2017

Workload-aware Scheduling Techniques for General Purpose Applications on Graphics Processing Units

Mihir Awatramani
Iowa State University

Workload-aware scheduling techniques for general purpose applications on graphics processing units

by

Mihir Awatramani

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Diane Rover, Co-major Professor
Joseph Zambreno, Co-major Professor
Arun Somani
Zhao Zhang
Glenn Luecke

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2017

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to those who guided and supported me through the years of my doctoral research. First and foremost, I would like to thank Dr. Diane Rover and Dr. Joseph Zambreno. Putting their faith in me gave me the courage and inspiration that I greatly needed in the initial years of my graduate career. Throughout the course of my graduate studies, their continued patience, guidance and encouragement gave me the support I required to persevere through my doctoral research.

I would also like to thank my committee members, Dr. Zhao Zhang, Dr. Arun Somani and Dr. Glenn Luecke, for taking a keen interest in my work and providing their support through the years. Additionally, I would like to thank Dr. Zhao Zhang, Dr. Phillip Jones and Dr. Joseph Zambreno for their teachings in advanced computer architecture, and Dr. Diane Rover for her teachings in systems design, which helped ignite my passion for the field and guided me to find the right path in the initial years of my graduate school.

Lastly, I would like to thank my parents, my brother and his family, and my fiance Nikita Chopra. Their continued love, patience, support, and words of encouragement has helped me throughout my graduate education.

ABSTRACT

In the last decade, there has been a wide scale adoption of Graphics Processing Units (GPUs) as a co-processor for accelerating data-parallel general purpose applications. A primary driver of this adoption is that GPUs offer orders of magnitude higher floating point arithmetic throughput and memory bandwidth compared to their CPU counterparts. As GPU architectures are designed as throughput processors, they adopt a manycore architecture with 10 to 100s of cores, each with multiple vector processing pipelines. A significant amount of the die area is dedicated to floating point units, at the expense of not having hardware units used for memory latency hiding in conventional CPU architectures. The quintessential technique used for memory latency tolerance is exploiting data-level parallelism in the workload, and interleaving execution of multiple SIMD threads, to overlap the latency of threads waiting on data from memory with computation from other threads.

With each architecture generation, GPU architectures are providing an increasing amount of floating point throughput and memory bandwidth. Alongside, the architectures support an increasing number of simultaneously active threads. We envision that to continue making advancements in GPU computing, workload-aware scheduling techniques are required. In the GPU computing work flow, scheduling is performed at three levels - the system or chip level, the core level and the thread level. The work proposed in the research aims at designing novel workload aware scheduling techniques at each of the three levels of scheduling. We show that GPU computing workloads have significantly varying characteristics, and design techniques that monitor the hardware state to aide at each of the three levels of scheduling. Each technique is implemented in a cycle level GPU architecture simulator, and their effect on performance is analyzed against state of the art scheduling techniques used in GPU architectures.

CHAPTER 1. INTRODUCTION

In the last decade, the adoption of Graphics Processing Units (GPUs) for accelerating general purpose applications has grown at a rapid pace [33, 35, 50]. The role of GPUs has shifted from being a co-processor for graphics rendering, to a general purpose accelerator for data-parallel workloads [58, 57]. Three reasons can be identified as primary drivers for this rapid growth.

First, the last decade has seen a tremendous growth in the number of highly data-parallel applications, which typically have high computational throughput requirements. Examples can be found in several domains [74] ranging from computational physics [62, 19], medical sciences [61, 70], molecular dynamics [69, 6], computational biology [65] and big data analytics [26]. Currently, a third of the systems in the top 500 supercomputer list use GPUs as accelerators [4]. A clear indication that the growth of GPU computing in the HPC domain would continue to increase in the near future is, the next three supercomputers funded by the U.S. Department of Energy's CORAL project employing GPUs [38, 56] or similar many-core architectures [36]. Moreover, with the rise of deep learning, GPU computing is expected to grow into several other domains, like automobiles, robotics and urban development [1].

