Dynamic Warp Formation: Exploiting Thread Scheduling for Efficient MIMD Control Flow on SIMD Graphics Hardware

by

Wilson Wai Lun Fung

B.A.Sc., The University of British Columbia, 2006

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

Master of Applied Science

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

The University Of British Columbia

(Vancouver)

September, 2008

Abstract

Recent advances in graphics processing units (GPUs) have resulted in massively parallel hardware that is easily programmable and widely available in commodity desktop computer systems. GPUs typically use *single-instruction, multiple-data* (SIMD) pipelines to achieve high performance with minimal overhead for control hardware. Scalar threads running the same computing kernel are grouped together into SIMD batches, sometimes referred to as warps. While SIMD is ideally suited for simple programs, recent GPUs include control flow instructions in the GPU instruction set architecture and programs using these instructions may experience reduced performance due to the way branch execution is supported by hardware. One solution is to add a stack to allow different SIMD processing elements to execute distinct program paths after a branch instruction. The occurrence of diverging branch outcomes for different processing elements significantly degrades performance using this approach. In this thesis, we propose dynamic warp formation and scheduling, a mechanism for more efficient SIMD branch execution on GPUs. It dynamically regroups threads into new warps on the fly following the occurrence of diverging branch outcomes. We show that a realistic hardware implementation of this mechanism improves performance by an average of 47% for an estimated area increase of 8%.

Table of Contents

Abstract ii

Table of Contents iii

List of Tables vi

List of Figures vii

Acknowledgements ix

1 Introduction 1

1.1 Motivation 2

1.2 Contributions 5

1.3 Organization 6

2 Background 7

2.1 Fundamental Concepts 7

2.1.1 Thread-Level Parallelism 7

2.1.2 Data-Level Parallelism 8

2.1.3 Single-Instruction, Multiple-Data (SIMD) 8

2.1.4 Multiple-Instruction, Multiple Data 9

2.1.5 Fine-Grained Multithreading 10

2.2 Compute Model 12

2.3 SIMD GPU Microarchitecture 13

2.4 Latency Hiding 14

2.5 SIMD Execution of Scalar Threads 16

2.6 Summary . . . 16

3 SIMD Control Flow Support . . . 17

3.1 SIMD Serialization . . . 18

3.2 SIMD Reconvergence . . . 18

3.3 Reconvergence Point Limit Study . . . 22

3.4 Summary . . . 23

4 Dynamic Warp Formation and Scheduling . . . 24

4.1 Register File Access . . . 25

4.2 Hardware Implementation . . . 27

4.2.1 Warp Pool . . . 29

4.3 Scheduling Policies . . . 30

4.4 A Majority Scheduling Policy Implementation . . . 31

4.4.1 Warp Insertion . . . 33

4.4.2 Warp Issue . . . 33

4.4.3 Complexity . . . 34

4.5 Summary . . . 34

5 Methodology . . . 35

5.1 Software Design of *GPGPU-Sim*—A Cycle Accurate GPGPU Simulator . . . 35

5.1.1 Shader Core . . . 36

5.1.2 Interconnection Network . . . 42

5.1.3 DRAM Access Model . . . 46

5.1.4 Interfacing with sim-outorder . . . 51

5.2 Baseline Configuration . . . 54

5.3 Benchmarks . . . 54

6 Experimental Results . . . 56

6.1 Effects of Scheduling Policies . . . 57

6.2 Detail Analysis of Majority Scheduling Policy Performance . . . 59

6.3 Effect of Limited Resources in Max-Heap . . . 62

6.4 Effect of Lane Aware Scheduling . . . 63
6.5 Effect of Cache Bank Conflict . . . 64
6.6 Sensitivity to SIMD Warp Size . . . 65
6.7 Warp Pool Occupancy and Max Heap Size . . . 67

7 Area Estimation . . . 68

8 Related Work . . . 72
8.1 SIMD Control Flow Handling . . . 72
8.1.1 Guarded Instruction/Predication . . . 72
8.1.2 Control Flow Reconvergence Mechanisms . . . 73
8.1.3 Conditional Streams . . . 73
8.2 Dynamic Grouping SIMD Mechanisms . . . 74
8.2.1 Dynamic Regrouping of SPMD Threads for SMT Processors . . . 74
8.2.2 Liquid SIMD . . . 75
8.3 Eliminating the Existence of Branch Divergence . . . 75
8.3.1 Complex SIMD Branch Instruction . . . 76
8.3.2 Vector-Thread Architecture . . . 76

9 Conclusions and Future Work . . . 77
9.1 Summary and Conclusions . . . 77
9.2 Contributions . . . 78
9.3 Future Work . . . 79
9.3.1 Better Warp Scheduling Policies . . . 79
9.3.2 Area Efficient Implementation of the Warp Pool . . . 79
9.3.3 Bank Conflict Elimination . . . 80
9.3.4 Improvements to GPGPU-Sim . . . 80

References . . . 82

List of Tables

3.1 Operational rules of the stack for reconvergence mechanism. 19

5.1 Relationships between constrain counters and DRAM commands. 50
5.2 Hardware Configuration 55
5.3 Benchmark Description 55

6.1 Memory bandwidth utilization. 58
6.2 Cache miss rates (pending hits classified as a miss). 58
6.3 Cache miss rates without pending hits. 58
6.4 Maximum warp pool occupancy and max-heap size 67

7.1 Area estimation for dynamic warp formation and scheduling. 69
7.2 CACTI parameters for estimating structure sizes in Table 7.1 70

List of Figures

1.1 Floating-Point Operations per Second for the CPU and GPU 3

1.2 Performance loss due to branching when executing scalar SPMD threads using SIMD hardware (idealized memory system). 4

2.1 Latency hiding with fine-grained multithreading. 11

2.2 Baseline GPU microarchitecture. 13

2.3 Detail of a shader core. 15

3.1 Implementation of immediate post-dominator based reconvergence. 19

3.2 Performance loss for PDOM versus SIMD warp size. 21

3.3 Contrived example for reconvergence point limit study. 22

3.4 Impact of predicting optimal SIMD branch reconvergence points. 23

4.1 Dynamic warp formation example. 25

4.2 Register file configurations. 26

4.3 Implementation of dynamic warp formation and scheduling. 28

4.4 Implementation of the warp pool. 30

4.5 Implementation of majority scheduling policy. 32

5.1 Overview of GPGPU-Sim 37

5.2 Software design of pipeline stages in the shader core model. 38

5.3 Dram access model. 47

5.4 DRAM organization overview and simplified DRAM bank state diagram. 49

6.1 Performance comparison of NREC, PDOM, and DWF versus MIMD. 57

6.2 Comparison of warp scheduling policies. 59

6.3 Warp size distribution. 59
6.4 Dynamic behaviour of Black-Scholes using DWF with Majority scheduling policy. 60
6.5 Dynamic behaviour of Black-Scholes using DWF with PDOM Priority scheduling policy. 61
6.6 Impact of resource limit on Max-Heap. 62
6.7 Impact of lane aware scheduling. 63
6.8 Performance of PDOM, DWF and MIMD with cache bank conflict. 65
6.9 Effect of SIMD warp size. 66
6.10 Cumulative distribution of warp pool occupancy 67

Acknowledgements

The work presented in this thesis would not have been possible without the help from several individuals and organizations. First, I would like to thank my supervisor, Professor Tor Aamodt, for his guidance and support for the last two years. I would also like to thank the members of my M.A.Sc. thesis committee, Professor Guy Lemieux and Professor Steve Wilton, for their valuable feedback.

I am in debt to my colleagues in the computer architecture group, Ivan Sham, George Yuan, Henry Wong, Xi Chen, and Ali Bakhoda, for the fun and insightful discussions we had, and for their precious time spent reviewing my thesis (and the conference paper that it is based on). I owe Professor Konrad Walus and the Microsystems and Nanotechnology (MiNa) Research Group (especially Nicholas Geraedts) a special thanks for sharing their computing cluster with us for data collection.

My peers in the Department of Electrical and Computer Engineering in University of British Columbia have been helpful and entertaining to work with, and they made my experience in graduate school an unforgettable one. In particular, I would like to thank Derek Ho, David Mak, Larix Lee, Samer Al-Kiswany, Abdullah H. Gharaibeh, Elizeu Santos-Neto, David Boen and Sunny Yuen.

I am very grateful to my family. My parents has been fully supportive of my decisions to pursue a graduate degree. The timely encouragement from my mother has been heart-warming. My father's interest in electronics and computers inspired me in pursuing a career in this field. Despite being a physician himself, he had taught me a great deal about computers and essentially shaped my very first impression of the computers in my childhood.

Financial support for this research was provided through a Natural Sciences and Engineering Research Council (NSERC) of Canada 'CGS M' award.

Chapter 1

Introduction

The computation potential of a single chip improves as semiconductor process technology continues to scale and transistor density increases exponentially following "Moore's Law" [48]. Leveraging process technology scaling to improve the performance of real world applications has always been a goal in computer architecture, but has become increasingly challenging as power limitations restrict clock frequency scaling [52]. Now, perhaps more than before, hardware must exploit parallelism *efficiently* to improve performance.

Single-instruction, multiple-data (SIMD) instruction scheduling is a technique for exploiting data level parallelism efficiently by amortizing data-independent control hardware across multiple processing elements. Multiple data elements are grouped and processed in "lock-step" in parallel with a single instruction. This hardware simplification, however, restricts these processing elements to have uniform dynamic control flow behavior. A data dependent branch, in which each processing element resolves to a different outcome, generates a hazard, known as *branch divergence* [67], on the SIMD hardware. Handling of this hazard usually involves serializing the execution of these grouped processing elements, lowering the utilization of the SIMD hardware. For some control-flow intensive applications, this performance penalty outweighs the benefit of using SIMD hardware.

Nevertheless, SIMD has gained popularity in many applications that require little control flow flexibility. Most of these applications, such as 3D rendering and digital signal processing [36, 38, 42, 53, 71], deal with large data sets and require frequent access to off-chip memory with long access latency. Specialized processors for these applications, such as graphics processing units (GPUs), often use *fine-grained multithreading* to proactively hide these long access latencies. However, as these specialized processors start offering more and more programmability to meet the demand of their users, *branch divergence* with SIMD hardware becomes a major performance

bottleneck.

This thesis proposes and evaluates a novel hardware mechanism, *dynamic warp formation*, for improving performance of control-flow intensive applications on a SIMD architecture with fine-grained multithreading. While this proposed mechanism is described in this thesis in the context of GPU microarchitecture, it is equally applicable to any microarchitecture that uses SIMD and fine-grained multithreading [13, 23, 25].

The rest of this chapter describes the motivation and background for this thesis, the methodology employed, the contribution of this thesis, and finally summarizes the thesis's organization.

1.1 Motivation

Until recently, the dominant approach for exploiting parallelism has been to extract more instruction level parallelism (ILP) from a single thread through increasingly complex scheduling logic and larger caches. As diminishing returns to ILP now limit performance of single thread applications [1], attention has shifted towards using additional resources to increase throughput by exploiting explicit thread level parallelism in software. In contrast to instruction level parallelism, which mainly relies on hardware instruction scheduling logic for improving performance, thread level parallelism is explicitly defined in the software as threads (sections of code that can be executed in parallel), and the hardware simply provides support for executing these threads in parallel to improve performance. The simplest way to do so is to have multiple copies of the same processor on a chip, an approach known as a chip multiprocessors (CMP). This forces software developers to share the responsibility for improving performance, but saves significant effort in hardware design verification while potentially yielding a greater performance gain in comparison to providing additional cache, for example.

The modern *graphics processing unit* (GPU), a hardware accelerator for 3D rendering widely available on commodity computer systems, can be viewed as an example of this throughput oriented approach [9, 40, 61]. Earlier generations of GPUs consisted of fixed function 3D rendering pipelines. This required new hardware to enable new real-time rendering techniques, which impeded the adoption of new graphics algorithms and thus motivated the introduction of programmability, long available in traditional offline computer animation [77], into GPU

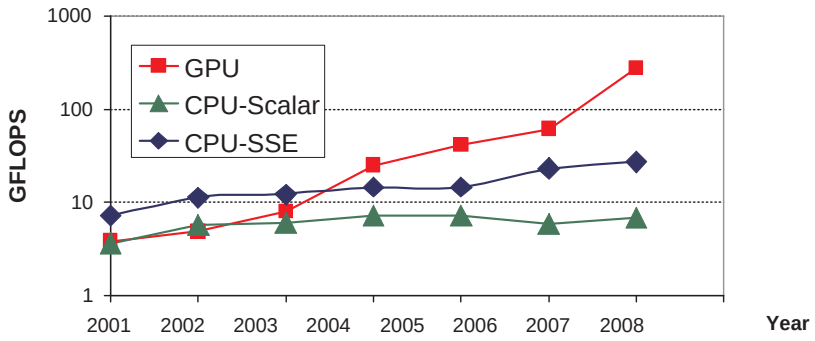


Figure 1.1: Floating-Point Operations per Second for the CPU and GPU

hardware for real-time computer graphics. In modern GPUs, much of the formerly hardwired pipeline is replaced with programmable hardware processors that run a relatively small *shader program* on each input vertex or pixel [40]. Shader programs are either written by the application developer or substituted by the graphics driver to implement traditional fixed-function graphics pipeline operations. The compute model provided by modern graphics processors for running non-graphics workloads is closely related to that of *stream processors* [18, 63].

The programmability of shader hardware has greatly improved over the past decade, and the shader processors of the latest generation GPUs are Turing-complete. Together with the impressive theoretical computation power of GPU when compared to conventional CMPs (see Figure 1.1), this opens up exciting new opportunities to speed up "general purpose" (i.e., non-graphics) applications. Based upon experience gained from pioneering efforts to generalize the usage of GPU hardware [9, 61], GPU vendors have introduced new programming models and associated hardware support to further broaden the class of applications that may efficiently use GPU hardware [3, 58].

Even with a general-purpose programming interface, mapping existing applications to the parallel architecture of a GPU is a non-trivial task. Although some applications can achieve speedups of up to 431 times over a modern CPU [26, 27], other applications, while successfully parallelized on different hardware platforms, show little improvement when mapped to a GPU [7]. One major challenge for contemporary GPU architectures is efficiently handling control flow in shader programs [67]. The reason is that, in an effort to improve computation

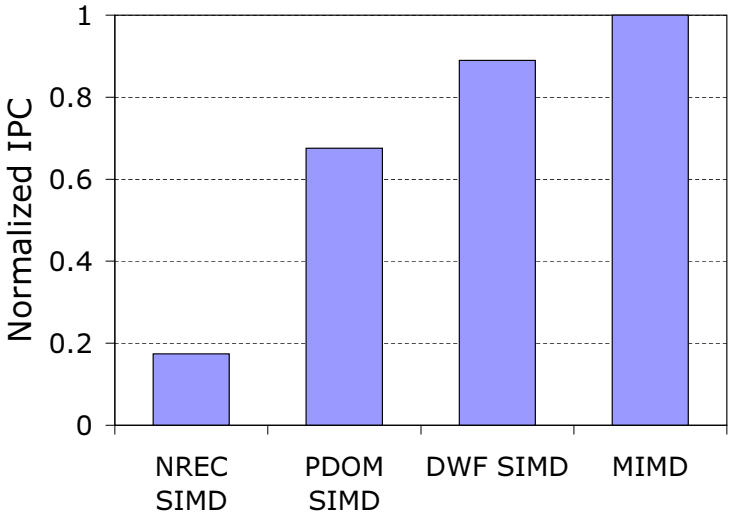


Figure 1.2: Performance loss due to branching when executing scalar SPMD threads using SIMD hardware (idealized memory system).[2]

density, modern GPUs typically batch together groups of individual threads running the same shader program, and execute them together in lock step on a SIMD pipeline [43, 45, 47, 49, 58]. Such thread batches are referred to as warps[1] by NVIDIA [58, 67]. This approach has worked well [37] for graphics operations such as texturing [6, 11], which historically have not required branch instructions. However, when shader programs *do* include branches, the execution of different threads grouped into a warp to run on the SIMD pipeline may no longer be uniform across SIMD elements. This causes a hazard in the SIMD pipeline [49, 80] known as branch divergence [43, 67]. We found that naïve handling of branch divergence incurs a significant performance penalty on the GPU for control-flow intensive applications relative to an ideal *multiple-instruction, multiple-data* (MIMD) architecture with the same peak IPC capability (See Figure 1.2).

[1] In the textile industry, the term "warp" refers to "the threads stretched lengthwise in a loom to be crossed by the weft" [21].

[2] *NREC SIMD* and *PDOM SIMD* are described in Chapter 3 while *DWF SIMD* is described in Chapter 4. Microarchitecture and Benchmarks are described in Chapter 2 and Chapter 5 respectively.

1.2 Contributions

This thesis makes the following contributions:

1. It quantifies the performance gap between the immediate post-dominator branch reconvergence mechanism and the performance that would be obtained on a MIMD architecture with support for the same peak number of operations per cycle. Thus, highlighting the importance of finding better branch handling mechanisms.

2. It proposes and evaluates a novel hardware mechanism, dynamic warp formation, for regrouping processing elements of individual SIMD warps on a cycle-by-cycle basis to greatly improve the efficiency of branch handling.

3. It highlights quantitatively that warp scheduling policy (the order in which the warps are issued from the scheduler) is an integral part to both the performance and area overhead of dynamic warp formation, and proposes an area efficient implementation of a well-performing scheduling policy.

4. It proposes and evaluates a detailed hardware implementation of dynamic warp formation and scheduling.

5. It provides an extensive simulation infrastructure for enabling future research on GPU architectures optimized to support non-graphics applications.

In particular, for a set of data parallel, non-graphics applications ported to our modern GPU-like SIMD streaming processor architecture, we find the speedup obtained by reconverging diverging threads within a SIMD warp at the immediate post-dominator of the diverging branch obtains a speedup of 45% over not reconverging. Furthermore, dynamically regrouping scalar threads into SIMD warps on a cycle by cycle basis increases speedup further to 114% (47% speedup versus reconverging at immediate post-dominator). We estimate the hardware required by this regrouping mechanism adds 8% to the total chip area.

1.3 Organization

The rest of the thesis is organized as follows:

- Chapter 2 provides an overview of the fundamental concepts and the baseline GPU microarchitecture that this thesis builds on.
- Chapter 3 describes the immediate post-dominator control-flow reconvergence mechanism, which represents a baseline equivalent to the performance of contemporary SIMD control-flow handling mechanisms.
- Chapter 4 describes our proposed dynamic warp formation mechanism, an improvement to the baseline. This is the main contribution of the work.
- Chapter 5 describes the simulation methodology of the proposed GPU microarchitecture, including a detailed description of *GPGPU-Sim*, a novel GPU microarchitecture simulator.
- Chapter 6 describes our experimental results.
- Chapter 7 gives an estimation of the area overhead for implementing dynamic warp formation.
- Chapter 8 describes related work.
- Chapter 9 summarizes this thesis and suggests future work.

Chapter 2

Background

This chapter provides an overview of the fundamental concepts, the compute model and the baseline SIMD GPU microarchitecture used for the rest of this thesis. This baseline is representative of contemporary GPUs used for accelerating non-graphics applications.

2.1 Fundamental Concepts

This section discusses the fundamental concepts and techniques that this thesis builds on. In particular, we describe several techniques fundamental to the contributions of this thesis: thread level parallelism, data level parallelism, single-instruction multiple-data, multiple-instruction multiple-data, and fine-grained multithreading.

2.1.1 Thread-Level Parallelism

Thread-level parallelism (TLP) refers to the parallelism that is specified explicitly by the software developer as threads in an application [76]. These threads run concurrently in the system, and each thread is expected to progress independently with a register set of its own. In some systems, these threads can be explicitly synchronized through message passing or locks[3] and barriers[4]. TLP can be exploited by adding extra processors to a system so that more threads execute in parallel (provided that there are enough threads). However, as mentioned later in Section 2.1.5, TLP can also be exploited as a way to increase efficiency in a system with high memory access latency.

[3]Mutually exclusive access to data shared among threads [16].

[4]Global synchronization among a set of threads: none get pass the barrier until all threads within the set have arrived [16].

2.1.2 Data-Level Parallelism

Data-level parallelism (DLP) exists in an application when a large pool of data is processed in a regular fashion, where each element of the output data is only dependent on a few elements from the input data pool [56]. A classic example of such an application is matrix multiplication, in which each element in the destination matrix is calculated by a sum of products of elements from the source matrices. As the outcome of each element does not depend on values of other output elements, multiple elements in the destination matrix can be calculated in parallel. Data-level parallelism can be exploited by adding extra processors, and distributing available work to these processors. This allows data-level parallelism to scale easily as data set increases—the larger the data set, the more work readily distributable to more processors [56]. In contrast to thread-level parallelism, which assigns each processing unit a unique thread, and instruction-level parallelism, which exploits independent operations within a single instruction stream [24], the regularity of DLP allows it to be exploited in a much more efficient manner, as discussed next.

2.1.3 Single-Instruction, Multiple-Data (SIMD)

SIMD has its roots in vector processors used for scientific computation. Vector processors are effective for applications which involve repeating a set of identical operations across a large amount of data (e.g., vector multiplication). As these repeated operations are independent with each other, they may be executed in parallel on different data. Vector processors exploit this DLP efficiently with an instruction set architecture (ISA) that operates on vectors of data. This allows control hardware to be shared by multiple processing elements, greatly reducing the hardware cost required to scale up the width of parallel operations. All that is required is to attach extra ALUs to the control signal bus driven by a common control logic and provide them with data. At a minimum, this saves instruction bandwidth and control logic otherwise required for adding extra processors to exploit the same amount of DLP. However, by sharing the control hardware, the processing elements are restricted to execute in lockstep, i.e., they all execute the same instruction and advance to the next instruction at the same time.

Today, SIMD hardware exists in commodity desktop computers in two main forms: special

purpose accelerators and short-vector instruction set architecture (ISA) extensions.

Short-vector ISA extensions refer to the SIMD instruction set extensions for general purpose CPUs, such as the Streaming SIMD Extension (SSE) for x86 architecture [73] and AltiVec for POWER architecture [44]. These SIMD instruction set extensions operate on short vector registers (64-bit or 128-bit) that hold multiple data elements that will be operated on with a single SIMD instruction. Multiple SIMD instructions are needed to process vectors that exceed the length of a single vector register. While these ISA extensions allow general purpose CPUs to exploit DLP in a limited way, the main focus of general purpose CPUs remains to be fast execution of serial programs.

Historically, special purpose accelerators are hardware that is less tightly integrated with a CPU and is designed to assist the CPU in specific applications. For example, Graphics Processing Units (GPUs) are dedicated processors specialized in 3D rendering. In GPUs, SIMD is used to exploit the DLP nature of 3D rendering—each pixel or vertex can be processed independently (see Chapter 2 for more detail). Other special purpose accelerators, such as digital signal processors (DSPs), are also using SIMD to exploit DLP in their application domains [71].

2.1.4 Multiple-Instruction, Multiple Data

In this thesis, the performance of various SIMD control flow techniques is often compared to that of a *multiple-instruction, multiple-data* (MIMD) architecture. According to Flynn's taxonomy [20], any architecture that is not constrained by the lockstep execution of processing elements as in SIMD is classified as a MIMD architecture. With this definition, any multiprocessor architecture is a MIMD architecture. In the context of this thesis, a MIMD architecture refers to an architecture with the same configuration as our baseline SIMD architecture in Chapter 2, except that the processing elements inside a shader core are free to execute any instruction. These processing elements still share a common scheduler, which can schedule any available thread to any free processing elements for execution.

