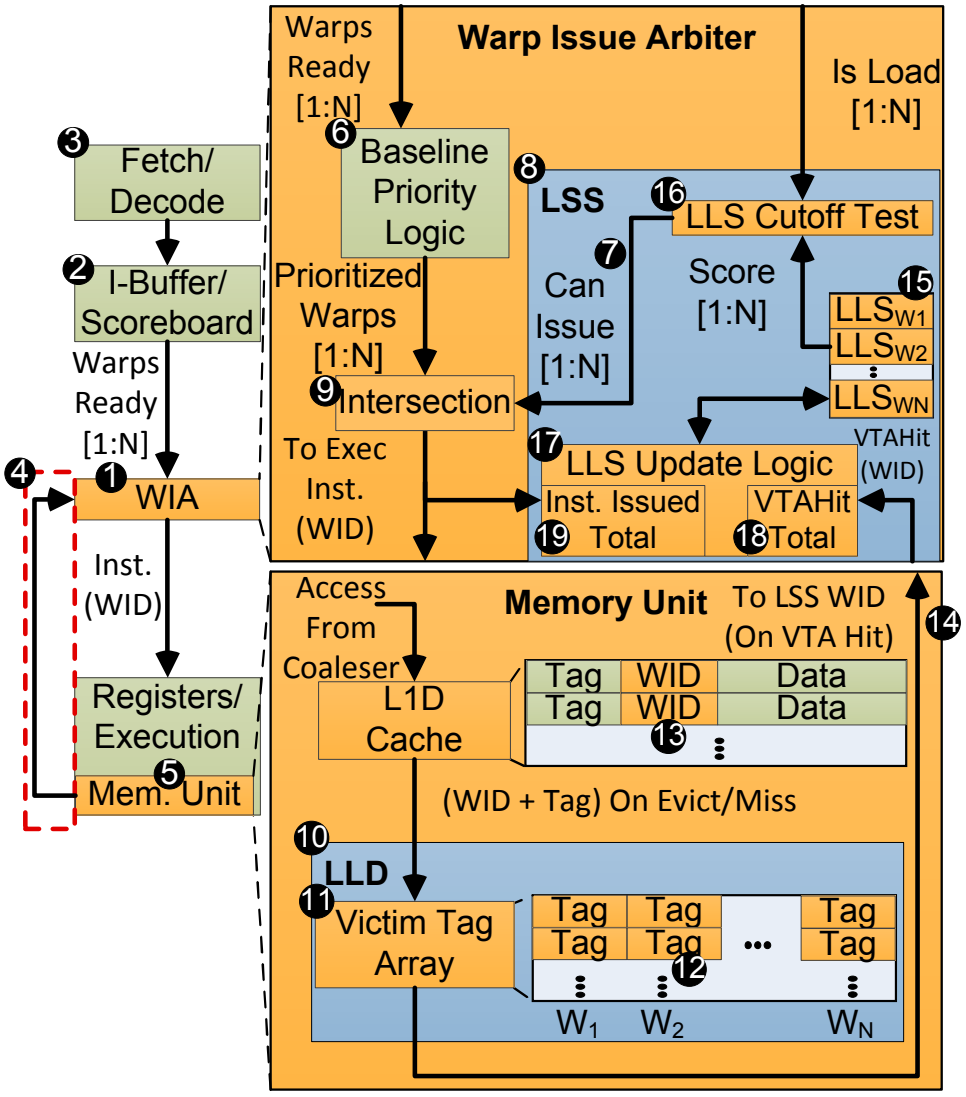


Figure 3.6: Modeled GPU core microarchitecture. N is the number of warp contexts stored on a core. LSS=*locality scoring system*, LLD=*lost intra-warp locality detector*, WID=*warp ID*, LLS=lost-locality score, VTA=*victim tag array*, I-Buffer=*instruction buffer*

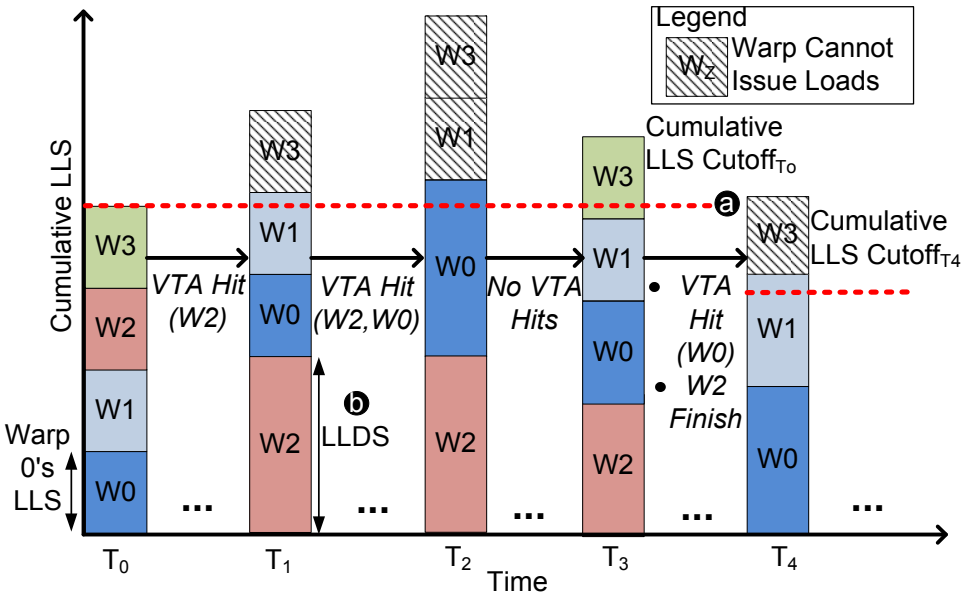


Figure 3.7: Locality scoring system operation example. LLS=*lost-locality score*, LLDS=*lost-locality detected score*

High-Level Description

The goal of CCWS is to dynamically determine the number of warps allowed to access the memory system and which warps those should be. At a high level, CCWS is a warp scheduler that reacts to access level feedback (❹ in Figure 3.6) from the L1D cache and a *victim tag array* (VTA) at the memory stage. CCWS uses a dynamic locality scoring system to make scheduling decisions.

The intuition behind why our scoring system works can be explained by Figure 3.7. At a high level, each warp is given a score based on how much *intra-warp locality* it has lost. These scores change over time. Warps with the largest scores fall to the bottom of a sorted stack (for example, W_2 at T_1), pushing warps with smaller scores above a cutoff (W_3 at T_1) which prevents them from accessing the L1D. In effect, the locality scoring system reduces the number of accesses between data re-references from the same warp by removing the accesses of other warps. The following subsections describe CCWS in more detail.

Effect on Baseline Issue Logic

Figure 3.6 shows the modifications to the baseline warp issue arbiter (❶) and memory unit (❺) required for CCWS. CCWS is implemented as an extension to the system's baseline warp prioritization logic (❻). This prioritization could be done in a greedy, round-robin or two level manner. CCWS operates by preventing loads that are predicted to interfere with intra-warp locality from issuing through a "Can Issue" bit vector (❼) output by the locality scoring system (❽). The intersection logic block (❾) selects the highest priority ready warp that has issue permission.

Lost Intra-Warp Locality Detector (LLD)

To evaluate which warps are losing intra-warp locality, we introduce the LLD unit (❿) which uses a *victim tag array* (VTA) (⓫). The VTA is a highly modified variation of a victim cache [82]. The sets of the VTA are sub-divided among the all the warp contexts supported on this core. This gives each warp its own small VTA (⓬). The VTA only stores cache tags and does not store line data. When a miss occurs and a line is reserved in the L1D cache, the *warp ID* (WID) of the warp reserving that line is written in addition to the tag (⓭). When that line is evicted from the cache, its tag information is written to that warp's portion of the VTA. Whenever there is a miss in the L1D cache, the VTA is probed. If the tag is found in that warp's portion of the VTA, the LLD sends a VTA hit signal to the locality scoring system (⓮). These signals inform the scoring system that a warp has missed on a cache line that may have been a hit if that warp had more exclusive access to the L1D cache.

Locality Scoring System Operation

Figure 3.7 provides a visual example of the locality scoring system's operation. In this example, there are four warps initially assigned to the SM. Time T_0 corresponds to the time these warps are initially assigned to this core. Each segment of the stacked bar represents a score given to each warp to quantify the amount of intra-warp locality it has lost. We call these values *lost-locality score*s (LLS). At T_0 we assign each warp a constant base locality score. LLS values are stored in a max heap (⓯) inside the locality scoring system. A warp's LLS can increase when

the LLD sends a VTA hit signal for this warp. The scores each decrease by one point every cycle until they reach the base locality score. The locality scoring system gives warps losing the most intra-warp locality more exclusive L1D cache access by preventing the warps with the smallest LLS from issuing load instructions. Warps whose LLS falls above the cumulative LLS cutoff (ⓐ in Figure 3.7) in the sorted heap are prevented from issuing loads. The value of the cumulative LLS cutoff is defined as $NumActiveWarps \times BaseLocalityScore$, where $NumActiveWarps$ is the number of warps currently assigned to this core.

The LLS cutoff test block (16) takes in a bit vector from the instruction buffer indicating what warps are attempting to issue loads. It also takes in a sorted list of LLSs, performs a prefix sum and clears the "Can Issue" bit for warps attempting to issue loads whose LLS is above the cutoff. The locality scoring system is not on the critical path, can be pipelined and does not have to update the score cutoffs every SM cycle. In our example from Figure 3.7, between T_0 and T_1, W_2 has received a VTA hit and its score has been increased to the *lost-locality detected score* (LLDS), (ⓑ in Figure 3.7). Section 3.2.3 explains the LLDS in more detail. W_2's higher score has pushed W_3 above the cumulative LLS cutoff, clearing W_3's "Can Issue" bit if it attempts to issue a load instruction. From a microarchitecture perspective, LLSs are modified by the score update logic (17). The update logic block receives VTA hit Signals (with a WID) from the LLD which triggers a change to that warp's LLS. We limit the amount one warp can dominate the point system by capping each warp's score at LLDS, regardless of how many VTA hits it has received. Other methods of capping a warp's LLS were attempted and we found that limiting them to the LLDS simplified the point system and yielded the best results. In the example, between T_1 and T_2 both W_2 and W_0 have received VTA hits, pushing both W_3 and W_1 above the cutoff. Between T_2 and T_3, no VTA hits have occurred and the scores for W_2 and W_0 have decreased enough to allow both W_1 and W_3 to issue loads again. This illustrates how the system naturally backs off thread throttling over time. Between time T_3 and T_4, W_2 finishes and W_0 has received a VTA hit to increase its score. This illustrates that when a warp is added or removed from the system, the cumulative LLS cutoff changes. Now that there are three warps active, the LLS cutoff becomes $3\times$ the base score. Having the LLS cutoff be a multiple of the number of active warps ensures the locality scoring

system maintains its sensitivity to lost-locality. If the LLS cutoff does not decrease when the number of warps assigned to this core decreases, it takes a higher score per warp to push lower scores above the cutoff as the kernel ends. This results in the system taking more time to both constrain multithreading when locality is lost and back off thread limiting when there is no lost locality.

Determining the Lost-Locality Detected Score (LLDS)

The value assigned to a warp's score on a VTA hit (the LLDS) is a function of the total number of VTA hits across all this SM's warps (18) and all the instructions this SM has issued (19). This value is defined by Equation (1).

$$LLDS = \frac{VTAHits_{Total}}{InstIssued_{Total}} \cdot K_{THROTTLE} \cdot CumLLSCutoff \tag{3.1}$$

Using the fraction of total VTA hits divided by the number of instructions issued serves as an indication of how much locality is being lost on this core per instruction issued. A constant ($K_{THROTTLE}$) is applied to this fraction to tune how much throttling is applied when locality is lost. A larger constant favors less multithreading by pushing warps above the cutoff value more quickly and for a longer period of time. Finding the optimal value of $K_{THROTTLE}$ is dependent on several factors including the number threads assigned to a core, the L1D cache size, relative memory latencies and locality in the workload. We intend for this constant to be set for a given chip configuration and not require any programmer or OS support. In our study, a single value for $K_{THROTTLE}$ used across all workloads captures 95.4% to 100% of the performance of any workload's optimal $K_{THROTTLE}$ value. This static value is determined experimentally and explored in more detail in Section 3.4.5. Like the LLS cutoff test, the lost-locality detected score can take several cycles to update and does not impact the critical path.

In algorithms that use synchronization primitives, CCWS does not introduce new deadlock conditions. The LLSs of warps preventing others from issuing will be backed off as time progresses while no lost locality is detected. This backing off insures that if no instructions are issuing for a prolonged period of time, every warp in the core will eventually be permitted to issue.

3.3 Experimental Methodology

We model the cache-conscious scheduling mechanisms as described in Section 3.2 in GPGPU-Sim [19] (version 3.1.0) using the configuration in Table 3.1. The Belady-optimal replacement policy [26], which chooses the line which is re-referenced furthest in the future for eviction, is evaluated using a custom *stand alone GPGPU-Sim cache simulator* (SAGCS). SAGCS is a trace based cache simulator that takes GPGPU-Sim cache access traces as input. Since SAGCS is not a performance simulator and only provides cache information, we do not present IPC results for the Belady-optimal replacement policy. To validate SAGCS, we verified the miss rate for LRU replacement using SAGCS and found that it was within 0.1% of the LRU miss rate reported using GPGPU-Sim. This small difference is a result of variability in the GPGPU-Sim memory system that SAGCS does not take into account.

We perform our evaluation using the high-performance computing GPU-enabled server workloads listed in Table 3.2 from Rodinia [36], Hetherington et al. [62] and Bakhoda et al. [19]. While the regularity of the HPC applications makes them particularly well suited for the GPU, they represent only one segment of the overall computing market [66] [67].

In addition to the cache-sensitive benchmarks introduced earlier, we also evaluate against a number of *cache-insensitive* (CI) benchmarks to ensure CCWS does not have a detrimental effect.

To make use of a larger input, the KMN benchmark was slightly modified to use global memory in place of both texture and constant memory.

All of our benchmarks run from beginning to end which takes between 14 million and 1 billion instructions.

3.3.1 GPU-enabled server workloads

This work uses two GPU-enabled server workloads. These benchmarks were ported to OpenCL from existing CPU implementations. They represent highly parallel code with irregular memory access patterns whose performance could be improved by running on the GPU.

Memcached-GPU (MEMC) Memcached is a key-value store and retrieval system. Memcached-GPU is described in detail by Hetherington et al. [62].

Table 3.1: Cache-conscious warp scheduling GPGPU-Sim Configuration

# Compute Units	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 128B line, 8-way assoc. LRU
L2 Unified Cache	128k/Memory Channel, 128B line, 8-way assoc. LRU
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	out of order (FR-FCFS)
Branch Divergence Method	PDOM [55]
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

The application is stimulated with a representative portion of the Wikipedia access trace collected by Urdaneta et al. [163].

Tracing Garbage Collector (GC) Garbage collection is an important aspect of many server applications. Languages such as Java use system-controlled garbage collection to manage resources [11]. A version of the tracing mark-and-compact garbage collector presented in Barabash et al. [21] is created in OpenCL. The collector is stimulated with benchmarks provided by Spoonhower et al. [154].

3.4 Experimental Results

This section is structured as follows, Section 3.4.1 presents the performance of SWL, CCWS, other related warp schedulers and the Belady-optimal replacement

Table 3.2: GPU Compute Benchmarks (CUDA and OpenCL)

Highly Cache Sensitive (HCS)			
Name	**Abbr.**	**Name**	**Abbr.**
BFS Graph Traversal [36]	BFS	Kmeans [36]	KMN
Memcached [62]	MEMC	Garbage Collection [21, 154]	GC
Moderately Cache Sensitive (MCS)			
Name	**Abbr.**	**Name**	**Abbr.**
Weather Prediction [36]	WP	Streamcluster [36]	STMCL
Single Source Shortest Path [19]	SSSP	CFD Solver [36]	CFD
Cache Insensitive (CI)			
Name	**Abbr.**	**Name**	**Abbr.**
Needleman-Wunsch [36]	NDL	Back Propagation [19]	BACKP
Speckle Red. Anisotropic Diff. [36]	SRAD	LU Decomposition [36]	LUD

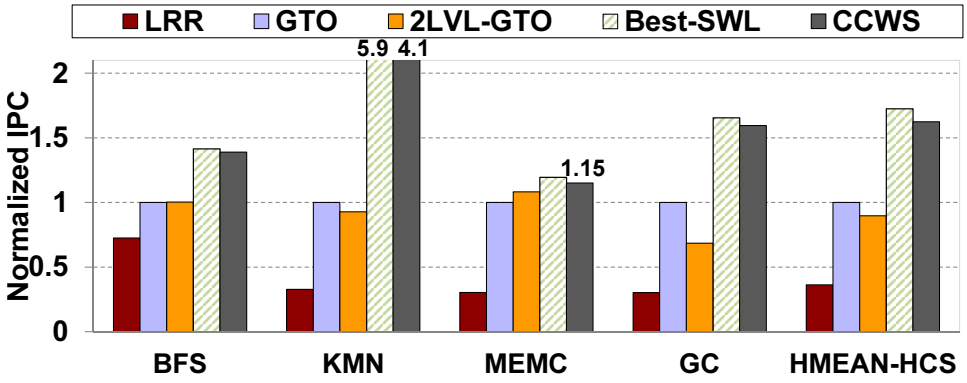


Figure 3.8: Performance of various schedulers and replacement policies for the highly cache-sensitive benchmarks. Normalized to the GTO scheduler.

policy using the system presented in Section 3.3. The results for CCWS presented in Section 3.4.1 represent a design point that maximizes performance increase over area increase. The remainder of this section is devoted to exploring the sensitivity of our design and explaining the behaviour of our benchmarks.

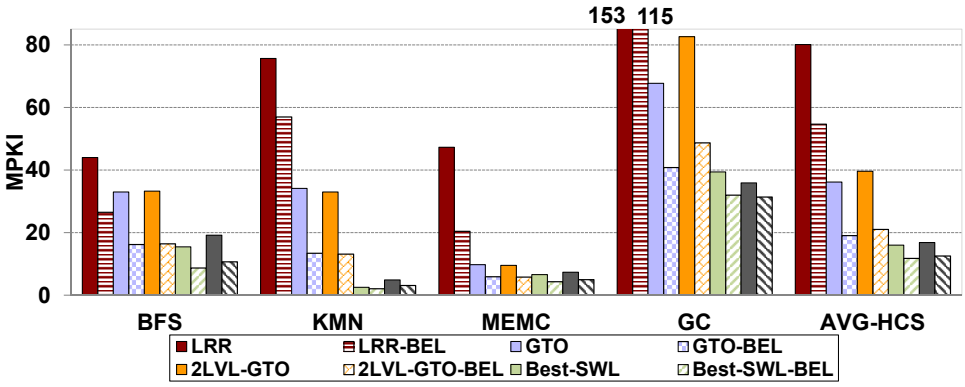


Figure 3.9: MPKI of various schedulers and replacement policies for the highly cache-sensitive benchmarks.

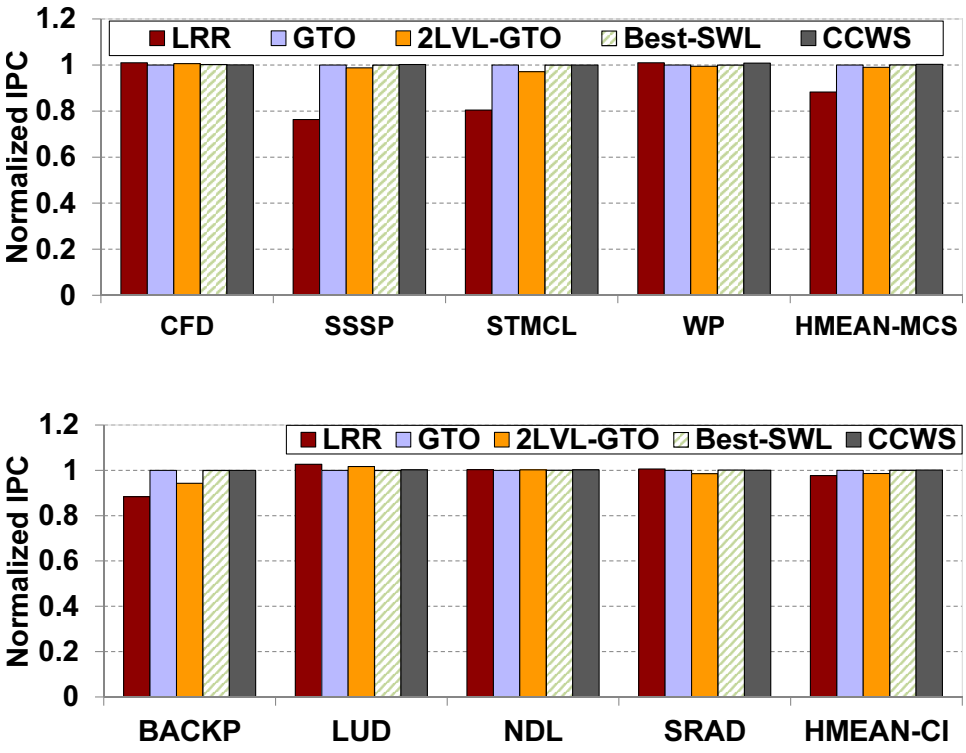


Figure 3.10: Performance of various schedulers and replacement policies for moderately cache-sensitive (top) and cache-insensitive (bottom) benchmarks. Normalized to the GTO scheduler.

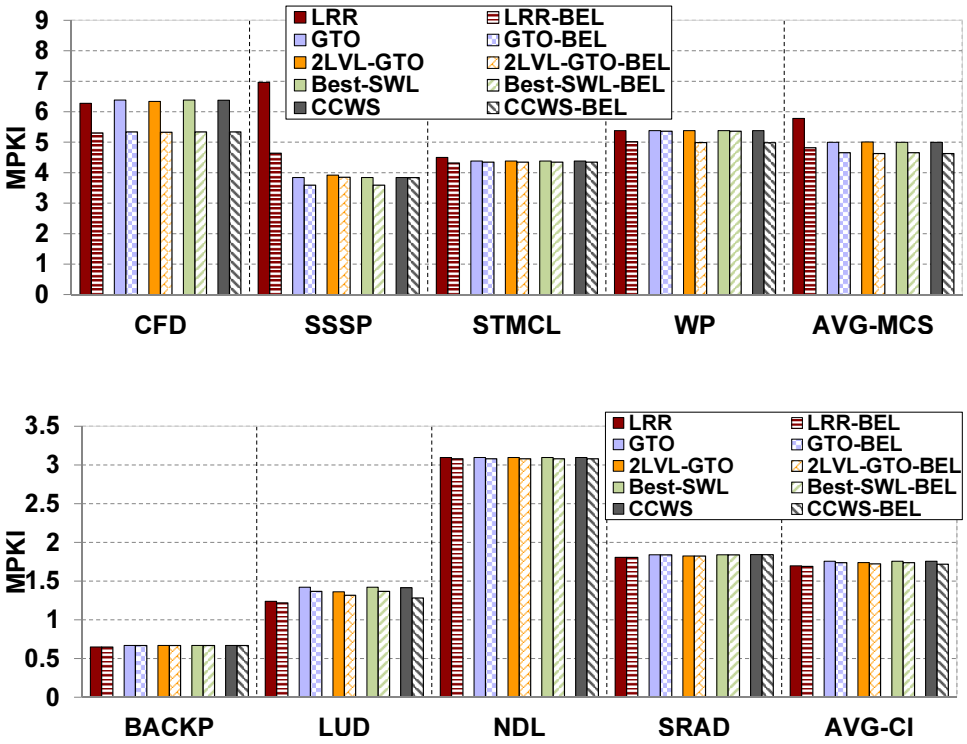


Figure 3.11: MPKI of various schedulers and replacement policies for moderately cache-sensitive (top) and cache-insensitive benchmarks (bottom).

3.4.1 Performance

The data in Figures 3.8, 3.9, 3.10 and 3.11 is collected using GPGPU-Sim for the following mechanisms:

LRR Loose round-robin scheduling. Warps are prioritized for scheduling in round-robin order. However, if a warp cannot issue during its turn, the next warp in round-robin order is given the chance to issue.

GTO A greedy-then-oldest scheduler. GTO runs a single warp until it stalls then picks the oldest ready warp. The age of a warp is determined by the time it is assigned to the core. For warps that are assigned to a core at the same time (i.e. they are in the same CTA), warps with the smallest threads IDs

are prioritized. Other greedy schemes (such as greedy-then-round-robin and oldest-first) were implemented and GTO scheduling had the best results.

2LVL-GTO A two-level scheduler similar to that described by Narasiman et al. [123]. Their scheme subdivides warps waiting to be scheduled on a core into *fetch groups* (FG) and executes from only one fetch group until all warps in that group are stalled. Narasiman et al. used a fetch group size of 8 and a round-robin scheduling policy to select among warps in a fetch group as well as among fetch groups. To provide a fair comparison against their scheduling technique in our simulator and on our workloads, all fetch group sizes were swept. We also explored alternate scheduling policies for intra-FG and inter-FG selection. We found using GTO for both of these policies was better than the algorithm they employed. A fetch group size of 2 using GTO for both intra-FG and inter-FG selection provides the best performance on our workloads and is what we present in our results. This disparity in optimal configuration can be explained by the nature of our workloads and our baseline architecture. Their core pipeline allows only one instruction from a given warp to be executing at a time. This means that a warp must wait for its previously issued instruction to complete execution before the warp can issue another instruction. This is different from our baseline which prevents a fetched instruction from issuing if the scoreboard detects a data hazard.

Best-SWL Static Warp Limiting as described in Section 3.2.2. All possible limitation values (32 to 1) were run and the best performing case is picked. The GTO policy is used to select between warps. The warp value used for each benchmark is shown in Table 3.3.

CCWS Cache-Conscious Warp Scheduling described in Section 3.2.3 with the configuration parameters listed in Table 3.3. GTO warp prioritization logic is used.

The data for Belady-optimal replacement *misses per thousand instructions* (MPKI) presented in Figures 3.9 and 3.11 is generated with SAGCS:

<scheduler>-BEL Miss miss rate reported by SAGCS when using the Belady-optimal replacement policy. SAGCS is stimulated with L1D access streams

generated by using GPGPU-Sim running the specified <scheduler>. Since SAGCS only reports misses, MPKI is calculated from the GPGPU-Sim instruction count.

Figure 3.8 shows that CCWS achieves a harmonic mean 63% performance improvement over a simple greedy warp scheduler and 72% over the 2LVL-GTO scheduler on HCS benchmarks. The GTO scheduler performs well because prioritizing older warps allows them to capture intra-warp locality by giving them more exclusive access to the L1 data cache. The 2LVL-GTO scheduler performs slightly worse than the GTO scheduler because the 2LVL-GTO scheduler will not prioritize the oldest warps every cycle. 2LVL-GTO only attempts to schedule the oldest FG intermittently, once the current FG is completely stalled. This allows loads from younger warps, which would not have been prioritized in the GTO scheduler, to be injected into the access stream, causing older warp's data to be evicted.

CCWS and SWL provide further benefit over the GTO scheduler because these programs have a number of uncoalesced loads, touching many cache lines in relatively few memory instructions. Therefore, even restricting to just the oldest warps still touches too much data to be contained by the L1D. The GTO, 2LVL-GTO, Best-SWL and CCWS schedulers see a greater disparity in the completion time of CTAs running on the same core compared to the LRR scheduler. Since all our workloads are homogeneous (at any given time only CTAs from one kernel launch will be assigned to each core) and involve synchronous kernel launches, the relative completion time of CTAs is not an issue. All that matters is when the whole kernel finishes. Moreover, the highly cache-sensitive workloads we study do not use any CTA or global synchronization within a kernel launch, therefore older warps are never stalled waiting for younger ones to complete.

Figure 3.8 also highlights the importance of scheduler choice even among simple schedulers like GTO and LRR. The LRR scheduler suffers from a 64% slowdown compared to GTO. Scheduling warps with a lot of intra-warp locality in a RR fashion strides through too much data to be contained in the L1D. Best-SWL is able to slightly outperform CCWS on all the benchmarks. The CCWS configuration used here has been optimized to provide the highest performance per unit area. If the VTA cache is doubled in size, CCWS is able to slightly outperform

Best-SWL on some workloads. CCWS is not able to consistently outperform Best-SWL because there is a start-up cost associated with detecting the loss of locality and a cool-down cost to back off the warp throttling. Adding to that, the execution time of these kernels is dominated by the code section that benefits from warp limiting. Therefore, providing the static scheme with oracle knowledge (through profiling) gives it an advantage over the adaptive CCWS scheme. Section 3.4.6 examines the shortcomings of the SWL under different run-time conditions.

Although not plotted here, it is worth mentioning the performance of the 2LVL-LRR scheduling configuration evaluated by Narasiman et al. On the HCS benchmarks the 2LVL-LRR scheduler is a harmonic mean 43% faster than the LRR scheduler, however this is still 47% slower than the GTO scheduler. Performing intra-FG and inter-FG scheduling in a round-robin fashion destroys the intra-warp locality of older warps that is captured by the GTO scheduler. However, in comparison to the LRR scheduler, which cycles through 32 warps in a round-robin fashion, cycling through smaller FG sized pools (each fetch group has 8 warps in their configuration) will thrash the L1 data cache less.

Figure 3.9 illustrates that the reason for the performance advantage of the warp limiting schemes is a sharp decline in the number of L1D misses. This figure highlights the fact that no cache replacement policy can make up for a poor choice in warp scheduler, as even an oracle Belady-optimal policy on the LRR access stream is outperformed by all the schedulers. The insight here is that even optimal replacement cannot compensate for an access stream that strides through too much data, at least for the relatively low associativity L1 data caches we evaluated.. Furthermore, the miss rate of CCWS outperforms both GTO-BEL and 2LVL-GTO-BEL. This data suggests L1D cache hit rates are more sensitive to warp scheduling policy than cache replacement policy.

Figures 3.10 and 3.11 present the performance and MPKI of our MCS and CI benchmarks. The harmonic mean performance improvement of CCWS across both the highly and moderately cache-sensitive (HCS and MCS) benchmarks is 24%. In the majority of the MCS and CI workloads, the choice of warp scheduler makes little difference and CCWS does not degrade performance. There is no degradation because the MPKI for these benchmarks is much lower than the HCS applications, so there are few VTA hits compared to instructions issued. As a result

Table 3.3: Configurations for Best-SWL (warps actively scheduled) and CCWS variables used for performance data.