Secondly, with the end of Denard scaling [17, 11] towards the middle of the last decade, improvements in single processor performance from frequency scaling ceased and CPU processor architectures shifted from single-core to multi-core designs [18]. However CPU cores are designed for low latency, and thus adopt an architecture with large caches and speculative out-of-order cores [28]. Their computational throughput does not scale to the demands required for the aforementioned high-throughput applications. On the contrary, as graphics rendering workloads have high computational throughput demands and relatively lower latency requirements, GPU architectures employ a many-core architecture and operate at lower frequencies with a

significantly higher number of in-order cores [55]. Consequently GPUs provide a significantly higher throughput and memory bandwidth compared to multi-core CPUs, and serve as a natural fit for high-throughput data-parallel workloads.

Lastly, towards the middle of the last decade, GPU architectures started a shift from fixed-function graphics rendering pipelines to programmable shaders [2]. While initial results of using GPU programmable shaders for data-parallel workloads showed significant speedups [13, 66], their adoption was limited due to the difficulty in programming them. With the development of languages like CUDA [12, 34, 49] and OpenCL [47], GPU computing became more widespread, and continues to grow in adoption. This has led to the emergence of a new parallel computing paradigm for executing general purpose applications on GPUs, referred to as GPGPU.

1.1 Motivation

GPU architectures are designed specifically for high throughput computing. For example, the current state of the NVIDIA GPU, the Volta V100, supports a peak double precision floating point throughput of 7024 GFLOPs and a peak memory bandwidth of 807 GB/s [55]. As a comparison, the state of the art Intel CPU, the Xeon E7 supports a peak double precision floating point throughput of 844 GFLOPs and a peak memory bandwidth of 102 GB/s [28]. To achieve such high computational throughputs, GPUs dedicate a large portion of their die area to cores and arithmetic units. As a trade-off, GPU chips do not include the hardware units which have been traditionally used for latency hiding in CPUs. For example, GPUs use a non-speculative in-order pipeline, do not have branch predictors, and have much smaller caches compared to CPUs. As a consequence, GPU architectures typically have an order of magnitude larger static memory load latencies, with dynamic latencies even larger due to the fact that 10s to 100s of cores are executing in parallel.

The primary mechanism to hide the large memory access latencies in GPUs is to perform massive multithreading in hardware. State of the art GPUs support about two thousand scalar active threads per core, and as high as 80 cores per chip [55]. GPGPU programming models divide the workload into three granularities. Consequently, threads are scheduled via three distinct hardware schedulers, each working independently at a different workload granularity.

Each function to be accelerated, is offloaded by the CPU as an independent program referred to as a kernel. A ***system-level kernel scheduler*** dictates the scheduling order of the different kernels launched by the CPU and maps each kernel to a set of GPU cores. The programming model groups threads of a kernel into an abstraction called called Thread Blocks (TBs). Once each kernel is assigned its respective core by the kernel-level scheduler, a ***core-level work distributor*** issues TBs from each kernel to the cores it has been mapped to. Lastly, threads in a thread block are grouped as SIMD units called warps [54] (or wavefronts [47]). Consequently, on each core, a ***warp-level scheduler*** interleaves execution of warps launched on a core and tries to overlap the latency of warps waiting on long latency operations with computation from other warps.

Efficient scheduling of the SIMD threads in hardware has a direct impact on the performance of GPGPU applications. However, due to the extremely high thread counts, efficient scheduling of these threads is non trivial. Moreover, with more general purpose workloads being offloaded to the GPU for acceleration, workload agnostic scheduling techniques that work well for a subset of applications, might fail for others. The motivation of this research is to develop efficient scheduling techniques by using information regarding the workload being executed, and the architecture runtime state. It is important to note that although the overarching goal of the three schedulers is to increase overall throughput, their individual objectives differ considerably as each of them operates at a different workload abstraction. Consequently, this work aims to develop workload-aware scheduling techniques at each of the three scheduling granularities.

