Hint-Assisted Scheduling on Modern GPUs

A Dissertation Presented
by

Xun Gong

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

April 2020

Contents

List of Figures

List of Tables

Acknowledgments

I would like to dedicate this thesis to my parents. Without their unlimited support and encouragement during every single step of this journey, it would not have been possible for me to write this dissertation.

I am thankful to my colleagues at the NUCAR group, especially Leiming Yu, Xiang Gong, Zhongliang Chen, Shi Dong, Rafael Ubal Tena, Yifan Sun, and Julian Gutierrez. Thank you for all the wisdom you shared and the generous help you offered during my study. I would also like to thank my Ph.D. committee members, Dr. Ningfang Mi and Dr. Rafael Ubal, for providing me valuable ideas and constructive feedback on this dissertation.

Finally, I would like to express my deepest gratitude to my advisor, Dr. David Kaeli, for his constant support and guidance. His optimism and positive attitude always inspired me a lot and helped me overcome obstacles during my studies.

Abstract

Hint-Assisted Scheduling on Modern GPUs

by

Xun Gong

Doctor of Philosophy in Electrical and Computer Engineering

Northeastern University, April 2020

Dr. David Kaeli, Advisor

Graphics Processing Units (GPUs) have become an attractive platform for accelerating challenging applications on a range of platforms, from High-Performance Computing (HPC) to full-featured smartphones. They can overcome computational barriers in a wide range of data-parallel kernels. GPUs hide pipeline stalls and memory latency by utilizing efficient thread preemption. As throughput-oriented processors, GPUs provide tremendous processing power by executing a massive number of threads in parallel. However, long memory latencies and limited throughput are still major bottlenecks in GPU applications. Although the cache hierarchy helps to reduce the memory system pressure, the massive thread-level parallel in GPU applications often causes cache contention. Given the demands on the memory hierarchy due to the growth in the number of compute cores on-chip, hiding memory stalls has become increasingly challenging.

Prior studies have explored a smarter warp scheduler to help GPU achieve peak performance, alleviating the stalls caused by long memory latencies. Most of these efforts have explored hardware-based warp schedulers solutions. hardware approach. In this thesis we take a different approach, developing a hardware/software co-designed warp scheduling strategy. Our scheduler selects independent instructions that are in the shadow of a long latency memory operation and executes them out-of-order when all warps are stalled due to memory system saturation. By introducing software-based scheduling hints provided by the programmer or compiler, our scheduler will make smarter scheduling decisions that will improve GPU performance.

To properly study the benefits of our novel hint-assisted warp scheduler, we need an accurate GPU simulation framework that supports GPU timing simulation. We have develop a cycle-accurate simulation framework that supports native execution of CUDA programs at the SASS (the GPU binary) level for the NVIDIA Kepler architecture. We have also developed a customized CUDA driver and runtime APIs, as well as GPU drivers to support simulation.

Chapter 1

Introduction

Over the past four decades, successive generations of microprocessors have been able to achieve faster clock speeds by exploiting Moore's Law, following the trend of CMOS scaling. Unfortunately, the top speed for today's microprocessors has peaked due to the exponential growth in power density of a single chip, commonly referred to as the *power wall*. Because of this trend, we have seen major shifts in processor architecture design over the past decade. Instead of continually scaling the clock speed of a single core central processing unit (CPU), multi-core architectures now dominate the commodity CPU market, dominated by ARM, Intel and AMD.

Given the rapid growth of scientific and engineering applications, where our inability to process massive datasets efficiently limits the rate of discovery, we need to move beyond current CPU-based platforms. Emerging applications include deep learning, autonomous navigation, computational chemistry, weather/climate modeling, and artificial intelligence for public good [43]. Faced with exponential growth in terms of the scale of the data to be processed, the performance gains of multi-core CPUs can not begin to meet the computational demands of these workloads.

Many researchers and industry leaders have proposed moving to heterogeneous system designs, where special-purpose accelerators, including graphic processing units (GPUs) and tensor processing units (TPUs) can work together with a multi-core CPU to offload compute intensive workload.

1.1 Growth of the GPU Market

GPUs have been used to accelerate a wide range of general-purpose applications. The range of applications include high performance computing (HPC) to full-feature smart phones. More than 128 systems appearing in the 2018 TOP500 of the fatest supercomputers in the world are equipped with AMD and NVIDIA GPUs as accelerators. GPUs have thousands of processing cores on chip, making them the platform of choice for accelerating dense matrix operations, neural network training and machine learning algorithms, as well as many other applications.

GPUs were originally designed render 3D graphics, though they have emerged to be both outstanding rendering engines for the gaming market, and high performance accelerators for a wide range of markets. Unlike many CPU applications, graphics workloads are much more sensitive to the overall throughput versus single-thread performance. As a result, GPUs are designed to be throughput sensitive, versus CPUs, which are latency sensitive.

To take full advantage of the massive parallel processing power of a GPU, Nvidia released its Compute Unified Device Architecture (CUDA) [48] development framework in 2007, the earliest widely used programming model for GPU compute applications. Two years later, Open Computing Language (OpenCL) [38] was developed and was widely supported by industry leaders including Apple, Intel and AMD. Compared to CUDA, a key feature of OpenCL is portability, supported by a variety of heterogeneous platforms such as digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and GPUs.

In addition to CUDA and OpenCL programming frameworks, additional APIs and programming models have been developed to support general purpose computing on GPUs (GPGPU). The DirectCompute API [41] was developed by Microsoft to support GPU compute applications on Microsoft Windows Vista, Windows 7 and later versions. DirectCompute is part of Microsoft DirectX APIs and was initially released with DirectX 11 [32]. Nvidia worked with The Portland Group (PGI) to develop a CUDA Fortran Compiler which supports Fortran compilation on Nvidia's CUDA-enabled GPUs. This support was released in 2010 and now supports Linux, Windows and MacOS. OpenACC [73] is a directive-based programming model developed by Cray, CAPS, PGI and Nvidia. It is designed to provide a simple yet powerful approach for parallel GPU compute programming without significant programming effort.

In addition, GPU-accelerated libraries have been developed to provide programmers with highly optimized implementations of commonly used algorithms to speed up their development. GPU-accelerated libraries for signal processing, linear algebra, image and video processing include: cuFFT, cuBLAS and cuRAND [44]. These libraries are widely used in compute-intensive applications supporting a acceleration across a range of fields including computational chemistry, modecular dynamics and signal processing.

Given the growing support for GPU compute, researchers have muliple choices on the best way to accelerate their application using a GPU. Figure 1.1 shows three commonly used methods to accelerate applications on a GPU. The first leverages accelerated libraries, commonly referred to

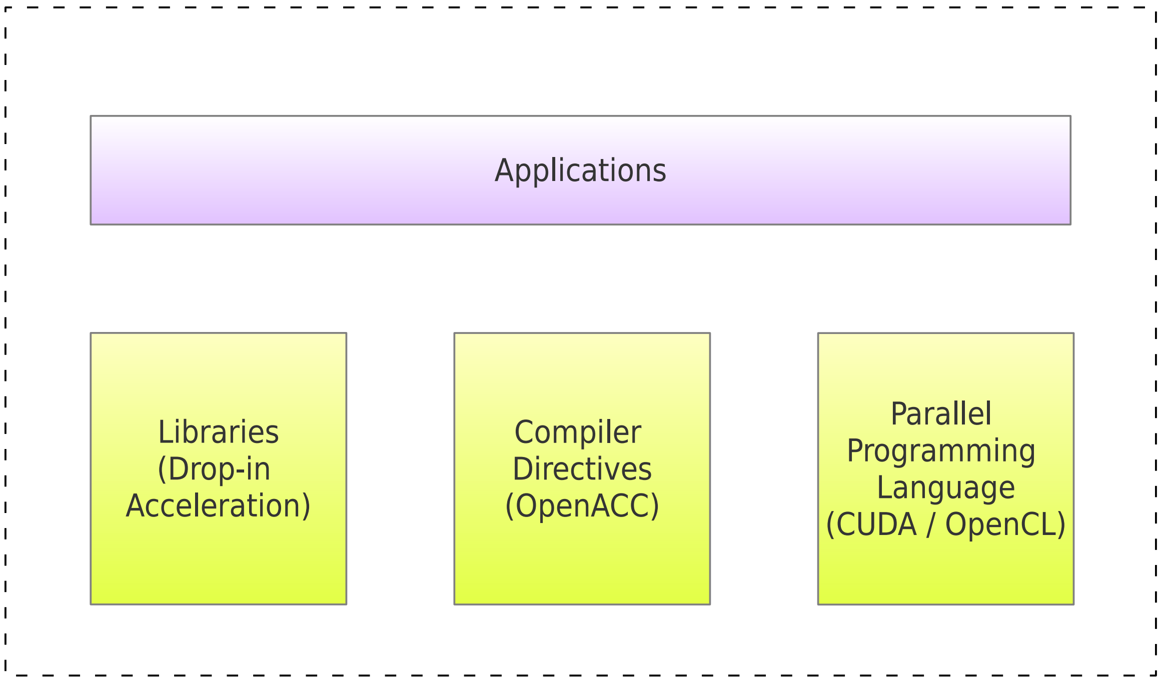

Applications
Libraries (Drop-in Acceleration)
Compiler Directives (OpenACC)
Parallel Programming Language (CUDA / OpenCL)

Figure 1.1: Different ways to accelerate demanding applications.

as *drop-in acceleration*, where programmers use libraries to execute parallel computing functions without in-depth knowledge of GPU programming[44].

Programmers can utilize compiler directives, a strategy supported on OpenACC [73]. This provides programmers with a easy way to develop applications that maximize performance and power efficiency benefits of heterogeneous system. Compared to parallel programming languages such as CUDA [48] and OpenCL [38], compiler directives are much easier for people to learn. Developers can write their algorithms sequentially and introduce directives to exploit any parallelism present in the algorithm. The programmer gives the compiler hints to turn on parallelism.

Parallel programming languages such as OpenCL [38] and CUDA [48], provide a third approach to accelerating applications with a GPU. Many times it can be challenging to implement a high-performance application in a parallel language. The major benefits of using OpenCL and CUDA is that, equipped with low-level APIs provided by GPU vendors, programmers have more control over the code and can maximize performance across different architectures.

1.2 GPU Computing Model

CPUs and GPUs represent two different design philosophies when it comes to architecture. CPUs are latency-oriented architectures that use large caches, branch prediction and complicated control logic to avoid stalls during execution. In contrast, GPUs are throughput-oriented systems that execute arithmetic operations and memory accesses concurrently on many data elements. GPUs do not provide complicated control flow logic. Instead, the GPU architecture is optimized to hide latency by providing massive parallelism and fast thread switching.

GPUs are designed to work as CPU co-processors to accelerate computation. The GPU is used to offload highly parallel and time consuming computation from the CPU, freeing up the CPU to process the remainder of the application. To achieve massive parallelism, GPUs adopt a Single Instruction Multiple Thread (SIMT) [31] programming model. To make best use of this programming model, today's GPUs are supported by a number of programming frameworks including OpenCL [38], CUDA [48] and HSA [27]. In this thesis, we will be targeting CUDA as run on an Nvidia GPU, so will use CUDA terminology.

A instances of a CUDA kernel is called a *thread*. Functions executed on a CUDA device are called kernels, which are organized in *thread blocks*. Multiple thread blocks are combined to for a *grid* of threads. When a kernel is called, N copies of the kernel are executed in parallel by N different CUDA threads. The threads executing the same instruction are grouped into a fixed sized batch, called a warp (NVIDIA) or a wavefront (AMD). All threads in the same warp share the same program counter and execute the same instruction.

1.3 Resource Underutilization

To achieve double-digit speedups on a variety of applications, GPUs exploit Thread-Level Parallelism (TLP). Parallelism must be exposed by the programmer using one of two different approaches. Recall that GPUs adopt a SIMT [31] execution model. A warp of 32 threads are grouped to execute together, similar to an OpenCL wavefront. They execute the same instruction on multiple execution lanes. Second, GPUs execute many warps concurrently on a single processing unit. When one warp is stalled, the warp can be preempted by the warp scheduler, allowing a new warp to execute. A large portion of GPU die area is dedicated to execution units to support a high degree of concurrency. Execution lanes on a GPU use in-order processing pipelines and do not perform branch prediction. A hardware scheduler can hide latencies associated a blocked thread by swapping threads. Threads can get blocked due to data dependencies and branch divergence. The impacts of these disruptions can be reduced or avoided by swapping computation to non-blocked warps.

However, even equipped with the mechanisms available that are designed to hide latencies, GPU resources remain frequently underutilized. For example, when a memory-intensive workload needs to load a block of data from off-chip DRAM, many threads compete for the avaiable memory bandwidth to DRAM, saturating the memory interconnect, stalling all warps until memory dependencies can be resolved. In this thesis, we propose and evaluate different techniques to utilize GPU resources more efficiently, especially for memory intensive workloads.

1.4 Motivation of this thesis

The warp scheduler on modern GPUs can swap warps that are stalled, when faced with a data dependency. However, one of the biggest challenges which prevents GPUs to achieve peak performance is the effective utilization of computation resources. When a memory intensive workload needs to load a block of data from off-chip DRAM, many threads compete for DRAM bandwith, saturating the memory interconnect. When we encounter this situation, a new memory request cannot be serviced until the older memory requests complete. Since memory requests are not pipelined, the warp scheduler can not hide most of the latency associated with these memory operations by swapping in runnable warps. As a result, warps will stall on instructions that depend on long latency memory instructions. The number of active warps will decrease as more warps hit the stall point. Eventually, all the warps will stall and wait until the long latency memory load is resolved.

Figure 1.2 shows the breakdown of scheduling cycles for 6 workloads. The portion of the stacked bars identified as Issued denotes that at least one warp in the GPU is eligible to be issued. The Long Latency Stall portion shows that all warps in GPU are stalled due to a long latency memory operation. Mixed Stall denotes that no warps can be issued in that cycle, which may be due to a variety of reasons (e.g., there is no available instruction in the fetch buffer, the pipeline is busy, a data dependency has occurred, etc.). In Figure 1.2, we can see that the fraction of long latency operation stalls in 4 out of 6 of our benchmarks is around, or more than, 25%. We classify these applications as memory intensive benchmarks. In three of these workloads, long latency stalls dominate over 30% of the execution. We classify the other two workloads as non-memory intensive, since less than 10% of their execution is dominated by long latency stalls.

For both memory intensive and non-memory intensive benchmarks, a large percentage of the total execution cycles (more than 50%) are spent waiting on different kinds of stalls. This trend is especially noticable in the memory intensive benchmarks, where the total stall cycles of some benchmarks consume more than 70% of the total cycles. How can we better utilize the GPU computing resources? We need to reduce the impact of long latency operation stall by leveraging intelligent scheduling.

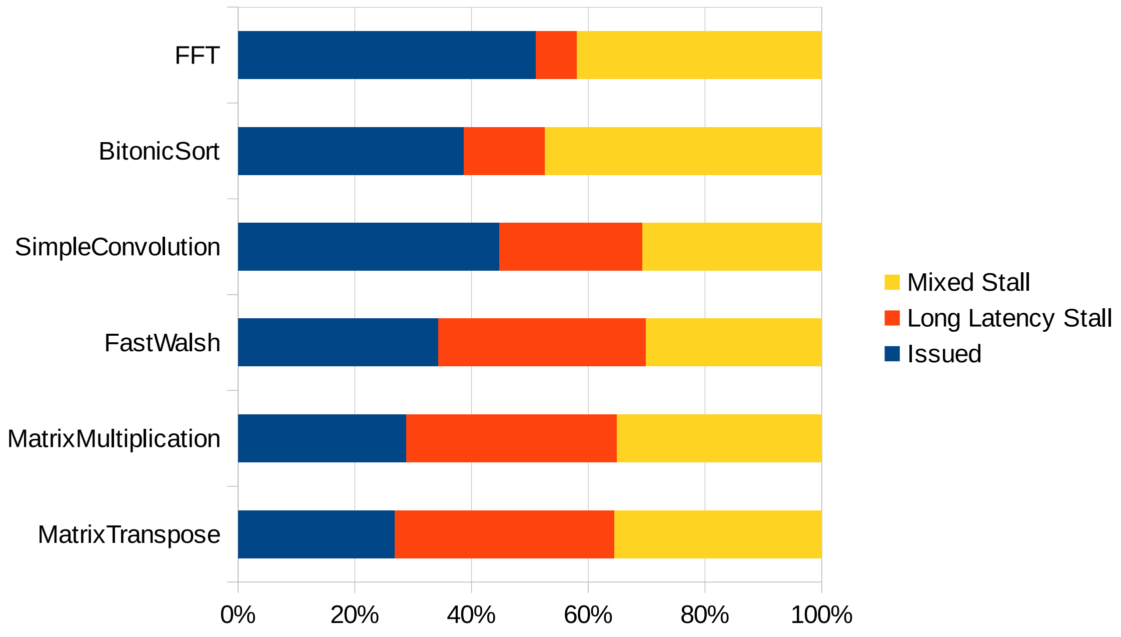


Figure 1.2: Breakdown of scheduling cycles.

The warp scheduler present in a GPUs Streaming Multiprocessor (SM) plays a pivotal role in achieving high performance for GPU applications, specifically by distributing hardware resources to different warps when the GPU stalls. Previous studies [28, 40, 37, 14, 30] developed different warp scheduling algorithms to improve GPU performance. However, all of these efforts focused on a pure hardware-based approach for the warp scheduler. Designing an effective warp scheduler in a hardware/software co-design approach has not been studied extensively on GPUs. In this thesis, we pursue an approach uses both hardware and software to increase GPU resource utilization and improve overall performance. We propose the following mechanisms:

1. Execute instructions in a selective out-of-order fashion in order to keep GPU computing resource busy whenever there is a long latency memory stall.

2. Assign thread blocks which have high spatial locality to execute on the same Streaming Multiprocessor - utilizing better warp scheduling to co-schedule warps that use the same data.

1.5 Challenges

1.5.1 Lack of GPU Simulation Framework

Modern computer architecture continue to grow in complexity, making them more challenging to evaluate. Simulation frameworks provide researchers and computer architects the ability to quantitatively evaluate architectural designs and evaluate design trade-offs. In general, architectural simulators [7] can be divided into two categories: i.) functional simulators and ii.) timing simulators. Functional simulators can accurately execute a program, simulating the functional aspects of the underlying architecture, but provide no timing information.

Timing simulators implement all of the hardware components and microarchitectural details, providing cycle-accurate timing information. Timing simulators capture detailed interactions between the compute pipelines and memory system. The typically take much longer time to execute as compared to a functional simulator, since a timing simulation must provide both functional accuracy, as well as timing accuracy.

To better analyze GPU performance and program behaviour, and to consider both hardware and software enhancements, a cycle-accurate GPU simulator is needed. However, only a limited number of open-source cycle-based GPU simulators have been developed in the academic research community. Prior to this thesis, there was no cycle-accurate open-source simulation framework that runs NVIDIA SASS-level (i.e., the executable binary level) executino. One key reason for this is that NVIDIA is highly proprietary about their instruction set architecture (ISA), driver information and architectural details. The lack of public information dramatically increases the burden on researchers that want to study GPU architecture. Developing such a simulator requires a significant investment in time and reverse-engineering to work out each instruction implementation, hardware timings, driver APIs and other hardware details.

This leads to our first challenge on developing a more advanced warp scheduling algorithm on GPU. Researchers cannot study GPU performance and analyze architectural details because of the lack of a simulation framework. To address this issue, in this thesis, we present Multi2Sim Kepler [16], a GPU simulation framework that support Nvidia SASS code execution. Equipped with this simulator, we can evaluate performance and tradeoffs when targeting the Nvidia Kepler architecture [46]. Multi2Sim [67] is a framework developed by Ubal et al., that can simulate both a number of CPU and GPU architectures. The work developed in this thesis supports a new ISA (Kepler) and a new programming framework (CUDA).

1.5.2 Lack of Control Logic

The performance of modern processors still suffers from the memory wall [35] and branch penalties. One way to alleviate this problem is to introduce instruction hints to increase the cache hit rate and decrease the branch misprediction rate. In some power-efficient processor models, architects elected to remove the hardware branch predictor and use a software branch hint to recover the lost performance. This was realized in the IBM's Cell Synergistic Processing Units [21]. When using software branch hints, hint instructions are inserted in the application, specifying that the branch instructions located at a specific PC address will jump to a specific target address. The processor will start to speculatively execute target instructions after executing a hint instruction, when the specified branch instruction is still in flight.

Compared with a CPU, a GPU is power-efficient and throughput-oriented processor. GPUs do not implement complicated control logic, such as branch predictor or reservation stations. Whenever a warp encounters a stall due to a data or control dependency, the warp has to wait until the dependency is resolved in order before continuing execution. Instructions cannot execute in out-of-order to bybass the stall and hide the performance penalty associated with the dependency. Since there are tens of thousands of threads running on the GPU simultaneous, it is impossible to add complex control logic while still maintaining the power efficiency on the GPU. This leads to our second challenge, GPUs lack sophisticated control logic to allow a warp to remain active when encountering long latency stalls.

To address this issue, our approach for GPUs is inspired by previous work applied to CPUs. We can introduce hints during complication to distribute thread blocks and schedule warps to better utilize GPU computing resources and improve application performance. We provide a compiler pass to analyze and mark the instructions that are independent of long latency memory instructions. These instructions are in the shadow of memory instructions. We design a compiler-generated hint that encodes this information and embeds it into the instruction stream. Based on these hints, the warp scheduler can choose to issue independent instructions and execute them out-of-order when all warps are stalled due to memory system saturation. The goal is to keep GPU resource utilization high, while allowing programs to execute past dependencies. Unlike CPUs, which use techniques such as reservation stations [66] and reorder buffers [26] to support out-of-order execution, it would be costly to replicate hardware components on a GPU, given that there could be hundreds of warps executing concurrently. Instead, Hint-Assisted Warp Scheduler (HAWS) uses hints provided by the compiler to schedule and execute instructions in a selective, out-of-order, fashion. HAWS executes non-speculative instructions and schedules warps based on the scheduling algorithm we choose (round-robin, GTO, etc.). When all warps stall because of long latency memory operations, HAWS triggers our hint-assisted scheduler, which will only fetch and execute the instructions that are independent from the long latency memory operation. We describe the details in Chapter 5.

1.5.3 Intensive Memory Contention in GPUs

As discussed earlier in this thesis, a GPU is only partially tolerant of memory delays. Whenever a warp generates a memory request, other warps also likely make memory requests in close time proximity. When thousands of memory requests compete for the limited memory bandwidth, this will increase the time required to fulfill those requests. GPU memory resources are not designed to handle this scale of parallelism in the memory system. This leads to our third challenge, addressing he need to achieve high resource utilization on a GPU.

To alleviate this problem, we have designed scheduling hints to exploit the spatial locality among different warps. For workloads such as matrix multiplication, which is a commonly used operation in machine learning and scientific applications, the memory reference pattern exhibits high spatial locality, especially across adjacent thread blocks and warps. We propose and design additional scheduling hints to provide detailed information which can be used by a warp scheduler to exploit any spatial locality inherent in a workload. For the adjacent thread blocks, the CTA scheduler will assign them to the same SM, versus distributing them to different SMs, in a round-robin fashion. For the warps which share high spatial locality, but are in different thread blocks, the warp scheduler will assign them to the same group to execute together according to the scheduling hints. This scheme can make better use of the available memory bandwidth. The details of this design are presented in Chapter 6.

1.6 Contributions of This Thesis

In this thesis, we pursue a hardware/software co-design approach to design a novelty warp scheduler to improve the utilization of GPU hardware resources and reduce memory contention. The key contributions are summarized below:

- We reverse engineer the Kepler GPU ISA, microarchitecture, memory hierarchy, as well as driver and runtime information.
- We identify and analyze the importance of running GPU simulation at the native SASS level and evaluate the difference between simulating at a higher intermediate language (PTX) level versus working at a SASS binary level.

- We develop a cycle-based GPU simulator based on the Nvidia Kepler architecture and demonstrate its accuracy as compared to execution on the real GPU hardware.
- Leveraging our simulator, we characterize a rich set of workloads to identify the sources of GPU resource underutilization.
- We implement a hinted-assisted warp scheduler in our simulation framework and show how it can improve GPU performance. We design a novel hint encoding format to embed the hint in the SASS instructions. We have proposed a novel warp scheduling algorithm named HAWS, which uses GPU resources more efficiently and can hide long latency memory operations.
- We analyze a broad set of benchmarks and find high degrees of spatial locality across adjacent thread blocks in some benchmarks. We classify these benchmarks and demonstrate we can better utilize spatial locality and reduce memory pressure using better thread block scheduling techniques.
- We develop a novel hint-assisted locality-aware warp scheduler to utilize this pattern and improve application performance. For different classifications of benchmarks, we design scheduling hints to group those warps and execute them together to fully enjoy the benefits of spatial locality and reduce memory bandwidth demands.

1.7 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 presents background information on general-purpose computing on GPUs, GPU architecture, and details on the GPU memory hierarchy and performance. Chapter 3 presents prior work that addresses how to hide main memory access latency to improve GPU application performance. We describe the details of our simulation framework in Chapter 4. The hint-assisted warp scheduler is described in Chapter 5. We present hint-assisted locality-aware scheduling in Chapter 6. Finally, in Chapter 7, we conclude the thesis and discuss directions for future work on this topic.

Chapter 2

Background

2.1 History of GPU computing

Computation on a GPU was first introduced by providing programmable shaders. In 2001, both OpenGL [74] and DirectX [32] added support for programmable shaders to give game developers and designers more space to create custom graphics effects. Graphics artists can now render their graphics by writing programs that execute directly on the GPU. Soon after, researchers discovered that GPUs are a good fit for many classes of data-parallel high performance computations in graphics applications, which motivated research into accelerating non-graphics applications on GPUs. At the beginning of GPU computing, a variety of high performance computing algorithms were ported from a CPU to a GPU, using high-level shading languages such as OpenGL and DirectX. Applications, including linear algebra, financial modeling, database queries, and image processing, have all achieved speedup when mapped to a GPU.

However, porting these general purpose applications from CPUs to GPUs, when leveraging a graphics programming language, can be very timing consuming and complicated for multiple reasons. First, computational algorithms need to be expressed in terms of textures, vertex coordinates, and shader programs, significantly increasing programming complexity. Second, programmers needed to have both a comprehensive knowledge of the GPU architecture and graphics APIs. Furthermore, the lack of double-precision and limitations on random memory access support restricted the type of applications that could be run on GPU.

To address these issues, GPU vendors started to explore general purpose computing on GPUs. CUDA [48] and OpenCL [38] were introduced by Nvidia and the Khronos Group in late 2000s. With the introduction of new general purpose programming models, programmers were no longer tied to using a graphics APIs, enabling them to focus on application performance and shorter development cycles.

Researchers from different domains started to use GPUs to accelerate applications. Medical imaging was one of the first applications to take advantage of a GPU's massive compute power. Mueller et.al [75] implemented cone-beam computed tomography(CBCT) algorithm in CUDA. They achieved a 500X speedup over the CPU implementation by using an NVIDIA GTX 8800 GPU. Sorensen et.al [64] demonstrated the use of a GPU in signal processing. They implemented a parallel Nonequi-spaced Fast Fourier Rransform (NFFT) on GPU, achieving a 85X speedup versus running on a state-of-the-art CPU. In bioinformatics, Schatz et.al [61] designed MUMmerGPU, an open-source high-throughput parallel pairwise local sequence alignment program. They achieved 10-fold speedup over a highly optimized serial CPU version of sequence alignment code.