Best-SWL		CCWS Config	
Benchmark	Warps Actively Scheduled	Name	Value
BFS	5	$K_{THROTTLE}$	8
KMN	4	Warp Base Score	100
MEMC	7	VTA Tag array	8-way
GC	4		16 entries per warp
All Others	32		(512 total entries)

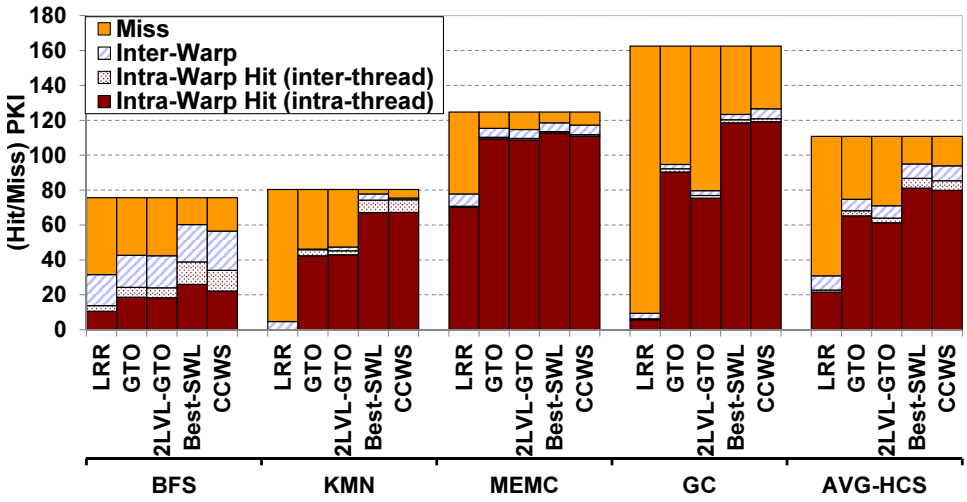


Figure 3.12: Breakdown of L1D misses, intra-warp locality hits (broken into intra-thread and inter-thread) and inter-warp locality hits per thousand instructions for highly cache-sensitive benchmarks. The configuration from Section 3.4.1 is used.

the lost-locality detected score as defined by Equation (1) stays low and the thread throttling mechanism does not take effect.

3.4.2 Detailed Breakdown of Inter- and Intra-Warp Locality

Figure 3.12 breaks down L1D accesses into misses, inter-warp hits and intra-warp hits for all the schedulers evaluated in Section 3.4.1 on our HCS benchmarks. In addition, it quantifies the portion of intra-warp hits that are a result of intra-thread

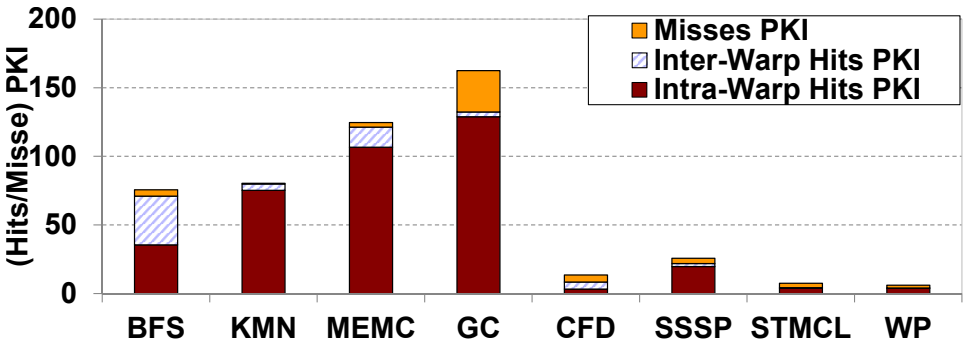


Figure 3.13: Breakdown of L1D misses, intra-warp locality Hits and inter-warp locality PKI using an unbounded L1 cache with 128 byte cache lines.

locality. It illustrates that the decrease in cache misses using CCWS and Best-SWL comes chiefly from an increase in intra-warp hits. Moreover, the bulk of these hits are a result of intra-thread locality. The exception to this rule is BFS, where 30% of intra-warp hits come from inter-thread locality and we see a 23% increase in inter-warp hits. An inspection of the code reveals that inter-thread sharing (which manifests itself as both intra-warp and inter-warp locality) occurs when nodes in the graph share neighbours. Limiting the number of warps actively scheduled increases the hit rate of these accesses because it limits the amount of non-shared data in the cache, increasing the chance that these shared accesses hit.

Figure 3.13 explores the access stream of all the cache-sensitive benchmarks using SAGCS and an unbounded L1D. It shows that with the exception of SSSP, the MCS benchmarks have significantly less locality in the access stream. The larger amount of intra-warp locality in SSSP is consistent with the significant performance improvement we observe for CCWS at smaller cache sizes when the working set of all the threads does not fit in the L1D cache (see Figure 3.15).

3.4.3 Sensitivity to Victim Tag Array Size

Figure 3.14 shows the effect of varying the VTA size on performance. With a larger victim tag array the system is able to detect lost intra-warp locality occurring at further access distances. Increasing the size of the VTA keeps data with intra-warp locality in the VTA longer and causes warp limiting to be appropriately applied.

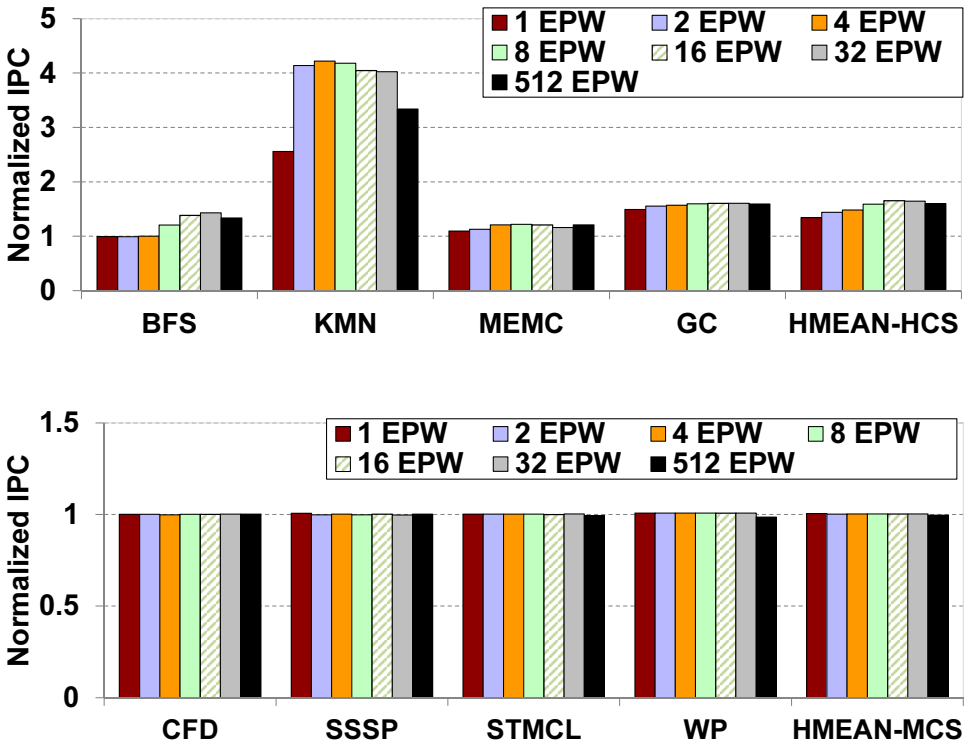


Figure 3.14: Performance of CCWS at various victim tag array sizes. Normalized to the GTO scheduler. EPW=Entries per Warp. EWP 1-4 are 1-4 set associative respectively. All other victim tag arrays are 8-way set associative.

However, if the VTA size is increased too much, the lost-locality detector's time sensitivity is diminished. The VTA will contain tags from data that was evicted from the L1 data cache so long ago that it would have been difficult to capture with changes to the scheduling policy. For example, at the 512 entry design point, each warp has a VTA that can track as much data as the entire L1D. In this configuration, a warp would need exclusive access to the L1 data cache to prevent all the detected loss of locality. The increase in detected lost-locality results in excessive warp constraining on some workloads. Based on this data, the best-performing configuration with 16 entries per warp is selected.

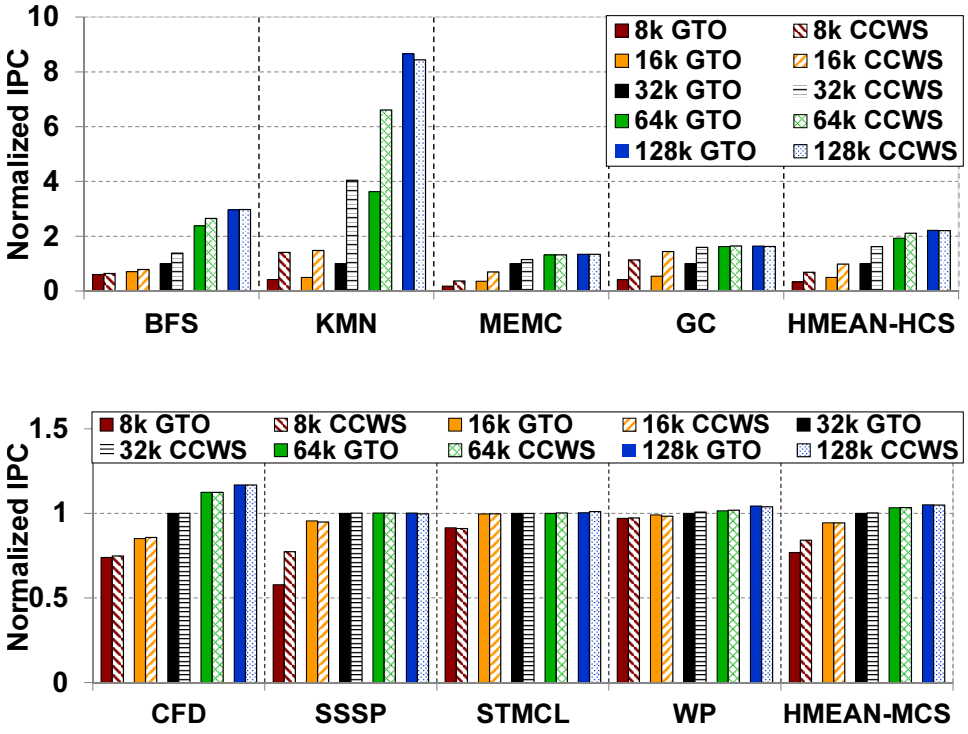


Figure 3.15: Performance of CCWS and GTO at various cache sizes. Normalized to the GTO scheduler with a 32k L1D. All caches are 8-way set associative. The VTA Size is 16 entries per warp for all instances of CCWS.

3.4.4 Sensitivity to Cache Size

Figure 3.15 shows the sensitivity of CCWS to the L1D size. As the cache size decreases, CCWS has a greater performance improvement relative to the GTO scheduler. This is because at small cache sizes it is even more desirable to limit multithreading to reduce cache footprint. In fact SSSP, which showed no performance gain at 32k shows a 35% speedup when the L1 cache is reduced to 8k. This is because SSSP has significant intra-warp locality but its footprint is small enough that it is contained by a 32k L1D. As the cache size increases, the effect of CCWS dwindles relative to the GTO scheduler because the working set of most warps fit in a larger cache. At a large enough cache size, the choice of warp scheduler makes little difference.

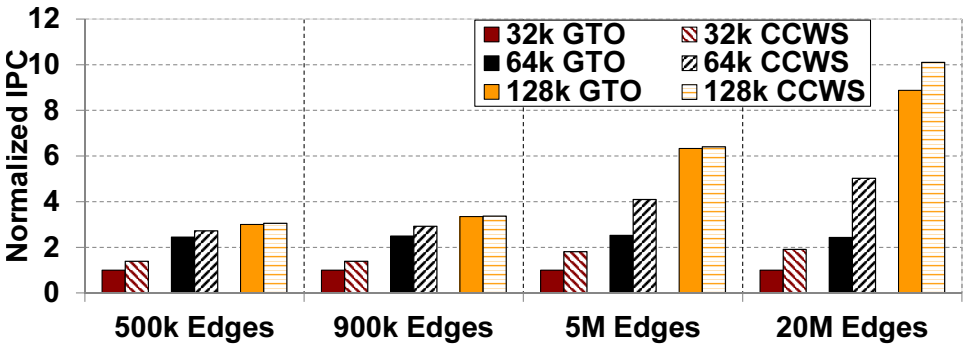


Figure 3.16: Performance of CCWS on BFS with different graph sizes when varying the L1D cache size and scheduler choice. Normalized to the GTO scheduler with a 32k L1D. The VTA size is 16 entries per warp for all instances of CCWS.

At 128k per L1D, CCWS shows little benefit over the GTO scheduler. This is because the input to these benchmarks is small enough that 128k captures most of the intra-warp locality. Since we are collecting results on a performance simulator that runs several orders of magnitude slower than a real device, the input to our benchmarks is small enough that they finish in a reasonable amount of time. Figure 3.16 show the effect of increasing the size of the BFS input graph from the baseline 500k edges to 20M edges. As the input size increases, the performance of CCWS over the GTO scheduler also increases even at a 128k L1 cache size. We observe that simply increasing the capacity of the L1 cache only diminishes the performance impact of CCWS with small enough input sets. Hence, we believe CCWS will have an even greater impact on data sizes used in real workloads.

3.4.5 Sensitivity to $K_{THROTTLE}$ and Tuning for Power

Figure 3.17 shows the effect of varying $K_{THROTTLE}$ on L1D misses and performance. $K_{THROTTLE}$ is the constant used in Equation (1) to tune the score assigned to warps when lost locality is detected (LLDS). At smaller $K_{THROTTLE}$ values, there is less throttling caused by the point system and more multithreading. At the smallest values of $K_{THROTTLE}$ multithreading is not constrained enough and performance suffers. As $K_{THROTTLE}$ increases, CCWS has a greater effect and the number of L1D misses falls across all the HCS benchmarks. In every HCS bench-

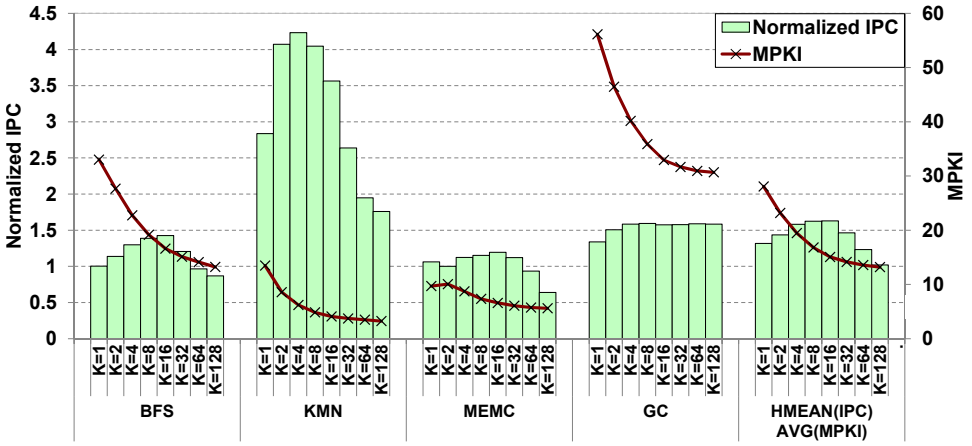


Figure 3.17: Performance of CCWS (normalized to the GTO scheduler) and MPKI of CCWS when varying $K_{THROTTLE}$.

mark, except GC, performance peaks then falls as $K_{THROTTLE}$ increases. However, since a miss in the L1D cache can incur a significant power cost it may be desirable to use a higher $K_{THROTTLE}$ value to reduce L1D misses at the cost of some performance. For example, at $K_{THROTTLE} = 32$ there is an average 18% reduction in L1D misses over the chosen $K_{THROTTLE} = 8$ design point. $K_{THROTTLE} = 32$ still achieves a 46% performance improvement over the GTO scheduler.

Figure 3.17 also demonstrates that each benchmark has a different optimal $K_{THROTTLE}$ value. However, the difference in harmonic mean performance between choosing each benchmark's optimal $K_{THROTTLE}$ value and using a constant $K_{THROTTLE} = 8$ is $< 4\%$. For this reason, we do not pursue an online mechanism for determining the value of $K_{THROTTLE}$. If other HCS benchmarks have more variance in their intra-warp locality then such a system should be considered.

The value of $K_{THROTTLE}$ makes no difference in the CI benchmarks since there is little locality to lose and few VTA Hits are reported. In the MCS benchmarks there are relatively few L1D MPKI, which keeps the product of $K_{THROTTLE}$ and $\frac{VTAHits_{Total}}{InstIssued_{Total}}$ low. In the MCS benchmarks, CCWS performance matches GTO scheduler performance until $K_{THROTTLE} = 128$. At this point there is a harmonic mean 4% performance degradation due to excessive throttling. Since their performance is largely unchanged by the value of $K_{THROTTLE}$, we do not graph the MCS

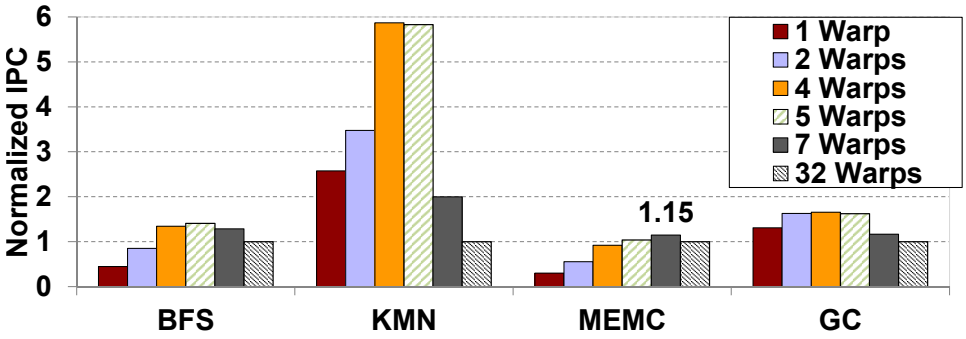


Figure 3.18: Performance of SWL at various multithreading limits. Normalized to 32 warps.

or CI benchmarks in Figure 3.17.

3.4.6 Static Warp Limiting Sensitivity

In Section 3.2 we noted that the optimal SWL limiting number was different for different benchmarks. We also indicated that this value changes when running the same benchmark with different input sets. Figure 3.18 illustrates that peak performance for each of the HCS benchmarks occurs with different multithreading limits. This happens because each workload has a different working set and access stream characteristics. Furthermore, Figure 3.19 shows that for different input graphs on BFS, the values of the peak performance point are different. This variation happens because the working set size is input data dependent. Finding the optimal warp limiting number in SWL would require profiling of each instance of a particular workload, making the adaptive CCWS more practical.

SWL also suffers in programs that have phased execution. The larger and more diverse the application is, the less likely a single warp limiting value will capture peak performance. This type of phased behaviour is not abundant in the HCS workloads we study, but as the amount and type of code running on the GPU continues to grow so too will the importance of adaptive multithreading.

SWL is also sub-optimal in a multi-programmed GPU. If warps from more than one type of kernel are assigned to the same SM, a per-kernel limiting number makes little sense. Even if there was no cache thrashing in either workload individually their combination may cause it to occur. CCWS will adapt to suit the needs

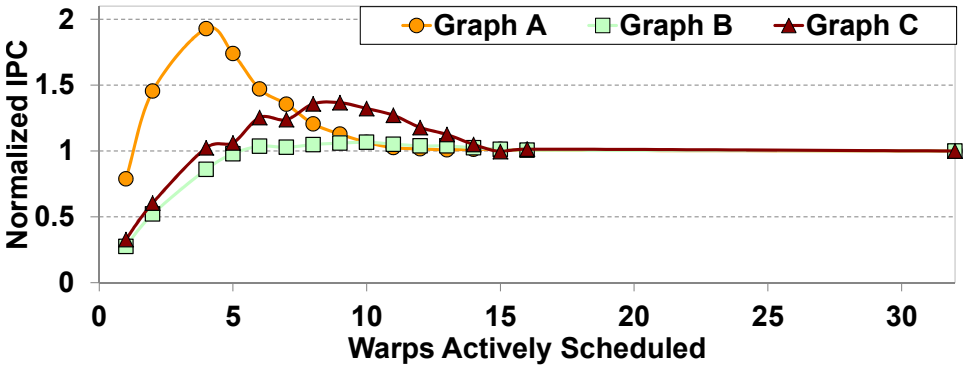


Figure 3.19: Performance of SWL with different multithreading values on BFS with different input graphs. Normalized to 32 warps.

of whatever warp combination is running on a SM and preserve their intra-warp locality. Since there will be no inter-warp locality among multi-programmed warps, preservation of intra-warp locality becomes even more important.

3.4.7 Area Estimation

The major source of area overhead to support CCWS comes from the victim tag array. For the configuration used in Table 3.3 and a 48-bit virtual address space, we require 40 bits for each tag entry in our VTA. Using CACTI 5.3 [171], we estimate that this tag array would consume 0.026 mm^2 per core at 55nm or 0.78 mm^2 for the entire 30 core system. This represents 0.17% of GeForce GTX 285 area, which our system closely models with the exception that we also model data caches. There are a variety of smaller costs associated with our design that are difficult to quantify and as a result are not included in the above estimation. Adding an additional 5-bits to each L1D cache line for the WID costs 160 bytes per core. There are 32 lost-locality score values, each represented in 10 bits which are stored in a max heap. Also, there are two counter registers, one for the number of instructions issued and another for the total VTA hit signals. In addition, there is logic associated with the scoring system. Compared to the other logic in a SM, we do not expect this additional logic to be significant.

3.5 Summary

This work introduces a new classification of locality for GPUs. We quantify the caching and performance effects of both intra- and inter-warp locality for workloads in massively multi-threaded environments.

To exploit the observation that intra-warp locality is of greatest importance on highly cache-sensitive workloads, this work introduces Cache-Conscious Warp Scheduling. CCWS is a novel technique to capitalize on the performance benefit of limiting the number of actively-scheduled warps, thereby limiting L1 data cache thrashing and preserving intra-warp locality. Our simulated evaluation shows this technique results in a harmonic mean 63% improvement in throughput on highly cache-sensitive benchmarks, without impacting the performance of cache-insensitive workloads.

We demonstrate that on massively multi-threaded systems, optimizing the low level thread scheduler is of more importance than attempting to improve the cache replacement policy. Furthermore, any work evaluating cache replacement on massively multi-threaded systems should do so in the presence of an intelligent warp scheduler.

As more diverse applications are created to exploit irregular parallelism and the number of threads sharing a cache continues to increase on both GPUs and CMPs, so too will the importance of intelligent HW thread scheduling policies, like CCWS.

Chapter 4

Divergence-Aware Warp Scheduling

This chapter uses hardware thread scheduling to improve the performance and energy efficiency of divergent applications on GPUs. We propose Divergence-Aware Warp Scheduling (DAWS), which introduces a divergence-based cache footprint predictor to estimate how much L1 data cache capacity is needed to capture intra-warp locality in loops. Predictor estimates are created from an online characterization of memory divergence and runtime information about the level of control flow divergence in warps. Unlike prior work on Cache-Conscious Warp Scheduling, which makes reactive scheduling decisions based on detected cache thrashing, DAWS makes proactive scheduling decisions based on cache usage predictions. DAWS uses these predictions to schedule warps such that data reused by active scalar threads is unlikely to exceed the capacity of the L1 data cache. DAWS attempts to shift the burden of locality management from software to hardware, increasing the performance of simpler and more portable code on the GPU. We show that DAWS achieves a harmonic mean 26% performance improvement over Cache-Conscious Warp Scheduling on a diverse selection of highly cache-sensitive applications, with minimal additional hardware.

Running irregular code on a GPU can cause both memory and control flow divergence. Memory divergence (or an uncoalesced memory access) occurs when threads in the same warp access different regions of memory in the same SIMT

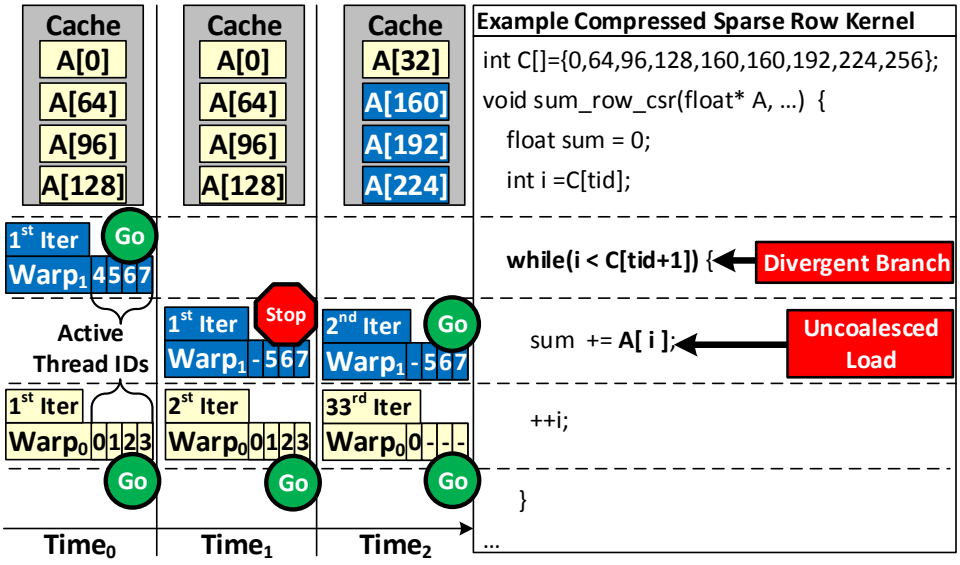


Figure 4.1: DAWS example. Cache: 4 entries, 128B lines, fully assoc. By $Time_0$, warp 0 has entered loop and loaded 4 lines into cache. By $Time_1$, warp 0 has captured spatial locality, DAWS measures footprint. Warp 1 is prevented from scheduling as DAWS predicts it will oversubscribe cache. By $Time_2$, warp 0 has accessed 4 lines for 32 iterations and loaded 1 new line. 3 lanes have exited loop, decreasing footprint. Warp 1 and warp 0 are allowed to capture spatial locality together.

instruction. Control flow (or branch) divergence occurs when threads in the same warp execute different control flow paths. This work focuses on improving the performance of several such irregular applications through warp scheduling.

Figure 4.1 presents a small example of divergent code to illustrate how scheduling can be used can make effective use of on-chip cache capacity. The example code sums each row of a *Compressed Sparse Row* (CSR) [22] data set. Each thread in the kernel sums one row using a loop. This code is divergent due to the data dependent nature of each sparse row's length and the position of each row's values in memory. This translates into branch divergence when threads within a warp travel through the loop a different number of times and memory divergence when threads access $A[i]$. This code has three key characteristics that can be leveraged to make effective use of cache capacity: (1) Each thread has spatial locality across

loop iterations, since i is incremented by 1. (2) Each warp's load to $A[i]$ can access multiple cache lines. (3) The number of cache lines accessed when a warp loads $A[i]$ is dependent on the warp's active mask. Figure 4.1 also illustrates how our proposed *Divergence-Aware Warp Scheduling* (DAWS) technique takes these characteristics into account to maximize on-chip cache utilization. In the example, two warps (each with 4 threads) share a cache with 4 entries. Warp 0 enters the loop first and each of its threads loads its section of A into the cache. During warp 0's execution of the loop, Divergence-Aware Warp Scheduling learns that there is both locality and memory divergence in the code. At Time_1, warp 1 is ready to enter the loop body. Divergence-Aware Warp Scheduling uses the information gathered from warp 0 to predict that the data loaded by warp 1's active threads will evict data reused by warp 0 which is still in the loop. To avoid oversubscribing the cache, Divergence-Aware Warp Scheduling prevents warp 1 from entering the loop by de-scheduling it. Now warp 0 captures its spatial locality in isolation until its threads begin to diverge. By Time_2, warp 0 has only one thread active and its cache footprint has decreased. Divergence-Aware Warp Scheduling detects this divergence and allows warp 1 to proceed since the aggregate footprint of warp 0 and warp 1 fits in cache.

The code in Figure 4.1 contains intra-warp locality. Intra-warp locality occurs when data is loaded then re-referenced by the same warp [137]. The programmer may be able to re-write the code in Figure 4.1 to remove intra-warp locality. Hong et al. [63] perform such an optimization to *Breadth First Search* (BFS). However, this can require considerable programmer effort. Another option is to have the compiler restructure the code independent of the programmer, however static compiler techniques to re-arrange program behaviour are difficult in the presence of data dependant accesses [152]. One of this chapter's goals is to enable the efficient execution of more workloads on accelerator architectures. We seek to decrease the programmer effort and knowledge required to use the hardware effectively, while adding little to the hardware's cost.

Previously proposed work on *Cache-Conscious Warp Scheduling* (CCWS) [137], presented in Chapter 3 uses a reactionary mechanism to scale back the number of warps sharing the cache when thrashing is detected. However, Figure 4.1 illustrates that cache footprints in loops can be predicted, allowing thread scheduling

decisions to be made in a proactive manner. Our technique reacts to changes in thread activity without waiting for cache thrashing to occur. By taking advantage of dynamic thread activity information, Divergence-Aware Warp Scheduling is also able to outperform a scheduler that statically limits the number of warps run based on previous profiling runs of the same workload [137].

This work focuses on a set of GPU accelerated workloads from server computing and high performance computing that are both economically important and whose performance is highly sensitive to *level one data* L1D cache capacity. These workloads encompass a number of applications from server computing such as Memcached [62], a key-value store application used by companies like Facebook and Twitter, and a sparse matrix vector multiply application [44] which is used in Big Data processing.

4.1 Divergence, Locality and Scheduling

A key observation of our work is that a program's memory divergence, control flow divergence and locality can be profiled, predicted and used by the warp scheduler to improve cache utilization. This section is devoted to describing this observation in detail and is divided into two parts. Section 4.1.1 explores where locality occurs in our highly cache-sensitive benchmarks and Section 4.1.2 classifies the locality in ways that are useful for our warp scheduler.

4.1.1 Application Locality

Figure 4.2 presents the hits and misses for all the static load instruction addresses (PCs) in our highly cache-sensitive benchmarks (described in Section 4.3). Each hit is classified as either an intra-warp hit (when data is loaded then re-referenced by the same warp) or an inter-warp hit (when one warp loads data that is hit on by another). This data was collected using Cache-Conscious Warp Scheduling. The loops in each program are highlighted by dashed boxes. This figure demonstrates that the bulk of the locality in our programs is intra-warp and comes from a few static load instructions. These load instructions are concentrated in the loops of the program.