2.1.5 Fine-Grained Multithreading

Fine-Grained Multithreading (FGMT) is a technique allowing multiple threads to interleave their execution on one or more hardware processors.

This technique differs from the traditional multitasking (time sharing) provided by an operating system on an uniprocessor CPU. In a FGMT processor, switching to another thread does not require flushing the pipeline nor offloading architectural registers to the main memory. Instructions from multiple different processes co-exist in the pipeline of each single hardware processor at the same time and the context (program counter and registers) of each thread is kept in the register file and stays there until the thread terminates. Every cycle, the hardware scheduler selects and issues an instruction from one of the threads sharing the processor. The details of thread selection are hidden from the operating system, so a single processor with FGMT appears as multiple single threaded processors.

FGMT was first proposed on the CDC 6600 as "barrel processing", a flexible interface to allow a single central processor to interface with multiple slower peripheral processors [75]. The Heterogeneous Element Processor (HEP) computer [31] later employed FGMT as a technique to provide a scalable interface for multi-threaded programs. In a HEP computer, each Process Execution Module (PEM) can hold up to 128 threads. These threads communicate with each other through the main memory, so that multi-threaded programs written for a single PEM can also be executed by multiple PEMs sharing the same memory space. The threads are expected to operate on different data, so that their instructions can exist in different stage in the PEM's pipeline at the same time without causing dependency hazards. In this way, instructions from different threads can be aggressively scheduled into the pipeline without incurring the performance penalty that a single-threaded processor would normally suffer (branch mispredictions and data hazard stalls).

The Horizon [74] architecture also uses FGMT to provide a scalable interface, but focuses on using this to hide the latency of memory accesses, which can take hundreds of cycles when hundreds of processors and memory modules are connected via a message-passing interconnection network.

Figure 2.1 shows a simplified version of the operation of FGMT. At any time, each pipeline

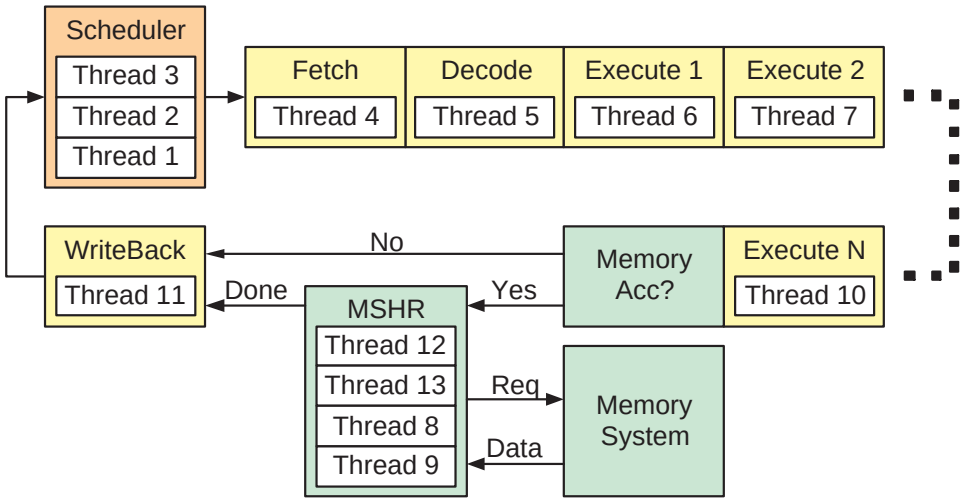


Figure 2.1: A simplified illustration of how fine-grained multithreading can hide memory latency. Notice it can also tolerate various long latency operations such as floating point division (represented by the long execution stages). MSHR = miss status hold register [34]

stage (Fetch, Decode, Execute) can be occupied by a different thread; Provided there are more threads than pipeline stages, the pipeline can be fully utilized. Threads that require access to memory are stored in the memory unit (MSHR in Figure 2.1). Their request will be sent to the memory system, while the threads wait inside the memory unit. In the meantime, other threads that are not blocked by the memory will be executed in the pipeline to keep it utilized. After threads inside the memory unit obtain their data from the memory system, they write the data to the register file and resume execution in the pipeline. With this organization, longer memory latency can be hidden using a larger number of threads sharing the pipeline.

While fine-grained multithreading allows efficient use of the pipeline, it requires multi-threaded software. Also, to be able to switch quickly between threads, the hardware processor stores the architectural states (registers and program counter) of all its threads. This requires a significantly larger register file than the ones in single-threaded processors. For instance, the Horizon architecture [74] allows up to 128 threads sharing a single processor, and each thread contains 32 general purpose registers, requiring a register file of 4096 registers per processor. For these reasons, FGMT's popularity remains limited to architectures that optimize for applications with plentiful thread-level parallelism.

Sun's Niagara processor [35] features a FGMT architecture. The target applications for

Niagara are server and database applications, which contains significant thread-level parallelism. FGMT enables Niagara to deliver higher power efficiency (performance per Watt ratio) compared to single threaded superscalar processors, while cache[5]misses are still a predominant factor for the performance of Niagara [35].

Similar to FGMT, simultaneous multithreading (SMT) is a hardware technique that has been used in general purpose CPUs to improve throughput by exploiting thread-level parallelism [76]. To understand the difference between FGMT and SMT it is first necessary to point out that all high performance CPUs today employ an approach to instruction processing called superscalar execution[6]. In contrast to FGMT, instead of allowing only one thread to issue an instruction to the pipeline each cycle, SMT allows multiple threads to compete for multiple superscalar issue slots in the same cycle, meaning that the superscalar pipeline's execution stage can accept instructions from different threads in a single cycle. This thesis, however, uses processor cores with a single-issue, SIMD pipeline, so FGMT is the only option.

Finally, graphics processing units (GPUs) uses FGMT to proactively hide memory access latency just like the Horizon architecture, but each core has a SIMD pipeline to increase the computation density [3, 39]. The SIMD pipeline shares data-independent control logic across multiple stream processors.

2.2 Compute Model

In this thesis, we have adopted a compute model that is similar to NVIDIA's CUDA programming model [58]. In this compute model, the application starts off as a single program running on the CPU. At some point during execution, the CPU reaches a kernel call and spawns a parallel section to the GPU to exploit data-level parallelism. At this point, the CPU will then stop its execution and wait for the GPU to finish the parallel section[7]. This sequence can repeat multiple times until the program completes.

Each parallel section consists of a collection of threads executing the same code which we call

[5]A cache is a small and fast storage area close to the processor that reduces average memory access latency by buffering data frequently accessed by the processor.

[6]A superscalar pipeline is designed to harness instruction-level parallelism within a single thread [24]. It has the capability to issue multiple instructions to multiple functional units in a single cycle.

[7]A more recent version of CUDA allows the CPU to continue execution in parallel with the GPU [59].

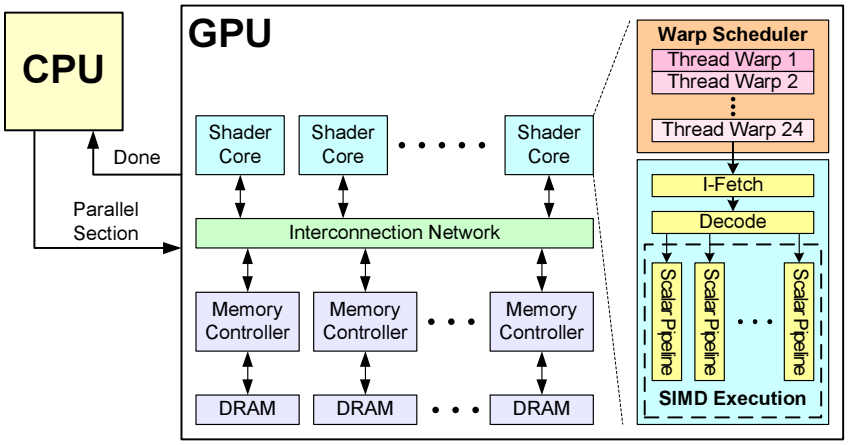


Figure 2.2: Baseline GPU microarchitecture. Blocks labeled 'scalar pipeline' include register read, execute, memory and writeback stages.

a *shader program*. Similar to many thread programming APIs, a shader program is encapsulated as a function call. In our implementation, at least one of the arguments is dedicated to pass in the thread ID, which each thread uses to determine its behaviour during the parallel section. For example, each thread may use its ID to index a different element in a vector. In this sense, the programming model employed in this thesis is essentially the *Single Program, Multiple Data* (SPMD) model commonly used to program shared memory multiprocessors.

All threads within a parallel section are expected to execute in parallel and share the same memory space. Unlike most shared memory multiprocessors, cache coherence and memory consistency are not enforced in our model as the threads are expected to be largely independent of each other. In the present study, to ensure that threads of the next parallel section will have access to the correct data, data caches are flushed and a memory fence operation is performed at the end of each parallel section.

2.3 SIMD GPU Microarchitecture

Figure 2.2 illustrates the baseline microarchitecture used in the rest of this thesis. In this figure, each shader core executes multiple parallel threads from the same parallel section, with each thread's instructions executed in-order by the hardware.[8] The multiple threads on a given core

[8]Our shader core is similar to CUDA's notion of a Streaming Multiprocessor (SM) [39].

are grouped into SIMD warps by the hardware scheduler. Each warp of threads executes the same instruction simultaneously on different data values in parallel scalar pipelines. Instructions read their operands in parallel from a highly banked register file. Memory requests access a highly banked data cache and cache misses are forwarded to the memory controller and/or cache levels closer to memory via an interconnection network. Each memory controller processes memory requests by accessing its associated DRAM, possibly in a different order than the requests are received to reduce row activate and precharge overheads [64]. The interconnection network we simulated is a crossbar with a parallel iterative matching allocator [17].

Since our focus in this thesis is non-graphics applications, graphics-centric details are omitted from Figure 2.2. However, traditional graphics processing still heavily influences this design: The use of SIMD hardware to execute SPMD software is heavily motivated by the need to balance "general purpose" compute kernel execution with a large quantity of existing graphics shaders that have simple control-flow [67]. However, it is important to recognize that shader programs for graphics may make increasing use of control flow operations in the future, for example to achieve more realistic lighting effects.

2.4 Latency Hiding

Since cache hit rates tend to be low for streaming applications, performance would be severely penalized if the pipeline had to stall for every memory request that misses the cache. This is especially true when the latency of memory requests can take several hundreds of cycles due to the combined effects of contention in the interconnection network and row-activate and precharge overheads at the DRAM. While traditional microprocessors can mitigate the effects of cache misses using out-of-order execution, a more compelling approach when software provides the parallelism is to interleave instruction execution from different threads.

With a large number of shader threads multiplexed on the same execution resources, our architecture employs fine-grained multi-threading, where individual threads are interleaved by the fetch unit [74, 75] to proactively hide the potential latency of stalls before they occur (as described in Section 2.1.5). As illustrated by Figure 2.3(a), instructions from multiple shader threads are issued fairly in a round-robin queue. When a shader thread is blocked by

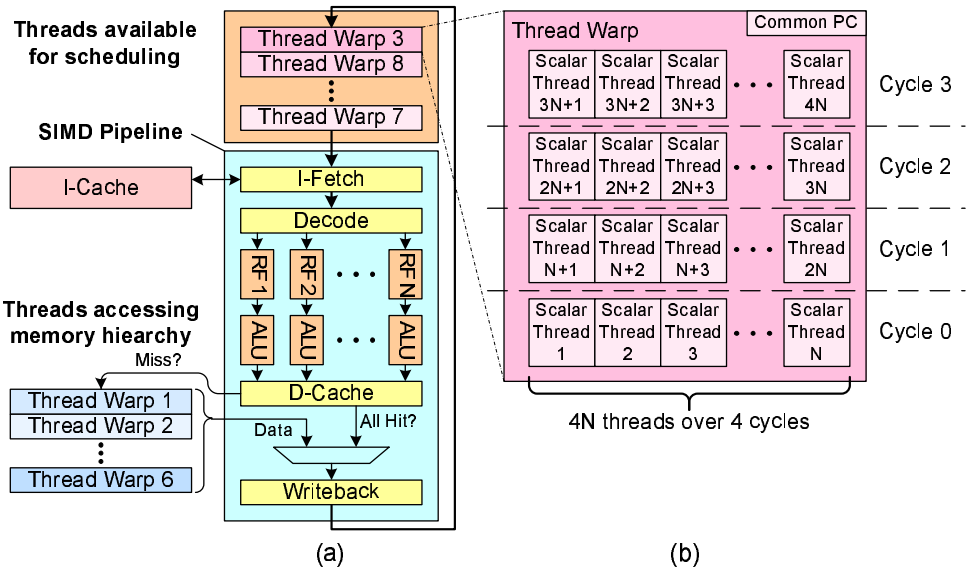


Figure 2.3: Detail of a shader core. (a) Using barrel processing to hide data memory access latency. N is the SIMD width of the pipeline. (b) Grouping 4N scalar threads into a SIMD warp executed over 4 cycles.

a memory request, the corresponding shader core simply removes that thread's warp from the pool of "ready" warps and thereby allows other shader threads to proceed while the memory system processes its request. With a large number of threads (768 per shader core, 12,288 in total in this thesis, similar to the Geforce 8800GTX) interleaved on the same pipeline, barrel processing effectively hides the latency of most memory operations since the pipeline is occupied with instructions from other threads while memory operations complete. Barrel processing also hides the pipeline latency so that data bypassing logic can be omitted to save area with minimal impact on performance. In this thesis, we also simplify the dependency check logic design by restricting each thread to have at most one instruction running in the pipeline at any time.

An alternative to barrel processing on a large number of threads is to interleave fewer threads, but provide a large number of registers to each thread [28]. Each thread executes on the pipeline until it encounters a dependency hazard, at which time the pipeline will switch its execution to another thread. Latency hiding is essentially achieved via software loop unrolling, which generates independent instructions to be inserted between a memory access and instructions depending on the access.

2.5 SIMD Execution of Scalar Threads

While barrel processing can hide memory latency with relatively simple hardware, a modern GPU must also exploit the explicit parallelism provided by the stream programming model [9, 58] associated with programmable shader hardware to achieve maximum performance at minimum cost. SIMD hardware [8] can efficiently support SPMD program execution provided that individual threads follow similar control flow. Figure 2.3(b) illustrates how instructions from multiple (M = 4N) shader threads are grouped into a single SIMD warp and scheduled together into multiple (N) scalar pipelines over several (M/N = 4N/N = 4) cycles. The multiple scalar pipelines execute in "lock-step" and all data-independent logic may be shared to greatly reduce area relative to a MIMD architecture. A significant source of area savings for such a SIMD pipeline is the simpler instruction cache support required for a given number of scalar threads.

SIMD instruction processing can also be used to relax the latency requirement of the scheduler and simplify the scheduler's hardware. With a SIMD warp size wider than the actual SIMD hardware pipeline, the scheduler only needs to issue a single warp every M/N cycles (M = warp size, N = pipeline width) [39]. The scheduler's hardware is also simplified as it has fewer warps to manage. This technique is used in the NVIDIA's GeForce 8 series [58], and the performance evaluations presented in this thesis assume the use of this technique (see Section 4.4 of Chapter 4).

2.6 Summary

In this chapter, we have given an overview of the fundamental concepts and the baseline architecture to be used for the rest of this thesis. Details of the interconnection network and memory subsystem are described in Chapter 5. In the following chapter, we describe a reconvergence mechanism to handle branch divergence in SIMD hardware, which represents a reasonable proxy of the performance of various contemporary control-flow handling mechanisms for SIMD. The performance of this reconvergence mechanism is compared against our proposed dynamic warp formation and scheduling mechanism in Chapter 6.

Chapter 3

SIMD Control Flow Support

To ensure the hardware can be easily programmed for a wide variety of applications, some recent GPU architectures allow individual threads to follow distinct program paths [3, 46, 58, 60]. We note that where it applies, predication [2] is a natural way to support such fine-grained control flow on a SIMD pipeline. However, predication does not eliminate branches due to loops and introduces overhead due to instructions with false predicates.

To support distinct control flow operation outcomes on distinct processing elements with loops and function calls, several approaches have been proposed: Lorie and Strong describe a mechanism using mask bits along with special compiler-generated priority encoding "else" and "join" instructions [43]. Lindholm and Moy [49] describe a mechanism for supporting branching using a serialization mode. Woop et al. [80] describe the use of a hardware stack and masked execution. Kapasi et al. [32] propose *conditional streams*, a technique for transforming a single kernel with conditional code to multiple kernels connected with inter-kernel buffers.

The effectiveness of a SIMD pipeline is based on the assumption that all threads running the same shader program expose identical control-flow behaviour. While this assumption is true for most existing graphics rendering routines [67], many existing parallel non-graphics applications (and potentially, future graphics rendering routines) tend to have more diverse control-flow behaviour. When a data-dependent branch leads to different control flow paths for different threads in a warp, *branch divergence* occurs because a SIMD pipeline cannot execute different instructions in the same cycle. The following sections describe two techniques for handling branch divergence, both of which were implemented in the simulator developed as part of this thesis.

The preliminary version of SIMD serialization mechanism presented in Section 3.1 was contributed by Henry Tran. The initial implementation of the reconvergence mechanism presented

in Section 3.2 and the limit study presented in Section 3.3 were contributed by Ivan Sham.

3.1 SIMD Serialization

A naïve solution to handle branch divergence in a SIMD pipeline is to serialize the threads within a warp as soon as the program counters diverge. A single warp with branch divergence (threads taking different execution paths) is separated into multiple warps each containing threads taking the same execution path. These warps are then scheduled and executed independently of each other, and they never reconverge back into a single warp. While this method is easy to understand and implement, it achieves poor performance. Without branch reconvergence, threads within a warp will continue diverging until each thread is executed in isolation from other threads in the original warp, leading to very low utilization of the parallel functional units as shown by *NREC SIMD* in Figure 1.2.

3.2 SIMD Reconvergence

Given the drawback of serialization it is desirable to use a mechanism for *reconverging* control flow. The opportunity for such reconvergence is illustrated in Figure 3.1(a). In this example, threads in a warp diverge after reaching the branch at A. The first three threads encounter a taken branch and go to basic block[9] B (indicated by the bit mask 1110 in Figure 3.1(a) inside basic block B), while the last thread goes to basic block F (indicated by the bit mask 0001 in Figure 3.1(a) inside basic block F). The three threads executing basic block B further diverge to basic blocks C and D. However, at basic block E the control flow paths reach a join point [50]. If the threads that diverged from basic block B to C wait before executing E for the threads that go from basic block B to basic block D, then all three threads can continue execution simultaneously at block E. Similarly, if these three threads wait after executing E for the thread that diverged from A to F then all four threads can execute basic block G simultaneously. Figure 3.1(b) illustrates how this sequence of events would be executed by the SIMD function units. In this part of the figure solid arrows indicate SIMD units that are active.

[9] A basic block is a piece of code with a single entry and exit point.

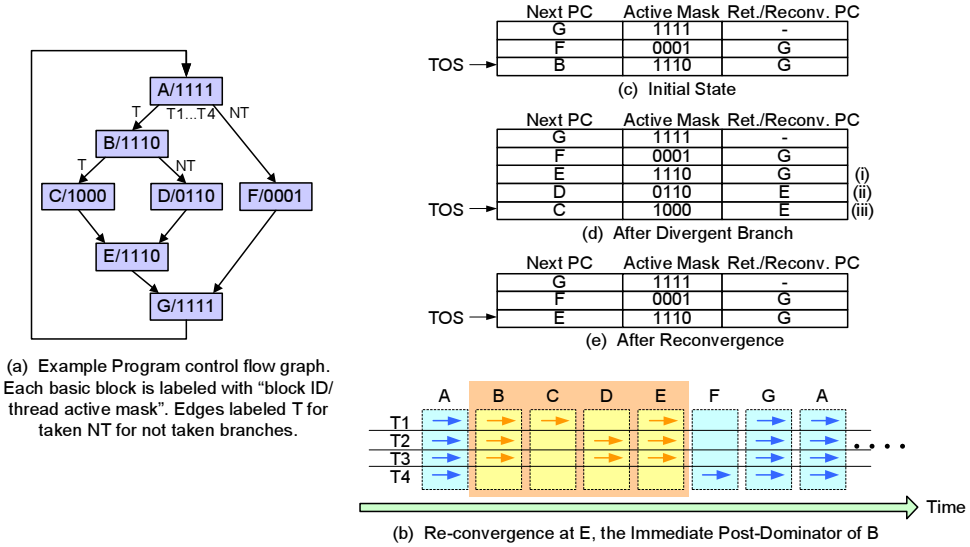


Figure 3.1: Implementation of immediate post-dominator based reconvergence.

Event	**Action**
No Divergence (single next PC)	Update the next PC field of the top of stack (TOS) entry to the next PC of all active threads in this warp.
Divergence (multiple next PC)	Modify the next PC field of the TOS entry to the reconvergence point. For each unique next PC of the warp, push a new entry onto the stack with next PC field being the unique next PC and the reconv. PC being the reconvergence point. The active mask of each entry denotes the threads branching to the next PC value of this entry.
Reconvergence (next PC = reconv. PC of TOS)	Pop TOS entry from the stack.

Table 3.1: Operational rules of the stack for reconvergence mechanism.

The behaviour described above can be achieved using a stack based reconvergence mechanism [80]. In this thesis, each warp has a private stack tracking its control flow status. Each entry in the stack consists of three fields: next PC, active mask and reconvergence PC. The next PC field of the top of stack (TOS) entry indexes to the instruction that the warp will execute at its next issue to the pipeline. The active mask field of the TOS entry indicates which thread in the warp is currently active. The reconvergence PC field specify the point in the shader program where these active threads will wait for other threads in the same warp before proceeding further down the program. Table 3.1 summarizes the operational rules of the stack.

An example illustrating how these rules implement the reconvergence mechanism is shown in Figure 3.1(c,d,e). Here we show how the stack is updated as the group of three threads in Figure 3.1(a) that execute B diverge and then reconverge at E. Before the threads execute the diverging branch at B, the state of the stack is as shown in Figure 3.1(c). When the branch divergence is detected, the stack is modified to the state shown in Figure 3.1(d). The changes that occur are the following:

1. The original top of stack (TOS) in Figure 3.1(c), also at (i) in Figure 3.1(d), has its next PC field modified to the instruction address of the reconvergence point E (the address could be specified through an extra field in the branch instruction).

2. A new entry (ii) is allocated onto the stack in Figure 3.1(d) and initialized with the reconvergence point address (E) along with the next PC value of the fall through of the branch (D), and a mask (0110) encoding which processing elements evaluated the branch as "not-taken".

3. A new entry (iii) is allocated onto the stack with the same reconvergence point address (E), the target address (C) of the branch, and a mask (1000) encoding which processing elements evaluated the branch as "taken".

Note that the mechanism described above supports "nested" branch hammocks as well as data-dependent loops. For a SIMD warp size of N, the size of this stack is bounded to $2N$ entries per warp, which is all consumed when each thread in a warp is diverged to its own execution path. At this point, no new entry will be pushed onto the stack because a single thread never diverges, even if it is runnnig in a loop.