2.2 GPU Evolution

As problem sizes have continued to grow, GPU vendors keep introducing more powerful architectures to fulfill the growing demands for both computation and energy efficiency. They integrate more transistors in a single die area to make GPUs have more computing power. Table 2.1 shows the parameters of the most recent five generations of NVIDIA GPUs. From Fermi [45] architecture to Volta [51] architecture, for a single GPU, we can see that the number of single-precision float point (FP32) CUDA cores has greatly increased, as well as double precision float point (FP64) CUDA cores, peak FP32 trillion floating-point operations per second (TFLOPS), and the total number of transistors. From Fermi to Kepler [46], the most significant upgrade we the number of FP32 CUDA cores per GPU, which is increased to 2880 [46], 6.4 times the amount in Fermi. The reason NVIDIA has managed to integrate so many cores onto a single die is that Kepler is the first chip produced on a smaller 28 nm process as compared to the 40 nm process in Fermi. Maxwell [47] adopted the same 28 nm process as did Kepler, so there is a limited increase in FP32 CUDA cores per GPU from Kepler to Maxwell. From Maxwell to Volta, the number of total transistors per GPU grew. The total number of transistors in Pascal [50] is almost double that in Maxwell. In Volta, this number is increased by 40% compared to Pascal, reaching 21.1 billion per GPU. The number of cores in the GV100 [51] almost doubles that in the GK180. As shown in the table, the peak FP32 TFLOPS for the GV100 is 2.6 times as high as for the GK180.

In the Fermi architecture, each SM contains 32 FP32 CUDA cores [45], which is 4X over the previous generation GT200. There are two warp schedulers, and each of them feeding the dispatch unit. Fermi's dual warp scheduler selects instructions from two different warps each cycle and dispatches one instruction from each warp to a group of 16 FP32 CUDA cores, 16 load/store

Architecture	Fermi	Kepler	Maxwell	Pascal	Volta
Release year	2009	2012	2014	2016	2017
Model	GF100	GK180	GM200	GP100	GV100
FP32 CUDA Cores / GPU	448	2880	3072	3584	5120
FP64 CUDA Cores / GPU	N/A	960	96	1792	2560
Peak FP32 TFLOPS	1.03	6	6.8	10.6	15.7
Transistors (billion)	3	7.1	8	15.3	21.1
GPU Die Size (mm^2)	529	551	601	610	815

Table 2.1: Comparison between recent 5 different NVIDIA GPU architectures.

Architecture	Fermi	Kepler	Maxwell	Pascal	Volta
Release year	2009	2012	2014	2016	2017
FP32 CUDA Cores / SM	32	192	128	64	64
FP64 CUDA Cores / SM	NA	64	4	32	32
Tensore Cores / SM	NA	NA	NA	NA	8
Register File size / SM (KB)	128	256	256	256	256
Warp Schedulers / SM	2	4	4	4	4
Dispatch Units / SM	2	8	8	8	4
Load / Store Units / SM	16	32	32	32	32
Special Function Units / SM	4	32	32	32	4

Table 2.2: Comparison between the 5 most recent NVIDIA SM architectures.

units, or 4 special functional units (SFUs). It takes two consecutive cycles to issue each FP32 instruction since 32 threads are running on 16 FP32 CUDA cores. In Kepler, each SM has 192 FP32 CUDA cores and four warp schedulers, which allows four warps can be issued every cycle [46]. Each warp scheduler is grouped with two dispatch units, issuing two independent instructions from the same warp simultaneously. Unlike Fermi, each FP32 instruction is executed on 32 cores, so it only takes one cycle to issue. SMs in Kepler also features FP64 CUDA cores to improve double precision performance. In Maxwell, NVIDIA integrated several improvements to further reduce redundant scheduling decisions to enhance power efficiency. Compared to the 192 FP32 CUDA cores per SM (a non power of 2 organization design) in Kepler, each SM in the Maxwell features 128 FP32 CUDA cores. They are partitioned into four different 32 FP32 CUDA core processing blocks, each has dedicated scheduling resources including a warp scheduler and dispatch unit [47]. In Pascal, each SM is integrated with 64 FP32 CUDA cores, half the total number of FP32 CUDA cores as Maxwell. The number of FP64 CUDA cores in Pascal is increased to 32 per SM from 4 in Maxwell, significantly increasing the chip's ability to handle double-precision arithmetic, which is the core part of many HPC applications such as artificial intelligence, linear algebra, and computational chemistry. Similar to Pascal, the SM in Volta also features 64 FP32 CUDA cores and 32 FP64 CUDA cores.

However, SM in Volta adopts a different partitioning method to improve the overall performance and resource utilization. In Pascal, the SM is separated into two processing blocks while the SM in Volta is partitioned into four processing blocks. Another signficicant update in Volta is that each SM is equipped with eight tensor cores [51], where each can perform 64 floating point FMA operations per clock cycle. Tensor cores deliver up to 12X higher peak TFLOPS in FP32 operations and 6X higher peak TFLOPS in standard FP16 operations, as compared to Pascal P100.

2.3 SIMT Programming Model

The Single Instruction Multiple Thread (SIMT) programming model is the execution model used in modern GPUs and is supported by several programming frameworks including OpenCL [38], CUDA [48] and HSA [27]. In this thesis, we will be targeting CUDA as run on an NVIDIA GPU, so will use CUDA terminology.

Figure 2.1 provides an overview of the execution element hierarchy defined in CUDA. Functions executed on a CUDA device are called *kernels*, which specify a *grid*. When a kernel is called, N copies of the kernel are executed in parallel by N different CUDA threads. An instance of a CUDA kernel is called a *thread*. A *warp* of 32 threads is grouped to run together, similar to an OpenCL *wavefront*. All threads in the same warp share the same program counter and execute the same instruction.

A *cooperative thread array* (CTA) or thread block consists of several warps. All threads in the same CTA have two basic properties: i.) within a single CTA, threads can perform efficient synchronization operations, and ii.) threads within the same CTA can share data through a low-latency shared memory. All CTAs from the same grid share a common global memory.

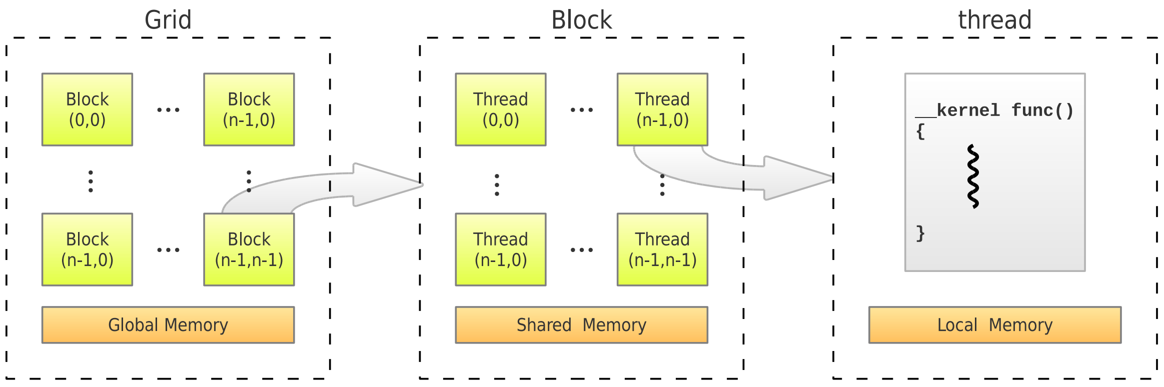


Figure 2.1: The CUDA Programming Model.

Figure 2.2 presents an overview of a typical GPU architecture. The GPU organization includes a giga-thread engine (thread block scheduler), a collection of Streaming Multiprocessors (SMs) and a memory hierarchy. The thread block scheduler processes the grid and maps waiting thread blocks onto the available SMs. Once a thread block is assigned to an SM, it remains in the SM until its execution completes. Resources, such as registers and shared memory, are not freed until the thread block finishes execution. Multiple thread blocks can be assigned to a single SM if there are available resources.

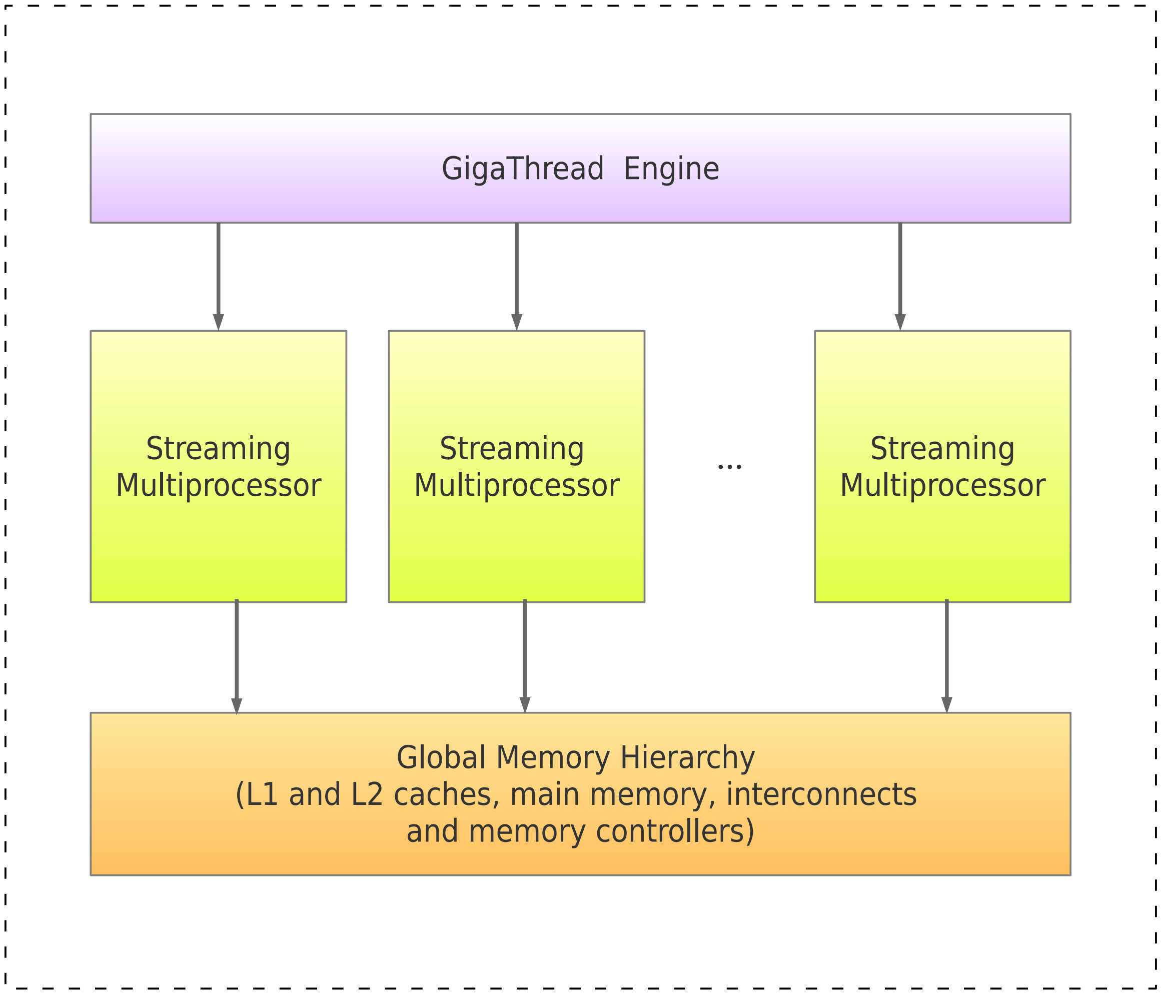


Figure 2.2: A typical GPU architecture.

CUDA defines the concept of a warp as a group of 32 threads executing in SIMT fashion. Each instruction is executed concurrently by every thread comprising a warp, although each thread has its data for computation. This model simplifies the instruction fetch hardware by implementing a common front-end for a whole warp. On every cycle, the SMs front-end fetches instructions from instruction memory for the different warps and sends them to the appropriate execution unit. For example, in the NVIDIA Kepler microarchitecture, each SM front-end features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. When a warp is stalled due to a long latency memory operation or other reasons, the warp scheduler can quickly switch and issue another warp to hide the latency.

As hundreds of thousands of threads are launched on GPUs, the scheduling of the threads can have a significant impact on the overall performance. To increase the performance, people need to consider how to distribute the thread blocks to SMs and schedule the warps to utilize the hardware resources better. An efficient warp scheduler can either improve resource utilization or hide the impact of long memory latency operations, keeping the GPU computing resources as busy as they can.

2.4 GPU Architecture

2.4.1 Steaming Multiprocessor Architecuture

A GPU is comprised of multiple SMs. During execution time, an SM can have one or more thread blocks allocated. These thread blocks are split into warps, which are assigned to SMs that execute threads. SMs are identical, with the major units and organization provided in described in Chapter 2.3. The warp scheduler is responsible for picking up available warps from the warp pool to execute every cycle. Since hundreds of warps are running on the same SM simultaneously, effective scheduling should try to hide memory latency, improving application performance. On every cycle, the SM front-end fetches instructions from instruction memory for the different warps and sends them to the processing cores. Each SM has tens or hundreds of processing cores (also called CUDA cores by NVIDIA). For example, in the Kepler architecture, there are 64 cores per SM. In the Maxwell architecture, there are 128 cores per SM. There are several types of processing cores, such as single-precision units (SPU/FP32), double-precision units (DPU/FP64), and SFUs. Each type of unit executes the corresponding arithmetic operations. The load/store units service all memory-related instructions.

2.4.2 GPU Memory Hierarchy

The GPU memory hierarchy is a bit complicated since it includes shared memory, texture memroy and constant memory, each with different characteristics including access latency, access scope and read-write access. By properly leveraging the GPU memory hierarchy, the programmer

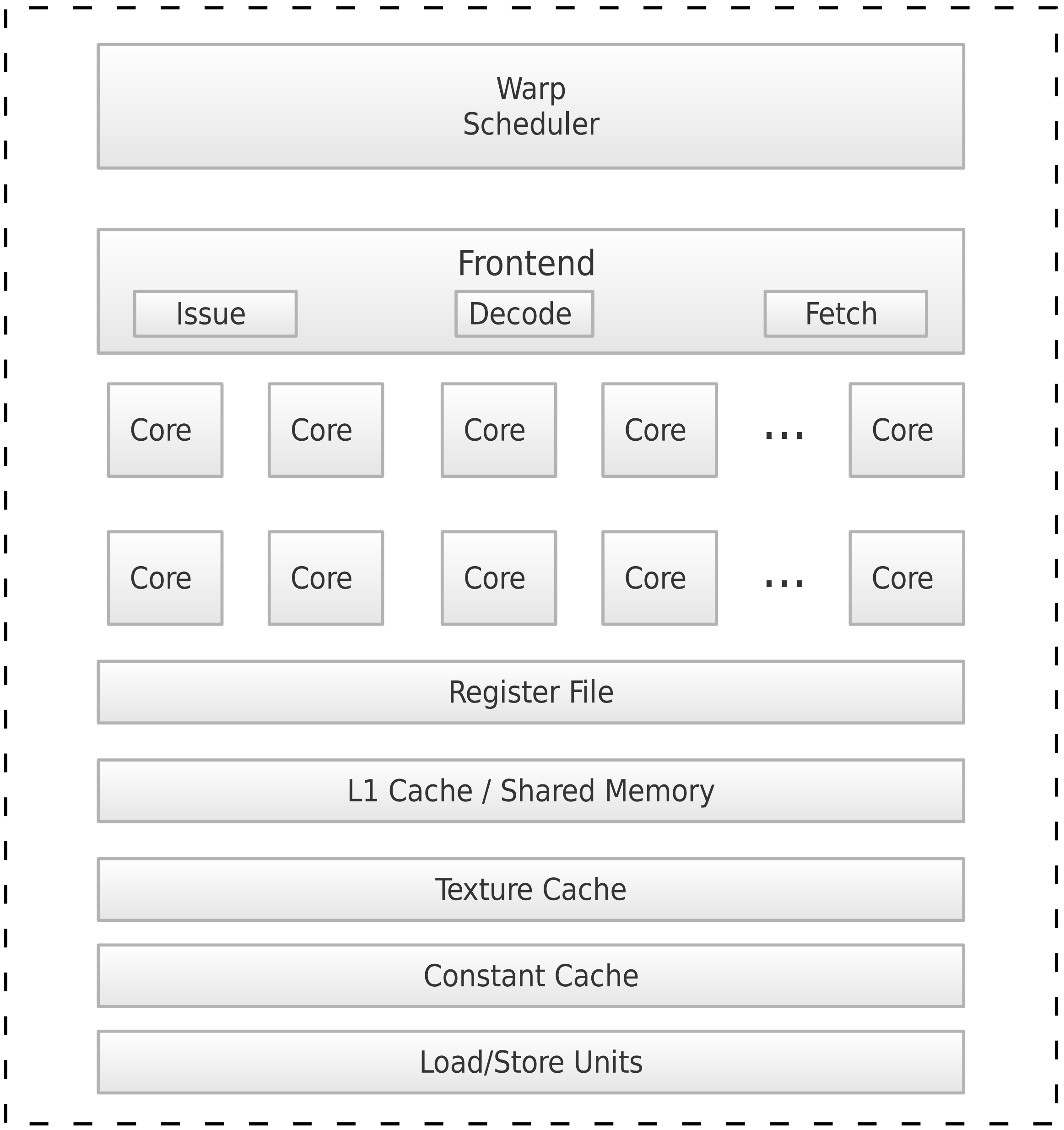


Figure 2.3: A typical SM architecture.

can reap better performance. It become critical for GPU programmers to understand the GPU's memory hierarchy in order to achieve better performance. Previous studies have explore how best to leverage the GPU's memory hierarchy [65], [60], [77], [8]. For example, on a GeForce 8800 GTX GPU, magnetic resonance imaging (MRI) can achieve 18.6 GFLOPS by using constant memory to reduce the average memory access time. Compared with the global memory version, utilizing constant memory can provide a 4X speedup [65]. Zhao et al. implemented G-BLASTN, which can achieve a 14.8x speedup, just by balancing shared and texture memory usage, as compared with the sequential NCBI-BLAST [77].

The on-chip GPU memory hierarchy is illustrated in Chapter 2.3. The hierarchy consists of a register file, L1 cache, shared memory, texture cache, and constant cache. The register file provides the fastest access in the memory hierarchy, which is private to each CUDA thread. In ithe Fermi architecture, the number of registers that can be accessed by a CUDA thread is 63 [45]. This number has been quadrupled in Kepler architecture, allowing each thread to access up to 255 registers [46] and is kept the same in the latest Volta architecture [51]. The number of registers used per thread is determined by the programmer and the compiler for GPUs. Since the total size of the register file is limited, the fewer registers each thread uses, the more threads can be launched simultaneously in a single SM.

Shared memory is another critical part of the GPU memory hierarchy. Since it is on-chip memory, shared memory is much faster and has much lower access latency as compared to the global memory. Shared memory is allocated per thread block, so any data stored in shared memory is visible to all threads in the same block. In other words, a thread can access data in shared memory, which is loaded by other threads from global memory if they are in the same thread block. To achieve high memory bandwidth, shared memory is separated into memory bank, allowing simultaneous memory accesses across different banks. When there are multiple memory accesses mapped to the same bank from different threads, the accesses will be serialized due to contention in the memory bank. An exception is when all threads in a warp access the same shared memory address. In this case, a broadcast will occur instead of serialized access, avoiding the penalty of bank contention.

The number of banks in shared memory is 16 for devices whose compute capability is 1.x. In this case, a shared memory request from a warp will be split into two consecutive cycles. For the Fermi architecture, the number of banks in shared memory is 32, which allows a warp to access shared memory within a single cycle. Starting with the Kepler architecture, programmers are allowed to use the CUDA API to configure the bank size to avoid shared memory bank conflicts. Also starting with the Kepler architecture, the on-chip memory in the SM can be partitioned between L1 cache and shared memory. Programmers can use the CUDA API to configure the shared memory size to improve the application performance.

The constant cache is a read-only cache for the SM to store data that does not change during kernel execution. Constant cache can broadcast the required data when all the threads in a warp access the same address, which reduces the required memory bandwidth significantly. Similar to constant cache, texture cache is also a read-only cache. NVIDIA initially designed texture memory for graphics rendering pipelines. It is beneficial when the memory access pattern in the kernel exhibits high spatial locality. The data in texture memory is cached in texture cache, which can reduce a lot of memory traffic and improve the overall performance.

2.5 Warp Scheduling

2.5.1 Scheduling Algorithms

Scheduling techniques play a vital role in improving GPU performance. For many GPU applications, hardware resources are not efficiently used, resulting in degradation of performance. One significant problem programmers try to address by using different scheduling techniques is to hide the impact of long latency memory instructions to improve overall performance. When there is a large number of memory requests over a short amount of time, memory bandwidth can become saturated, serializing subsequent accesses to memory. Additional warps will have to wait until the memory instructions are finished to resolve data dependencies, which results in low utilization of GPU computing resources.

Several previous studies focused on this problem by using different scheduling techniques. Narasiman et al. [40] introduced a two-level warp scheduler which separates warps into different groups, preventing them from reaching the same long latency instruction at the same time. This design allows the streaming multiprocessor to hide long latency operations by switching between different groups while ensuring memory locality within the same group. Jog et al. [19] presented OWL, a CTA-aware scheduler that uses data locality information to limit the number of CTAs in each SM and reduce cache contention.

Unlike previous work that leveraged pure hardware GPU scheduling techniques, in this thesis, we use a hardware/software co-design approach to deliver a novel warp scheduler which can improve GPU performance.

Chapter 3

Related work

3.1 Using Hinting in Microprocessors

Instruction hints have been present in modern microprocessors in previous designs. They have been used primarily for resolving branch mispredictions and cache misses in CPUs. The Intel Itanium 2 [36] ISA defined hint instructions that are used to provide the hardware early information about a future branch, and also direct the instruction prefetch engine to prefetch one or many L2 cache lines to increase cache hit rates.

Another popular example is the Synergistic Processing Unit (SPU) in the IBM Cell processor [21]. Jointly developed by Sony, Toshiba, and IBM, architects decided to remove the hardware branch predictor and use the software branch hinting to recover lost performance. Compared to the Intel Itanium, the Cell SPUs do not have any hardware branch predictor and rely solely on software branch hints to make the design more power efficient.

In the EPIC (Explicitly Parallel Instruction Computing) [62] ISA family, introduced by Hewlett Packard and Intel, the compiler performs direct cache placement and actively manages replacement policies through cache hints. The cache hints allow the compiler to have more control on cache behavior. The compiler can schedule instructions explicitly in parallel to hide long latency loads based on the true latency of instructions informed by the cache hints. The target cache hints are used to indicate the cache level at which the data will be kept after the instruction is executed.

Beyls et al. [6] proposed to generate cache hints from a reuse distance metric. Since the reuse distance indicates the cache behavior, irrespective of the cache size and associativity, it can be used for making caching decisions for all levels of cache hierarchy. Both static and dynamic approaches were proposed in this prior work. The static approach uses profiling to assign a cache hint to a memory instruction statically. The dynamic approach is based on an analytical model used to select the most appropriate hint dynamically.

Wang et al. [72] developed an analytical model that predicts which data will be reused by using compiler hints. These hints are provided to improve replacement decisions in set-associative caches. Their work explores options for reducing cache misses by adopting compiler hints to improve replacement decisions.

On the GPU side, NVIDIA started to inject instruction hint starting from the Kepler architecture [46]. They insert instruction hint for every eight instructions to assist their hardware scheduler. But due to the proprietary nature of this information, we have not found documentation describing the details of these hints.

3.2 Warp Scheduling

Previous studies have shown that the warp scheduler plays an essential role in improving GPU performance. Lakshminarayana and Kim [28] evaluated different fetch and DRAM scheduling policies on the performance of GPU applications. They ran different combinations of fetch and DRAM scheduling policies on a series of benchmarks and showed that, for applications that execute symmetric instruction length per warp, a fairness based warp and DRAM access scheduling policy improves performance. For applications that have various instruction counts per warp, different applications show benefits with different combinations of fetch and DRAM policies.

Narasiman et al. [40] introduced a two-level warp scheduler which separates warps into different groups, preventing them from reaching the same long latency instruction at the same time. This design allows the SMs to hide long latency operations by switching between different groups, while ensuring memory locality within the same group. Compared with the baseline round-robin scheduling policy, they can achieve a 9.9% speedup on average.

Meng et al. [37] proposed Dynamic Warp Subdivision (DWS), which splits a warp into two warps in the presence of a branch divergence or memory divergence. This method allows an SM to interleave computation down different branch paths in order to hide memory latency. In addition, DWS allows the threads that hit in the cache to continue to execute aggressively, even if some of their peer wavefronts have encountered a miss (i.e., a memory divergence).

Gebhart et al. [14] demonstrated that a combination of two-level warp scheduling and register file caching techniques could provide power savings without losing performance for applications. Register file caching can replace accesses to the large main register file, using a smaller hardware structure, which saves energy. Combined with register file caching, a two-level warp scheduler can further reduce energy usage by limiting the temporary register cache resources to currently active groups of threads. They showed that by using a 6-entry per thread register file cache, they could achieve a 36% reduction in terms of register file energy.

Lee et al. [30] proposed Criticality-Aware Warp Scheduling (CAWS) for GPU workloads, implementing a warp scheduling algorithm based on warp criticality. They showed that for specific workloads, such as breadth-first-search, the overall performance of the application is limited by the execution time of critical warps (the warps have a longer execution time). They identified the sources of criticality in warps and designed CAWS to balance the execution time of different warps in the same thread block. CAWS assigns priority to the critical warps, given them more opportunities to execute as compared with regular warps, maximizing the hardware resource utilization and reducing the overall execution time.

Lee et al. [29] proposed an alternative thread block scheduling designed to maximize GPU resource utilization. There are two main techniques they introduced. The first one is Lazy CTA Scheduling (LCS), which reduces the number of thread blocks allocated to an SM dynamically, avoiding performance degradation due to resource contention. The second technique they proposed is Block CTA Scheduler (BCS) that schedules adjacent CTAs to the same SM to improve spatial locality and improve performance. According to their report, they achieved an average of 16% speedup over the baseline GTO scheduler by combining LCS and BCS.

Rogers et al. proposed Variable Warp Sizing(VWS) scheduling [57], using a smaller warp size to execute when control flow and memory divergence occurs, improving the performance of divergent applications. VWS evolves the GPU into a more suitable computing device for irregular applications by adopting smaller warps to increase thread-level parallelism (TLP) and SIMD efficiency. VWS also improves energy efficiency by grouping the small warps to execute together, trying to build a MIMD-like computing model, but with SIMD model efficiency. According to their report, VWS can achieve a 35% performance improvement on divergent workloads.

3.3 Exploiting Thread Level Parallelism (TLP) on GPUs

Many warp scheduler studies have focused on choosing the right amount of TLP for the memory sub-system to avoid over-saturation, and to reduce latency. The Cache-Conscious Wavefront Scheduler (CCWS) proposed by Rogers et al. [58] preserves intra-wavefront locality by using additional hardware to adjust the amount of TLP per SM, keeping the L1 data cache from thrashing and maintaining intra-wavefront locality. They also introduced static wavefront limiting (SWL), which allows the programmer to set a limit on the number of active wavefronts per compute unit, during kernel launch. Based on their study, CCWS can provide a 24% performance improvement over a round-robin scheduling policy across a set of cache-sensitive workloads.