To understand the locality in these loops, Figure 4.3 presents a classification of

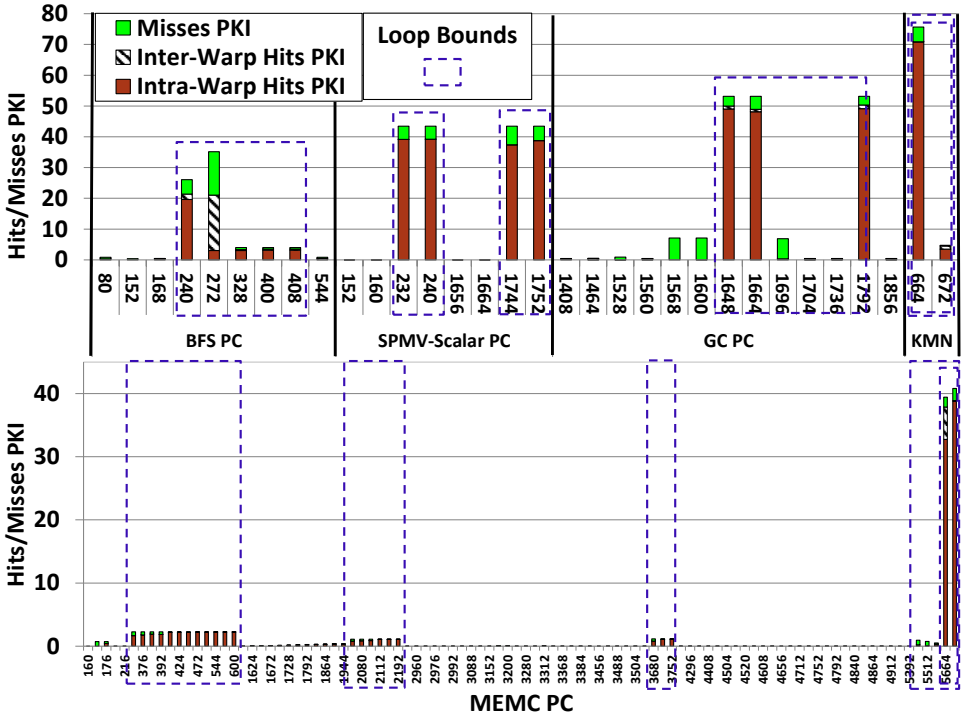


Figure 4.2: Intra-warp hits, inter-warps hits and misses *per thousand instructions* (PKI) for all the static load instructions in each of our highly cache-sensitive benchmarks, identified by PC. The PCs contained in loops are highlighted in dashed boxes.

intra-warp hits from loads within the loops of each application. Loads are classified as *Accessed-This-Trip* if the cache line was accessed by another load on this loop iteration. If the value in cache was not *Accessed-This-Trip*, then we test if it was accessed on the previous loop trip. If so, it is classified as *Accessed-Last-Trip*. If the line was not accessed on either loop trip, it is classified as *Other*, indicating that the line was accessed outside the loop or in a loop trip less recent than the last one. This data demonstrates that the majority of data reuse in these applications is *Accessed-Last-Tip*. If the scheduler can keep the data loaded by a warp on one loop iteration in cache long enough to be hit on in the next loop iteration, most of the locality in these applications can be captured.

To illustrate the source of this locality in the code, consider the code for SPMV-

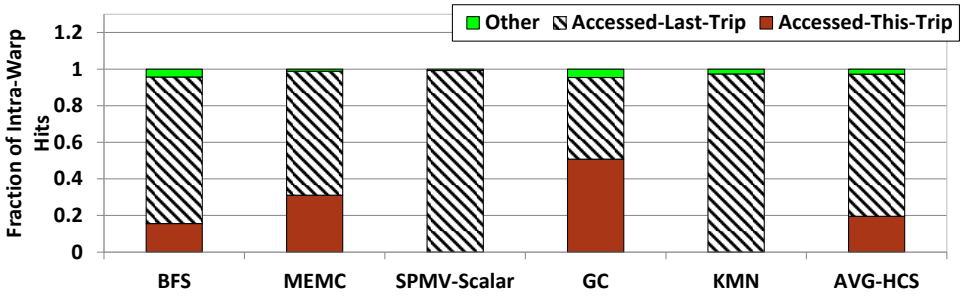


Figure 4.3: Classification of intra-warp hits within loops using an 8M L1D cache. Accessed-This-Trip=hit on data already accessed this loop iteration. Accessed-Last-Trip=hit on data accessed in immediately-previous loop iteration.

Scalar in Example 5.1. Figure 4.3 indicates that all of the intra-warp locality within the loop of this code is *Accessed-Last-Trip*. This comes from the loading $cols[j]$ and $val[j]$. When inside this loop, each thread walks the arrays in 4 byte strides since j is incremented by one each iteration.

Based on these observations, we design our scheduling system to ensure that when intra-warp locality occurs in a loop, much of the data loaded by a particular warp in one iteration remains in the cache for the next iteration. We attempt to ensure this happens by creating a cache footprint prediction for warps executing in loops. The prediction is created from information about the loads inside the loop and the current level of control flow divergence in a warp on its current loop iteration.

4.1.2 Static Load Classification

To predict the amount of data each warp will access on each iteration of the loop, we start by classifying the static load instructions inside the loop. We classify each static load instruction based on two criteria, memory divergence (detailed in Section 4.1.2) and loop trip repetition (Section 4.1.2).

Memory Divergence

If the number of memory accesses generated by a load equals the number of lanes active in the warp that issues it, then the load is completely diverged. Loads that

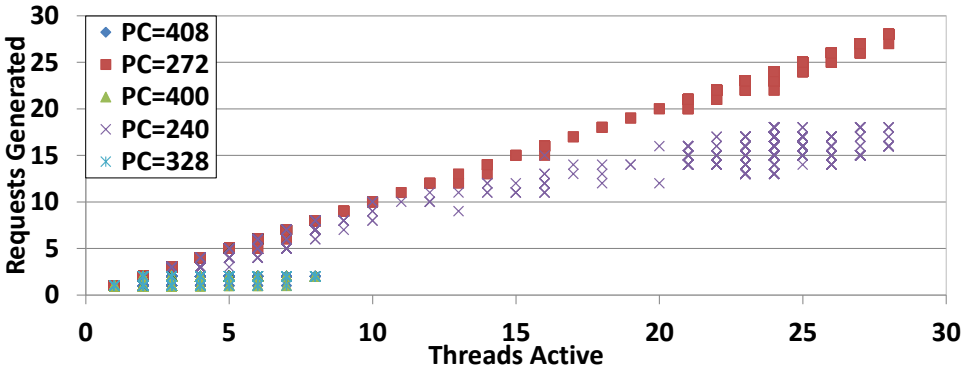


Figure 4.4: Number of threads active and number memory accesses generated for each dynamic load in BFS's loop. Accesses are grouped by PC.

generate one or two accesses no matter how many threads are active are completely converged. Anything in between is somewhat diverged. To understand the relationship between memory divergence and static instructions, consider Figure 4.4. Figure 4.4 plots the number of threads active and accesses generated for every dynamic load instruction in BFS, grouped by the load instruction's PC. This figure illustrates that memory divergence behaviour can be characterized on a per-PC basis. Some PCs are always converged (328, 400 and 408 in Figure 4.4), some are almost always completely diverged (272) and others are consistently somewhat diverged (240). This result is consistent across all the highly cache-sensitive applications we studied. For simplicity, DAWS classifies each static load instruction that is not consistently converged as diverged.

This figure also demonstrates that there is a significant amount of control flow divergence in this application. This control flow divergence makes a solution that statically limits the number of warps when the kernel is launched [137] suboptimal, since it does not adapt to thread activity as the program executes. Some of the static loads in Figure 4.4 never have more than 8 threads active (for example, PC 328). These loads occur inside a branch within the loop and are only generated in BFS when a thread is processing a node with an unexplored edge.

Additionally, all 32 threads are never active in this loop due to branch divergence occurring prior to loop execution. The loop is only executed if a node is

on the program's exploration frontier, which can be relatively sparsely distributed across threads. This illustrates that there is an opportunity to improve the estimated cache footprint for a loop by taking advantage of branch prediction. However, for the cache footprint prediction generated by DAWS, we assume the worst possible case (i.e., all of the loads in the loop get uncovered by all threads active on this loop iteration). Exploring branch prediction is beyond the scope of this work.

Loop Trip Repetition

Multiple static loads within one loop-trip may reference the same cache line. The *Accessed-This-Trip* values in Figure 4.3 demonstrate this can be significant. These loads do not increase the cache footprint because the data they access has already been accounted for by another load. We introduce the concept of a repetition ID to filter them out. All loads predicted to reference the same cache line are assigned the same repetition ID. When predicting the cache footprint of a loop, only one load from each repetition ID is counted. Classification the repetition ID is done either by the compiler (predicting that small offsets from the same pointer are in the same line) or by hardware (described in Section 4.2.2).

4.2 Divergence-Aware Warp Scheduling (DAWS)

The goal of DAWS is to keep data in cache that is reused by warps executing in loops so that accesses from successive loop iterations will hit. DAWS does this by first creating a cache-footprint prediction for each warp. Then, DAWS only allows load instructions to be issued from warps whose aggregate cache footprints are predicted to be captured by the L1D cache.

Figure 4.5 illustrates how DAWS works at a high level. A prediction of the cache footprint for each warp is created. These predictions are summed to create a total cache footprint. At time T_0, all warps have no predicted footprint. Warps that enter loops with locality are assigned a prediction and consume a portion of the estimated available cache. When a warp exits the loop its predicted footprint is cleared. When the addition of a warp's prediction to the total cache footprint exceeds the effective cache size, that warp is prevented from issuing loads. The value of the effective cache size is discussed later in Section 4.2.1. To illustrate

DAWS in operation, consider what happens at each time-step in Figure 4.5. Between time T_0 and T_1, warp 0 enters a loop. From a previous code characterization, DAWS has predicted that this loop has intra-warp locality and one divergent load. Sections 4.2.1 and 4.2.2 present two variations of DAWS that perform this code characterization in different ways. Warp 0's active mask is used to predict that warp 0 will access 32 cache lines (one for each active lane) in this iteration of the loop. The value of the footprint prediction for more complex loops is discussed in detail in Section 4.2.1. Between time T_1 and T_2, warp 1 enters the loop with only 16 active threads and receives a smaller predicted footprint of 16. Between T_2 and T_3, warp 2 reaches the loop. The addition of Warp 2's predicted cache footprint to the current total cache footprint exceeds the effective cache size, therefore warp 2 is prevented from issuing any loads. Between T_3 and T_4, 16 of warp 0's 32 threads have left the loop (causing control flow divergence) which frees some predicted cache capacity, allowing warp 2 to issue loads again.

The DAWS warp throttling mechanism is somewhat similar to the lost locality scoring system presented in CCWS [137], however there are several key differences. In CCWS, scores are assigned based on detected lost locality. Warps losing the most locality are given more exclusive cache access by preventing warps losing the least locality from issuing loads. CCWS is a *reactive* system that has to lose locality before trying to preserve it. DAWS is a *proactive* system that tries to prevent lost locality before it happens. DAWS is also proactive in decreasing the level of thread throttling. As threads within warps progress through a loop a different number of times, the data accessed by their divergent loads is reduced causing DAWS to decrease their predicted cache footprint. DAWS takes this control flow divergence into account immediately and scales up the number of warps allowed to issue load instructions as appropriate. In contrast, CCWS scales back thread throttling by a constant factor each cycle, unless more lost locality is detected. In CCWS, when warps with the most exclusive cache access stop losing locality, their exclusivity is lost and they have to start missing again to get it back. DAWS ensures that all warps in loops with intra-warp locality do not lose their cache exclusivity until they exit the loop.

Figure 4.6 presents the microarchitecture required to implement our two proposed Divergence-Aware Scheduling Techniques. Section 4.2.1 details *Profiled*

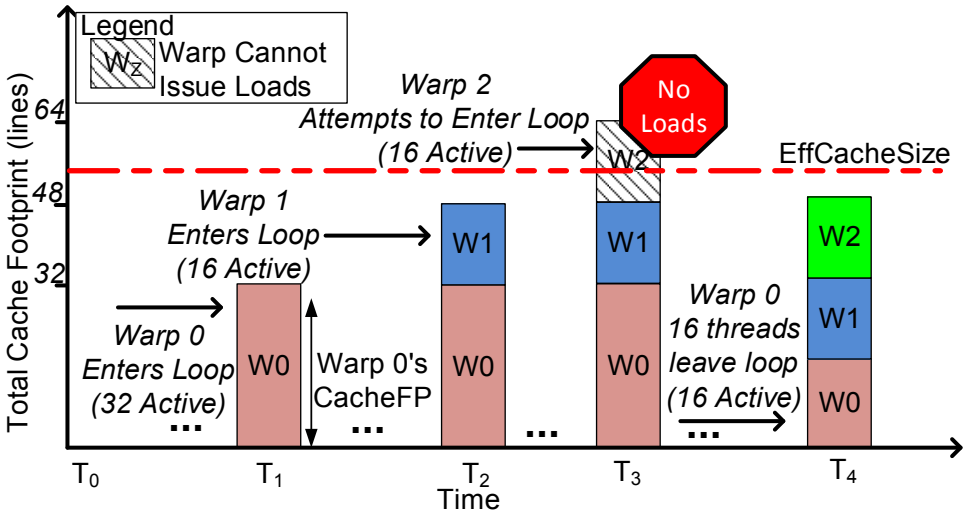


Figure 4.5: High level view of how DAWS's cache footprint prediction mechanism dynamically throttles the number of threads sharing the cache. *CacheFP=Cache Footprint*

Divergence-Aware Warp Scheduling (Profiled-DAWS), which uses off-line profiling to characterize memory divergence and locality. Section 4.2.2 presents *Detected Divergence-Aware Warp Scheduling* (Detected-DAWS), which detects both locality and memory divergence as the program executes. Both techniques make use of feedback from the branch unit (Ⓐ in Figure 4.6) which tells the Warp Issue Arbiter the number of active lanes for any given warp. Detected-DAWS is implemented on top of Profiled-DAWS. In Detected-DAWS, locality and memory divergence information is detected as the program runs based on feedback from the memory system (Ⓑ). This feedback allows Detected-DAWS to classify static load instructions based on dynamic information about how much locality each instruction has and how many memory accesses it generates.

4.2.1 Profiled Divergence-Aware Warp Scheduling (Profiled-DAWS)

Figure 4.6 presents the microarchitecture for both Profiled- and Detected-DAWS. Both versions of DAWS are implemented as an extension to the WIA's baseline warp prioritization logic. The output of the scheduler is a *Can Issue* bit vector

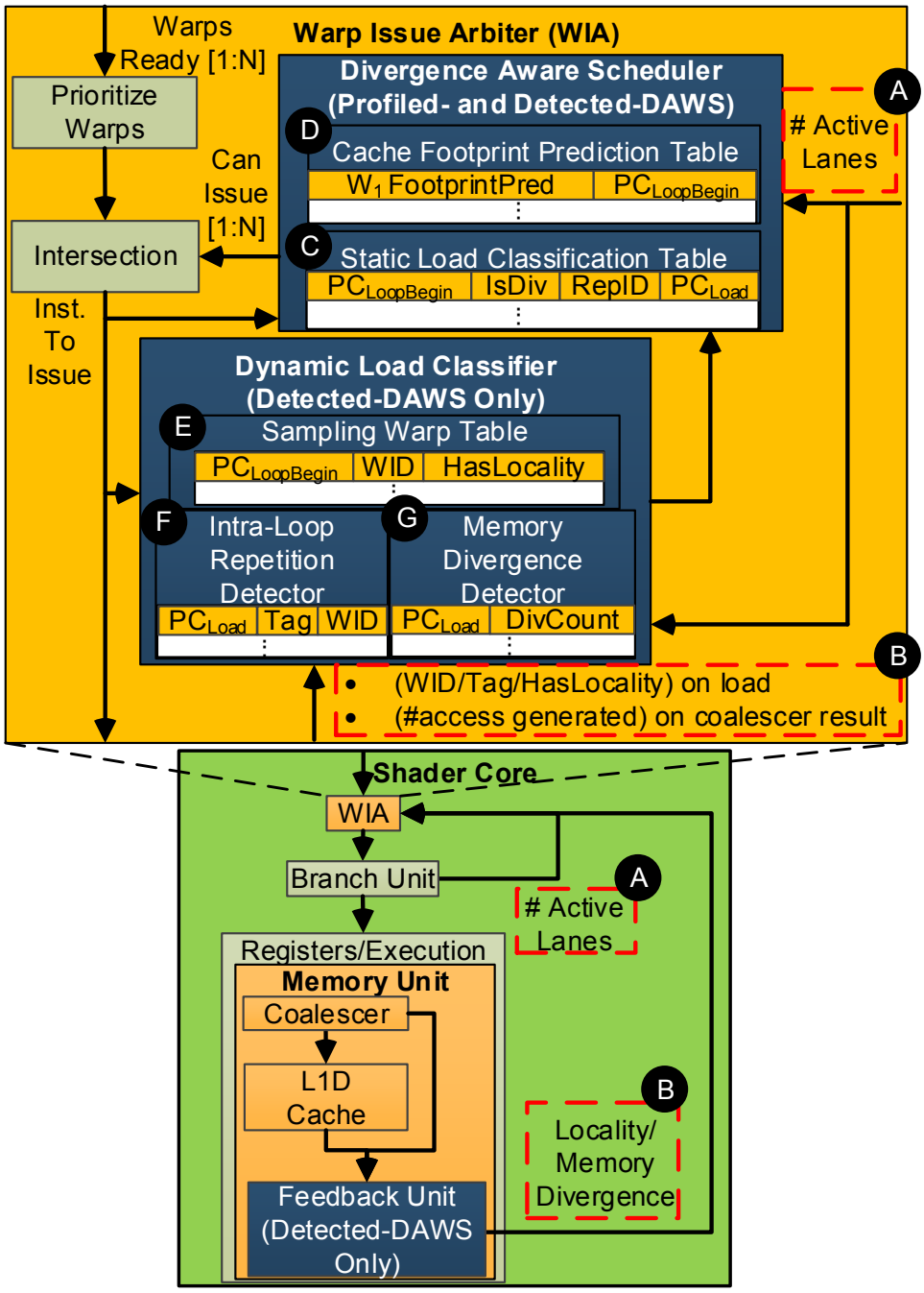


Figure 4.6: Detailed core model used for our DAWS solutions. N is the number of warp issue slots on the core.

that prevents warps from issuing. The task of the scheduler is to determine this bit vector. As described in Section 4.2, this is driven by cache footprint predictions.

To create the cache footprint prediction for each warp, DAWS must classify the behaviour of static load instructions in loops. One method to predict the behaviour of static load instructions is to do a profiling pass of the application. To provide a bound on the potential of an online solution, we propose Profiled-DAWS. We classify each static load instruction using the two criteria presented in Section 4.1.2: (1) Is the load converged or diverged? (2) Does the load contribute to the footprint for this iteration of the loop (i.e., The load's repetition ID)? To collect this information for Profiled-DAWS, we perform an analysis of compiled assembly code and use runtime information gathered from a profiling pass of each application. Determining if a load is converged or diverged is done by profiling all the accesses of each load, similar to the analysis done on BFS in Section 4.1.2. To determine intra-loop repetition we do not use profile information. Instead, we examine the assembly and assume that all loads using the same base address register whose displacement values are within one cache line are repeated in a loop iteration. Profiling similar to the analysis in Section 4.2 is performed to determine which loops in the code have locality.

From a microarchitectural perspective, the classification information for all static load instructions in loops is stored in a static load classification table (Ⓒ). Each entry in the table contains the PC of the first instruction in the loop where the load is located ($PC_{LoopBegin}$), a flag indicating if it is a diverged load (IsDiv) and a repetition ID (RepID) that is used to indicate the intra-loop repetition ID of the load. Although only necessary for Detected-DAWS, the PC of the load instruction PC_{Load} is also stored here. Profiled-DAWS populates this table when a kernel is launched. These values are based on profiling information from previous runs of the kernel. The table is finite in size and can be spilled to memory, however our applications have at most 26 static load instructions within loops.

The cache footprint prediction for each warp is stored in the cache footprint prediction table (Ⓓ). This table has one entry for each warp issue slot on the core. In our baseline this is 32 entries. Each entry of the table contains the value of the predicted footprint (in cache lines) and the PC identifying the loop ($PC_{LoopBegin}$). The scheduler checks instructions as they are issued, looking for loop begin/end

points. To identify the loop bounds, we require that the compiler adds markers to the first and last instruction of each loop. This can be implemented by using two previously unused bits in the opcode (one bit for loop start, one bit for loop end), or by adding an additional instruction to indicate loop start/end. The current CUDA compiler already outputs the loop bounds in the form of comments. We anticipate that our small addition of loop bound markers would have a minor impact. NVIDIA GPUs use a virtual ISA, which has made it easier to modify the hardware ISA in each of the last 3 architecture iterations.

When the scheduler detects that a warp has issued the first instruction of a loop, it uses the number of active lanes in the warp (Ⓐ) to create the warp's prediction for this loop iteration. This value is written to the warp's entry in the cache footprint prediction table. Section 4.2.1 details how the cache footprint prediction is computed. The update logic also writes the PC of the first instruction in the loop to the table ($PC_{LoopBegin}$). When the warp leaves the loop, the prediction table entry for the warp is cleared. To prevent deadlock, predicted footprints are also cleared while a warp waits at a barrier.

To determine the aggregate cache footprint, a prefix sum of each warp's cache footprint is performed, starting with the oldest warps. All of the warps whose prefix sum is less than our calculated effective cache size (defined in Equation 1) are eligible for issuing. Warps whose prefix sum is greater than the effective cache size are prevented from issuing load instructions.

$$EffCacheSize = kAssocFactor \cdot TotalNumLines \tag{4.1}$$

To decide how many cache lines DAWS should assume are available in the L1D cache (i.e., determining our $EffCacheSize$ value), we need to take the associativity of the cache into account. If we had a fully associative cache, we could assume that an LRU replacement policy would allow us to take advantage of every line in the cache. Since the L1D caches we study are not fully associative (our baseline is 8-way) our technique multiplies the number of lines in the cache by the $kAssocFactor$. The value of $kAssocFactor$ is determined experimentally and explored in more detail in section 4.4.2.

If the working set of one warp is predicted to exceed the L1D cache capacity,

then no warps are de-scheduled and scheduling proceeds in an unthrottled fashion inside this loop. Doing no de-scheduling inside loops that load more data than is predicted to fit in cache reverts the system to hiding latency via multithreading again. We did not observe these large predictions in our workloads.

The prediction update logic is run each time the first instruction in a loop is issued. This way the prediction is reflective of threads leaving the loop because of differing loop trip counts across the warp.

Warp-Based Cache Footprint Prediction

This section explains how the number of cache lines accessed for a given warp in a single iteration of a loop with significant intra-warp locality is predicted. In a single threaded system, predicting the number of cache lines accessed in a loop iteration could be achieved by summing all the static load instructions predicted to be issued in the loop, while accounting for repetition caused by multiple static loads accessing the same data. However, to create a prediction of the data accessed by a warp in one loop iteration, both memory and control flow divergence must be taken into account. We first find which loop the warp in question is executing within by looking at the $PC_{LoopBegin}$ for this warp in the prediction table. Next, we query the static load classification table for all the entries with this $PC_{LoopBegin}$ (i.e., entries for all of the loads in this loop). It sums all the entries returned as follows. If the entry indicates that the load is diverged (i.e., the IsDiv bit is set), then this entry contributes as many cache lines as there are active threads. If the entry is converged (and there is more than one thread active), then this entry contributes two cache lines to the prediction. All entries with one active thread contribute one cache line. During the summation, each intra-loop repetition group (identified by RepID) is only counted once. If there are different divergence characteristics within the same repetition ID, then we count it as diverged. In our applications, we did not observe a diverged load accessing data loaded by a converged load (or vice-versa) in the same loop iteration. The result of this summation is written to this warp's entry in the cache footprint prediction table.

Predicted Footprint of Warps Outside Loops

In the previous sections, we only considered de-scheduling warps within loops because this is where the bulk of the application's memory accesses are. However, some applications may load a significant amount of data outside of loops. Figure 4.2 shows that PCs 1568 and 1600 from the GC benchmark both occur outside of the program's loop and access a significant amount of data, which can interfere with the accesses of warps within the loop. For this reason, if there are warps executing inside a loop, warps outside of loops can be de-scheduled. If any of the entries in the cache footprint prediction table is non-zero (i.e., at least one warp is in a loop), loads issued by warps outside of loops have their predictions updated as if they are executing their closest loop. Ideally a warp's closest loop is the next loop they will execute. For our purposes, we define a warp's closest loop as the next loop in program order. Since warps may skip loops, this may not always be the case, but in our applications this approximation is usually true.

Dealing with Inner Loops

DAWS also detects when a warp has entered an inner loop. When a warp issuing a new loop begin instruction already has a $PC_{LoopBegin}$ value in the cache footprint prediction table that is less than the PC of the new instruction, then we assume the warp has entered an inner loop. When this happens, the footprint prediction table entry for the warp is updated normally, giving the warp the prediction of the inner loop. However, when the warp leaves the inner loop, it does not clear either the prediction value or the $PC_{LoopBegin}$. When the outer loop begins its next iteration, it detects it is an outer loop (because the PC entry in the table is greater than the outer loop's beginning PC) and it recomputes the predicted footprint based on the inner loop's loads. This effectively limits the warps that can enter the outermost loops based on the predicted footprint of the innermost loop. We made this design decision because we observed that the majority of data reuse came from the innermost loop and there is significant data reuse between successive runs of the innermost loop. If we do not limit the number of warps entering the outer loop based on the inner loop, then there is the potential for multiple warps to interleave their inner loop runs, which can evict data repeatedly used by the inner loop. This

can be applied to any arbitrary loop depth, but none of our applications had a loop depth greater than two.

4.2.2 Detected Divergence-Aware Warp Scheduling (Detected-DAWS)

Profiled-Divergence-Aware Warp Scheduling (Profiled-DAWS) relies on two key pieces of profile information. First, it requires that loops with intra-warp locality be known in advance of running the kernel. Second, it requires that all the global and local memory loads in those loops are characterized as converged or diverged and that all the intra-loop-trip repetition between those loads is known. *Detected-Divergence-Aware Warp Scheduling* (Detected-DAWS) requires no profile information. The only requirement for Detected-DAWS is that the compiler mark the beginning and ending of the program's loops. Detected-DAWS *detects* both memory divergence and intra-loop-trip repetition at runtime and populates the static load classification table (Ⓒ in Figure 4.6) dynamically using the dynamic load classifier. Detected-DAWS operates by following the execution of a sampling warp through a loop. The first warp with more than two active threads that enters a loop is set as the sampling warp for the loop. The sampling warp id (WID) and ($PC_{LoopBegin}$) for each loop being sampled are stored in the sampling warp table (Ⓑ). When the sampling warp leaves the loop, the next warp to enter with two or more active threads becomes the new sampling warp for the loop. At any given time, multiple loops can be sampled but only one warp can sample each loop. The sampling warp table also stores a locality counter (HasLocality) that is used to indicate if loads for this loop should be entered into the static load classification table. Like the static load classification table, the sampling warp table is finite in size. Each of our applications has at most five loops. The dynamic load classifier interprets memory system feedback about loads issued from sampling warps.

It is worth noting that, other than the addition of PC_{Load} to each static load classification table entry, nothing about the divergence aware scheduler used in Profiled-DAWS changes. The scheduler just operates with incomplete information about the loops until the dynamic load classifier has filled the static load classification table.

The following subsections describe how the dynamic load classifier uses the

memory system feedback to populate the static load classification table.

Finding Loops with Locality

This section describes how Detected-DAWS determines which loops have intra-warp locality. Memory system feedback (Ⓑ) informs the scheduler when loops have intra-warp locality. The feedback unit sends signals to the dynamic load classifier on each load issued signifying if the load has intra-warp locality. The feedback unit reports both captured and lost intra-warp locality. To report this locality, cache lines in the L1D cache are appended with the WID of the instruction that initially requested them. Lost intra-warp locality is detected through the warp ID filtered victim tags mechanism described in CCWS [137]. Hits in the L1D cache on data that one warp loads and re-references are reported as captured intra-warp locality. If a load has neither lost nor captured intra-warp locality then the feedback unit informs the dynamic load classifier that the load has no intra-warp locality. Whenever the classifier is informed that a load from a sampling warp has taken place, it modifies that loop's locality counter in the sampling warp table. If the load was an instance of intra-warp locality, the counter is incremented otherwise the counter is decremented. DAWS creates cache footprint predictions for loops with positive locality counters.

Dynamically Classifying Static Loads in Hardware

Once a loop is marked as having intra-warp locality, the dynamic load classifier starts generating static load classification table entries for the loop. To avoid having more than one entry for each static load in the static load classification table, Detected-DAWS requires the PC of the load be stored (PC_{Load}). Before inserting a new entry into the table, the dynamic load classifier must ensure that this PC_{Load} does not already exist in the table. If the entry does exist, the dynamic classifier updates the existing entry. The classifier consists of two components, an intra-loop repetition detector (Ⓕ) and a memory divergence detector (Ⓖ).

Memory Divergence Detector: The memory divergence detector is used to classify static load instructions as convergent or divergent. It receives information about load coalescing from the memory feedback unit. After an instruction

passes through the memory coalescer, the resulting number of memory accesses is sent to the dynamic load classifier. The classifier reads this value in combination with the active thread count of the load instruction. If more than two threads in the instruction were active when the load was issued, the number of accesses generated is tested. If the number of accesses generated is greater than two, the divergence counter for this PC is incremented. If two or less accesses are generated, the counter is decremented. If the divergence counter is greater than one, this load is considered diverged, otherwise it is considered converged.

Intra-Loop Repetition Detector: The *Intra-Loop Repetition Detector* (ILRD) dynamically determines which static load instructions access the same cache line in the same loop iteration. It is responsible for populating the RepID field of the static load classification table. Each entry in the detector contains a tag, PC_{Load} and WID. On each load executed by a sampling warp, the ILRD is probed based on the tag of the load. If the tag is not found, the tag and PC/warp id for the instruction that issued the load are written to the table. If the tag is found, then both the PC issuing the new load and the PC in the table are marked as intra-loop repeated and assigned the same repetition ID. When the sampling warp branches back to the start of the loop, all the values in the ILRD for this warp are cleared. Without the WID, multiple loops could not be characterized concurrently because the sampling warp for one loop could clear the entries for another. The ILRD is modeled as a set associative tag array, with an LRU replacement policy.

