View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University
Purdue e-Pubs

Open Access Theses Theses and Dissertations

Spring 2014

Characterizing the Intra-warp Address Distribution and Bandwidth Demands of GPGPUs

Calvin Holic
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_theses

Part of the Computer Engineering Commons

Recommended Citation

Holic, Calvin, "Characterizing the Intra-warp Address Distribution and Bandwidth Demands of GPGPUs" (2014). *Open Access Theses*. 192.
https://docs.lib.purdue.edu/open_access_theses/192

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Graduate School ETD Form 9
(Revised /)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Calvin R. Holic

Entitled
Characterizing the Intra-warp Address Distribution and Bandwidth Demands of GPGPUs

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

MITHUNA S. THOTTETHODI

ANAND RAGHUNATHAN

T. N. VIJAYKUMAR

To the best of my knowledge and as understood by the student in the
C Disclaimer (Graduate School Form), this thesis/dissertation
Purdue University's "Policy on Integrity in Research" and the use of
copyrighted material.

Approved by Major Professor(s): MITHUNA S. THOTTETHODI

Approved by: Michael R. Melloch 04/29/14

Head of the Graduate Program Date

CHARACTERIZING THE INTRA-WARP ADDRESS DISTRIBUTION AND BANDWIDTH DEMANDS OF GPGPUS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Calvin R. Holic

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2014

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

First, I'd like to thank my advisor Dr. Mithuna Thottethodi for all of his guidance and help in navigating graduate school and research. Thanks to my committee members, Dr. T.N. Vijaykumar and Dr. Anand Raghunathan, for their time and feedback.

Second, I'd like to thank the entire SAIG research group for their support and feedback. In particular, thanks to Tim Pritchett and Eric Villaseñor for their guidance and mentorship from day one of grad school.

Last, and most importantly, I'd like to thank my friends here at Purdue that have gotten me through the day-to-day struggles of grad school, and my friends from home that have kept me on track. Manny Cordisco, Nadra Guizani, Héctor Rodríguez, Crystal Cory, Tyler Green, Tanmay Prakash, and everyone else who keeps me sane, my sincere thanks to you for all that you do.

TABLE OF CONTENTS

Page

LIST OF FIGURES vi
ABSTRACT viii
1 INTRODUCTION 1
2 BACKGROUND 5
2.1 Graphics Processing Unit Background 5
2.2 NVIDIA Architecture Background 6
2.2.1 Global Memory 7
2.2.2 Texture Memory 7
2.2.3 Local Memory 8
2.2.4 Constant Memory 8
2.2.5 Shared Memory 8
3 MEMORY ACCESS CHARACTERIZATION 9
3.1 Coalescer Design: The Challenge 9
3.2 Exploring Monotonicity 10
3.3 Bandwidth Characterization 11
4 METHODOLOGY 13
4.1 Simulator 13
4.2 Benchmarks and Data Sets 14
4.2.1 Back Propagation (backprop) 14
4.2.2 Breadth-First Search (bfs) 15
4.2.3 B+ Tree (b+tree) 15
4.2.4 Gaussian Elimination (gaussian) 15
4.2.5 Heart Wall Tracking (heartwall) 16
4.2.6 HotSpot (hotspot) 16

Page

4.2.7 K-means (kmeans) 16
4.2.8 LavaMD (lavaMD) 17
4.2.9 Leukocyte (leukocyte) 17
4.2.10 LU Decomposition (lud) 17
4.2.11 MUMmerGPU (mummergpu) 18
4.2.12 Myocyte (myocyte) 18
4.2.13 k-Nearest Neighbors (nn) 19
4.2.14 Needleman-Wunsch (nw) 19
4.2.15 Particle Filter (particlefilter) 19
4.2.16 PathFinder (pathfinder) 20
4.2.17 Speckle Reducing Anisotropic Diffusion (SRAD)(srad) . . . 20
4.2.18 Streamcluster (streamcluster) 21
4.3 Analysis 21
4.3.1 Monotonicity 21
4.3.2 Bandwidth Demand 22
4.3.3 Non-common-case Predictors 22
5 RESULTS 24
5.1 Warp Address Monotonicity 24
5.1.1 Global Memory 24
5.1.2 Texture Memory 27
5.1.3 Local Memory 28
5.1.4 Constant Memory 28
5.1.5 Shared Memory 29
5.2 Warp access-width Characterization 30
5.2.1 Global Memory 30
5.2.2 Texture Memory 32
5.2.3 Local Memory 34
5.2.4 Constant Memory 34

Page

5.2.5 Shared Memory 34
5.3 Predictability based on PC 36
5.3.1 Monotonicity Prediction 36
5.3.2 Bandwidth Prediction 38
6 RELATED WORK 41
7 SUMMARY 43
LIST OF REFERENCES 45

LIST OF FIGURES

Figure Page

2.1 Simplified GPGPU Architecture. Memory spaces are not drawn to scale. 6

2.2 Simplified Streaming Multiprocessor Overview. Units are not drawn to scale. 7

3.1 Coalescer within a Streaming Multiprocessor 10

3.2 Cache Spectrum Design 12

5.1 Monotonicity of Global Memory 25

5.2 Turning Points of Global Memory 26

5.3 Turning Points of bfs, gaussian, and mummergpu in Global Memory . . 26

5.4 Monotonicity of Texture Memory 27

5.5 Turning Points of Texture Memory 28

5.6 Monotonicity of Shared Memory 29

5.7 Turning Points of Shared Memory 30

5.8 Cumulative Distribution of Unique Blocks in Global Memory 32

5.9 Cumulative Distribution of Unique Pages in Global Memory 33

5.10 Cumulative Distribution of Unique Blocks and Pages in Texture Memory 34

5.11 Cumulative Distribution of Unique Addresses in Shared Memory 35

5.12 Monotonicity Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Monotonicity in Global Memory 36

5.13 Monotonicity Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Monotonicity in Texture Memory 37

5.14 Bandwidth Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Blocks 38

5.15 Bandwidth Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Pages 39

5.16 Bandwidth Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Blocks 40

Figure Page

5.17 Bandwidth Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Pages 40

ABSTRACT

Holic, Calvin R. M.S.E.C.E., Purdue University, May 2014. Characterizing the Intra-warp Address Distribution and Bandwidth Demands of GPGPUs. Major Professor: Mithuna S. Thottethodi.

General-purpose Graphics Processing Units (GPGPUs) are an important class of architectures that offer energy-efficient, high performance computation for data-parallel workloads. GPGPUs use single-instruction, multiple-data (SIMD) hardware as the core execution engines with (typically) 32 to 64 lanes of data width. Such SIMD operation is key to achieving high-performance; however, if memory demands of the different lanes in the "warp" cannot be satisfied, overall system performance can suffer.

There are two challenges in handling such heavy demand for memory bandwidth. First, the hardware necessary to coalesce multiple accesses to the same cache block—a key function necessary to reduce the demand for memory bandwidth—can be a source of delay complexity. Ideally, all duplicate accesses must be coalesced into a single access. Memory coalescing hardware, if designed for the worst-case, can result in either high area and delay overheads, or wasted bandwidth. Second, bandwidth demands can vary significantly. Ideally, all memory accesses of a warp must proceed in parallel. Unfortunately, it is prohibitively expensive to design a memory subsystem for the worst-case bandwidth demand where each lane accesses a different cache block.

The goal of this thesis is to characterize the memory-access behavior of GPGPU workloads within warps to inform memory subsystem designs. The goal is *not* to propose and evaluate hardware optimizations based on this characterization. I leave such optimizations for future work with my collaborator Héctor Enrique Rodríguez-Simmonds. Specifically, I characterize two properties which have the potential to lead

to optimizations in the memory subsystem. First, I demonstrate that there is significant access monotonicity at both the cache-block and page levels. This is significant because my collaborator's work reveals that access monotonicity can be leveraged to significantly simplify address coalescing logic. Second, I characterize the memory bandwidth patterns by the number of unique blocks and pages accessed on a per-warp basis. My study motivates a novel horizontal cache organization called a "cache spectrum" (in contrast to traditional, vertical cache hierarchies) to maximize the number of unique accesses that can be served simultaneously. Finally, further optimizations are possible if the warps that access a large number of blocks are predictable. I examine two simple techniques to measure predictability of access patterns for intra-warp bandwidth demands. My (negative) results reveal that more sophisticated predictors may need to be explored.

1. INTRODUCTION

Over the past decade, there has been enormous and growing interest in the use of general purpose graphics processing units (GPGPUs) for high throughput, energy-efficient, data-parallel computing. Several of the TOP500 supercomputing platforms leverage GPUs as compute accelerators, such as the current number two system Titan and the number six system Piz Daint [1].

GPUs are effective at harnessing data parallelism because they are based on single-instruction, multiple-data (SIMD) hardware wherein a single instruction operates on a vector of data. Current GPUs typically have a vector width of 32 (NVIDIA) or 64 (AMD). Another key characteristic of GPUs is the use of the single-instruction, multiple-thread (SIMT) programming model to program the SIMD hardware. Unlike the traditional vector programming model wherein a single thread of control operates on a vector of data, the SIMT model offers the programmer an abstraction of multiple threads that are operating in parallel on scalar data. However, because the underlying hardware is still SIMD, execution of these parallel threads effectively makes lock-step progress. A group of such SIMT threads is referred to as a “warp” (NVIDIA terminology) or a “wavefront” (AMD terminology). The hardware preserves the SIMT illusion when threads diverge (e.g., because of control divergence) by automatically splitting the warp into two and masking off threads appropriately.

While there have been other characterizations of the memory access behavior of GPGPU workloads [2,3], my study is the first to focus on intra-warp access patterns. Specifically, I examine intra-warp accesses because the intra-warp access pattern has a direct impact on two hardware-units—the coalescer and the GPGPU cache—which I describe below.

Consider a warp of N (where $N = 32$ or $N = 64$) threads, each making a memory access. These N parallel accesses have the potential to impose high bandwidth

demands on underlying memory structures like TLBs and caches. It is impractical to build the memory structures for the worst case (i.e. where all N accesses go to unique pages/blocks). As such, practical systems use two techniques to address such warp accesses.

First, GPUs use memory coalescers to detect and coalesce accesses to the same block (or page in the case of TLBs). If in the common case, there is significant overlap with multiple threads accessing the same cache-block, the coalescers ensure that (1) only unique cache block accesses are made, and (2) that the data from the each block/page access is distributed to all threads of the warp that accessed that block/page. Coalescing hardware exists in current GPUs [4], although implementation details are not public to the best of my knowledge.

Second, TLBs and caches may support some fixed number of cache-block accesses in parallel via a combination of banking and multiporting. For example, a cache may allow at most 8 cache blocks to be accessed in parallel. If the warp attempts to access more than 8 unique (because the coalescer eliminates duplicates) cache blocks, the warp effectively incurs memory divergence which prevents the warp from moving forward in lock-step.

Both the coalescer and the GPU cache are important hardware units that affect the overall performance of GPUs. The goal of this thesis is to study the characteristics of intra-warp memory accesses to inform and guide the design of the coalescer and the cache. (The design of the coalescer and cache is not part of this thesis; rather my focus is on the memory access characterization only. My collaborator Héctor Enrique Rodríguez-Simmonds is working on coalescer and cache design that leverages the insights from my study. Our goal is to submit a single paper that describes our findings to an upcoming conference for review and publication.)

My analysis of the Rodinia benchmark suite [2, 3] based on simulation using GPGPU-Sim [5] reveals two key properties of intra-warp memory accesses. First, I observe that addresses accessed by threads in a warp are predominantly monotonic at both the block and page granularities. Further, the address monotonicity property

is common for all classes of memory accesses, such as shared memory, global memory, and texture memory. Monotonicity enables the use of simpler and faster coalescer hardware.