Fung et al. [13] explored the impact of branch divergence on GPU performance and proposed dynamic warp formation (DWF). DWF attempts to mitigate control flow divergence by dynamically creating new warps when threads in the same warp that take different paths after branch instructions execution. They implemented the hardware required for DWF and achieved a 20.7% speedup on average, with an additional 4.7% area overhead.

Fung and Aamodt [12] proposed thread block compaction, a novel mechanism to improve TLP when divergence occurs. Threads within the same thread block are compacted into new warps when they encounter a divergent branch. The compacted warps synchronize again at the reconvergence point and resume in their original arrangements before divergence. Compared with their previous study on dynamic warp formation, thread block compaction ensures a sufficient number of threads are available at the divergent branch to be compacted into new warps. Their simulation results show they can obtain an overall 22% speedup over a per-warp reconvergence stack baseline when running divergent applications, with no additional performance penalty.

Vaidya et al. [68] proposed two compaction techniques: 1) Basic Cycle Compression (BCC), and 2) Swizzled Cycle Compression (SCC) which use idle SIMD lanes to execute consecutive instructions. BCC squeezes out cycles for other instructions in the execution pipeline when there are any a set of 4 aligned threads are inactive. SCC is an improvement over BCC. SCC combines the active unaligned threads to execute in consecutive lanes. Their evaluation results show that BCC and SCC can reduce execution cycles in divergent applications up to 42% (20% on average) on Intel Ivy Bridge GPUs.

Rhu et al. [56] introduced the Dual-Path execution model (DPE), a model which exploits intra-warp parallelism by interleaving the execution of different paths when divergence occurs in a warp. Unlike prior solutions to handling divergence, DPE does not require an extensive redesign of the microarchitectural components. Instead, DPE extends the stack to support two concurrent execution paths. Using a series of benchmarks, the authors found that dual-path execution either matches the performance of the baseline single-path stack architecture or outperforms single-path execution by 14.9% on average, and by over 30% in some cases.

Kayiran et al. [23] proposed a dynamic CTA scheduling technique (DYNCTA) that attempts to allocate the optimal number of CTAs per core based on application demands. Their work showed clearly that executing the maximum number of CTAs per core is not always the best solution to boost performance, due to high cache and memory contention. DYNCTA provides a 28% performance improvement, on average, as compared to the default CTA scheduler, where the maximum number of CTAs are executing on the SMs.

Yu et al. [76] presented a Stall-Aware Warp Scheduling (SAWS) policy, which dynamically optimizes the TLP according to pipeline stalls. SAWS can effectively improve pipeline efficiency by reducing structural hazards without introducing new data hazards. SAWS monitors the structural hazards for all executed instructions and adjusts the number of active warps dynamically according to the number of structural hazards. SAWS achieves a 14.7% performance improvement on average compared to round-robin scheduling.

Kloosterman et al. [25] proposed WarpPool to identify spatial locality between threads in multiple warps and merge their requests being sent to the L1 cache. This reduces the memory bandwidth usage and relieves the bottleneck between the load store unit and the L1 cache. WarpPool achieves a 38% speedup on memory throughput-limited kernels by increasing the throughput to the L1 by 8% and reducing the number of L1 misses by 23%.

Wang et al. [71] introduced an Occlusion Aware Warp Scheduler (OAWS) that focuses on TLP on memory resources. Their scheduler monitors and predicts memory resource usage, scheduling wavefronts that can be satisfied by the available memory resource. Both static and dynamic prediction methods were designed and implemented to predict the demand of MSHR entries of divergent memory instruction, maximizing the number of concurrent warps without memory occlusion. Based on their evaluation, OAWS can achieve up to a 73.1% performance gain compared to the GTO scheduler.

3.4 Improving Instruction Level Parallelism on GPUs

Instead of reducing latency, several techniques focus on improving the overlap of compute and memory operations to exploit ILP on GPUs. Very Long Instruction Word (VLIW) instruction set architectures are designed to exploit ILP by encoding multiple independent operations in a single VLIW instruction. VLIW was adopted on AMD's Evergreen GPU architecture [4]. However, VLIW suffers from limited parallelism opportunities, as it solely relies on static analysis.

Gong et al. [15] proposed the Twin Kernel Multiple Thread (TKMT) execution model, which takes advantage of pursuing multiple instruction schedules in the compiler to improve the overlap of compute and memory operations. TKMT better distributes memory requests by reordering instructions for some wavefronts at compile time. This can significantly reduce the pressure on memory bandwidth, using the limited L1 cache more efficiently. TKMT can achieve a 12% average speedup over the baseline SIMT implementation model on various benchmarks on AMD Radeon GPUs.

Kim et al. [24] proposed a warped pre-execution approach on GPUs. In this technique, wavefronts try to issue future instructions that are independent of the stalling instructions. Their method relies on hardware to dynamically detect such opportunities and may take several attempts before finding a proper candidate. In contrast to Kim et al.'s approach, we are leveraging static analysis at compile time, so our hinting approach is free from hardware dependency detection. In addition, guided by the information provided in hints, our approach can commit the correct program state more efficiently when wavefronts return to normal execution.

3.5 Improving GPU memory hierarchy utilization

Many researchers have proposed novel techniques to utilize the GPU cache and memory hierarchy more efficiently. Jog et al. [20] demonstrated that existing warp scheduling policies are unable to effectively incorporate data prefetching techniques due to scheduling consecutive warps. They proposed a prefetch-aware warp scheduling policy to schedule consecutive warps separately, which can work with a simple prefetcher nicely to tolerate memory latencies. They evaluated their design across a diverse set of applications and achieved a 25% speedup over a round-robin scheduler.

Rogers et al. proposed Divergence-aware warp scheduling (DAWS) [59], which introduces a divergence-based cache footprint predictor to maximize the utilization of intra-warp locality for the current active warps. The goal of DAWS is to keep the data that is reused by a warp maintained in cache so future accesses across loop iterations will result in a hit. According to their results, DAWS can achieve a 26% performance improvement over Cache Conscious Wavefront Scheduling [58].

Jog et al. [19] presented OWL, a CTA-aware scheduler that uses data locality information to limit the number of CTAs in each SM in order to reduce cache contention. OWL achieves these benefits by improving both the L1 cache hit rate and latency tolerance, improving DRAM bank parallelism and improving both DRAM row locality and cache hit rates.

MASCAR [63] introduces a bimodal warp scheduling scheme, along with a cache access re-execution system to increase overlap. Instead of distributing warps in a round-robin fashion, MASCAR prioritizes memory requests among warps when there is memory saturation detected to better overlap compute and memory instructions. Using the re-execution queue, they could better utilize cache locality and eliminate structural hazards when the memory subsystem becomes saturated. MASCAR achieves a 34% speedup over a baseline round-robin warp scheduler for memory intensive benchmarks.

Wang et al. [70] proposed a Divergence-Aware Cache (DaCache) management scheme that can orchestrate the warp scheduling policy and the L1 cache management together for GPUs. In DaCache, the insertion position in the cache of any incoming data block is decided by the current warp's priority. Blocks of the warp with higher priority can enjoy a longer lifetime in cache. DaCache also prioritizes coherent loads over divergent loads durnig insertion, better utilizing intra-warp locality. Their experiments demonstrate that DaCache achieves a 40.4% performance improvement over the baseline GPU.

Oh et.al [52] proposed Adaptive Prefetching and Scheduling (APRES) to improve GPU cache efficiency. They demonstrated certain static load instructions in GPU applications have very high locality. Additionally, for some load instructions that have no locality, the addresses are heavily strided. APRES tries to group all the warps which execute the same load instruction within a short time and schedules them to utilize locality better. In addition, APRES tracks the inter-warp striding behavior and issues prefetches for different warps which execute the same load instructions. For memory intensive benchmarks, APRES can achieve a 31.7% performance improvement as compared to a baseline GPU model.

3.6 GPU Simulation Frameworks

Given that a major contribution of this thesis is a GPU simulator, we will review prior GPU simulator efforts.

GPGPU-Sim [5] is a popular simulator that supports functional and cycle-level timing simulation based on the NVIDIA Fermi architecture. It has provided the GPU research community with important insights. It supports simulation on NVIDIA Parallel Thread Execution ISA (PTX) level. GPGPUSim also supports PTXPlus, which is an extended form of PTX, introduced in GPGPU-Sim 3.x. The developers claim it allows for a near one-to-one mapping of most GT200 (i.e., Fermi) SASS instructions to PTXPlus instructions. However, it only supports execution running at an intermediate language level and does not capture SASS execution.

Multi2Sim [67] [16] provides a rich simulation framework for heterogeneous computing. It includes models for superscalar, multithreaded and multicore CPUs, and manycore GPUs. It provides cycle-level simulation for a X86 CPU and GPU simulation for both AMD & NVIDIA GPU architectures. It supports the Evergreen and Southern Island GPU architectures from AMD. As part of this thesis, we have developed a version of Multi2Sim for the Kepler architecture. Compared to GPGPU-Sim, Multi2Sim supports the real GPU instruction set architecture for both the AMD and NVIDIA architectures.

Gem5-GPU [55] is a heterogeneous simulator that models tightly-integrated CPU-GPU systems. It builds upon Gem5, a modular full-system CPU simulator, and GPGPU-Sim. Besides, they add support for the state-of-the-art GCN3 ISA to Gem5s GPU compute model. For both NVIDIA and AMD architecture, they support simulation at the intermediate language level.

Barra [10] provides an ISA-level functional simulation for the NVIDIA Tesla GPU architecture. It runs CUDA executables without modification. Since NVIDIA's ISA documentation is not publicly available, similar to our approach, the user needs to reverse engineering the NVIDIA ISA. It can also perform parallel functional simulation using a multicore processor or SIMD instruction set. Unfortunately, we cannot produce accurate performance models for GPU architectures, lacking the support of cycle-level timing simulation.

Ocelot [11] is a widely used dynamic compilation framework, as well as a functional simulator that executes programs working at the PTX virtual ISA level. Taking NVIDIAs CUDA PTX code as input, the user can either emulate or dynamically translate instructions to multiple platforms such as multicore CPUs, NVIDIA GPUs, and AMD GPUs. Working off an existing compiler infrastructure, Ocelot supports GPU compiler research, including interfaces to provide an internal representation of the PTX programs in support of optimization passes for massively data parallel computing kernels.

Simulating a GPU is especially challenging compared to simulating a CPU. Unlike CPU ISAs, GPU ISAs are often proprietary and change frequently, which makes it more challenger for developers to deliver GPU simulation frameworks. To support simulation at the shader assembly level requires a lot of effort while reverse engineering the GPU's ISA, driver APIs, microarchitecture, etc.. Thus, in the previous work we reviewed, most of the previous GPU simulator only model execution at an intermediate language simulation. The research community lacks an open-source simulation framework that runs cycle-accurate detailed architectural simulation at the GPU SASS level (i.e., the executable binary level), especially for the NVIDIA architecture.

Chapter 4

Simulation Framework

4.1 Simulation Methodology

The development of our Kepler model in Multi2Sim follows three phases of development, as presented in Figure 4.1, with the steps flowing from left to right. To deliver a cycle-accurate simulator, development involves the design and implementation of three independent software modules: i.) a disassembler, ii.) a functional simulator and iii.) a detailed simulator.

A dissembler can be used standalone to disassemble or can drive later simulation stages (i.e., emulation or timing simulation). In the first case, the disassembler reads directly from a program binary generated by a compiler (i.e.,a CUDA application binary), and dumps a text-based output of all fragments of the instruction set architecture (ISA) code found in the file. In the second case, the disassembler reads from a binary buffer in memory and outputs machine instructions, one-by-one, in the form of custom data structures that break down each instruction into its various fields.

The purpose of the functional simulator, also called the emulator, is to reproduce the original behavior of the CUDA program, providing the illusion that it is running natively on an NVIDIA GPU. Functional simulation is especially handy for debugging programs, and gathering statistics about the general behavior of a workload.

The detailed simulator, interchangeably referred to as the timing or architectural simulator, is the software component that models hardware structures in the Kepler GPU, and keeps track of execution time. The modeled hardware includes Streaming Multiprocessors (SMs), functional units, pipeline stages, pipe registers, instruction queues, cache memories, and other features.

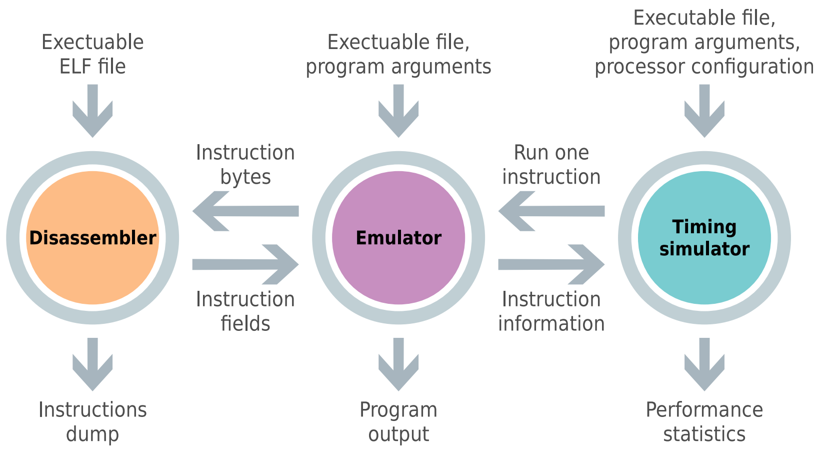


Figure 4.1: Kepler's simulation paradigm.

4.2 CUDA Support

The Multi2Sim Kepler model is a new GPU architecture extension to our existing Multi2Sim framework [67]. Multi2Sim is a free, open-source, cycle-accurate heterogeneous simulation framework that can model superscalar, multithreaded, and multicore x86 CPUs, ARM CPUs, MIPS CPUs, AMD GPUs, and now NVIDIA GPUs.

The current version of our Kepler functional simulator supports the execution of many different benchmarks without any porting effort by the user. It also supports the NVIDIA CUDA SDK benchmarks, as well as custom CUDA benchmarks we have developed. The architectural simulator models a detailed NVIDIA Kepler GPU, complete with an associated memory hierarchy, interconnection network, and additional components.

Multi2Sim supports truly heterogeneous execution. Every simulated program begins execution on the CPU. Multi2Sim Kepler supports unmodified NVIDIA CUDA execution on the Kepler GPU simulator. When CUDA programs are executed, the host (i.e., the CPU) portion of the program is run using the CPU simulator. When CUDA API calls are encountered, they are intercepted and transfer execution to the Kepler simulation model.

4.2.1 CUDA Programming Model

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. CUDA enables software developers to access the GPU's virtual instruction set and parallel computational elements directly for compute kernel execution. With millions of CUDA-enabled GPUs in use today, software developers, scientists, and researchers are finding new uses for GPU computing with CUDA [69].

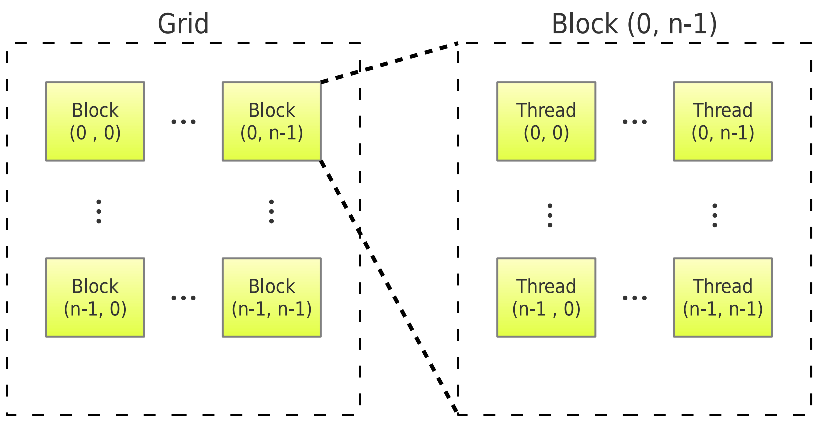


Figure 4.2: The CUDA programming model.

Figure 4.2 provides an overview of the execution element hierarchy defined in CUDA. CUDA C extends C by allowing the programmer to define C functions, called kernels. When a kernel is called, N copies of the kernel are executed in parallel by N different CUDA threads. An instance of a CUDA kernel is called a thread, which can access its pool of local memory. Within a grid, threads are grouped into blocks with two basic properties: i.) within a single block, threads can perform efficient synchronization operations, and ii.) threads within the same block can share data through a low-latency shared memory. All thread blocks form a grid and share a common global memory.

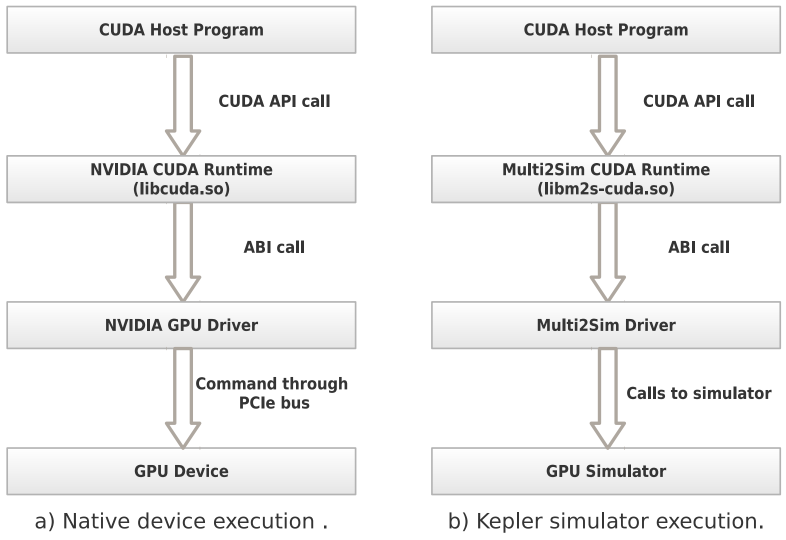


Figure 4.3: Comparison of native and simulated Kepler execution environments.

4.2.2 CUDA Simulation

The call stack of a CUDA program running on the Multi2Sim simulator differs from the native call stack, starting at the CUDA library call, as shown in Figure 4.3. We implemented a series of CUDA runtime [3] and driver [2] APIs, as well as a GPU device driver for the simulator. When a CUDA API function call is issued, our implementation of the CUDA runtime (libm2s-cuda.so) handles the request. This call is intercepted by the CPU simulation module, which transfers control to the GPU module as soon as the guest application launches the device kernel. This infrastructure allows unmodified x86 binaries (i.e., precompiled CUDA host programs) to run on Multi2Sim with total binary compatibility with the native environment.

4.3 Kepler Microarchitecture

Next, we provide an overview of our simulator infrastructure and describe the architecture of a generic NVIDIA Kepler GPU device [46]. We focus our discussion on hardware components devoted to the general purpose computing of CUDA kernels (versus the graphics pipeline). We provide insights into the GPU architectural details of the Kepler architecture [31], including the frontend, streaming multiprocessor design, and instruction pipeline. Note that many of these elements remain undocumented in publicly available information from NVIDIA.

4.3.1 The Kepler GPU Architecture

The GPU organization includes a giga-thread engine, a collection of SMs, and a memory hierarchy. The giga-thread engine processes the grid and maps waiting thread blocks onto the available SMs. Once a thread block is assigned to an SM, it remains in the SM until its execution completes. Resources, such as registers and shared memory, are not freed until the thread block finishes execution. Multiple thread blocks can be assigned to a single SM if there are available resources.

A detailed description of an SM is shown in Figure 4.4. An SM consists of a frontend, a register file, and some execution units. CUDA defines the concept of a warp as a group of 32 threads executing in Single Instruction Multiple Data (SIMD) fashion. Each instruction is executed concurrently by every thread comprising a warp, although each thread has its own data for computation. This model simplifies instruction fetch hardware by implementing a common frontend for a whole warp. On every cycle, the SM's frontend fetches instructions from instruction memory for the different warps and sends them to the appropriate execution unit. In the Kepler microarchitecture,

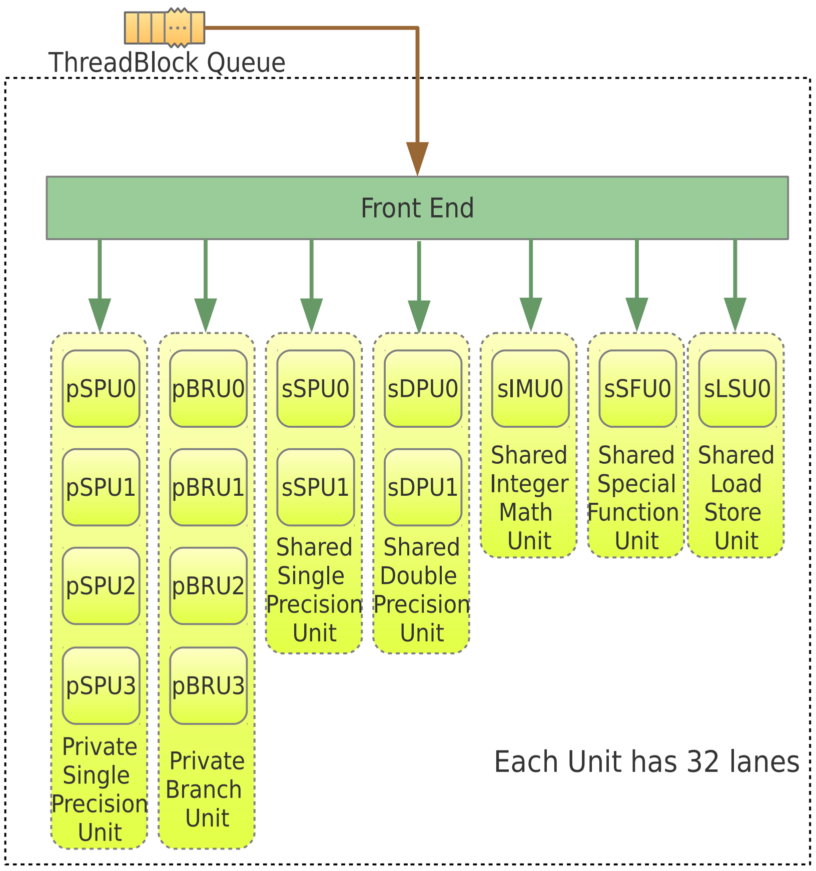
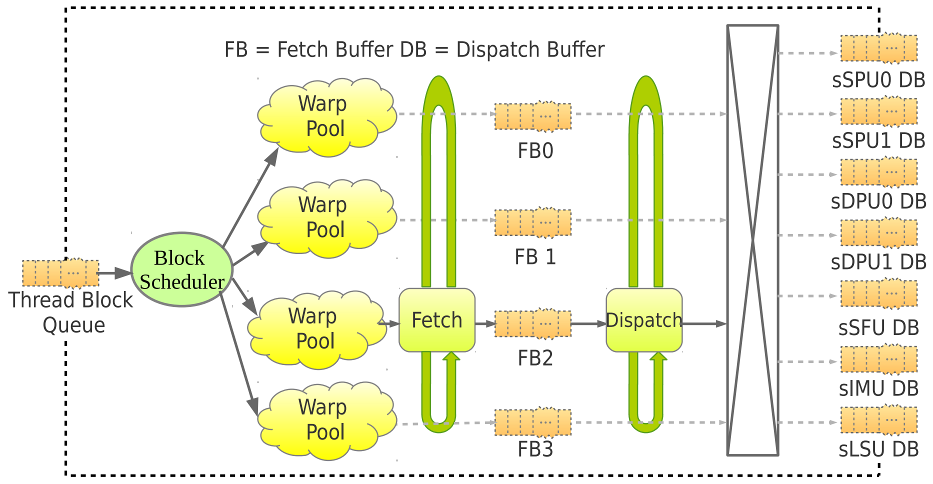


Figure 4.4: Streaming Multiprocessor architecture.

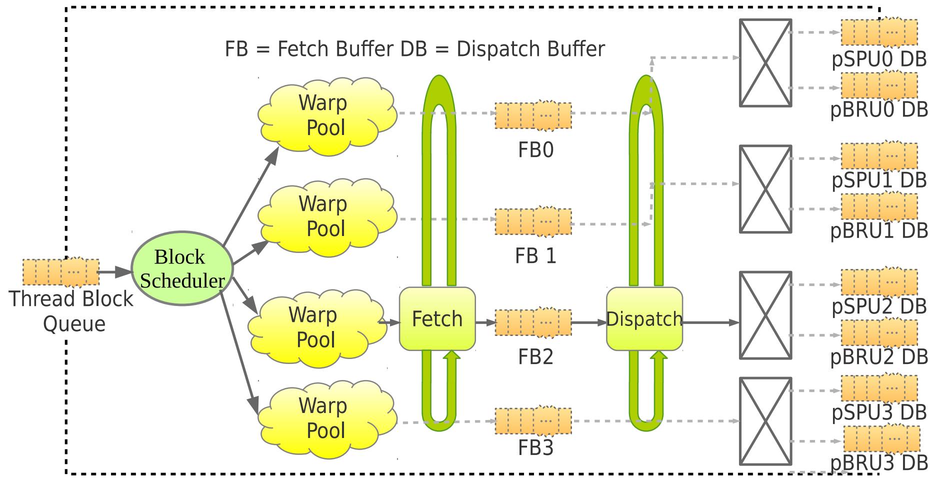
each SM frontend features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently.

The execution units present in an SM are the branch unit (BRU), the load store unit (LSU), the single-precision unit (SPU), the double-precision unit (DPU), the special function unit (SFU) and the integer math unit (IMU). Each unit consists of 32 lanes (32 CUDA cores in NVIDIA terminology), and each lane is devoted to the execution of one thread's arithmetic operation. This means that each unit can allow all 32 threads in a warp to execute the same instruction simultaneously. In Kepler, there are only two kinds of execution units. The first kind is the private unit, which is owned by each warp scheduler and can only execute SIMD instructions from the scheduler it belongs to (i.e., a private single-precision unit (pSPU) and a private branch unit (pBRU)). The other kind of unit is the shared unit, which is shared among the four warp schedulers (i.e., a shared single-precision unit (sSPU) and a shared integer math unit (sIMU)). A shared unit can execute SIMD instructions from either one of the four warp schedulers.

Figure 4.5 shows the architectural details of the SM front-end. It includes a set of warp



(a) The front-end pipeline for shared units.



(b) The front-end pipleline for private units.

Figure 4.5: A block diagram of the front-end pipeline.

pools, a fetch stage, a collection of fetch buffers, and a dispatch stage. The number of warp pools matches exactly the number of warp schedulers in an SM. When blocks are initially mapped to an SM, their associated warps are evenly distributed to the available warp pools. In the fetch stage the warps are selected from the pool, instructions are read from instruction memory using the warp's current program counter (PC), and these instructions are placed in the associated fetch buffer. In the next cycle, the fetch stage operates on the next warp in the warp pool, following a round-robin policy.

In our model, we fetch a maximum of 2 instructions per warp, per cycle, depending on resource availability. At the beginning of the dispatch stage, the recently fetched instructions are decoded and stored in their corresponding entries in the buffer. The warp scheduler can pick a warp to issue to the available warps. Before issuing the ready instructions to the execution units, the scoreboard performs dependency checks and generates a ready bit if the instruction passes the check. If the instruction is ready to dispatch, and there are still available resources, it is sent to the corresponding unit to execute.