4.3 Experimental Methodology

We model Profiled-DAWS and Detected-DAWS as described in Section 4.2.1 and 4.2.2 in GPGPU-Sim [19] (version 3.1.0) using the configuration in Table 4.1. Loop begin and end points are inserted manually in the assembly.

The highly cache-sensitive and cache-insensitive workloads we study are listed in Table 4.2, four of which come from the CCWS infrastructure available online [136]. The SPMV-Scalar benchmark comes from the SHOC benchmark suite [44].

Our benchmarks are run to completion which takes between 14 million and 1 billion instructions.

Table 4.1: Divergence-aware warp scheduling GPGPU-Sim Configuration

# Compute Units	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 128B line, 8-way assoc. LRU
L2 Unified Cache	128k/Memory Channel, 128B line, 8-way assoc. LRU
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	out of order (FR-FCFS)
Branch Divergence Method	PDOM [55]
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

4.4 Experimental Results

This section is organized as follows, Section 4.4.1 examines the performance of our workloads using Profiled-DAWS, Detected-DAWS and other warp schedulers. The remainder of this section is devoted to analyzing varying aspects of our design and exploring its sensitivity.

4.4.1 Performance

All data was collected using GPGPU-Sim running the following scheduling mechanisms:

GTO A greedy-then-oldest scheduler [137]. GTO runs a single warp until it stalls then picks the oldest ready warp. Warp age is determined by the time the

Table 4.2: GPU Compute Benchmarks (CUDA and OpenCL)

Highly Cache Sensitive			
Name	**Abbr.**	**Name**	**Abbr.**
BFS Graph Traversal [36]	BFS	Kmeans [36]	KMN
Memcached [62]	MEMC	Garbage Collection [21, 154]	GC
Sparse Matrix Vector Multiply (Scalar) [44]	SPMV-Scalar		
Cache Insensitive (CI)			
Name	**Abbr.**	**Name**	**Abbr.**
Needleman-Wunsch [36]	NDL	Back Propagation [19]	BACKP
Hot Spot [36]	HOTSP	LU Decomposition [36]	LUD
Speckle Red. Anisotropic Diff. [36]	SRAD		

Table 4.3: Configurations for Best-SWL and CCWS.

Best-SWL		**CCWS Config**	
Benchmark	**Warps Actively Scheduled**	**Name**	**Value**
BFS	5	$K_{THROTTLE}$	8
MEMC	7	Victim Tag Array	8-way
SPMV-Scalar	2		(512 total entries)
GC	2		16 entries per warp
KMN	4	Warp Base Score	100
All Others	32		

warp is assigned to the shader core. For warps that are assigned to a core at the same time (i.e., they are in the same thread block), warps with the smallest scalar threads IDs are prioritized. Other simple schedulers (such as oldest-first and loose-round-robin) were implemented and GTO scheduling performed the best.

Best-SWL Static Warp Limiting as described in [137]. Warp limitation values from 1 to 32 are attempted and the highest performing case is selected. A GTO policy is used to select between warps. The warp limiting value used for each application is shown in Table 4.3.

Table 4.4: Configuration parameters used for DAWS

DAWS Config	
ILRD size	64 entries per core, 8-way set associative
Associativity Factor	0.6
Victim Tag Array	Same as CCWS in Table 4.3

CCWS Cache-Conscious Warp Scheduling as described in [137]. The configuration parameters presented in Table 4.3 are used.

Profiled-DAWS Profiled Divergence-Aware Warp Scheduling as described in Section 4.2.1. Loop profiles were generated manually based on PC statistics collected in sampling application runs. The applications were profiled with input data different from the evaluation data. GTO prioritization logic is used.

Detected-DAWS Detected Divergence-Aware Warp Scheduling as described in Section 4.2.2, with the configuration used in Table 4.4 GTO prioritization logic is used.

Figure 4.7 presents the *Instructions Per Cycle* (IPC) of our evaluated schedulers, normalized to CCWS. It illustrates that Profiled-DAWS and Detected-DAWS improve performance by a harmonic mean 25% and 26% respectively over CCWS on our highly cache-sensitive applications. In addition, they do not cause any performance degradation in the cache-insensitive applications. The cache-insensitive applications have no loops with detected intra-warp locality. Profiled-DAWS and Detected-DAWS are able to outperform Best-SWL by a harmonic mean 3% and 5% respectively. The performance of Profiled-DAWS and Detected-DAWS against Best-SWL is highly application dependent. Detected-DAWS is able to outperform Best-SWL on BFS by 20%, however it sees a 4% slowdown on SPMV-Scalar.

Figure 4.8 can help explain the skewed performance results against Best-SWL. It presents the control flow divergence in each of our highly cache-sensitive applications. It shows warp lane activity for all issued instructions. Bars at the bottom of each stack indicate less control flow divergence, as more lanes are active on

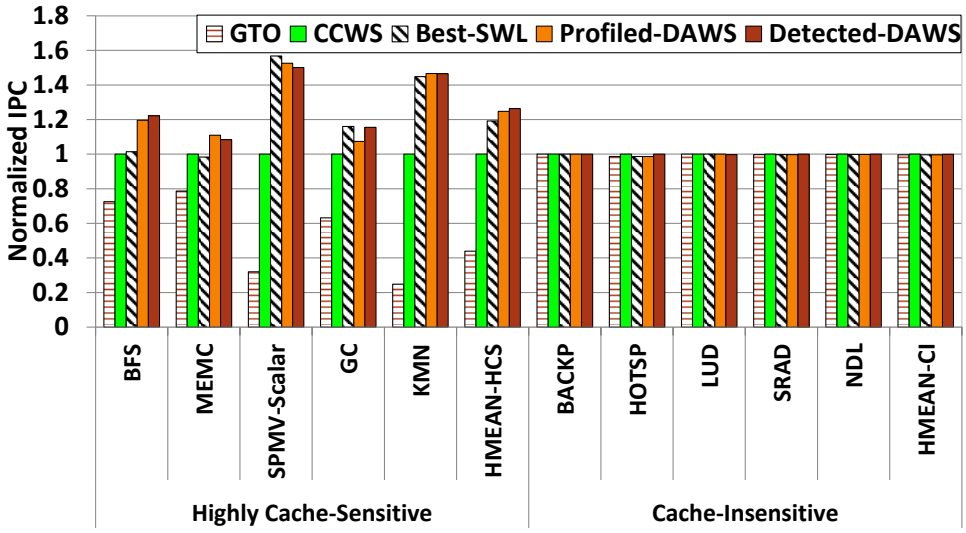


Figure 4.7: Performance of various scheduling techniques, normalized to CCWS.

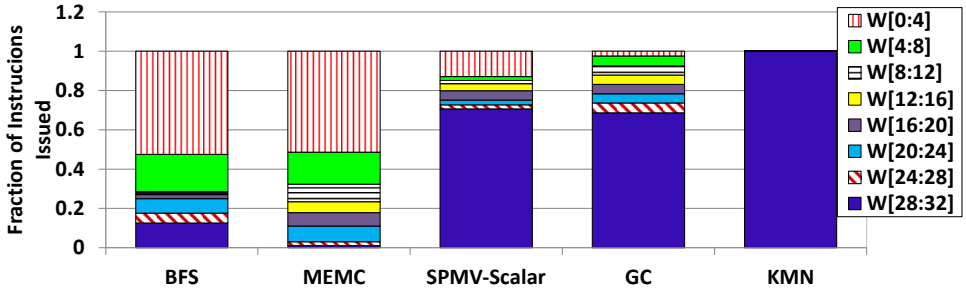


Figure 4.8: Breakdown of warp lane activity. Breakdown is presented as a fraction of total instructions executed. W[0:4] means 0 to 4 of an instruction's 32 lanes are active.

each issued instruction. The two applications where DAWS improves performance relative to Best-SWL (BFS and MEMC) also have the most control flow divergence. The performance of Best-SWL is hampered most when there is significant control flow divergence. Selecting the same limiting value for every core over the course of the entire kernel is not optimal. This divergence occurs because of both loop-trip count variation across a warp and a discrepancy in the level of control flow divergence on each shader core. We also evaluated Detected-DAWS without

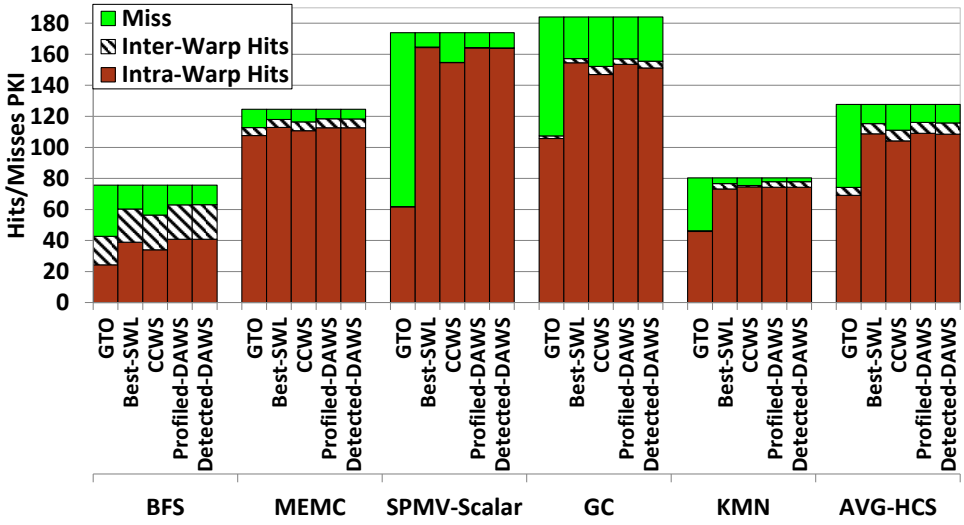


Figure 4.9: L1D intra-warp hits, inter-warp hits and misses *per thousand instructions* (PKI) of various schedulers.

control flow awareness by assuming all lanes were active on every loop iteration. Removing control flow awareness results in a 43% and 91% slowdown for BFS and MEMC respectively versus Detected-DAWS. Other applications showed no significant performance change.

Figure 4.9 presents the L1D cache misses, intra-warp hits and inter-warp hits *per thousand instructions* (PKI) for our highly cache-sensitive benchmarks. It demonstrates that Profiled-DAWS and Detected-DAWS result in fewer cache misses than CCWS, which can account for a portion of the overall speedup. Since Profiled-DAWS and Detected-DAWS are able to predict the cache footprint of warps *before* they lose locality based on profile information created by other warps they can apply thread limiting before CCWS, removing the unnecessary cache misses. In addition, Profiled-DAWS and Detected-DAWS do not de-prioritize warps once they have entered a loop with locality. The scheduling point system in CCWS can potentially de-prioritize warps hitting often in cache when they stop producing accesses that hurt locality. We performed experiments and found that 46% of CCWS's lost locality occurs after a warp has been de-scheduled while in a loop. CCWS prioritizes warps based solely on detected lost locality. Warps may be de-scheduled in-

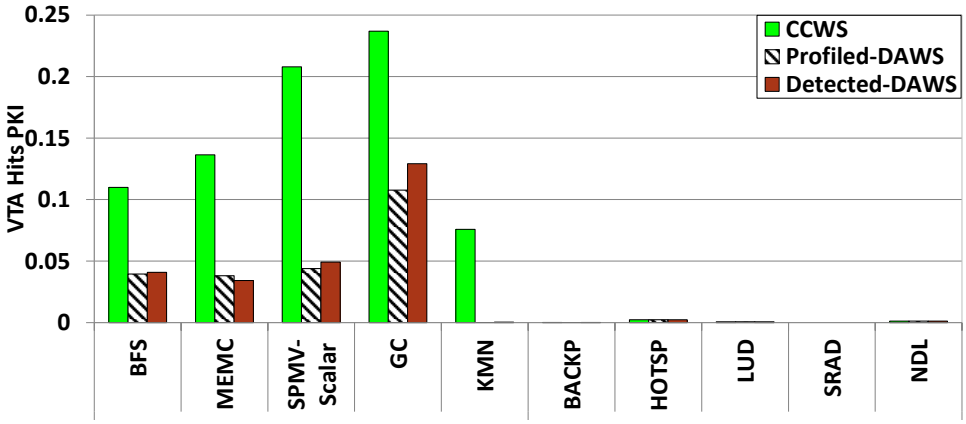


Figure 4.10: Victim tag array hits per thousand instructions (PKI) (indicating lost intra-warp locality).

side a high-locality loop before they complete the loop, resulting in the eviction of their reused data. Once loops are properly classified, this type of lost locality never occurs using DAWS. DAWS ensures that once a warp enters a high-locality loop, it is not de-scheduled until the warp exits the loop or encounters a barrier. None of our highly cache-sensitive applications have barrier instructions. Figure 4.9 also demonstrates that the cache miss rate in Profiled-DAWS and Detected-DAWS is similar to that of Best-SWL. This suggests that the performance increase seen by Profiled-DAWS and Detected-DAWS over Best-SWL comes from decreasing the level of warp limiting when the aggregate footprint of threads scheduled can still be contained by the cache.

Figure 4.10 plots victim tag array hits, which indicate a loss of intra-warp locality. There is no victim tag array required to implement Profiled-DAWS, but for the purposes of this data, one is added. This figure illustrates that there is a large reduction in detected instances of lost locality when using the DAWS solutions. In addition, this figure shows a slight increase in detected lost locality in Detected-DAWS versus Profiled-DAWS. This is because Detected-DAWS requires some time to classify static load instructions before appropriate limiting is able to take effect.

Figure 4.11 breaks down core activity into cycles where an instruction issues, cycles where there are no instructions to issue (i.e., no warps are ready to be is-

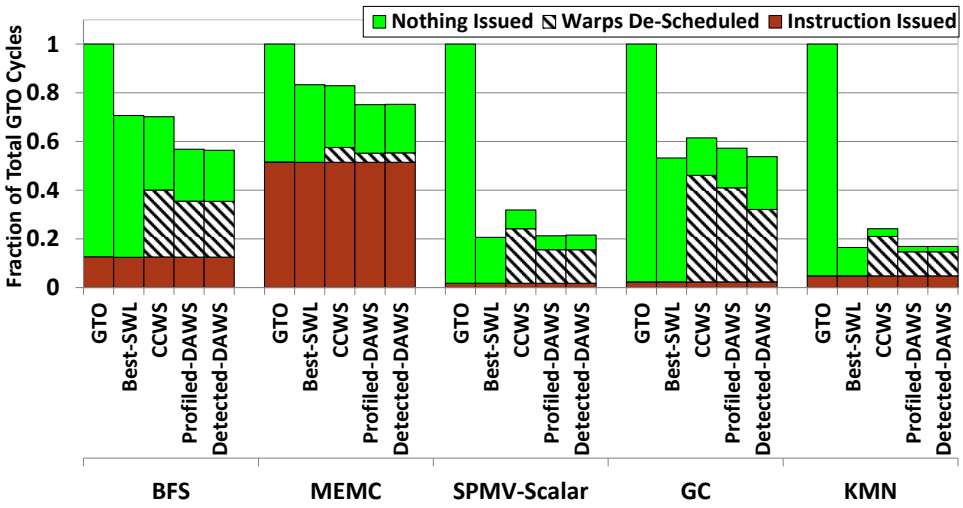


Figure 4.11: Breakdown of core activity normalized to GTO's total cycles for each application.

sued) and cycles where an instruction could have issued, if its warp had not been de-scheduled by the scheduling system. This is aggregate information collected over all the shader cores. This figure demonstrates that both Profiled-DAWS and Detected-DAWS reduce the number of cycles spent de-scheduling warps versus CCWS.

4.4.2 Determining the Associativity Factor

Figure 4.12 plots the performance change of our highly cache-sensitive applications as the *kAssocFactor* is swept. All the applications consistently peak at 0.6, except BFS which peaks shows a small performance gain at 0.7 versus 0.6. This is consistent with the assertion that *kAssocFactor* should be mostly independent of the application. The slight performance improvement for BFS at 0.7 can be explained by the fact that it has branches inside its loop that cause some of the loads to be infrequently uncovered, as discussed in Section 4.1.2. Since DAWS overestimates by assuming all the loads in the loop are uncovered, a larger *kAssocFactor* makes up for this per-warp overestimation by raising the effective cache size cutoff.

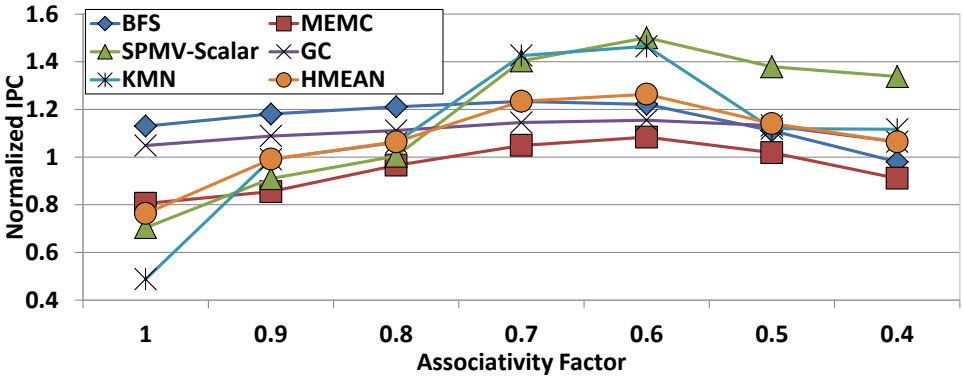


Figure 4.12: Detected-DAWS performance as the cache associativity factor is swept. Normalized to CCWS.

4.4.3 Area Estimation

The tables added for Profiled-DAWS (i.e., the cache footprint prediction table and the static load classification table) are each modeled with only 32 entries and are negligible in size. The additional area added by Detected-DAWS comes from a victim tag array, the other tables are 64 entries or less. A victim tag array is also used in CCWS, so there is negligible area difference between Detected-DAWS and CCWS. However, compared to Best-SWL or GTO schedulers both CCWS and Detected-DAWS have a CACTI [171] estimated area overhead of 0.17% which is discussed in more detail in [137].

4.4.4 Dynamic Energy Estimation

We investigated two energy models for GPUs to evaluate the effect DAWS has on energy, GPUSimPow [108] and GPUWattch [102]. Due to the recent release date of these simulators, we were unable to fully integrate our solution into their framework. However, we extracted the nJ per operation constants used in GPUWattch for DRAM reads, DRAM pre-charges and L2/L1D cache hits/misses, which are the metrics that dominate the overall energy consumed in the highly cache-sensitive applications and are the key metrics effected by DAWS. This calculation shows that DAWS consumes 2.4× less and 23% less dynamic energy in the memory system than GTO and CCWS respectively. This power reduction is primarily due to

an increase in the number of L1D cache hits, reducing power consumed in the memory system. This estimate does not include the dynamic energy required for Detected-DAWS or CCWS tables, victim tag array or logic. We anticipate this energy will be small in comparison to the energy used in the memory system.

4.5 Summary

This work quantifies the relationship between memory divergence, branch divergence and locality on a set of workloads commonly found in server computing. We demonstrate that divergence and locality characteristics of static load instructions can be accurately predicted based on previous behaviour. Divergence-Aware Warp Scheduling uses this predicted code behaviour in combination with live thread activity information to make more locality-aware scheduling decisions. Divergence-Aware Warp Scheduling is a novel technique that proactively uses predictions to prevent cache thrashing before it occurs and aggressively increases cache sharing between warps as their thread activity decreases.

Our simulated evaluations show that our fully dynamic technique (Detected-DAWS) results in a harmonic mean 26% performance improvement over Cache Conscious Warp Scheduling [137] and 5% improvement over the profile-based Best-SWL [137]. Performance relative to Best-SWL is improved as much as 20% when workloads have significant control flow divergence.

Our work increases the efficiency of several highly divergent, cache-sensitive workloads on a massively parallel accelerator. Our programmability case study demonstrates that Divergence-Aware Warp Scheduling can allow programmers to write simpler code without suffering a significant performance loss by effectively shifting the burden of locality management from software to hardware.

Chapter 5

A Programmability Case Study

This chapter presents a case study using two implementations of Sparse Matrix Vector Multiply (SPMV) from the SHOC benchmark suite [44] [1]. This case study is chosen because it is a real example of code that has been ported to the GPU then optimized. Example 5.1 presents SPMV-Scalar which is written such that each scalar thread processes one row of the sparse matrix. This is similar to how the algorithm might be implemented on a multi-threaded CPU. The bold code in SPMV-Scalar highlights its divergence issues. Example 5.2 shows SPMV-Vector which has been optimized for performance on the GPU. Both pieces of code generate the same result and employ the same data structure. The bold code in SPMV-Vector highlights the added complexity introduced by GPU-specific optimizations.

One goal of this work is to enable less optimized code such as Example 5.1 to achieve performance similar to the optimized code in Example 5.2. In SPMV-Scalar, the accesses to $cols[j]$ and $val[j]$ will have significant memory divergence and the data-dependent loop bounds will create branch divergence. Like the code in Figure 4.1, SPMV-Scalar has spatial locality within each thread since j is incremented by one each iteration. Divergence-Aware Warp Scheduling seeks to capture this locality.

In the SPMV-Vector version each warp processes one row of the sparse matrix. Restructuring the code in this way removes much of the memory divergence

[1] For brevity, some keywords in the original version of Examples 5.1 and 5.2 were removed. All of our experiments are run without modifying the original kernel code.

Figure 5.1: Highly divergent SPMV-Scalar kernel

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * blockDim.x
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Loads
            int col = cols[j];
            t += val[j] * vecTexReader(col);

        }
        out[myRow] = t;
    }
}
```

present in the scalar version since the accesses to $cols[j]$ and $val[j]$ will have spatial locality across each SIMT instruction. However, this version of the code forces the programmer to reason about warp length, the size of on-chip shared memory, and it requires a parallel reduction of partial sums to be performed for each warp. In addition to writing and debugging the additional code required for SPMV-Vector, the programmer must tune thread block sizes based on which machine the code is run on. Even if the programmer performed all these optimizations correctly, there is no guarantee that SPMV-Vector will outperform SPMV-Scalar since the shape and size of the input matrix may render the optimizations ineffective. Previous work has shown that sparse matrices with less non-zero elements per row than the GPU's warp width do not take advantage of the potential increase in coalesced accesses offered by SPMV-Vector [27].

This reliance on per-machine tuning and the unpredictability of manual optimization techniques can make programming GPUs difficult. In Section 5.1 we

Figure 5.2: GPU-optimized SPMV-Vector kernel

```
__global__ void
spmv_csr_vector_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float * out)
{
    int t = threadIdx.x;
    int id = t & (warpSize-1);
    int warpsPerBlock = blockDim.x / warpSize;
    int myRow = (blockIdx.x * warpsPerBlock)
              + (t / warpSize);
    texReader vecTexReader;

    __shared__ volatile
        float partialSums[BLOCK_SIZE];

    if (myRow < dim)
    {
        int warpStart = rowDelimiters[myRow];
        int warpEnd = rowDelimiters[myRow+1];
        float mySum = 0;
        for (int j = warpStart + id;
             j < warpEnd; j += warpSize)
        {
            int col = cols[j];
            mySum += val[j] * vecTexReader(col);
        }
        partialSums[t] = mySum;

        // Reduce partial sums
        if (id < 16)
            partialSums[t] += partialSums[t+16];
        if (id <  8)
            partialSums[t] += partialSums[t+ 8];
        if (id <  4)
            partialSums[t] += partialSums[t+ 4];
        if (id <  2)
            partialSums[t] += partialSums[t+ 2];
        if (id <  1)
            partialSums[t] += partialSums[t+ 1];