Second, I measure that many important benchmarks have a tapered distribution of number of block accesses per warp wherein more warps access a small number of blocks and fewer warps access a larger number of blocks. If the above distribution had a clear "knee" (i.e. if the number of warps that access more than say k blocks drops precipitously), one could simply design a cache that can supply as many blocks as indicated by the knee-point k. Unfortunately, the measured tapered distribution leaves a significant fat-tail of warps that access a large number of blocks. As a counter-point, if the fat-tail is heavy, one could argue that the worst-case design is the only design that can adequately capture such a distribution. However, that is also not true, as indicated by the tapered profile. The above characterization leads to an interesting design challenge. On one hand, a fixed-bandwidth cache that can support a modest (say 4-to-8) number of parallel accesses can be built, but it results in memory divergence for the fat-tail. On the other hand, the worst-case design to fully support the fat-tail results in expensive (and possibly low-capacity) caches. The tapered profile enables the use of a "cache spectrum" which is composed of multiple caches of varying capacity and bandwidth. To match the tapered distribution, one may employ a larger cache with lower block bandwidth in conjunction with smaller caches with higher bandwidth.

One may think that a simple design where the heavy-head is captured by a cache with limited number of ports is adequate; wider accesses from the tail may be satisfied over multiple cycles. However, even modest tails can lead to severe performance penalties. For example, assume a simple case where 90% of accesses are captured under the heavy head. The remaining 10% are distributed such that 2% require 2 accesses, 2% require 3 accesses, 2% require 4 accesses, an additional 2% require 5 accesses, and the final 2% require 6 accesses. Without bringing in additional warps to

hide this latency[1], the execution time is equal to $(90\times1+2\times(2+3+4+5+6)) = 130\%$ of an ideal case where 100% can be satisfied in a single cycle. Note that this 30% is under conservative conditions where 90% of accesses are under the head and 0% of accesses require 7 or 8 accesses to the cache.

Also, note that the use of a set of caches with increasing capacity (or decreasing bandwidth) already occurs in traditional memory hierarchies. However, I distinguish my cache spectrum from traditional memory hierarchies in that my caches are organized "horizontally" and are equidistant from the processing units whereas in traditional hierarchies, the larger/lower-bandwidth caches are farther from the processor because of their "vertical" organization. Again, I emphasize that the actual cache-spectrum design is part of my collaborator's thesis work beyond the scope of this thesis. My contribution is the workload characterization whose insight leads to the design of the cache spectrum.

My characterization focuses on the Rodinia Benchmark suite [2,3] and uses GPGPU-Sim [5]. Among these benchmarks, we find that several "well-behaved" benchmarks have 100% monotonicity among warp addresses and even the more irregular applications have significant (80%+) monotonicity with the exception of one benchmark. Similarly, when considering the number of accesses per warp, we find that the "well-behaved" subset have modest bandwidth requirements, but the irregular applications demonstrate the tapered profile that can benefit from further optimization.

The remainder of this thesis is organized as follows. Chapter 2 describes relevant background of GPGPUs and gives an overview of GPGPU architecture. Chapter 3 describes the design challenges at hand for modern GPGPU memory subsystems, provides motivation behind my work, and illustrates the value of my contribution. Chapter 4 describes the experimental methodology used for my work, including simulator and benchmark details. Chapter 5 presents my results, Chapter 6 discusses related work, and Chapter 7 summarizes this thesis.

[1]Bringing in additional warps puts pressure on the number of warps needed to hide latency; such upward pressure is also undesirable.

2. BACKGROUND

This chapter outlines the background of Graphics Processing Units (GPUs), their application to general purpose computation, and an overview of important architectural features and terminology. AMD and NVIDIA are the main designers of modern GPUs, and while their architectures are largely the same, I use NVIDIA specifics and terminology for this thesis unless stated otherwise.

2.1 Graphics Processing Unit Background

Graphics Processing Unit (GPU) architectures are designed for highly data parallel workloads, to serve the computational needs of graphics applications. This results is a design that uses a high number of lightweight computational units that operate on different pieces of data. To reduce hardware complexity, one set of decoding and control logic is used for multiple computational units, resulting in a "warp" (NVIDIA) or "wavefront" (AMD) of 32 or 64 data items operated on in lockstep per instruction. This design is known as a single-instruction, multiple-data (SIMD) architecture. Since these architectures are designed for such massively data parallel applications, memory latency can often be hidden by running other warps while memory requests are serviced. This differs from modern CPU design, where latency is more important, and large cache hierarchies are used to hide memory latency. As such, GPUs have much smaller caches in comparison, allowing more chip area to be used for execution units rather than caches.

All of these differences result in an architecture whose overall raw computational power dwarfs that of conventional CPUs. For this reason, there has been a trend toward leveraging GPU hardware for data-parallel, non-graphics, computationally

intensive workloads. In turn, this has resulted in more flexible GPU designs for general purpose use, known as General Purpose Graphics Processing Units (GPGPUs) [6].

2.2 NVIDIA Architecture Background

To support general purpose computation on their devices, NVIDIA created their Compute Unified Device Architecture (CUDA) programming model, which is supported in various programming languages [4]. CUDA provides a programming interface to transfer data between the CPU (host) and GPGPU (device), as well as an interface to run "kernels" on the GPGPU. These kernels support a single-Instruction, multiple-thread (SIMT) programming model, which allows programmers to specify a sequence of instructions to be operated on multiple threads, with the hardware responsible for forming warps (of 32 threads) to execute together. These warps then execute on what is called a "Streaming Multiprocessor" (SM), which is a collection of the previously mentioned lightweight computational units as well as memory and special function units [4]. Each SM has an L1 cache, and all SMs share an L2 cache. Figures 2.1 and 2.2 illustrate a simplified view of this design.

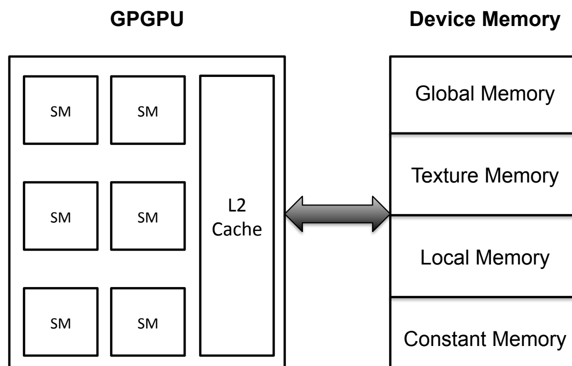


Fig. 2.1. Simplified GPGPU Architecture. Memory spaces are not drawn to scale.

NVIDIA's CUDA programming model specifies five different memory spaces, each of which have unique characteristics such as size and scope, outlined below. Device

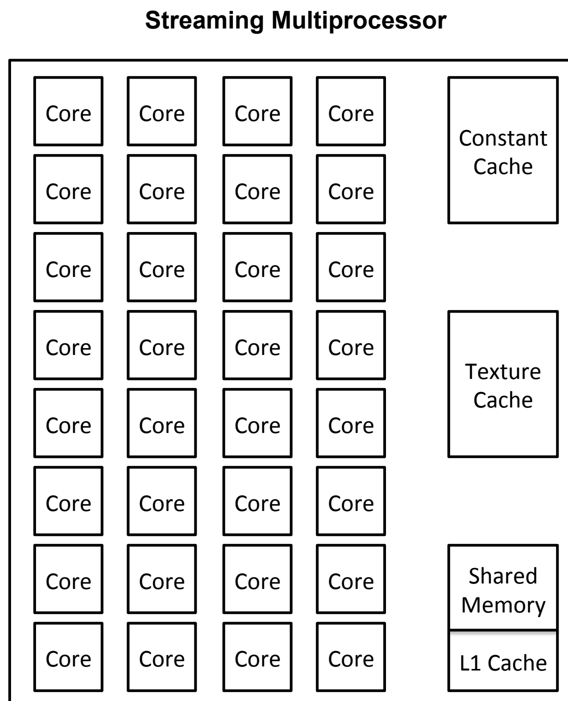


Fig. 2.2. Simplified Streaming Multiprocessor Overview. Units are not drawn to scale.

memory refers to the main, off-chip memory of the GPGPU. For the purpose of this background, an NVIDIA Compute Capability of 2.0 or higher is assumed [4].

2.2.1 Global Memory

Global memory is most similar to a traditional CPU view of memory. It is located in device memory, and is likewise the largest and slowest of the memory spaces. Global memory is read/write, cached, and can be accessed by all threads on the GPGPU. Global memory is cached in each SM's L1 cache and the shared L2 cache.

2.2.2 Texture Memory

Texture memory is similar to global memory in that it resides in device memory, is likewise slow, and is cached. Also like global memory, it is accessible by all threads

on the GPGPU. It differs, however, in that it is read-only, and can only be accessed by using device function calls [4]. Texture memory also has its own separate texture cache. While texture memory is used more in true graphics workloads, it is seen in some general purpose applications as well.

2.2.3 Local Memory

Local memory also resides in device memory, so it experiences the same high latency that Global and Texture memory experience. It has read/write access and is private to each individual thread. Local memory, like global memory, is cached in each SM's L1 cache as well as the shared L2 cache.

2.2.4 Constant Memory

Constant memory is located in device memory, is read-only, and is accessible to all threads. However, it has a very high speed, small cache that allows for cache hits to be accessed as quickly as registers. As its name implies, constant memory is intended for use with constants that don't change in value throughout the execution of the entire kernel.

2.2.5 Shared Memory

Shared memory is unique in that it is the only memory space that is located on-chip, and thus acts as a software-managed cache. Likewise, it is rather small and fast. Each Streaming Multiprocessor has its own shared memory structure, and its scope is limited to threads in the same thread block. In fact, shared memory and the L1 cache share the same hardware structure, and the programmer can configure how the structure is divided into shared memory and L1 cache space [4]. Shared memory is an address-level structure (opposed to block) and is banked [4]. It has read/write access.

3. MEMORY ACCESS CHARACTERIZATION

3.1 Coalescer Design: The Challenge

Consider what needs to be done to coalesce memory accesses across a warp. As input, we have a collection of memory addresses (typically 32 or 64). With these addresses, the coalescing hardware must find all of the unique addresses (or blocks or pages, depending on the memory space's granularity), and supply these addresses to be fetched from memory. As these memory spaces are cached, coalescing must occur between the functional units and the cache (or TLB), as shown in Figure 3.1.

Since this coalescing hardware lies in the critical path before the cache, its design has considerable latency constraints. These latency constraints are the main motivation behind my work—finding regular memory access patterns can allow this design to be simplified.

In general, coalescing requires every address to be compared to every other address to ensure that all unique addresses/blocks/pages are found and that no duplicates are missed. This all-to-all comparison is extremely expensive with respect to latency and/or area, and does not scale well—the number of necessary comparsions is $\mathcal{O}(n^2)$.

One may think that coalescing latency is unimportant because GPUs are throughput engines; one could always tolerate (i.e. hide) the latency of coalescing logic by executing other warps in the meantime. However, such latency-oblivious approaches increase the number of warps needed. Modern GPUs have a limited number of warps, and they are needed for tolerating other latencies in the system. Unnecessarily increasing the required number of warps puts pressure on these precious resources.

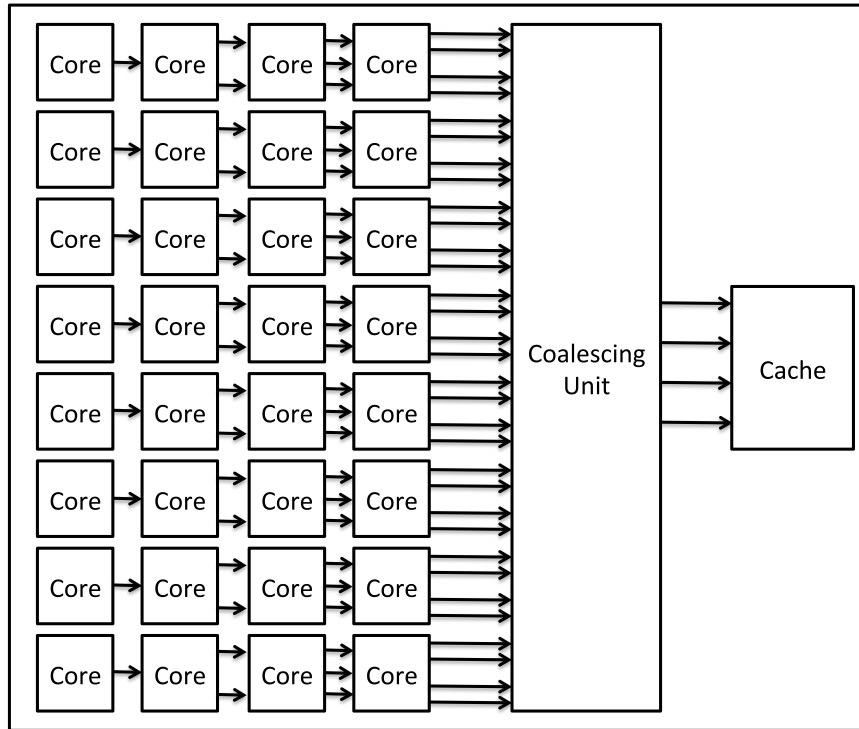


Fig. 3.1. Coalescer within a Streaming Multiprocessor

3.2 Exploring Monotonicity

In search of an access pattern to allow for simplified coalescing hardware, I consider how this coalescing hardware could be simplified. To do so, we must reduce the number of comparisons made to coalesce accesses. If addresses are sorted in increasing or decreasing order, then simply comparing each address to its left and right neighbor will allow for the detection of unique addresses. Comparisons to any other addresses are unnecessary, since this sorted ordering is transitive.