***Thesis Statement*:** *Thread scheduling techniques play a pivotal role in GPUs to enable harnessing the high floating point and memory throughput that the architectures support. In this work, we design efficient hardware scheduling techniques at each of the scheduling granularities in the GPU architecture. We propose that scheduling techniques should be aware of the workload characteristics, and the architecture state at runtime to maximize the benefits gained from thread level parallelism. Information about workload characteristics is derived at compile time, and the architectural state is monitored via hardware counters at runtime. This information is used at all levels of scheduling to improve scheduler efficiency and maximize overall throughput.*

1.2 Thesis Contributions

GPU computing programming models divide the work offloaded to the GPU into a three level hierarchy. Scheduling is performed by hardware units that operate independently at the three levels. In this section, we describe the specific contributions proposed by this research at each of the three levels of scheduling.

1.2.1 Level 1: Intra-core Thread Scheduler

The primary mechanism to hide memory access latency in GPU architectures is to exploit data parallelism in the workload and overlap latency of memory accesses with computation. The context of all threads launched on each core of the GPU is kept live in the respective core's register file. This enables the thread-level hardware scheduler on each core, called the warp scheduler, to switch among them with low overhead. This scheduler is pivotal in being able to achieve a throughput which is close to peak, as it largely affects how well the computation and memory latency is overlapped. In this work, we show that the performance of existing state-of-the-art warp schedulers is highly dependent on what we refer to as kernel phase behavior [9]. Specifically, the instruction stream of a GPGPU kernel has blocks of compute instructions separated by long latency operations, which we refer to as phases. We thoroughly analyze phase behavior in GPGPU kernels and demonstrate that performance of state-of-art thread schedulers is affected by characteristics of these phases, which vary significantly across kernels. To this end, we design a compiler assisted warp scheduling policy that is aware of kernel phase behavior. The instruction stream of the kernel is analyzed at compile time and information regarding phases is inserted in program instructions. The hardware warp scheduler on the core uses this information to make scheduling decisions. We demonstrate that the phase-aware warp scheduling policy is more robust to kernel phase behavior compared to the existing state of the art warp schedulers.

1.2.2 Level 2: Inter-core Thread Block Scheduler

This scheduler operates at a granularity above the warp scheduler, and is responsible for issuing thread blocks to the GPU cores. The primary goal of the thread block scheduler is to keep each core as occupied as possible, by launching threads until the core resources are exhausted. The rationale is, a higher number of threads would give the warp scheduler on the core more opportunities to overlap memory latency and computation, thus resulting in better performance. Consequently, with each architectural generation, as the peak compute throughput and memory bandwidth supported by GPUs continue to increase, each generation supports an increasing number of warps to increase thread level parallelism. For example, the previous generation NVIDIA Pascal chip, the P100, has a peak FP32 throughput of 4.9 TFLOPs, a peak memory bandwidth of 720 GB/s, and supports concurrent execution of 3584 warps [52]. In comparison, the current generation chip, the V100, supports a peak FP32 throughput of 1.4x, a peak memory bandwidth of 1.2x and supports concurrent execution of 1.4x more warps [55].

While increasing thread level parallelism improves performance of latency-limited workloads, the same is not true if the performance of a workload is throughput-limited. We demonstrate that once a workload reaches its peak achievable memory or compute throughput, increasing number of active threads does not improve performance. Consequently, these kernels can be executed with fewer than maximum number of threads without affecting performance. We refer to the number of warps at which performance of a GPU kernel saturates as the kernel's optimal thread count. Executing fewer than the maximum threads reduces the hardware resources required by the kernel, and enables opportunities for reducing energy consumption by power-gating the unused resources. In this work, we propose two hardware techniques to detect the optimal thread count of GPU kernels at runtime:

1.2.2.1 Executing optimal number of warps on each core

Increasing the number of warps executing on a core decreases the time spent by a kernel waiting on stalls due to data hazard. At the same time, increasing number of warps might

increase the number of pipeline stalls due to hardware resource contention. In this work, we analyze the effect of number of active warps on the breakdown of the warp scheduler activity on each core and demonstrate the above effect [8]. We then propose a hardware mechanism, called Perf-Sat, which monitors the scheduler activity on each core at runtime, and detects the optimal thread count independently at each core. The optimal thread count detected by Perf-Sat is used by the thread block scheduler to limit the number of active thread blocks on each core. Our results show that with a performance loss of less than 1%, Perf-Sat is able to achieve core resource savings of 18.32% on average.