        // Write result
        if (id == 0)
        {
            out[myRow] = partialSums[t];
        }
    }
}
```

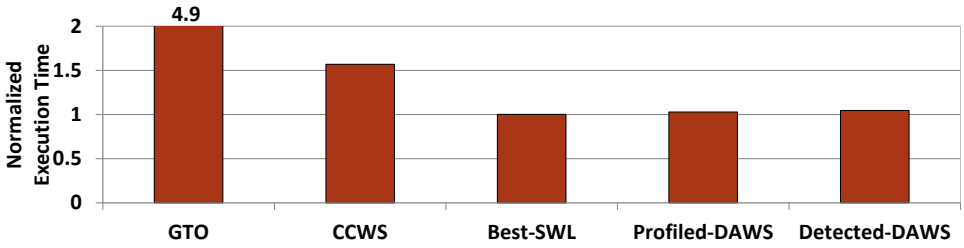


Figure 5.3: Execution time (lower values are faster) of SPMV-Scalar using various warp schedulers normalized to the best performing scheduler from SPMV-Vector.

demonstrate that Divergence-Aware Warp Scheduling allows the programmer to write the simpler, more portable SPMV-Scalar while still capturing almost all of the performance benefit of SPMV-Vector.

This case study should not be construed to suggest that Divergence-Aware Warp Scheduling can replicate the performance of any hand tuned optimization or generally solve the performance issues surrounding divergence on GPUs. The study is presented as one real world example of optimized GPU code to demonstrate how intelligent warp scheduling can capture almost as much locality as this particular hand tuned implementation.

5.1 Case Study Results

In this section we examine the results of our case study. To run these experiments, the size of on-chip scratchpad memory was increased to 48k, while leaving the L1D cache size constant. This was done so that shared memory usage would not be a limiting factor for SPMV-Vector and our results would not be biased towards SPMV-Scalar. The input sparse matrix is randomly generated by the SHOC framework. The matrix has 8k rows with an average of 82 non-zero elements per row. Figure 5.3 presents the execution time of SPMV-Scalar from Example 5.1 using our evaluated schedulers normalized to the GPU-optimized SPMV-Vector from Example 5.2 using its best performing scheduler. Like the other cache-insensitive applications we studied, the scheduler choice for SPMV-Vector makes little difference. There is $< 1\%$ performance variation between all the schedulers we evaluated. This figure demonstrates that SPMV-Scalar suffers significant performance loss

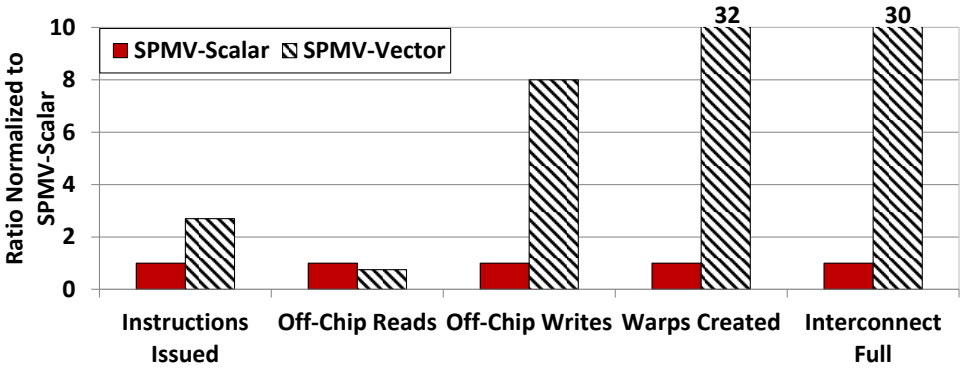


Figure 5.4: Ratio of various metrics for SPMV-Scalar using Detected-DAWS vs. SPMV-Vector using its best performing scheduler. *Interconnect Full*=instances where cores cannot access the interconnect due to contention.

when using previously proposed schedulers like GTO and CCWS. Best-SWL captures almost all the performance of SPMV-Vector, but requires the user to profile the application/input data combination with different limiting values before running. Detected-DAWS does not requiring any profiling information or additional programmer input and its execution time is within 4% of SPMV-Vector's.

Figure 5.4 compares several properties of SPMV-Scalar using Detected-DAWS to SPMV-Vector using its best performing scheduler. This graph shows that SPMV-Scalar has some advantages over SPMV-Vector, if Detected-DAWS is used. SPMV-Scalar executes 2.8x less dynamic instructions, decreasing the amount of dynamic power consumed on each core. SPMV-Scalar also requires 32x fewer warps, decreasing shader initialization overhead (which is not modeled in GPGPU-Sim) and the number of scheduling entities the GPU must deal with.

Since SPMV-Vector and SPMV-Scalar both perform the same computation on the same input, they fundamentally read and write the same data to and from memory. However, cache system performance and memory coalescing result in a discrepancy in the amount of off-chip traffic generated by each workload. Reads in SPMV-Vector are coalesced since lanes in each warp access consecutive values. However, since DAWS captures much of SPMV-Scalar's spatial locality in the L1D cache, there is only a 25% increase in read traffic. As a reference point,

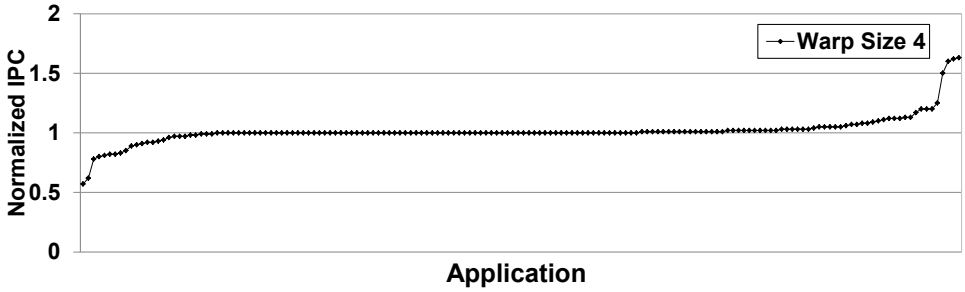
SPMV-Scalar using GTO produces $> 15\times$ more reads than SPMV-Vector. In addition, off-chip writes using SPMV-Vector are increased 8 fold. This happens because SPMV-Scalar is able to coalesce writes to the output vector since each warp attempts to write multiple output values in one SIMT instruction. SPMV-Vector must generate one write request for each row of the matrix and since the L1D caches evict global data on writes, all of these writes go to memory. The last metric compared indicates that contention for the interconnect is greatly increased using SPMV-Vector.

Chapter 6

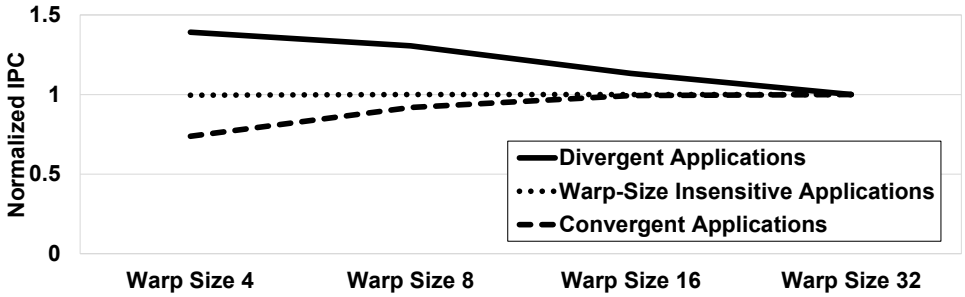
A Variable Warp-Size Architecture

This chapter studies the effect of warp sizing and scheduling on performance and efficiency in GPUs. We propose Variable Warp Sizing (VWS) which improves the performance of divergent applications by using a small base warp size in the presence of control flow and memory divergence. When appropriate, our proposed technique groups sets of these smaller warps together by ganging their execution in the warp scheduler, improving performance and energy efficiency for regular applications. Warp ganging is necessary to prevent performance degradation on regular workloads due to memory convergence slip, which results from the inability of smaller warps to exploit the same intra-warp memory locality as larger warps. This paper explores the effect of warp sizing on control flow divergence, memory divergence, and locality. For an estimated 5% area cost, our ganged scheduling microarchitecture results in a simulated 35% performance improvement on divergent workloads by allowing smaller groups of threads to proceed independently, and eliminates the performance degradation due to memory convergence slip that is observed when convergent applications are executed with smaller warp sizes.

Figure 6.1 plots the *Instructions Per Cycle* (IPC) resulting from shrinking warp size from 32 threads to 4 threads while keeping the machine's total thread-issue throughput and memory bandwidth constant. Figure 6.1a shows the effect of shrinking the warp size on a large suite of real world applications, while Figure 6.1b



(a) Performance of 165 real world applications using a warp size of 4, normalized to a warp size of 32.



(b) Performance versus warp size using a representative subset of applications presented in 6.1a. These applications are described in more detail in Section 6.3.

Figure 6.1: A survey of performance versus warp size.

plots the harmonic mean performance of 15 applications which are selected to represent the 3 classes of workloads we study throughout this paper. We classify a workload as being *divergent* when performance increases as the warp size decreases, *convergent* when performance decreases as the warp size decreases, and *warp-size insensitive* when performance is independent of warp size. Figure 6.1 demonstrates that application performance is not universally improved when the warp size is decreased. This data indicates that imposing a constant machine-dependent warp size for the varied workloads running on GPUs can degrade performance on divergent applications, convergent applications, or both.

A large set of existing, highly regular GPU applications do not see any performance improvement at a smaller warp size. However, the divergent applications which do see a performance improvement represent a class of workloads that are important for future GPUs. Prior work such as [31, 32, 113, 118] has shown great

potential for increasing the performance and energy efficiency of these types of workloads by accelerating them on a GPU. These applications include future rendering algorithms such as raytracing, molecular dynamics simulations, advanced game physics simulations, and graph processing algorithms among many others. The goal of our proposed architecture is to evolve GPUs into a more approachable target for these parallel, but irregular, applications while maintaining the efficiency advantages of GPUs for existing codes.

Figure 6.2 plots the performance and resulting SIMT lane utilization of different warp sizes for each of the applications we study. Control-divergent applications have a low utilization rate at a warp size of 32 and see utilization increase as the warp size is decreased. These workloads are able to take advantage of executing different control flow paths simultaneously by using a smaller warp size. Convergent applications have a high lane utilization rate at a warp size of 32 and see their utilization decrease as the warp size is reduced. This reduction in utilization occurs because of increased pressure on the memory system caused by destroying horizontal locality across a larger warp. Horizontal locality occurs when threads within a warp or thread block access similar memory locations. Modern GPUs coalesce memory requests from the same warp instruction that access the same cache line. By allowing smaller groups of threads to proceed at different rates, the locality that existed across the same static instruction is spread over multiple cycles, causing additional contention for memory resources. We call this effect *memory convergence slip*.

In addition to the performance benefit convergent applications experience with larger warps, convergent and warp-size insensitive applications gain energy efficiency from executing with larger warps. A larger warp size amortizes the energy consumed by fetch, decode, and warp scheduling across more threads. When there is no performance benefit to executing with smaller warps, the most energy-efficient solution is to execute with as large a warp size as possible.

Our paper first examines the effect of providing a variable warp size, which can be statically adjusted to meet the performance and energy efficiency demands of the workload. We then propose Variable Warp Sizing, which gangs groups of small warps together to create a wider warp and dynamically adjusts the size of each gang running in the machine based on the observed divergence characteristics

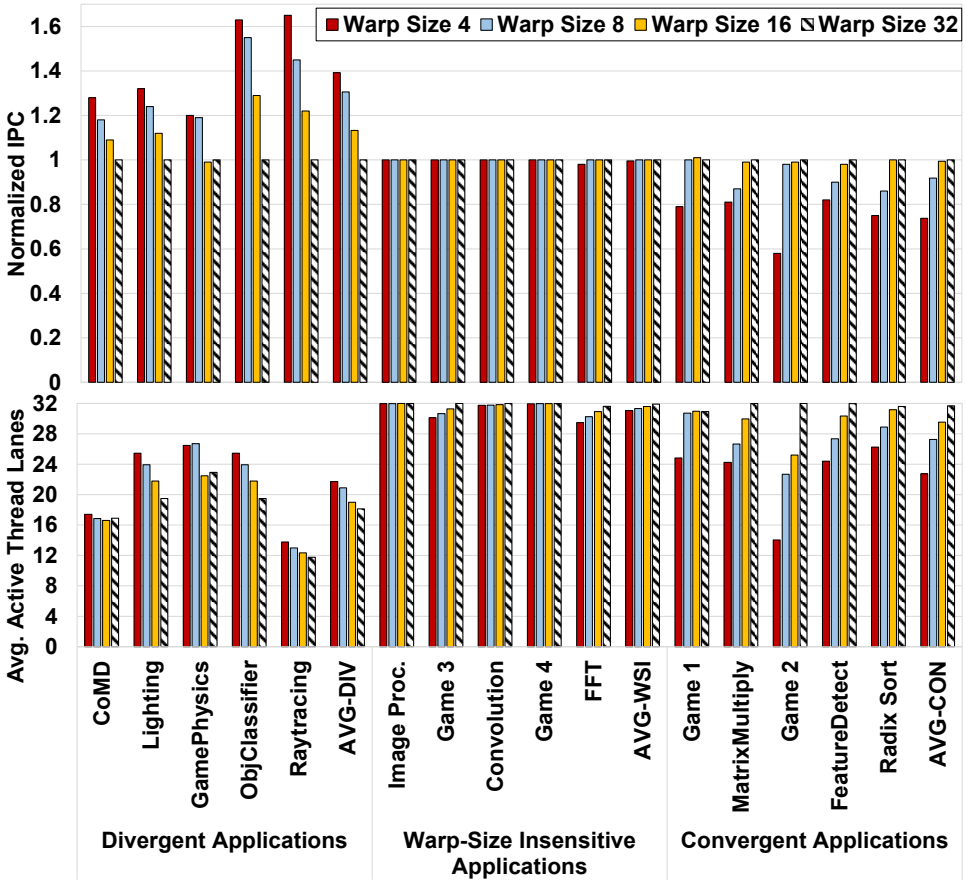


Figure 6.2: Normalized IPC (top) and the average number of active thread lanes on cycles when an instruction is issued (bottom). All configurations can issue 32 thread instructions per cycle.

of the workload.

Prior work such as [30, 48, 53, 55, 115, 123, 132–134] proposes various techniques to improve *Single Instruction Multiple Data* (SIMD) efficiency or increase thread level parallelism for divergent applications on GPUs. However, the use of small warps is the only way to improve both SIMD efficiency and thread level parallelism in divergent code. These prior works focus on repacking, splitting, and scheduling warps under the constraint of a fixed-size warp. Our work approaches the problem from the other direction. We simplify the acceleration of divergent workloads by starting with a small warp size and propose a straightforward gang-

ing architecture to regain the efficiencies of a larger warp. Prior work can improve the performance of divergent applications when the total number of unique control paths is limited and the number of threads traversing each respective path is large. Starting with smaller warps allows our microarchitecture to natively execute many more concurrent control flow paths, removing this restriction. Section 6.5 presents a more detailed quantitative and qualitative comparison to prior work.

6.1 Trade-offs of Warp Sizing

This section details the effect of warp size on both the memory system and SM front-end. This data motivates an architecture that is able to dynamically vary warp size.

6.1.1 Warp Size and Memory Locality

Figure 6.3 shows the effect warp size has on L1 data cache locality in terms of hits, misses, and *Miss Status Holding Register* (MSHR) merges *Per Thousand Instructions* (PKI) for the applications we study. As the warp size is decreased, some applications see an increase in the number of L1 data cache accesses. This phenomenon occurs when memory accesses that were coalesced using a larger warp size become distributed over multiple cycles when smaller warps are used, an effect we term *memory convergence slip*.

In the divergent applications, memory convergence slip does not significantly degrade performance for two reasons. First, an application that is control flow diverged has less opportunity for converged accesses because fewer threads are participating in each memory instruction. Second, even when convergence slip occurs on a divergent application, as it does in CoMD, ObjClassifier, and Raytracing, it also often results in more cache hits, mitigating the effect on performance. While the control-divergent Raytracing application also sees an increase in misses, the performance cost of these misses is offset by the increased lane utilization observed with smaller warps.

In the convergent applications, memory convergence slip has a greater effect on performance. All of these applications see both an increase in the total number of memory accesses and cache misses at smaller warp sizes. Radix Sort and Game

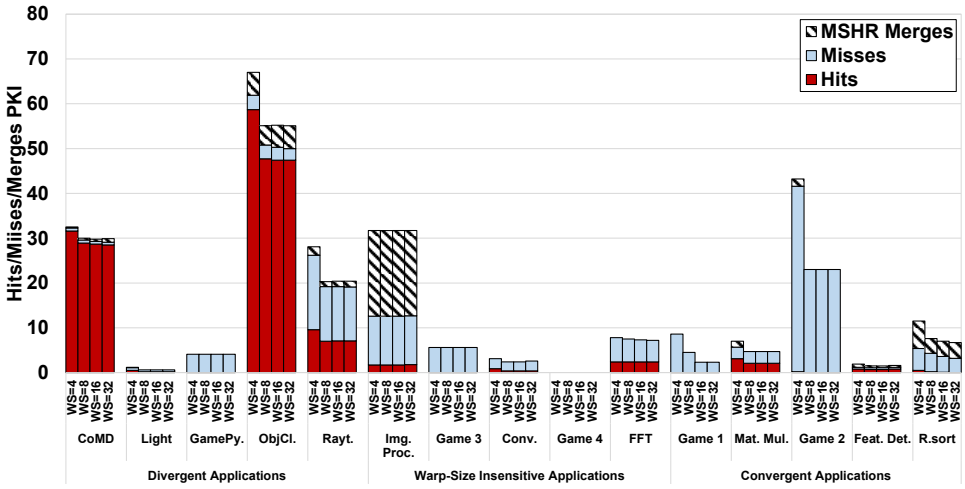


Figure 6.3: L1 data cache hits, misses, and MSHR merges *per thousand instructions* (PKI) at different warp sizes.

2 also see an increase in MSHR merges. The loss in throughput caused by additional traffic to the L2 data cache and DRAM in these applications is not offset by any increase in lane utilization, as these applications already have high SIMT utilization at larger warp sizes. Perhaps not surprisingly, L1 locality for the warp-size insensitive applications is insensitive to the warp size.

6.1.2 Warp Size and SM Front-end Pressure

Figure 6.4 plots the average number of instructions fetched per cycle at various warp sizes. Decreasing the warp size places increasing pressure on the SM's front-end. Convergent and warp-size insensitive applications see a nearly linear increase in fetch requests as the warp size is reduced. This data indicates that a fixed 4-wide warp architecture increases front-end energy consumption for non-divergent applications, even if the performance does not suffer. While divergent applications also see increased front-end activity, the ability of the architecture to exploit many more independent control paths is fundamental to increasing the performance of these applications. Our design focuses on creating a flexible machine that is able to expand and contract the size of warps executing in the system. The best warp size for a given application balances the demands for independent control flow with

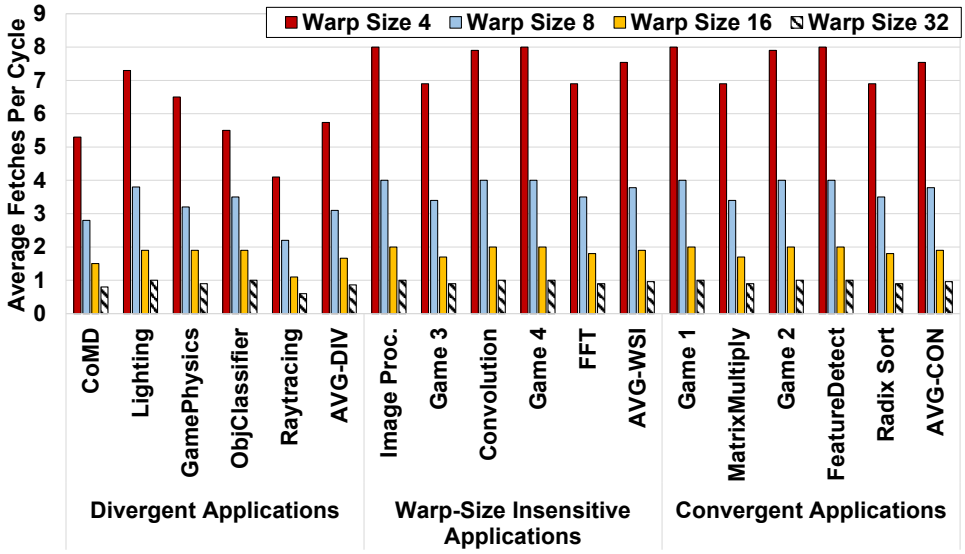


Figure 6.4: Average instructions fetched per cycle. Fetch bandwidth is scaled to match issue bandwidth for each warp size.

the limitations due to memory convergence slip.

6.2 Variable Warp Sizing

This section describes the high level operation of Variable Warp Sizing, discusses the key design decisions, and details the operation of each architectural component. We selected four threads as the minimum warp size for three reasons: (1) graphics workloads commonly process threads in groups of four known as quads, (2) the performance opportunity for the compute workloads we examined reaches diminishing returns at warp sizes smaller than four, and (3) the area overhead rises notably at warp sizes smaller than four. We discuss the area trade-offs of different warp sizes in Section 6.4.7.

6.2.1 High-level Operation

The goal of VWS is to create a machine that is able to dynamically trade off MIMD-like performance with SIMD-like efficiencies depending on the application. Our proposed variable warp sized machine shrinks the minimum warp size to four threads by splitting the traditional GPU datapath into eight unique slices.

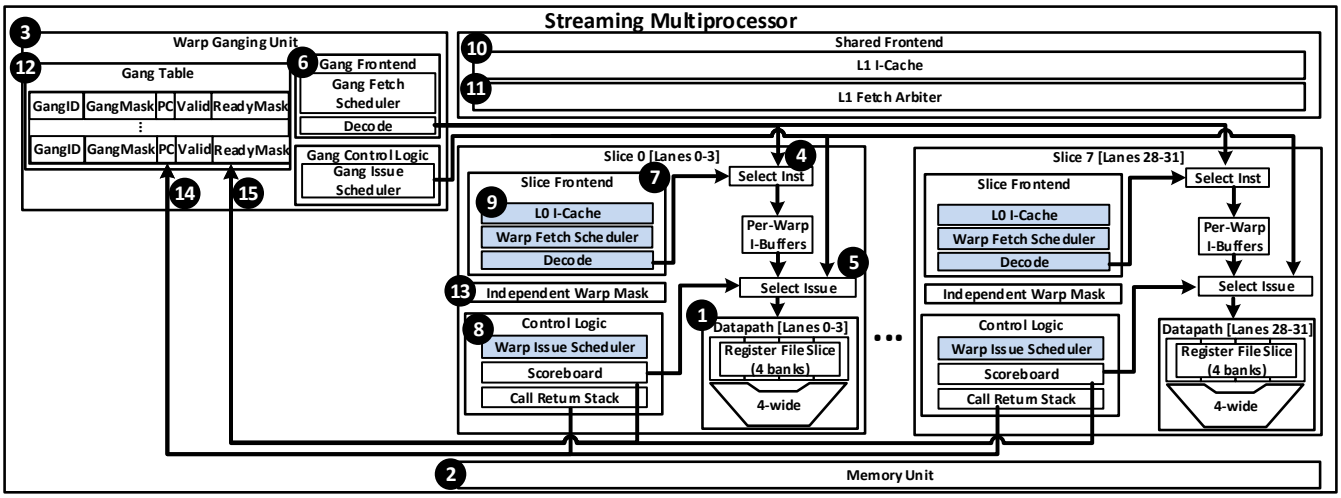


Figure 6.5: Variable Warp Sizing SM microarchitecture. Shaded units are disabled when operating in ganged mode to save energy.

Each slice can fetch, decode, and issue instructions independent of other slices. Figure 6.5 presents the microarchitecture of our proposed design. Each slice is statically assigned threads in a linear fashion: threads 0-3 are assigned to slice 0, 4-7 to slice 1, and so on. Threads cannot migrate between slices.

VWS does not change the number of register file banks in the SM or impose any additional communication between them. As in our baseline, each four-lane slice of the datapath receives its own set of four register file banks (❶ in Figure 6.5). VWS requires no changes to the memory unit (❷), which includes the shared scratchpad memory, L1 data cache, and texture cache. All memory requests generated by any slices in the same cycle are presented to the memory unit as a single 32-thread access in the same manner as when executing 32-wide warps. The coalescing unit also operates in the same fashion as the baseline; multiple threads accessing the same cache line in the same cycle generate only one memory request.

To facilitate warp sizes greater than four, we introduce the warp ganging unit (❸), which is able to override local per-slice fetch/decode (❹) and issue (❺) decisions. The gang front-end (❻) performs instruction fetch and decode once for all small warps participating in a gang. The gang issue scheduler enforces lock-step execution of all slices participating in a given gang. The warp ganging unit is

discussed in more detail in Section 6.2.2.

When VWS is operating in ganged-only mode, the per-slice front-end logic (❼) and warp issue scheduler (❽) are disabled to save energy. When operating in slice-only mode, each SM slice uses its independent front-end to fetch and decode instructions. When both gangs and independent warps are present in the system at the same time, gangs are given both fetch and issue priority. This policy ensures that gangs remain in lock-step as long as possible. When possible, independent warps are used to fill in holes in the executing gang. Each slice front-end includes an L0 I-cache (❾) to reduce pressure on the larger L1 I-cache (❿) which is shared by all slices in the SM. Without L0 I-caches, providing peak throughput in slice-only mode would require 8× the L1 I-cache bandwidth. Our microarchitecture allows 9 separate fetch schedulers (one for each of eight slices and one for gangs) to request instructions from the L1 I-cache. We study the effects of scaling L1 I-cache bandwidth in Section 6.4. Arbitration to determine which scheduler is granted L1 access is done by the L1 fetch arbiter (⓫), described in more detail in Section 6.2.6

This microarchitecture can be run in gang-only or slice-only mode (effectively locking the warp size at 32 or 4 respectively). However, our proposed solutions evaluated in Section 6.4 and described in the remainder of Section 6.2 operate by beginning execution in ganged mode. Sections 6.2.4 and 6.2.5 describe how gangs can be split and reformed on a per-gang basis.

6.2.2 Warp Ganging Unit

The goal of the *warp ganging unit* is to force independent slices to fetch, decode, and execute instructions in lock-step *gangs* when threads across multiple slices are control-convergent. Several factors motivate such ganging. First, issuing memory accesses from convergent applications without lock-step execution places significantly more pressure on the memory system and degrades performance. Second, the system can amortize front-end energy consumption across more threads when some or all small warps across the slices in an SM are executing the same instruction.

When a kernel begins and thread blocks are assigned to an SM, gangs are cre-

ated from the thread blocks in the same fashion as 32-wide warps are created in our baseline system. Each gang is statically assigned eight 4-wide warps, one from each slice. Information about which warps are participating in which gang is stored in the gang table (12). Each entry in the gang table contains a GangID, an 8-bit GangMask (indicating which slices are participating in the gang), the current PC of the gang, a valid bit (which is cleared when the gang's instruction buffer entries are empty), and a ReadyMask which indicates which warps in the gang can issue. To simplify the design of the gang unit, warps are not allowed to migrate between gangs. We implemented more complex gang forming and reforming schemes, but saw no significant performance or energy advantage for our workloads. All warps not participating in a gang (*unganged warps*) are managed independently by their respective slice. Each slice stores an independent warp mask (13) indicating which of its warps are managed independent of the warp ganging unit.

6.2.3 Gang Table

The *gang table* tracks all information necessary for scheduling gangs as well as for managing gang splitting and reformation. The baseline SM described in Chapter 2 has a capacity of 1024 schedulable threads organized into 32 warps of 32 threads each. The VWS SM has the same total thread capacity, but organized into a total of 256 warps of 4-threads each, or 32 4-thread warps per slice. At kernel launch, the threads are aggregated into maximally-sized gangs of eight 4-wide warps, or 32 threads per gang to match the baseline architecture. The term *original gang* is used throughout this paper to describe a gang of warps that is created when a thread block is initially assigned to an SM.

When a gang splits, more entries in the gang table become necessary. Because individual warps are not managed by the warp ganging unit, a gang of 8 warps can split into at most 4 gangs, with a minimum of two warps per gang. Further subdivision yields singleton warps which are managed within each slice. Thus the maximum number of entries needed in the gang table to track the smallest gangs is 128 (32 original gangs $\times$ 4). These 128 entries can be organized in a set-associative manner with 32 sets, one set per original gang and four entries representing up to 4 different gang splits.

Each entry in the gang table contains a unique *GangID* identifier and *GangMask* that indicates which slices are participating in this gang. Since warps can only be ganged with other members of their original gang, all warps from the same original gang access the same set in the gang table and must have GangIDs that are in the same group. For example, warp 0 in each slice can only be a member of gangs 0–3. With this organization, each warp's index in the GangMask is simply the warp's slice number.

To perform fetch and issue scheduling, the warp ganging unit requires information from the slices. Specifically, the gang front-end must know the next PC for each gang, and the gang issue scheduler must know when all warps in a gang have cleared the scoreboard. Per warp call return stack (or reconvergence stack) tracking is done locally in each slice. To track per-gang PCs and handle gang splitting when control flow divergence is encountered, each slice signals the warp ganging unit when the PC at the top of a warp's stack changes (14). Instruction dependence tracking is also done in each slice, even when operating in gang-only mode. Keeping the dependence information local to each slice makes transferring warps from ganged to unganged simpler and decreases the distance scoreboard control signals must travel. The warp ganging unit tracks dependencies for an entire gang in a ReadyMask by receiving scoreboard ready signals from each slice (15).

The gang table also contains a per-entry valid bit to track instruction buffer (I-Buffer) status. The warp gang unit is responsible for both fetching and issuing of gangs. The gang unit front-end stores decoded instructions in each member warp's per-slice I-Buffer. The valid bit is set by the gang fetch scheduler when a gang's per-slice I-Buffer entries are filled and is cleared by the gang issue scheduler when the associated instruction has been issued. All member warps in a gang issue their instructions in lockstep from their respective slice-local I-Buffers. This bit is managed internally by the warp ganging unit and does not require any input from the slices.

6.2.4 Gang Splitting

The warp ganging unit decides when gangs are split and reformed based on a set of heuristics evaluated in Section 6.4. To make splitting decisions, the warp gang unit

observes when control flow and memory divergence occurs. Control flow divergence is detected by observing the PCs sent to the ganging unit by each slice. PCs from the slices undergo a coalescing process similar to global memory accesses. If all warps in a gang access the same PC, no splitting is done. If any warp in the gang accesses a different PC, the gang is split. If more than one warp accesses a common PC, a new gang is formed for these warps. If only one warp accesses a given PC, that warp is removed from the control of the ganging unit and a signal is sent to that warp's slice, transferring scheduling to the local slice. All VWS configurations explored in this work split gangs whenever control flow divergence is detected.

In addition to control flow divergence, memory latency divergence is another motivation for gang splitting. Memory latency divergence can occur when some threads in a warp hit in the data cache while other threads must wait for a long-latency memory operation to complete. Prior work such as Dynamic Warp Subdivision [115] has suggested warp subdivision to tolerate memory latency divergence.

Section 6.4 evaluates VWS architecture configurations that can split gangs when memory latency divergence is observed among member warps. Memory latency divergence is detected when scoreboard ready bits for different warps in a gang are set at different times when completing memory instructions. Tracking which warps in a gang are ready is done through the ReadyMask. We evaluate VWS with two different types of gang splitting on memory divergence. *Impatient Splitting* is the simplest form of gang splitting on memory divergence. If any warp in a gang sets its ready bit before any other member warps, the gang is completely split; all members participating in the gang become independent warps. Impatient splitting simplifies the splitting process and allows highly memory divergent workloads to begin independent execution as quickly as possible. *Group Splitting* enables warps that depend on the same memory access to proceed together as a new gang. When more than one warp in a gang has its ready bit set in the same cycle, a new gang is created from those warps. Any singleton warps that result from this process are placed under independent, per-slice control.

6.2.5 Gang Reformation

In addition to splitting gangs, VWS supports the reformation of gangs that have been split. The warp ganging unit decides if warps or gangs from the same original gang should be re-ganged. While we explored numerous policies, two simple but effective choices emerged: (1) opportunistic reformation and (2) no reformation. To simplify the re-ganging hardware, only one gang can be reformed each cycle. To perform opportunistic gang reformation, one original gang is selected each cycle, in round-robin order. The hardware compares the PCs from each of the original gang's new gangs or independent warps, with a worst-case 8-way comparison if the gang has completely split apart. If any groups of two or more of these warps or gangs have the same PC, they are merged. Section 6.4 describes policies to promote more gang reformation by forcing gangs and warps to wait at common control flow post dominator points in the code.

6.2.6 Instruction Supply

To avoid building a machine with 8× the global fetch bandwidth when VWS is operating in completely independent slice mode, the fetch bandwidth of the L1 instruction cache is limited. We evaluated several different L1-I cache bandwidths and determined that with modestly sized L0 I-caches, L1 I-cache bandwidth can be scaled back to two fetches per cycle and achieve most of the performance of allowing 8 fetches per cycle. The *global fetch arbiter* determines which fetch schedulers access the L1 I-cache's 2 ports on any given cycle. The gang fetch scheduler is always given priority to maximize the number of lanes serviced. The remaining fetch bandwidth is divided among the per-slice warp fetch schedulers. Individual warps are distributed to the slices in round-robin fashion (warp 0 is assigned to slice 0, warp 1 to slice 1, and so on). An arbitration scheme prioritizes slice requests to ensure that each slice gets fair access to the L1 I-cache.

6.3 Experimental Methodology

The results in this paper are collected using a proprietary, cycle-level timing simulator that models a modern GPU streaming multiprocessor (SM) and memory hierarchy similar to that presented in Chapter 2. The simulator is derived from

Table 6.1: Variable warp sizing simulator configuration.

# Streaming Multiprocessors	1
Execution Model	In-order
Warp Size	32
SIMD Pipeline Width	32
Shared Memory / SM	48KB
L1 Data Cache	64KB, 128B line, 8-way LRU
L2 Unified Cache	128KB, 128B line, 8-way LRU
DRAM Bandwidth	32 bytes / core cycle
Branch Divergence Method	ISA Controlled Call Return Stack
Warp Issue Scheduler	Greedy-Then-Oldest (GTO) [137]
Warp Fetch Scheduler	Loose Round-Robin (LRR)
ALU Latency	10 cycles

a product development simulator used to architect contemporary GPUs. Table 6.1 describes the key simulation parameters. The simulator processes instruction traces encoded in NVIDIA's native ISA and generated by a modern NVIDIA compiler. Traces were generated using an execution-driven, functional simulator and include dynamic information such as memory addresses and control flow behavior. We simulate a single SM with 32 SIMT execution lanes that execute 32-wide warps as our baseline, similar to that described in [57]. For warps smaller than 32, we use the same memory system but maintain a fixed count of 32 execution lanes sliced into the appropriate number of groups. We model a cache hierarchy and memory system similar to contemporary GPUs, with capacity and bandwidth scaled to match the portion available to a single SM.

The trace set presented was selected to encompass a wide variety of behaviors. Traces are drawn from a variety of categories, including High Performance Computing (HPC), games, and professional/consumer compute application domains such as computer vision. A third of the selected traces belong to each of the three categories described earlier: (1) divergent codes that prefer narrow warps, (2) convergent codes that prefer wider warps, and (3) codes that are mostly insensitive to warp size.

6.4 Experimental Results

This section details experimental results for the Variable Warp Sizing microarchitecture. First, we quantify performance for several configurations of VWS and then characterize instruction fetch and decode overhead and the effectiveness of mitigation techniques. We perform several sensitivity studies exploring various design decisions for gang scheduling, splitting, and reforming. We demonstrate how gang membership evolves over time for some sample workloads. Finally, we examine area overheads for the proposed design.

6.4.1 Performance

Figure 6.6 plots the performance of multiple warp sizes and VWS, using different warp ganging techniques. All techniques can issue 32 thread instructions per cycle. Fetch and decode rates are scaled with the base warp size; WS4 and WS32 can fetch and decode eight instructions per cycle and one instruction per cycle, respectively. The VWS configurations use a base warp size of 4 and can fetch up to 8 instructions per cycle from the L1 I-cache. Simulating our ganging techniques with 8 times the L1 I-cache fetch throughput allows us to explore the maximum pressure placed on the global fetch unit without artificially constraining it. Section 6.4.2 demonstrates that VWS using the L0 I-caches described in Section 6.2 and an L1 I-cache with only 2× the bandwidth achieves 95% of the performance of using 8× the L1 I-cache bandwidth on divergent applications. Warp-size insensitive and convergent applications are insensitive to L1 I-cache bandwidth. We chose the following VWS configurations based on an exploration of the design space detailed in the rest of this section.

WS 32: The baseline architecture described in Section 2 with a warp size of 32.

WS 4: The baseline architecture described in Section 2 with a warp size of 4.

I-VWS: Inelastic Variable Warp Sizing with a base warp size of 4, where gangs are split only on control flow divergence. Warps are initially grouped together into gangs of 8 warps (32 threads total). Upon control flow divergence, gangs are split based on each warp's control flow path. Once split,

they are never recombined. The ganging unit selects up to two gangs to issue each cycle. Slices that do not receive a ganged instruction pick the next available warp from their pool of unganged warps. The ganged scheduler uses a Big-Gang-Then-Oldest (BGTO) scheduling algorithm, where gangs with the most warps are selected first. Gangs with the same number of warps are prioritized in a Greedy-Then-Oldest (GTO) fashion. Per-slice schedulers manage independent warps using a GTO scheduling mechanism.

E-VWS: Elastic Variable Warp Sizing. Warps are split on control flow divergence and combined in an opportunistic fashion when multiple gangs or singleton warps arrive at the same PC on the same cycle. Gangs can only be created from members of an original gang. A maximum of 2 gangs or warps can be combined per cycle.

E-VWS-ImpatientMem: Warp ganging similar to E-VWS, except that gangs are also split when memory divergence occurs across warps in the same gang. Whenever any memory divergence occurs in a gang, the entire gang is split. Gangs are recombined in the same opportunistic fashion as E-VWS.

Figure 6.6 shows that the I-VWS warp ganging microarchitecture is able to achieve a 35% performance improvement on divergent applications over a static warp size of 32. This improvement is within 3% of using a warp size of 4 on divergent applications and it results in no performance loss on convergent applications where simply using a warp size of 4 results in a 27% slowdown. This data also demonstrates that splitting gangs on control flow divergence without performing any gang recombining, the simplest solution, provides the best overall performance for these workloads. Adding opportunistic gang recombining (E-VWS in Figure 6.6) actually results in a small performance decrease on divergent applications. This decrease is caused by scheduling and packing problems associated with attempting to issue across more slices at once. When gangs are larger, there is a greater likelihood that multiple gangs need to issue to the same slice on the same cycle.

Elastically splitting and regrouping makes no performance difference on convergent and warp-size insensitive applications because these applications experi-

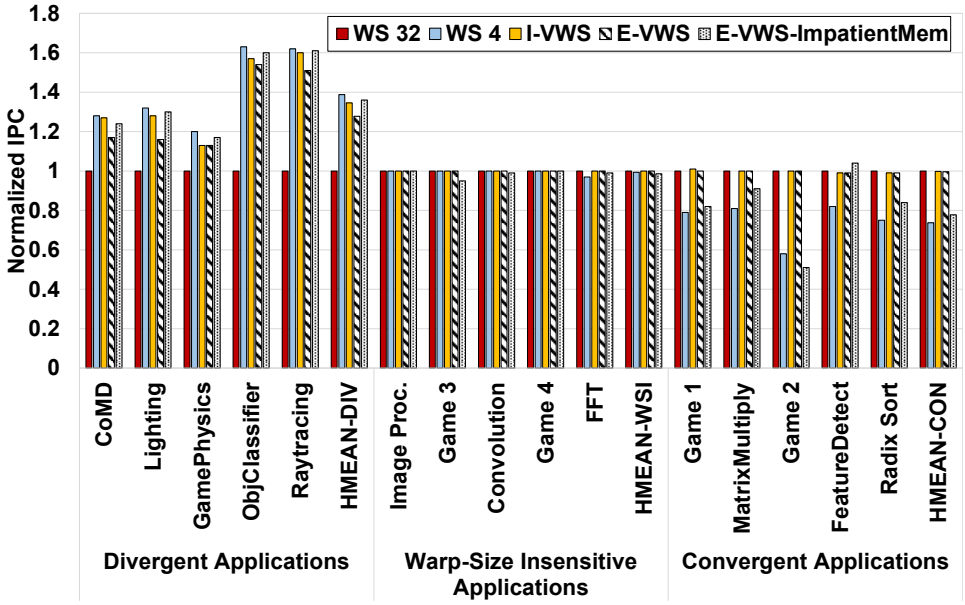


Figure 6.6: Performance (normalized to WS 32) of large warps, small warps, and different warp ganging techniques.

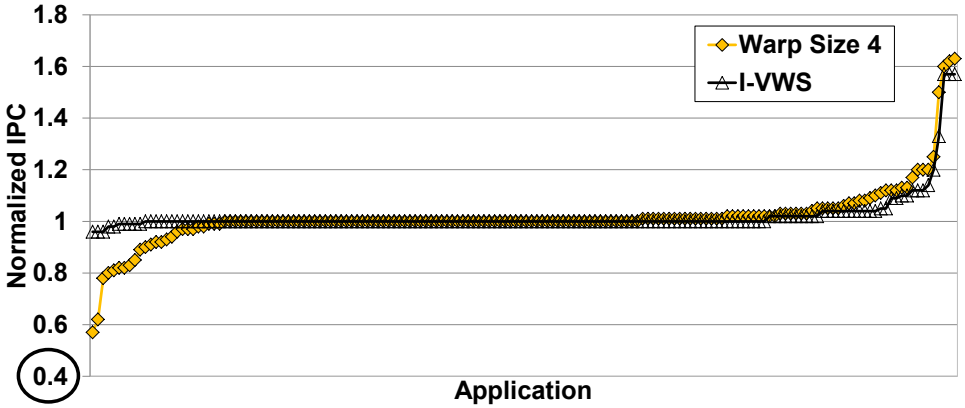


Figure 6.7: Performance (normalized to WS 32) of I-VWS and WS 4 on all the applications from in Figure 6.1a.

ence little or no control flow divergence. Recombining gangs for the divergent workloads makes little performance difference when the hardware has the ability to issue many smaller gangs (or single 4-sized warps) because remaining unganged is unlikely to result in a loss of utilization. Having the ability to concurrently issue multiple paths at the slice granularity makes control flow reconvergence less performance critical than when only one path can be executed concurrently.

Figure 6.6 also quantifies the effect of splitting gangs on memory divergence (E-VWS-ImpatientMem). Reducing the effect of memory divergence helps some of the divergent applications like Lighting, ObjClassifier, and Raytracing and provides a modest 2% performance increase over I-VWS on the divergent applications. However, allowing gangs to split based on memory divergence results in significant performance degradation on Game 1, Game 2, and Radix Sort in the convergent application suite, resulting in an average slowdown of 22% on the convergent applications. Like 4-sized warps, this loss in performance can be attributed to memory convergence slip. Formerly coalesced accesses become uncoalesced and create excessive pressure on the memory system causing unnecessary stalls.

Figure 6.7 shows the performance of all 165 applications. The figure demonstrates that the ganging techniques used in I-VWS are effective for all the applications studied. I-VWS tracks warp size 4 performance on the divergent applications and eliminates warp size 4 slowdown on the convergent applications at the left side of the graph.

6.4.2 Front-end Pressure

Figure 6.8 plots the fetch pressure of several warp sizes and ganging configurations. For the divergent applications, I-VWS results in 57% fewer fetch/decode operations required each cycle versus a warp size of 4. This reduction in fetch/decode represents a significant energy savings while providing almost all of the performance of 4-sized warps. By opportunistically recombining gangs for divergent applications, E-VWS requires a further 55% less fetch/decode bandwidth than I-VWS, at the cost of some performance. On divergent applications, E-VWS-ImpatientMem increases fetch/decode pressure versus E-VWS but not more than I-VWS.

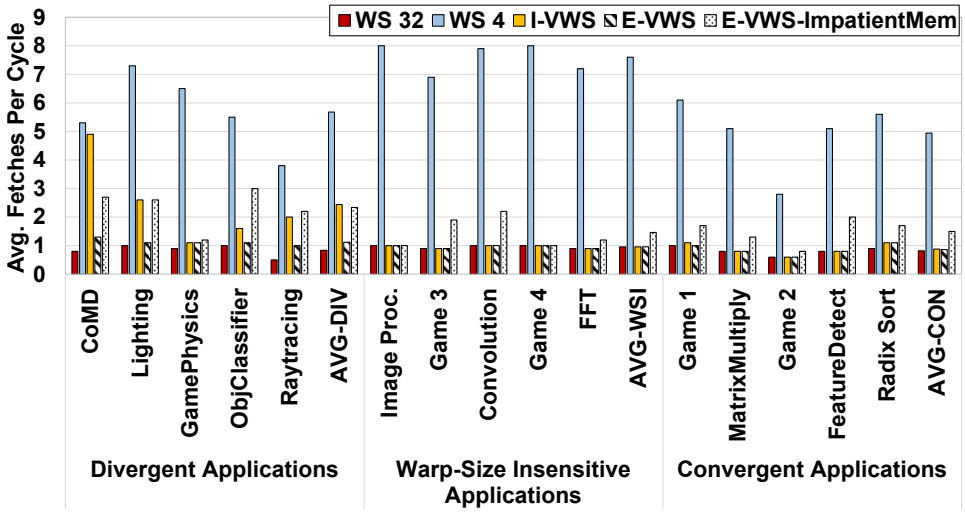


Figure 6.8: Average fetches per cycle with different warp sizes and ganging techniques.

On the convergent and warp-size insensitive applications, the ganging configurations that do not split on memory divergence show fetch pressure equal to that of warp size 32. Because these applications lack control flow divergence, gangs rarely split and I-VWS operates exclusively in ganged mode. However, when gangs are split on memory divergence, the skewing of memory access returns causes a significant increase in the number of fetch/decodes per cycle.

Figure 6.9 plots the performance of I-VWS at different L1 I-cache bandwidths and L0 I-cache sizes. Because the divergent applications traverse multiple independent control flow paths, restricting L1 I-cache bandwidth results in a significant performance loss. However, the inclusion of per-slice L0 I-caches, which are probed first when independent warps fetch instructions, vastly decreases the performance loss. With only 2$\times$ the L1 I-cache bandwidth of the baseline architecture, the addition of small 256B L0s are able to cover most of the bandwidth deficiency at the L1. Since they remain in ganged operation, the warp-size insensitive and convergent applications are insensitive to L1 I-cache fetch bandwidth.

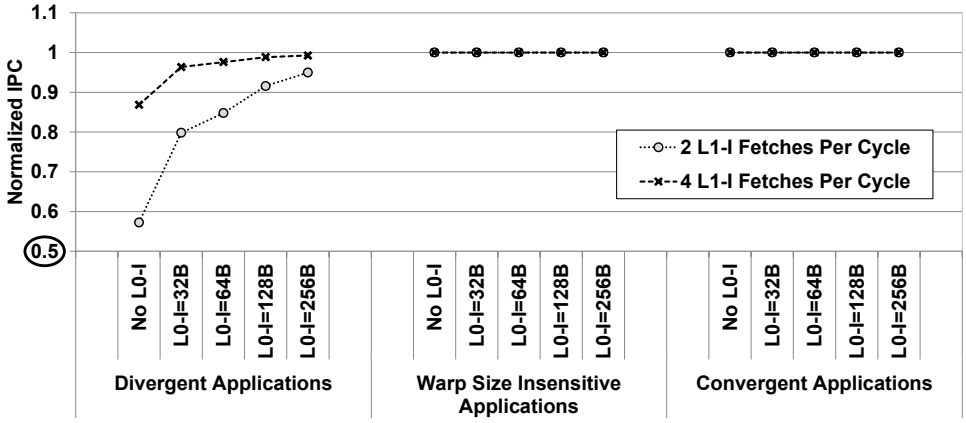


Figure 6.9: Average performance of I-VWS at different L1 I-cache bandwidths and L0 I-cache sizes. Normalized to I-VWS with 8x L1-I cache bandwidth.

6.4.3 Gang Scheduling Policies

We measured the sensitivity of performance and instruction fetch bandwidth to several different gang scheduling policies. All gang schedulers attempt to issue up to two gangs per cycle, and local per-slice schedulers attempt to issue on any remaining idle slices. We examine the following policies:

I-VWS: As described in Section 6.4.1, the gang issue scheduler prioritizes the largest gangs first Big-Gangs-Then-Oldest (BGTO) and per-slice schedulers are Greedy-Then-Oldest (GTO).

I-VWS-GTO: Similar to I-VWS, except the gang issue scheduler uses a greedy-then-oldest policy.

I-VWS-LRR: Similar to I-VWS, except both the gang issue scheduler and per-slice schedulers use a Loose-Round-Robin (LRR) scheduling policy.

I-VWS-LPC: Similar to I-VWS, except both the gang issue scheduler and per-slice schedulers prioritize gangs/warps with the lowest PC first.

I-VWS-LGTO: Similar to I-VWS, except the gang issue scheduler prioritizes gangs with the fewest warps first Little-Gangs-Then-Oldest (LGTO). Per-slice schedulers use a GTO policy.

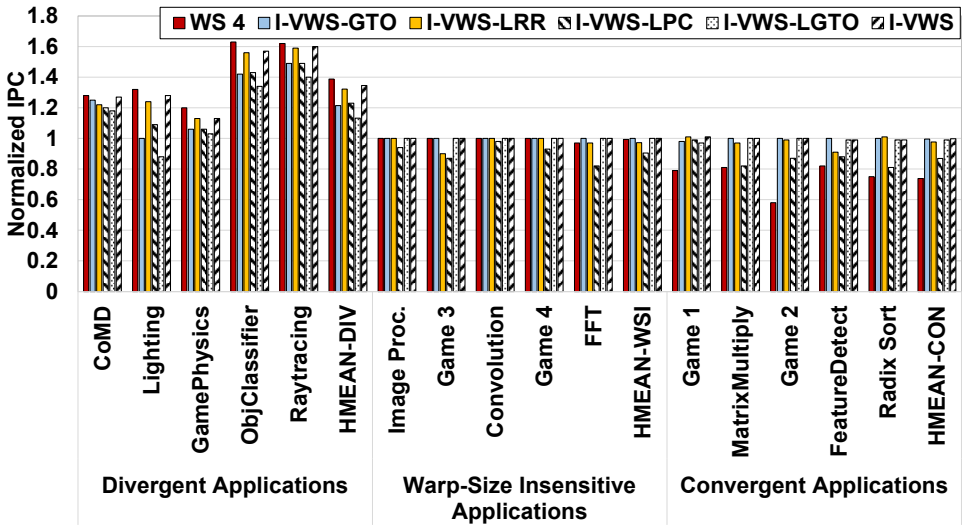


Figure 6.10: Performance (normalized to WS 32) of warp ganging with different schedulers.

Figure 6.10 shows that the performance of the divergent applications is sensitive to the gang scheduler choice. The lowest-PC-first configuration results in a universal performance reduction across all the applications. Little-Gangs-Then-Oldest (I-VWS-LGTO) creates a scheduling pathology on the divergent applications. Prioritizing the smallest gangs first is bad for performance because the gang issue scheduler can only select a maximum of 2 gangs for execution each cycle; giving the smallest ones priority can limit utilization by delaying the execution of gangs with many warps. We also observed that prioritizing little gangs was detrimental even when more than two gangs could be scheduled per cycle because little gangs block the execution of larger gangs. Figure 6.11 shows the resulting fetch and decode requirements for different gang scheduling policies. Although the choice of gang scheduler has a significant effect on performance, it has little effect on fetch/decode bandwidth. This insensitivity occurs because gang scheduling has nothing to do with gang splitting when gangs are split only for control flow divergence and are not recombined. When splitting gangs on memory divergence is enabled, the effect of scheduling on the fetch rate is much greater.

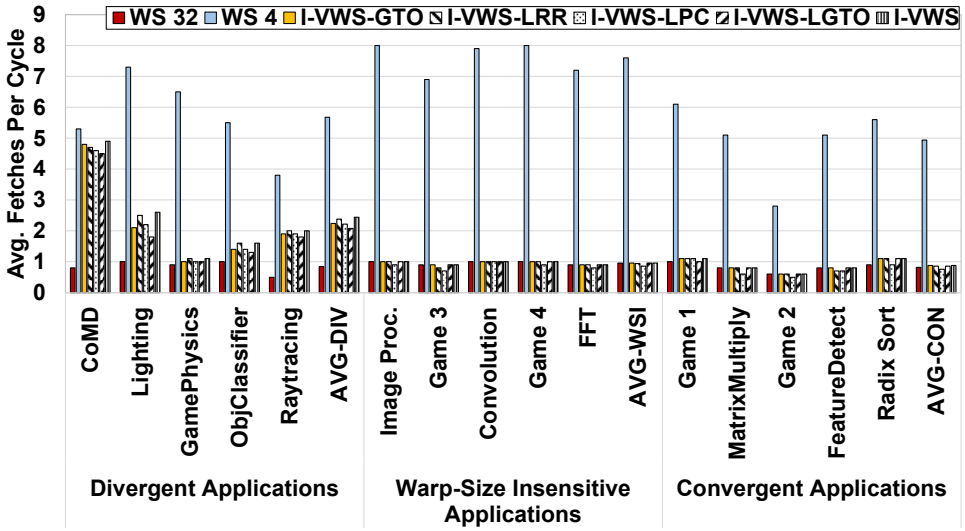


Figure 6.11: Averages fetches per cycle with different schedulers.

Figure 6.12 plots performance when the number of gangs selectable per cycle by the gang issue scheduler is set to one, two, or unlimited (up to four). This data shows that limiting the gang scheduler to a single gang per cycle reduces the performance of the divergent applications by 10% versus the baseline of two gangs per cycle. Allowing the gang scheduler to pick unlimited gangs per cycle results in performance that is within 1% of two gangs per cycle. Any slices not consumed by the gang scheduler may be used whenever possible by any singleton warps managed by local slice schedulers. We choose to limit the gang scheduler to two gangs per cycle to balance performance and scheduler complexity.

6.4.4 Gang Reformation Policies

Figures 6.13 and 6.14 plot performance and instruction fetches per cycle when the following policies are used to reform gangs after they have been split:

E-VWS: As described in Section 6.4.1, gangs are reformed on an opportunistic basis only.

E-VWS-Sync<XX>: Similar to E-VWS, except that when warps reach a compiler-inserted call-return stack sync instruction, they wait for recombination. These

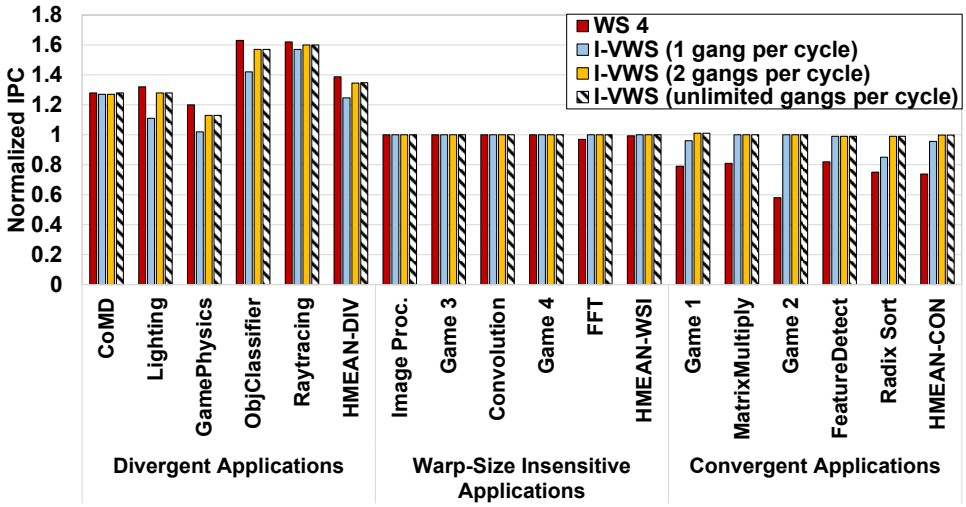


Figure 6.12: Performance (normalized to WS 32) when the number of gangs able to issue each cycle is changed.

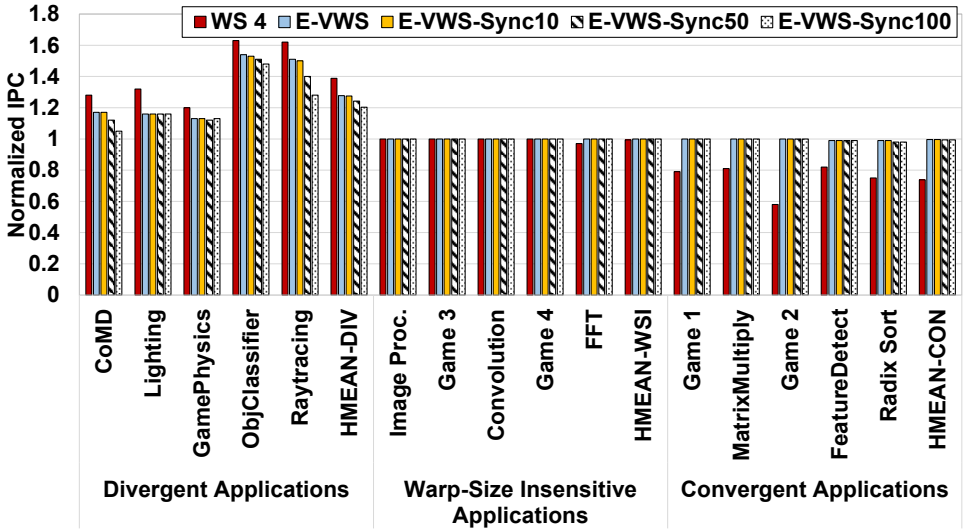


Figure 6.13: Performance (normalized to WS 32) of elastic gang reformation techniques.

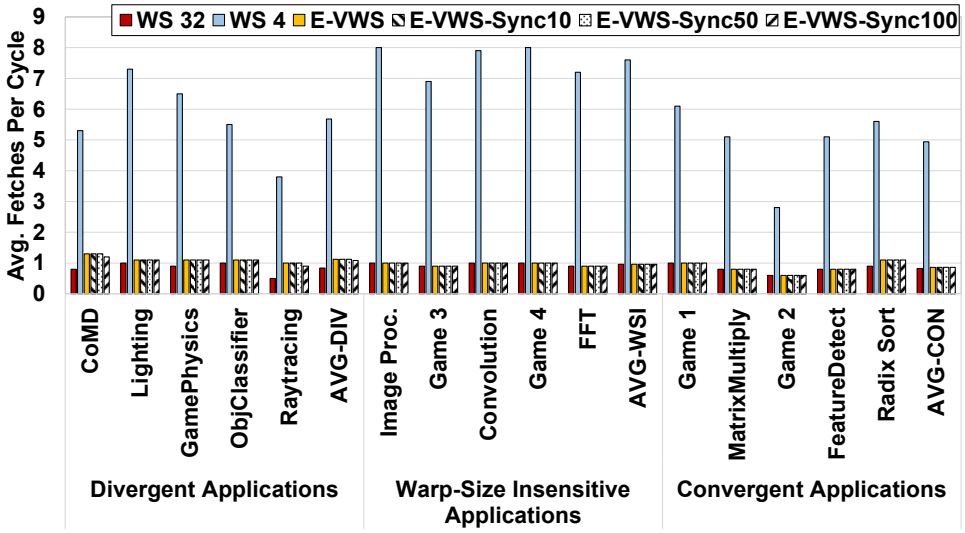


Figure 6.14: Average fetches per cycle with different gang reformation techniques.

instructions are already inserted, typically at basic block post-dominators, to enforce the NVIDIA call-return stack architecture. <XX> indicates how many cycles a warp will wait at the sync point for potential reganging.

Forcing warps to wait at control flow post-dominator points can potentially improve gang reformation, leading to more or larger gangs and reduced front-end energy while hopefully resulting in minimal performance degradation. Figures 6.13 and 6.14 demonstrate that waiting at sync points results in a performance loss on our divergent applications. We see minimal decrease in the number of fetches per cycle as waiting time is increased, and any energy efficiency gained from this reduction would be more than offset by the loss in performance. The warp-size insensitive and convergent applications contain fewer compiler-inserted sync points, experience little or no control flow divergence, and may spend much or all of their execution time fully ganged. As a result, their performance is largely unaffected by wait time at infrequent sync points. Thus we conclude that forcing warps to wait at post-dominators provides little to no benefit; most of the reduction in fetch pressure is captured by opportunistic reganging in E-VWS.

6.4.5 Gang Splitting Policies

Figure 6.15 explores the use of the following gang splitting policies without any gang reformation:

I-VWS: As described in Section 6.4.1. Warps are split only on control flow divergence.

I-VWS-GroupMem: Warp ganging similar to I-VWS except gangs are also split on memory divergence. As memory results return for a gang, all warps in a gang that are able to proceed based on the newly returned value form a new gang. Gangs are never recombined.

I-VWS-ImpatientMem: Warp ganging similar to I-VWS-GroupMem except gangs that experience any memory divergence are completely subdivided into individual warps.

As in Section 6.4.1, Figure 6.15 demonstrates that splitting on memory latency divergence can have a small performance advantage on some divergent applications, but has a large performance cost on convergent ones. Minimizing the amount of splitting that occurs on memory divergence (I-VWS-GroupMem) gains back some of the performance lost for Game 2 but creates problems in Radix Sort. Overall, splitting on memory divergence is a net performance loss due to its negative effect on convergent applications.

Figure 6.16 plots the resulting number of instructions fetched per cycle when different gang splitting policies are used. This data demonstrates that even though splitting on memory divergence may be a small performance win for divergent applications, the number of instructions fetched increases greatly as a result, by 41% and 69% for I-VWS-GroupMem and I-VWS-ImpatientMem, respectively.

6.4.6 Gang Size Distribution

Figure 6.17 visualizes how gang sizes change over time for two example divergent workloads, GamePhysics and Lighting. Each warp assigned to the SM on any given cycle is classified according to the size of the gang to which it belongs. For example in the Lighting application, execution begins with 120 4-wide warps assigned to the

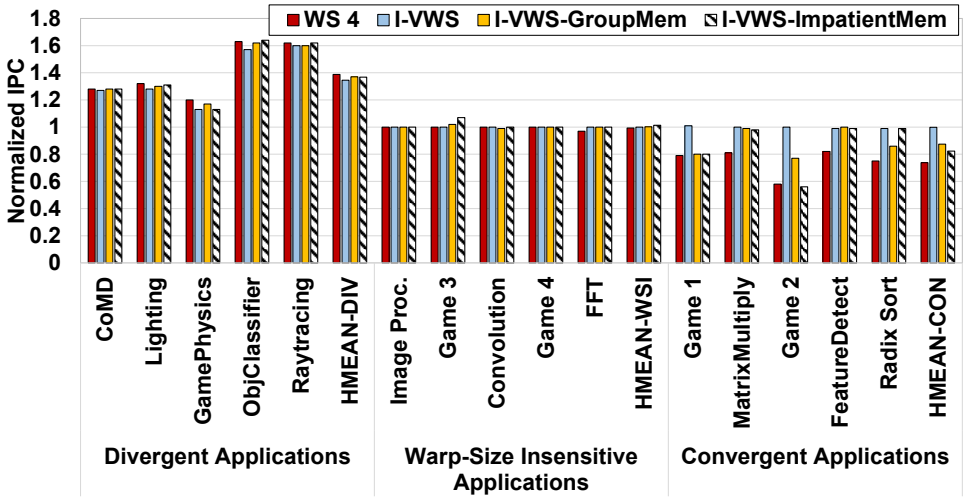


Figure 6.15: Performance (normalized to WS 32) of different gang splitting policies.

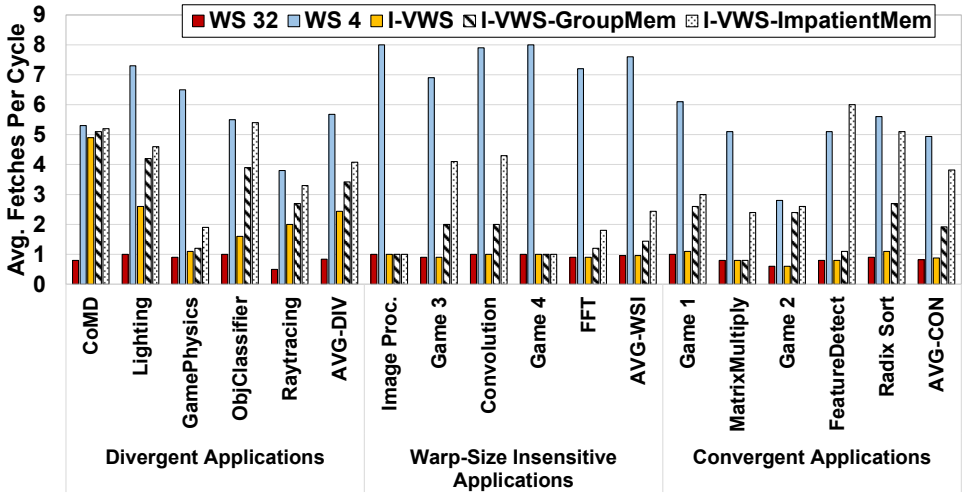


Figure 6.16: Average fetches per cycle using different gang splitting policies.

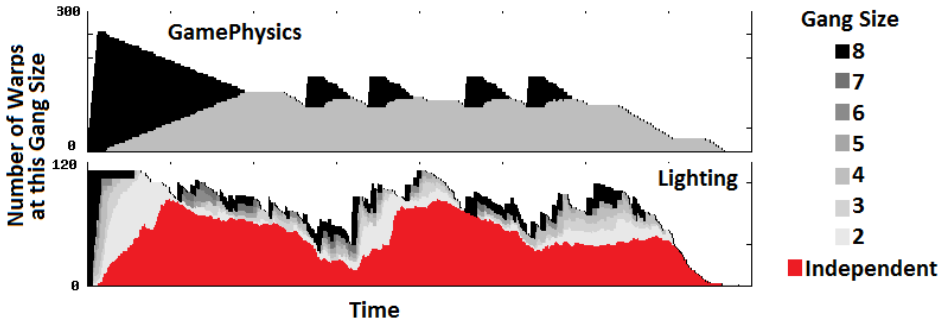


Figure 6.17: Gang sizes versus time for I-VWS.

SM. The black bar at cycle 0 indicates that all warps start out in their original gangs of size 8. As time progresses, the original gangs split apart into smaller gangs until eventually most warps in the SM are executing independently. In contrast, GamePhysics exhibits much more structured divergence. The SM begins execution with 300 warps all in their original gangs. Over time, the warps split in two (the grey color in the GamePhysics graph represents warps participating in a gang of 4). One half of the gang exits, while the other half continues executing in lock step. These two plots illustrate how I-VWS reacts to different kinds divergence. Most of the divergent workloads studied react similar to Lighting. Similar plots for E-VWS show gangs splitting and reforming as time progresses. The plots collected for the convergent applications show that warps stay in their original gang throughout execution.

6.4.7 Area Overheads

Table 6.2 presents an estimate of the area required to implement I-VWS in a 40nm process. Column two presents the raw area estimate for each I-VWS component, while columns three and four present a rolled-up incremental SM area increase for 4-wide and 8-wide warps, respectively. We model the L1 I-cache using CACTI [171] at 40nm. The L0 I-cache, decoded I-Buffers, and the gang table are small but dominated by the storage cells required to implement them. We estimate the area of these structures by using the area of a latch cell from the NanGate 45nm Open Cell library and scaling it to 40nm. We multiply the resulting cell area

Table 6.2: Area overhead estimates.

Component	Component Area	Additional SM Area	
		4-wide warps	8-wide warps
Single-ported L1 I-cache (64KB)	0.087mm^2		
Dual-ported L1 I-cache (64KB)	0.194mm^2	0.108mm^2	0.108mm^2
L0 I-cache (256B)	0.006mm^2	0.052mm^2	0.026mm^2
Decoded I-Buffers (4Kbits)	0.013mm^2	0.103mm^2	0.052mm^2
Gang Table (128 entries)	0.026mm^2	0.026mm^2	0.026mm^2
Scoreboard (1800 bits)	0.019mm^2	0.154mm^2	0.077mm^2
Additional control	0.160mm^2	1.280mm^2	0.640mm^2
Total SM area increase		1.7mm^2	0.93mm^2
Percent SM area increase		11%	6%
Total GPU area increase		25.8mm^2	13.9mm^2
Percent GPU area increase		5%	2.5%

($2.1\mu m^2$) by the number of bits and a factor of 1.5 to account for area overheads including control logic. For the per-slice scoreboards, we use a larger FlipFlop cell ($3.6\mu m^2$ scaled to 40nm) from the NanGate library and 3$\times$ area overhead factor to account for the comparators necessary for an associative lookup. Compared to the scoreboard described in [48], ours has fewer bits and noticeably less area. Finally, to estimate the area cost of the additional control logic required for slicing the SIMD datapath, we examine published literature on the percentage of total core area other processors devote to control [3, 23, 109, 110]. Based on these studies and the high datapath densities found in GPUs, we estimate that 1% of the Fermi SM area (16mm^2) is devoted to the datapath control logic that needs to be replicated in each slice. In total, we estimate that I-VWS adds approximately 11% and 6% to the area of an SM for 4-wide and 8-wide warps, respectively. For a Fermi-sized 15 SM GPU (529mm^2), I-VWS adds approximately 5% and 2.5% more area for 4-wide and 8-wide warps, respectively.

6.5 Comparison to Previous Work

This section first presents a quantitative comparison of I-VWS against two previously proposed techniques which address the effect of branch divergence on GPUs. It then presents a qualitative characterization of our work and prior art in the branch and memory divergence mitigation design space.

6.5.1 Quantitative Comparison

We compare I-VWS to two previously proposed divergence mitigation techniques: *Thread Block Compaction* (TBC) [53] and *Dynamic Warp Formation* (DWF) [55]. This data was collected using TBC's published simulation infrastructure [54], which is based on GPGPU-Sim 2.x [19]. We run TBC and DWF with the exact configuration specified in [54] and implement I-VWS with a warp size of 4 on top of their infrastructure. Figure 6.18 plots the result of this study on 5 divergent applications taken from the TBC simulation studies: raytracing (NVRT) [13], face detection (FCDT) [110], breadth first search (BFS) [36], merge sort (MGST) [36], and nearest neighbor (NNC) [36]. NVRT does not run when using DWF [53]. We chose these applications because their divergence behavior highlights the advantages of I-VWS. We also ran I-VWS on the rest of the applications in the TBC paper and observed little performance difference. On the five applications, I-VWS achieves an average 34% and 15% performance improvement over the baseline 32-wide warp and TBC respectively. The increased issue bandwidth and scheduling flexibility of I-VWS enables divergent applications to better utilize the GPU.

6.5.2 Qualitative Comparison

Table 6.3 presents a qualitative characterization of previously proposed divergence mitigation techniques versus small warps and I-VWS. We classify previous work into three categories: (1) techniques that dynamically form or compact warps to regain SIMD efficiency [53, 55, 123, 132, 134]; (2) techniques that subdivide warps in the presence of memory and control flow divergence to expose more thread level parallelism [115, 156]; and (3) techniques that allow the interleaved execution of multiple branch paths within a warp by scheduling multiple paths from the control flow stack [48, 133].

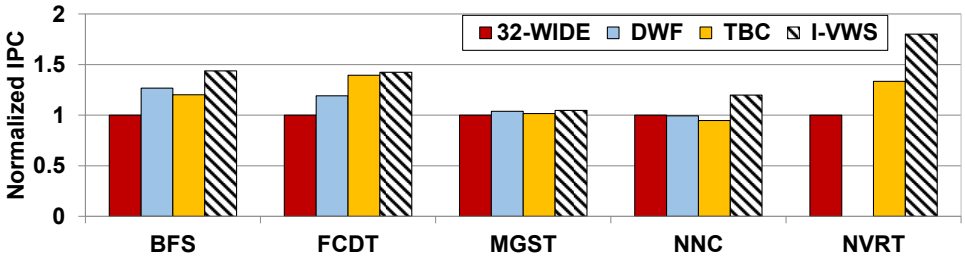


Figure 6.18: Performance (normalized to the 32-wide warp baseline) using the released TBC infrastructure [54].

The fundamental characteristic that sets I-VWS and the use of small warps apart from prior work is the ability to concurrently issue many more unique PCs by scaling and distributing instruction fetch bandwidth. Additionally, I-VWS can seamlessly adapt to memory latency divergence by breaking gangs. *Dynamic Warp Subdivision* (DWS) is also able to break lock-step warp execution on memory divergence [115]. However, DWS does this with a loss of SIMD efficiency and requires additional entries to be added to a centralized warp scheduler for each subdivided warp. In contrast, I-VWS allows smaller warps to continue independently, without losing SIMD efficiency; management of these smaller warps is distributed among many smaller scheduling entities.

While formation and compaction techniques increase SIMD efficiency, they pay for this increase with a reduction in available thread level parallelism, since forming and compacting warps decreases the number of schedulable entities. This decrease in TLP degrades the SM's ability to tolerate latencies. Conversely, subdivision and multipath stack techniques can increase latency tolerance but are limited by the number of scheduling entities in a monolithic warp scheduler and the number of concurrent entries on the call return stack.

All of the techniques perform well on convergent applications with the exception of small warps which suffer from memory convergence slip. We consider two classes of divergent applications: those that have a limited number of unique control flow (CF) paths and those that have many unique control flow paths. All of the proposed techniques perform well on applications that have a limited number of control flow paths. However, only smaller warps and I-VWS can maintain good

Table 6.3: Qualitative characterization of divergence mitigation techniques.

Characteristic	**Form/ Compact** [53, 55, 123]	**Subdivide** [115]	**Multipath** [48, 133]	**Small Warps**	**I-VWS**
# PCs per Cycle	1	1	1	Many	Many
Mem. Divergence Adaptive	No	Yes	No	Yes	Yes
Latency Tolerance	Low	Limited	Limited	High	High
Performance					
Convergent apps	High	High	High	Low	High
Divergent apps, limited CF	High	High	High	High	High
Divergent apps, many CF	Limited	Limited	Limited	High	High
Energy Efficiency					
Convergent apps	High	High	High	Low	High
Divergent apps, limited CF	High	High	High	Medium	Medium
Divergent apps, many CF	Low	Low	Low	High	High

performance when the number of unique control flow paths is high. Compaction and formation techniques require candidate threads to be executing the same instruction (PC). Subdivision and multipath stack techniques increase the number of schedulable entities in the SM, but do not improve lane utilization and become limited by the number of entries in a large, centralized structure when the number of unique control flow paths is large. Energy efficiency largely follows performance. On diverged applications with limited control flow paths, small warps and I-VWS lose some energy efficiency by fetching and decoding multiple times from smaller distributed fetch/decode structures, while prior work fetches one instruction from a larger structure. However, on divergent applications with many control flow paths, prior work inefficiently fetches one instruction repeatedly from a larger structure, while small warps and I-VWS distribute this repeated fetching over smaller structures. The smaller warps in I-VWS make it the only technique which improves both SIMD efficiency and thread level parallelism, while still exploiting the performance and energy efficiencies presented by convergent and warp-size insensitive code.

6.6 Summary

This chapter explores the design space of a GPU SM with the capability to natively issue from multiple execution paths in a single cycle. Our exploration concludes that convergent applications require threads to issue in lock-step to avoid detrimental memory system effects. We also find that the ability to execute many control flow paths at once vastly decreases a divergent application's sensitivity to reconvergence techniques.

We propose *Variable Warp Sizing* (VWS) which takes advantage of the many control flow paths in divergent applications to improve performance by 35% over a 32-wide machine at an estimated 5% area cost when using 4-wide warps. An 8-wide design point provides most of that performance benefit, while increasing area by only 2.5%. VWS evolves GPUs into a more approachable target for irregular applications by providing the TLP and SIMD efficiency benefits of small warps, while exploiting the regularity in many existing GPU applications to improve performance and energy efficiency.

Chapter 7

Related Work

This chapter summarizes and contrasts the work done in this dissertation against related work in thread scheduling, locality management and GPU divergence mitigation. Section 7.1 discusses work relating to Cache-Conscious Warp Scheduling and Divergence-Aware Warp Scheduling. Section 7.2 details work related to A Variable Warp-Sized Architecture that examines exploiting horizontal locality as well as mitigating branch and memory divergence in GPUs and vector processors.

7.1 Work Related to Cache-Conscious Warp Scheduling and Divergence-Aware Warp Scheduling

This section is subdivided into five subsections that classify work related to CCWS and DAWS: throttling to improve performance, GPU thread scheduling techniques, GPU caching, CPU thread scheduling techniques, cache capacity management and finally, locality detection.

7.1.1 Throttling to Improve Performance

Bakhoda et al. [19] present data for several GPU configurations, each with a different maximum number of CTAs that can be concurrently assigned to a core. They observe that some workloads performed better when less CTAs are scheduled concurrently. The data they present is for a GPU without an L1 data cache, running a round-robin warp scheduling algorithm. They conclude that this increase in per-

formance occurs because scheduling less concurrent CTAs on the GPU reduces contention for the interconnection network and DRAM memory system. In contrast, the goal of CCWS and DAWS is to use L1 data cache feedback to preserve locality. Our techniques focus on fine-grained, issue level warp scheduling, not coarse-grained CTA assignment.

Guz et al. [59] use an analytical model to quantify the "performance valley" that exists when the number of threads sharing a cache is increased. They show that increasing the thread count increases performance until the aggregate working set no longer fits in cache. Increasing threads beyond this point degrades performance until enough threads are present to hide the system's memory latency. In effect, CCWS and DAWS dynamically detect when a workload has entered the machine's performance valley and they scale back the number of threads sharing the cache to compensate.

Cheng et al. [40] introduce a thread throttling mechanism to reduce memory latency in multithreaded CPU systems. They propose an analytical model and memory task throttling mechanism to limit thread interference in the memory stage. Their model relies on a stream programming language which decomposes applications into separate tasks for computation and memory and their technique schedules tasks at this granularity.

Ebrahimi et al. [47] examine the effect of disjointed resource allocation between the various components of a chip-multiprocessor system, in particular in the cache hierarchy and memory controller. They observed that uncoordinated fairness-based decisions made by disconnected components could result in a loss of both performance and fairness. Their proposed technique seeks to increase performance and improve fairness in the memory system by throttling the memory accesses generated by CMP cores. This throttling is accomplished by capping the number of MSHR entries that can be used and constraining the rate at which requests in the MSHR are issued to the L2.

Prior work in networking such as those by Thottethodi et al. [158], Baydal et al. [24], Lopez et al. [106, 107] and Scott and Sohi [145] use feedback to generate estimations of network congestion and dynamically tune network injection rates to improve performance. CCWS and DAWS utilize source throttling at the instruction issue stage and optimize for cache capacity, not network bandwidth. An interesting

extension to CCWS and DAWS would be to examine their effect on the GPU's network congestion and if there are network injection control techniques that might be a better fit at the instruction issue stage as opposed to the network injection stage.

7.1.2 GPU Thread Scheduling Techniques

Lakshminarayana and Kim [91] explore numerous warp scheduling policies in the context of a GPU without hardware managed caches and show that, for applications that execute symmetric (balanced) dynamic instruction counts per warp, a fairness based warp and DRAM access scheduling policy improves performance. In contrast to our work, their study did not explore scheduling policies that improve performance by improving cache hit rates.

Fung et al. [55] explore the impact of warp scheduling policy on the effectiveness of their Dynamic Warp Formation (DWF) technique. DWF attempts to mitigate control flow divergence by dynamically creating new warps when scalar threads in the same warp take different paths on a branch instruction. They propose five schedulers and evaluate their effect on DWF. Fung and Aamodt [52] also propose three thread block prioritization mechanisms to compliment their Thread Block Compaction (TBC) technique. The prioritization mechanisms attempt to schedule threads within the same CTA together. Their approach is similar to concurrent work on two-level scheduling proposed by Narasiman et al. [123], except thread blocks are scheduled together instead of fetch groups. In contrast to both these works, CCWS and DAWS explore the impact of scheduling on cache locality using existing control flow divergence mitigation techniques.

Gebhart and Johnson et al. [57] introduce the use of a two-level scheduler to improve energy efficiency. Experiments run using their exact specification yielded mixed results. They note that the performance of their workloads increases less than 10% if a perfect cache is used instead of no cache at all. For this reason, they run all their simulations with a constant 400 cycle latency to global memory. As a result, their scheme switches warps out of the active pool whenever a compiler identified global or texture memory dependency is encountered. We find that obeying this constraint causes performance degradation because it does not take cache hits into account. However, if this demotion to the inactive pool is changed to just

those operations causing a stall (i.e. those missing in cache) its operation is similar to concurrent work on two level scheduling done by Narasiman et al. [123] which is evaluated in Section 3.4.

The work done by Narasiman's et al. [123] focuses on improving performance by allowing groups of threads to reach the same long latency operation at different times. This helps ensure cache and row-buffer locality within a fetch group is maintained and the system is able to hide long latency operations by switching between fetch groups. In contrast, our work focuses on improving performance by adaptively limiting the amount of multithreading the system can maintain based on how much intra-warp locality is being lost.

Meng et al. [116] introduce Dynamic Warp Subdivision (DWS) which splits warps when some lanes hit in cache and some lanes do not. This scheme allows individual scalar threads that hit in cache to make progress even if some of their warp peers miss. DWS improves performance by allowing run-ahead threads to initiate their misses earlier and creates a pre-fetching effect for those left behind. DWS attempts to improve intra-warp locality by increasing the rate data is loaded into the cache. In contrast, CCWS and DAWS attempt to load data from less threads at the same time to reduce thrashing.

Since the publication of CCWS, Jog et al. [78] and Kayiran et al. [83] have proposed locality aware thread block schedulers that seek to limit the number of thread blocks sharing the L1D cache. Their techniques apply warp limiting at a coarse grain. CCWS and DAWS seek to maximize cache usage using runtime cache feedback, fine grain divergence information and code region characterization. Lee et al. [94] and Jog et al. [79] explore prefetching on the GPU, with the latter focusing on prefetching-aware scheduling. In contrast to prefetching, which cannot improve performance in bandwidth limited applications, CCWS and DAWS make more effective use of on-chip storage to reduce memory bandwidth.

Sethia et al. [148] introduce Mascar which attempts to better overlap computation with memory accesses in memory intensive workloads. Mascar consists of a memory aware warp scheduler that prioritizes the execution of a single warp when MSHR and L1 miss queue entries on the chip are oversubscribed. This prioritization helps improve performance even when workloads do not contain data locality by enabling warps executing on the in-order core to reach their computa-

tion operations faster, enabling overlap of the prioritized warp's computation with other warp's memory accesses. Mascar also introduces a cache access re-execution mechanism to to help avoid L1 data cache thrashing by enabling hits-under-misses when warps with data in the cache are blocked from issuing because low-locality accesses are stalling the memory pipeline.

Ausavarungnirun et al. [17] propose a series of improvements at the shared L2 and memory controller that mitigate memory latency divergence in irregular GPU applications. The techniques, collectively named Memory Divergence Correction (MeDiC), exploit the observation that there is heterogeneity in the level of memory latency divergence across warps in the same kernel. Based on how they interact with the shared L2 cache, each warp in a kernel can be characterized as all/mostly hit, all/mostly miss or balanced. The authors demonstrate that there is little benefit in having warps that are not all hit, since warps that mostly hit must wait for the slowest access to return before they are able to proceed. They also demonstrate that queueing latency at the L2 cache can have a non-trivial performance impact on all-hit warps and that this effect can be mitigated by bypassing the L2 cache for all the requests made by any warp that is not all-hit. This decreases the access latency for all-hit warps by reducing queueing delay. In addition to the adaptive bypassing technique, they propose modifications to the cache replacement policy and the memory controller scheduler in an attempt to minimize latency for warps detected to be all-hit warps.

7.1.3 GPU Caching

Work by Lee et al. [94] has explored the use of prefetching on GPUs. However, prefetching cannot improve performance when an application is bandwidth limited whereas CCWS and DAWS can help in such cases by reducing off-chip traffic.

Jia et al. [75] characterize GPU L1 cache locality in a current NVIDIA Tesla GPU and present a compile time algorithm to determine which loads should be cached by the L1D. In contrast to our work, which focuses on locality between different dynamic load instructions, their algorithm and taxonomy focus on locality across different threads in a single static instruction. Moreover, since their analysis is done at compile time they are unable to capture any locality with input data

dependence.

Published after CCWS and DAWS, Li et al. [105] make the observation that in the interest of improving L1 cache efficiency CCWS and DAWS potentially leave other resources such as L2 capacity and memory bandwidth underutilized. To mitigate these effects, they propose a token-based cache line allocation scheme to determine which warps are permitted to allocate lines in the L1, allowing other warps to bypass the L1 cache and consume the memory bandwidth and L2 cache capacity made available by thread throttling schemes such as CCWS and DAWS.

7.1.4 CPU Thread Scheduling Techniques

Prior simulation and analytical modeling work on CPU multiprocessors has investigated degraded cache and network performance due to multithreading and multiprocessing [10, 28, 143, 168]. GPU work examining the effect of massive multithreading on system contention, like CCWS and DAWS, evaluate multithreading at a scale previously not imaged by CPU efforts. Both the baseline hardware and workloads examined on these machines are fundamentally different. The sheer number of threads in modern massively multithreaded processors opens up a new design space in the way they these threads are scheduled. As a result of this multithreading, the latency tolerance of GPUs adds an additional dimension to the optimization problem. GPU designs also have to account for thread aggregation in the form of warps. Finally, the homogeneity of the exposed parallelism in GPU languages like CUDA and OpenCL present new opportunities for hardware optimizations.

Thekkath and Eggers [157] examine the effectiveness of multiple hardware contexts on multithreaded CPU and CMP designs. They find that increasing the number of contexts on a single CPU core has a limited benefit, since cache conflict misses and network contention are both increased as the number of hardware contexts increases. They also demonstrate that the performance of less optimized code degrades quickly as the number of hardware contexts increases. They determine that this is primarily due to cache contention. This finding that less-optimized applications perform best with relatively few contexts is consistent with our observation that less optimized GPU code benefits from constrained multithreading.

OS level thread scheduling has been studied as a way to increase cache performance in uniprocessor systems [126] by dynamically launching threads that execute independent iterations of loops in sequential code. There is a body work studying the effects of fine grained and simultaneous multithreading (SMT) [160, 161] in CPUs. SMT differs from FGMT in that it allows instructions from multiple threads to issue on the same cycle (as opposed to on adjacent cycles) on a superscalar CPU. Work on SMT scheduling has focused on thread scheduling at the coarse grained OS level that use runtime monitors to determine which threads should be co-scheduled on the same core. Work on symbiotic SMT job scheduling [150, 151] attempts to identify the affinity of threads in a SMT machine and ensure that threads that perform well together are assigned to the same core.

Suleman et al. [155] examine a feedback-driven technique to reduce multithreading when data synchronization and off-chip memory bandwidth become performance limiting factors. Their technique seeks to reduce both the execution time and energy consumed in CMPs by throttling multithreading in the threading system software based on an analytical model constructed from available hardware performance counters.

Concurrent to our work, Jaleel et al. [74] propose the CRUISE scheme which uses LLC utility information to make high level scheduling decisions in multiprogrammed CMPs. Our work focuses on the first level cache in a massively multithreaded environment and is applied at a much finer grain. Scheduling decisions made by CRUISE tie programs to cores, where CCWS and DAWS make issue level decisions on which bundle of threads should enter the execution pipeline next.

7.1.5 Cache Capacity Management

There is a body of work attempting to increase cache hit rate by improving the replacement or insertion policy [20, 35, 73, 77, 114, 130, 172]. All these attempt to exploit different heuristics of program behavior to predict a block's re-reference interval and mirror the Belady-optimal [26] policy as closely as possible. While CCWS and DAWS also attempt to maximize cache efficiency, they do so by shortening the re-reference interval rather than by predicting it. CCWS and DAWS balance the shortening of the re-reference interval, achieved by limiting the num-

ber of eligible warps, while maintaining sufficient multithreading to cover most of the memory and operation latencies. Further multithreaded CPU work attempts to manage interference among heterogeneous workloads [72, 129], while each thread in our workload has roughly similar characteristics.

Software-based tiling techniques accomplished by the programmer or compiler [9, 15, 42, 56, 92] have been shown to efficiently exploit locality in both uniprocessor and multiprocessor systems. The runtime scheduling performed by CCWS and DAWS could be considered a dynamic form of tiling a multithreaded problem by constraining which subset of the problem is operated on at any given time.

The are several works evaluating cache-conscious data placement [33, 41] as a means of increasing cache hit rate. Our work on CCWS and DAWS does not change the placement of data in memory but instead focuses on exploiting locality that already exists in the data structures used by the program.

Agrawal et el. [12] present theoretical cache miss limits when scheduling streaming applications represented as directed graphs on uniprocessors. Their work shows that scheduling the graph by selecting partitions comes within a constant factor of the optimal scheduler when heuristics such as working set and data usage rates are known in advance.

7.1.6 Locality Detection

Pomerene et al. [128] make use of a "shadow directory" to improve cache prefetching by storing address pairs. They store a parent address, along with a descendant address, where an access to the descendent was observed after the last access to the parent. They then fetch data for the descendent once an access to the parent is re-observed. By using the shadow directory, they can track information for more parent blocks than can be stored in the cache.

Johnson and Hwu [81] employ a locality detection mechanism to characterize large regions or "macroblocks" of memory based on per-marcoblock accesses counters. The values of these access counters are used to influence the cache management policy and helps prevent cache thrashing and pollution, by prioritizing the storage of more frequently accessed macroblocks.

A number of previous works in CPU caching have used spatial locality detection to improve the utilization of data fetched from main memory and avoid wasting cache space on data that is never accessed [37, 89, 131]. Theses techniques to detect spatial locality are orthogonal to our locality detection technique which focuses on temporal cache line reuse. An interesting extension to CCWS and DAWS would be to examine how these techniques to capture spatial locality interact with our scheduling techniques.

Beckmann et al. [25] use victim tag information to detect locality lost due to excessive replication in the cache hierarchy and adapt the replication level accordingly. The lost locality detector in CCWS and DAWS differs from their technique in that it subdivides the victim tag array by warp ID and makes use of the this information to influence thread scheduling.

Seshadri et al. [146] also make use of cache replacement victim addresses in their Evicted Address Filter (EAF) to detect blocks that high locality. The EAF uses a periodically cleared bloom filter to classify the reuse characteristics of individual blocks in a cache. They capitalize on the observation that blocks with high reuse which are evicted from the cache prematurely due to cache pollution or thrashing are typically reused very soon after their eviction and they use this information to inform their cache insertion policy. It would be an interesting extension to CCWS and DAWS to replace the victim tag array with the bloom-filter used in the EAF to cut down on the hardware cost and energy consumption associated with victim tag array's storage of complete tags and associative lookup.

7.2 Branch and Memory Divergence Mitigation

Traditional vector processors have examined the concept of executing conditional SIMD code with the use of predicated instructions [29]. Predicated instructions are used on contemporary GPUs for simplistic, short branches. Recent work on compiler managed SIMT execution has explored a compiler-driven software only solution to the branch divergence problem [101]. Work on software-based branch divergence mitigation is orthogonal to our work on variable warp sizing and could be used in combination with this approach.

Previous work has explored architectures that attempt to achieve performance

on divergent code that is similar to what would be achieved with narrower warps. Fung, et al. [53, 55] and Rhu, et al. [132] explore techniques to "repack" threads from different divergent warps (but which share the same control-flow path) into a single warp issue slot. Narasiman, et al. [123] describe a large-warp microarchitecture that effectively performs a similar repacking of threads to mitigate control-flow divergence costs.

Other architectures have been proposed which support narrow SIMT execution, but lack the ganging features of I-VWS [85, 88, 100]. In contrast to their work, I-VWS studies the effects of enabling true 8-way execution on GPU workloads and focuses on a hierarchical gang scheduler that attempts to dynamically set the warp size. Rhu and Erez [134] evaluate SIMD Lane Permutation which attempts to mitigate control flow divergence by statically re-arranging how threads are assigned to warps. Trajan et al. [156] propose the diverge on miss technique which intentionally decreases SIMD utilization when threads in a warp experience memory latency divergence to increase GPU memory latency tolerance. Meng et al. [115] present Dynamic Warp Subdivision (DWS) which focuses on enabling different control-flow paths to be independently scheduled (as opposed to being strictly managed by a reconvergence stack-enforced ordering, as in conventional GPU architectures). DWS does pack threads from different warps together for simultaneous issue, but, rather, focuses on increasing memory-level parallelism by allowing memory-related stalls from both paths of a branch to be overlapped in time. This basic idea is extended by Rhu and Erez [133] who propose allowing simultaneous issue of two control flow paths in the same cycle. ElTantawy et al. [48] further generalize this approach to enable simultaneous co-issue from many independent control-flow paths.

DWS also proposed splitting warps based on memory-latency divergence, and, like the I-VWS-ImpatientMem and I-VWS-GroupMem gang-splitting policies we studied, found some workloads suffered due to the lack of subsequent reconvergence.

Our variable warp sizing approach shares with these proposals the similar goal of performing well on divergent code, while running on an underlying wide-SIMD architecture. The above proposals take increasingly complex steps to make a wide SIMD machine to have performance similar to a narrow-SIMD (or, in the limit,

MIMD) processor when executing divergent code.

Meng et al. [117] describe an approach called Robust SIMD that determines whether an application would be best served with a given SIMD width. This scheme can slice warps into independently schedulable entities, providing more, narrower warps as needed. Unlike our variable warp sizing scheme, this approach does not exploit the available datapath hardware by allowing simultaneous issue of the narrower warp-slices. This approach tends to provide the most benefit to applications that suffer from significant memory address and latency divergence. Enabling a larger number of narrower warps increases memory-level parallelism and latency hiding for these workloads (and eliminates some of the losses due to a wide warp waiting for the longest-latency load result). Wang, et al. [167] describe a "Multiple-SIMD, Multiple Data (MSMD)" architecture that supports flexible-sized warps with multiple simultaneous issue from different control-flow paths. Their approach requires a complex microarchitecture that is quite different than a traditional GPU. Like our variable warp sizing scheme, however, they provide a number of small instruction buffers to mitigate the impact on the front-end of the machine for multiple parallel instruction issues. Lashgar, et al. [93] perform an investigation on the effects of warp size on performance. They also note that some workloads lose performance with small warps, and deduce that it is due to lost memory coalescing opportunities. They compare a small-warp machine with aggressive memory coalescing (similar to our baseline small-warp architecture) to a large-warp machine with simultaneous multi-path execution to mitigate control-flow divergence costs. They determine that an architecture like our small-warp baseline tends to outperform a wide-warp architecture that supports aggressive multi-path execution.

Jablin et al [71] revisit trace scheduling [51] on GPUs. Trace scheduling is a microcoded CPU technique that divides code into traces and attempts to expose ILP across branch boundaries by statically scheduling traces in the compiler. Their warp-aware trace scheduling technique increases statically exposed ILP in GPGPU applications by adapting traditional CPU trace scheduling to attend to SIMT divergence behaviour on GPUs.

Chapter 8

Conclusions and Future Work

This chapter concludes the dissertation and proposes potential future work based on its findings.

8.1 Conclusions

The breakdown in single thread performance scaling in the last decade has left computer architects looking for more drastic innovations to improve computing capability. Massively parallel architecture, like GPUs, are a very real design alternative that can potentially sustain energy-efficient general purpose computing performance moving forward. However, as they exist today, GPUs face several key challenges that hinder their general acceptance as a mainstream computing platform. This dissertation proposes hardware innovations that help solve some of these challenges, in particular: memory locality management, control flow irregularity and general programmability. This dissertation focus on innovations to GPU microarchitecture because the GPU is a concrete example of an implementable, in-use design. However, the questions posed, solutions presented and insights gained in this dissertation could be applied to parallel computer hardware in general and are not specifically tied to GPUs. Even the name GPU is a nod to the machine's legacy (and continued success) in the 3D rendering space. In the coming decades, as the computing landscape continues to shift and evolve, I believe the lessons learned both in this dissertation and others on GPU architecture will have

a much wider impact on computing at large. Indeed one of the goals of this thesis is to enable this continued expansion of the massively parallel computing platform by increasing the generality of the hardware through the innovations proposed. Our case study on the programmability of GPUs highlights how the microarchitectural improvements we propose can make the massively parallel platform more approachable for programmers. The techniques in this thesis help isolate the application developer from hardware-specific details, making their code more portable and removing the need to obfuscate application code with GPU-specific optimizations.

This dissertation proposed three novel microarchitectural enhancements to contemporary GPUs that improve performance and expand the class of applications that can take advantage of massively parallel acceleration. The first two innovations, cache-conscious warp scheduling (CCWS) and divergence-aware warp scheduling (DAWS), are techniques aimed at exploiting vertical memory reference locality. The third proposes a variable warp-size architecture (VWS) which slices the GPU datapath to improve performance in the presence of control flow divergence and uses a ganged scheduling mechanism to recapture horizontal memory reference locality and instruction fetch locality.

CCWS (Chapter 3) exploits the observation that locality in cache-sensitive GPGPU applications tends to occur vertically within a warp. To capitalize on this observation, CCWS uses a novel lost locality detector to drive a reactionary warp-throttling mechanism when over-subscription of the on-chip data caches is detected. Simulated evaluations using CCWS demonstrate a 63% performance improvement on a suite of memory irregular, highly cache-sensitive workloads. This work goes on to contrast the warp scheduling problem with the more traditional CPU technique of cache management which involves innovations to the cache replacement policy. By comparing against an optimal, oracle cache replacement policy, CCWS demonstrates that innovations to the low-level hardware thread scheduler can have a more significant performance impact than any change to the replacement policy using an inferior thread scheduler. Since the publication of CCWS, a number of other researchers have studied warp scheduling in other contexts, confirmed our observations, built upon CCWS and proposed alternatives to CCWS [8, 18, 34, 38, 39, 76, 78–80, 83, 84, 86, 90, 95, 97, 98, 103–

105, 111, 120, 124, 125, 127, 135, 147, 148, 153, 159, 164–166, 174–180] among others. Many of these works also confirm that the limited set economically important, irregular applications we studied in CCWS were an accurate representation of a class of forward-looking GPU workloads, as the characteristics we observed in those applications have been observed in a number of other workloads.

DAWS (Chapter 4) quantifies the relationship between branch divergence, memory divergence and locality in a suite highly cache-sensitive, irregular GPU workloads. DAWS builds on the insights of CCWS by using an online characterization of locality in kernel code sections combined with runtime information about the level of control flow divergence experienced by warps. It uses this information to create a cache footprint prediction for each warp in the GPU that evolves as warps experience control flow divergence and move into new code sections. Using this footprint, DAWS is able to create more accurate cache usage estimates than CCWS and further improve performance by 26%. Chapter 5 goes on to demonstrate the effect both CCWS and DAWS have on GPU programmability. A case study of an irregular GPU application demonstrates that hardware thread scheduling enables less optimized code to perform within 4% of GPU-optimized code without any programmer input.

Finally, Chapter 6 presents VWS which studies the affect GPU warp sizing has on locality, performance and instruction fetch overheads. VWS proposes a GPU microarchitecture that is able to dynamically adjust its warp size based on the application. VWS enables the efficient execution of highly control flow diverged workloads by enabling the GPU to execute with a more narrow warp size, while maintaining the horizontal memory locality efficiencies and fetch amortizations of a wide warp when appropriate. VWS demonstrates a 35% performance improvement on a set of control flow irregular applications while maintaining performance and energy-efficiency on regular workloads.

8.2 Future Directions

This section details some potential directions for future work based upon the work in this dissertation.

8.2.1 Capturing Locality in a Variable Warp-Size Architecture

The two vertical locality capturing techniques proposed in this thesis (CCWS and DAWS) both operate on an SM with a fixed warp size. I anticipate that there are a number of interesting research questions that will arise if we attempt to combine locality preservation with a variable warp-sized machine. The following subsections describe some anticipated opportunities and challenges that would come from combining CCWS and/or DAWS with a Variable Warp-Size Architecture.

Increased Scheduling Flexibility

A machine that is capable of executing with a smaller warp size means fewer threads are controlled with each issue-level scheduling decision. Therefore, the granularity of locality that can be captured is increased. In code without any branch divergence, both CCWS and DAWS make a decision of 32-threads at a time, if this number is decreased to 4, there is an opportunity to refine the cache-footprint prediction for each of the 4-wide slices further and potentially increase multithreading when the cache is underutilized in the current design. A simple way to implement a cache-conscious VWS machine might be to simply turn off slices in the presence of high intra-warp locality code and have the slices re-gang after the execution of their high-locality code sections. Some challenge imposed by this type of design would be coordinating these individual schedulers and determining when it might be best to form gangs based on predicted cache locality.

Transformation of Intra-Warp locality to Inter-Warp Locality

Another interesting side effect of having smaller warps might be that a non-trivial portion of the intra-warp locality we observed in CCWS and DAWS might turn into inter-warp locality with a smaller warp size. This would further motivate a mechanism that can capture inter-warp locality in the scheduler and would create a tradeoff between between breaking gangs to improve SIMD utilization and keeping gangs together to maintain the intra-warp locality CCWS and DAWS are designed to exploit.

Increased Opportunities to Exploit Memory Latency Divergence

Chapter 6 showed that there was very limited opportunity to exploit memory latency divergence in VWS as it was presented. However, this lack of improved performance when gangs were split on memory divergence might have come from excessive cache trashing that could be mitigated by applying CCWS and/or DAWS to the split gangs. Memory latency divergence might be a good indicator of when multithreading should start being constrained, as it indicates that only a subset of the threads in the gang are capturing locality in the data caches.

Combining VWS and CCWS and/or DAWS could expose interesting opportunities in the inter-play of branch divergence and memory locality. In code that is both branch divergent and has data locality, when is it best to execute multiple control flow path simultaneously and when is it best to limit the number of exposed control flow paths in the interest of loading less data into the caches.

8.2.2 Exploiting Shared (Inter-Warp) Locality

Neither CCWS nor DAWS explicitly exploited sharing patterns among warp, i.e. the inter-warp locality identified in that work. This work would involve performing an initial examination of global data sharing patterns in benchmarks where inter-warp locality is important. To illustrate the potential of capturing this locality, consider the BFS application, whose inner kernel loop is shown in Example 2.

Example 2 BFS CUDA kernel inner loop where inter-warp sharing occurs.