4.3.2 Kepler Instruction Set Architecture (ISA)

NVIDIA has developed several major architectures, including the Fermi (SM 2.x) [45], Kepler (SM 3.x) [46], Maxwell (SM 5.x) [47] and Volta (SM 7.0)[51] architectures. For each new families, new instructions have been added as NVIDIA updated their architecture. For example, Kepler added a 64-bit funnel shift instruction that concatenates two 32-bit values together, shifts the resulting 64-bit value left or right, and then returns the most significant or least significant 32 bits.

```
/*00e0*/        PBK 0x190;              /*0130*/        FMUL R5, R3, R0;
/*00e8*/        ISETP.LT.AND P0, PT,    /*0138*/        SHL R3, R18, 0x2;
                R18, R17, PT;           /*0148*/        IADD R4, R2, R3;
/*00f0*/        PSETP.AND.AND P0, PT,   /*0150*/        MOV R3, R4;
                !P0, PT, PT;            /*0158*/        LD R3, [R3];
/*00f8*/    @P0 BRK;                    /*0160*/        FMUL R5, R3, R5;
/*0108*/        BRA 0x110;              /*0168*/        MOV R3, R4;
/*0110*/        SHL R3, R18, 0x2;       /*0170*/        ST [R3], R5;
/*0118*/        IADD R3, R16, R3;       /*0178*/        IADD R18, R18, R19;
/*0120*/        MOV R3, R3;             /*0188*/        BRA 0xe8;
/*0128*/        LD R3, [R3];
```

Figure 4.6: Sample SASS code.

When the GPU functional simulator receives the CUDA kernel to execute, an emulation loop starts executing Kepler instructions. The basic format of the NVIDIA Kepler ISA can be observed in the sample code in Figure 4.6. This block of native NVIDIA SASS code represents a *for-loop* in a CUDA kernel. It starts with a PBK instruction that specifies the target address to resume execution after the end of the loop, working with the BRK instruction to control the loop. The ISETP and PSETP instructions are used to calculate the predicate register value, deciding whether we are going to jump out from the loop.

4.3.2.1 Comparison between SASS and PTX

PTX [49] is a low-level parallel-thread-execution virtual machine and instruction set defined by NVIDIA. PTX is a high-level intermediate language as compared to native NVIDIA

SASS. There are two compilation stages required before a kernel written in CUDA can be executed on the GPU. The first stage compiles the CUDA code into the PTX virtual GPU ISA. The second stage compiles PTX into the actual SASS. The SASS code changes for each different generation of NVIDIA GPU, while the PTX code is architecture-independent.

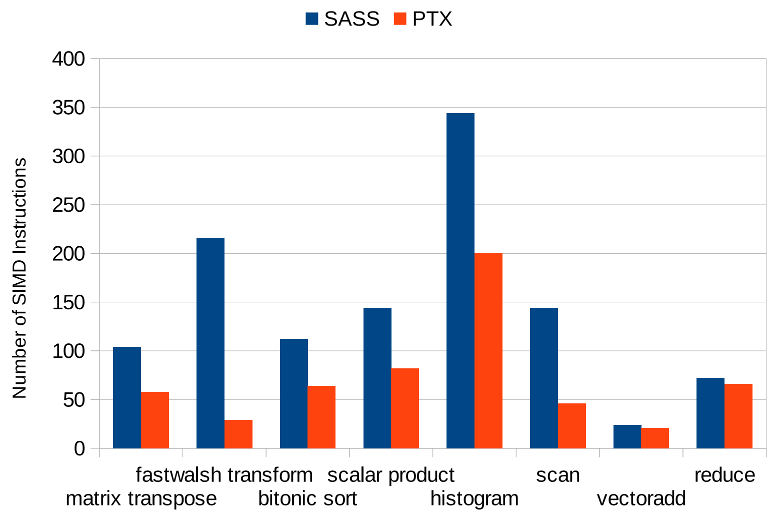


Figure 4.7: The number of SIMD instructions in SASS and PTX.

The number of SIMD instructions from our set of benchmarks is shown in Figure 4.7. In the kernels include here, the number of SIMD instructions in SASS is more than the number in PTX. Additionally, in some kernels, the number of SIMD instructions executed in SASS is eight times larger than in PTX (e.g., the Fast Walsh Transform). We analyzed the kernel and found that there is a reciprocal (RCP) operation in the code. When the real hardware executes the RCP operation, it requires many additional instructions to perform overflow protection, which makes the number of SIMD instructions in SASS much larger.

From our simple example above, we can observe a couple of important differences between working with SASS versus PTX. PTX instructions are not mapped to SASS one-to-one. In Kepler SASS, there are a limited number of general-purpose registers and a larger number of instructions, so there are more restrictions when the scheduler decides whether the instruction can be dispatched to the execution units. In general, there are many ISA-specific issues when we run SASS that need not be considered when working with an intermediate language. Thus, running SASS simulation is much closer to the actual GPU execution in those aspects.

4.3.2.2 CUDA Kernel Execution

When a CUDA kernel is launched by a host program, the kernel configuration is provided to the GPU. Thread blocks are then created and successively assigned to SMs when there are execution resources available. The following hardware constraints determine the number of thread blocks that can be assigned to a single SM: i) the maximum number of thread blocks supported per SM, ii) the maximum number of warps allowed per SM, iii) the number of registers on an SM, and iv) the amount of shared memory on an SM. Maximizing the number of assigned thread blocks per SM can have a tremendous impact on overall performance, which can be easily evaluated in our simulator.

Each block assigned to an SM is partitioned into warps, which are then placed into four warp pools equally. The warp scheduler selects warps from the warp pools for execution, based on a warp scheduling algorithm. The benefits of different warp scheduling algorithms can also be evaluated in our simulator.

4.3.2.3 Control flow and Branch Divergence

Branch divergence leads to performance degradation in the SIMD execution model when threads within a single warp take different paths. The Kepler ISA provides each warp with an active mask to resolve branch divergence present during SIMD instruction execution. The active mask is a 32-bit vector, where each bit indicates whether the corresponding thread is active (or not) in the warp. If a thread is labeled as inactive, the result of any operation performed in the associated lane is ignored, preventing the thread from changing the kernel state.

An active mask stack is used to support control flow instructions by pushing and popping active masks. When branch divergence happens, active masks, representing different paths, are pushed onto the stack, and the threads in the active mask, at the top of the stack, keep executing. The control paths taken by the threads in a warp are traversed one at a time until there are no more paths. Different execution paths are executed serially. All threads will re-converge when all the paths are finished, with the active masks representing the different paths being popped from the stack.

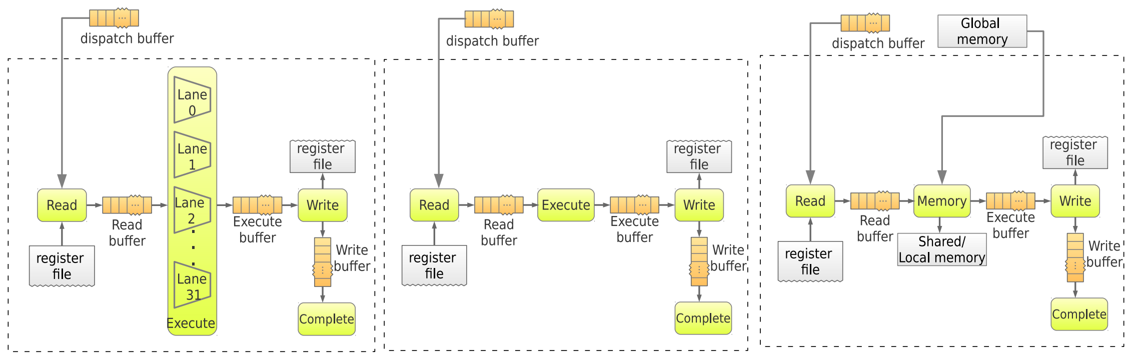
4.3.3 The Instruction Pipeline

In an SM, different kinds of instructions are sent to the appropriate execution unit by the dispatcher. Figure 4.8a presents a block diagram of the math instruction pipeline. Within each pipeline, decisions about latencies, scheduling policies, and buffer sizes must be made. All of these factors have performance implications and provide another opportunity for architecture researchers to experiment with design tradeoffs using our simulator.

The Kepler SM features an SPU (both shared and private), DPU, IMU, and SFU for arithmetic operations. Each unit has 32 lanes. Each SPU has fully pipelined floating-point and integer arithmetic logic units. The Kepler SM also uses the SFU for fast approximate transcendental operations, the IMU for complex integer operations (e.g., integer scalar adds (ISCADD) and bit field extractions (BFE)) and the DPU for double-precision operations.

After the fetch and dispatch stages, decoded instructions are placed into the dispatch buffer. The read stage consumes the instruction and reads the source operands from the register file for each thread in the warp. The execute stage issues an instance of the instruction to each lane in the unit on every cycle. The number of lanes in a unit in the Kepler matches the number of threads in a warp. The result of the computation is written back to the destination register in the write stage.

Similar pipelines are provided for the BRU and the LSU, as shown in Figures 4.8b and 4.8c, respectively. In the Kepler SM, the LSU handles all the load and store operations from the memory hierarchy, including global memory, shared memory and private memory operations. This design is different from AMD's, where shared memory operations and global memory operations are handled by different units.



(a) Block diagram of ALU pipeline. (b) Block diagram of BRU pipeline. (c) Block diagram of LSU pipeline.

Figure 4.8: Pipeline of different units.

4.3.4 The Memory Hierarchy

The GPU memory subsystem contains different regions for data storage (global, shared, local). In our Kepler simulator, we use Mult2Sim's existing memory model, which is highly configurable. The model includes customizable settings for the number of cache levels, memory capacities, block sizes, number of banks, and ports.

In our model, each thread has non-contended access to the register file. Multiple warps referencing the many units can access the register file, simultaneously, in a given cycle. A separate

Table 4.1: Baseline GPU configurations

device configuration		
compute resources	number of SM	14
	Warp size	32
	Number of SPUs / SM	6
	Number of BRUs / SM	4
	Number of DPUs / SM	2
	Number of IMUs / SM	1
	Number of SFUs / SM	1
	Number of LSUs / SM	1
	Number of lanes / Unit	32
Register file & Shared memory	Number of registers / SM	65536
	Shared memory / SM (KB)	16
L1 Cache	Associativity	4
	Block size	128B
	Total L1 cache size	16KB
L2 Cache	Associativity	16
	Block size	128B
	Total L2 cache size	1536KB
Others	Max warps / SM	64
	Max thread blocks / SM	16
	Max threads / SM	2048

shared memory module is present in each SM. GPU global memory is accessible by all SMs. In Multi2Sim, the global memory hierarchy has a configurable number of cache levels and interconnects. Each cache in the global memory hierarchy is connected to the lower-level cache (or global memory) using an interconnection network. Interconnects are organized as point-to-point connections using a switch, which contain two disjoint inner subnetworks, each devoted to package transfers in opposite directions.

Table 4.2: List of CUDA benchmarks used for evaluation

Benchmark	Input feature	Configuration
Bitonic Sort (BS)	Vector size	256 - 16k
Fast Walsh Transform (FW)	Array size	1k - 512k
Histogram (HM)	Array size	8k - 1M
Inline Math (IM)	Grid size	32k - 1280k
Matrix Multiplication (MM)	Matrix sizes	64 x 64 - 512 x 512
Matrix Transpose (MT)	Matrix size	128 x 128 - 1k x 1k
Scalar Production (SP)	Vector size	32k - 1M
Scan (SC)	Array size	4k - 1M
Vector Addition (VA)	Array size	64k - 1M

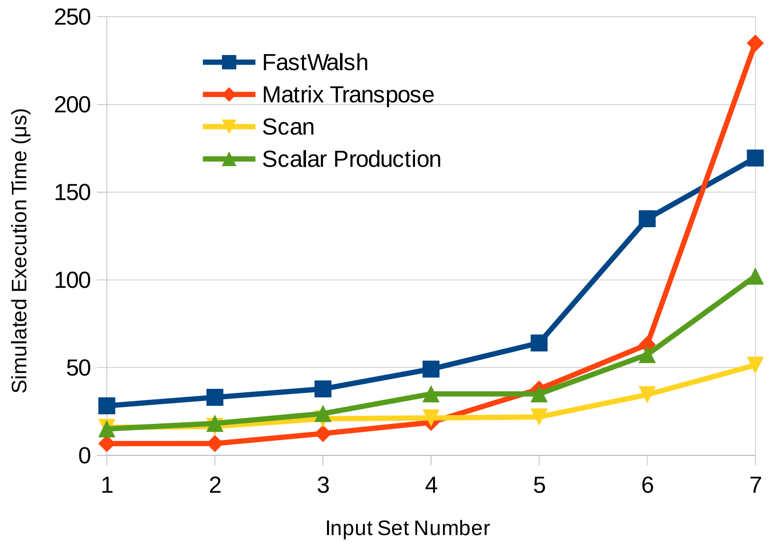
4.4 Evaluation

To demonstrate the utility of Multi2Sim Kepler, we present a set of experiments for validating and demonstrating both functional and architectural simulation features discussed in Chapter 4.3. All experiments are based on a baseline GPU model very similar to the NVIDIA Telsa K20X [1], with the hardware specification details summarized in Table 4.1.

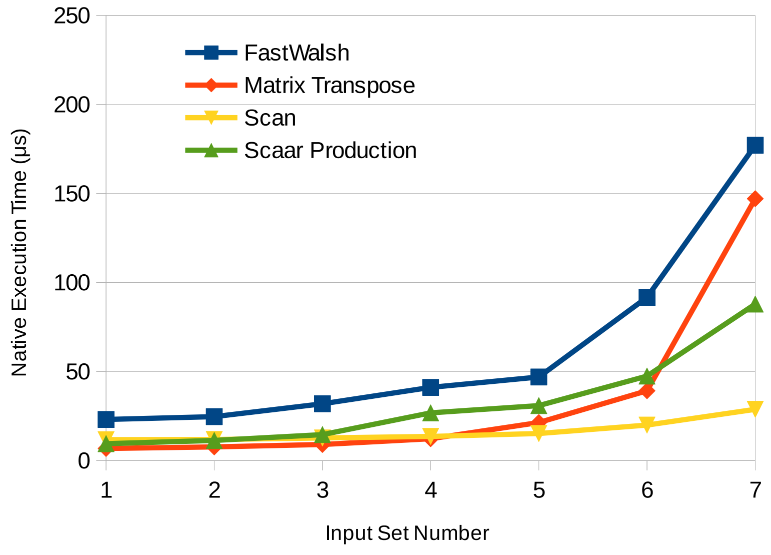
We elected to utilize a representative subset of the NVIDIA CUDA SDK benchmarks. The benchmarks included provide us with a rich set of compute-intensive and memory-intensive benchmarks, as well as a mix of both, representing a wide range of application behaviors. The benchmarks discussed in this thesis are listed in Table 2, along with some of their application properties. We include a description of the input datasets and a summary of the programs.

4.4.1 Baseline Model Validation

We present our simulator validation process in a number of steps. First, we start by validating our runtime library and device driver to make sure the device is properly initialized. Second, for the essential elements in our simulated architecture, we follow two different validation strategies for considering the correctness of both the functional and the architectural simulation models. For the functional simulator, we validate our instruction decoder by comparing the disassembled SASS code with the output generated by the NVIDIA disassembler. We also confirmed the correctness of each benchmark by comparing the simulated application output with the output of the application run directly on the K20X. All simulations generate correct results for all the benchmarks across a range of input sizes.



(a) Simulated execution time reported by Kepler Simulator



(b) Native execution time on NVIDIA Tesla K20X.

Figure 4.9: Validation for the architectural simulation, comparing trends between simulated and native execution times.

In terms of the precision of our architectural model, we compare our simulation results against the performance of the NVIDIA Tesla K20X hardware for the benchmarks and input sizes shown in Table 4.2. The native execution time is calculated as the average time of 1000 kernel executions for each benchmark while varying the input size. We measured the hardware execution time using the NVIDIA profiler, nvprof. Since our architectural simulator is a cycle accurate model and native execution performance on the GPU is measured by execution time, it is a bit challenging to compare the two results directly. In order to measure our architectural model, we use the K20X shader base clock frequency of 732 MHz documented by NVIDIA to turn Multi2Sim cycles into execution time.

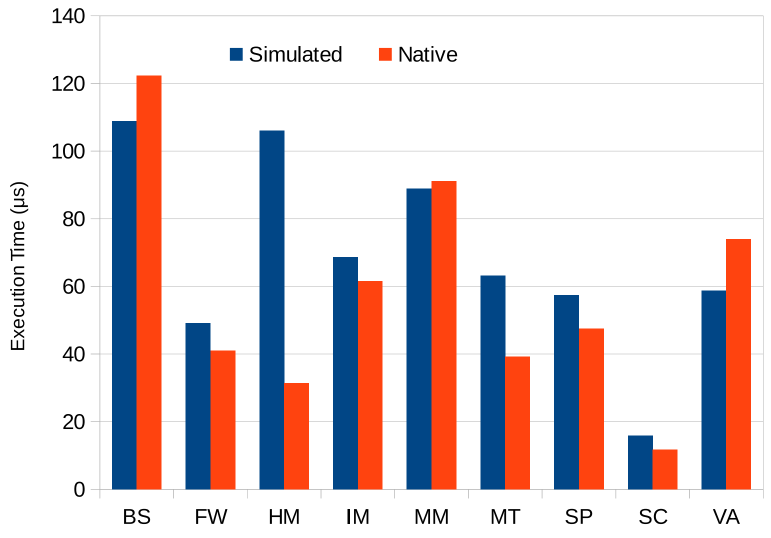
The differences in simulated and native performance are plotted in Figures 4.9a and 4.9b. In all of the benchmarks, the simulated execution time and the native GPU execution time exhibit the same trend, increasing the execution time that follows the growth in the input size. In other words, a change in the problem size for a benchmark has the same relative performance impact for both native and simulated execution.

Figure 4.10a plots the simulated and native execution time. In Figure 4.10b, we show a scatter plot of the normalized simulation execution time obtained for our simulator versus the normalized execution time, measured using NVIDIA K20X GPU hardware. The points that lie closest to the trend-line indicate that the simulated and native execution times track each other. For some of the benchmarks, execution times vary significantly. One outlier is the HM benchmark, as seen in Figure 4.10. The execution time produced by Multi2Sim is more than 2X the execution time on the K20X hardware. This outlier is due HM's heavy use of ST.WT and LD.CV instructions. These instructions change the cache policy and are not presently modeled faithfully in Multi2Sim's memory model. Besides this one source of inaccuracy (that is presently being updated), we have also identified the following sources of imprecision in our model:

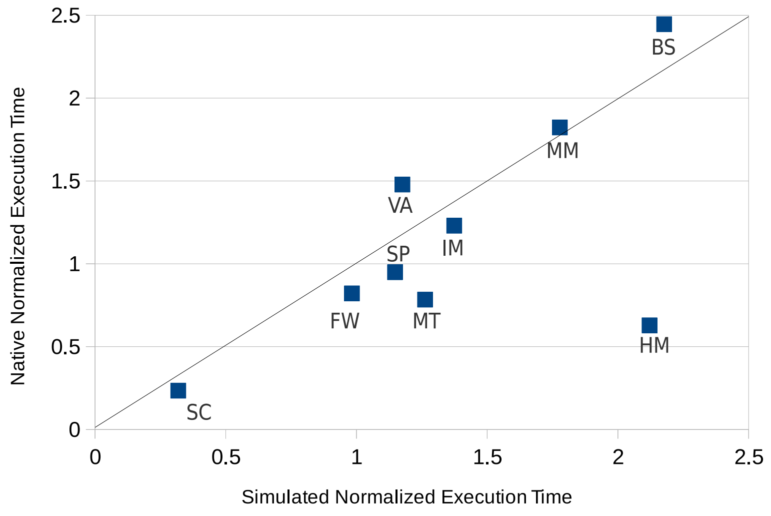
Cache Parameters & Interconnects. Some sources of simulation inaccuracy can be attributed to cache parameters, since the latencies of the different levels of the cache hierarchy have not been made public by NVIDIA. The specification of the interconnect network between the L1 and L2 caches has also been approximated due to a lack of public information. We continue to improve upon this model through reverse engineering.

Execution resource partitioning. As we discussed in the previous chapter, in a Kepler SM, there are shared execution resources (e.g., sSPUs, sIMU and sDPUs), which are shared across all four warp schedulers. But there is no documentation detailing how they share resources in each cycle. We provide a further discussion on this point below.

However, we still see a strong correlation between native execution and the associated simulator results for all benchmarks. In general, the benchmarks that perform well on the K20X hardware perform well when run on our simulator. The benchmarks that exhibit poor performance



(a) Comparision between simulated execution time and native execution time



(b) Normalized execution time.

Figure 4.10: Validation for the architectural simulation, comparing simulated and native execution times

also have poor performance on our simulator.

4.4.2 Effect of Execution Resource Partitioning

On the GPU, several execution units are shared across all four warp schedulers in each SM. These include DPUs, SFU (each unit has 32 lanes which can execute 32 threads simultaneously). In

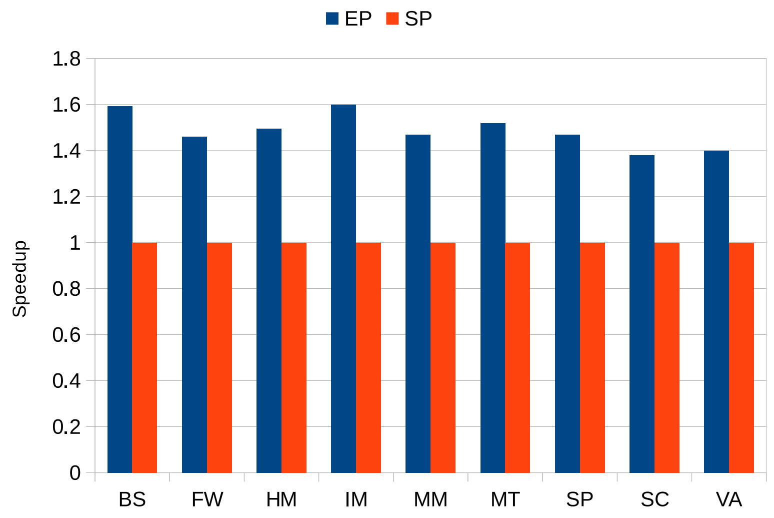


Figure 4.11: Performance comparison between EP and SP.

this section, we explore the impact that different partitioning strategies have on performance. We study and compare two partitioning algorithms, one that performs even partitioning (EP), allowing each warp scheduler to have a quarter of each shared unit (8 lanes). It will take four consecutive cycles for a warp to execute all 32 threads if its SIMD instructions are dispatched only to its own subset of the shared units. The second partitioning scheme is static partitioning (SP). With this scheme, we send the SIMD instructions to the appropriate unit from the warp scheduler, whose index is the lowest among all qualified schedulers. In this case, it takes one cycle for a warp to execute all 32 threads when the SIMD instruction is sent to one of the shared units.

Figure 4.11 compares the performance between our two partitioning algorithms, EP and SP. All of the benchmarks experience a speedup when choosing EP over SP. Scan achieves the least speedup across all of the benchmarks, which is nearly 40%. The other benchmarks are even higher. Using our built-in performance metrics in Multi2Sim, we find that EP provides higher utilization of execution resources than SP. When choosing SP, the warps dispatched by warp scheduler 0 are always faster than the warps dispatched by the other schedulers. When all warps dispatched by warp scheduler 0 are finished, the warps from the other schedulers are still running, without using the private units belonging to warp scheduler 0. Furthermore, when there are only warps from the last warp scheduler left, all the private units belonging to the other three warp schedulers are freed, which results in a very low utilization of execution resources.

4.4.3 Benchmark Characterization

To further illustrate the power of our infrastructure, we present a characterization of the CUDA benchmarks used in this thesis. We characterize programs based on instruction mix, branch efficiency, and control flow efficiency. These statistics are dynamic in nature and are reported by our Kepler simulator as part of its simulation reports. The purpose here is two-fold: 1) explore the characteristics of the workloads that will be used to explore hinting, and 2) demonstrate the utility of our Kepler-based GPU simulator.

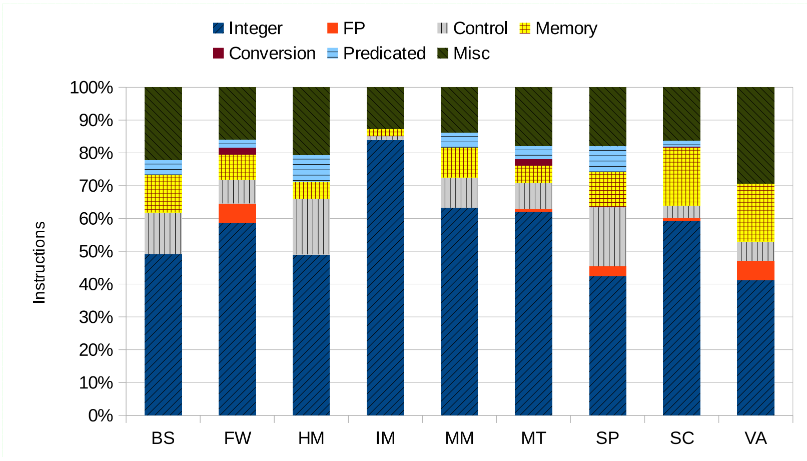
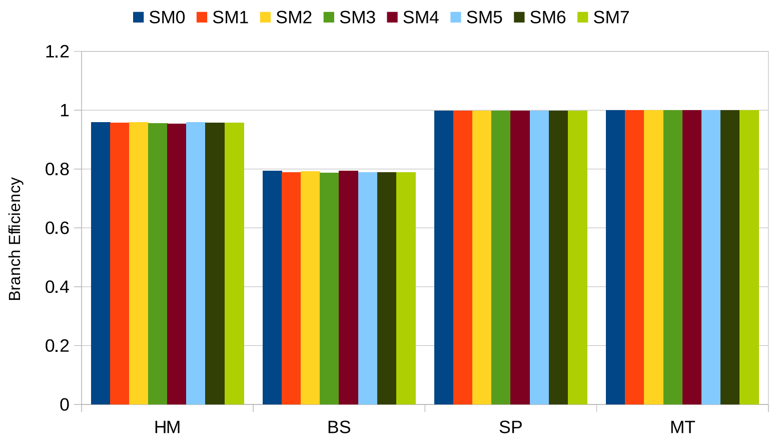


Figure 4.12: Instruction classification.

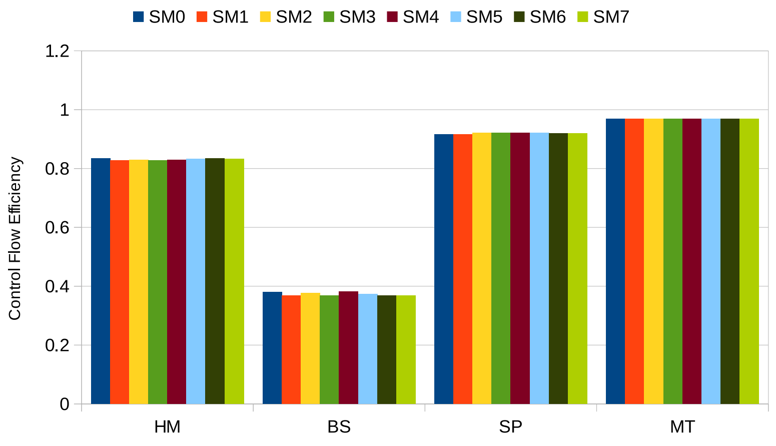
Figure 4.12 shows Kepler SASS instruction mixes for each benchmark. The instruction categories include: integer, floating point (FP), control flow instructions (branch, function call, jump and etc.), memory, data conversion (different data type conversions, such as integer to floating point), predicate (predicate register operations) and miscellaneous instructions (e.g., NOP, BAR, etc.).