Sorting in hardware can be as expensive as a naive coalescing hardware implementation. However, if address distributions are monotonically increasing or decreasing across a warp by way of their access pattern, then these addresses don't *need* to be sorted to take advantage of this simpler neighbor-to-neighbor comparison design. As such, I investigate the monotonicity rates of warp memory accesses. Note that this

"simpler" design is with respect to the common-case latency; full coalescing hardware may still be necessary to account for the non-common-case.

3.3 Bandwidth Characterization

The cache accesses of a given warp may have bandwidth demands that fall in a wide range. On one extreme, it is possible that all threads in the warp access data in a single cache block (or page). If such behavior were the common-case behavior, the coalescer would require a single memory access. As such, a single ported cache (or TLB) would be adequate. On the other extreme, it is also possible that each thread in the warp accesses data in a unique cache block (or page). If such behavior were the common-case behavior, the only design would be an impractical high-bandwidth memory that can serve 32 independent accesses in parallel. (In general, the worst case design would need true multiporting; banking would not be adequate because one can always construct adversarial access patterns where all addresses are conflicted on a single bank.)

The true distribution of intra-warp bandwidth demands lies somewhere between the above two extreme possibilities. The second prong of my thesis studies this distribution.

For an efficient hardware design, we would like to see a sharp "knee" in the bandwidth distribution, such that a large percentage of overall accesses is captured by a very small increase in the number of parallel accesses. In this case, the hardware can be designed with that width of parallel accesses, and will be used efficiently. Ideally, this knee would occur at a relatively small number of parallel accesses. In contrast, if there is a large percentage of accesses that require a full 32 parallel accesses, then the worst-case design will be necessary. However, neither of these access patterns fully represent current workloads.

The true distribution, we will see later in Chapter 5, is a mixture of well-behaved knees with low bandwidth requirements, knees with higher bandwidth requirements,

and tapered distributions. Likewise, a more novel cache design is required to efficiently capture the bandwidth demands of these access patterns.

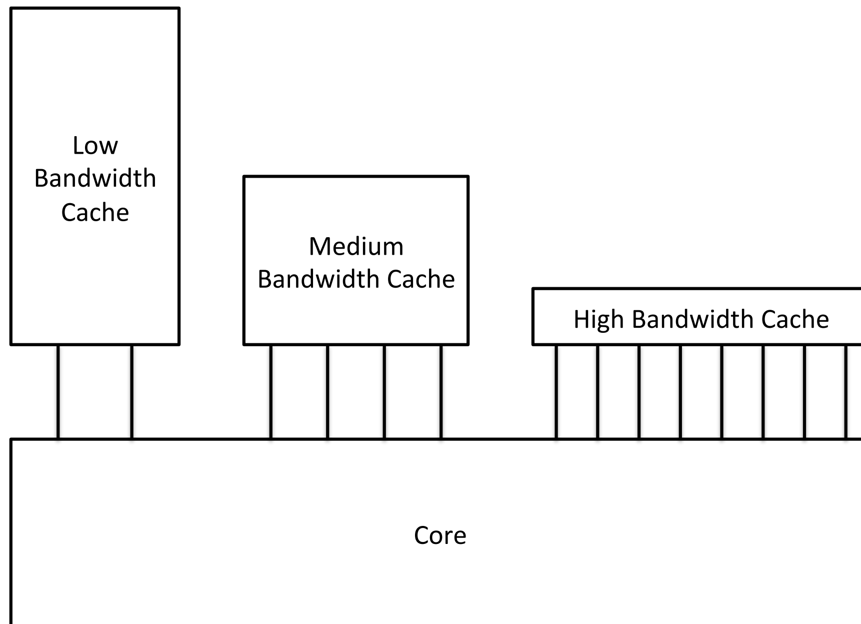


Fig. 3.2. Cache Spectrum Design

To address these bandwidth requirements, I propose a horizontally oriented spectrum of caches, with varying sizes and bandwidths, as shown in Figure 3.2. On one end of the cache spectrum will be a large, low-bandwidth cache that will serve the majority of warp memory accesses. On the other end of the spectrum lies a very small, but heavily multiported cache to serve the small percentage of warps who have large bandwidth demands. I only provide an overview of the type of cache design inspired by my characterization; the details will be in my collaborator's work.

4. METHODOLOGY

In order to characterize the address distribution and bandwidth patterns of GPGPU applications, I use a GPGPU simulator to collect detailed memory access patterns. This simulator, as well as the benchmarks I use, are outlined below. The simulator and benchmarks were run on a cluster on Intel Xeon E5310 servers running Red Hat Enterprise Linux Workstation release version 6.5. I used CUDA implementations of benchmarks compiled with NVIDIA's CUDA Toolkit (version 4.0.17).

4.1 Simulator

For my simulator, I use GPGPU-Sim developed at the University of British Columbia [5]. GPGPU-Sim simulates NVIDIA's parallel thread execution (PTX) virtual instruction set, and supports both CUDA and OpenCL code and libraries. The simulator was made specifically to study non-graphics, general purpose applications, and is thus an appropriate fit for my studies.

I use version 3.2.2 of GPGPU-Sim, with the included configuration file to simulate an NVIDIA Tesla C2050. I selected this configuration as it was one of the configurations with the highest Compute Capability (2.0) that also most closely matched actual NVIDIA hardware I use for functional verification of my GPGPU-Sim build. The Tesla C2050 implements NVIDIA's Fermi architecture [7].

All changes to the simulator were made only for the purposes of gathering data and statistics pertaining to the memory access patterns of the applications run on the simulator. No functional changes were made to the simulator.

4.2 Benchmarks and Data Sets

For my workload analysis, I selected the Rodinia benchmark suite developed at the University of Virginia [2,3]. Designed for heterogeneous computing, Rodinia contains benchmarks that focus on high levels of parallelism, written in CUDA (as well as OpenMP and OpenCL), to help architects study designs of accelerators such as GPUs. Rodinia's benchmarks span across a variety of domains (such as medical imaging, data mining, linear algebra, and others) and a variety of applications (structured grid, unstructured grid, graph traversal, dense linear algebra, and dynamic programming). I use version 2.4 of the Rodinia suite, which includes 19 benchmarks.

For all of my Rodinia benchmarks, I start with the default run settings provided in version 2.4, and grow and shrink their input to fit reasonable run time constraints. This allows me to increase the number of memory accesses for many benchmarks, and for others allows me to gather data for a complete simulation. In practice, this typically equates to a half of a day to one week in overall simulation run time. For those that use included datasets, I use only those included sets, selecting the most appropriately sized one. When possible, I try to match the input configurations used in [2,3].

Of the 19 included benchmarks, I run and collect data for all but one—CFD Solver. This is due to the fact that while running the smallest included dataset, the benchmark would run for an extended period of time (over 9 days) and then crash. Specific details pertaining the remaining 18 benchmarks can be found below.

4.2.1 Back Propagation (backprop)

Back Propagation is an unstructured grid, pattern recognition application that employs a machine-learning algorithm to train weights of nodes of a neural network for facial recognition [2]. After starting with the default number of input nodes as 64K, I increased this to 1M nodes to increase run time and the number of memory accesses. This decision was guided by similar sized data provided for Breadth-First

Search. Note that this is an even larger input size than used for real hardware in [2,3], so I consider it sufficiently large. This input size runs for about 5 hours on my system, and provides over an order of magnitude more memory accesses than the original 64K input.

4.2.2 Breadth-First Search (bfs)

Breadth-First Search is an implementation of the common graph algorithm that traverses all connected components of a graph. Very large graphs (millions of vertices) are common in real-world engineering and scientific domains [2]. Rodinia includes 3 graphs for use with bfs, along with an input generator. After preliminary tests using the default 64K-sized graph, I upgraded to the largest provided graph, 1M. This is the same input size as used on real hardware in [2, 3]. Like with backprop, this provides a large increase in the number of overall memory accesses (just over one order of magnitude). Total run time on my system was about 3 and a half hours.

4.2.3 B+ Tree (b+tree)

B+ Tree is another graph traversal application for searching B+ trees It uses a simpler B+Tree implementation rather than a more sophisticated one as used in industry to focus on theoretical performance [8]. B+ Tree was added to the Rodinia suite after the main characterization papers were written. As Rodinia only provides one dataset for use with b+tree, I use that for my data collection. As such, this was one of the quickest running benchmarks I use, taking about an hour to complete.

4.2.4 Gaussian Elimination (gaussian)

Gaussian Elimination is an implementation of the common algorithm to solve a system of linear equations. This implementation synchronizes between iterations, with each row by row calculation being done in parallel within each iteration [9]. This

is another benchmark added to the suite after the Rodinia papers. The default 4x4 matrix included with Rodinia finishes almost instantaneously, and as such generates a very small number of memory accesses. Likewise, I upgraded to the largest matrix provided, a 1024x1024 matrix. On my system, this completes in about one week.

4.2.5 Heart Wall Tracking (heartwall)

Heart Wall is a structured grid, medical imaging application that tracks the walls of mouse heart over a sequence of ultrasound images [3]. Rodinia came with only one set of images (in AVI format), so I use that video file. The default run configuration has the number of frames parameter set to 5 (the number of frames used in [3] was not specified). Under these default settings, the benchmark takes about 2 days to complete on my system.

4.2.6 HotSpot (hotspot)

HotSpot is a structured grid, physics simulation application. It is used to estimate the temperature of a processor given simulated power measurements along with the processor's architectural floor plan [2]. HotSpot is another benchmark whose default settings finish nearly instantaneously. I changed from the default 64x64 temperature and power matrices to match the configuration used in [2]. This uses a 512x512 matrix ([2, 3] used a 500x500 matrix) and runs for 360 iterations as specified in [2].

4.2.7 K-means (kmeans)

K-means is a dense linear algebra, data mining application. It "identifies related points by associating each data point with its nearest cluster, computing new cluster centroids, and iterating until convergence" [2]. The Rodinia characterization papers use a 819,200 point dataset in [2] and a 204,800 point dataset in [3]. Since the default run configuration is in the middle of those two figures at 494,020 points and

takes about 4 and a half days to complete, I use that default configuration for my simulations.

4.2.8 LavaMD (lavaMD)

LavaMD is an n-body, molecular dynamics simulation that "calculates particle potential and relocation due to mutual forces between particles within a large 3D space" [9]. LavaMD was added to Rodinia after the characterization papers, and doesn't use a dataset. Instead, it takes in an argument defining the number of boxes in one dimension to use in the simulation. Rodinia's default run uses 10 boxes in each dimension, and takes about 4 days to run. As such, I use this default for all of my simulation runs.

4.2.9 Leukocyte (leukocyte)

Leukocyte is a structured grid, medical imaging application. It detects white blood cells known as leukocytes in a video microscopy of blood vessels [2]. Leukocyte is another example of a benchmark that, like its medical imaging counterpart heartwall, includes only one set of data (a video to analyze), along with a number of frames to process as an argument. It is worth noting that the default Rodinia 2.4 suite includes run configuration files in both the top-level leukocyte folder and in a nested CUDA directory, and they have different default values for the number of frames. I use 10 frames, the default in the second-level run file. While [2] uses 25 frames, my 10-frame configuration takes over 5 days to run, and generates a large number of memory accesses.