In this thesis we use the *immediate post-dominator* [50] of the diverging branch instruction as the reconvergence point[10]. A *post-dominator* is defined as follows: A basic block X post-dominates basic block Y (written as "X pdom Y") if and only if all paths from Y to the exit node (of a function) go through X. A basic block X, distinct from Y, *immediately post-dominates* basic block Y if and only if X pdom Y and there is no basic block Z such that X pdom Z and Z

[10]While Rotenberg et al. [65] also identified immediate post-dominators as control reconvergence points, to our knowledge we are the first to propose this scheme for SIMD control flow.

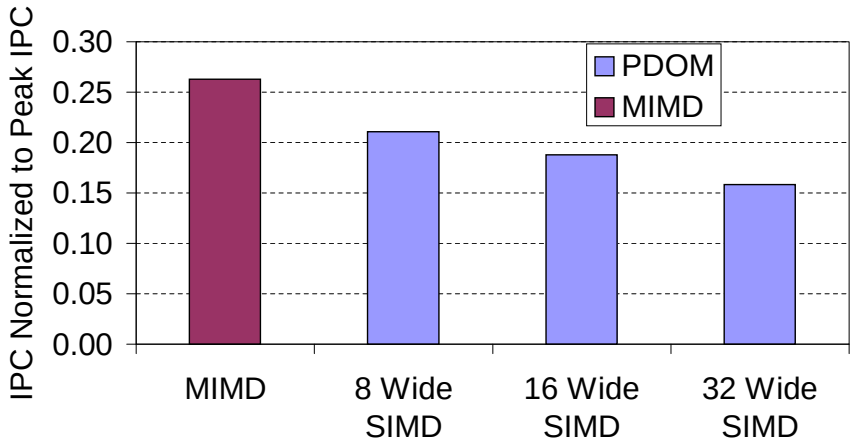


Figure 3.2: Performance loss for PDOM versus SIMD warp size (realistic memory system).[11]

pdom Y. Immediate post-dominators are typically found at compile time as part of the control flow analysis necessary for code optimization.

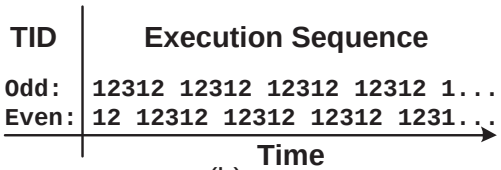
The performance impact of the immediate post-dominator reconvergence technique (which we abbreviate in the rest of this thesis as PDOM) depends upon the SIMD warp size. Figure 3.2 shows the harmonic mean IPC of the benchmarks described in Chapter 5 compared to the performance of MIMD hardware for 8, 16, and 32 wide SIMD execution assuming 16 shader cores. Hardware utilization decreases from 26% for MIMD to 21% for 8-wide, to 19% for 16-wide, and down to 16% for 32-wide.[11] This increase of performance loss is due to a higher impact of branch divergence when SIMD warps are widened and each warp contains more threads. Although PDOM works well, the MIMD performance indicates that a performance gain of 66% is still possible for a better branch handling mechanism if it can perfectly eliminate the performance loss due to branch divergence.

In the following section we explore whether immediate post-dominators are the "best" reconvergence points, or whether there might be a benefit to dynamically predicting a reconvergence point past the immediate post-dominator.

[11] The peak throughput remains the same for all three configurations of different SIMD widths. The configuration with a wider SIMD-width is implemented as a scheduler that issue a wider warp at a slower rate. See Section 2.5 of Chapter 2 for more detail.

```
   void shader_thread(int tid, int *data) {
1:     for (int i = tid % 2; i < 128; ++i) {
2:         if (i % 2)
3:             data[tid]++;
       }
   }
```

(a)



(b)

Figure 3.3: (a) Contrived example for which reconvergence at points beyond the immediate post-dominator yields the significant improvement in performance shown on Figure 3.4. The parameter `tid` is the thread ID. (b) Execution sequence for threads with odd or even `tid`. Note the example in Figure 3.4 used 2D arrays.

3.3 Reconvergence Point Limit Study

While reconverging at the immediate post-dominator recovers much of the performance lost due to branch divergence compared to not reconverging at all, Figure 3.3(a) shows an example where this reconvergence mechanism is sub-optimal. In this example, threads with an even thread ID diverge from those with an odd thread ID each iteration of the loop. If even threads allow the odd threads to "get ahead" by one iteration, all threads can execute in lock-step until individual threads reach the end of the loop. This suggests that reconverging at points beyond the immediate post-dominator may yield better performance. To explore this possibility we conducted a limit study assessing the impact of always predicting the best reconvergence point assuming oracle knowledge of each thread's future control flow.

For this limit study, dynamic instruction traces are captured from only the first 128 threads. SIMD warps are formed by grouping threads by increasing thread ID, and an optimal alignment for the instruction traces of each thread in a warp is found via repeated applications of the Needleman-Wunsch algorithm [54]. With four threads per warp, the optimal alignment is found by exhaustively searching all possible pair-wise alignments between threads within a warp. The best reconvergence points are then identified from the optimal alignment.

Figure 3.4 compares the performance of immediate post-dominator reconvergence versus the performance when reconverging at the predicted reconvergence points derived using this method assuming a warp size of 4. In this figure we assume an idealized memory system (all cache accesses hit) and examine both a contrived program with the behaviour abstracted in Figure 3.3 and the benchmarks described in Chapter 5 (depicted by the bars labeled "Real Pro-

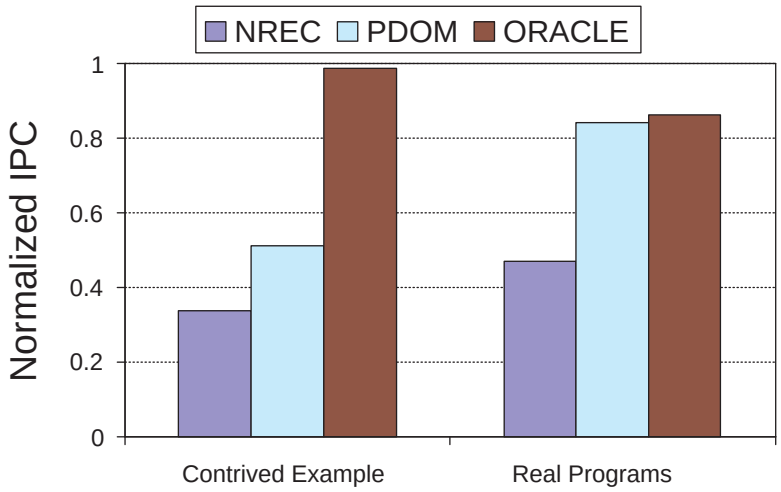


Figure 3.4: Impact of predicting optimal SIMD branch reconvergence points. NREC = No Reconvergence. PDOM = Reconvergence at immediate post-dominators. ORACLE = Reconvergence at optimal post-dominators.

grams"). While the contrived example experiences a 92% speedup with oracle reconvergence point prediction, the improvement on the real programs we studied is much less (2.6%). Interestingly, one of the benchmarks (bitonic sort) does have similar even/odd thread dependence as our contrived example. However, it also contains frequent barrier synchronizations that ensure loop iterations execute in lock-step. It is important to recognize the limitation of the preceding limit study: We explored a very limited set of applications and used short SIMD width, and a relatively small number of threads.

3.4 Summary

This chapter described a branch handling mechanism which reconverges control flow of threads at the immediate post-dominators of the diverging branches and compared it against MIMD hardware. It also performed a limit study to evaluate the potential performance gain of using more sophisticated reconvergence mechanisms. As this limit study seems to suggest that more sophisticated reconvergence mechanisms may not improve performance significantly, this thesis focuses on a mechanism which combines threads from distinct warps following branch divergence. This mechanism is discussed in the next chapter.

Chapter 4

Dynamic Warp Formation and Scheduling

While the post-dominator reconvergence mechanism described in the previous chapter is able to mitigate performance loss resulting from diverging branches, it does not fully utilize the SIMD pipeline relative to a MIMD architecture with the same peak IPC capability (losing 40% of performance for a warp size of 32 relative to MIMD). In this section, we describe our proposed hardware mechanism for recovering the lost performance potential of the hardware.

The performance penalty due to branch divergence is hard to avoid with only one thread warp since the diverged parts of the warp cannot execute simultaneously on the SIMD hardware in a single cycle. Dynamic warp formation attempts to improve upon this by exploiting the fine-grained multithreading aspect of the GPU microarchitecture: With fine-grained multithreading employed to hide memory access latency, there is usually more than one thread warp ready for scheduling in a shader core. Every cycle, the thread scheduler tries to form new warps from a pool of ready threads by combining scalar threads whose next program counter (PC) values are the same. As the shader program executes, diverged warps are broken up into scalar threads to be regrouped into new warps according to their branch targets (indicated by the next PC value of each scalar thread). In this way, the SIMD pipeline is fully utilized even when a shader program executes diverging branches.

Figure 4.1 illustrates this idea. In this figure, two warps, Warp x and Warp y, are executing the example program shown in Figure 4.1(a) on the same shader core and both suffer from branch divergence. Figure 4.1(b) shows the interleaved execution of both warps using the reconvergence technique discussed in Chapter 3.2, which results in the SIMD pipeline utilization below 50% when basic blocks C, D and F are executed. As shown in Figure 4.1(c), using dynamic

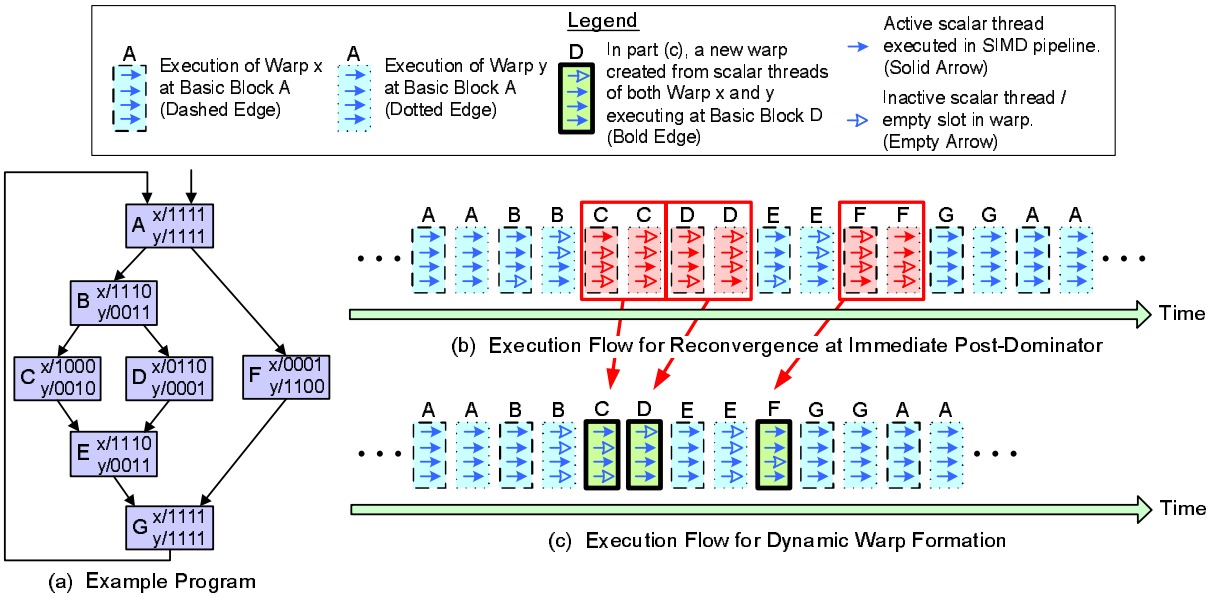


Figure 4.1: Dynamic warp formation example.

warp formation to regroup scalar threads from both warps in Figure 4.1(b) into a single warp in Figure 4.1(c) for these blocks can significantly increase the pipeline utilization (from 65% in Figure 4.1(b) to 77% in Figure 4.1(c)).

Implementing dynamic warp formation requires careful attention to the details of the register file, a consideration we explore in Section 4.1. In addition to forming warps, the thread scheduler also selects one warp to issue to the SIMD pipeline every scheduler cycle depending upon a scheduling policy. We explore the design space of this scheduling policy in detail in Section 4.3. We show that the thread scheduler policy is critical to the performance impact of dynamic warp formation in Chapter 6.

4.1 Register File Access

To reduce area and support a large number of ports in a SIMD pipeline, a well-known approach is to implement the register file in multiple banks, each accessible from a single scalar pipeline (or lane) of the SIMD pipeline as shown in Figure 4.2(a). This hardware is a natural fit when threads are grouped into warps "statically" before they begin executing instructions and each thread stays in the same lane until it is completed. For our baseline architecture, each register

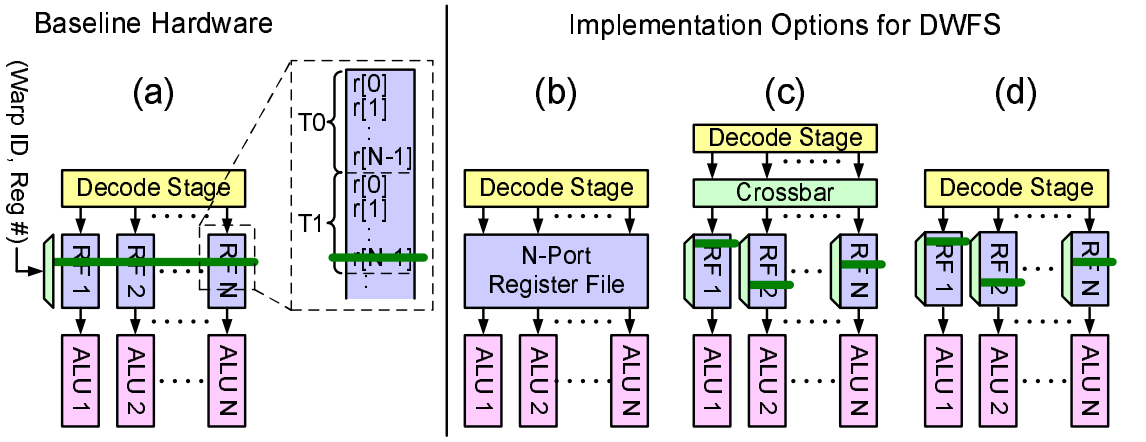


Figure 4.2: Register file configuration for (a) static warp formation, (b) ideal dynamic warp formation and MIMD, (c) unconstrained dynamic warp formation, (d) lane aware dynamic warp formation. The solid line running across each register file bank in (a), (c) and (d) represents whether individual banks are addressed by a common decoder (continuous line in part (a)) or each bank has its own decoder for independent addressing ((c) and (d)).

file bank contains a separate set of registers per warp. Each scalar thread from the same warp is statically assigned to a unique lane and always accesses the corresponding register file bank for that lane. The registers used by each scalar thread within a given lane are then assigned statically at a given offset based upon the warp ID.

However, so far we have not explicitly considered the impact of such static register assignment on dynamic warp formation. As described so far, dynamic warp formation would require each register to be equally accessible from all lanes, as illustrated in Figure 4.2(b). While grouping threads into warps dynamically, it is preferable to avoid the need to migrate register values with threads as they are regrouped into different warps. To accomplish this, the registers used by each scalar thread are assigned statically to the register banks in the same way as described above. However, if we dynamically form new warps *without* consideration of the "home" lane of a scalar thread's registers, we must design the register file with a crossbar as in Figure 4.2(c). Furthermore, warps formed dynamically may then have two or more threads with the same "home" lane, resulting in bank conflicts. These bank conflicts introduce stalls into all lanes of the pipeline and significantly reduce performance as shown in Chapter 6.

A better solution, which we call *lane aware* dynamic warp formation, ensures that each thread remains within its "home" lane. In particular, lane aware dynamic warp formation

assigns a thread to a warp only if that warp does not already contain another thread in the same lane. While the crossbar in Figure 4.2(c) is unnecessary for lane aware dynamic warp formation, the traditional hardware in Figure 4.2(a) is insufficient. When threads are grouped into warps "statically", each thread's registers are at the same "offset" within the lane, thus requiring only a single decoder. With lane aware dynamic warp formation, the offsets to access a register in a warp will not be the same in each lane. Instead, the offset in each lane is calculated according to the thread assigned to the lane in the dynamically formed warp. Note that each lane is still executing the same instruction in any given cycle—the varying offsets are a byproduct of supporting fine grained multithreading to hide memory access latency combined with dynamic warp formation. This yields the register file configuration shown in Figure 4.2(d), which is used for our area and performance estimation in Chapter 6 and Chapter 7.

One subtle performance issue affecting the impact of lane aware scheduling for one of our benchmarks (Bitonic Sort) is related to the type of pathological even/odd thread identifier control dependence described in Chapter 3. For example, if threads in all even lanes see a branch as taken, while threads in all odd lanes see the same branch as not-taken, then it is impossible for dynamic warp formation to create larger warps. A simple solution we employ is to alternately swap the position of even and odd thread home lanes every other warp when threads are first created (an approach we call *thread swizzling*).

4.2 Hardware Implementation

Figure 4.3 shows a high level block diagram illustrating how dynamic warp formation and scheduling can be implemented in hardware. A detailed implementation for the Majority scheduler is described in Section 4.4. Referring to Figure 4.3, the two warp update registers, labeled (a), store information for different target PCs of an incoming, possibly diverged warp; the PC-warp LUT, labeled (b), provides a level of indirection to guide diverged threads to an existing or newly created warp in the warp pool, labeled (c). The warp pool is a staging area holding all the warps to be issued to the SIMD pipeline.

When a warp arrives at the last stage of the SIMD pipeline, its thread identifiers (TIDs) and next PC(s) are passed to the thread scheduler (Figure 4.3(a)). For conditional branches,

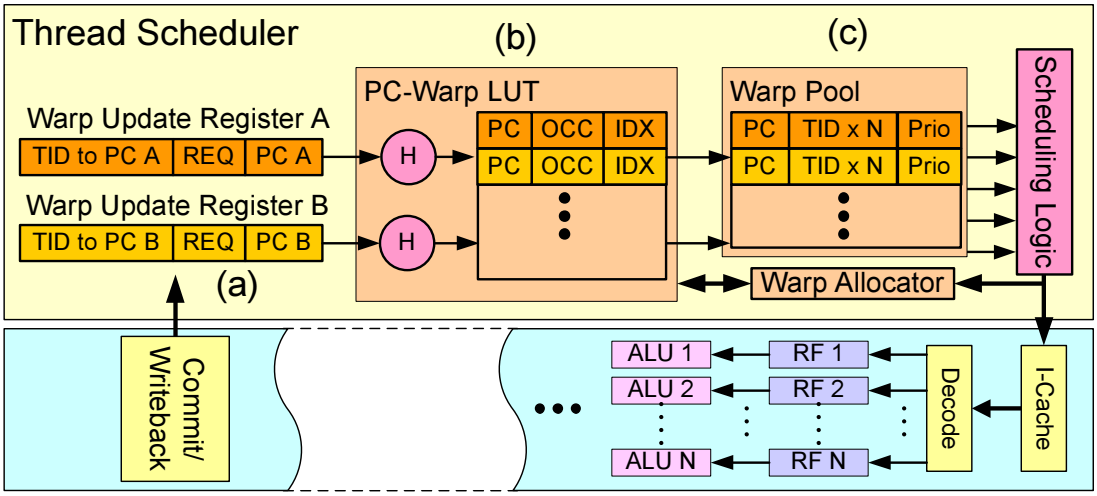


Figure 4.3: Implementation of dynamic warp formation and scheduling. In this figure, *H* represents a hash operation. *N* is the width of the SIMD pipeline. See text for detail description.

there are at most two different next PC values.[12] For each unique next PC sent to the scheduler from writeback, the scheduler looks for an existing entry in the PC-warp LUT already mapped to the PC and allocates a new entry if none exists[13] (Figure 4.3(b)).

The PC-warp LUT (Figure 4.3(b)) provides a level of indirection to reduce the complexity of locating warps in the warp pool (Figure 4.3(c)). It does this by using the IDX field to point to a warp being formed in the warp pool. This warp is updated with the thread identifiers of committing threads having this next PC value. Each entry in the warp pool contains the PC value of the warp, N TID entries for N lanes in an N-wide SIMD pipeline, and some policy-specific data (labeled "Prio") for scheduling logic. A design to handle the worst case where each thread diverges to a different execution path would require the warp pool to have enough entries for each thread in a shader core to have its own entry. However, we observe that for the benchmarks we simulated only a small portion of the warp pool is used, and we can shrink the warp pool significantly to reduce area overhead without causing a performance penalty (see Chapter 6).

To implement the lane aware scheduler mentioned in Section 4.1, each entry in the PC-warp

[12] Indirect branches that diverge to more than two PCs can be handled by stalling the pipeline and sending up to two PCs to the thread scheduler every cycle.

[13] In our detailed model we assume the PC-warp LUT is organized as a small dual-ported 4-way set associative structure.

LUT has an occupancy vector (OCC) tracking which lanes of the current warp are free. This is compared against the request vector (REQ) of the warp update register that indicates which lanes are required by the threads assigned to this warp. If a required lane is already occupied by a thread, a new warp will be allocated and the TIDs of the threads causing the conflict will be assigned into this new warp. The TIDs of the threads that do not cause any conflict will be assigned to the original warp. In this case, the PC-warp LUT IDX field is also updated to point to the new warp in the warp pool. The warp with the older PC still resides in the warp pool, but will no longer be updated. A more aggressive approach would be to continually try to merge threads into the earlier warp, but this is beyond the scope of this thesis.

Each cycle, a single warp in the warp pool may be issued to the SIMD pipeline according to one of the scheduling policies described in the next section. Once issued, the warp pool entry used by the warp is returned to the warp allocator.

4.2.1 Warp Pool

As mentioned above, the warp pool holds the all the warps in the scheduler. Every cycle, the incoming threads with the same PC value may be assigned to either an existing warp or a newly allocated warp in the warp pool. To handle the case of a branch that diverges into two groups of threads each with a distinct PC value, the warp pool must allow parallel access to four different warps in each cycle.[14] A naïve implementation of the warp pool with a single memory array would require four write ports, which significantly increases the area requirement of dynamic warp formation. However, with the observation that each incoming warp only contains a thread executing in each SIMD lane (assuming lane aware scheduling is used here), a more efficient implementation of the warp pool, shown in Figure 4.4(a), is possible. This implementation eliminates the need of four write ports by separating the warp pool into banks, with each bank storing the TIDs of threads executing in the same SIMD lane. Each bank requires only one write port, because every incoming warp, no matter how diverged, never contains more than one thread with the same home SIMD lane. The PC value and any scheduler specific data of each warp are stored in a separate bank from the TIDs with two write ports.

[14] Threads in the same group may be assigned to an existing warp or a new warp. Both warps are updated in the worst case.