Figure 4.13a shows the branch efficiency (the ratio of uniform control flow decisions over all executed branch instructions) per-SM (only 8 SMs are shown due to space limitation). This metric can be used to gauge the overall divergence rate. Different SMs have a similar branch efficiency in all of the benchmark tested because they did similar amounts of work. Figure 4.12 shows that MT, BS, HM and SP contain from 8 % to 18 % control flow instructions. However, control flow intensity does not necessarily translate into high branch divergence. It depends more on the sensitivity of individual thread execution in a warp. Although SP has the highest percentage of control flow instructions across the four benchmarks, only 0.1 % of them translate to a branch divergence (shown in Figure 4.13a). In contrast, BS has 12 % control flow instructions, but 21% of them result in divergent execution.

Branch efficiency is an interesting metric which can provide us with the divergence ratio of the branch instructions in a benchmark. However, this metric does not have any knowledge of the



(a) Branch efficiency for different SMs.



(b) Control efficiency for different SMs.

Figure 4.13: Branch efficiency and control flow efficiency.

extent of the code length generated by the divergence. This information is insufficient to debug the impact of a divergence with respect to kernel execution performance. For example, one divergent control flow decision executed by a branch may be negligible if it contains very few lines of code; but it may have a tremendous impact if the warp needs to execute many different code paths with thousands of instructions.

Figure 4.13b shows the control flow efficiency (the ratio of active threads, over the maximum number of threads per warp for each executed SIMD instruction) per-SM. It provides us with the efficiency when using the available execution units. This metric is a good measure that can identify the impact of divergence on the overall performance of a benchmark. In Figure 4.13b, BS has the lowest control flow efficiency in each SM, while less than 40% of instructions are executed

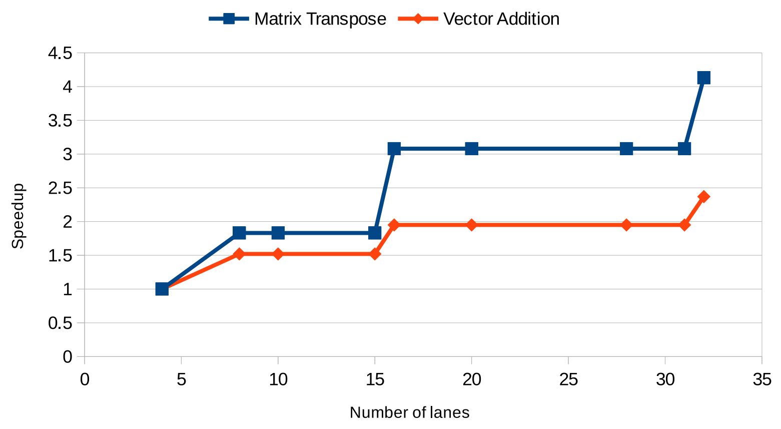


Figure 4.14: Scaling the number of lanes.

with full utilization of execution units, which means that each divergence path in the kernel contains a significant number of instructions. SP and MT have a similar branch efficiency, but MT has a higher control flow efficiency (97%), while SP is only slightly lower (91%). Clearly, each divergence in SP has a more severe impact on performance than seen in MT.

4.4.4 Architectural Exploration

The architectural simulator implemented in the Multi2Sim Kepler simulator provides researchers a flexible simulation framework that can evaluate a wide diversity of design points. To further illustrate the value of Multi2Sim, we present a case study to demonstrate the flexibility of the simulator, where performance significantly varies based on different hardware configurations. Performance is measured using the number of instructions per cycle (IPC).

Figure 4.14 shows performance achieved by modifying the number of lanes for each private SPU per SM. Our baseline model has four lanes in each SPU, and the other performance results are normalized to the baseline performance. In the two benchmarks evaluated, we observe an increase in performance when the number of lanes is a multiple of the warp size (32). This behavior is due to the number of lanes in each private SPU, which determines the number of sub-warps that the private SPU deals with for each warp. When an increase in the number of lanes causes a decrease in the number of sub-warps (e.g., 7 to 8, 15 to 16 and 31 to 32), performance improves. When the number of lanes in each private SPU matches the number of threads in a warp, bottlenecks due to stream serialized core utilization disappear. Compared to matrix transpose, vector addition is more memory intensive. We can see in Figure 4.14 that matrix transpose has about 5% memory instructions, while vector addition has 18%. This is why when we expand to 16 lanes, we see an extra performance boost in matrix transpose.

4.5 Summary

In this chapter, we presented Multi2Sim Kepler, a detailed performance simulator that simulates NVIDIA's shader assembly (SASS) code based on the NVIDIA Kepler architecture. To illustrate the utility of this framework, we provided several examples of architectural studies, exploring the micro-architecture of the SM front-end, SIMD instruction pipeline, and compute resource partitioning. We also discussed features of the NVIDIA SASS code and highlighted the differences between SASS and PTX through a series of benchmarks.

The framework has been developed through intensive analysis and reverse engineering of the Kepler hardware. We have developed a cycle-accurate architectural simulation at the ISA level. Based on this simulation framework, researchers will have a much clearer look into NVIDIA's architectural features when they evaluate their designs using our simulator. To the best of our knowledge, this is the first GPU simulator running on NVIDIA Kepler SASS.

Chapter 5

Hint-Assisted Warp Scheduling

HAWS selects the independent instructions that are in the shadow of a long latency memory operation and executes them out-of-order when all warps ware stalled due to memory system saturation. In this chapter, we discuss our hint format and describe the details of HAWS microarchitecture. We also evaluate our design and present simulation results.

Our preliminary work on HAWS that is presented here is developed on the AMD Southern Islands instruction set. This choice was made since our initial work on hinting depends on the Multi2C compiler, which only supports OpenCL compilation and generates binaries for the AMD Southern Islands ISA. To insure the work in this thesis is more cohensive, we also evaluate HAWS on our Kepler-based GPU simulator.

5.1 Selective Out-of-order Execution

Given that this thesis will explore using *hints* to optimize execution, we motivate this approach by first walking through a sample GPU execution flow using a simple code example. Figure 5.1 shows an instruction snippet for a matrix multiplication kernel, which is commonly used in a wide class of GPU applications, including machine learning [9]and image processing [53].

In the kernel, there is a loop that calculates each element in the output matrix. Every time the kernel calculates the pvalue in the loop, it needs to read data from both input matrices md and nd. The kernel also calculates the value of k during each iteration, whose value is independent of the values in md and rd. Inspecting the assembly code, the kernel reads a value from global memory from matrices md and nd, which are instructions 10 and 16 in assembly code listing. When the PC reaches instruction 17, which is a multiplication of md and nd, since R9 is a source operand whose values is generated by the instruction 16, the warp will stall due to the long latency memory operation. What

```
1  ISETP.LT.AND P0, PT, R7, R4, PT;
2  @P0 BRA I22;
3  IMUL R9, R5, R4;
4  IADD R9, R9, R7;
5  SHL R9, R9, 0x2;
6  IMUL R9, R5, R4;
7  IADD R9, R9, R7;
8  SHL R9, R9, 0x2;
9  IADD R9, R0, R9;
10 LD R10, [R9];
11 IMUL R9, R7, R4;
12 IADD R9, R9, R6;
13 SHL R9, R9, 0x2;
14 IADD R9, R2, R9;
15 MOV R9, R9;
16 LD R9, [R9];
17 FMUL R9, R10, R9;     RAW Stall
18 FADD R8, R8, R9;
19 IADD R7, R7, 0x1;
20 BRA I1;
21 IMUL R0, R5, R4;
```

```
{ int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
  int col = blockIdx.x * TILE_WIDTH + threadIdx.x;
  float pvalue = 0;
  for (int k = 0; k < width; ++k)
          pvalue += md[row * width +k] * nd[k * width + col];
  pd[row * width + col] = pvalue;
}
```

Figure 5.1: Assembly instructions from a GPU kernel.

is worse, as future warps stall at this point, memory contention will further increase the stall time. However, not all of the instructions after instruction 17 are dependent on instruction 16, so some instructions can still be fetched and executed if we selectively perform out-of-order execution. As we discussed earlier, for each iteration, the kernel calculates the value of k and compares it with the value of width. Both the values of k and width are independent of md and rd. Returning to the assembly code, it is clear that instruction 19, 20, and instruction 1-9, whose source operands are independent of register R9, which is the destination operand of instruction 16. So there are a significant number of instructions we can fetch and execute if we keep issuing instructions in a selective out-of-order fashion in this example.

Unlike GPUs, CPUs rely on extra hardware, such as reservation stations and reorder buffers, to support aggressive out of order execution and increase instruction level parallelism (ILP). The GPU's pipeline design tends to be overly conservative, providing for in-order execution. Our proposed hinting approach named HAWS modifies a GPU to incorporate some CPU-like features, our approach uses static analysis by the compiler to identify independent instructions, producing instruction hints to assist the warp scheduler. HAWS can support selective out of order execution by following the guidance provided by a hint. Unlike reservation stations, HAWS can increase GPU instruction level parallelism (ILP) by only adding a small amount of extra hardware. The compiler handles all data dependency analysis. Similar to IBMs Cell Synergistic Processing Units [21], HAWS does not significantly impact the GPUs power efficiency while increasing performance.

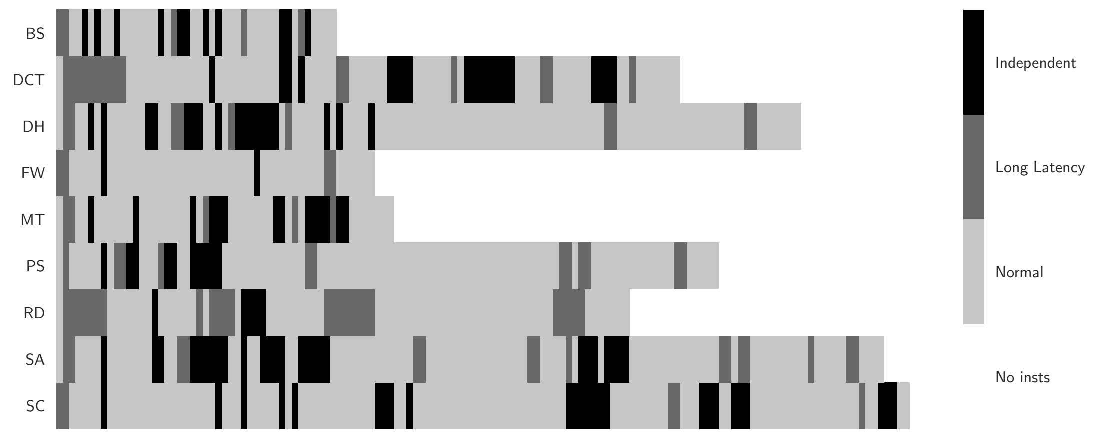


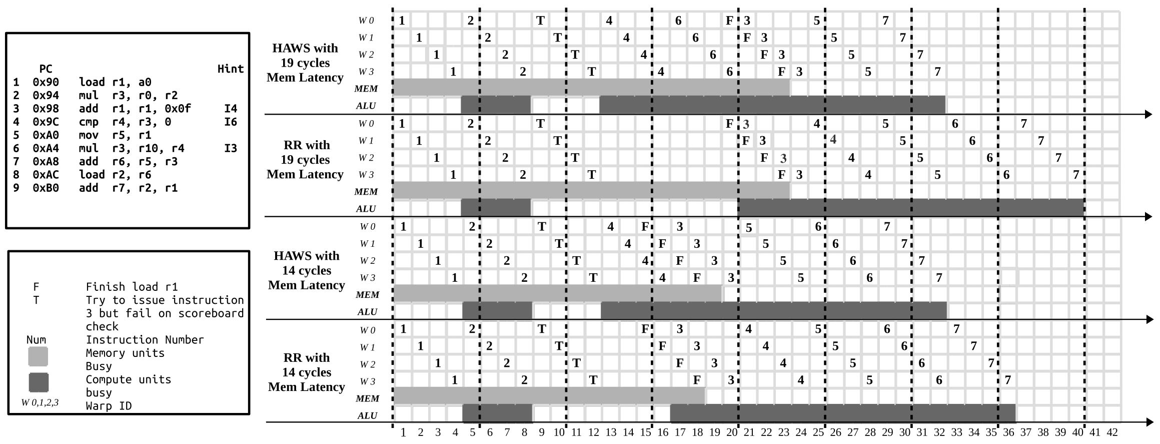
Figure 5.2: Instruction sequences produced for different benchmarks.

HAWS selects the instructions that are in the shadow of a long latency memory operation, and are independent of the memory operation. To quantify the number of independent instructions present in the program, and evaluate the potential benefits of HAWS, we analyze a series of benchmarks. In Figure 5.2, we show the instruction sequence for a kernel binary. The dark grey bars represent long latency memory operations, and the black bars represent the instructions that are independent of the previous memory instruction. We can see some black bars appear after the dark gray bars. This trend is present in all of our benchmarks, suggesting that there are many opportunities in GPU applications for HAWS to improve GPU memory performance.

5.2 Hint Assisted Execution

We begin by describing how we insert hints in GPU instructions. We then demonstrate the potential benefits of the presence of a hint on kernel execution performance using a detailed example. While we are targeting an AMD Southern Islands ISA, we will use NVIDIA terminology in this section to reduce the number of unique terms.

First, we define an instruction that has a dependency with a long latency memory operation as a *stall point*. In Figure 5.3, instructions 3 and 9 are stall points. We also define the instructions between any two stall points as a *hint group*. When a warp stalls due to encountering a long latency memory operation, the hint will assist the warp scheduler in fetching and issuing independent



```
     PC                                 Hint
1  0x90   load r1, a0
2  0x94   mul  r3, r0, r2
3  0x98   add  r1, r1, 0x0f              I4
4  0x9C   cmp  r4, r3, 0                 I6
5  0xA0   mov  r5, r1
6  0xA4   mul  r3, r10, r4               I3
7  0xA8   add  r6, r5, r3
8  0xAC   load r2, r6
9  0xB0   add  r7, r2, r1
```

F Finish load r1
T Try to issue instruction 3 but fail on scoreboard check
Num Instruction Number
Memory units Busy
Compute units busy
W 0,1,2,3 Warp ID
HAWS with 19 cycles Mem Latency
RR with 19 cycles Mem Latency
HAWS with 14 cycles Mem Latency
RR with 14 cycles Mem Latency
W 0
W 1
W 2
W 3
MEM
ALU

Figure 5.3: Execution timeline for round-robin scheduler and HAWS. The Top 2 are running with a 19 cycles memory latency. The bottom 2 are running with a 14 cycles memory latency.

instructions within a hint group, fetching as many as it can, until the scoreboard releases the long latency flag.

In Figure 5.3, we show the execution timeline for a GPU, guided by a round-robin scheduler and HAWS. For simplicity, we assume arithmetic instructions take one cycle to execute, memory instructions take 14 or 19 cycles in our example. There are four warps running in parallel, with only one of them being issued each cycle. Each warp will fetch a new instruction every cycle. We vary the latency of memory to better characterize dependencies in our experiments.

Experiment 1: Memory latency is 19 cycles: When the warp scheduler tries to issue instruction 3, from cycles 9 to 12, due to the dependency with instruction 1, the warp stalls in the GPU with a round-robin scheduler and the long latency operation flag in the scoreboard is set to 1. With HAWS, the warp will keep searching for independent instructions within the hint group. If one is found, the hint will offer to the warp scheduler the offset of the next independent instruction. HAWS decides to issue instruction 4 in cycle 13 and keeps executing, informed by the instruction hint. Warps that use round-robin scheduling still stall, due to long latency memory operations. After fetching instruction 4, HAWS reads the offset of the next instruction that should be fetched from the hint, which is instruction 6. After instruction 6 is finished, since there are no independent instructions we can issue, HAWS will go back to the original stall point, fetches instruction 3, and waits for the memory instructions to finish. The memory instructions are completed, starting in cycle 20. Both HAWS and the Round-Robin schedulers receive results from r1, and the scoreboard releases the long latency flag so that instruction 3 is issued successfully for both schedulers. Considering the execution latencies, since HAWS better utilizes the computational resources of the GPU by carrying out selective out-of-order execution in the shadow of a memory result, this work reduces execution time by 25% speedup.

Experiment 2: Memory latency is 14 cycles: In this case, there are limited benefits for HAWS, since the memory latency is reduced by five cycles. Just as in experiment 1, HAWS begins selective out-of-order execution in cycle 13, issuing instruction 4. Starting in cycle 15, the memory results are available, so the scoreboard is updated to reset the long latency operation flag. HAWS receives the flag and transitions the warp back to in-order execution by flushing the fetch buffer and re-fetching instruction 3. HAWS provides a dynamic decision. Whenever a long latency operation is resolved, HAWS returns to in-order execution. In other words, instructions in the warps are executing speculatively in terms of hint-assisted mode. That is why we see instruction 3 issued in cycle 17 for HAWS, and instead of instruction 6. More architectural details will be provided in the next section. In this case, since the memory latency is reduced, HAWS leads to a performance increase of 12.5%. Instruction hints provide detailed information to the HAWS scheduler to support the next hint.

5.3 Microarchitecture of HAWS

5.3.1 HINT Format

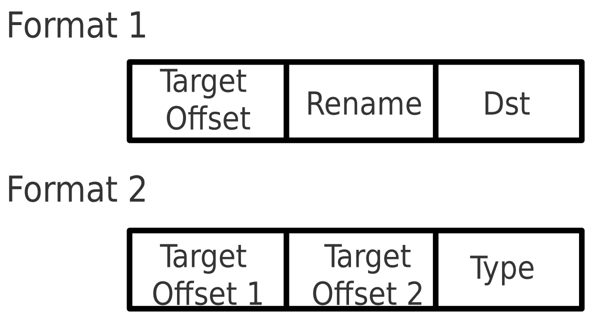


Figure 5.4: Hint encoding format for HAWS.

As discussed earlier in this proposal, HAWS is guided by compiler-generated hints to select the next instruction to fetch/execute. The two different formats for instruction hints are shown in Figure 5.4. The formats differ for different classes of instructions: format 2 is for branch instructions, while format 1 is for all the other types of instructions.

For format 1, the target offset field (9 bits) is used to help the warp scheduler find the next independent instruction to be fetched and issued. The target offset is the relative distance to the next out-of-order instruction, starting from the last stalled instruction (stall point), which triggered hint-assisted scheduling. The rename bit is used to communicate to the warp scheduler if this instruction needs to perform register renaming based on the destination register. If the rename bit is set, then the Dst field provides the renamed register number. Otherwise, it's the original destination register number.

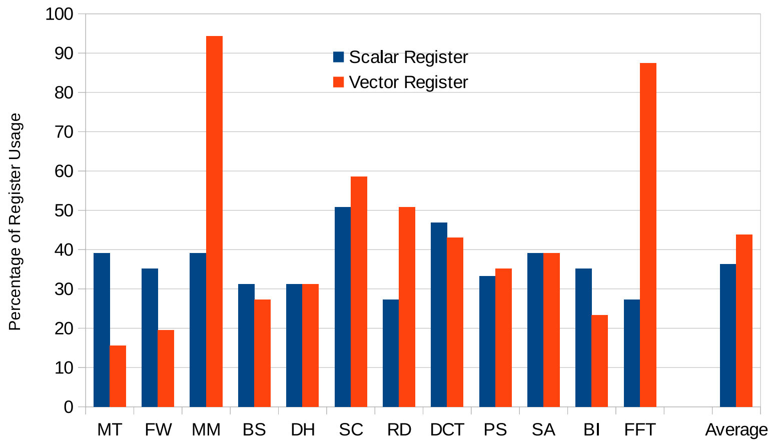


Figure 5.5: Percentage of register usage.

In HAWS execution, when instructions retire, we use register renaming to avoid WARs and WAWs when storing results, as necessary. Register files in the GPU are underutilized in many applications, which has been discussed in many recent studies [17, 34, 18]. Based on these studies and our evaluation, we propose to use the underutilized registers for register renaming. We also analyzed a variety of benchmarks, and none of them has a 100% register usage. Figure 5.5 shows that in the 12 benchmarks we evaluated, there is an average of 36.3% scalar register usage and 43.8% vector register usage. In other words, there is a lot of space for supporting register renaming. There are a couple of ways to implement register renaming efficiently. We can use the compiler to implement register renaming and insert the renamed register number directly in the hint. Using this scheme, we can utilize the registers more efficiently by renaming selected registers with static analysis.

Please note we only rename registers if necessary. For instructions that do not have a WAR dependency with any previous skipped instruction during HAWS execution, associated registers are not renamed. In other words, we only rename a register if their new value will affect the source register of any of the previous skipped instructions when the warp returns to normal execution. More details are provided in the following example.

In format 2, we need two fields to hold the target offset in the hint metadata, since there are two destinations for a conditional branch. The type field specifies the instruction type with 1 bit, 0 indicating a conditional branch, and 1 indicating a direct branch. In format 2, each offset field is 11 bits, two more bits than the offset assumed in format 1 since we are dealing with branch instructions.

HAWS only adds hints to instructions that are independent of the long latency memory instruction within a hint group. For the instructions that have dependencies on long latency memory instructions, HAWS will not add any hints, reserving instruction hints for only a limited number of instructions.

5.3.2 Compiler Work

As mentioned previously, we use a compiler (Multi2C) [22] that is designed specifically to work with Multi2Sim, and can generate instruction hints to support HAWS execution. The compiler plays two roles in the implementation of HAWS. First, it analyzes data dependencies and decides which instructions can be scheduled and executed by HAWS, providing associated instruction hints. Second, it implements register renaming and inserts operations directly into the hint, guiding HAWS to use registers efficiently.

Referring back to Figure 5.3, the compiler finds independent instructions I4 and I6 after analyzing data dependencies, adding corresponding hints to I3 and I4 to assist the warp scheduler to find the correct instructions for fetching and issuing. The compiler also inserts the destination register number in the instruction hint. More details will be provided next.

5.3.3 The Detailed HAWS Pipeline

When warps stall due to a long latency memory operation, the long latency operation flag of the scoreboard is set, which triggers HAWS. The warp state flag bit is set 1 to indicate that the warp is in the hint-assisted execution state.

In the fetch stage, HAWS fetches the instructions specified by the hint, one by one, using the hint's guidance. HAWS will not attempt to access any instruction that is not pointed to by a hint. After fetching and decoding the selected instructions, HAWS will send the source and destination register numbers to the scoreboard to see if there are any dependencies. Since HAWS will issue and execute new instructions, which will affect the state of the original scoreboard, there are a few solutions to solve this issue. One is to use two scoreboards for tracking the pending register state to support using hints. Every time HAWS is triggered, the in-order scoreboard is copied to the HAWS scoreboard, since there may be some other instructions in flight. But this method increases hardware complexity by too much, so we adopt an alternative approach which dramatically decreases hardware costs and saves GPU energy. More details will be provided in the following paragraphs.

Branch instructions: A branch instruction is executed normally in HAWS if the branch is identified as independent of any earlier long latency instructions. To support branch execution in HAWS, we duplicate the SIMT stack, since we need to track program control flow and provide the active mask for each warp. We copy the data from the original SIMT stack every time hint mode is triggered.

Barrier instructions: In regular (i.e., non-HAWs) GPU execution, since we usually execute instructions in order, whenever a warp reaches a barrier, it will stop and wait for the other warps to reach the barrier. In HAWS execution, the hint in HAWS will not select any barrier instructions, but the instructions past a barrier can still be selected and executed if they are independent of the previous long latency instruction. This feature can provide us with some significant performance benefits.

Memory instructions: Memory instructions will not be identified by a hint if a data hazard could occur. An exception to this rule are memory instructions beyond barriers will be ignored by HAWS, even if they will not generate any data hazards. The reason for this decision is based on the GPU memory model. To ensure correct results when parallel work-items cooperate, all stores from work-items in other warps in the same work-group are only visible to the current warp after a barrier.

After execution completes, the original destination register number, as well as the offset field and the renamed register number, are stored in the Result Collection Unit (RCU). This unit includes a result buffer with several entries. The unit has multiple roles in the design. First, the RCU stores the results of the out-of-order retired instructions, maintaining the mapping between the original register and the newly assigned register. The RCU uses the information provided by the hint to assist the warp schedulers, issuing instructions more efficiently and bypassing unnecessary instructions whenever the warp switches back to in-order execution. The detailed organization of the RCU is present in figure 5.6. The RCU consists of a buffer with several entries, and each entry has six different fields. Every instruction scheduled and executed by HAWS will have an entry in the result buffer.

Each entry in the result buffer has the following fields: i) offset (9 bits), ii) dst (8 bits), iii) renamed (8 bits), iv) ready (1 bit), v) control (1 bit), and vi) valid (1 bit). The six fields maintain information about the offset of the current instruction, the destination register number, the renamed destination register number, a status indicating whether the result is ready to use, if this instruction is a branch, and whether the entry is valid or not. The total size of the result buffer entry is 28 bits. We set the depth of the RCU to 8, which was selected based on our experiments.

Second, as mentioned above, we do not duplicate the scoreboard for HAWS execution. To reduce the hardware cost, and also improve the energy efficiency during HAWS execution, we reuse the dst and renaming fields in the RCU to assist HAWS execution. Since the RCU stores the original destination register number of the instructions executed by HAWS, when a new instruction is issued in HAWS mode, the warp scheduler will check not only the scoreboard but also the previous entries

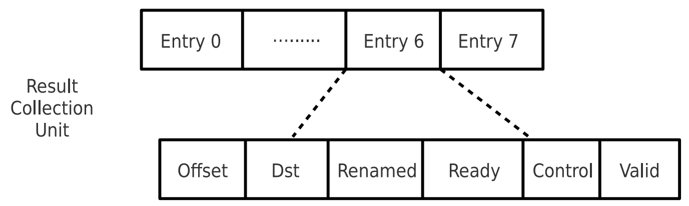


Figure 5.6: Organization of Result Collection Unit.

in the RCU to be sure it does not have a RAW dependency (WARs and WAWs are handled using a hint and register renaming). Although the RCU holds the renamed destination register number for each instruction executed by HAWS, we also introduce a register renaming table that maintains the mapping between the original register number and the most recently associated renamed register number. During HAWS execution, multiple instructions may have the same destination register, which could be renamed to different registers. Besides, the renaming table always records the latest mapping of a register and its renamed register, whenever a potential RAW hazard could occur. Thus, the following instruction will read the correct register value from the renaming table.