1.2.2.2 Executing the kernel on optimal number of cores

An alternative technique (compared to executing optimal number of threads on each core) is to execute the kernel on fewer than maximum core available on the chip. We demonstrate that for memory-throughput limited kernels, executing the workload on fewer than maximum available cores does not significantly impact performance [79]. At the same time, power gating the unused cores results in significant energy savings. We then propose ONAC (Optimal Number of Active Cores detector), a hardware mechanism that detects the optimal number of active cores at runtime. ONAC uses an estimation model inspired by Roofline [76], and estimates the effect of number of active cores on the chip's average IPC. Our results show that, for memory-intensive kernels, ONAC reduces energy consumption by 20% with negligible impact on performance.

1.2.3 Level 3: System-wide Kernel Scheduler

The kernel level scheduler operates at the third and highest level of scheduling on a GPU chip. It manages binding of kernels to cores on which they would be executed. In GPU architectures, concurrently launched kernels are scheduled sequentially unless kernels do not have enough threads to occupy all cores. Moreover, when kernels are scheduled concurrently, they are executed on separate cores. As mentioned in the previous subsection, the number of threads required by a kernel to achieve peak throughput is workload dependent, and throughput saturating workloads can be executed with a lower thread count without affecting performance.

Executing fewer than the maximum number of warps supported by an architecture frees hardware resources, and opens up opportunities for multi-tasking. Sharing GPU cores among threads from multiple kernels is a relatively new technique, with its own set of challenges [7]. Most previous works have focused primarily on resource partitioning techniques that determine how many cores should be assigned to each kernel. In this research, we focus on the effect of warp scheduling policies in concurrent kernel scheduling scenarios. We demonstrate that kernels limited by complementing resources (memory bandwidth and floating point throughput) show significant speedups when executed concurrently. Our results show that interference among threads from kernels concurrently executing on GPU cores has a performance overhead that effects the gains achieved from concurrent kernel execution.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: in chapter 2, we provide background on the GPU hardware architecture and programming model. Next, we discuss in detail the three levels of scheduling, namely the intra-core warp scheduler, the inter-core thread block scheduler, and the system-wide kernel scheduler, in chapters 3, 4, and 5 respectively. Each of the three chapters are organized as a. A background of current state of the art, b. A discussion of the issues that motivate our work, c. A description of our contributions, followed by an in-depth analysis of the experimental results, and d. A discussion of the related published works at the respective granularity of scheduling. Chapter 7 concludes the thesis by providing a summary of the conclusions, lessons learned and directions for future work.

CHAPTER 2. BACKGROUND

In this chapter we provide an overview of the programming model used for General Purpose computing on Graphics Processing Units (GPGPU) and the GPU hardware architecture. Both, the GPGPU programming model and the GPU compute hardware architecture have been described extensively in published literature. Consequently, this chapter provides only a brief introduction of the concepts, with more focus on the scheduling work flow in GPU computing. Further details regarding the GPU hardware architecture can be found in [55, 10, 35, 48, 21, 20, 22]. Further details regarding the concepts related to the GPGPU programming model can be found in [34, 12, 47, 54, 51, 74].

2.1 The GPGPU Programming Model

In GPGPU programming models, the GPU is used as a co-processor to the CPU. Sequential portions of the application are executed on the CPU (referred to as the *host*), and parallel portions of the application are executed on the GPU (referred to as the *device*). The two most widely used GPGPU programming languages are OpenCL [47] and CUDA C [54]. To evaluate our work, we used applications written in CUDA C. However, the concepts are directly applicable to applications written in OpenCL as well.