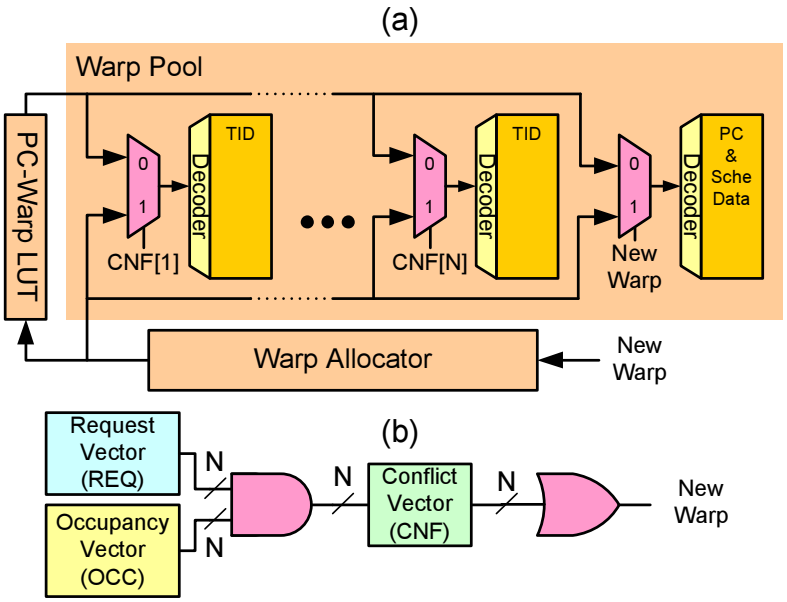


Figure 4.4: Implementation of the warp pool. (a) Implementation with lane aware scheduling. (b) Generation of conflict vector (CNF) and the new warp signal. N is the width of the SIMD pipeline.

In Figure 4.4(a), the write ports to each bank with TIDs is indexed by either the IDX from PC-warp LUT or a new IDX from warp allocator. The choice is determined by a bit from the conflict vector (CNF) which is the logical AND of the request vector (REQ) and the occupancy vector (OCC). The input address to the bank containing PC value and scheduler specific data is determined by a *new warp* signal which is set whenever CNF is not all zero. This *new warp* signal also requests the warp allocator for an index to a free warp. Figure 4.4(b) summarizes how CNF and *new warp* are generated.

A further refinement, that exploits the fact that a single warp may be executed over multiple cycles (four cycle in our baseline configuration—see Chapter 5) is to use a single memory array with one read port and one write port and perform the four warp updates over four cycles (one every cycle). However, due to time constrains we do not explore this further in this thesis.

4.3 Scheduling Policies

Even though dynamic warp formation has the potential to fully utilize the SIMD pipeline, this will only happen when the set of PC values currently being executed is small relative to the

number of scalar threads. If each scalar thread progresses at a substantially different rate, then all threads will eventually map to entirely different PCs. To avoid this, all threads should have a similar rate of progress. We have found that the warp scheduling policy, namely, the order in which warps are issued from the warp pool, has a critical effect on performance (as shown in Chapter 6). We explored the following policies:

Time Stamp (DTime): Warps are issued in the order they arrive at the scheduler. Note that when a warp misses the cache, it is taken out of the scheduler until its requested data arrives from memory and this may change the order that warps are issued.

Program Counter (DPC): In a program sequentially laid out in instruction memory, the program counter value itself may be a good indicator of a thread's progress. By giving higher issue priority to warps with smaller PCs, threads lagging behind are given the opportunity to catch up.

Majority (DMaj): As long as a majority of the threads are progressing at the same rate, the scheduling logic will have a large pool of threads from which to create a new warp every cycle. The majority policy attempts to encourage this behaviour by choosing the most common PC among all the existing warps and issuing all warps at this PC before choosing a new PC.

Minority (DMin): If a small minority of threads diverges away from the rest, the Majority policy tends to leave these threads behind. In the minority policy, warps with the least frequent PCs are given priority with the hope that, by doing so, these warps may eventually catch up and converge with other threads.

Post-Dominator Priority (DPdPri): Threads falling behind after a divergence need to catch up with other threads after the immediate post-dominator. If the issue priority is set lower for warps that have gone beyond more post-dominators, then the threads that have yet to go past the post-dominator tend to catch up.

4.4 A Majority Scheduling Policy Implementation

Majority scheduling logic, the best performing policy among the presented ones (as shown in Chapter 6), can be implemented in hardware with a max-heap and a lookup-table as shown in Figure 4.5. This hardware implementation may also be applied to other scheduling policies

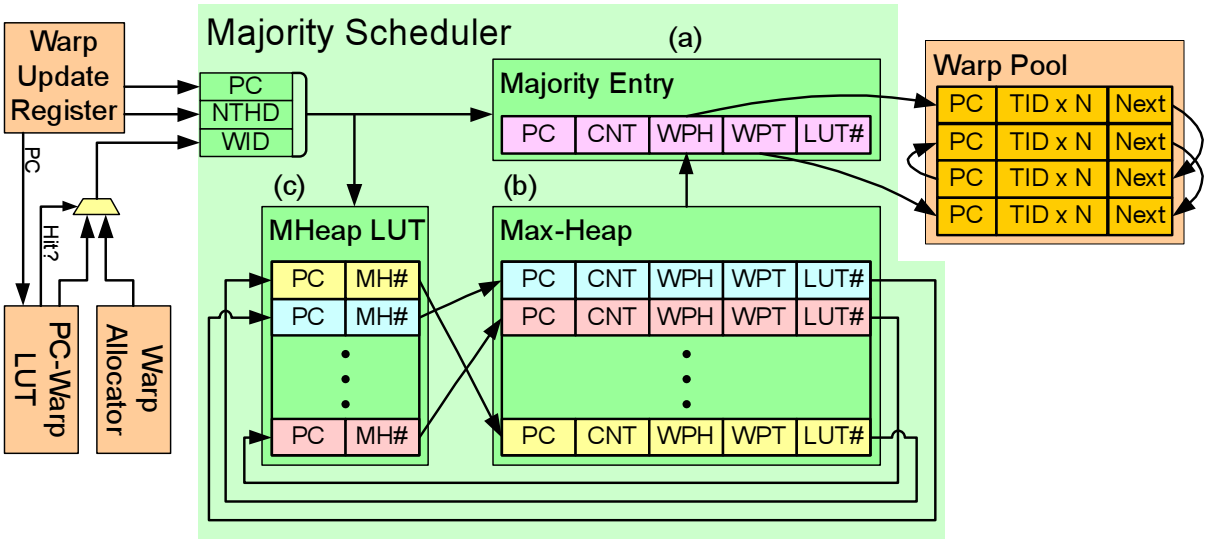


Figure 4.5: Implementation of majority scheduling policy with max-heap. See text for detail.

that require priority management, such as DPC and DPdPri, with some modifications, but this thesis will focus on the implementation for Majority scheduling policy. The area of this hardware implementation is included as part of the area overhead estimation for dynamic warp formation and scheduling in Chapter 7. In Figure 4.5, the Majority Entry, labeled (a), keeps track of the current majority PC value and a group of warps with this PC value[15]; the Max-Heap, labeled (b), is a full binary tree of PC values (and its group of warps) sorted by number of threads with this given PC value using the algorithm described in Chapter 6 of Cormen et al. [15], and the MHeap LUT, labeled (c), provides a level of indirection for incoming warps to update their corresponding entries in the Max-Heap (i.e., the function of the PC-warp LUT in Figure 4.3). While we find a simple max-heap performing one swap per cycle per warp to be sufficient for our usage, using a pipelined max-heap hardware implementation [4, 29] may further reduce the bandwidth requirement of the max-heap.

Each entry in the Max-Heap represents a group of warps with the same PC value, and this group of warps has CNT threads. This group of warps forms a linked list in the warp pool with a "Next" entry referring to the next warp in the list. WPH keeps the head of this list and WPT keeps the tail (so that the list can grow as warps arrive). LUT# is a back pointer to the entry

[15]This entry is separated because we finish executing all warps at a given PC value before selecting a new PC value.

in MHeap LUT, and is used to update the LUT for correctness after a swap (see detail below).

4.4.1 Warp Insertion

When a warp arrives at the thread scheduler, it is placed in the warp update register. The warp's next PC value (PC); the number of threads, or thread count, of this warp (NTHD); and the ID of the warp (WID) in the warp pool (acquired via PC-warp LUT or warp allocator as in Section 4.2), where these threads will be placed, are sent to the Majority scheduler. Inside the Majority scheduler, the PC value is checked against the one stored in the Majority Entry, and if they match, the Majority Entry will be updated directly. Otherwise, the MHeap LUT will provide the index (MH#) into the Max-Heap entry with this PC value, or return an index of an inserted new entry in the Max-Heap if an existing entry with this PC is not found in the MHeap LUT. This Max-Heap entry, denoted by MH#, is then updated: CNT is incremented by NTHD, and if WID is different from WPT, WPT will be updated with WID, and WID will be assigned to the "Next" entry in the warp referred by the original WPT (essentially inserting the warp referred by WID into the linked list of this entry). If this is an newly inserted entry, the value of WID will be assigned to both WPH and WPT.

After the entry in the Max-Heap is updated or inserted, this entry is compared with its parent entry and swapped up the binary tree until it encounters a parent with a larger thread count (CNT) or it becomes the root of the binary tree. Whenever a swap occurs, the corresponding entry in MHeap LUT (denoted by LUT# in the swapping Max-Heap entries) is updated. During this operation, no entries in the Max-Heap can be updated.

4.4.2 Warp Issue

In every scheduler cycle[16], a warp from the warp pool will be issued to the SIMD pipeline. If the Majority Entry has any warp remaining (CNT>0), the warp denoted by WPH will be issued to the pipeline. WPH will then be updated with the value in the "Next" entry of the issued warp, and CNT will be decremented by the number of threads in the issued warp. If the Majority Entry runs out of warps (CNT=0), the root entry of the Max-Heap will be popped and will become the Majority Entry. If the Max-Heap is in the process of rebalancing itself after a warp insertion, no warp will be issued until the Max-Heap is eventually balanced. The Max-Heap

then balances itself according to the algorithm described by Cormen et al. [15]. Similar to the case of warp insertion, the MHeap LUT will be updated in parallel to any swap operation, and during this operation, no entries in the Max-Heap can be updated.

4.4.3 Complexity

As max-heap is a tree structure, every warp insertion only requires as many as $log(N)$ swaps for a N-entry max-heap to rebalance, and experimentally we found that usually fewer swaps are required. We find in Chapter 6 that a design with both the Max-Heap and the MHeap LUT each having 2 read ports and 2 write ports (to perform 2 swaps in parallel to handle 2 warp insertion from each part of the incoming warp every scheduler cycle[16]) is sufficient for keeping the Max-Heap balance in the common case. This is assuming that both structures are clocked as fast as the SIMD pipeline, so that a total of 8 reads and 8 writes can be performed in every scheduler cycle.[16] If the number of PCs in flight exceeds the max-heap capacity, a mechanism such as spilling the entries from the max-heap could be implemented. However, we observed in Chapter 7 that with a 64 entry max-heap, the need for this never arises for our benchmarks. We leave the exploration of the performance impact of spilling the max-heap with smaller max-heap capacity as future work.

4.5 Summary

This chapter described our proposed branch handling mechanism—dynamic warp formation and scheduling. It discussed various implementation details including a ways to avoid register file bank conflict (lane aware scheduling) and a hardware implementation of the *Majority* scheduling policy. Next chapter describes the simulation methodology we use in this thesis to evaluate this mechanism.

[16] A scheduler cycle is equivalent to several pipeline cycles, because each warp can take several cycles (e.g. 4 for a warp size of 32) to execute in the pipeline (see Section 2.5 of Chapter 2), so the scheduler may operate at a slower frequency.

Chapter 5

Methodology

While simulators for contemporary GPU architectures exist [19, 66], none of them model the general-purpose GPU architecture described in this thesis. Therefore, we developed a novel simulator, *GPGPU-Sim*, to model various aspects of the massively parallel architecture used in modern GPUs with highly programmable pipelines.

The benchmark applications used for this study were selected from SPEC CPU2006 [69], SPLASH2 [79], and CUDA [57]. Each benchmark was manually modified to extract and annotate the compute kernels, which is a time-consuming task limiting the number of benchmarks we could consider. The programing model we assume is similar to that of CUDA [58], which uses a system call to a special GPU device driver in the operating system to launch parallel sections on the GPU. In our simulator, this system call mechanism is emulated by a spawn instruction, which signals the out-of-order core to launch a predetermined number of threads for parallel execution of a compute kernel on the GPU simulator. If the number of threads to be executed exceeds the capacity of the hardware configuration, the software layer is responsible for organizing threads into *blocks*. Threads within a block are assigned to a single shader core, and all of them have to finish before the shader core can begin with a new block.

The rest of this chapter describes our new simulator, *GPGPU-Sim*, in detail and gives an overview of the benchmarks used in this thesis.

5.1 Software Design of *GPGPU-Sim*—A Cycle Accurate GPGPU Simulator

This section describes the software design of GPGPU-Sim, the cycle accurate GPU simulator used in this thesis.

GPGPU-Sim consists of two top level components: functional simulation and performance (timing) simulation. Functional simulation provides the behaviour that the program expects from the architecture, whereas the performance simulation provides an estimate of how fast the program would execute if it were to be run on the modeled microarchitecture. The two parts of simulation are usually decoupled in a simulator to allow developers to introduce approximations to the performance model for fast prototyping of novel ideas. For example, our DRAM performance model does not model DRAM refreshes. While DRAM refresh is an important feature to maintain functional correctness of a DRAM module, omitting it has minimal impact on the accuracy of a DRAM performance model due to the rare occurrence of DRAM refresh (once every several million cycle). For this reason, like most modern architecture simulators in widespread uses (such as SimpleScalar [10], SMTSIM [76], PTLsim [81]... etc.), GPGPU-Sim employs the separation of functional simulation from performance simulation in its software design.

GPGPU-Sim was constructed "from the ground up" starting from the instruction set simulator (sim-safe) of SimpleScalar version 3.0d [10]. We developed our cycle-accurate GPU performance simulator, modeling the microarchitecture described in Chapter 2, around the SimpleScalar PISA instruction set architecture (a RISC instruction set similar to MIPS) and then interfaced it with sim-outorder, which only provides timing for code run on the CPU in Figure 2.2.

Figure 5.1 provides an overview of GPGPU-Sim, with each source code file classified into one of three main modules: Shader Core, Interconnection Network, and DRAM Model. Files that do not fit into any of these three modules either contain code that provides miscellaneous tools to all the modules (`gpu-misc` and `delayqueue`), or is glue code that connects all modules together (`gpu-sim`). The following sections give an overview to the three main modules.

5.1.1 Shader Core

Each shader core contains a simple pipeline, much like the classic MIPS 5-stage in-order pipeline described in [24]. This simple pipeline is extended to have fine-grained multithreading, SIMD, miss status hold registers (MSHRs), the branch handling mechanisms that was discussed in this thesis so far (Chapter 3 and Chapter 4), and various other microarchitecture details.

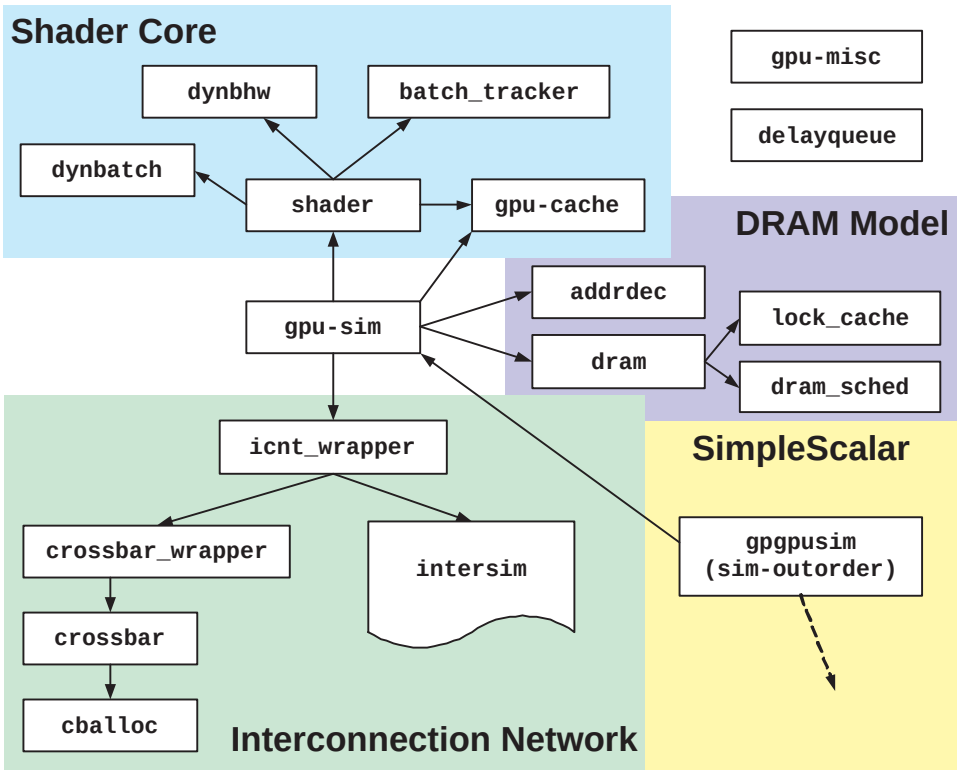


Figure 5.1: Overview of GPGPU-Sim's software design. A → B means "A is uses B".

Figure 5.2(a) lists the pipeline stages in a shader core, and briefly illustrates the functionality of each stage. Note that in the simulator, these stages are simulated in reverse order of the pipeline flow to eliminate the need for two copies of each pipeline register. This is a well-known technique, also used in SimpleScalar's out-of-order simulator (sim-outorder).

Each shader core connects to the simulated memory subsystem (the memory system and the interconnection network) via an abstract software interface that treats the memory subsystem as an out-of-order queue with requests coming back in any order:

`fq_has_buffer` Query the memory subsystem for buffer space to hold a list of memory requests, each with its requested memory address specified.

`fq_push` Push memory request (memory address, number of bytes to access, and whether the access is a read or a write) into the memory subsystem.

`fq_pop` Pop serviced memory request from the memory subsystem and push it into the return queue.

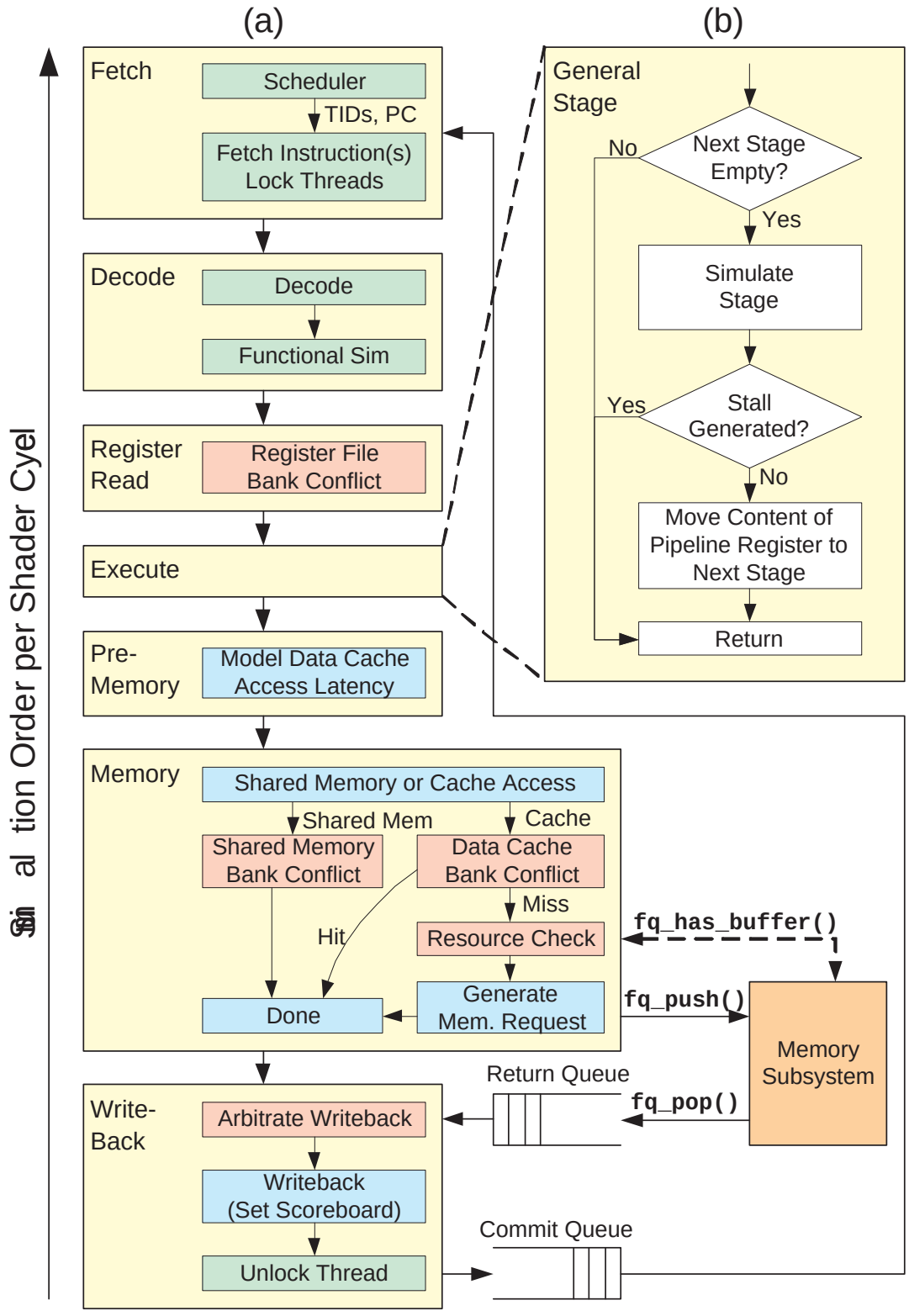


Figure 5.2: Software design of pipeline stages in the shader core model.

This abstract interface decouples the internal design of the shader core module from the rest of the simulator.

The behaviour of every stage in a shader core is modeled by a function call in the simulator with the general structure shown in Figure 5.2(b), and each stage has a set of pipeline registers acting as the input into that stage. Each stage starts with a check to see if the pipeline register between the current stage and the next stage is empty. If this pipeline register is not empty, the next stage must be stalled, therefore the current stage should stall as well. Since each cycle the activity of the pipeline stages are evaluated in reverse order, this implies that a hazard at the end of the pipeline can stall the entire pipeline in a single cycle. If the pipeline registers are empty, the simulator goes on simulating the behaviour of the stage. Whenever a stall is generated in the simulator, the function call modeling the stage will just return, leaving the content of its pipeline register unchanged. When the stage proceeds without generating a stall, the content of the pipeline register of the current stage is copied to the next stage's pipeline register (to imitate the control signal transferring from one stage to the next in hardware). The pipeline register of the current stage is then cleared, indicating to the earlier stage that no stall has been generated.

The following is a description of each stage:

Fetch

The fetch stage starts by selecting the warp that will be issued to the pipeline. The selection is performed by one of the schedulers implemented in the simulator as specified in the simulator's configuration, which can be specified either on the command line or via a configuration file. Each scheduler implements one of the various branch handling mechanism discussed in Chapter 3 and Chapter 4. The MIMD architecture is modeled with a scheduler which can freely schedule threads into the lanes of the pipeline regardless of their PC values.

After selecting which threads to issue, the scheduler passes the thread IDs and their PC values to the fetch block, which will fetch the instruction from memory[17] and put them into the pipeline register of the decode stage. The fetch block also locks the issued threads by setting a flag, indicating to the scheduler that this thread will not be available for scheduling until it

[17] We currently assume that all accesses to instruction cache hit because of the small kernel code size.

is unlocked in later stages (this is done at the write-back stage). The warp size distribution (a key statistic reported in Chapter 6) is collected in the fetch stage.