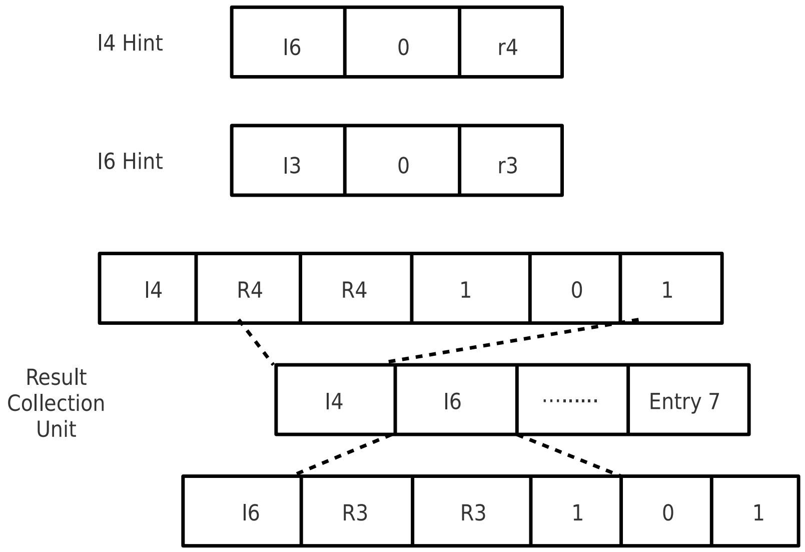


Figure 5.7: Example of hint information and RCU operation for instructions in the first experiment of figure 5.1. For clarity, all offset fields are shown as the instruction it points to.

When an instruction that was scheduled by HAWS is sent to a functional unit to execute, all fields for this entry in the RCU are updated, except for the ready field. We use the instructions in Figure 5.3 as an example. The hint information, as well as RCU operation for I4 and I6, are shown in figure 5.7. Please note for clarity, we show the relationship between instructions for individual offset fields.

Before we analyze the execution, we first focus on the instruction hint. We can see the rename bit for both instructions is 0, which means we do not have to rename the destination registers. For I4, since it is the first instruction in the hint group which can be potentially scheduled by the HAWS scheduler, there is no instruction before I4 that will use r4 as a source register, so we do not need to rename r4. For I6, the instruction before I6, which is skipped by HAWS scheduling, is I5. Since I6 does not have a WAR dependency with I5, we do not have to rename r5 either.

In the first set experiments in Figure 5.3, when I4 is fetched by HAWS, the RCU receives the information and updates the result buffer with information for I4. In our example, I4 passes the check on to the scoreboard, and since there are no valid previous entries in the RCU, the HAWS scheduler issues I4. HAWS continues at I6. Before issuing I6, the HAWS scheduler will check both the scoreboard and the previous entries in the RCU. Since I4 is already complete at this time, the RAW dependency based on the source register r4 in I6 is released, and then I6 is issued.

When I4's execution is finished, the result is written to the destination register, and the ready bit is set to 1, indicating that this instruction is finished and its results are ready to use. If there is no register renaming, the renamed field is the same as the dst field. If renaming is used, the register renaming table in the hint mode will be updated.

Although HAWS does selective out-of-order execution, the instruction with the shortest offset will be chosen first by HAWS. So all of the entries in the RCU are ordered accordingly in increasing distance values from the dependent instruction.

5.3.4 Returning to In-order Execution

Once the long latency operation completes, the long latency operation flag maintained in the scoreboard is cleared. When the warp scheduler receives this information, instruction execution switches back to in-order execution. The first instruction that is dependent on a long latency operation is allowed to issue, and the warp starts to execute instructions one-by-one in the program order. At the same time, the warp scheduler also has to check if the instruction to be fetched has already been selected by HAWS. Using the information provided by the RCU, these issues can be resolved intelligently, only fetching the necessary instructions.

Figure 5.8 shows the fetch logic when the warp returns back to normal execution, which assists the warp in finding the necessary instructions to fetch. In the first step, the fetch logic checks if the current pc is equal to the pc in the head entry of the RCU. If they are the same, the fetch logic will keep checking whether the result of the instruction is written in the original register or renamed register by comparing the renaming field with dst field in the entry. If the result is written in the original register, the scheduler can skip this instruction and fetch the next instruction. If the result

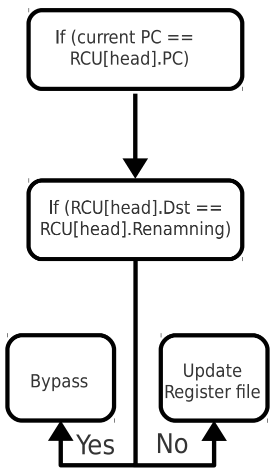


Figure 5.8: Fetch logic.

is not written to the original register, the instruction will be fetched and issued. Since the result of the instruction is stored in the renamed register, we need to copy this value to the original register. After the instruction is issued, it will bypass the execution stage and update the register file. Please note that it is necessary to fetch and issue the instructions whenever register renaming occurs since the scoreboard needs to be updated and perform dependency checking to prevent WAW and WAR hazards.

In our example in Figure 5.6, when the PC reaches I4, it checks with the head entry in the RCU to find which register where the result is written. Since there is no register renaming happening during I4's execution, the scheduler skips I4 and fetches the next instruction. The same steps are repeated when the PC reaches I6.

5.3.5 The Advantages of HAWS over Static Scheduling

We can also use the compiler to reorder independent instructions to try to hide some of the long memory latency stall time. We provide the following example to compare HAWS scheduling with static compiler-based scheduling, demonstrating the advantages of HAWS.

In this example, we have a block of code that includes three main parts: blocks 1, 2, and 3. Each block represents 15t execution cycles. There are also two components associated with each long memory stall, stall 1 and stall 2, which are 5t and 3t cycles, respectively. During blocks 2 and 3, we execute code c and f, respectively, which are instructions that are not dependent on any long latency memory operations present in the whole block.

For the normal execution in A, the program instruction blocks are issued and executed in the original order. Block 1 is executed first, after 5t cycles, a long memory stall occurs, and then

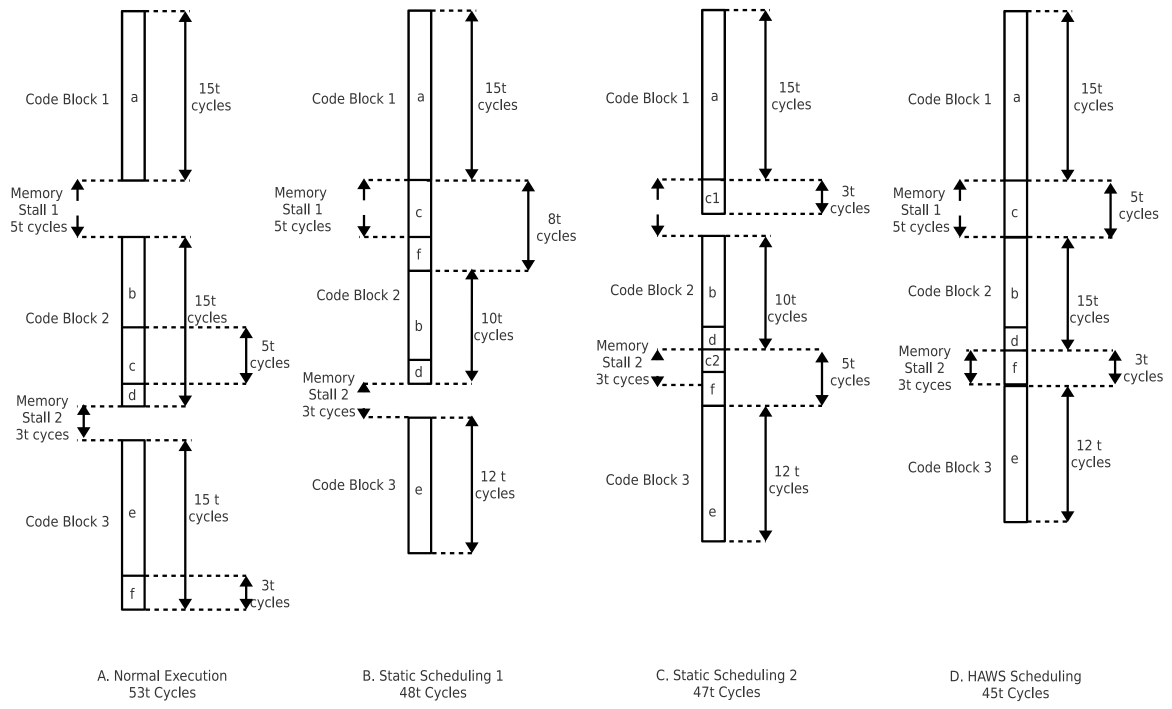


Figure 5.9: Comparison of normal execution, statically-scheduled execution and HAWS execution.

block 2 is executed. Block 3 is executed 3t cycles after block 2 completes, due to another long latency stall. The total execution time is 53t cycles. Since we can execute some independent instructions during both memory stalls, the compiler performed a reordering of instructions in those portions of c and f.

For example B, the compiler schedules all independent instructions in sections c and f after the program encounters the first memory stall. It takes 8t cycles in total to finish the execution of c and f. Although the code block 2 can be scheduled after 5t cycles, it has to wait until section f is finished. But when encountering the second stall, since there are no more independent instructions, the program will remain there for 3t cycles. In this case, the total execution time is reduced to 48t cycles, since the static scheduling performed by the compiler fully covers the first memory stall by identifying independent instructions. But the compiler can not predict how long the memory stall will take, so it over-scheduled independent instructions after the first memory stall, which leads to sub-optimal performance.

For example C, the compiler breaks code section c into subsections c1 and c2, first executing c1 (which takes 3t cycles) when the program encounters the first memory stall. It then executes c2 and f (which take 5t cycles) when the program meets the second memory stall. Since we execute some instructions to overlap with the memory stall time, the total number of execution cycles is reduced to 47t cycles. Again, similar to example B, the compiler can not know how long it will take for each long latency memory stall to be resolved. It may not schedule enough independent instructions to execute to hide the first stall.

For example D, we use HAWS scheduling. When the program encounters the first stall at the end of block 1, HAWS will try to identify as many independent instructions as possible to be included in the hint group. This will allow the hardware to execute all of the instructions in part c, completely covering memory stall 1. When the program hits memory stall 2, HAWS executes all instructions in part f, which overlap nicely with the duration of stall 2. The total execution time of HAWS execution is 45t cycles, which is 2t cycles faster than using static scheduling in example C and 3t cycles faster than example B.

Comparing examples B, C, and D, when the program encounters the first stall, the number of independent instructions scheduled by the compiler are either too few to completely hide the delays associated with the long latency stall, or too many, impeding independent instructions from covering other stalls. In contrast, HAWS will try to execute as many instructions as possible, which will cover all of the stall time generated by long latency stall. This is why HAWS scheduling does better than static scheduling in this example.

Since there are many factors which may influence the duration of long latency memory stalls during execution, even for the same kernel, the stall time for the long latency memory operation could be dependent on the input data size used for the kernel. It is challenging to use static scheduling to cover all memory stalls. Since HAWS makes dynamic decisions in concert with the warp scheduler, guided by a compiler-generated instruction hint, and it tries to execute as many instructions as possible. HAWS will provide us with better performance in most cases.

5.3.6 Area Estimation

As described earlier, we will have to add extra hardware to support HAWS execution. The Result Collection Unit (RCU) must be added, which includes eight entries, with 28 bits per entry. Our baseline GPU can support up to 40 warps per SM, so the total size of the RCU is 8960 bits. We need to copy the SIMT stack when there is a branch instruction scheduled by HAWS. Each entry in the SIMT stack contains the active mask (64 bits), the current pc, and the reconvergence pc, with 32 bits for each field. The SIMT stack is 128 bits per entry, with eight entries. So the total size for the SIMT stack is 40,960 bits per SM. The renaming table consists of two parts, the scalar register renaming table, and the vector register renaming table. Since there are eight entries in the RCU, supporting up to 8 instructions executed whenever the warp switches to HAWS mode, we also set the renaming tables to 8 entries. For scalar registers, each entry has 16 bits, 8 bits for the original register number, and 8 bits for the renaming register number. We assume the same design for the vector register renaming table, with 16 bits per entry. The total size for the register renaming table is 10,240 bits. The overall overhead is 7.34 KBytes/SM.

Based on CACTI modeling [39], the additional elements associated with HAWS take $0.045mm^2$ per SM. Compared with the baseline GPU model, which is based on the AMD Radeon 7970 [33] which contains 32 SMs and a die size of $352mm^2$, the total area overhead is only 0.4% of the overall chip size.

5.4 Evaluation

5.4.1 Methodology

Table 5.1: List of benchmark applications

Applications	Abbr.	Type
Binary Search	BS	M
Bitonic Sort	BI	N
Discrete Cosine Transform	DCT	M
DwtHaar1D	DH	M
FFT	FFT	N
FastWarshTransform	FW	M
MatrixMultiplication	MM	M
MatrixTransform	MT	M
PrefixSum	PS	M
Reduction	RD	M
ScanLargeArrays	SA	M
SimpleConvolution	SC	M

To evaluate our design, we select representative benchmarks from the AMD APP SDK 2.5 benchmark suite. The benchmarks provide us with a rich set of both memory-intensive and compute-intensive benchmarks, as well as a mix of both, representing a wide range of application behaviors. The 12 benchmarks evaluated in this work are listed in Table 5.1, along with some of their application properties. Again HAWS scheduling is targeting memory intensive benchmarks, so a majority of the benchmarks on our list are memory intensive, though we include compute-intensive to make sure we do not impact their performance.

As mentioned earlier, we model our baseline GPU architecture using the Mult2Sim simulator [67, 16], which is a cycle-level heterogeneous simulation framework supporting both GPU and CPU simulation. All experiments are based on a baseline GPU model similar to the AMD Radeon

Table 5.2: Baseline GPU model configurations

GPU Model	AMD Radeon 7970
Clock Frequency	1000 MHZ
Number of SMs	32
Max Num. Warp / SM	40
Scheduler Policy	GTO
SIMT Units / SM	4
SIMT lane width	16
L1 Cache / SM	16 KB, 64 Sets, 4 way, 64B / line
L2 Cache	128 KB, 128 Sets, 16 way, 64B / line
Scalar Registers / SM	2048
Vector Register / SM	65536
LDS / SM	64 KB

7970, whose hardware specifications are summarized in Table5.2. We extended the timing part of the AMD GPU model in Multi2Sim to support HAWS. The Radeon 7970 is a high-end AMD GPU that supports instruction-level parallelism, allowing multiple instructions to be issued in the same cycle. AMD used the code name *Southern Islands* to identify the ISA used in the 7970.

5.4.2 Performance

We compare HAWS scheduling against a Greedy-Than-Oldest (GTO, our baseline) [58], CTA-Aware scheduling [19], and warped-preexecution scheduling techniques [24]. Figure 5.10 shows the overall performance achieved by HAWS and the competing techniques for both memory intensive and non-memory intensive applications, as compared to the baseline GPU model. We separate memory intensive benchmarks and those are not in the chart.

CTA-aware tries to reduce cache contention and utilize data locality by prioritizing subgroups of warps to execute and access memory, giving them a better chance to better utilize the data and maximize reuse opportunities. We observe a speedup in most of the benchmarks for CTA-aware over the baseline model. CTA-aware not only exploits intra-warp locality but also exploits inter-warp locality, while GTO only gives priority to the oldest warps. However, due to the improvement of our baseline GPU model, CTA-aware is not as effective as reported in the older architecture. It achieves 4% speedup in overall.

Note that the scheduler for both the baseline GPU model and the regular execution in HAWS use a greedy-than-oldest (GTO) scheme. In all of the benchmarks, HAWS achieves a maximum performance improvement of 34.2%, as found in the BS benchmark. From the results, we can also see HAWS does not negatively impact non-memory intensive applications, providing

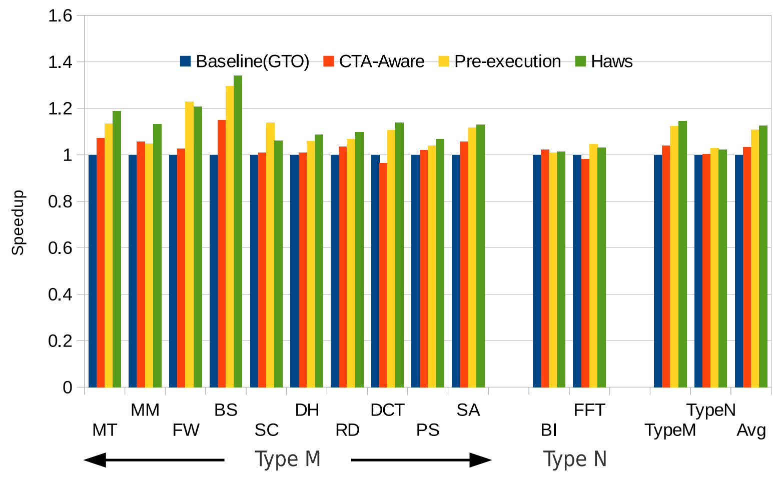


Figure 5.10: Speedup of HAWS, relative to the baseline model.

an average of 2.3% improvement for FFT and BI, since we benefit from the dynamics of the warp scheduler, based on the current instruction hint to increase instruction-level parallelism of the GPU.

Pre-execution scheduling tries to issue future independent instructions whenever a warp stalls by relying on the hardware to dynamically detect such opportunities. For most of the workloads we have studied, we observe that HAWS has a better speedup than pre-execution scheduling. The reason for this is HAWS can locate the next independent instruction directly and execute it according to the instruction hints, while pre-execution scheduling needs to use hardware to dynamically detect if the next instruction is independent with the long latency operation or not. Sometimes it takes longer to find an appropriate candidate to fetch and execute. While pre-execution scheduling provides a 12.4% average speedup for memory-intensive workloads, HAWS offers a higher speedup of 14.6%.

The speedup of HAWS scheduling, when running memory-intensive workloads, is higher than for non-memory intensive benchmarks, given that memory-intensive applications have more long latency memory stalls, which provides the warp schedulers with more opportunities to trigger HAWS scheduling.

5.4.3 Impact on Memory Stalls

Figure 5.11 shows the breakdown of scheduling cycles for all of the benchmarks. The portion of the stacked bars identified as *Issued* denotes that at least one warp in the GPU is eligible to be issued. The *Long Latency Stall* portion is defined as all warps in the GPU that are stalled due to a long latency memory operation. Since multiple warps are allowed to be issued by the same warp scheduler in the same cycle in AMD GPUs, we label the rest of the stalls as *Mixed Stall*. This denotes that no warp can be issued in that cycle, which may be due to a variety of reasons (e.g., there are no

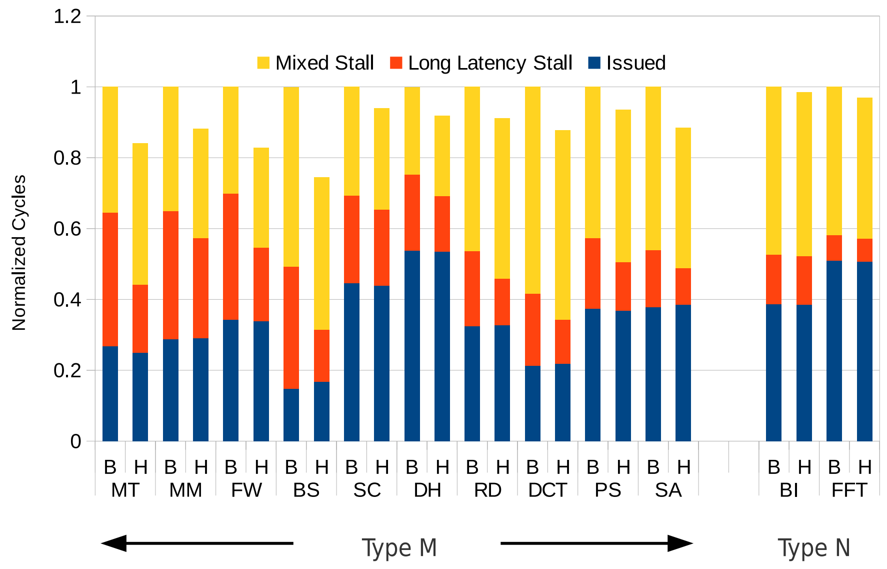


Figure 5.11: Breakdown of Scheduling Cycles (B:Baseline, H:HAWS).

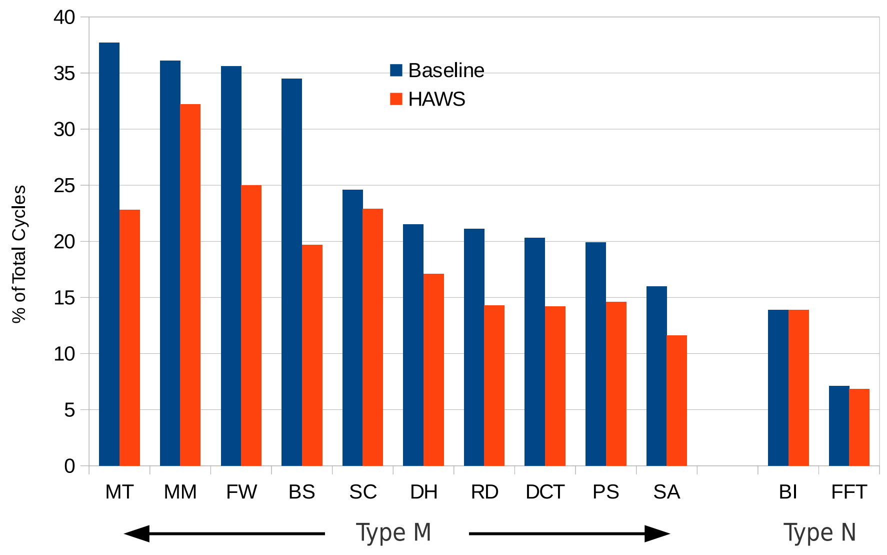


Figure 5.12: Percentage of long latency stall, comparing HAWS with baseline model.

available instructions in the fetch buffer, the pipeline is busy, a data dependencies occurred, etc.). For the same benchmark, both the baseline and HAWS execution have a similar total number of *Issued* cycles, which makes perfect sense since they are executed while considering the same resource. In type M benchmarks, the *Long Latency Stall* cycles are significantly reduced by the HAWS scheduler, as compared to the baseline model. We take a closer look at this in Figure 5.12.

Figure 5.12 shows the percentage of long latency RAW stalls in the benchmarks when using HAWS. For comparison, we presented earlier the long latency stalls percentages for the baseline model in Figure 5.12. For those applications that enjoy significant speedup using HAWS, including BS, MT, and FW, HAWS can significantly reduce stalls due to long latency RAWs in those benchmarks up to 14.9%. We can see improved speedup for these benchmarks in Figure 5.10. For non-memory intensive benchmarks, the long latency stalls can only be reduced marginally by HAWS, as discussed previously, since the total percentage of long latency stalls is not as high.

5.4.4 Opportunities for Instruction Level Parallelism

Figure 5.13 shows the average number of SIMT instructions executed by a warp whenever using HAWS scheduling. As mentioned previously, HAWS will only select instructions within a hint group to execute. From our study, across all of the benchmarks, 0.55 instructions (less than 1) are executed on average by a warp whenever we switch to hint-assisted mode. The memory intensive workloads play a significant role in dominating this average. For example, in the DCT workload, we see 1.23 instructions executed on average per switch to HAWS.

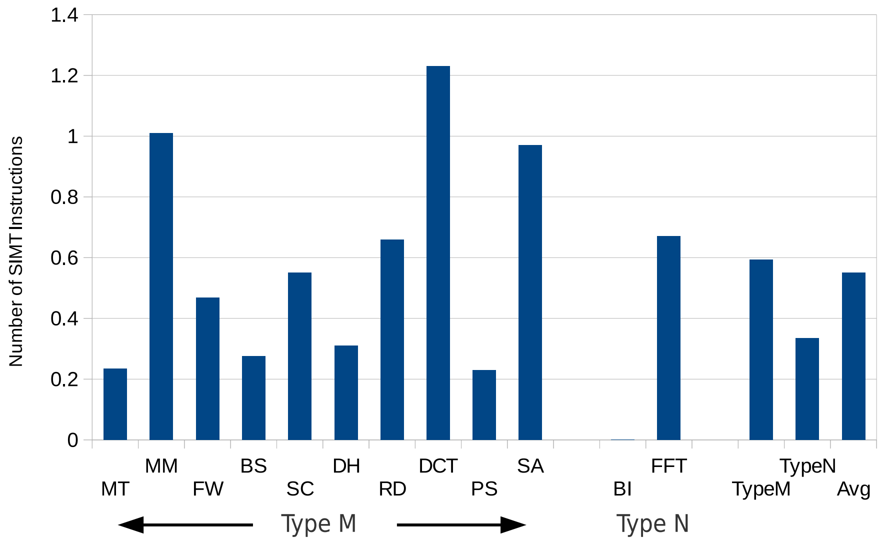


Figure 5.13: Average number of SIMT instructions executed per warp for per HAWS scheduling.

Figures 5.14 and 5.15 show a breakdown of the instruction types executed by HAWS. As shown in Figure 5.14, MM executes almost 20% of its total ALU instructions in hint mode, while FFT only executes 0.39% of its total ALU instructions. BS executes nearly 20% of its total long latency memory instructions in hint mode, as shown in Figure 5.15. Combining these two figures, we can conclude that for a workload to benefit from HAWS, it will need to have either have a high percentage of ALU instructions. In applications such as MM that contain a limited number

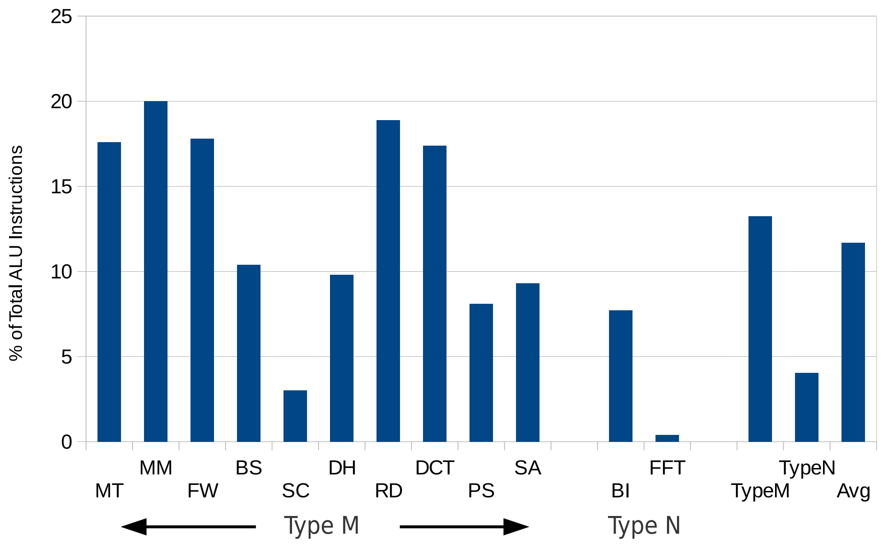


Figure 5.14: Percentage of total ALU instructions executed by HAWS.