```
1:  for(int i=0; i<node_degree; i++) {
2:      int id = g_graph_edges[first_edge + i];
3:      if(!g_graph_visited[id]) {
4:          g_cost[id]=g_cost[tid]+1;
5:          g_updating_graph_mask[id]=true;
6:      }
7:  }
```

BFS is partitioned by assigning each thread in the program to a node. Each thread loops through all the edges connecting that node to its neighbors. If the neighbor has not yet been visited, the depth for that node is incremented. Initial investigations indicate that the most divergent load in the program is also the load

with the most inter-warp sharing. Line three in the above example is the source line responsible for it. Memory divergence occurs at this line because the index into the array is data dependent. Line two loads the index from memory and effectively acts like a pointer into the array at line three. Inter-warp sharing occurs here because nodes in the same graph can share neighbours. The project could use this information to help make decisions on which warps/threads should be co-scheduled. Since a thread will know which portion of the array it will index prior to issuing the load on line three, it may be beneficial to co-schedule warps with similar indexes to maximize captured locality.

Additionally, there may be benefit in capturing the inter-warp locality that is inherent in the I-cache access stream (since warps in the same kernel share the same code). There may be a very interesting trade-off in exploiting both intra-warp locality in the data access stream and inter-warp locality in the instruction access stream.

8.2.3 Adaptive Cache Blocking and Warp Scheduling

CCWS, DAWS and the work proposed in Section 8.2.2 do little for programs that have been optimized to use the architecture's in-core scratchpad memory (or local data store). This project attempts to solve the dichotomy that exists between user controlled shared scratch-pad memory and L1 data caches. L1D caches in current GPUs are similar in design to a CPU cache even though a CPU runs significantly less threads than a GPU. Programmers are encouraged to make use of shared scratchpad memory to capture locality and data reuse among threads. An observation of how shared memory is used on current GPU programs reveals two important characteristics:

1. Programmers partition their launched CTAs to ensure their shared-memory footprint can fit in the shared on chip memory.

2. Programmers explicitly schedule their CTA's use of this memory using barrier instructions.

The code in Example 3 attempts to capture a generalized example of this. Lines one through six load data from global memory into the scratchpad. The amount of

scratchpad memory used is set at compile time or before the code is launched to the GPU (the SIZE variable on line two). After all data is loaded into the scratchpad, a barrier operation is used on line six to ensure data from all the threads in this CTA is loaded before proceeding. Lines seven through eleven perform an iterative calculation on each thread based on data in shared memory. The data in the scratchpad is accessed repeatedly in both an intra- and inter-warp fashion. Some benchmarks employ synchronization in this step to enable true data sharing between threads in the same CTA. After this computation is done, the program writes the end result (which can be stored in either local variables or the scratchpad) out to global memory. The kernel may repeat this process multiple times depending on it's nature, each time explicitly blocking all the data required for an internal loop or high locality access stream into the scratchpad.

Example 3 Generalized CUDA code sample that uses shared scratchpad memory