Decode

The decode stage decodes the instructions at the pipeline register between fetch and decode, determining the instruction type (Arithmetic/Branching/Memory) and the registers that will be used. It also checks for any violation of a dependency hazard (checking the scoreboard for any use of the marked registers). If a dependency hazard is detected, it nullifies the instruction (i.e., clears its input pipeline register). If no hazard is detected, a scoreboard entry will be set to indicate that the output registers of these instructions are in use.

If the simulator is configured to perform "parallel functional simulation mode", the decode stage will also functionally simulate the instructions at its input pipeline registers using SimpleScalar's instruction set simulation kernel. Parallel functional simulation mode is described in Section 5.1.4.

Register Read

The register read stage is only used when modeling the effects of register file bank conflicts. The register file bank to be accessed by each thread is determined by the thread's thread ID. For each warp, the register read stage counts the number of threads accessing each bank, and stalls when the count for any of the bank exceeds one. The stage then stalls for a number of cycles equal to the maximum count of threads accessing any given bank. This models the serialization of warp execution to resolve bank conflicts. This stage is only activated when using the dynamic warp formation scheduler.

Execution

The execution stage is just an empty stage as functional simulation is done at the decode stage. It may be retrofitted in the future to model the latency of transcendental functions (sine, cosine, square root, reciprocal... etc.) and interpolator featured in modern GPU architectures [39].

Pre-Memory

The pre-memory stage is an optional stage modeling the cache access latency, which can possibly be longer than one shader cycle with a large cache. It is a series of pipeline registers in a queue buffering up the instructions going from the execution stage to the memory stage. Each instruction has to traverse through the length of the queue (equivalent to the number of cycles it takes to access the cache minus one) to reach the Memory stage.

Memory

The memory stage handles accesses from the incoming threads to the data cache or to the shared memory (a small, fast scratch pad memory featured in CUDA [58]).

To simulate this software controlled fast scratch pad memory, it first classifies each data access as an access to the shared memory or to the data cache by the memory address. If the data access falls into the region belonging to the shared memory[18], it is classified as an access to shared memory.

For instructions accessing the shared memory, the memory stage checks for bank conflicts according to the addresses of all the accesses[19], and stalls for a number of cycles equal to the maximum number of threads accessing the same bank (similar to the register read stage).

For instructions accessing the data cache, the memory stage also checks for bank conflicts at the data cache[19] and stalls accordingly. Once the stalls due to bank conflicts are over, the data cache is accessed. If all accesses hit, the threads will be sent to the write-back stage. If some of the accesses miss, the memory stage will arbitrate for the resources (for example, miss status hold registers [34] (MSHRs) and input buffers at the interconnect to store the requests) required to generate all the required memory requests. If the allocation failed, the memory stage stalls and tries again next cycle. Note that some of the misses can be merged with other misses and serviced with the in-flight (already sent) memory requests, and in that case, only MSHRs are allocated for these misses.

Once the resources are allocated, memory requests will be generated and sent to the inter-

[18] This is specified by the user and is unique to each benchmark. See Section 5.1.4 for details.

[19] The banks in shared memory and the data cache are determined by effective memory address rather than by thread ID (as done for the register file).

connection network and the memory system modules via an abstract interface. In this case, the contents of the input pipeline registers will be stored in the MSHR, and will not be sent to the write-back stage.

Write-Back

The write-back stage has two inputs, one directly from the memory stage, and another from the return queue, which holds threads that missed the data cache and were serviced by a memory access. The first step in the Write-Back stage is to arbitrate between the two inputs. In the current version of GPGPU-Sim, priority is always given to the threads from the return queue, which may generate sub-optimal results, as the pipeline can be stalled very frequently in yielding to the return queue.

The threads that won the arbitration proceed to the rest of write-back stage. Scoreboard entries corresponding to the output registers of the threads will be cleared, indicating that the output values have already been written into the register file. Any subsequent instructions from the same thread accessing the same registers will not cause a dependency hazard. After that, the threads are unlocked and their IDs are sent to the commit queue, indicating to the scheduler that these threads are ready to be scheduled again.

5.1.2 Interconnection Network

The interconnection network module is responsible for relaying messages between the shader cores and the memory controllers (containing the DRAM model). It does not have any knowledge about the content of the messages, other than the size of the message in bytes. This module also models the timing and congestion for each message in the interconnect. It provides the following abstract software interface to other modules (such as shader core and dram timing model):

`icnt_has_buffer`	Query the interconnect for buffer space to hold all the given messages, each with its size and output specified, at a given input.
`icnt_push`	Push a message (whose content is passed as `void*`) into the interconnection network from a specific source to a specific destination.
`icnt_pop`	Pop a message from the interconnection network at a given output.
`icnt_transfer`	Advance one cycle in the interconnection network.
`icnt_busy`	Checks if the interconnect still has undelivered messages.
`icnt_drain`	Transfer all the messages to their destination buffers.

Currently there are two implementations of the interconnection network interface. One of them is Intersim, a modified version of a general interconnection simulator created by Dally and Towles associated with their textbook [17]. We have modified it to allow other modules to inject and extract packets to the modeled interconnection, and to enable the packets to carry actual payload (so that it is actually relaying messages from one module to another). Another, used in this thesis, is a simplified implementation of the crossbar described in the same work [17]. The general structure of this interconnection model features two crossbars, one relaying the messages from the shader cores to the memory controllers and the other doing the opposite. This design simplified the allocator design, because it enables the use of two simple allocators handling one-way traffic instead of one complex allocator handling bidirectional traffic. Sun's Niagara [70] also employs two crossbars to enable bidirectional traffic between processor cores and the memory subsystem (Chapter 6 of [70]).

Crossbar Allocator

The crossbar allocator orchestrates the cross-points to satisfy multiple requests from different inputs asking for cross-points to some desired outputs. When there are conflicts among these requests, only one can be fulfilled, and the others will be delayed to the next cycle. The requests from all N inputs to M outputs can be summarized into a $N \times M$ request matrix R (with a request from input i to output j indicated by a 1 at row i column j), and the allocator will

generate a $N \times M$ grant matrix G indicating which of the request has been granted:

$$R = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ or } \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \text{ or } \ldots$$

The above example shows a request matrix with 4 inputs and 4 outputs. Notice how with the same request matrix the allocator can often generate many different grant matrices (two of them shown above), with many of them being suboptimal. This shows that the crossbar allocator is crucial to the crossbar's bandwidth utilization.

In this thesis, we have chosen to use the Parallel Iterative Matching (PIM) allocator, a simple separable allocator described in [17, Chapter 19]. First, the allocator randomly selects one of the requested outputs for each input buffer (note: Not input, but input buffer) and neglects all other requests from the input buffer. Each unmatched output then randomly chooses one of the remaining requests asking for that output. Using the example above, the intermediate matrix Y randomly chosen in the first step might be as follows:

$$Y = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow G = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ or } \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The right-most output in Y has more than one remaining request, and no matter which of the requests the output choses, the overall bandwidth utilization is below optimal. This is illustrated by the only two grant matrices G that can result from Y. Both grant matrices have no requests routed to the second output and this fails to fully utilize the crossbar bandwidth.

Input Speedup

This non-optimal nature of the PIM allocator can be alleviated using multiple passes, allowing the idle outputs to choose from the neglected requests in previous passes, at a cost of longer

allocation latency. Another way to improve utilization is to use input speedup [17, Chapter 17], which splits the input buffer into multiple parts each containing message to be delivered to a subset of the outputs. In this thesis, we use a crossbar configuration with a input speedup of two. Each input has two input buffers, one connects exclusively to the odd outputs, and the other to the even outputs. The doubled input buffers reduces the chance of a request being neglected during the first step of PIM allocation, increasing the probability that an output will have a bidding request per cycle. Dally and Towles [17, Chapter 17] has shown that with a random traffic pattern, a input speedup of two can increases the bandwidth utilization of a PIM allocator from 64% to 87% for a 16 by 16 crossbar.

Input Buffers and Output Buffers

To minimize the effect of contention within the interconnect to the host system (either shader core or memory controller), incoming messages are first stored in the input buffer when they are pushed into the interconnection network (using the `icnt_push()` abstract interface). Inside the input buffer, messages are broken down into flits. A flit is a small data packet that can be transfered through the crossbar fabric in a single cycle. This allows the allocator to work at a fine grained manner to schedule the transfer of different messages to achieve higher efficiency. At the output buffer, these flits are assembled into their original messages, and wait to be popped by the destination system.

In our implementation of the crossbar, we model the finite sizes of input buffers and output buffers using credits [17, Chapter 13]. Each buffer starts with an amount of credits equivalent to its maximum capacity in number of flits. When an incoming message is pushed into an input buffer, a number of credits equal to the number of flits that the message occupies are deducted. As flits are transfer out of the input buffer, credits are returned to the buffer. The same applies to the output buffers, where credits are deducted as flits arrive from the crossbar and are returned when messages are popped. For both cases, when credits run out, it means the buffer is full. Our current interconnection interface forces its user to check for availability of input buffer space with the `icnt_has_buffer()` query, which checks the amount of credits available for the queried input buffer, before pushing messages into the interconnection network. On the other hand, a buffer-full at an output is handled internally by disabling the transfer

of flits to that particular output, even if the allocator granted the request, until output buffer space is available again. This behaviour allows congestion at the destination to propagate back to the input. This propagation property is vital in modeling an interconnection system.

Bandwidth Modeling

The theoretical peak bandwidth of the modeled interconnection network is defined by the clock rate of the crossbar, as well as the flit size. The clock rate of the crossbar is essentially the calling rate of `icnt_transfer()` in the simulator, which is currently identical to the shader core clock rate. Future work will include implementing a mechanism to model relative clock domains between different modules.

5.1.3 DRAM Access Model

Due to the high computational throughput in a GPU architecture, memory bandwidth and latency often become a performance bottleneck. Capturing this behavior is vital to the precision of a GPU simulator. ATTILA, one related cycle accurate GPU simulator, models a simplified GDDR memory accounting for read-write transition penalty and bandwidth, while each memory module is treated as a single bank [19]. This fails to capture the memory latency hiding mechanism via multiple banks that plays a significant role in bandwidth utilization [64]. Therefore, a detailed DRAM access model was created to capture this behaviour. While an existing simulator, DRAMsim, provides an access model of slightly better accuracy (modeling DRAM refreshes as well) than our access model [78], we did not know of its existence until our access model was created.

Figure 5.3 presents an overview of the DRAM access model. It is divided into the access timing model, the request scheduler, and the address decoder. An incoming memory request is first processed by the address decoder, which breaks down the request address into *chip ID*, *row*, *column* and *bank*. This information, together with the request itself, is passed through the interconnection network to the appropriate memory controller (determined by the chip ID). Inside the memory controller, the request scheduler determines when the request will be sent to the access timing model. The access timing model then fulfills each incoming request by issuing the appropriate commands in accordance to the DRAM timing specification.

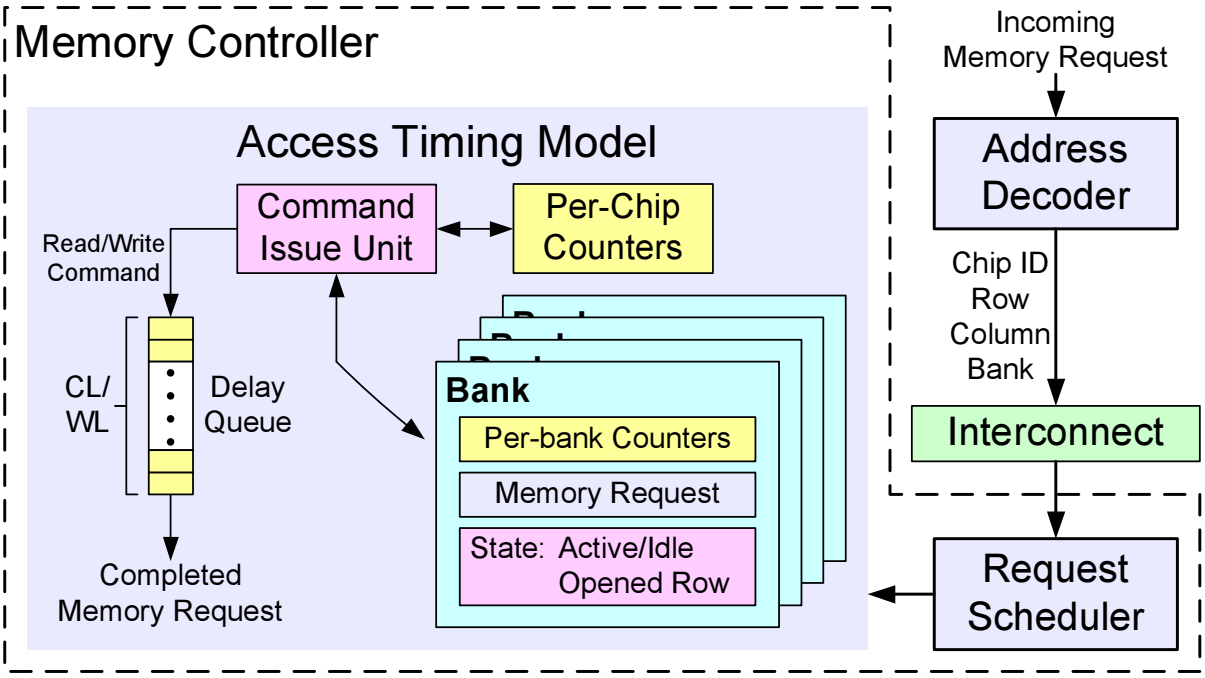


Figure 5.3: Dram access model.

Address Decoder

The address decoder translates a given memory address into multiple parts: chip ID, row, column and bank. Chip ID determines which of the DRAM chips the memory address is mapped to, while row, column and bank refers to structures inside each DRAM chip.

We have observed that address decoding has significant impact to the performance of a system. For example, an application would be bottlenecked by memory bandwidth, no matter how many DRAM chips are available in the system, if all of its memory request are mapped to a single DRAM chip. The importance of address mapping to cache miss rate in a direct-mapped cache has been evaluated in [82].

In effect, the address decoding module is designed to be as flexible as possible for future research. The address mapping is defined by the user as a series of bit masks, one for each of chip ID, row, column and bank. Each mask define which part of the address bits should be mapped to which entry in the decoded address. For example, when given with a set of bit masks:

chip ID	00000000 00000000 00011010 00000000
Row	00001111 11111111 00000000 00000000
Column	00000000 00000000 11100000 11111111
Bank	00000000 00000000 00000101 00000000

The address decoder will create a mapping as follows:

31—24	23—16	15—8	7—0
$XXXXR_{11}R_{10}R_9R_8$	$R_7R_6R_5R_4R_3R_2R_1R_0$	$C_{10}C_9C_8K_2K_1B_1K_0B_0$	$C_7C_6C_5C_4C_3C_2C_1C_0$

where R_X refers to the X^{th} bit in the row address, C_X refers to the X^{th} bit in the column address, B_X refers to the X^{th} bit in the bank address, and K_X refers to the X^{th} bit in the chip ID. The flexibility of this interface allow the user to specify address mappings that are not restricted to contiguous bits.

DRAM Request Scheduler

The request scheduler controls the order at which the memory requests are dispatched to individual DRAM banks. It detects free DRAM banks (those not servicing any memory request) and tries to dispatch a new request to the free bank. Currently, there are two different scheduling policies implemented in the request schedulers:

FIFO (First-Come-First-Serve) All the requests are dispatched in the order they arrive at the memory controller. When the oldest request is pending on a busy bank, all other requests are not dispatch, even if the banks they bid for are free.

FR-FCFS (First-Ready First-Come-First-Serve) Reorder the requests to prioritize requests that access an already opened row. This allows multiple requests with row locality (accessing the same row) to be scheduled together and amortize the overhead of a single row activation among these requests. To take advantage of this, the access timing model is configured to open the row after serving a request. This is an implementation of the *open row* scheduling policy proposed by Rixner et al. [64].

While we have chosen to use the FR-FCFS scheduling policy in this thesis, we acknowledge that there exist more advanced policies that take fairness and overall performance into account, such

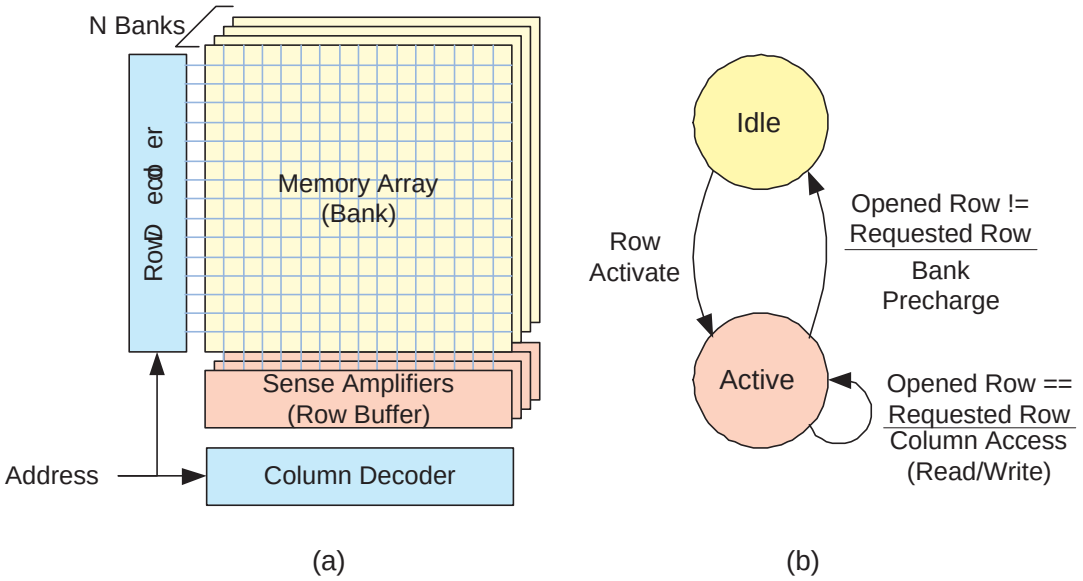


Figure 5.4: DRAM organization overview and simplified DRAM bank state diagram.

as the ones proposed by Nesbit et al. [55] and Mutlu and Moscibroda [51]. We may implement these or other more sophisticated policies in the request scheduler in the future.

Access Timing Model

Figure 5.4(a) shows the organization in a modern DRAM that the access timing model tries to model. It consists of multiple memory arrays, called banks, each with its own set of sense amplifiers, which double as row buffers storing the data for one of the rows in the memory array. All banks share a single row decoder and column decoder. To access data in one of the banks, the memory controller first issues a Row Activation command to retrieve the row containing the requested data from the memory array to the row buffer. It then issues a Column Access command to access the requested data in the row buffer. To access data in another row, the memory controller needs to issue a Bank Precharge command to write the content of the row buffer back to the memory array, and then issue a Row Activation command for the new row.

Figure 5.4(b) captures the actions required to satisfy a single memory request in a DRAM bank into a state diagram with two states: Active and Idle. Essentially, whenever the opened row in a bank is different from the one required by the request it is servicing, the command issue unit precharges the bank, activates the requested row, and then services the request with

Counter Name	Scope	Set by Command	Constrained Command
RRD	Chip	Activate	Activate
CCD	Chip	Read/Write	Column Access (Read/Write)
WTR	Chip	Column Access (Write)	Column Access (Read)
RTW	Chip	Column Access (Read)	Column Access (Write)
RCD	Bank	Activate	Column Access (Read/Write)
RAS	Bank	Activate	Precharge
RP	Bank	Precharge	Activate
RC	Bank	Activate	Activate

Table 5.1: Relationships between constrain counters and DRAM commands.

column accesses. After the request is completed, the row is left open for that bank to taking advantage of row locality. Subsequent requests to the same row can perform column accesses immediately without reopening the row.

Figure 5.3 shows the details of the access timing model. The model keeps track of the current request that each bank is trying to service, and the hardware state of each bank: active or idle, and which row is currently opened in the bank if it is active. This information is used by the command issue unit in junction with the behaviour stated in Figure 5.4(b) to decide the command to issue for each bank. The timing constraints specified by the DRAM specification are modeled using two sets of constraint counters: One set representing timing constraints that are local to a bank (per-bank counters), and another set representing timing constraints that apply to all banks within a single DRAM chip (per-chip counters).

Every cycle, the command issue unit traverse through all the banks in a round-robin manner to look for a command that is ready to be issued (i.e. all of its constraint counters are zero). There is no priority to the command type, and the first command that is found to be ready will be issued. After a command is issued, the per-chip counters and the per-bank counters of the banks serviced by the issued command are set to reflect the timing constraints created by that command. All of the counters are decremented every cycle until zero, and stay at zero until they are set again by another command. Table 5.1 shows the relationship between the constraint counters and the commands.

When a column access command is issued, the data associated with the command will not have access to the bus until the CAS latency (CL) has elapsed. Within this period, another

read-write command to the same row or to another activated bank may be issued, and these commands can be pipelined for the data bus access. This pipelining behavior is modeled using a delay queue, which is a queue with fixed length shifting its data towards its output every cycle. In this way, while a single command takes a number of cycles to go through a delay queue, a group of commands issued consecutively appears in the same order at the output of the queue. The length of the delay queue can be changed dynamically from CAS latency (CL) to write latency (WL) when the bus switches from read to write and vice versa. The burst length of a column access command is modeled by pushing two commands into this delay queue when the command is issued. All timing constrains and latencies in this timing model can be configured by the user via command line options.

5.1.4 Interfacing with sim-outorder

As described in Section 2.2 of Chapter 2, the SimpleScalar out-of-order core (modeling the CPU in our simulator) waits for the GPU when it reaches a parallel section. After GPU simulation of the compute kernel is completed, program control is returned to the out-of-order core. This repeats until the benchmark finishes.

The above behaviour is modeled in GPGPU-Sim by first providing the simulator with the list of PC values of where the spawn call to a parallel section occurs in the benchmark. Inside sim-outorder, the dispatch stage (where the functional simulation is done in sim-outorder) is modified to check the target PC against this list of PC values whenever a function call is dispatched. If the target PC of this function call matches with one of the PC values on this list, a parallel section is detected and it is launched to the GPU performance simulator. To prevent erroneously launching a parallel section during speculative execution from a branch misprediction, a parallel section can only be launched when a spawn instruction commits. Since the number of cycles for an instruction to go through the superscalar pipeline is small relative to the length of time it takes a parallel section to complete, in the simulator we launch the parallel section in dispatch stage when the functional simulation indicates that spawn instruction is on the correct path.

The steps involved in launching a parallel section in the GPU can be different depending on the mode of simulation to be done. Currently, we have implemented two modes of simulation

in the GPU performance simulator: trace mode, and parallel functional simulation mode.

Trace Mode Simulation

In trace mode simulation, the parallel section to be launched in the GPU is first functionally simulated serially as if each thread in the parallel section is just an iteration in a loop with its loop counter as the thread ID. Instruction traces (the PC value, instruction type, and if applicable, address of memory accessed for each dynamic instruction executed) for each thread are collected during this functional simulation. These traces are then used for driving the behaviour of the GPU performance simulator.

This simulation mode guarantees the functional correctness of the simulation, so that some impreciseness in the GPU performance simulator will not have a drastic effect on the simulation results. Also, the traces only need to be collected once and can then be used for multiple performance simulations later, potentially saving a significant amount of simulation time.