4.2.10 LU Decomposition (lud)

This benchmark is another dense linear algebra algorithm used to solve a set of linear equations. In contrast to gaussian elimination, this decomposes a matrix into

upper and lower triangular matrices, and has many interdependencies between rows and columns, resulting in data sharing between threads [3]. In [3], a 256x256 matrix was used. I found this to compute very quickly, and changed to the largest provided 2048x2048 matrix. This increased the number of memory accesses by a factor of over 400, while completing in about a day and a half.

4.2.11 MUMmerGPU (mummergpu)

MUMmerGPU is a graph traversal, bioinformatics application [3]. It is a DNA sequence alignment program adapted from MUMmer for use on GPUs [10]. This benchmark did not run with the stock build included with Rodinia 2.4. When running on the simulator, the benchmark would exit prematurely due to a failed CUDA call to cuMemGetInfo(). Since MUMmerGPU also has a CPU emulation mode, I replaced the CUDA call with the emulation mode approach of hardcoding the total and free memory available on the device. In [3], an input size of 50,000 25-character queries was used. This query dataset, along with the reference set, are included in Rodinia 2.4. I use this configuration for my simulations which takes about 2 hours to run to completion.

4.2.12 Myocyte (myocyte)

Myocyte is a structured grid, biological simulation application. It simulates the behavior of heart muscle cells known as cardiac myocytes. This can be useful in identifying the development of heart failure [11]. Myocyte has two different parallelization modes:

- Parallelize within each simulation instance
- Parallelize across instances of simulation

Rodinia's default settings use only one simulation instance, and likewise parallelizes within instances. The default configuration simulates 100 milliseconds. As Myocyte

was added after the Rodinia papers, I use these settings for my simulations. This configuration takes about 7 hours to complete on my system.

4.2.13 k-Nearest Neighbors (nn)

Nearest Neighbors is a dense linear algebra, data mining application. Given a set of records with latitude and longitude coordinates, along with a target latitude and longitude, it returns the k-nearest neighbors to the target coordinates [9].

Like with MUMmerGPU, this benchmark did not run without modification. It failed on a CUDA cudaMemGetInfo() call like MUMmerGPU, so I used the same hardcoded memory information approach described for MUMmerGPU.

Rodinia comes with a very small set of data for this benchmark, along with an input generator to create larger datasets. Of the data sizes included in their generator script, I use the largest option—32M records. Generating a set of records larger than this would result in memory constraint issues. This dataset takes about 2 and a half hours to run on my system.

4.2.14 Needleman-Wunsch (nw)

Needleman-Wunsch is another DNA sequencing algorithm. It is a dynamic programming, bioinformatics application [2]. In [2, 3], a 2048x2048 set of data points is used as input. This is the default in Rodinia, which I opted to use as well. While I did try growing the input size, memory constraint issues quickly arose, so I run my simulations with the 2048x2048 input size.

4.2.15 Particle Filter (particlefilter)

Particle Filter is a structured grid, medical imaging application. It is a statistical estimator that uses a Bayesian framework [12]. The default run configuration for this benchmark uses a 128x128 resolution input with 10 frames and 1,000 particles.

Using [12] to guide my input size, I increased my number of particles to 10,000. I tried growing to 100,000 particles, the maximum size used in [12], but this exceeded reasonable run time constraints. Since I am using a simulator rather than real hardware, and am focused on memory access patterns rather than application speedup, I find this to be an acceptable input size.

4.2.16 PathFinder (pathfinder)

PathFinder is a dynamic programming, grid traversal application that finds the lowest weighted path from the bottom to top row of a 2D grid, always moving straight or diagonally forward [9]. Rodinia's default width for PathFinder's input is 100K. While PathFinder was added to the Rodinia suite after the characterization papers, another study that analyzed Rodinia's OpenMP performance on real hardware ran Rodinia on an input size up to a width of 400K, so I adopt this data size for my simulations as well [13]. Using this configuration, my simulations complete in about 3 and half hours.

4.2.17 Speckle Reducing Anisotropic Diffusion (SRAD)(srad)

SRAD is an unstructured grid, image processing application. It is a diffusion algorithm commonly used in ultrasonic and radar imaging to reduce speckles in images via the use of partial differential equations [2]. Rodinia includes two versions of the SRAD kernel. Version 1 uses an actual input image and is more computationally intensive than version 2. Version 2 forgoes using an actual input image and instead randomizes the input, and also makes more use of the GPU's shared memory. As I am interested in memory access patterns, I use the second version of the kernel for shared memory patterns.

In [2], an input size of 2048x2048 is used, and ran for 100 iterations. Due to run time constraints, I use this same 2048x2048 sized input, but for only 10 iterations. This takes about one and a half days to run on my system.

4.2.18 Streamcluster (streamcluster)

Streamcluster is a dense linear algebra, data mining application that is a part of the PARSEC benchmark suite [2]. As said in [14], "for a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center." In [2, 3], an input size of 64K points and 256 dimensions is used. Unfortunately, this input is simply too large for us to run on my system with GPGPU-Sim—it ran for about a month before crashing. As such, I turn to PARSEC's data sizes. I use the simmedium input size of 8,192 input points and 64 dimensions [14]. Using this, my simulations take about 17 hours to complete.

4.3 Analysis

All vector loads and stores for each of the memory spaces outlined in Chapter 2 were captured for monotonicity and uniqueness at both block and page granularities (for caches and TLBs respectively) for my analysis. One- and two-bit predictors were then used with the intent to predict the non-common-case for monotonicity and uniqueness. Further details can be found below.

4.3.1 Monotonicity

To measure the monotonicity of memory accesses, I compare each memory address (at block and page granularities) in a warp in increasing thread ID order. Threads that are inactive (due to thread divergence, for example) are ignored and skipped over (i.e. if threads 4 and 6 are active but thread 5 is inactive, thread 4's address will be compared to thread 6's address). If, after these comparisons it is found that the sequence of active threads' memory addresses are monotonically increasing or monotonically decreasing, I count that warp's memory access to be monotonic.

To gain insight into warp memory accesses that were not monotonic, I also measure the number of "turning points" that occur at each thread ID. A turning point occurs

when there is a break in monotonicity, and the sequence of memory addresses reverses direction. This was motivated by the fact that if these turning points occur at regular locations (i.e. if monotonicity was consistently broken only once, halfway through the warp), the access could be treated as multiple sets of monotonic accesses.

4.3.2 Bandwidth Demand

For my bandwidth access-width characterization, I collect the number of unique addresses, unique blocks, and unique pages that are accessed within each warp. For this, I use 128B blocks, as specified in [4]. Information on page sizes and TLBs for GPUs is sparse, so I use 4K pages in the interest of heterogeneous memory spaces, as others have done [15].

4.3.3 Non-common-case Predictors

The common case behaviors found in my results are that memory accesses are highly monotonic, and memory bandwidth demands are usually low. Likewise, I investigate the predictability of the non-common-case properties—non-monotonic accesses and high bandwidth demands. The reason for predicting non-monotonicity is to know when to use simplified coalescing hardware that accounts for monotonicity rather than worst-case coalescing hardware. The motivation behind predicting bandwidth demands is to allow the appropriate cache in a cache spectrum to be selected to retrieve data from.

For my per-warp monotonicity and bandwidth predictor analysis, I used infinite tables effectively, with no aliasing. Because existing benchmarks are small, I do not believe that my conclusions will change significantly because of finite tables. Furthermore, since my results are already negative, any performance penalties associated with finite-sized tables are irrelevant. I experiment with simple 1-bit and 2-bit predictors to demonstrate the value of prediction, each of which initially predicts the common case. For bandwidth demands, I used thresholds of 2 and 4 blocks and 1 and

2 pages. For example, the 4-block predictor would predict if the warp would have more than 4 or fewer blocks, or more than 4 blocks.

5. RESULTS

The key results of this thesis are summarized below.

- I show that address monotonicity property largely holds. This observation motivates hardware techniques for simpler address coalescing in the common case.
- I show that common case access width is low but the average does not reveal the full story. There exists a significant fraction especially among irregular benchmarks whose access width is high. This observation motivates the use of a bandwidth spectrum.
- I show that neither the monotonicity property nor the access width are easily predictable based on the program counter. One-bit and two-bit per-PC predictors are no better than a static predictor.

5.1 Warp Address Monotonicity

In my simulations, all 18 benchmarks accessed global memory, 10 accessed shared memory, 3 accessed texture memory, 3 accessed constant memory, and 1 accessed local memory.

5.1.1 Global Memory

Global memory was accessed by all of the 18 benchmarks I ran. Their monotonicity rate at block and page granularities is shown in Figure 5.1. As you can see, the benchmarks' global memory accesses were largely monotonic. Of the 18 benchmarks, 11 were 100% monotonic at both the block and page levels. Of the remaining 7, 4 of them (heartwall, leukocyte, particlefilter, and streamcluster) had greater than 98%

monotonicity at the block and page levels. This leaves only 3 benchmarks that do not exhibit extremely high rates of monotonicity: bfs, gaussian, and mummergpu. Of these, bfs and mummergpu, both of which are graph traversal applications, still have a reasonably high rate of monotonicity at about 85%. Gaussian elimination is the only real outlier with a monotonicity rate of about 25%.

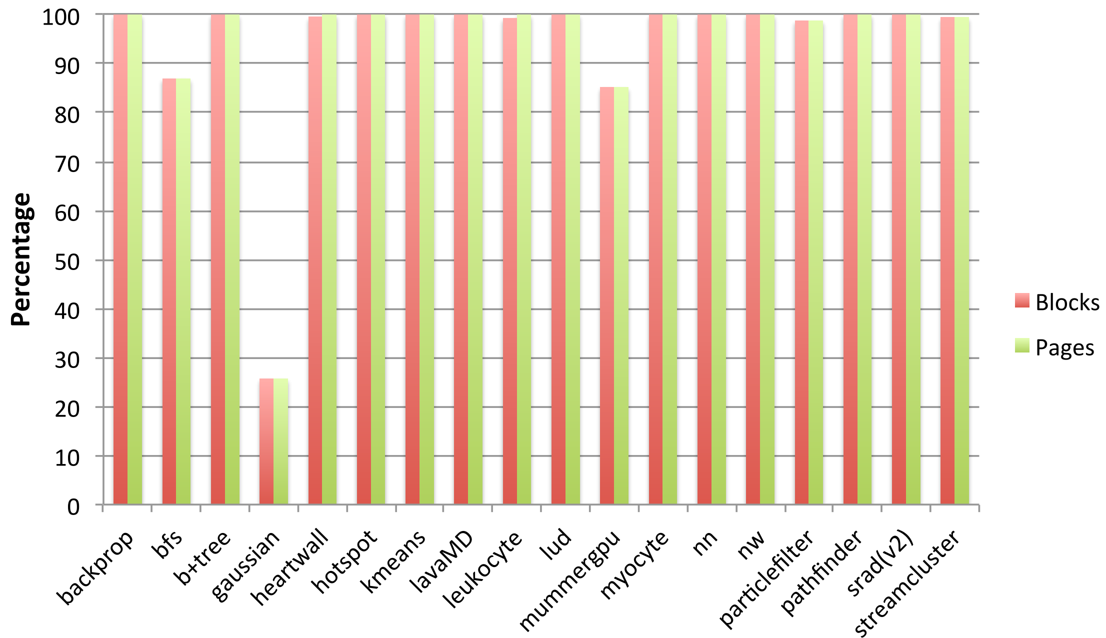
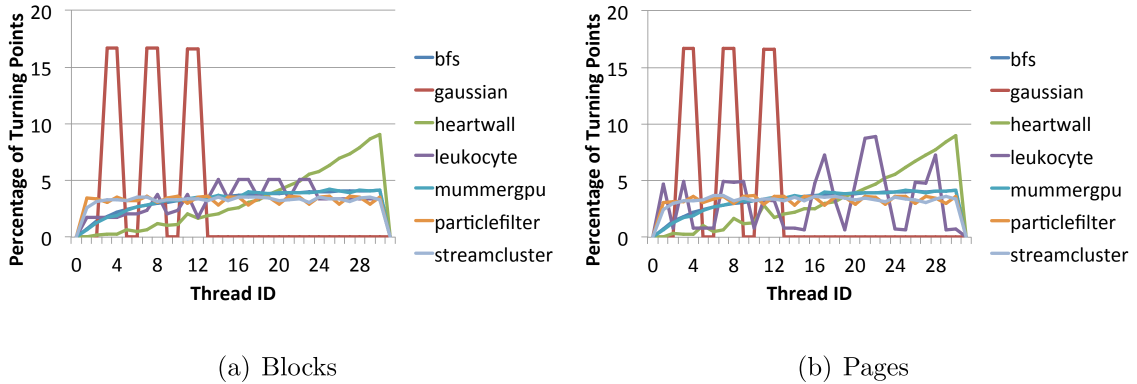