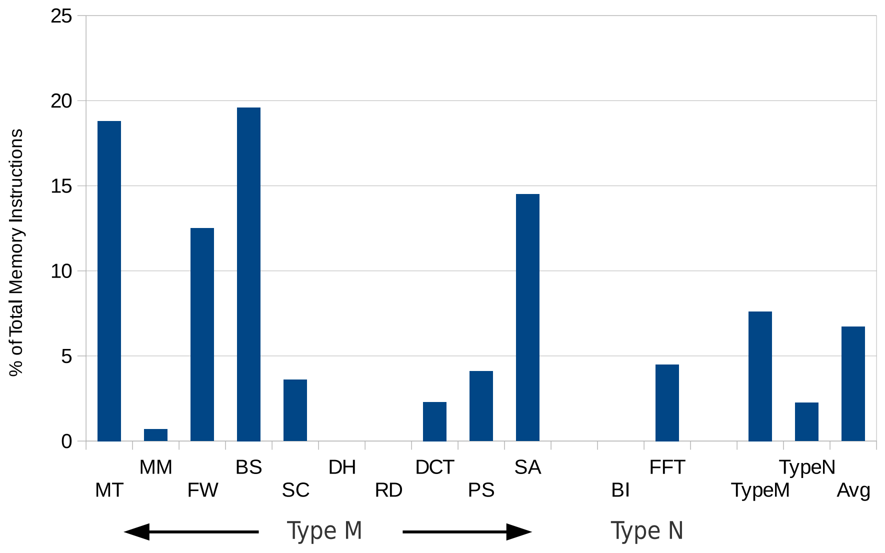


Figure 5.15: Percentage of total Memory instructions executed by HAWS.

of memory instructions executed in HAWS mode, if they have a high rate of ALU instructions executed, they can still achieve excellent performance with HAWS. Inspecting the code closer, both MM and DCT have long latency memory instructions inside a loop. These instructions are used for loading operands that are the inputs for vector multiplication - these are operations that can trigger HAWS execution. Every time the program stalls due to a long latency memory instruction, HAWS schedules ALU instructions for execution. Since HAWS will be triggered repeatedly, this increases the percentage of ALU instructions executed by HAWS.

5.4.5 Experiments on Kepler GPU

We also evaluate HAWS on the Kepler architecture. Since the NVIDIA compiler is proprietary, we manually inserted instruction hints in our benchmarks as shown in Table 5.3. We use the same GPU configuration, as shown in 4.1. In figure 5.16 we can observe that HAWS outperforms the baseline round-robin scheduler. Similar to our own previous results, HAWS performs better on memory-intensive benchmarks and can achieve a maximum speedup of 26% on the FW benchmark. On average, it can achieve 15% across five workloads evaluated.

Table 5.3: List of benchmark applications

Applications	Abbr.	Type
Matrix Multiplication	MM	M
Matrix Transpose	MT	M
Scalar Product	SP	M
Inline Math	IM	M
Bitonic Sort	BS	N

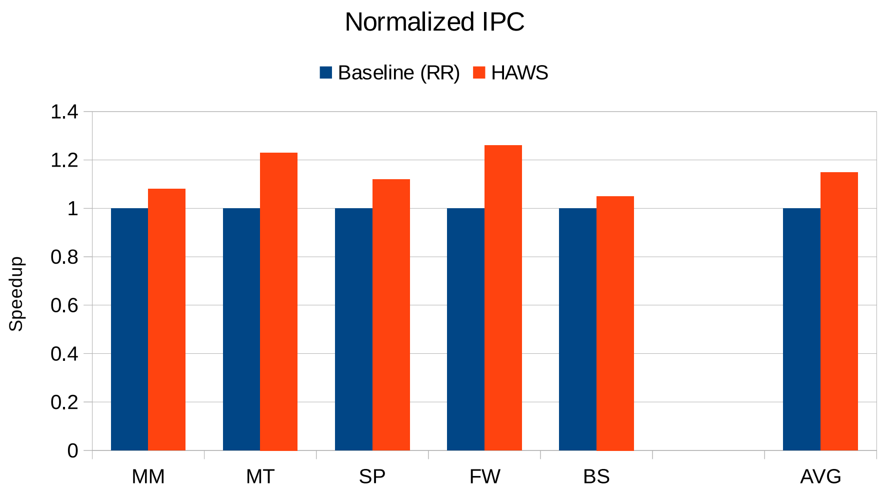


Figure 5.16: Speedup of HAWS on simulated Kepler GPU.

5.5 Summary

In this chapter, we have presented a Hint Assisted Warp Scheduler, a novel GPU warp scheduler which uses compiler-generated hints to better utilize GPU hardware resources. Our hints are used to identify opportunities for out-of-order execution in the shadow of warp stalls caused by long latency memory operations. Compared to static approaches, HAWS provides us with a dynamic solution that can take full advantage of unused cycles due to memory stalls. We presented the design and evaluated potential performance benefits across a variety of benchmarks. Our results show HAWS can improve performance, on average, by 15.3% for memory intensive applications. For non-memory intensive workloads, HAWS is 2.3% faster than the baseline model.

Chapter 6

Hint-Assisted Locality-Aware Scheduling

6.1 Redundant MSHRs allocation across the GPU

Whenever a streaming multiprocessor (SM) executes an instruction, the instruction is fetched and decoded by the front-end of the SM and sent to the appropriate execution unit. Specifically, the Load/Store unit executes all memory instructions, generating a request for each instruction and sends them to the L1 cache. On a GPU, each SM has an on-chip L1 cache. When the L1 cache receives memory requests from the Load/Store unit, it compares the address tag first and checks if the request is a cache hit. If the request is a hit, the L1 cache will respond with the corresponding data in the cache. Otherwise, the L1 cache will inspect the Miss Status Holding Registers (MSHRs) to see if the requested cache line also being requested by another thread. The L1 cache will append source information such as the warp ID and the thread ID to the existing MSHR entry if the request is already registered. If not, a new entry will be allocated in the MSHR. Since the GPU uses a non-blocking L1 cache, the number of total cache misses it can process simultaneously depends on the number of MSHR entries. Whenever all the MSHR entries are occupied by outstanding memory requests, the upcoming memory requests will be blocked in the pipeline and will be retried in future cycles until a MSHR is freed. In this situation, every instruction that has a dependency on the blocked memory address will wait until the memory instruction completes, resulting in underutilizedd GPU computing resources.

Since the maximum number of memory requests that an SM can process simultaneously depends on the number of MSHR entries, we study how MSHR entries are utilized during GPU application execution. We first define the notion of a redundant MSHR entry. An MSHR entry

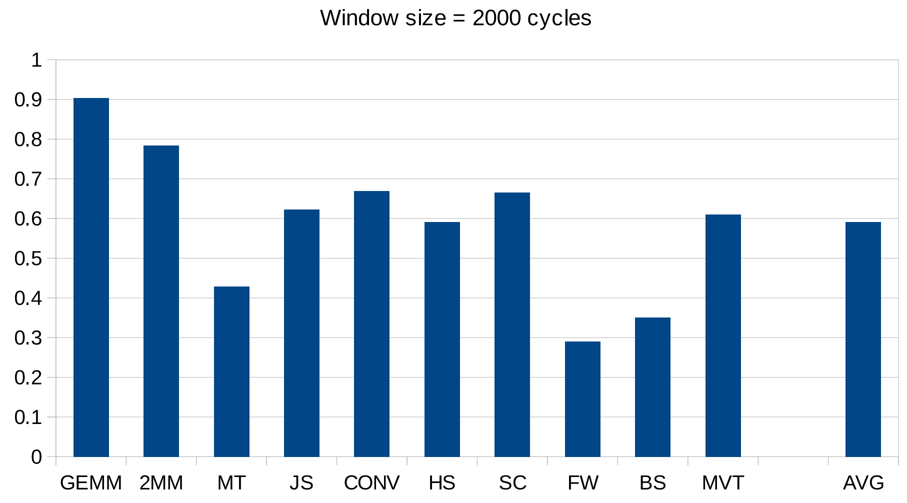


Figure 6.1: Percentage of MSHRs allocated for duplicate cache lines in a given time. The window size is 2000 cycles in the diagram.

is determined to be redundant if the cache line it is allocated for has been assigned to a previous allocated MSHR entry from other SMs. The previous allocation needs to have happened within a particular timeframe. Figure 6.1 shows the percentage of MSHRs allocated for duplicate cache lines within 2000 cycles. The reason we choose 2000 cycles as the window size is because the L1 cache miss latency is up to a few thousand cycles. Different benchmarks exhibit a range of behaviors in terms of the percentage of duplicate MSHR entries. On average, for the given window size of 2000 cycles, we find that almost 60% of MSHR entries allocated are duplicates, which means that for those MSHR entries, there is another MSHR entry belonging to a different SM which has requested the same cache line. GEMM and 2MM have a very high degree of redundant MSHR requests, as high as 90%. The number of MSHRs entries is limited for the GPU's L1 cache, so redundant allocation of MSHR entries easily consumes the MSHR space, blocking upcoming memory requests and degrading overall performance. In the NVIDIA Kepler GPU, there is room of 8 MSHR entries. In general, the more requests from different warps an MSHR can manage, the better we can utilize the MSHRs efficiently.

6.2 Benchmark Characterization

Since the number of MSHRs is a key design point that determines whether SMs can serve additional memory requests to feed compute instructions, it is essential to investigate why the redundancy percentage is so high across the SMs on a GPU. The main reason for the high redundancy is due to the inter-block locality. Figure 6.2 shows how blocks are distributed during application execution. Traditionally, adjacent blocks from the same grid which share data locality are mapped to different SMs in round-robin fashion. We identify three main categories of the inter-block locality as follows.

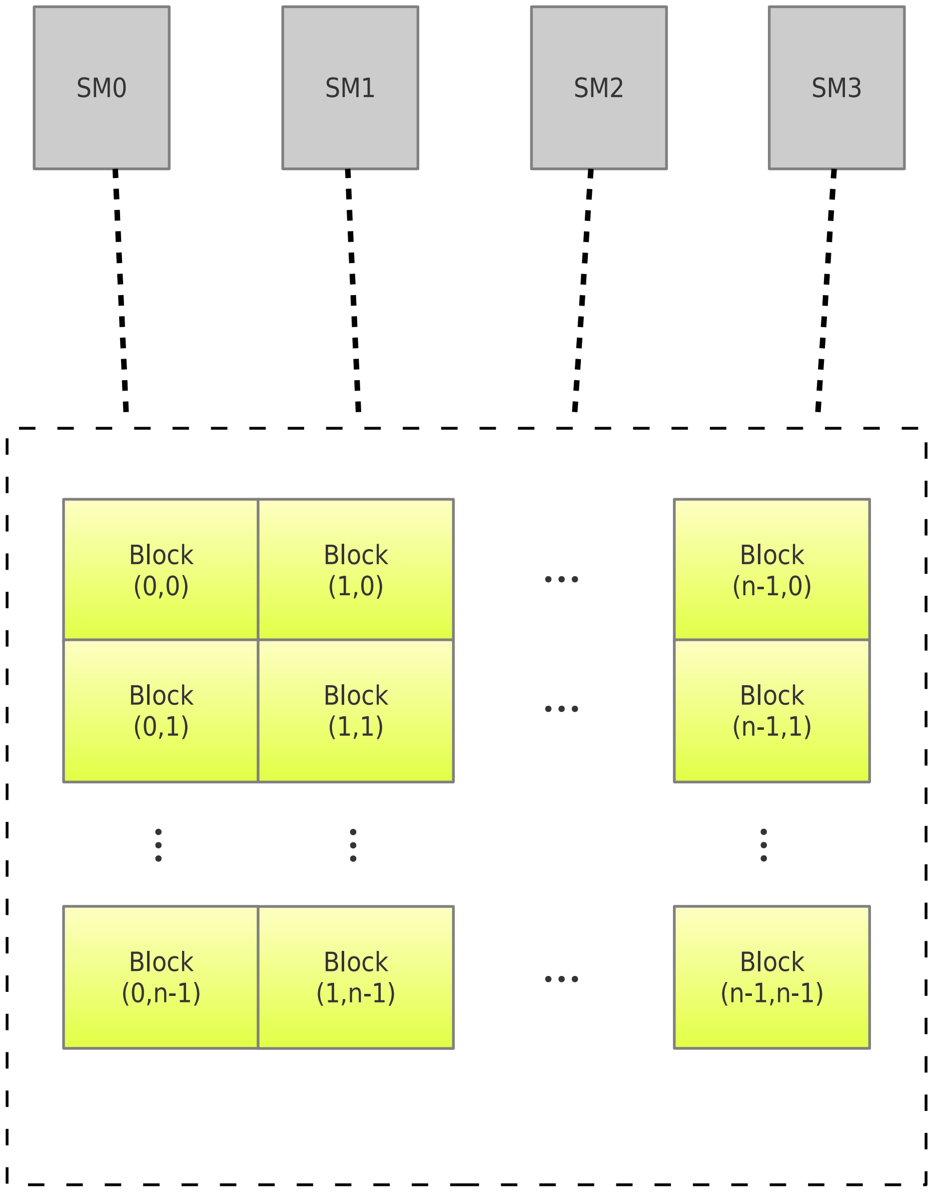


Figure 6.2: Block scheduling inside GPU. Adjacent blocks are assigned to different SMs in round-robin fashion.

- Multiple cache line shared among inter-blocks

Inter-block locality exists in many workloads, especially in some 2D benchmarks. There is a high degree of spatial locality across adjacent blocks. The adjacent blocks will share multiple cache lines in some applications, such as matrix multiplication. Figure 6.3 shows the process of matrix multiplication in the GPU. We calculate C = A x B in GPU, where each input matrix is divided into n x n blocks. Each block is responsible for calculating the corresponding elements of the result matrix. To calculate the first element in the block(0,0) of result matrix, we need to read the first row of matrix A and the first column of matrix B. To calculate the first element in the block(1,0) of result matrix, we will also need to read the first row of matrix A and the first column of matrix B. Based on this access pattern, we find that whenever we calculate the elements with the same thread ID inside different blocks which have the same blockIdx.y, the blocks will read the same row of the input matrix multiple times. That helps to explain why the percentage of redundant MSHR allocations is high in benchmarks GEMM and 2MM since adjacent blocks are distributed on different SMs. High spatial locality exists in this category of benchmarks.

Next, we take a closer look at the data access pattern for each warp across different blocks. When we calculate all elements inside warp 0 from the block(0,0) in matrix C, the data we need to read from matrix A is all elements inside warp 0 from the block(0,0), all elements inside warp 0 from the block(0,1), and all elements inside warp 0 from all other blocks whose blockIdx.y is 0. How about calculating all elements inside warp 0 from the block(1,0) in matrix C? We find that we still need to read all data from warp 0 inside the blocks whose blockIdx.y is 0. Although they will read different columns from matrix B, to calculate warp 0 in the block(0,0) and warp 0 in the block(0,1), 50% of the addresses are the same. If warp 0 from the block(0,0) and warp 0 from the block(0,1) are issued within a short time period, the MSHR will collapse 50% of the total requests, reducing the memory pressure. Furthermore, all warps that share the same ID in different blocks, as whose blockIdx.y is the same, will experience 50% of data reuse. If we can schedule these warps together, we can further reduce the memory pressure and insert more requests from different warps in every MSHR entry.

- Boundary cache lines shared among inter-blocks

  In many GPU workloads such as stencils and convolutions, we observe that adjacent blocks share boundary cache lines. The number of cache lines shared depends on the specific algorithm and halo size. In Figure 6.4, we show a diagram of a 2D stencil workload running on a GPU. We can observe that there are cache lines shared between adjacent blocks.

  For example, block (1,0) shares a common cache line with blocks (0,0), (0,1) and (1,1) in the

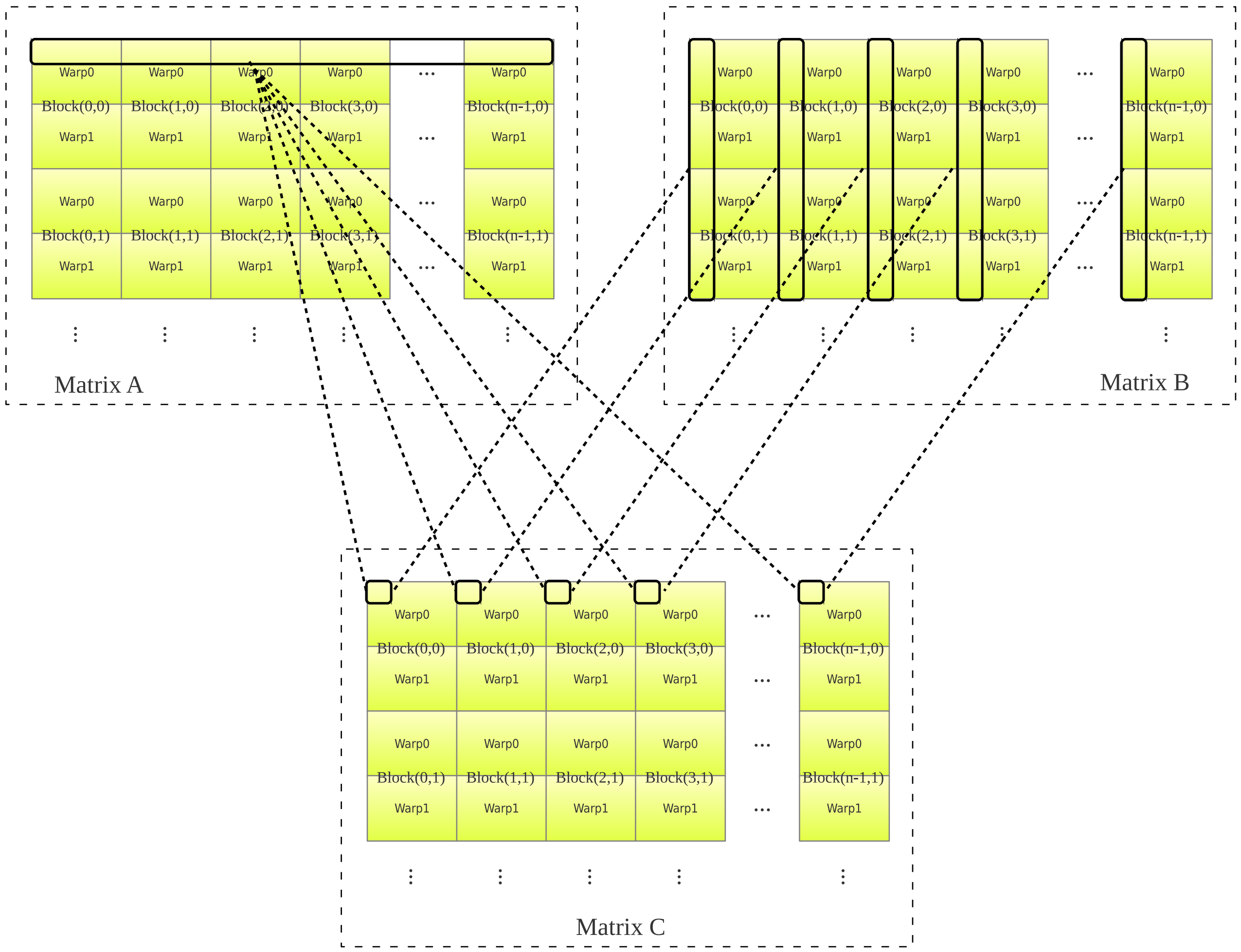


Figure 6.3: High degree of spatial locality across adjacent blocks. Multiple cache line are shared.

picture. For blocks (1,0) and (1,1), one or more cache lines may be shared depending on the halo size in the workload.

Again, since adjacent blocks are distributed to different SMs in a round-robin order, benchmarks such as JS and CONV experience a high redundancy percentage (up to 68%). Since only boundary cache lines are shared among adjacent blocks, the benchmarks in this category do not have as high spatial locality as the first category. We can observe that from Figure 6.1, GEMM and 2MM have a higer rate of redundant MSHR allocations, but still have high spatial locality.

- Some common data shared among inter-blocks

We also find that different blocks access common addresses in main memory in some GPU applications. The exact memory access pattern depends on the benchmarks. Figure 6.5 shows

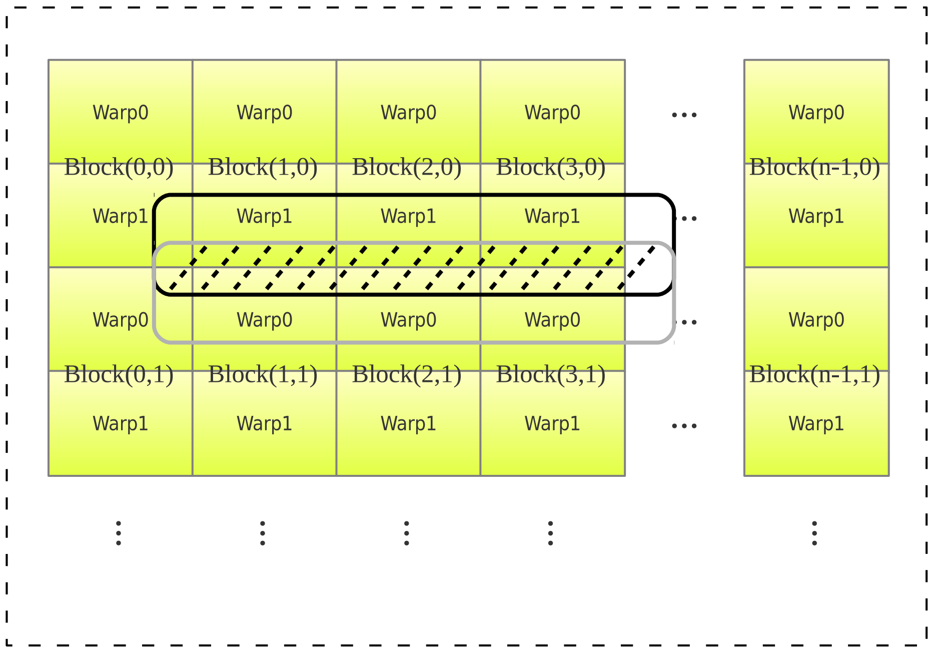


Figure 6.4: High degree of spatial locality across adjacent blocks. Boundary cache line are shared.

a snippet of code for the Histogram benchmark. When the blocksIdx.x and i are the same for different blocks, the threads from different blocks access the same addresses in main memory. Besides accessing common addresses, these blocks also share some constant data in some benchmarks. It is common that two different blocks will share the same data, but compared to the patterns associated with the other two categories, they have lower spatial locality. From Figure 6.1 we can see that HS, FW and BS have a lower percent of redundant MSHR allocations.

```
uint sum = 0;
for (uint i = threadIdx.x; i < histogramCount; i += MERGE_THREADBLOCK_SIZE)
        sum += d_PartialHistograms[blockIdx.x + i * HISTOGRAM256_BIN_COUNT];
```

Figure 6.5: Low-Mid degree of spatial locality across adjacent blocks. Some data are shared. Code sample from HS benchmark.

6.3 Hint Assisted Locality Aware Scheduling

Based on our analysis of why there are so many redundant MSHR allocations, we propose hint-assisted locality-aware scheduling. First, we design a new scheduling algorithm based on the different classes of locality discussed above. Then we extend the CUDA runtime API to allow users to input scheduling hints, assisting the GPU to adopt different scheduling algorithms. Our goal is to make each MSHR entry hold more requestis from different warps and reduce MSHR pressure.

6.3.1 User Input Scheduling Hint

According to our previous analysis, we define three different address patterns based on the cache lines shared among blocks in the workload. To better assist the scheduler when making smart decisions during execution, we extend the CUDA APIs to allow users to input a scheduling hint based on different address patterns present in their workload.

First, we introduce how we insert the hint. When we launch a CUDA kernel in an application, we call the CUDA runtime API shown in Figure 6.6. Besides the original arguments from the kernel, we also need to pass another two variables, which are the grid size and the block size. These variables are stored in a special register in the GPU. When the GPU starts to execute the kernel program, it will use the value stored in the special registers to decide the block size and grid size. We extend the official CUDA API and introduce an extra field when we launch the kernel, which is shown in Figure 6.7. In addition to the grid and block size in the official CUDA API, we add one more variable, which is the hint value. This value will tell the scheduler which kind of address pattern is associated with this kernel. The scheduler can then make better scheduling decisions for the blocks and warps during execution. The detailed design of the hint encoding is shown in Table 6.1.

There are four types of hints the user can specify. When the hint value is 1, it indicates the workload has high spatial locality and multiple cache lines are shared across blocks. When the hint value is 2, it means boundary cache lines are shared among blocks in your workload, and if it is 3, then some common data and constant values are shared among blocks. We use the value 0 to disable hinting. Whenever the user is not sure about which type pattern is present, we can pass 0, resulting in using the default warp scheduler. For every hint a users inputs when launching kernels, the scheduler chooses the appropriate scheduling algorithm to maximize the performance. The details about how we redesigned the scheduler to exploit the hint are described next.

```
Kernel_name <<<gridsize, blocksize >>> (Argument1, Argument2, ...);
```

Figure 6.6: Kernel launch using official CUDA API.

```
Kernel_name <<<gridsize, blocksize, hint_value >>> (Argument1, Argument2, ...)
```

Figure 6.7: Kernel launch using our extended CUDA API.

Table 6.1: Hint encoding

Hint value	Workload Type
0	Not specified
1	Multiple cache lines shared among blocks
2	Boundary cache lines shared among blocks
3	Some data and constant value shared among blocks

6.3.2 Locality Aware Scheduling

When the hint value equals to 1, 2 or 3, data locality exists among adjacent blocks. To reduce the number of redundant MSHR entries across different SMs and improve overall performance, at the block level, our locality-aware block scheduler assigns adjacent blocks to the same SM. Figure 6.8 shows the details of the distribution of blocks. Compared with a round-robin scheduling scheme, now we assign adjacent blocks to the same SM, maximizing the chance for the warps to access the same cache line in the same SM. For the workloads whose hint value is 0, the locality-aware scheduler uses the original block scheduler.

After the blocks are assigned to the SMs, the locality-aware warp scheduler will schedule the warps according to the programmer-guided scheduling hint. As we discussed previously, the locality-aware warp scheduler will pick the appropriate scheduling algorithm based on the hint value. The details are as follows:

- Multiple cache lines shared among blocks.

  For workloads that share multiple cache lines among blocks, the hint value is set to 1. Since multiple cache lines are shared among blocks, this type of access pattern is accompanied by high spatial locality. To maximize the chance to fully exploit the data locality and reduce MSHR pressure, we propose to use a hint-based two-level warp scheduling algorithm. As we discussed previously, in workloads such as matrix multiplication, the warps share the same ID inside blocks and the blockIdx.y is the same, they will access 50% of the same addresses. In contrast to a traditional two-level warp scheduler, for all the blocks which have the same blockIdx.y, we will group the warps with the same warp id within the same block together.