However, due to the shear number of threads in a parallel section, the instruction traces size becomes substantial (e.g. 16GB for HMMer) and cannot be completely loaded into memory during performance simulation. While this problem can be solved by streaming the instruction traces from disk on demand and allocating large buffer to minimize the number of disk access required, we found the performance simulation seriously bottlenecked on the network file system when we execute the simulation on a cluster using this mode (which was implemented). Moreover, simulating each thread serially fails to capture some important parallel aspects of the compute model. For example, it is impossible to simulate applications that make extensive use of software controlled "shared memory" with a simple serial functional simulation. One of the common usage cases for the shared memory involves having a block of threads cooperatively loading data into the on-chip shared memory. This block of threads then waits for each other at a synchronization barrier before operating on the data in the shared memory. These two problems motivated the development of parallel functional simulation mode[20].

[20] Other alternatives include using trace compression such as the one proposed by Liu and Asanovic [41], and a functional simulator that switch threads at synchronization barrier.

Parallel Functional Simulation Mode

Parallel functional simulation mode was implemented in the GPU simulator to address the problems encountered with trace mode simulation. Instead of collecting instruction traces before the performance simulation as in trace mode simulation, threads in a parallel section are functionally simulated at the decode stage of the GPU performance simulator. In this way, the threads are scheduled and functionally simulated as if they are executed on the true parallel hardware, so that features such as barrier synchronization are exploited by our benchmarks.

On the other hand, the GPU performance simulator now has to handle the functional correctness of the simulation as well. For example, consider a mechanism where instructions are nullified and reissued later at a cache miss. With trace mode simulation, the instruction reissue can be implemented as rolling back on the instruction trace. With parallel functional simulation mode, the performance simulator has to ensure that the same instruction is not executed twice for functional correctness.

When launching a parallel section with parallel functional simulation mode, contents of the out-of-order core's architecture register file is replicated to the register file of every thread. A different offset is added to the stack pointer of each thread so ensure that register spill from a thread does not corrupt the stack of another thread. Also, because the PISA instruction set uses part of the out-of-order core's stack to pass parameters into function call, this part of stack is copied to the private stack of each thread so that they can be accessed by each thread. Many of these issues are more related to the properties of PISA itself, but they are nevertheless described here to illustrate the sort of functional correctness issues that need to be handled with parallel functional simulation mode.

Shared Memory

Shared memory is a scratchpad memory local to a block of threads in a multiprocessor in the CUDA programming model. While this feature is not used by the benchmarks evaluated in this thesis, a contribution of this thesis is to implement it to allow CUDA applications written with this feature to be easily ported over to GPGPU-Sim in the future. In the GPU performance simulator, the shared memory is accessible via accessing to a special address range, which is the

address range of a statically allocated array in the benchmark. This address range is unique for each benchmark and needs to be specified explicitly by the user in the code info file.

5.2 Baseline Configuration

Table 5.2 shows the baseline configuration we simulated. The configuration is chosen to approximate the Geforce 8800GTX [39] as much as possible for a valid area overhead estimation present in Chapter 7. Because GPGPU-Sim does not support a non-power-of-two number of memory channels, we have changed the shader core to DRAM clock ratio, lowered the width per channel, and increased the number of memory controllers, to ensure that the amount of bandwidth each shader core receives per cycle is equivalent to that of Geforce 8800GTX (4Bytes/Shader core/cycle). We have used the GDDR3 timing parameters provided by Qimonda for their 512-Mbit GDDR3 Graphics RAM clocked at 650MHz [62]. While the Geforce 8800GTX does not have a L1-data cache, we have long latency L1-data cache to approximate a memory-side L2-cache. The sizes for the dynamic warp formation scheduler's internal structures listed in Table 5.2 are optimized for the best possible performance area ratio.

5.3 Benchmarks

Table 5.3 gives a brief description of the benchmarks simulated for evaluating dynamic warp formation and scheduling in this thesis. The benchmarks are compiled with the SimpleScalar PISA GCC version 2.7.2.3 cross compiler. The compiled binaries are then analyzed by a set of scripts to extract the list of PC values of all the spawn calls to parallel sections, as well as the starting PC value of the shader program of each parallel section. This information is stored into a code information file unique to each benchmark, and is loaded into GPGPU-Sim at the start of the simulation. The scripts also analyze the compiled binaries to extract the immediate post-dominator of each branch instruction inside the shader program. The PC values of these branch instructions and their immediate post-dominators are also stored in the code information file and are loaded into GPGPU-Sim at the start of the simulation to emulate the effect of adding an extra field in the branch instruction to specify the reconvergence point.

Shader Core	
Shader Core ALU Frequency	650 MHz
# Shader Cores	16
SIMD Warp Size	32 (issued over 4 clocks)
SIMD Pipeline Width	8
# Threads per Shader Core	768
Memory System	
# Memory Modules	8
DRAM Controller Frequency	650 MHz
GDDR3 Memory Timing	t_{CL}=9, t_{RP}=13, t_{RC}=34 t_{RAS}=21, t_{RCD}=12, t_{RRD}=8
Bandwidth per Memory Module	8Byte/Cycle (5.2GB/s)
Memory Controller	out of order (FR-FCFS)
Data Cache Size (per core)	512KB 8-way set assoc. 64B line
# Data Cache Banks (per core)	16
Data Cache Hit Latency	10 cycle latency (pipelined 1 access/thread/cycle)
Dynamic Warp Formation Hardware	
Default Warp Scheduling Policy	Majority
PC-Warp LUT	64 entries, 4-way set assoc.
MHeap LUT	128 entries, 8-way set assoc.
Max-Heap	64 entries

Table 5.2: Hardware Configuration

	From Suite	**Description**	**Branch Divergence**
HMMer	SPEC CPU2006	A modified version of 456.hmmer: Protein sequence analysis using profile hidden Markov models.	High
LBM	SPEC CPU2006	A modified version of 470.lbm: Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D.	Medium
Black	CUDA	Black-Scholes Option Pricing: Evaluates fair call and put prices for a given set of European options by Black-Scholes formula.	Low
Bitonic	CUDA	Bitonic Sort [5]: A simple parallel sorting algorithm. We have extended the version from the CUDA website to work with large arrays (by not using the shared memory).	High
FFT	SPLASH	Complex 1D FFT: We have modified the benchmark so that multiple 1M-point arrays are processed by 12288 threads.	Medium
LU	SPLASH	Blocked LU Decomposition: Each thread is responsible for the decomposition inside a block.	High
Matrix	—	A simple version of matrix multiply: Each thread computes the result for a data element on the destination matrix independently.	None

Table 5.3: Benchmark Description

Chapter 6

Experimental Results

First we consider dynamic warp formation and scheduling with the detailed implementation described in Section 4.2 of Chapter 4 using the Majority scheduling policy with the detailed max-heap implementation described in Section 4.4 of Chapter 4. The hardware is sized as in Table 7.1 (on page 69) and employs the thread swizzling mechanism described in Section 4.1 of Chapter 4. This implementation uses the lane aware scheduling mechanism discussed in Section 4.1 and 4.2 of Chapter 4. We also model bank conflicts at the data cache.

Figure 6.1 shows the performance of the different branch handling mechanisms discussed in this thesis and compares them to a MIMD pipeline with the same peak IPC capability. Here we use the detailed simulation model described in Chapter 5 including simulation of memory access latencies. On average (HM), PDOM (reconvergence at the immediate post-dominator described in Chapter 3.2) achieves a speedup of 44.9% versus not reconverging (NREC). Dynamic warp formation (DWF) achieves a further speedup of 47.4% using the Majority scheduling policy. The average DWF and MIMD performance only differs by 9.5%. The 47.4% speedup of DWF versus PDOM justifies the 8% area cost (see Chapter 7) of using dynamic warp formation and scheduling on existing GPUs. We note that this magnitude of speedup could not be obtained by simply spending this additional area on extra shader cores.

For benchmarks with little diverging control flow, such as FFT and Matrix, DWF performs as well as PDOM, while MIMD outperforms both of them with its "free running" nature. The significant slowdown of Black-Scholes (Black) of DWF is a phenomenon exposing a weakness of our default Majority scheduling policy. This will be examined in detail in Section 6.2.

For most of the benchmarks with significant diverging control flow (HMMer, LBM, Bitonic, LU), MIMD performs the best, and DWF achieves a significant speedup over PDOM. Among them, DWF achieves a speedup for Bitonic and LU purely due to better branch divergence

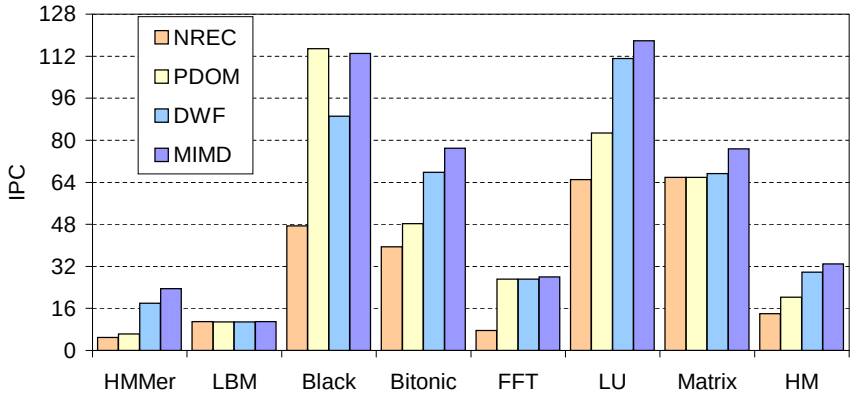


Figure 6.1: Performance comparison of NREC, PDOM, and DWF versus MIMD.

handling, while for HMMer, DWF achieves a speedup in part due to better cache locality as well, as observed from Table 6.2.

While LBM also has significant diverging control flow, it is memory bandwidth limited, as shown in Table 6.1 and therefore sees little gain from DWF. Although the cache miss rate of Bitonic is higher than that of LBM, its much higher pending hit[21] rate significantly lowers the memory bandwidth requirement of this benchmark (see Table 6.3).

6.1 Effects of Scheduling Policies

Figure 6.2 compares all the warp scheduling policies described in Section 4.3 of Chapter 4, with a realistic memory sub-system, but ignoring the impact of lane conflicts (e.g. hardware in Figure 4.2(b) with unlimited register file ports) to show the potential of each policy. Overall, the default Majority (DMaj) policy performs the best, achieving an average speedup of 66.0%, but in some cases, its performance is not as good as the PC policy (DPC) or PDOM Priority (DPdPri) described in Section 4.3 of Chapter 4.

To provide additional insight into the differences between the scheduling policies, Figure 6.3 shows the distribution of warp sizes issued each cycle for each policy. Each bar is divided into segments labeled W0, W4-1, ... W32-29, which indicate if the SIMD hardware executed operations for 0, (1 to 4), ...(29 to 32) scalar threads on a given cycle. "Stall" indicates a

[21] A pending hit occurs when a memory access misses the cache, but can be merged with one of the in-flight memory requests already sent to the memory subsystem.

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	57.47%	94.71%	8.18%	50.94%	75.43%	0.84%	84.02%
DWF	63.61%	95.04%	6.47%	71.02%	74.89%	1.47%	63.71%
MIMD	64.37%	94.99%	8.06%	80.48%	80.99%	1.52%	98.26%

Table 6.1: Memory bandwidth utilization.

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	13.37%	14.15%	1.52%	21.25%	15.52%	0.05%	7.88%
DWF	5.05%	14.56%	1.50%	30.67%	14.68%	0.06%	9.71%
MIMD	3.75%	15.30%	1.20%	30.63%	14.18%	0.05%	7.86%

Table 6.2: Cache miss rates (pending hits[21] classified as a miss).

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	13.21%	13.55%	0.21%	3.86%	11.77%	0.03%	5.72%
DWF	5.03%	13.57%	0.21%	3.84%	12.53%	0.04%	4.23%
MIMD	3.74%	13.35%	0.21%	3.83%	12.92%	0.04%	5.75%

Table 6.3: Cache miss rates without pending hits[21].

stall due to writeback contention with the memory system (see Figure 2.3(b) on Page 15). For policies that do well (DMaj, DPdPri, DPC), we see a decrease in the number of low occupancy warps relative to those policies which do poorly (DMin, DTime). Cycles with no scalar thread executed (W0) are classified into "Mem" (W0_Mem) and "Idle" (W0_Idle). W0_Mem denotes that the scheduler cannot issue any scalar thread because all threads are waiting for data from global memory. W0_Idle denotes that all warps within a shader core have already been issued to the pipeline and are not yet ready for issue. These "idle" cycles occur because threads are assigned to shader cores in blocks. In our current simulation, the shader cores have to wait for all threads in a block to finish before beginning with a new block. The warp size distribution for Black-Scholes reveals that one reason for the Majority (DMaj) policy's poor performance on this benchmark is a significant number of "Idle" cycles, which can be mostly eliminated by the PDOM Priority (DPrPri) policy. The data also suggest that it may be possible to further improve dynamic warp formation by exploring scheduling policies beyond those proposed in this thesis.

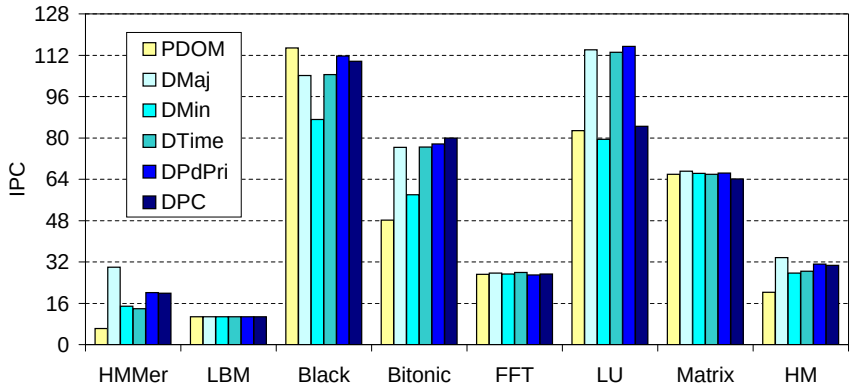


Figure 6.2: Comparison of warp scheduling policies. The impact of lane conflicts is ignored.

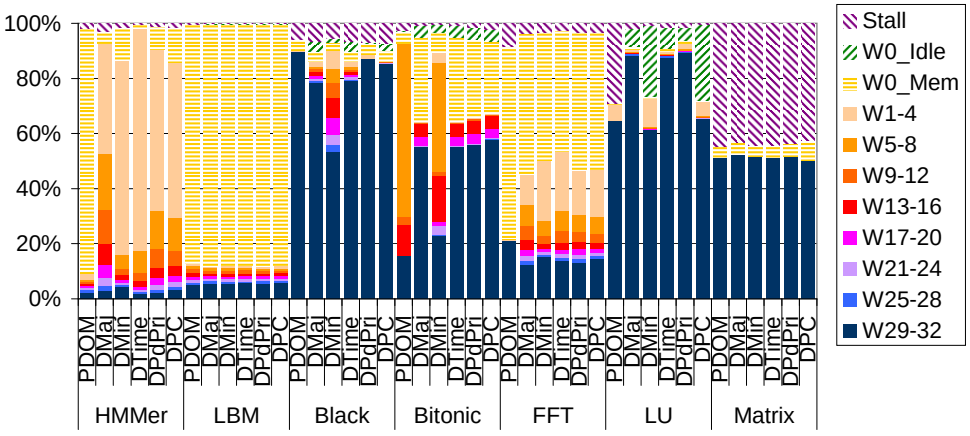


Figure 6.3: Warp size distribution.

6.2 Detail Analysis of Majority Scheduling Policy Performance

The significant slowdown of Black-Scholes (Black) results from several phenomenon. First, in this work we restrict a shader core to execute a single block. Second, we use software subroutines for transcendentals, and these subroutines contain branches that diverge. Third, a quirk in our default Majority scheduling policy can lead some of the threads in a shader core to starvation. We explore the latter phenomenon in detail in this section. Figure 6.4 shows the runtime behaviour (IPC and incomplete thread count[22]) of a single shader core using Dynamic Warp Formation with the Majority scheduling policy.

Under the Majority scheduling policy, threads which have different control flow behaviour

[22] A thread is incomplete if it has not reached the end of the kernel call.

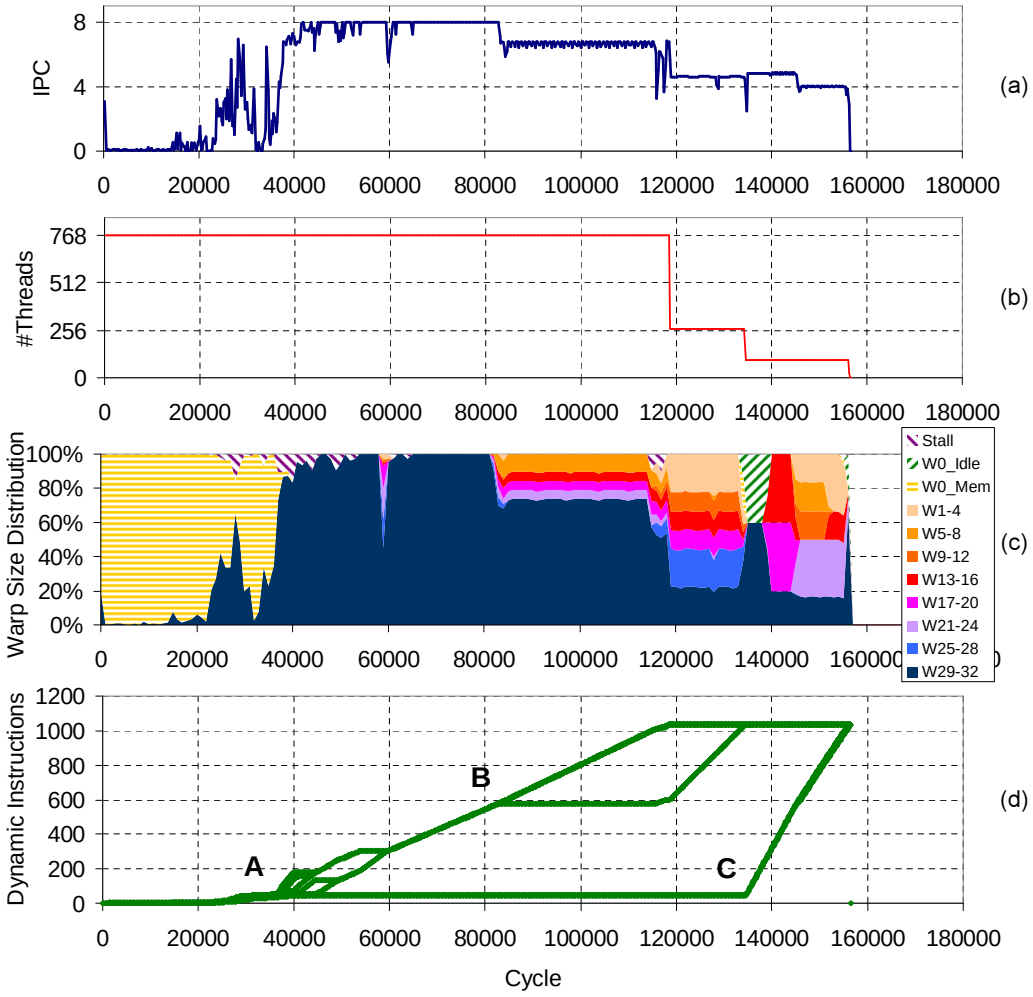


Figure 6.4: Dynamic behaviour of Black-Scholes using DWF with Majority scheduling policy in a single shader core (max IPC of 8). The warp size distribution time series in (c) uses the same classification as in Figure 6.3.

from the rest of the threads can be starved during execution. Black-Scholes has several rarely executed, short basic blocks that suffer from this effect, leaving behind several groups of minority threads[23]. When these minority threads finally execute after the majority of threads have finished, they form incomplete warps and the number of warps formed are insufficient to fill up the pipeline (in our simulations, each thread is only allowed to have one instruction executing in the pipeline, as described in Section 2.4 in Chapter 2). This behaviour is illustrated in Figure 6.4(d) which shows the dynamic instruction count of each thread. After several

[23]We say a thread is a "minority" thread if its PC value is consistently given a low priority by the Majority scheduling policy throughout its execution.

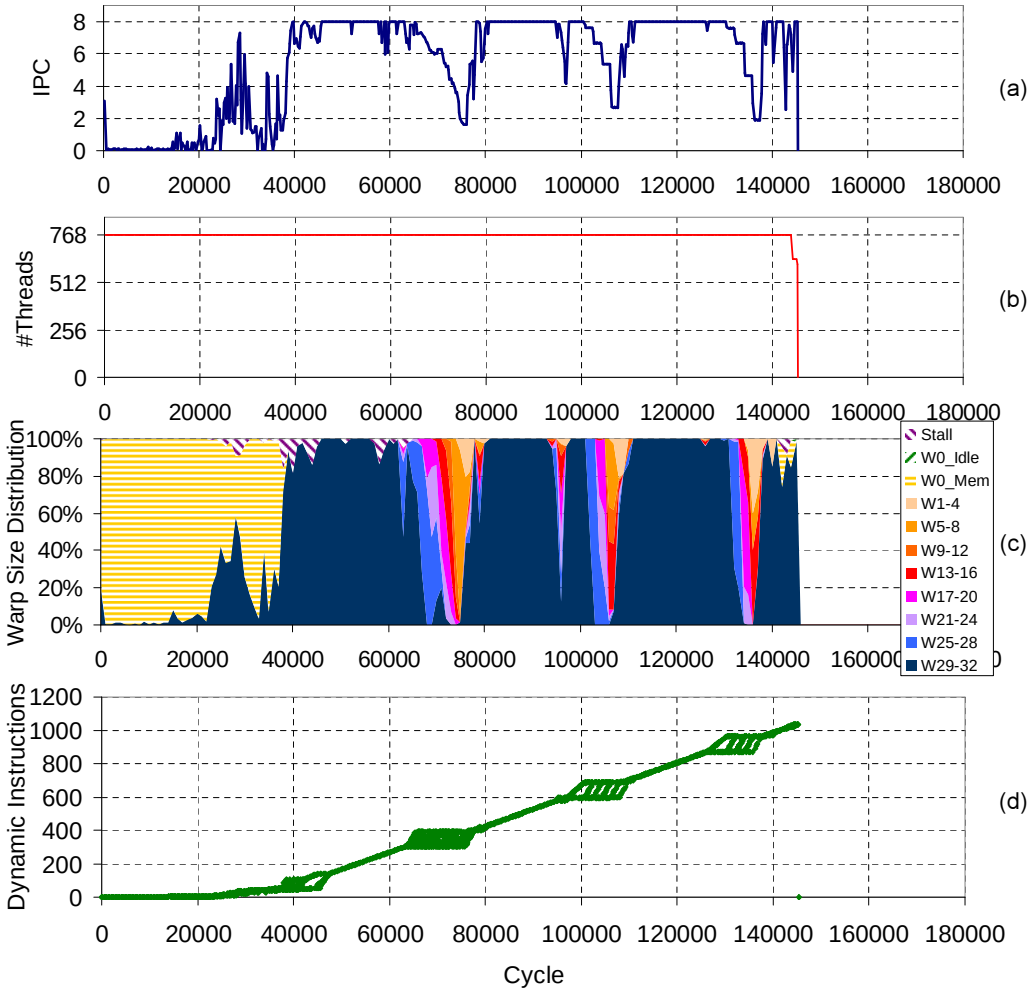


Figure 6.5: Dynamic behaviour of Black-Scholes using DWF with PDOM Priority scheduling policy.

branches diverged at A and B, groups of threads are starved (indicated by the stagnant dynamic instruction count in Figure 6.4(d)), and are only resumed after C when the majority groups of threads have finished their execution (indicated by the lower thread count after cycle 120000 in Figure 6.4(b)). This is responsible for the low IPC of DWF after cycle 120000 in Figure 6.4(a).