Fig. 5.1. Monotonicity of Global Memory

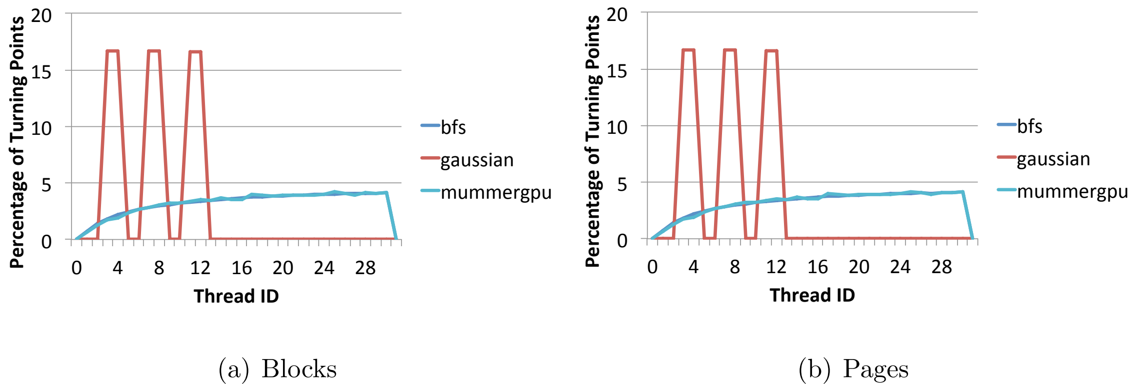
For further insight, I turn to my turning point data. Figure 5.2 shows the turning points for all non-monotonic benchmarks, and Figure 5.3 shows the turning points only for the benchmarks with lower monotonicity rates. Looking at Figure 5.2, the turning points seem to be fairly evenly distributed on the whole, with a slightly increasing trend with increasing thread ID. Figure 5.3 shows that bfs and mummergpu both follow this trend, with nearly identical behavior, at both block and page granularities. Gaussian, however, has a very unique trend. Threads 3, 4, 7, 8, 11, and 12 have all of the turning points distributed nearly evenly for both blocks and pages (a closer look reveals the real percentages to be 16.69% for threads 3 and 4, 16.67%

for threads 7 and 8, and 16.64% for threads 11 and 12). Looking at the underlying data, the number of turning points that occur at each of these threads matches up nearly identically to the number of non-monotonic accesses, implying that virtually all non-monotonic warps have turning points at all 6 of those threads. This implies that gaussian has an access stride of 4 monotonic addresses per block (and page).



(a) Blocks (b) Pages

Fig. 5.2. Turning Points of Global Memory



(a) Blocks (b) Pages

Fig. 5.3. Turning Points of bfs, gaussian, and mummergpu in Global Memory

5.1.2 Texture Memory

Texture memory was accessed by only three of the benchmarks ran: kmeans, leukocyte, and mummergpu. Figure 5.4 shows the monotonicity results for blocks and pages. As you can see, kmeans and leukocyte have 100% monotonicity for both blocks and pages, while mummergpu has very little monotonicity—just under 14% for blocks and pages. One possible cause for this difference is the fact that mummergpu uses 2-dimensional textures while kmeans and leukocyte both use 1-dimensional textures. As such, it may be difficult to create simple coalescing logic that handles 2D texture accesses well.

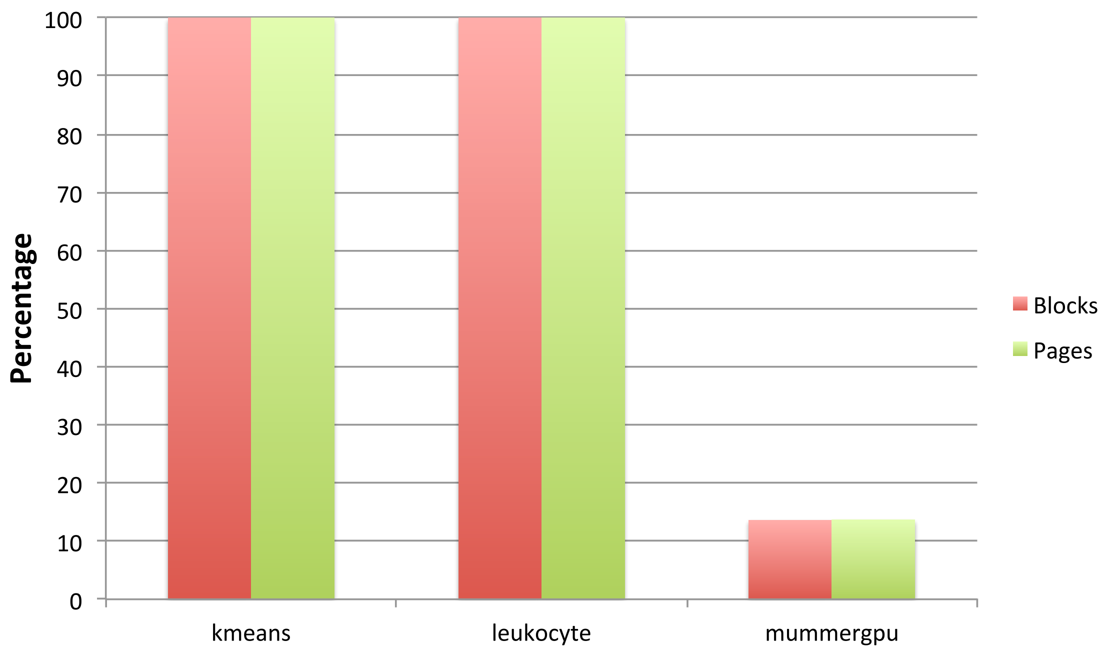
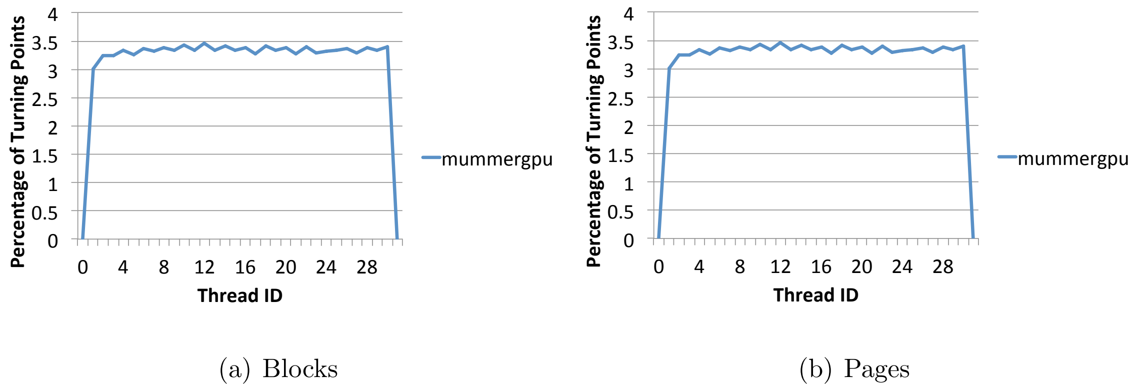


Fig. 5.4. Monotonicity of Texture Memory

Given this low monotonicity rate for mummergpu, I examine my turning point data for more information. Figure 5.5 shows the percentage of turning points that occur at each thread ID for mummergpu's texture accesses. As you can see, the turning points are very evenly distributed throughout the warp. Unfortunately, this

result gives us no clear insight into which, if any, parts of a warp access is monotonic with any regularity.



(a) Blocks

(b) Pages

Fig. 5.5. Turning Points of Texture Memory

5.1.3 Local Memory

Local memory was used only by mummergpu. For this benchmark, accesses were 100% monotonic at both the block and page granularities. While the sample size is small, it's worth noting that every local memory instruction accessed only one unique block per warp, so coalescing logic may be able to be greatly simplified.

5.1.4 Constant Memory

Constant memory was also only used by three benchmarks: heartwall, kmeans, and leukocyte. For all of these benchmarks, accesses were 100% monotonic, at both the block and page granularities. In fact, every single access from all three benchmarks accessed only one address per warp. As such, there is no need for coalescing logic.

5.1.5 Shared Memory

Shared memory was used by 10 of the 18 benchmarks. As mentioned in Chapter 2, shared memory is an address-level memory space. As such, I look at the monotonicity of addresses rather than blocks and pages. The monotonicity rates of those benchmarks that used shared memory can be seen in Figure 5.6. Of the 10 benchmarks shown, 6 of them (backprop, lavaMD, nw, particlefilter, pathfinder, and srad) are 100% monotonic, 3 of them (heartwall, hotspot, and leukocyte) are highly monotonic (97% or higher), and only 1 exhibits poor monotonicity: lud.

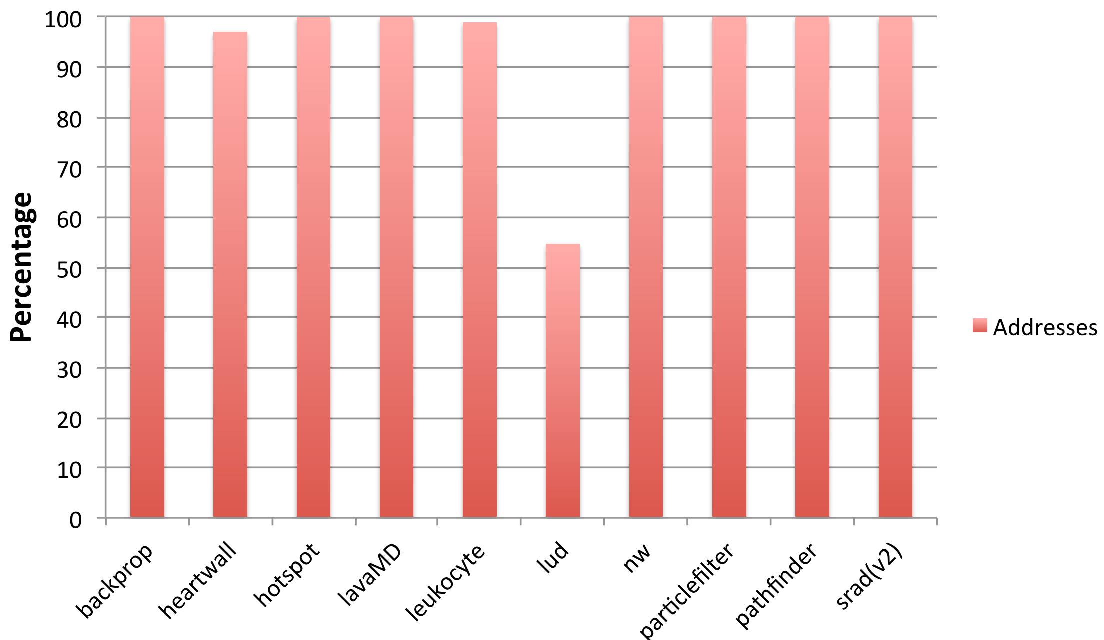


Fig. 5.6. Monotonicity of Shared Memory

Looking at the turning point data for addresses in shared memory, there is a clear pair of turning points for lud at the halfway point of the warp. A look at the underlying data shows that these turning points, at threads 15 and 16, occur for *every* warp that is not monotonic. This implies that while nearly half of the warps are not monotonic, all of the ones that aren't are broken into two sets of monotonic access

patterns, split in the exact middle of the warp. While lud is an outlier, this access pattern can easily be accounted for in coalescing logic.