```
1:  // The first code section loads data into the shared scrachpad memory
2:  extern __shared__ float scratchpad_mem[SIZE];
3:  for(int i=0; i<size; i++)
4:      scratchpad_mem[f(i,tid)] = global_mem[f(i,tid)];
6:  __syncthreads();
7:  // All the data going to be re-used is in the scratchpad memory
8:  // Iterative computation is done on the scratchpad memory
9:  float local_variable = 0.0;
10: for(int i=0; i<iterations; i++)
11:     local_variable += scratchpad_mem[f(i,tid)] + ...;
12: g_out[f(tid)] = local_variable;
```

There are several problems with this type of approach:

1. It forces programmers to explicitly partition and schedule their application which complicates the programming process.

2. This exercise is more difficult or impossible if the locality in the application is input data dependent. When writing the code, the programmer does not know how large to make the application's CTAs and how much data each thread will need.

3. The application will require tuning if used on an architecture with different

restrictions on CTA size or usable scratchpad memory space.

4. Programmer scheduling is applied at fairly coarse grain. With a more adaptive scheduling and replacement technique old data could be evicted sooner to make way for new data without the need for coarse grained synchronization points. This could help overlap computation from some warps with memory accesses from others.

The premise of this project is that a more ideal situation would see the programmer write the same kernel accessing global memory directly as done in Example 4. A hardware/compiler solution would schedule the threads and manage the cache under the hood to capture the locality.

Example 4 Generalized code sample without shared memory