  Figure 6.9 shows details of how to schedule warps in the same group. We group all warp 0s from blocks with the same blockIdx.y together (warp 0 in block(0,0), warp 0 in block(0,1), etc.). We do the same for warp 1. The reason we use a two-level warp scheduler is that we can group warps that have high data locality, but from different blocks, together. With our two-level warp scheduler, warps within the same group have an equal opportunity to fetch and issue instructions and are scheduled in a round-robin fashion. In this case, there is a very

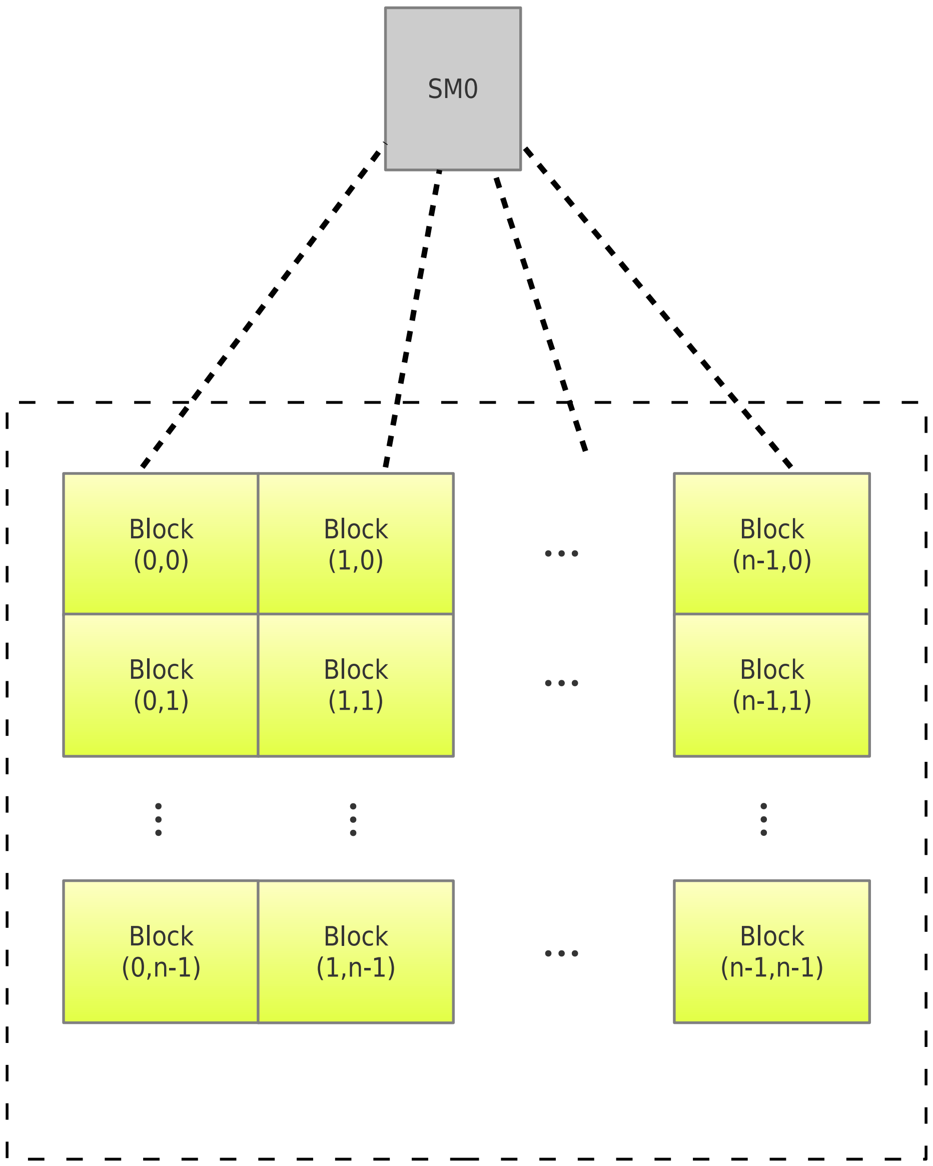


Figure 6.8: Block distribution for benchmarks which have data locality among blocks.

high chance for the warps inside the same groups to issue their memory loads in close time proximity. The MSHR entry can maximize its capacity to append all these memory requests together. Once all the warps in the current group are stalled, the warp scheduler will switch to another group. The switching policy among groups is also round-robin.

- Boundary cache lines shared among blocks.

  We take another look at Figure 6.4, where the workload hint value is 2, meaning that adjacent blocks share boundary cache lines. For example, the lower boundary of warp 1 in block(1,0)

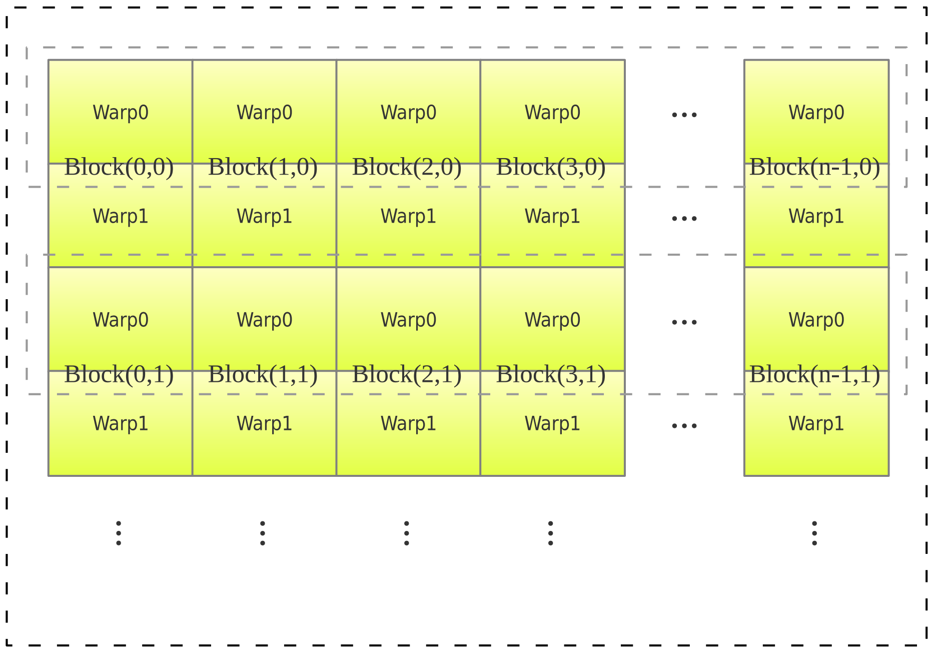


Figure 6.9: Warp scheduling for workloads which have multiple cache lines shared among blocks.

shares common data with warp 1 in block(0,0), warp 1 in block(2,0), warp 0 in block(0,1), warp 0 in block(1,1) and warp 0 in block(2,1). In all the boundary warps which share the common addresses with warp 1 in the block(1,0), we can observe the warp 0 in block(1,1) shares all the addresses in the entire cache line, while the other warps only share part of the cache line.

Based on this observation, we will group warps whose blockIdx.x is the same together. In this case, warp 1 from the block(0,0) and warp 0 from the block(0,1) will be in the same group.

- Some common addresses shared among inter-blocks

  For this type of workload, the amount of data shared among blocks varies in the benchmarks. We distribute warps in a round-robin fashion to different groups.

6.4 Evaluation

6.4.1 Methodology

To evaluate our hinting design, we select a rich set of benchmarks from the CUDA SDK [42] and Polybench benchmark suite [54]. Here we define type 1 as kernels that share multiple cache lines among adjacent blocks, type 2 for kernels that share boundary cache lines among blocks, and

Table 6.2: List of benchmark applications

Applications	Abbr.	Type
General Matrix Multiplication	GEMM	Type 1
2 Matrix Multiplication	2MM	Type 1
Matrix Transpose	MT	Type 1
Jacobi 2D stencil	JS	Type 2
Convolution 2D	CONV	Type 2
Histogram	HS	Type 3
Scan	SC	Type 3
Fast Walsh Transform	FW	Type 3
Bitonic Sort	BS	Type 3
Matrix Vector Product and Transpose	MVT	Type 3

Table 6.3: Simulated GPU configuration.

Clock Frequency	732 MHZ
Number of SMs	14
Max Num. Warp / SM	64
Scheduler Policy	Round Robin
Number of registers / SM	65536
Shared memory / SM(KB)	16
Number of MSHR entries / SM	16
Max memory requests / MSHR	8
L1 Cache / SM	16 KB, 4 way, 128B / line
L2 Cache	1536 KB, 16 way, 128B / line

type 3 for kernels that share some random data and constant data. The asssembled set of benchmarks provide us with a rich set of all three categories of patterns. The ten benchmarks evaluated in this work are listed in Table 6.2, along with their application properties. Hint-assisted locality-aware scheduling is used for benchmarks with high spatial locality. A majority of the benchmarks on our set exhibit high data locality. We also include some benchmarks with low to medium data locality, enabling us to evaluate how our scheduler performs on different degrees of locality.

We model our baseline GPU architecture using the Multi2Sim Kepler simulator [67, 16], which is a cycle-level heterogeneous simulation framework supporting both CPU and GPU simulation. All our experiments are based on the baseline GPU model, whose hardware specifications are shown in Table 6.3. As discussed previously, to support hint-assisted locality-aware scheduling, we also extended the CUDA runtime and driver APIs in our simulator to support hint insertion.

6.4.2 Performance

We compare our hint-assisted locality-aware scheduler with a baseline round-robin scheduler, Jog et al.'s CTA-Aware scheduler [19] and Sethia et al.'s MASCAR [63] scheduling technique. Figure 6.10 shows the overall performance achieved by our hint-assisted locality-aware scheduler, as compared to the other competing schedulers for all three types of workloads.

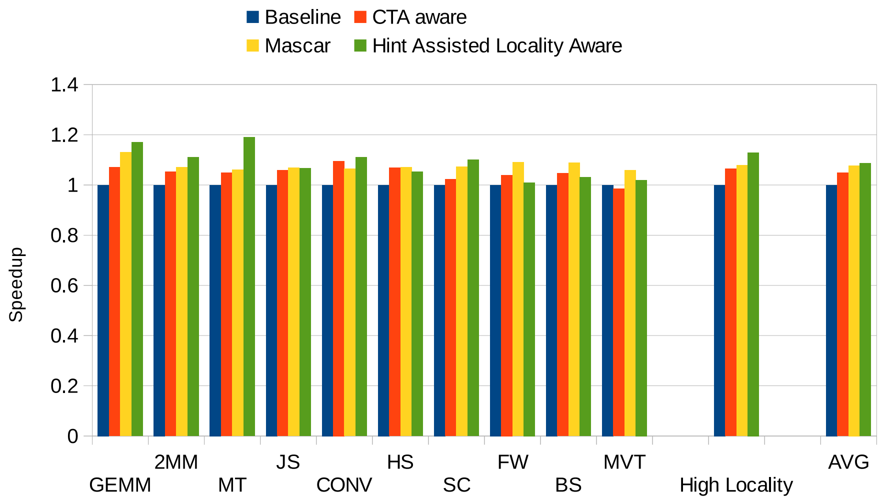


Figure 6.10: Speedup of hint-assisted locality-aware scheduler, relative to the baseline round robin model.

The CTA-aware scheduler tries to reduce cache contention and utilize temporal locality by prioritizing sub-groups of warps to execute and access L1 cache, giving them more opportunity to benefit from temporal locality and maximize data reuse. We can observe a speedup in most of the benchmarks for the CTA-aware scheduler over the baseline round-robin model. The CTA-aware scheduler exploits both intra-warp locality and inter-warp locality. However, since their design was based on an older GPU generation due to improvements in the baseline GPU model, the CTA-aware scheduler is not as effective as previously reported, achiving only a 5% speedup overall.

MASCAR is another warp scheduling technique that tries to prioritize one warp to keep executing whenever the GPU hits a long latency stall. MASCAR introduces a cache re-execution model to reduce memory pressure and increase the cache hit rate. MASCAR provides an overall speedup of 7.7% on average. We can observe that MASCAR performs better on cache sensitive benchmarks such as BS and MVT, since the cache re-execution fully utilizes temporal locality which can reduce memory pressure.

Our hint-assisted locality-aware scheduler outperforms both CTA-aware and MASCAR. In all of the benchmarks, it achieves a maximum performance improvement of 19% in the MT benchmark. From the overall performance, we can also see that our hint-assisted locality-aware scheduler performs even better on the benchmarks that have high spatial locality (type 1 and type 2). It achieves a 13% speedup on average. The reason for this speedup is that many of the benchmarks have high spatial locality, so the hint-assisted locality-aware scheduler can issue many more memory requests from different warps, heavily reducing memory bandwidth pressure. Although MASCAR and CTA aware can also provide performance improvements, they can not alleviate the memory pressure by as much as the hint-assisted locality-aware scheduler. Besides, our approach requires minimal hardware overhead.

For the benchmarks that have low to medium spatial locality, the speedup achieved by the hint-assisted locality-aware scheduler varies across kernels. For the MVT benchmark which calculates a matrix-vector production, work is divided across different blocks. Each block reads a different part of the input matrix and different chunk of the input vector, so there is little spatial locality among adjacent blocks. That is the reason why MASCAR outperforms hint-assisted locality-aware scheduler for this workload.

6.4.3 Reduction in Duplicated Cache Line Allocations

As discussed previously in this chapter, data locality across inter-blocks generates a high number of duplicate MSHR allocations in different SMs. Figure 6.11 shows the percentage of MSHRs allocated for duplicated cache lines for hint-assisted locality-aware scheduler. We can see that for all benchmarks except MT, the number of duplicate MSHR entries is reduced as compared with the baseline model. The reason why there is no reduction in MT benchmark is that little data locality exists among blocks. For benchmarks with high spatial locality, we observe at least a 10% reduction on duplicate MSHR entries. This further explains how our hint-assisted locality-aware scheduler reduces MSHR pressure and improves overall performance.

6.4.4 Number of Memory Requests per MSHR Entry

As shown in Table 6.2, in our baseline model each MSHR entry can insert up to 8 warp requests for the same cache line. If we take a closer look at Figure 6.12, we can see that the number of memory requests each MSHR entry is handling is increased in all benchmarks when using the hint-assisted locality-aware scheduler, as compared with the baseline model. For some benchmarks, the number of memory requests handled doubles for the hint-assisted locality-aware scheduler, such as MT and SC. In the MT benchmark, high data locality exists among blocks, so the hint-assisted warp scheduler increases the MSHR efficiency dramatically. By increasing the number of entries

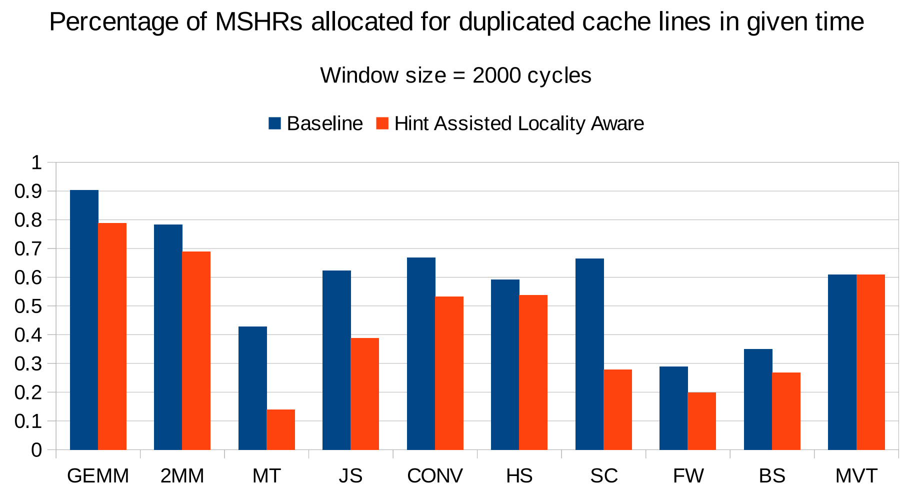


Figure 6.11: Percentage of MSHRs allocated for duplicated cache lines in every 2000 cycles, relative to the baseline model.

used in each MSHR, the hint-assisted locality-aware scheduler reduces the memory pressure and increase GPU performance.

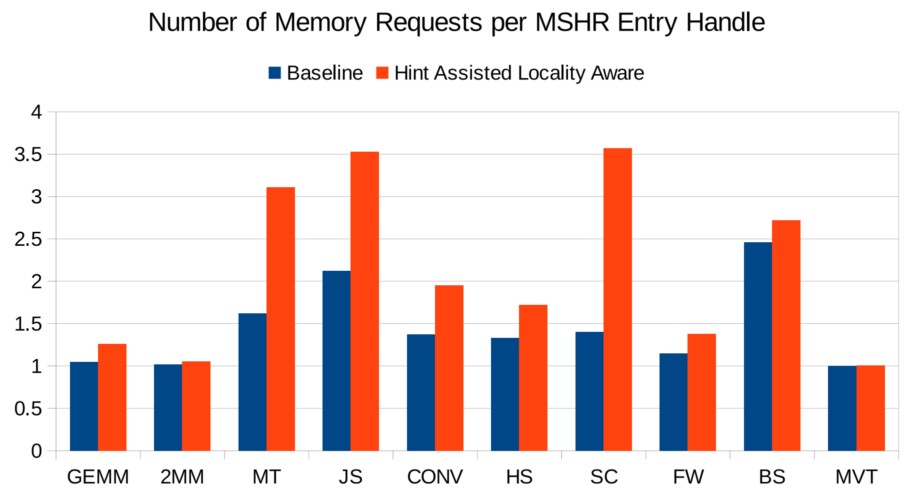


Figure 6.12: Number of memory requests per MSHR entry handle.

6.4.5 Summary

In this chapter, we have presented the hint-assisted locality-aware warp scheduler, a novel GPU scheduler which uses programmer-guided hints to guide the warp scheduler to make smarter decisions. By adopting different scheduling algorithms, grouping different warps together according to scheduling hints, the hint-assisted locality-aware scheduler can provide a 13% speedup for workloads with the high spatial locality.

Chapter 7

Conclusion

7.1 Conclusion

In this thesis, we have focused on addressing improving GPU performance and resource utilization leveraging a hardware/software co-design approach. To summarize, this thesis makes the following contributions:

- We have developed a cycle-based GPU simulator based on the NVIDIA Kepler GPU architecture running at the SASS level (binary level). We have demonstrated its accuracy as compared with GPU hardware. Due to the fact that the NVIDIA architecture is proprietary, we reverse-engineered the NVIDIA Kepler GPU ISA, architectural details, as well as driver and runtime information. We have developed our own CUDA runtime and driver APIs, as well as the GPU driver calls to support GPU simulation with our framework. To our best of knowledge, this is the first GPU simulator that supports NVIDIA GPU on the SASS level.
- Leveraging our simulator, we characterized a rich set of workloads to identify the sources of poor resource utilization. We identified that long latency memory stalls are a major source of low resource utilization, especially for memory-intensive workloads. We have designed a novel hint-assisted warp scheduling policy to improve GPU performance by executing independent instructions in a selective out-of-order fashion when the program encounters long latency stalls, making better use of compute resources.
- Long memory latencies and limited memory bandwidth are still significant bottlenecks in GPU applications. For benchmarks that have inherent spatial locality, we have designed a hint-assisted locality-aware warp scheduling policy to exploit spatial locality to reduce memory pressure. First, we extended the CUDA runtime API to support the use of scheduling hints.

Second, based on the availability of different scheduling hints, our hint-assisted locality-aware scheduler can make better decisions about grouping warps which share high spatial locality to increase MSHR efficiency and reduce memory pressure. Our hint-assisted locality-aware scheduler improves GPU performance, while adding a modest amount of hardware.

7.2 Future Work

Our focus on a hardware/software co-design scheduling approach has enabled optimization opportunities for future GPU architecture researchers. Since the GPU is a power-efficient device, we believe that additional co-design scheduling techniques will be explored to further reduce energy consumption on GPUs. One exciting direction would be to insert hardware-specific latency information in the hint when scheduling warps. In this case, the GPU will know the precise timing for a warp, which will guide the scheduling of arithmetic instructions. The GPU inactive this warp and give other warps to run to improve performance.

For our hint-assisted locality-aware warp scheduler, we develop customized scheduling hints based on the type of workload. To better optimize the scheduling algorithm, further opportunities can be exploited by adding workload specific parameters in the hint. For example, we can add information to the hint that identifies boundary cache lines to assist the scheduler to make a smarter decisions. We also believe that future co-designed approaches can be optimized for power versus just performance, which would benefit embedded and edge devices.

Bibliography

[1] *NVIDIA Tesla K20X GPU Accelerator*, 2013. `http://www.nvidia.com/content/pdf/kepler/tesla-k20x-bd-06397-001-v07.pdf`.

[2] *CUDA Driver API, v8.0*, 2016. `http://docs.nvidia.com/cuda/pdf/CUDA_Driver_API.pdf`.

[3] *CUDA Runtime API, v8.0*, 2016. `http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf`.

[4] AMD. *Evergreen Family Instruction Set Architecture*, 2012. `https://developer.amd.com/wordpress/media/2012/10/AMD_Evergreen-Family_Instruction_Set_Architecture.pdf`.

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

[6] K. Beyls and E. D'Hollander. Compile-time cache hint generation for epic architectures. In *2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC-2)*, 2002.

[7] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Comp. Arch. News*, 25(3):13–25, 1997.

[8] L. Chen and G. Agrawal. Optimizing mapreduce for gpus with effective shared memory usage. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 199–210. ACM, 2012.

[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[10] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A parallel functional simulator for gpgpu. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 351–360. IEEE, 2010.

[11] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 9. ACM, 2011.

[12] W. W. Fung and T. M. Aamodt. Thread block compaction for efficient simt control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 25–36. IEEE, 2011.

[13] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420. IEEE Computer Society, 2007.

[14] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 235–246. IEEE, 2011.

[15] X. Gong, Z. Chen, A. K. Ziabari, R. Ubal, and D. Kaeli. Twinkernels: an execution model to improve gpu hardware scheduling at compile time. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 39–49. IEEE Press, 2017.

[16] X. Gong, R. Ubal, and D. Kaeli. Multi2sim kepler: A detailed architectural gpu simulator. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 269–278. IEEE, 2017.

[17] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram. Gpu register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 420–432. ACM, 2015.

[18] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang. An energy-efficient and scalable edram-based register file architecture for gpgpu. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 344–355. ACM, 2013.

[19] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Owl: cooperative thread array aware scheduling techniques for

improving gpgpu performance. In *ACM SIGPLAN Notices*, volume 48, pages 395–406. ACM, 2013.

[20] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated scheduling and prefetching for gpgpus. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 332–343. ACM, 2013.

[21] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

[22] C. Kalra, X. Gong, R. Ubal, and D. Kaeli. Lcpc 2015.

[23] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 157–166. IEEE Press, 2013.

[24] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram. Warped-preexecution: A gpu pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 163–175. IEEE, 2016.

[25] J. Kloosterman, J. Beaumont, M. Wollman, A. Sethia, R. Dreslinski, T. Mudge, and S. Mahlke. Warppool: sharing requests with inter-warp coalescing for throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 433–444. ACM, 2015.

[26] G. Kucuk, D. Ponomarev, and K. Ghose. Low-complexity reorder buffer architecture. In *Proceedings of the 16th international conference on Supercomputing*, pages 57–66. ACM, 2002.

[27] G. Kyriazis. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit*, page 21, 2012.

[28] N. B. Lakshminarayana and H. Kim. Effect of instruction fetch and memory scheduling on gpu performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, volume 88. Citeseer, 2010.

[29] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 260–271. IEEE, 2014.

[30] S.-Y. Lee and C.-J. Wu. Caws: criticality-aware warp scheduling for gpgpu workloads. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 175–186. ACM, 2014.

[31] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2), 2008.

[32] F. Luna. *Introduction to 3D game programming with DirectX 11*. Stylus Publishing, LLC, 2012.

[33] M. Mantor. Amd radeon hd 7970 with graphics core next (gcn) architecture. In *Hot Chips 24 Symposium (HCS), 2012 IEEE*, pages 1–35. IEEE, 2012.

[34] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li. An energy-efficient gpgpu register file architecture using racetrack memory. *IEEE Transactions on Computers*, 2017.

[35] S. A. McKee. Reflections on the memory wall. In *Proc. of the 1st conference on Computing Frontiers*, page 162. ACM, 2004.

[36] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, 2003.

[37] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *ACM SIGARCH Computer Architecture News*, 38(3):235–246, 2010.

[38] A. Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[39] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, pages 22–31, 2009.

[40] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317. ACM, 2011.

[41] T. Ni. Direct compute: Bring gpu computing to the mainstream. In *GPU Technology Conference*, page 23, 2009.

[42] Nvidia. *NVIDIA CUDA Toolkit*, 2007. https://developer.nvidia.com/cuda-downloads/.

[43] Nvidia. 2009. https://www.nvidia.com/object/gpu-applications-domain.html.

[44] Nvidia. 2009. https://developer.nvidia.com/gpu-accelerated-libraries.

[45] Nvidia. *NVIDIA Fermi Compute Architecture Whitepaper*, 2009. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[46] Nvidia. *NVIDIA Kepler GK110 Architecture Whitepaper*, 2012. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[47] Nvidia. *NVIDIA GeForce GTX 980*, 2014. https://www.techpowerup.com/gpu-specs/docs/nvidia-gtx-980.pdf.

[48] Nvidia. *NVIDIA CUDA C Programming Guide, v8.0*, 2016. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[49] Nvidia. *NVIDIA PTX:Parallel Thread Execution ISA, v5.0*, 2016. http://docs.nvidia.com/cuda/parallel-thread-execution/.

[50] Nvidia. *NVIDIA Tesla P100 Architecture Whitepaper*, 2016. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[51] Nvidia. *NVIDIA TESLA V100 GPU ARCHITECTURE*, 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[52] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram. Apres: improving cache efficiency by exploiting load characteristics on gpus. *ACM SIGARCH computer architecture news*, 44(3):191–203, 2016.

[53] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. Kim. Design and performance evaluation of image processing algorithms on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):91–104, 2010.

[54] L.-N. Pouchet and S. Grauer-Gray. Polybench: The polyhedral benchmark suite (2011). *URL http://www-roc. inria. fr/~ pouchet/software/polybench*, 2015.

[55] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2014.

[56] M. Rhu and M. Erez. The dual-path execution model for efficient gpu control flow. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 591–602. IEEE, 2013.

[57] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler. A variable warp size architecture. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 489–501. ACM, 2015.

[58] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[59] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 99–110. ACM, 2013.

[60] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[61] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC bioinformatics*, 8(1):474, 2007.

[62] M. S. Schlansker and B. R. Rau. Epic: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[63] A. Sethia, D. A. Jamshidi, and S. Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 174–185. IEEE, 2015.

[64] T. S. Sørensen, T. Schaeffter, K. Ø. Noe, and M. S. Hansen. Accelerating the nonequispaced fast fourier transform on commodity graphics hardware. *IEEE Transactions on Medical Imaging*, 27(4):538–547, 2008.

[65] S. S. Stone, H. Yi, W. Hwu, J. Haldar, B. P. Sutton, and Z.-P. Liang. How gpus can improve the quality of magnetic resonance imaging. In *The First Workshop on General Purpose Processing on Graphics Processing Units*, pages 39–55, 2007.

[66] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[67] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344. IEEE, 2012.

[68] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi. Simd divergence optimization through intra-warp compaction. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 368–379. ACM, 2013.

[69] W.-M.Hwu. *GPU Computing Gems Emerald Edition*. Elsevier, 2011.

[70] B. Wang, W. Yu, X.-H. Sun, and X. Wang. Dacache: Memory divergence-aware gpu cache management. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 89–98. ACM, 2015.

[71] B. Wang, Y. Zhu, and W. Yu. Oaws: memory occlusion aware warp scheduling. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 45–55. IEEE, 2016.

[72] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 199–208. IEEE, 2002.

[73] S. Wienke, P. Springer, C. Terboven, and D. an Mey. Openaccfirst experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[74] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[75] F. Xu and K. Mueller. Real-time 3d computed tomographic reconstruction using commodity graphics hardware. *Physics in Medicine & Biology*, 52(12):3405, 2007.

[76] Y. Yu, W. Xiao, X. He, H. Guo, Y. Wang, and X. Chen. A stall-aware warp scheduling for dynamically optimizing thread-level parallelism in gpgpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 15–24. ACM, 2015.

[77] K. Zhao and X. Chu. G-blastn: accelerating nucleotide alignment by graphics processors. *Bioinformatics*, 30(10):1384–1391, 2014.