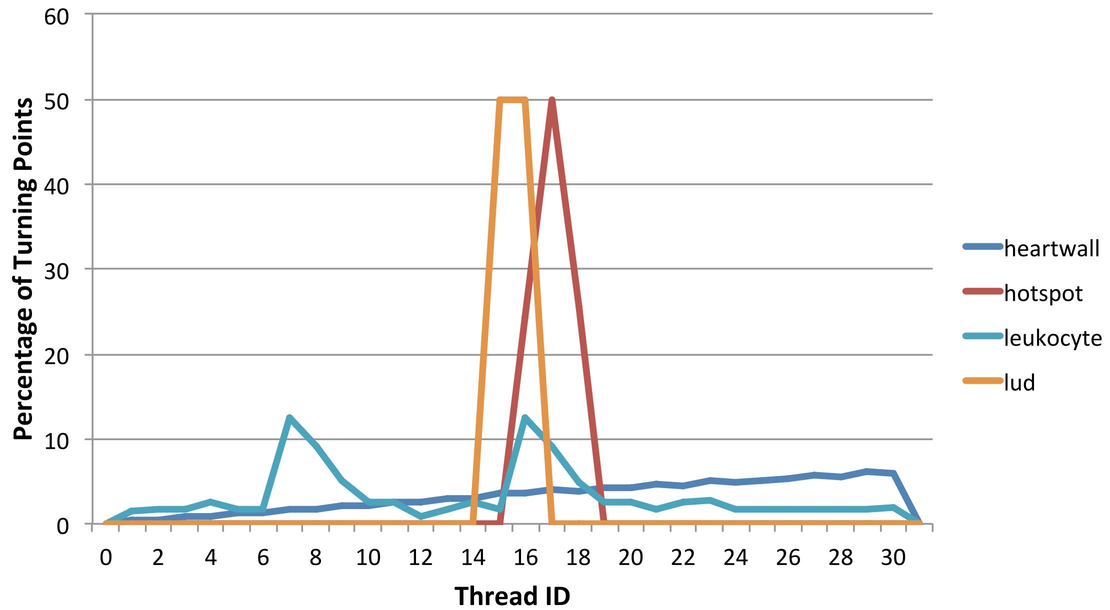


Fig. 5.7. Turning Points of Shared Memory

5.2 Warp access-width Characterization

For my bandwidth demand characterization, I measure the number of unique addresses/blocks/pages on a per-warp basis, and show the cumulative distribution of accesses.

5.2.1 Global Memory

Figure 5.8 shows the cumulative distribution of global memory block accesses, and Figure 5.9 shows the cumulative distribution of memory page accesses.

Figure 5.8(a) shows the benchmarks whose bandwidth demands are very low—all 7 of these benchmarks have a maximum of 2 unique blocks accessed per warp

throughout the entire execution of the application. Their percentage of 1-block accesses varies, as seen in the graph.

Benchmarks shown in Figure 5.8(b) also have low memory bandwidth demands—they all have over 90% of accesses requiring only 2 blocks. The benchmarks here do have a slight tail, but they all reach 100% of accesses at 16 unique blocks.

Figure 5.8(c) shows benchmarks that still have moderately low bandwidth demands, with over 95% of accesses requiring at most 4 unique blocks. However, these benchmarks have more variation and longer tails than the previous two graphs. Particlefilter, for instance, requires all 32 blocks to be unique for a small fraction of its accesses.

The least well-behaved benchmarks can be seen in Figure 5.8(d). Here, we see a few different trends. MUMmerGPU and bfs each have a tapered profile, with no distinct knees. LavaMD exhibits a large knee, but at a moderate bandwidth requirement of 8 unique blocks. Lastly, kmeans has a unique and challenging trend. While 2 unique blocks covers the majority of memory accesses, there is also a significant percentage (about 30%) of accesses that require a full bandwidth of 32 unique blocks.

Turning attention to page accesses in Figure 5.9, we observe trends largely similar to those found in block accesses.

Figure 5.9(a) shows 6 benchmarks whose unique page demands are very low. Note that scale on this figure—at least 97% of all of the benchmarks require only 1 unique page. Furthermore, 100% of their accesses require a maximum of 2 unique pages.

In Figure 5.9(b), we observe a similar trend as the previous graph. While 100% of accesses still require at most 2 unique pages, the difference lies in the percentage of accesses that require only 1 unique page. Here, this percentage ranges from under 3% to about 84%.

In contrast, benchmarks in Figure 5.9(c) have a large percentage of accesses that require only 1 unique page, but they have longer tails before reaching 100% of accesses. In particular, nw has a small knee from 97% to 100% at 16 unique pages.

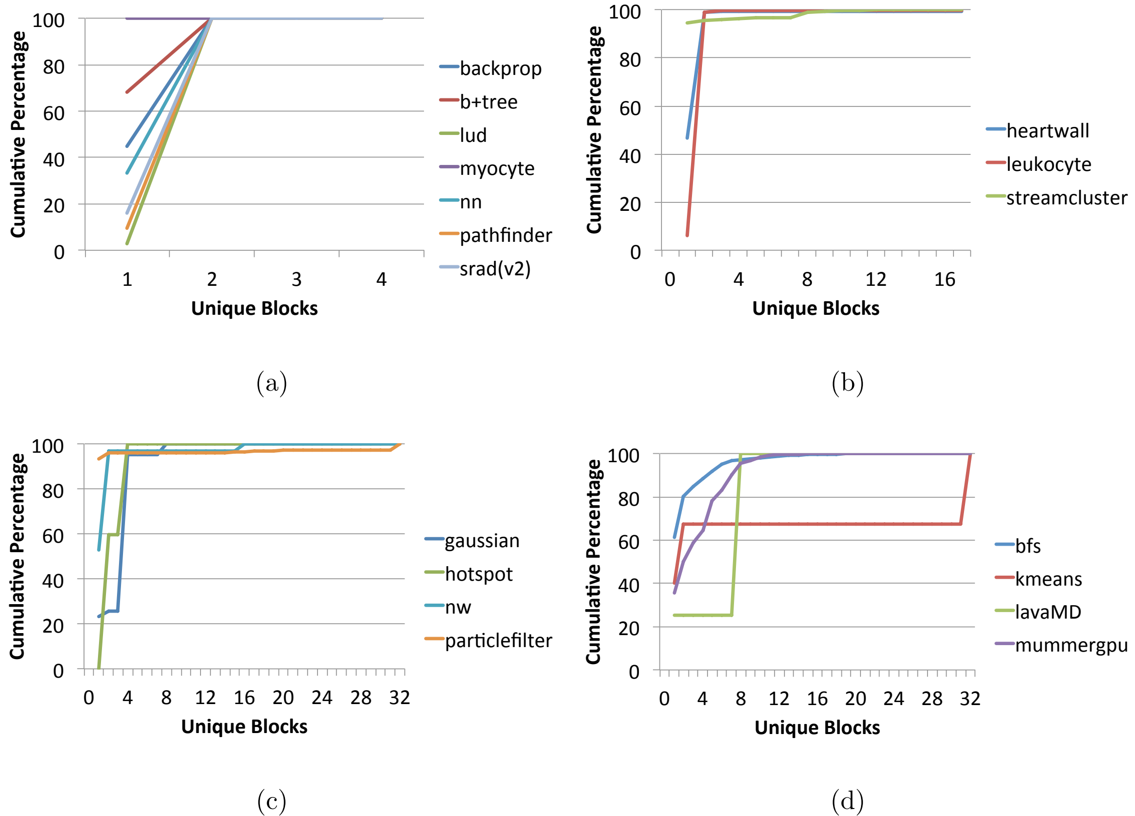


Fig. 5.8. Cumulative Distribution of Unique Blocks in Global Memory

Figure 5.9(d) shows the benchmarks with more unique memory demands. As with blocks, mummergpu and bfs illustrate a gradual tapered distribution, with no distinct knee. By comparison, gaussian does have a very significant knee, with nearly 75% of accesses requiring 4 unique pages.

5.2.2 Texture Memory

The cumulative access distribution (of both blocks and pages) of the three benchmarks that utilized texture memory can be seen in Figure 5.10. While not immediately visible, kmeans and leukocyte overlap nearly identically in their trend, and both have 100% of their accesses covered by a bandwidth of 2 unique blocks and 2 unique pages. Kmeans and leukocyte have a small percentage of accesses that require only

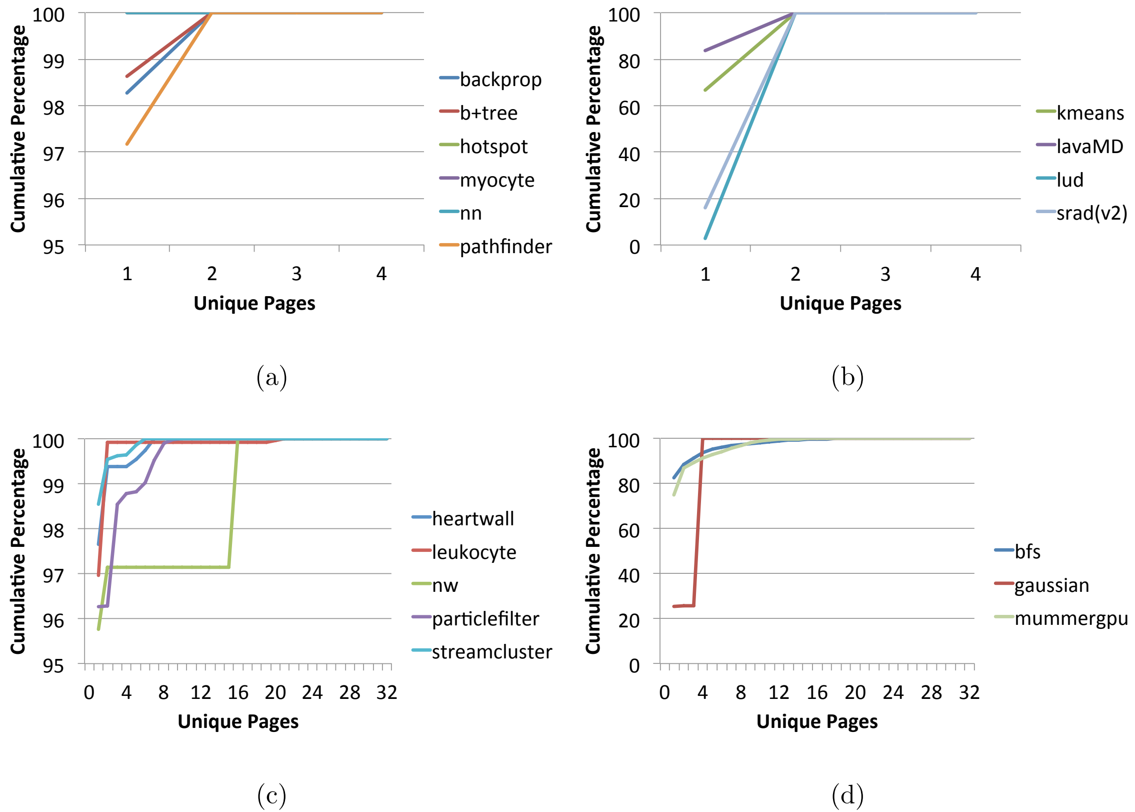


Fig. 5.9. Cumulative Distribution of Unique Pages in Global Memory

one block (about 14%), but a large percentage of accesses are covered by one unique page (about 97% of accesses).

While kmeans and leukocyte are well-behaved, mummergpu has a very unique access pattern that is difficult to account for. While its cumulative distribution is largely tapered, there is also a sharp knee at the end of the distribution, requiring 32 unique blocks about 40% of the time and 32 unique pages about 35% of the time. As such, this benchmark may be very challenging to account for in a bandwidth spectrum design.