Meanwhile, although the majority of threads are proceeding ahead after branch divergences at A and B, the pipeline is not completely utilized (indicated by the IPC<8 from cycle 80000 to 120000 in Figure 6.4(a)) due to the existence of incomplete warps (see the warp size distribution time series in Figure 6.4(c)). These incomplete warps are formed because the number of threads taking the same execution path is not a multiple of the SIMD width. They could have been

combined with the minority threads after the diverging branch to minimize this performance penalty, but this does not happen when the minority threads are starved by the Majority policy.

Figure 6.5 shows how both of these problems can be mitigated by having a different scheduling policy—PDOM Priority. The dynamic instruction count in Figure 6.5(d) shows that any thread starvation due to divergence is quickly corrected by the policy and all the threads have a similar rate of progress. The IPC stays at 8 for most of the execution (see Figure 6.5(a)), except when the policy is giving higher priorities to the incomplete warps inside the diverged execution paths (shown in Figure 6.5(b)) so that they can pass through the diverged execution paths quickly to form complete warps at the end of the diverged paths. Overall, threads in a block finish uniformly as shown in Figure 6.5(b), indicating that starvation does not happen with this scheduling policy.

6.3 Effect of Limited Resources in Max-Heap

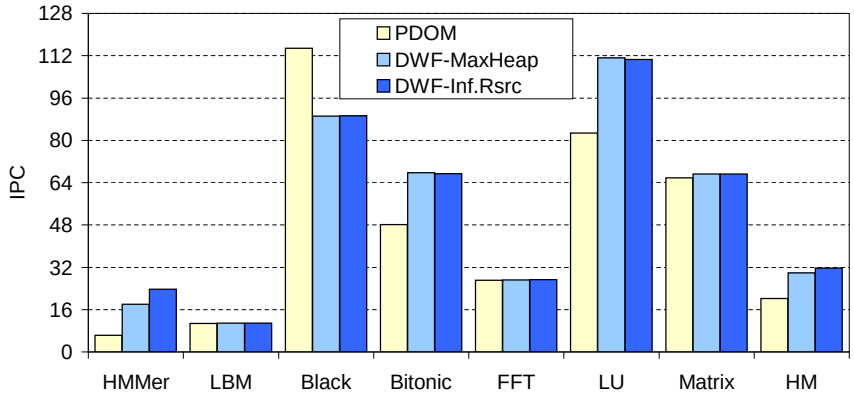


Figure 6.6: Performance comparison of a resource-limited version of Max-heap (DWF-MaxHeap) with an ideal, infinite resource implementation of Majority scheduler (DWF-Inf.Rsrc).

Figure 6.6 shows the performance increase achievable by dynamic warp formation with an infinite amount of hardware resources (infinitely ported and unlimited entries for MHeap LUT and Max-Heap) given to the Majority scheduler. This unbounded version of the Majority scheduler has a speedup of 6.1% over its resource-limit counterpart, and is 56.3% faster than PDOM on average. This 6.1% speedup comes completely from HMMer, which is both a control

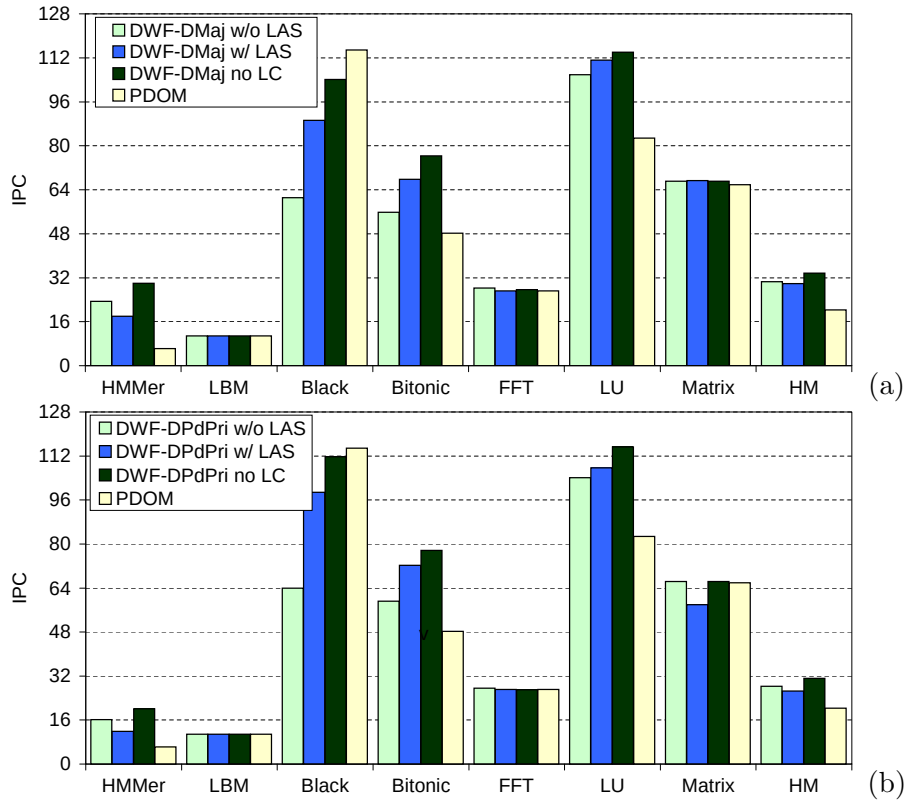


Figure 6.7: Performance of dynamic warp formation evaluating the impact of lane aware scheduling and accounting for lane conflicts and scheduler implementation details. (a) Using Majority scheduling policy. (a) Using PDOM Priority scheduling policy. LAS = Lane Aware Scheduling LC = Lane Conflict

flow and data flow intensive benchmark. These properties of HMMer results in a large number of in-flight PC values among threads as memory access from a warp following a branch divergence can introduce variable slowdown among the threads in a warp ranging from tens to hundreds of cycles. The large number of in-flight PC values in turn requires a large Max-Heap, which takes more swaps to re-balance every scheduler cycle and introduces stalls when the limited bandwidth resources are exhausted. With other benchmarks, however, the resource-limited version of the Majority scheduling policy logic is sufficient to achieve similar performance.

6.4 Effect of Lane Aware Scheduling

To reduce register file design complexity of DWF we have chosen to use the organization in Figure 4.2(d) on Page 26 which necessitates the use of lane aware scheduling discussed in

Sections 4.1 and 4.2 of Chapter 4. The data in Figure 6.7(a) compares the detailed scheduler hardware model we have considered so far with an idealized version of dynamic warp formation and scheduling ignoring the impact of lane conflict and hardware resources (DWF-DMaj no LC). This figure shows the impact on the performance of lane aware scheduling and, for comparison, also shows the impact when not using lane aware scheduling but assuming the register file organization in Figure 4.2(b) and modeling register bank conflicts when multiple threads from the same "home" lane are grouped into a single warp (DWF-DMaj w/o LAS). While the idealized version of DWF is on average 66.0% faster than PDOM, the realistic implementation of DWF we have considered so far is able to achieve 89% of the idealized DWF's performance. The average performance of DWF improves without lane aware scheduling, because we did not impose a hardware resource limit on the scheduler.

We also evaluated the performance of the PDOM Priority policy (DPdPri) with lane aware scheduling (see Figure 6.7(b)), and found performance for Black-Scholes improves (IPC of 98.9 vs. IPC of 89.3 for DMaj) while that of HMMer is reduced (IPC of 11.9 vs. IPC of 23.7 for DMaj) with an overall average speedup of 30.0% over PDOM. Thus, DMaj is the best scheduling policy overall.

6.5 Effect of Cache Bank Conflict

Our baseline architecture model assumes a multi-ported data cache that is implemented using multiple banks of smaller caches. Parallel accesses from a warp to different lines in the same banks create cache bank conflicts, which can only be resolved by serializing the accesses and stalling the SIMD pipeline.

With a multi-banked data cache, a potential performance penalty for dynamic warp formation is that the process of grouping threads into new warps dynamically may introduce extra cache bank conflicts. We have already taken this performance penalty into account by modeling cache bank conflicts in our earlier simulations. However, to evaluate the effect of cache bank conflicts on different branch handling mechanisms, we rerun the simulations with an idealized cache allowing threads within a warp to access any arbitrary lines within the cache in parallel.

The data in Figure 6.8 compares the performance of PDOM, DWF and MIMD ignoring

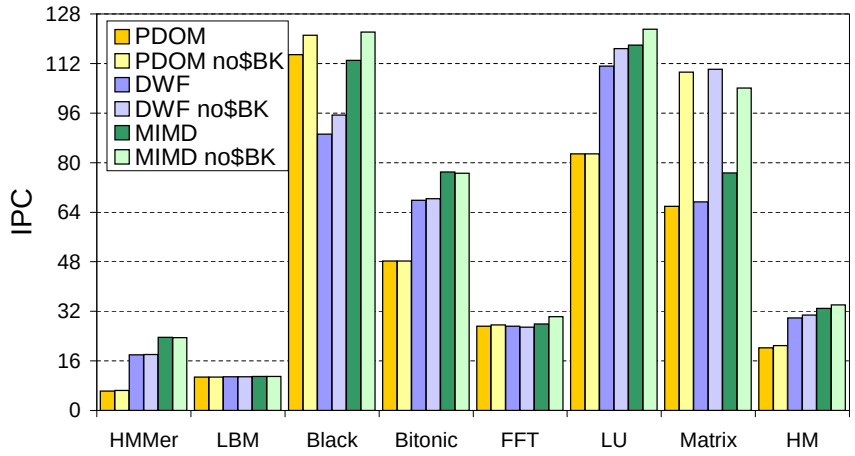


Figure 6.8: Performance of PDOM, DWF and MIMD with cache bank conflict. "no$Bk" = Ignoring Cache Bank Conflict

cache bank conflicts. It indicates that cache bank conflicts have more effect on benchmarks that are less constrained by the memory subsystem (Black-Scholes, LU and Matrix). Overall, performance of these benchmarks increase by a similar amount with an ideal multi-ported cache regardless of the branch handling mechanism in place. Hence, we find DWF still has a speedup of 47% over PDOM.

6.6 Sensitivity to SIMD Warp Size

While DWF provides a significant speedup over PDOM with our default hardware configuration, we can gain further intuition into its effectiveness in handling branch divergence as SIMD warp size increases. Figure 6.9 shows the performance of each mechanism for three configurations with increasing warp size from 8 to 32. Notice that the width of the SIMD pipeline is not changed, so that a decrease in warp size translates to a shorter execution latency for each warp[24].

None of the benchmarks benefit from SIMD warp size increases. This is expected as increasing warp size in an area constrained fashion, as described above, does not increase the peak throughput of the architecture while it constraints control flow and thread scheduling flexibility. The benefit of increasing SIMD warp size is to improve computational density by relaxing

[24] As described in Section 2.5 of Chapter 2, a warp with 32 threads takes 4 cycles to execute on a 8-wide SIMD pipeline, whereas a warp with 8 threads can be executed in a single cycle on the same pipeline.

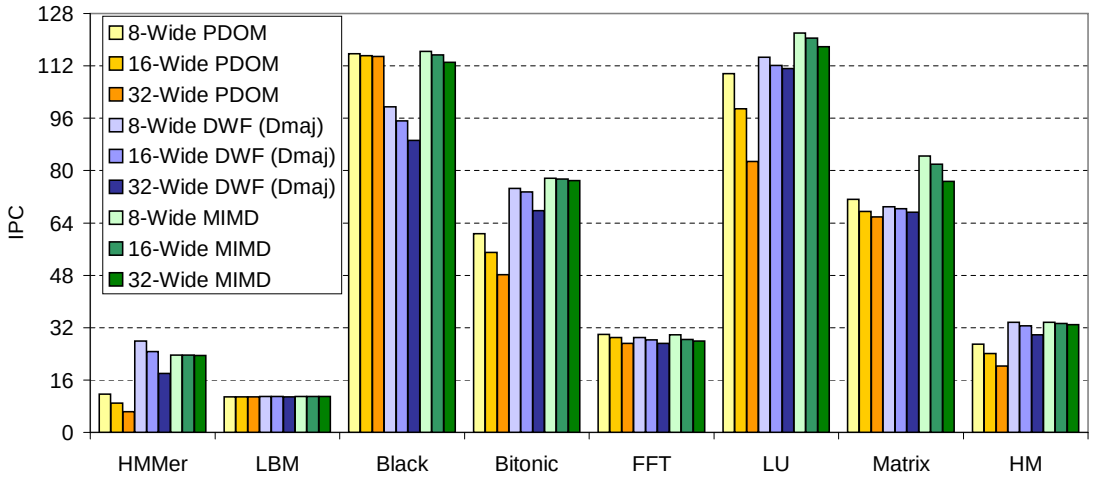


Figure 6.9: Performance comparison of PDOM, DWF and MIMD with a realistic memory subsystem as SIMD warp size increases. The theoretical peak IPC remains constant for all configurations.

the scheduler's latency requirement, reducing the number of warps to schedule (simplifying scheduler hardware) and lowering instruction cache bandwidth requirements.

The slowdown of MIMD as SIMD warp size increases in various benchmarks is a result of a longer scheduler cycle, which places some constraints of SIMD scheduling onto the MIMD scheduler. For example, with a warp size of 32, after the MIMD scheduler has issued less than 32 threads (other threads are not issuable as they are pending for data from memory) in a scheduler cycle, it has to wait until the next scheduler cycle (4 pipeline cycles later) to issue more threads to the pipeline, even if more threads become available for execution in the meantime. For benchmarks that are constrained by the memory subsystem (HMMer, lbm and Bitonic), this effect does not cause any significant slowing for MIMD as the scheduler frequently runs out of threads that are ready to be issued due to memory access latency.

Overall, as SIMD warp size increases from 8 to 32, the average performance of PDOM decreases by 24.9%, while the overall performance of DWF and MIMD decreases by 11.2% and 2.0% respectively. Most of the slowdowns experienced by PDOM as SIMD warp size is increased is attributed to the control flow intensive benchmarks (HMMer, Bitonic, and LU), while these slowdowns are alleviated with DWF. This trend shows that branch divergence becomes a more serious performance bottleneck in control flow intensive applications as SIMD warp size is increased to improve computational density, but a significant portion of this performance loss

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
Max(Warp Pool Occupancy)	90	25	42	39	62	40	23
Max(Max-Heap Size)	44	16	10	8	24	11	7

Table 6.4: Maximum warp pool occupancy and max-heap size (in #Entries) for each benchmark.

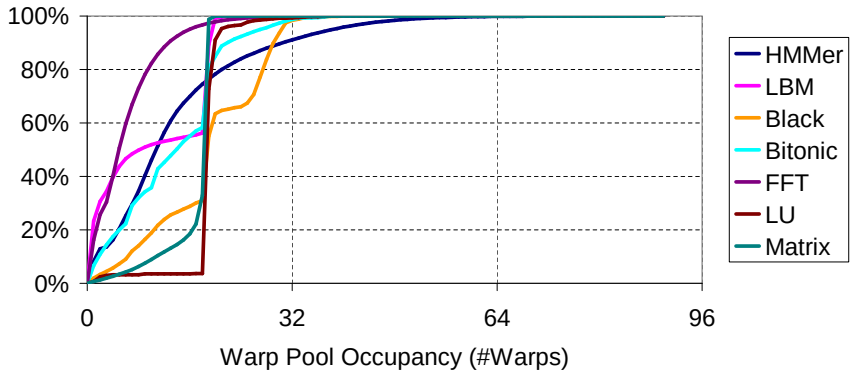


Figure 6.10: Cumulative distribution of warp pool occupancy for each benchmark.

can be regained using dynamic warp formation and scheduling.

6.7 Warp Pool Occupancy and Max Heap Size

Table 6.4 shows the maximum warp pool occupancy (the number of warps inside the warp pool) and the maximum dynamic size of the max-heap for each benchmark. As shown in the table, all of the benchmarks, including the control flow and memory intensive HMMer, use less then 1/6 of a warp pool sized to the worst case requirement. This indicates that it should be possible to use a warp pool with 128 entries (1/6 of the 768-entry warp pool designed for the absolute worst case) without causing any performance penalty. The same argument can be used to justify that 64 entries is sufficient for the max-heap used in implementing the Majority scheduling policy.

Furthermore, Figure 6.10 shows the cumulative distribution of warp pool occupancy for each benchmark used in this thesis. This distribution indicates that it may be feasible to reduce the size of the warp pool to 64 entries, and handle <0.15% warp pool overflow (or 10% if we reduce the warp to 32 entries) by spilling existing warp pool entries to the global memory. As our simulator does not currently model this, we do not use this assumption in the area estimation in Chapter 7 and leave this to future work.

Chapter 7

Area Estimation

The total area cost for dynamic warp formation and scheduling is the amount of hardware added for the logic in Figure 4.3 (Page 28) plus the overhead of an independent decoder for each register file bank. We have estimated the area of the five major parts of the hardware implementation of dynamic warp formation and scheduling with CACTI 4.2 [72]: Warp update registers, PC-warp LUT, warp pool, warp allocator, and scheduling logic. Table 7.1 lists the implementation of these structures and their area and storage estimates. We use our baseline configuration (32-wide SIMD with 768 threads) listed in Table 5.2 to estimate the size of the PC-Warp LUT and MHeap LUT. Both are set-associative cache-like structures with two read ports and two write ports capable of two warp lookups in parallel to handle requests from the two diverged parts of a single incoming warp. Based on the maximum occupancy data in Section 6.7 of Chapter 6, we use a 128-entry Warp Pool and a 64-entry Max-Heap. The Max-Heap is implemented with a memory array with two read ports and two write ports as discussed in Section 4.4 of Chapter 4, and the Warp Pool is implemented using a banked structure described in Section 4.2.1 in Chapter 4.

For this area estimation, we have chosen to use a 10-bit thread ID, which should be sufficient to identify the 768 threads in each shader core. We acknowledge that more bits may be needed when shader cores can execute multiple blocks simultaneously as in Geforce 8800GTX, but their impact on the overall area is not significant. We assumed that a 24-bit integer is sufficient to store the PC value for each thread as the maximum number of instructions per kernel call in CUDA is limited to 2 million instructions [58]. While this can be represented with a 21-bit number, we have allocated a few extra bits to account for the possibility of the GPU executing multiple parallel sections concurrently (for example, the Geforce 8 Series can execute vertex, geometry, and pixel shaders in parallel). Referring to Figure 4.3, 32 bits are required by both

Structure	# Entries	Entry Content	Struct. Size (bits)	Implementation	Area (mm^2)
Warp Update Register	2	TID (10-bit) × 32 PC (24-bit), REQ (32-bit)	752	Registers (No Decoder)	0.0080
PC-Warp LUT	64	PC (24-bit) OCC (32-bit) IDX (7-bit)	4032	4-Way Set-Assoc. Mem. (2 RP, 2 WP)	0.2633
Warp Pool	128	TID (10-bit) × 32 PC (24-bit) Scheduler Data (10-bit)	44928	Memory Array (33 Banks) (1 RP, 1 WP)	0.6194
Warp Allocator	128	IDX (7-bit)	896	Memory Array	0.0342
Mheap LUT	128	PC (24-bit) MH# (7-bit)	4992	8-Way Set-Assoc. Mem. (2 RP, 2 WP)	0.4945
Max-Heap	64	PC (24-bit), CNT (10-bit) WPH (8-bit), WPT (8-bit) LUT# (8-bit)	3712	Memory Array (2 RP, 2 WP)	0.2159
Total			59312		**1.6353**

Table 7.1: Area estimation for dynamic warp formation and scheduling. RP = Read Port, WP = Write Port.

REQ and OCC to represent the lanes in a 32-wide SIMD warp. Both the IDX field in the PC-Warp LUT and one in the Warp Allocator uses only 7 bits because this is sufficient to address all 128 entries in the Warp Pool. Referring to Figure 4.5, in the Max-Heap, while WPH and WPT also index into the Warp Pool, they both require an extra valid bit. The same argument applies to the Scheduler Data in the Warp Pool, which translates to the "Next" pointer in Majority scheduling policy. LUT# and MH# in Figure 4.5 both also use an extra bit to represent an invalid index. Finally, the CNT field in the Max-Heap has to be 10-bits to account for the case when all 768 threads in a shader core have the same PC.

Overall, we have estimated the area of the dynamic warp scheduler in 90nm process technology to be 1.635 mm^2 per core. The exact CACTI parameters we used for each structure are listed in Table 7.2. Notice that we have skipped listing the CACTI parameters for the Warp Update Registers because they are too small to be evaluated in CACTI. The parameter entries for the Warp Pool is separated into the ones for TID banks and the ones for PC banks. As discussed in Section 4.2.1 of Chapter 4, each Warp Pool is comprised of 32 TID banks and 1 PC bank.

Structure	Model	SRAM/ Cache Size (Bytes)	Line Size (Bytes)	Assoc.	#Banks	Tech. Node	#RWP	#RP	#WP	#SERP	#Bits Out	#Tag Bits	Type
PC-Warp LUT	Cache	512	8	4	1	90nm	0	2	1	0	48	24	Fast
Warp Pool (TID Bank)	SRAM	160	10	N/A	N/A	90nm	0	1	1	0	10	N/A	Fast
Warp Pool (PC Bank)	SRAM	704	11	N/A	N/A	90nm	0	1	2	0	42	N/A	Fast
Warp Allocator	SRAM	128	8	N/A	N/A	90nm	0	1	1	0	14	N/A	Fast
Mheap LUT	Cache	1024	8	8	1	90nm	0	2	2	0	10	24	Fast
Max-Heap	SRAM	512	8	N/A	N/A	90nm	0	2	2	0	64	N/A	Fast

Table 7.2: CACTI parameters for estimating structure sizes in Table 7.1. RWP = Read/Write Port, RP = Read Port, WP = Write Port, SERP = Single Ended Read Port.

To evaluate the overhead of having the individual decoders for dynamic warp formation and scheduling as described in Chapter 4, we first need to estimate the size of the register file. The SRAM model of CACTI 4.2 [72] estimates a register file with 8192 32-bit registers and a single decoder reading a row of eight 32-bit registers to be 3.863 mm^2. After trying various line sizes, we found CACTI 4.2 predicts that a line size of 512 Bytes results in minimum area. On the other hand, since the SRAM model of CACTI 4.2 does not support banking directly, we estimate the area of combining eight identical copies of a register file by 1024 32-bit registers. The register file area is then estimated to be 0.5731 $mm^2 \times 8 = 4.585$ mm^2. For this area estimation, we have divided the 512 Bytes line in the single decoder estimation into eight 64 Bytes lines in each bank. Notice that both register file configurations have 2 read ports and 1 write port, and each port is accessing 32-bit. The area difference between the two register files is 0.722 mm^2. This is our estimation of the area requirement for adding a decoder to every register file bank to support DWFS. We acknowledge that this method may under-estimate the area requirement for adding a decoder to every register file bank since it may not entirely capture all wiring complexity as the decoders are driven by different sources in our proposal. However, this is already proportionally larger than the 11% overhead estimate by Jayasena et al. [30] with CACTI and a custom floorplan for implementing a stream register file with indexed access, which also requires multiple decoders per register file (11% of 3.863 mm^2 is only 0.425 mm^2, which is smaller than our 0.722 mm^2.). Ultimately, without an actual layout of a GPU, the 0.722 mm^2 overhead is our best estimate.