```
1:  float local_variable = 0.0;
2:  for(int i=0; i<iterations; i++)
3:      local_variable += global_mem[f(i, tid)] + ...;
4:  g_out[f(tid)] = local_variable;
```

This problem might be solved by unifying the scheduling system and the cache replacement policy to provide a finer grained solution than the programmer could have achieved through explicit scratchpad use. The steps to solve this problem will require some work to be done either at the compiler stage or by high level hints given by the programmer. The first step in this process would involve flagging pieces of code where significant locality may exist. The most obvious candidates are inner loops. If the programmer was able to statically arrange the accesses to shared memory as they did in Example 3, then a static analysis algorithm should be able to identify which threads will share data and an analysis of loop bounds can help quantify how much data will be loaded. Doing this pre-processing in the compiler is preferred, since it eases the burden on the programmer, however even if some slight annotation is required it can still help mitigate problems two through four outlined above. If the loop bounds or accesses patterns are input data dependent, then the programmer would not have been able to capture this in locality in a scratchpad memory implementation. In this case live working set

analysis could be used (similar to the locality detection implemented in CCWS) to predict when data should be evicted or warps should be prioritized. The scheduling and replacement policy can be driven by a prioritization system similar to the point system proposed in CCWS. High priority warps will receive preference from the warp scheduler and the their data will be protected from eviction. The scoring system can bias groups of threads to be co-scheduled if they share data. In addition to the replacement scheme protecting data, after this data is used as many times as static or live working set analysis predicts, the replacement policy can de-prioritize the data and create capacity for other threads.

8.2.4 A Programmable Warp Scheduler for Debugging and Synchronization

In contemporary GPUS, the software stack has very little control over how warps are scheduled on an SM. The application programmer can dictate, at a course gain, when threads within a CTA must synchronize using CTA barriers. However, this level of control is not fine-grained enough to enable tools like race detectors [144] to formally verify if a GPU program is race free. This project would involve exposing a low-level warp scheduler API to software, enabling debuggers, race detectors and other tools to have more fine-grained control of the warp schedule to help ease the GPU debugging process. Additionally, a programmable warp scheduler might enable expert programmers or systems developers to create more efficient synchronization primitives on the GPU by dictating the warp schedule.

Bibliography

[1] Intel Cilk Plus Manual. https://www.cilkplus.org/. Accessed July 6, 2015. → pages 2

[2] Intel Thread Building Blocks Manual. https://www.threadingbuildingblocks.org/. Accessed July 6, 2015. → pages 2

[3] Diamond Standard 108Mini Controller: A Small, Low-Power, Cache-less RISC CPU. http://ip.cadence.com/uploads/pdf/108Mini.pdf. Accessed July 6, 2015. → pages 106

[4] Intel 4004 Datasheet. http://www.intel.com/Assets/PDF/DataSheet/4004_datasheet.pdf, 1987. Accessed July 6, 2015. → pages 1

[5] NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf , 2009. Accessed July 6, 2015. → pages 14, 16

[6] NVIDIA CUDA C Programming Guide v4.2, 2012. → pages 2, 5, 11

[7] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK-110. http://www.nvidia.ca/content/PDF/kepler/ NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. Accessed July 6, 2015. → pages 14

[8] M. Abdel-Majeed, D. Wong, and M. Annavaram. Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 111–122, 2013. → pages 123

[9] W. Abu-Sufah, D. Kuck, and D. Lawrie. Automatic Program Transformations for Virtual Memory Computers. *In Proceedings of the 1979 National Computer Conference*, 1979. → pages 118

[10] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 1992. → pages 116

[11] O. Agesen, D. Detlefs, and J. E. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 269–279, 1998. → pages 30

[12] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo. Cache-Conscious Scheduling of Streaming Applications. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 236–245, 2012. → pages 118

[13] T. Aila and S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Eurographics/ACM SIGGRAPH High Performance Graphics conference (HPG)*, 2009. → pages 107

[14] AMD. Compute cores white paper. https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, 2014. Accessed July 6, 2015. → pages 12

[15] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1993. → pages 118

[16] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. Accessed July 6, 2015. → pages 1

[17] R. Ausavarungnirun, S. Ghose, O. Kayran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015. → pages 115

[18] M. Awatramani, J. Zambreno, and D. Rover. Perf-Sat: Runtime Detection of Performance Saturation for GPGPU Applications. In *Proceedings of 43rd International Conference on Parallel Processing Workshops (ICCPW)*, pages 1–8, 2014. → pages 123

[19] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, 2009. → pages 29, 31, 63, 65, 107, 111

[20] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 187–200. USENIX Association, 2004. → pages 117

[21] K. Barabash and E. Petrank. Tracing Garbage Collection on Highly Parallel Platforms. In *International Symposium on Memory Management (ISMM)*, pages 1–10, 2010. → pages 30, 31, 65

[22] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994. → pages 22, 47

[23] J. Baxter. *Open Source Hardware Development and the OpenRISC Project*. PhD thesis, KTH Computer Science and Communication, 2011. → pages 106

[24] E. Baydal, P. Lopez, and J. Duato. A Simple and Efficient Mechanism to Prevent Saturation in Wormhole Networks. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 617–622, 2000. → pages 112

[25] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 443–454, 2006. → pages 5, 119

[26] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78 –101, 1966. → pages 29, 117

[27] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2009. → pages 74

[28] B. Boothe and A. Ranade. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 214–223, 1992. → pages 116

[29] W. Bouknight et al. The Illiac IV System. *Proceedings of the IEEE*, 60(4): 369 – 388, apr. 1972. → pages 119

[30] N. Brunie, S. Collange, and G. Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 49–60, June 2012. → pages 82

[31] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-Based Barnes Hut N-Body Algorithm. In W. Hwu, editor, *GPU Computing Gems, Emerald Edition*, pages 75–92. Elsevier, 2011. → pages 80

[32] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 141–151, November 2012. → pages 80

[33] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 1998. → pages 118

[34] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 128–139, 2014. → pages 123

[35] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 401–412, 2009. → pages 117

[36] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. → pages 29, 31, 65, 107

[37] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and Complexity-Effective Spatial Pattern Prediction. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2004. → pages 119

[38] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 343–355, 2014. → pages 123

[39] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W.-M. W. Hwu. Adaptive Cache Bypass and Insertion for Many-core Accelerators. In *Proceedings of International Workshop on Manycore Embedded Systems*, pages 1:1–1:8, 2014. → pages 123

[40] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang. Memory Latency Reduction via Thread Throttling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 53–64, 2010. → pages 112

[41] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999. → pages 118

[42] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 279–290, 1995. → pages 118

[43] W. J. Dally. The Last Classical Computer. *Information Science and Technology (ISAT) Study Group*, 2001. → pages 2

[44] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, 2010. → pages 49, 63, 65, 73

[45] R. H. Dennard, F. H. Gaensslen, and K. Mai. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. In *IEEE Journal of Solid-State Circuits*, October 1974. → pages 1

[46] J. Dongarra. *Performance of Various Computers Using Standard Linear Equations Software*. University of Tennessee Computer Science Technical Report Number, 2015. → pages 3

[47] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 335–346, 2010. → pages 112

[48] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt. A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 248–259, February 2014. → pages 82, 106, 107, 109, 120

[49] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011. → pages 2

[50] Feng, W. and Cameron K. The Green 500 List. http://http://www.green500.org/. Accessed July 6, 2015. → pages 3

[51] J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *Computers, IEEE Transactions on*, (7):478–490, July 1981. → pages 121

[52] W. Fung and T. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 25 –36, 2011. → pages 113

[53] W. Fung and T. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 25–36, February 2011. → pages 82, 107, 109, 120

[54] W. W. L. Fung. Thread Block Compaction Simulation Infrastructure. http://www.ece.ubc.ca/~wwlfung/code/tbc-gpgpusim.tgz, 2012. Accessed July 6, 2015. → pages 107, 108

[55] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 407–420, 2007. → pages 15, 30, 64, 82, 107, 109, 113, 120

[56] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988. → pages 118

[57] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 235–246, 2011. → pages 92, 113

[58] H. Goldstine and A. Goldstine. The Electronic Numerical Integrator and Computer (ENIAC). *Annals of the History of Computing, IEEE*, (1):10–16, 1996. → pages 1, 4

[59] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. Many-Core vs. Many-Thread Machines: Stay Away From the Valley. *Computer Architecture Letters*, pages 25 –28, jan. 2009. → pages 112

[60] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 108–120, 1997. → pages 12

[61] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE*, 32(2):48 –60, march-april 2012. → pages 6

[62] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 88 –98, 2012. → pages 19, 29, 31, 49, 65

[63] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 267–276, 2011. → pages 48

[64] W.-m. W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. → pages 3

[65] IBM Corp. IBM Automatic Sequence Controlled Calculator, 1945. → pages 1, 4

[66] IDC. Worldwide Server Market Rebounds Sharply in Fourth Quarter as Demand for Blades and x86 Systems Leads the Way, Feb 2010. → pages 29

[67] IDC. HPC Server Market Declined 11.6% in 2009, Return to Growth Expected in 2010, Mar 2010. → pages 29

[68] IEEE. The OpenMP API Specification for Parallel Programming. http://openmp.org/wp. Accessed July 6, 2015. → pages 2

[69] IEEE. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language), 1996. → pages 2

[70] *Intel Xeon Phi Coprocessor Brief*. Intel. → pages 6

[71] J. A. Jablin, T. B. Jablin, O. Mutlu, and M. Herlihy. Warp-aware Trace Scheduling for GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 163–174, 2014. → pages 121

[72] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 208–219, 2008. → pages 118

[73] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 60–71, 2010. → pages 20, 117

[74] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. CRUISE: Cache Replacement and Utility-Aware Scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 249–260, 2012. → pages 117

[75] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and Improving the use of Demand-Fetched Caches in GPUs. In *Proceedings of the ACM international conference on Supercomputing*, pages 15–24, 2012. → pages 115

[76] W. Jia, K. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 272–283, 2014. → pages 123

[77] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM's Special Interest Group on Measurement and Evaluation (SIGMETRICS)*, pages 31–42, 2002. → pages 117

[78] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013. → pages 114, 123

[79] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013. → pages 114

[80] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-Aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, pages 1:1–1:8, 2014. → pages 123

[81] T. L. Johnson and W.-m. W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 315–326, 1997. → pages 118

[82] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990. → pages 26

[83] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013. → pages 114, 123

[84] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. Managing GPU Concurrency in Heterogeneous Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 114–126, 2014. → pages 123

[85] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, September/October 2011. → pages 120

[86] M. Khairy, M. Zahran, and A. G. Wassal. Efficient Utilization of GPGPU Cache Hierarchy. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU)*, pages 36–47, 2015. → pages 123

[87] Khronos Group. OpenCL. http://www.khronos.org/opencl/. Accessed July 6, 2015. → pages 2, 11

[88] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 52–63, 2004. → pages 120

[89] S. Kumar and C. Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 357–368, 1998. → pages 119

[90] H.-K. Kuo, B.-C. C. Lai, and J.-Y. Jou. Reducing Contention in Shared Last-Level Cache for Throughput Processors. *ACM Transactions on Design Automation of Electronic Systems*, 20:12:1–12:28, 2014. → pages 123

[91] N. B. Lakshminarayana and H. Kim. Effect of Instruction Fetch and Memory Scheduling on GPU Performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010. → pages 113

[92] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 63–74, 1991. → pages 118

[93] A. Lashgar, A. Baniasadi, and A. Khonsari. Towards Green GPUs: Warp Size Impact Analysis. In *International Green Computing Conference (IGCC)*, pages 1–6, June 2013. → pages 121

[94] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 213–224, 2010. → pages 114, 115

[95] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving GPGPU Resource Utilization through Alternative Thread Block Scheduling. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 260–271, 2014. → pages 123

[96] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 101–110, 2009. → pages 17

[97] S.-Y. Lee and C.-J. Wu. CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 175–186, 2014. → pages 123

[98] S.-Y. Lee, A. Arunkumar, and C.-J. Wu. CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 515–527, 2015. → pages 123

[99] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM. → pages 3

[100] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 129–140, June 2011. → pages 120

[101] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanović. Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 101–113, 2014. → pages 119

[102] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013. → pages 71

[103] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou. Understanding the Tradeoffs Between Software-Managed vs. Hardware-Managed Caches in GPUs. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 231–242, 2014. → pages 123

[104] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015. → pages

[105] D. Li, M. Rhu, D. Johnson, M. O'Connor, M. Erez, D. Burger, D. Fussell, and S. Redder. Priority-Based Cache Allocation in Throughput Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 89–100, 2015. → pages 116, 124

[106] P. Lopez, J. M. Martnez, J. Duato, and F. Petrini. On the Reduction of Deadlock Frequency by Limiting Message Injection in Wormhole Networks. In *In Proceedings of Parallel Computer Routing and Communication Workshop*, 1997. → pages 112

[107] P. Lopez, J. Martinez, and J. Duato. DRIL: Dynamically Reduced Message Injection Limitation Mechanism for Wormhole Networks. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 535–542, 1998. → pages 112

[108] M. Maas, P. Reames, J. Morlan, K. Asanović, A. D. Joseph, and J. Kubiatowicz. How a Single Chip Causes Massive Power Bills GPUSimPow: A GPGPU Power Simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013. → pages 71

[109] A. Mahesri. *Tradeoffs in Designing Massively Parallel Accelerator Architectures*. PhD thesis, University of Illinois at Urbana-Champaign, 2009. → pages 106

[110] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. Tradeoffs in Designing Accelerator Architectures for Visual Computing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 164–175, November 2008. → pages 106, 107

[111] M. Mao, J. Hu, Y. Chen, and H. Li. VWS: A Versatile Warp Scheduler for Exploring Diverse Cache Localities of GPGPU Applications. In *Proceedings of the Design Automation Conference (DAC)*, pages 83:1–83:6, 2015. → pages 124

[112] M. R. Marty and M. D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 309–320, 2006. → pages 5

[113] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A GPU Implementation of Inclusion-based Points-to Analysis. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 107–116, August 2012. → pages 80

[114] J. Meng and K. Skadron. Avoiding Cache Thrashing due to Private Data Placement in Last-Level Cache for Manycore Scaling. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 282–288, 2009. → pages 117

[115] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 235–246, 2010. → pages 82, 90, 107, 108, 109, 120

[116] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 235–246, 2010. → pages 114

[117] J. Meng, J. Sheaffer, and K. Skadron. Robust SIMD: Dynamically Adapted SIMD Width and Multi-Threading Depth. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 107–118, May 2012. → pages 121

[118] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 117–128, August 2012. → pages 80

[119] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 117–128, 2012. → pages 19

[120] S. Mittal. A Survey of Techniques for Managing and Leveraging Caches in GPUs. *Journal of Circuits, Systems and Computers*, 23(08), 2014. → pages 124

[121] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965. → pages 1

[122] S. Naffziger, J. Warnock, and H. Knapp. When Processors Hit the Power Wall (or "When the CPU hits the fan"). In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, 2005. → pages 1

[123] 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 *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 308–317, December 2011. → pages 18, 34, 82, 107, 109, 113, 114, 120

[124] C. Nugteren, G.-J. van den Braak, and H. Corporaal. A Study of the Potential of Locality-Aware Thread Scheduling for GPUs. In *Euro-Par 2014: Parallel Processing Workshops*, Lecture Notes in Computer Science, pages 146–157. 2014. → pages 124

[125] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 37–48, 2014. → pages 124

[126] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread Scheduling for Cache Locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 1996. → pages 117

[127] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the*

International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS), pages 743–758, 2014. → pages 124

[128] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio. US Patent #4,807,110: Prefetching system for a cache having a second directory for sequentially accessed blocks, Feb. 21 1989. → pages 118

[129] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 423–432, 2006. → pages 118

[130] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 381–391, 2007. → pages 5, 117

[131] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 250–259, 2007. → pages 119

[132] M. Rhu and M. Erez. CAPRI: Prediction of Compaction-adequacy for Handling Control-divergence in GPGPU Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 61–71, June 2012. → pages 82, 107, 120

[133] M. Rhu and M. Erez. The Dual-Path Execution Model for Efficient GPU Control Flow. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 235–246, February 2013. → pages 107, 109, 120

[134] M. Rhu and M. Erez. Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 356–367, June 2013. → pages 82, 107, 120

[135] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 86–98, 2013. → pages 124

[136] T. G. Rogers. CCWS Simulation Infrastructure. http://www.ece.ubc.ca/~tgrogers/ccws.html, 2013. Accessed July 6, 2015. → pages 63

[137] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012. → pages iv, 9, 48, 49, 52, 54, 62, 64, 65, 66, 71, 72, 92

[138] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Thread Scheduling for Massively Multithreaded Processors. *IEEE Micro, Special Issue: Micro's Top Picks from 2012 Computer Architecture Conferences*, 2013. → pages iv

[139] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-Aware Warp Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013. → pages iv

[140] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Learning Your Limit: Managing Massively Multithreaded Caches Through Scheduling. *Communications of the ACM*, December 2014. → pages iv

[141] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler. A Variable Warp Size Architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015. → pages iv

[142] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere. An Experimental Study on Performance Portability of OpenCL Kernels. In *Application Accelerators in High Performance Computing*, 2010. → pages 5

[143] R. Saavedra-Barrera, D. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990. → pages 116

[144] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, pages 391–411, 1997. → pages 130

[145] S. L. Scott and G. S. Sohi. The Use of Feedback in Multiprocessors and Its Application to Tree Saturation Control. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):385–398, 1990. → pages 112

[146] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The Evicted-address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 355–366, 2012. → pages 119

[147] A. Sethia and S. Mahlke. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 647–658, 2014. → pages 124

[148] A. Sethia, D. Jamshidi, and S. Mahlke. Mascar: Speeding up GPU Warps by Reducing Memory Pitstops. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 174–185, 2015. → pages 114, 124

[149] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja. Sparc T4: A Dynamically Threaded Server-on-a-Chip. *Micro, IEEE*, 32(2):8 –19, march-april 2012. → pages 6

[150] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 234–244, 2000. → pages 117

[151] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 66–76, 2002. → pages 117

[152] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995. → pages 48

[153] S. Song, M. Lee, J. Kim, W. Seo, Y. Cho, and S. Ryu. Energy-Efficient Scheduling for Memory-Intensive GPGPU workloads. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, 2014. → pages 124

[154] D. Spoonhower, G. Blelloch, and R. Harper. Using Page Residency to Balance Tradeoffs in Tracing Garbage Collection. In *Proceedings of*

International Conference on Virtual Execution Environments (VEE 2005), pages 57–67. → pages 30, 31, 65

[155] Suleman, M. Aater and Qureshi, Moinuddin K. and Patt, Yale N. Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 277–286, 2008. → pages 117

[156] D. Tarjan, J. Meng, and K. Skadron. Increasing Memory Miss Tolerance for SIMD Cores. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, November 2009. → pages 107, 120

[157] R. Thekkath and S. J. Eggers. The Effectiveness of Multiple Hardware Contexts. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 328–337, 1994. → pages 116

[158] M. Thottethodi, A. Lebeck, and S. Mukherjee. Self-tuned congestion control for multiprocessor networks. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 107–118, 2001. → pages 112

[159] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez. Adaptive GPU Cache Bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, pages 25–35, 2015. → pages 124

[160] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 392–403, 1995. → pages 117

[161] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 191–202, 1996. → pages 117

[162] A. Turing. On Computable Numbers, with an Application to the Entscheidungs problem. *Proceedings of the London Mathematical Society*, 42:230, 1936. → pages 1

[163] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks*, 53(11): 1830–1845, 2009. → pages 30

[164] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu. A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 41–53, 2015. → pages 124

[165] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. Real-world Design and Evaluation of Compiler-managed GPU Redundant Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 73–84, 2014. → pages

[166] B. Wang, Z. Liu, X. Wang, and W. Yu. Eliminating Intra-warp Conflict Misses in GPU. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 689–694, 2015. → pages 124

[167] Y. Wang, S. Chen, J. Wan, J. Meng, K. Zhang, W. Liu, and X. Ning. A Multiple SIMD, Multiple Data (MSMD) Architecture: Parallel Execution of Dynamic and Static SIMD Fragments. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 603–614, February 2013. → pages 121

[168] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 273–280, 1989. → pages 116

[169] M. V. Wilkes. The EDSAC Computer. *Managing Requirements Knowledge, International Workshop on*, page 79, 1951. → pages 1

[170] F. Williams and T. Kilburn. The University of Manchester Computing Machine. *Manchester University Computer Inaugural Conference*, pages 5–11, 1951. → pages 1

[171] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, May 1996. → pages 44, 71, 105

[172] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011. → pages 117

[173] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH computer architecture news*, pages 20–24, 1995. → pages 5

[174] P. Xiang, Y. Yang, and H. Zhou. Warp-level divergence in GPUs: Characterization, Impact, and Mitigation. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 284–295, 2014. → pages 124

[175] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for gpus. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 76–88, 2015. → pages

[176] Q. Xu and M. Annavaram. PATS: Pattern Aware Scheduling and Power Gating for GPGPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 225–236, 2014. → pages

[177] M. K. Yoon, Y. Oh, S. Lee, S. H. Kim, D. Kim, and W. W. Ro. DRAW: investigating benefits of adaptive fetch group size on GPU. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–192, 2015. → pages

[178] Y. Yu, X. He, H. Guo, Y. Wang, and X. Chen. A Credit-Based Load-Balance-Aware CTA Scheduling Optimization Scheme in GPGPU. *International Journal of Parallel Programming*, pages 1–21, 2014. → pages

[179] Y. Yu, W. Xiao, X. He, H. Guo, Y. Wang, and X. Chen. A Stall-Aware Warp Scheduling for Dynamically Optimizing Thread-level Parallelism in GPGPUs. In *Proceedings of the ACM international conference on Supercomputing*, pages 15–24, 2015. → pages

[180] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive Cache and Concurrency Allocation on GPGPUs. *Computer Architecture Letters*, 2014. → pages 124

[181] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 129–142, 2010. → pages 20