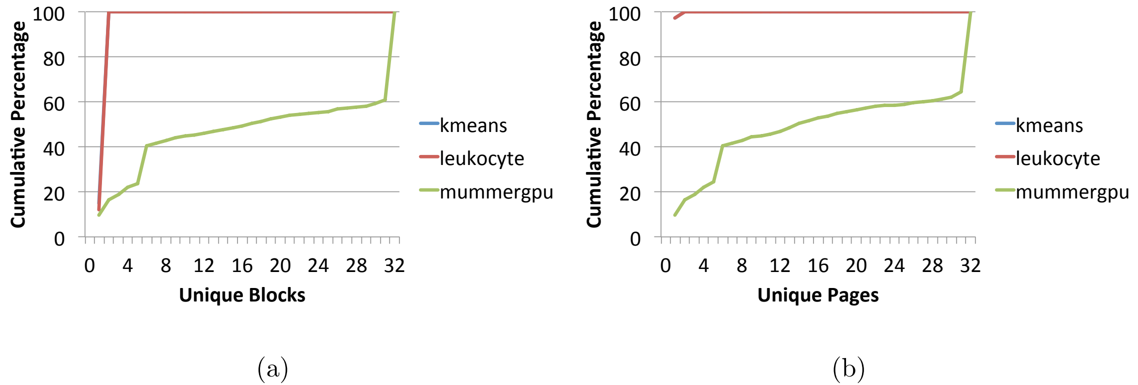


Fig. 5.10. Cumulative Distribution of Unique Blocks and Pages in Texture Memory

5.2.3 Local Memory

For local memory, mummergpu is the only benchmark that has any memory accesses. For every one of these accesses, only one unique address is ever accessed. As such, a bandwidth spectrum would not add any value at this level.

5.2.4 Constant Memory

Constant memory was used by three of the benchmarks ran. Across all of these benchmarks, every warp accessed only one single address, just like the results for mummergpu in local memory. Thus, a cache spectrum in constant memory is unnecessary.

5.2.5 Shared Memory

Since shared memory is an address-level structure, we look specifically at the cumulative distribution of addresses to characterize bandwidth demands. Figure 5.11 shows this distribution for each of the 10 benchmarks that utilize shared memory.

Figure 5.11(a) shows three benchmarks whose distribution includes two or more distinct knees. Backprop and lud each have knees at 2, 16, and 32. For both of them,

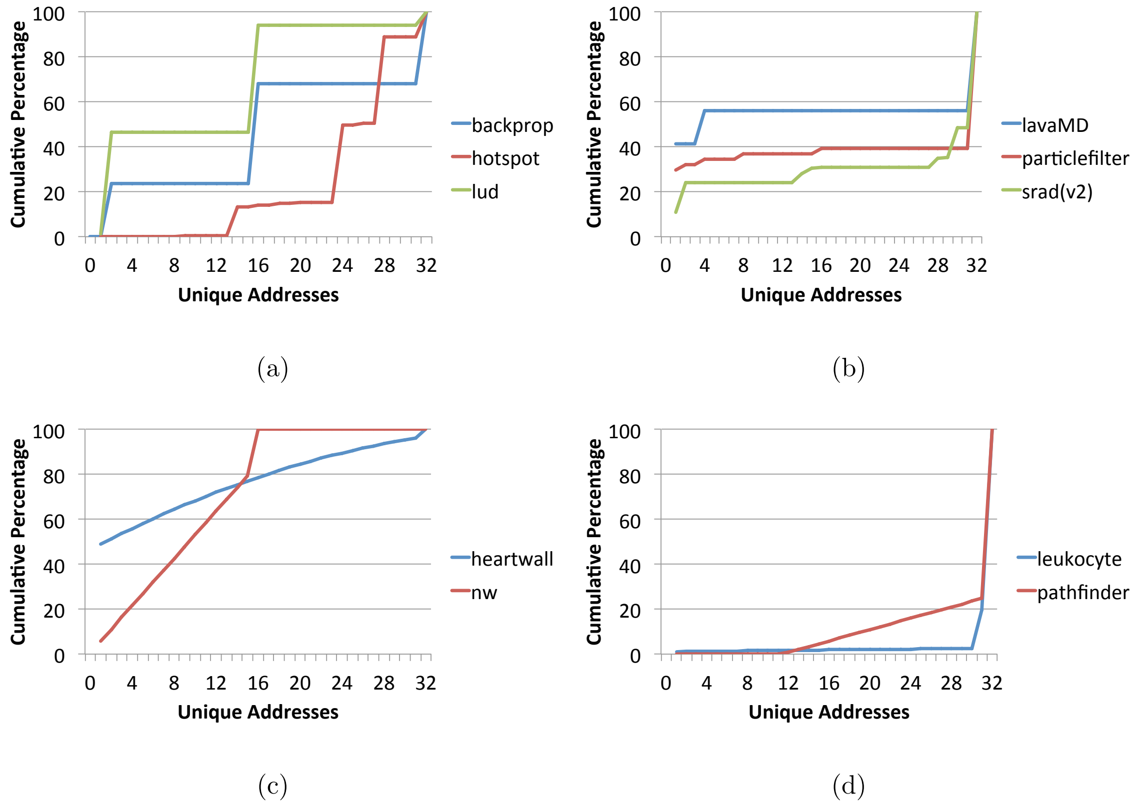


Fig. 5.11. Cumulative Distribution of Unique Addresses in Shared Memory

the jump to 16 unique addresses is largest. While also exhibiting distinct knees, hotspot's behavior is different in both number and magnitude of these knees, as well as where they occur. For hotspot, large jumps occur at 14, 24, 28, and 32 unique addresses.

Benchmarks in Figure 5.11(b) exhibit a slightly different trend than those of the previous graph. Here, the three benchmarks shown have a largely "flat" trend, with large jumps coming only early on at low bandwidth demand, and at the end of the distribution at 32 unique addresses.

Figure 5.11(c) shows two benchmarks whose cumulative distribution have a gradual trend. For heartwall, the access distribution starts at about 50% and slowly grows to 100% at 32 unique accesses. NW also starts with a steeper gradual trend from

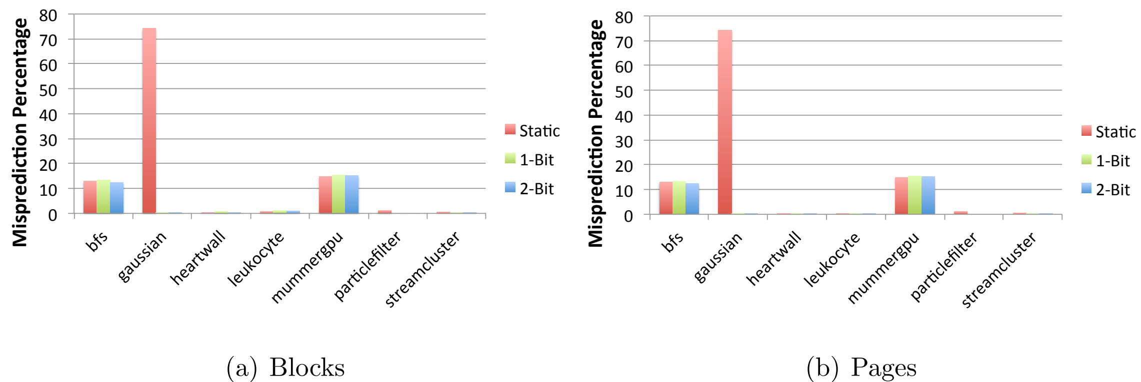
5% at 1 unique address to 80% at 15 unique addresses, at which point it has a knee, hitting 100% at 16 unique addresses.

Last, we have two benchmarks who exhibit trends that cannot be coalesced. Figure 5.11(d) shows these trends. Leukocyte maintains a very low percentage of accesses that have low memory bandwidth requirements (under 3% up to 30 unique addresses), and thus a very high percentage of accesses requiring high bandwidth. Pathfinder performs similarly, with the exception that it has a very gradual increase from 12 to 31 unique threads, before requiring full 32-address bandwidth for 75% of accesses.

5.3 Predictability based on PC

5.3.1 Monotonicity Prediction

As mentioned in Chapter 4, I implemented 1- and 2-bit predictors for monotonicity. To compare, I also calculate the misprediction rate of a static predictor. Since local and constant memory accesses were 100% monotonic, predictors do not have a use. I did not collect predictor data at the address-level, so shared memory is also absent from this prediction analysis.

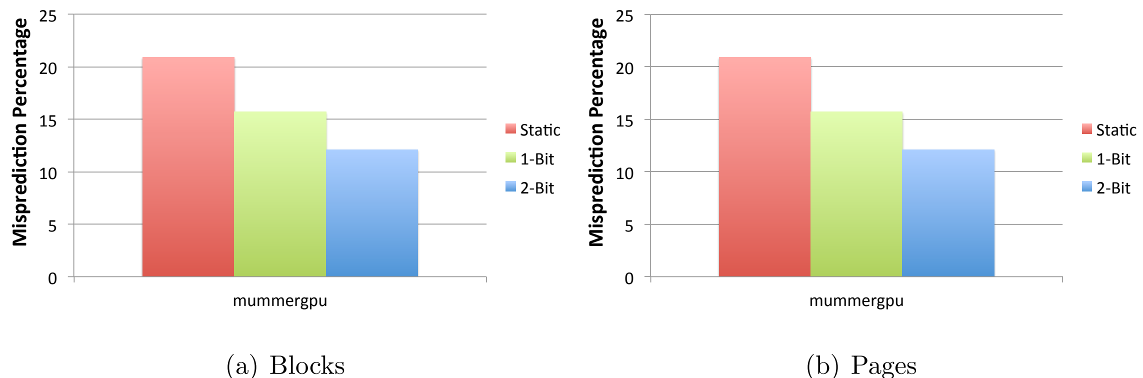


(a) Blocks

(b) Pages

Fig. 5.12. Monotonicity Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Monotonicity in Global Memory

Figures 5.12 shows the misprediction rates of monotonicity at block and page granularities for global memory. I include only those benchmarks that were not 100% monotonic in global memory. Recall that of the 18 benchmarks that were run, bfs, gaussian, and mummergpu were the 3 that were not highly monotonic. As such, I focus on those 3 here, as they are the benchmarks that would benefit most from prediction. As you can see in both figures, a simple static predictor is on par with both 1- and 2-bit predictors for both bfs and mummer. This implies that simple prediction on a per-PC basis is ineffective. The other benchmarks that were highly (but not fully) monotonic had similar results with respect to their predictors. Gaussian elimination, however, does not follow this pattern. With the worst monotonic rate of all benchmarks, gaussian has the most to gain. Here, the 1- and 2- bit predictors provide over a 99% accuracy rate for monotonic prediction. Note that the block and page results are very similar.



(a) Blocks

(b) Pages

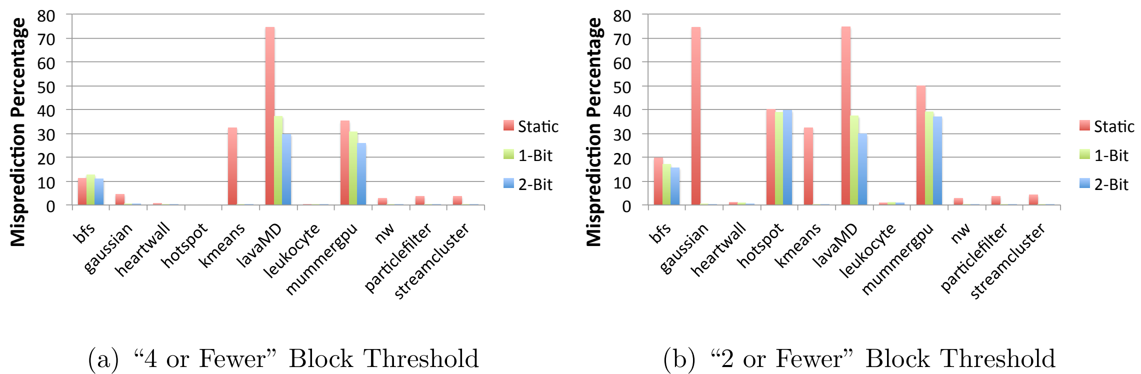
Fig. 5.13. Monotonicity Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Monotonicity in Texture Memory

MUMmerGPU is the only benchmark that uses texture memory and doesn't have 100% monotonic accesses. The results for block and page monotonicity prediction can be found in Figure 5.13(a) and in Figure 5.13(b), respectively. Here, the 1- and 2-bit predictors do each decrease misprediction rates; however, the misprediction rates are still reasonably high.