Combining the two estimations above, the overall area consumption of dynamic warp formation and scheduling for each core is 1.6353 + 0.722 = 2.357 mm^2. With 16 cores per chip as per our baseline configuration, this becomes 37.72 mm^2, which is 8.02% of the total area of the GeForce 8800GTX (470 mm^2) [39].

Chapter 8

Related Work

This chapter discusses research related to this thesis. Section 8.1 discusses various mechanisms for supporting control flow on SIMD hardware. Section 8.2 explores two prior proposals that involve dynamically regrouping of scalar threads into SIMD instructions. Section 8.3 discusses prior proposals for eliminating branch divergence in SIMD hardware.

On average, dynamic warp formation and scheduling performs significantly better than most existing mechanisms for supporting control flow on SIMD hardware. It is applicable to a widely SIMD architectures with fine-grained multithreading and does not require any software modifications.

8.1 SIMD Control Flow Handling

This section discusses existing mechanisms for supporting control flows on SIMD hardware. We classify these mechanisms into three groups: guarded instructions, control flow reconvergence, and conditional streams.

8.1.1 Guarded Instruction/Predication

While supporting branches is a relatively new problem for GPU architectures, it has long been a consideration in the context of traditional vector computing. Most of the approaches to supporting branches in a traditional SIMD machine have centered around the notion of guarded instructions [8].

A guarded instruction, also known as a predicated or vector masked instruction, is an instruction whose execution is dependent on a conditional mask controlled by another instruction [8]. If the conditional mask is set, appropriately the predicated instruction will not update architecture state. In a masked SIMD instruction, a vector of conditional masks, each con-

trolled by an element in a stream, is functionally equivalent to a data dependent branch. This approach has been employed by existing GPU architectures to eliminate short branches and potential branch divergences [3, 58]. However, for longer branches and loops, guarded instructions are inefficient because the SIMD hardware has to execute every single execution path, regardless of whether they are taken by any of the SIMD elements.

This inefficiency is mitigated by a proposed technique called *branches on superword condition codes* (BOSCCs) [68]. BOSCC permits a sequence of consecutive vector instructions guarded by the same vector predicate to be bypassed if all fields in the guarding vector predicate are false. Shin et al. [68] examine compiler generation of BOSCCs for handling control flow for SIMD instruction set extensions and show that it is effective for some kernels, but do not explore the scalability of the approach to long SIMD vector width.

8.1.2 Control Flow Reconvergence Mechanisms

Guarded instructions and their variants, put constraints on input dependent loops. Branch divergence may be inevitable, but the period of divergence can be kept short with reconvergence to minimize performance lost due to unfilled SIMD pipelines. A patent filed by Lindholm et al. describes in detail how threads executing in a SIMD pipeline are serialized to avoid hazards [49], but does not indicate the use of reconvergence points to recover from such divergence. The notion of reconvergence based on control flow analysis in SIMD branch handling was described in a patent by Lorie and Strong [43]. However, this patent proposes to insert the reconvergence point at the beginning of a branch and not at the immediate post-dominator as proposed in this thesis.

8.1.3 Conditional Streams

Kapasi et al. [32] introduce *conditional streams*, a code transformation that creates multiple kernels from a single kernel with conditional code and connects these kernels via inter-kernel communication to increase the utilization of a SIMD pipeline. While being more efficient than predication, it may only be practical to implement on architectures with a software managed on-chip memory designed for inter-kernel communication, such as the stream register file on Imagine [63] and Merrimac [18]. With conditional streams, each kernel ends at a conditional

branch, which splits a data stream into two, each to be processed by one of the two subsequent kernels (each representing a execution path from the conditional branch). The data streams are then stored to the stream register file, and a filter would restrict the data stream to only be loaded when its destined kernel is executing. When the two kernels representing execution paths after the conditional branch have finished processing all the data elements, a "merger kernel" merges the two diverged data streams back into one again and proceeds on processing them uniformly.

For this scheme to be effective, an interprocessor switch is added to route data streams from the stream register file to the appropriate processing elements for load balancing. Also, the overhead of creating a filter and merger for each diverging conditional branch can be a performance bottleneck in a control flow intensive kernel. Dynamic warp formation differs from this approach in that it is a hardware mechanism exploiting the dynamic conditional behaviour of each scalar thread, and implementation does not require a stream register file nor data movement between register lanes.

8.2 Dynamic Grouping SIMD Mechanisms

This section discuss two proposals that are similar to our proposed dynamic warp formation in so far as they describe scalar threads or instructions are grouped into SIMD instructions at runtime.

8.2.1 Dynamic Regrouping of SPMD Threads for SMT Processors

The notion of dynamically regrouping the scalar SPMD threads comprising a single SIMD "task" after control flow divergence of the SPMD threads was described by Cervini [12] in the context of simultaneous multithreading (SMT) on a general purpose microprocessor that provides SIMD function units for exploiting subword parallelism. However, the mechanism he proposes does not specify any mechanism for grouping the diverged SPMD threads, whereas such an implementation is one of the main contributions of this thesis.

Also, the mechanism Cervini proposes requires that tasks have their register values reloaded each time threads are regrouped. To avoid performance penalties, Cervini proposes that the

register file contain additional ports to enable the register values to be loaded concurrently with ongoing execution. In addition, Cervini's mechanism uses special "code stops" and tags the control flow state of a SPMD thread with a loop counter list (in addition to the program counter). We point out that in the context of a modern GPU the constant movement of data in this proposal could increase power requirements per processing element, perhaps mitigating the improvements in processing efficiency given the growing importance of the so-called "power wall" [52]. In contrast, our proposal uses a highly banked large register file and maintains a thread's registers in a single location to eliminate the need for movement of register values.

8.2.2 Liquid SIMD

Clark et al. [14] introduce *Liquid SIMD* to improve SIMD binary compatibility on general purpose CPUs by forming SIMD instructions at runtime by translating annotated scalar instructions with specialized hardware. In their proposal, when the annotated scalar instructions reach the commit stage of a superscalar pipeline, a special hardware unit will translate these scalar instructions into microarchitecture-specific SIMD instructions (i.e., SIMD instructions with fixed width according to the hardware), and run the translated instructions on the SIMD functional units in the processor. In this manner, the grouping of scalar instructions into SIMD instruction is only done once in the hardware before their execution. In contrast, this thesis focuses on improving control flow efficiency of throughput-oriented architectures with fine-grained multithreading. As dynamic control flow behaviour is unknown prior to a program's execution, our proposed mechanism is capable of regroup threads into new SIMD warps even after threads start executing.

8.3 Eliminating the Existence of Branch Divergence

This section discuss two schemes to eliminate branch divergence in SIMD hardware. One scheme is to used a complex SIMD branch instruction which always has only one branch outcome for a SIMD instruction. Another is to allow processing elements in MIMD hardware to share instruction bandwidth in the common case, and permit them to operate independently when control flow diverges.

8.3.1 Complex SIMD Branch Instruction

Besides using a stack-based reconvergence mechanism to handle diverging control flow in their Ray Processing Unit, Woop et al. [80] proposed a complex SIMD branch instruction which combines the branch outcome of a set of masked elements in a SIMD batch to a single find branch outcome with a reduction function. For example, the programmer may specify the branch to be taken only when all the elements inside a SIMD batch evaluate to true. In this case, the reduction function would be a logical AND of the branch outcomes of all elements in that SIMD batch. In this manner, branch divergence never exists as the final branch outcome is always consistent for all elements in a SIMD warp (as there is only one branch outcome). However, this proposal may not be suitable for general-purpose applications requiring the flexibility of allowing each thread to traverse in a unique execution path.

8.3.2 Vector-Thread Architecture

Krashinsky et al. [33] propose the vector-thread architecture which exposes instruction fetch in the ISA. It provides two fetch mechanisms: vector-fetch and thread-fetch. Vector-fetch is issued by a control processor to command all virtual processors (VPs) (similar to our notion of scalar threads) to fetch the same atomic instruction block (AIB) for execution, whereas thread-fetch can be issued by each VP to fetch an AIB to itself alone. Instruction bandwidth is efficiently shared by multiple VPs as in normal SIMD hardware when they are executing on instruction fetched by vector-fetch. Branch divergence does not exist in this architecture as each VP executes on it own after a vector-fetch. However, instruction bandwidth can become a bottleneck when each VP is executing a different execution path and may increase bandwidth pressure on the L1-cache with thread-fetches. Also, this architecture requires each processing element to have its own control logic, which eliminates a key merit of having SIMD hardware. One the other hand, our proposed dynamic warp formation retains this merit by grouping scalar threads dynamically into SIMD warps in the scheduler and issue them to a SIMD pipeline for execution.

Chapter 9

Conclusions and Future Work

This chapter concludes the thesis. First, Section 9.1 provides a brief summary and conclusions. Second, Section 9.2 lists the contributions made by this thesis. Finally, Section 9.3 discusses several important areas for future work in this area.

9.1 Summary and Conclusions

In this thesis, we explore the impact of branch divergence on GPU performance for non-graphics applications. Without any mechanism to handle branch divergence, performance of a GPU's SIMD pipeline degrades significantly. While existing approaches to reconverging control flow at join points such as the immediate post-dominator improve performance, we found significant performance improvements can be achieved with our proposed dynamic warp formation and scheduling mechanism. We described and evaluated a implementation of the hardware required for dynamic warp formation and tackle the challenge of enabling correct access to register data as thread warps are dynamically regrouped and found performance improved by 47.4% on average over a mechanism comparable to existing approaches—reconverging threads at the immediate post-dominator. Furthermore, we estimated the area of our proposed hardware changes to be around 8% if it is to be implemented on a modern GPU such as Geforce 8800GTX [39].

Our experimental results also highlight the importance of careful prioritization of threads for scheduling in such massively parallel hardware, even when individual scalar threads are executing the same code in the same program phase.

9.2 Contributions

This thesis makes the following contributions:

1. It quantifies a performance gap of 66% between the immediate post-dominator branch reconvergence mechanism and a MIMD architecture with the same peak operation throughput. Thus, highlighting the importance of finding better branch handling mechanisms.

2. It proposes and evaluates a novel hardware mechanism, dynamic warp formation, for regrouping threads of individual SIMD warps on a cycle-by-cycle basis to greatly improve the efficiency of branch handling. For the data parallel, non-graphics applications we studies in this thesis, dynamic warp formation and scheduling achieves a speedup of 47% over the immediate post-dominator branch reconvergence mechanism (with 768 threads per shader core).

3. It shows quantitatively that warp scheduling policy (the order in which the warps are issued from the scheduler) affects both the performance gains and area overhead of dynamic warp formation, and proposes an area efficient implementation of a well-performing *Majority* scheduling policy with max-heap.

4. It proposes and evaluates a detailed hardware implementation of dynamic warp formation and scheduling. We estimate the hardware required by this hardware implementation adds 8% to the total chip area if it is implemented on a modern GPU such as Geforce 8800GTX ($470mm^2$) [39]. This does not include any area exclusive to the immediate post-dominator branch reconvergence mechanism that might be subtracted from the baseline hardware.

5. It provides an extensive simulation infrastructure for enabling future research on GPU architectures optimized to support non-graphics applications.

9.3 Future Work

The work in this thesis is leads to several new areas of research. This section briefly lists and discusses some of them.

9.3.1 Better Warp Scheduling Policies

In this thesis, we have observed that warp scheduling policies have significant impact on both area (0.71 mm^2, or 43%, of the 1.64 mm^2 dynamic warp formation scheduler area is spent on implementing the Majority policy—See Chapter 7) and performance of dynamic warp formation. Performance varied from 37% to 66% across the scheduling policies we evaluated in Section 6.1 of Chapter 6. While we have explored several different scheduling policies and obtained significant speedup, the analysis in Chapter 6 indicates that our best performing policy (*Majority*) may only provide a fraction of the performance potential of dynamic warp formation. A more effective policy, designed to better encourage warp recombination may further improve performance and be more robust (i.e., it should not cause starvation of threads seen in Section 6.2 of Chapter 6).

In addition, our best performing Majority policy requires sorting hardware which significantly increases the area overhead (by about 77%) of dynamic warp formation. In this respect, policies that need less area should be explored in the future as well.

9.3.2 Area Efficient Implementation of the Warp Pool

As mentioned in Chapter 4, it is possible to implement the warp pool with a single memory array and thus reduce the area of the warp pool significantly if it is clocked at the SIMD pipeline ALU frequency and serializes the 4 updates required per scheduler cycle. This brings up an interesting trade-off between power and area, as clocking the warp pool faster may consume more energy but require a smaller structure.

Another option, we have discussed in Section 6.7 of Chapter 6, is to reduce the area of the warp pool by spilling existing warp pool entries to global memory. While this can improve area overhead of dynamic warp formation, such a spilling mechanism may also introduce a performance penalty, as warps spilled to the global memory may no longer be merged with

threads from incoming warps at the scheduler. It is also unclear how the scheduling policy should be changed to accommodate such a mechanism. In particular, it is uncertain what the scheduler should do when the policy selects a warp that is spilled to the global memory. Or, taking this one step further, the scheduling policy may have to account for memory access latency required to load the spilled warps from global memory. The exact warp spilling mechanism, as well as how the scheduling policy should accommodate this change in general should be explored in the future.

Note that these two proposals are independent from each other, and it may be possible to combine them for a highly area efficient implementation of the warp pool.

9.3.3 Bank Conflict Elimination

Bank conflicts in shared memory access, as noted by Hwu et al. [27], have been a major performance bottleneck in GPU's performance. While this thesis focuses on using dynamic warp formation for improving control flow efficiency in a SIMD pipeline, we have also observed that it is possible to regroup threads, in a fashion similar to how we propose to do this for lane aware scheduling, to eliminate bank conflict in cache or shared memory access. The exact implementation and hardware cost of this idea should be explored in the future.

9.3.4 Improvements to GPGPU-Sim

While GPGPU-Sim was initially created solely to evaluate the performance of dynamic warp formation, we recognize that this simulator can be further improved in both accuracy and speed to prototype other architectural ideas as well. Improvements that we currently envision include:

- Exploration into more sophisticated DRAM scheduling policies: As mentioned in Chapter 5, DRAM request scheduling has significant impact on the overall system performance. Modeling various aspect of a massively parallel architecture, GPGPU-Sim is a suitable tool for exploring this design space.

- Direct interface to CUDA binary: Currently, GPGPU-Sim runs on benchmarks created with the SimpleScalar PISA GCC, which limits our choice of benchmarks. Having a direct

interface to CUDA binary will greatly broaden our choice benchmarks likely resulting in additional insights.

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of 10th Symposium on Principles of Programming Languages (POPL '83)*, pages 177–189, 1983.

[3] *ATI CTM Guide*. AMD, Inc., 1.01 edition, 2006.

[4] Arkaprava Basu, Nevin Kirman, Meyrem Kirman, Mainak Chaudhuri, and Jose Martinez. Scavenger: A new last level cache architecture with global block priority. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432, 2007.

[5] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31 (2):135–167, 1998.

[6] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[7] Jeffrey Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, 22(3): 917–924, 2003.

[8] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick. The Illiac IV System. *Proceedings of the IEEE*, 60(4):369–388, 1972.

[9] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH*, pages 777–786, 2004.

[10] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. http://www.simplescalar.com, 1997.

[11] E.A. Catmuli. Computer display of curved surfaces. In *Proceedings of Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pages 11–17, May 1975.

[12] Stefano Cervini. European Patent EP 1531391 A2: System and method for efficiently executing single program multiple data (SPMD) programs, May 2005.

[13] Tzi cker Chiueh. Multi-threaded vectorization. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 352–361, 1991.

[14] Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Kriszián Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proceedings of International Symposium on High Performance Computer Architecture*, pages 216–227, 2007.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[16] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[17] W. J. Dally and B. Towles. *Interconnection Networks*. Morgan Kaufmann, 2004.

[18] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and

Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *Proceedings of Supercomputing*, 2003.

[19] V.M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and Espasa E. Attila: a cycle-level execution-driven simulator for modern gpu architectures. *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 231–241, 19-21 March 2006.

[20] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec. 1966.

[21] H.W. Fowler, F.G. Fowler, and Della Thompson, editors. *The Concise Oxford Dictionary.* Oxford University Press, 9th edition, 1995.

[22] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2007.

[23] Jukka Helin. Performance analysis of the CM-2, a massively parallel SIMD computer. In *ICS '92: Proceedings of the 6th International Conference on Supercomputing*, pages 45–52, 1992.

[24] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann, May 2002.

[25] Tetsuya Higuchi, Tatsumi Furuya, Kenichi Handa, Naoto Takahashi, Hiroyasu Nishiyama, and Akio Kokubu. IXM2: a parallel associative processor. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 22–31, 1991.

[26] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. ClawHMMer: A Streaming HMMer-Search Implementation. In *Proceedings of Supercomputing*, page 11, 2005.

[27] Wen-Meii Hwu, Daviid Kiirk, Shane Ryoo, Chriistopher Rodriigues, John Stratton, and Kuangweii Huang. Model Performance Insights on Executing

Non-Graphics Applications on CUDA on the NVIDIA GeForce 8800 GTX. http://www.hotchips.org/archives/hc19/2_Mon/HC19.02/HC19.02.03.pdf, 2007.

[28] *Intel 965 Express Chipset Family and Intel G35 Express Chipset Graphics Controller Programmer's Reference Manual*. Intel Corporation, January 2008.

[29] Aggelos Ioannou and Manolis G. H. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking*, 15(2):450–461, 2007.

[30] Nuwan Jayasena, Mattan Erez, Jung Ho Ahn, and William J. Dally. Stream register files with indexed access. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 60, 2004.

[31] Harry F. Jordan. Performance measurements on HEP - a pipelined MIMD computer. In *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 207–212, 1983.

[32] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 159–170, 2000.

[33] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. The vector-thread architecture. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 52–63, 2004.

[34] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[35] James Laudon. Performance/watt: the new server focus. *SIGARCH Computer Architecture News*, 33(4):5–13, 2005.

[36] Hyunseok Lee, Trevor Mudge, and Chaitali Chakrabarti. Reducing idle mode power in software defined radio terminals. In *ISLPED '06: Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, pages 101–106, 2006.

[37] Adam Levinthal and Thomas Porter. Chap - a SIMD graphics processor. In *SIGGRAPH*, pages 77–82, 1984.

[38] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. SODA: A low-power architecture for software radio. In *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, 2006.

[39] Erik Lindholm and Stuart Oberman. The NVIDIA GeForce 8800 GPU. http://www.hotchips.org/archives/hc19/2_Mon/HC19.02/HC19.02.01.pdf, 2007.

[40] Erik Lindholm, Mark J. Kligard, and Henry P. Moreton. A user-programmable vertex engine. In *SIGGRAPH*, pages 149–158, 2001.

[41] R.F. Liu and K. Asanovic. Accelerating architectural exploration using canonical instruction segments. *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 13–24, 19-21 March 2006.

[42] Markus Lorenz, Peter Marwedel, Thorsten Dräger, Gerhard Fettweis, and Rainer Leupers. Compiler based exploration of DSP energy savings by SIMD operations. In *ASP-DAC '04: Proceedings of the 2004 Conference on Asia South Pacific Design Automation*, pages 838–841, 2004.

[43] Raymond A. Lorie and Hovey R. Strong. US Patent 4,435,758: Method for conditional branch execution in SIMD vector processors, 1984.

[44] IBM Ltd. Power ISA Version 2.05, 2007.

[45] David Luebke and Greg Humphreys. How GPUs work. *Computer*, 40(2):96–100, 2007.

[46] Michael Mantor. Radeon R600, a 2nd generation unified shader architecture. http://www.hotchips.org/archives/hc19/2_Mon/HC19.03/HC19.03.01.pdf, 2007.

[47] John Montrym and Henry Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, 2005.

[48] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38 (8):114–117, 1965.

[49] Simon Moy and Erik Lindholm. US Patent 6,947,047: Method and system for programmable pipelined graphics processing with branching instructions, 2005.

[50] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmanns, 1997.

[51] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160, 2007.

[52] Sam Naffziger, James Warnock, and Herbert Knapp. When Processors Hit the Power Wall (or "When the CPU hits the fan"). In *Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC)*, pages 16–17, February 2005.

[53] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a SIMdD DSP architecture. In *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 2–11, 2003.

[54] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of tow proteins. *Molecular Biology*, (48):443–453, 1970.

[55] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, 2006.

[56] John R. Nickolls and Jochen Reusch. Autonomous SIMD flexibility in the MP-1 and MP-2. In *SPAA '93: Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 98–99, 1993.

[57] NVIDIA Corp. NVIDIA CUDA SDK code samples. http://www.nvidia.com/object/cuda_get_samples.html.

[58] *NVIDIA CUDA Programming Guide*. NVIDIA Corporation, 1.0 edition, 2007.

[59] *NVIDIA CUDA Programming Guide*. NVIDIA Corporation, 1.0 edition, 2007.

[60] *Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview*. NVIDIA Corporation, November 2006.

[61] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH*, pages 703–712, 2002.

[62] Qimonda. 512-Mbit GDDR3 Graphics RAM, Revision 1.3 (Part No. HYB18H512321BF-08). http://www.qimonda.com, December 2007.

[63] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of 31st International Symposium on Microarchitecture*, pages 3–13, 1998.

[64] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, 2000.

[65] Eric Rotenberg, Quinn Jacobson, and James E. Smith. A study of control independence in superscalar processors. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 115–124, 1999.

[66] J. W. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 85–94, 2004.

[67] Mike Shebanow. ECE 498 AL : Programming massively parallel processors, lecture 12. http://courses.ece.uiuc.edu/ece498/al1/Archive/Spring2007, February 2007.

[68] Jaewook Shin, Mary Hall, and Jacqueline Chame. Introducing control flow into vectorized code. In *(to appear in) Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[69] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmarks. http://www.spec.org/cpu2006/.

[70] *OpenSPARC^{TM} T2 Core Microarchitecture Specification*. Sun Microsystems, Inc., 2007.

[71] Edwin J. Tan and Wendi B. Heinzelman. DSP architectures: past, present and futures. *SIGARCH Computer Architecture News*, 31(3):6–19, 2003.

[72] David Tarjan, Shyamkumar Thoziyor, and Norman P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett Packard Laboratories Palo Alto, June 2006.

[73] Shreekant (Ticky) Thakkar and Tom Huff. Internet streaming SIMD extensions. *Computer*, 32(12):26–34, 1999.

[74] Mark R. Thistle and Burton J. Smith. A processor architecture for Horizon. In *Proceedings of Supercomputing*, pages 35–41, 1988.

[75] James E. Thornton. Parallel operation in the control data 6600. In *AFIPS Proceedings of FJCC*, volume 26, pages 33–40, 1964.

[76] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[77] Steve Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Reading, Mass: Addison-Wesley, 1990.

[78] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. DRAMsim: a memory system simulator. *SIGARCH Computer Architecture News*, 33(4):100–107, 2005.

[79] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations.

In *Proceedings of 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.

[80] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Transactions on Graphics*, 24(3):434–444, 2005.

[81] M.T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34, 25-27 April 2007.

[82] Chuanjun Zhang. Reducing cache misses through programmable decoders. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–31, 2008.