5.3.2 Bandwidth Prediction

To measure bandwidth predictability, I use 1- and 2-bit predictors at various thresholds for memory access bandwidth. As mentioned in Chapter 4, I use thresholds of 2 and 4 blocks and 1 and 2 pages. As with monotonicity, I compare these values to the misprediction rate of a static predictor at each threshold. Since local and constant memory accesses only 1 unique address per warp in all of my benchmark runs, predictors are unnecessary. Again, since I did not implement address-level prediction, shared memory is excluded from this analysis.

Figure 5.14 shows the misprediction rates of these three predictors at the "4 or fewer" and "2 or fewer" block thresholds.



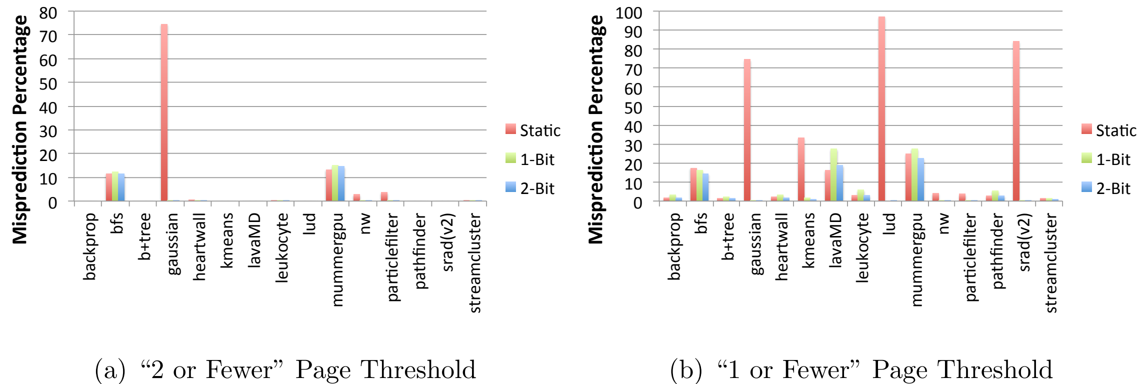
(a) "4 or Fewer" Block Threshold

(b) "2 or Fewer" Block Threshold

Fig. 5.14. Bandwidth Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Blocks

Recall from section 5.2.1 that bfs, kmeans, lavaMD, and mummergpu exhibit unique bandwidth distributions at the block-level. The graph traversal benchmarks bfs and mummergpu prove to be difficult to design for, as their predictability is poor. However, kmeans is highly predictable for its higher bandwidth accesses, which would aid in the use of a potential bandwidth spectrum cache design. Furthermore, lavaMD, while still exhibiting a large misprediction rate, does benefit greatly with 1- or 2-bit predictors over static predictors. At the 2-block level, gaussian also benefits greatly from a 1- or 2-bit predictor, while hotspot exhibits low predictability; however,

hotspot has a large knee in its access pattern and is fully captured at the 4-block level. All other benchmarks whose bandwidth distribution was already well behaved experience similar or better predictability using 1- or 2-bit predictors in comparison to static predictors.



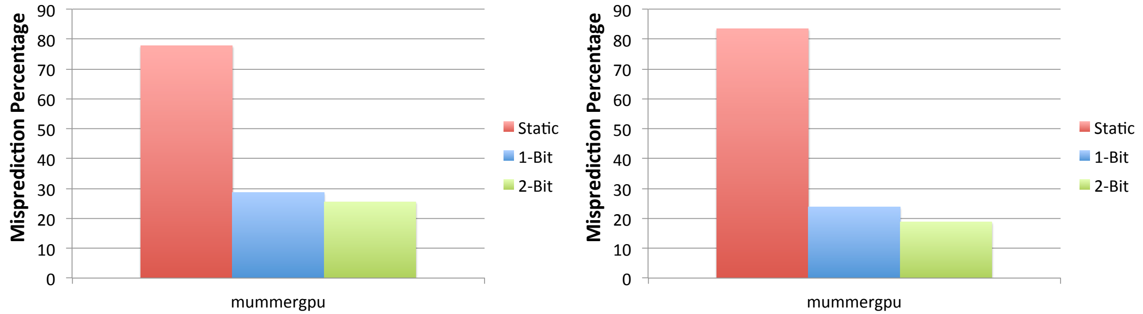
(a) “2 or Fewer” Page Threshold (b) “1 or Fewer” Page Threshold

Fig. 5.15. Bandwidth Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Pages

Figure 5.15 shows the misprediction rates at the “2 or fewer” and “1 or fewer” page thresholds. Recall that bfs, gaussian, and mummergpu were the benchmarks to exhibit unique access distributions at the page-level. Here, we see bfs and mummergpu continue to show poor predictability. However, gaussian is highly predictable for its higher demand accesses. As with blocks, lavaMD’s predictability at the page-level is also poor. The remaining benchmarks exhibit high predictability by way of their already well-behaved access patterns that have low bandwidth demands. Note that in Figure 5.15(a) many benchmarks have a 0% misprediction rate since they never access more than 2 pages.

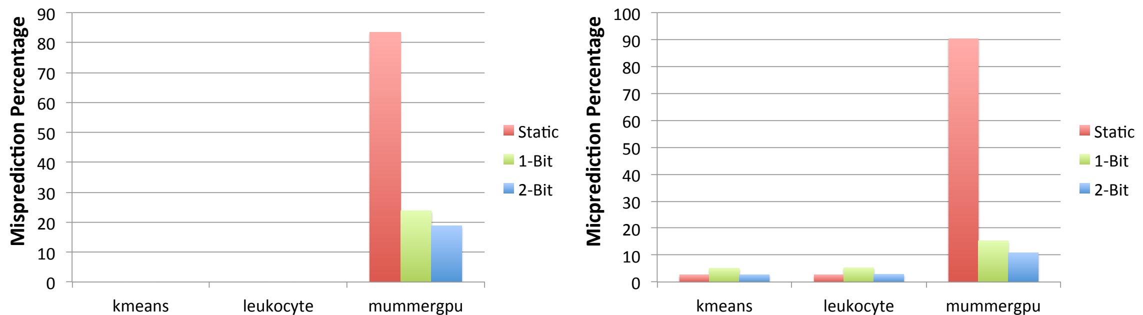
Turning to texture memory, Figure 5.16 shows the misprediction rate at the block level for mummergpu. Since kmeans and leukocyte never access more than 2 unique blocks, their predictors are always 100% accurate at these thresholds, so they are excluded from the figure. Here, we can see that while the 1- and 2-bit predictors are big improvements over static predictors, mummergpu continues to exhibit poor pre-

dictability and difficult characteristics. Figure 5.17 shows the page-level misprediction rates. Again, mummergpu's predictability is less than desired. Kmeans and leukocyte both exhibit high predictability at the 1-page level, and are 100% predictable at the 2-page level, since they never access more than 2 pages.



(a) "4 or Fewer" Block Threshold (b) "2 or Fewer" Block Threshold

Fig. 5.16. Bandwidth Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Blocks



(a) "2 or Fewer" Page Threshold (b) "1 or Fewer" Page Threshold

Fig. 5.17. Bandwidth Misprediction Rates of Static, 1-bit, and 2-bit Predictors for Pages

6. RELATED WORK

Che et.al. develop and characterize the Rodinia benchmark suite in [2]. Their characterization of these benchmarks focuses on other aspects of performance such as speedup achieved on a GPGPU over a multicore CPU, power dissipation, and the distribution of run time over various system aspects (such as CPU-GPU communication, CPU execution, GPU execution, etc.). They further characterize the suite in [3] with performance metrics such as IPC and thread divergence. With respect to the memory system, they break down distribution of accesses across memory spaces (as I did as well) and characterize speedup with respect to increased memory bandwidth. This characterization simply varies the number of memory channels simulated, and does not consider the hardware implications and does not directly analyze the bandwidth demand of the applications.

Bakhoda et.al. introduce GPGPU-Sim and characterize various workloads on their simulator in [5]. Like [3], they present the distribution of memory accesses (with respect to the different memory spaces) for their benchmarks, as well as warp occupancy. They also investigate potential speedup from coalescing accesses; however, they do not characterize the access patterns that benefit from this coalescing.

Jang et.al. look into exploiting access patterns to improve memory performance in [16]. They present a mathematical model that captures memory access patterns, and use it to develop optimizations at the software-level to take advantage of memory coalescing.

Pichai et.al. study the effects of various TLB designs and parameters in [15], also use Rodinia and GPGPU-Sim. Given the lack of public knowledge of TLBs in GPGPUs, this work contributes valuable insight for GPGPU TLB design. Their results center around different designs, however, and do not directly look at the

underlying access patterns to inform particular designs with respect to coalescing or bandwidth demands.

Power et.al. also study TLBs in GPGPUs in [17]. For their study, they use Rodinia and gem5-gpu, a simulator that integrates the CPU simulator gem5 with the GPGPU simulator GPGPU-Sim. As such, their work focuses on this heterogeneous system rather than only the GPGPU, and the effects of TLBs and Page Walk Caches on performance, area, and energy.

7. SUMMARY

General-purpose, Graphic Processing Units (GPGPUs) represent an important class of computing platforms. For data-parallel workloads, their large amount of raw computational power allows for energy-efficient computation. As such, these workloads have been widely studied.

While the computational throughput offered by GPGPUs is significant, their memory system poses key challenges. Consider a single "warp" or a SIMD unit of work. Given a vector memory access, modern GPGPUs must (a) contend with limited bandwidth for the common case and (b) ensure this limited bandwidth is fully and efficiently used. To do so, it is important to understand the demands placed on GPGPU hardware.

As such, two aspects of GPGPU design—the cache system and the memory coalescer—require insight into intra-warp memory access patterns. While various aspects of GPGPU memory subsystems have been investigated, they do not examine and present the intra-warp access patterns needed to inform coalescer and cache design.

The key contributions of my thesis are these characterizations of GPGPU intra-warp access patterns. With respect to memory coalescing, I investigate the monotonicity of access patterns across threads in a warp, and illustrate that the majority of Rodinia benchmarks exhibit very high (at least 98%) rates of monotonicity. This finding motivates a coalescing hardware design that requires only neighbor-to-neighbor comparisons for unique access detection, opposed to a naive design requiring all-to-all comparisons. With respect to the memory bandwidth demands of each warp, I measured the number of unique accesses per warp, and illustrate that these demands vary significantly by benchmark, and that their access patterns motivate the use of a cache spectrum.

The insight of these access patterns leads to simpler and/or faster hardware designs, which my collaborator is further investigating.

LIST OF REFERENCES

LIST OF REFERENCES

[1] TOP500, "Chinas tianhe-2 supercomputer maintains top spot on 42nd top500 list," Press Release, November 2013, http://s.top500.org/static/lists/2013/11/PressRelease201311.pdf.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[3] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec 2010, pp. 1–11.

[4] *CUDA C Programming Guide*, NVIDIA, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, April 2009, pp. 163–174.

[6] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x

[7] *Tesla C2050 / C2070*, NVIDIA, http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf.

[8] J. Fix, A. Wilkes, and K. Skadron, "Accelerating braided b+ tree searches on a gpu with cuda," *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC)*, 2011.

[9] *Rodinia:Accelerating Compute-Intensive Applications with Accelerators*, Rodinia, https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page.

[10] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 1, p. 474, 2007. [Online]. Available: http://www.biomedcentral.com/1471-2105/8/474

[11] L. G. Szafaryn, K. Skadron, and J. J. Saucerman, "Experiences accelerating matlab systems biology applications," in *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits*, 2009, pp. 1–4.

[12] M. A. Goodrum, M. J. Trotter, A. Aksel, S. T. Acton, and K. Skadron, "Parallelization of particle filter algorithms," in *Proceedings of the 2010 International Conference on Computer Architecture*, ser. ISCA'10. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 139–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24322-6_12

[13] J. Shen and A. L. Varbanescu, "A detailed performance analysis of the openmp rodinia benchmark," Delft University of Technology, PDS Technical Report PDS-2011-011, 2011. [Online]. Available: http://www.pds.ewi.tudelft.nl/fileadmin/pds/reports/2011/PDS-2011-011.pdf

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[15] 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 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942

[16] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 105–118, Jan 2011.

[17] J. Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *Proceedings of the 20th IEEE International Symposium On High Performance Computer Architecture*. HPCA, 2014.