The CUDA programming model provides constructs that expose thread-level parallelism and data-level parallelism to the programmer on top of other high level languages. Several languages are supported, like C, C++, Fortran, Python and Java. In CUDA C, the portion of the program executing on the CPU is written in C, and the language provides interfaces (APIs: Application Programming Interfaces) to the programmer to illustrate GPU related functions. A few examples of the APIs provided are, interfaces to query specifications of the

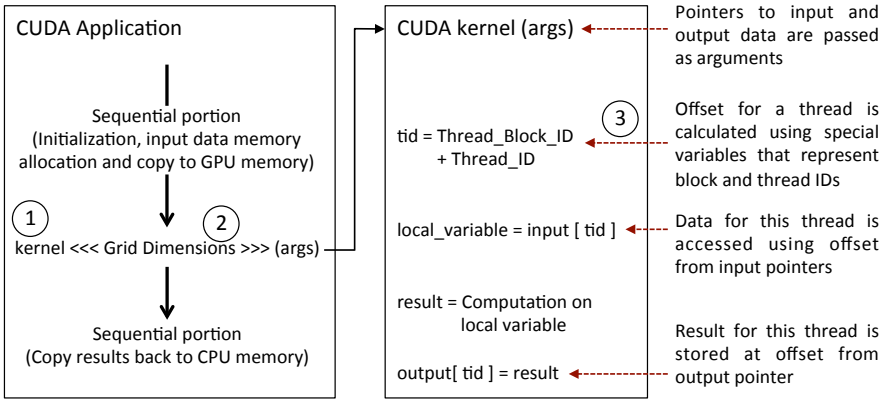


Figure 2.1: Depiction of basic structure of a GPGPU program with one kernel.

GPU (device) on the system (cudaGetDeviceProperties), to allocate/deallocate data on the device (cudaMalloc/cudaFree), to transfer data between the CPU (host) and device memories (cudaMemcpy), to launch a program (referred to as a kernel) on the device (refer to Fig. 2.1), and APIs that provide synchronization between the host and the device [64, 49, 54] . The application is compiled into a single binary; the sequential portions execute on the CPU, and GPU related functions are executed by the GPU driver (refer to Fig. 2.3).

Fig. 2.1 depicts the basic structure of a CUDA application. As mentioned previously, portion of the application to be executed on the GPU is written as a separate function, called a kernel ①. The data which would be operated on by the threads of a kernel is allocated and copied to GPU memory before the kernel is launched. Similarly, the results are copied back from GPU memory after the kernel finishes. The function launch syntax specifies the total number of threads that would execute this kernel within the <<< >>> structure (Fig. 2.1 ②).

In CUDA, workload for a kernel is described using a two level hierarchy. Fig. 2.2 provides a depiction of this hierarchy via examples of two CUDA kernel launches. The total number of threads launched for an invocation of the kernel is referred to as a grid. Number of threads in a grid are described within the <<< >>> structure as *(Number of thread blocks, Number of threads in each thread block)*. Threads within a grid are divided into groups of threads, called thread blocks (or CTAs: Cooperative Thread Arrays). The programming model allows for threads within a thread block to use synchronization barriers and share data using an on-chip SRAM. Number of threads in each thread block are fixed for a given kernel launch. In Fig. 2.2, kernels

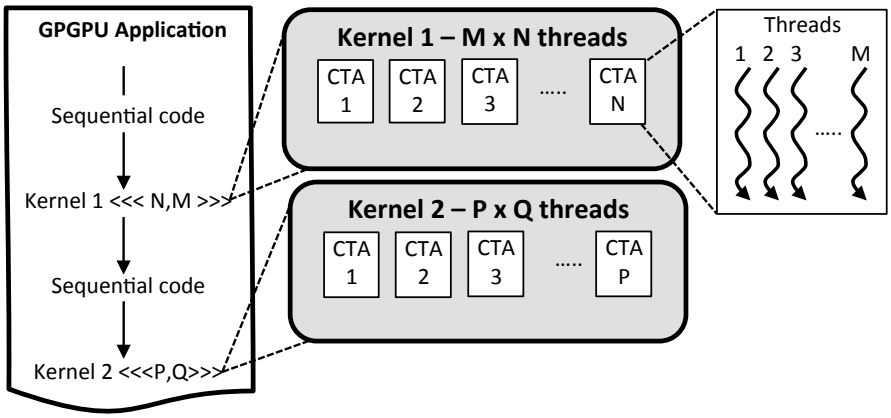


Figure 2.2: Depiction of the thread hierarchy for workload description of a CUDA C kernel.

1 and 2 launch N CTAs of M threads, and P CTAs of Q threads respectively. Each thread executes the kernel instructions on its respective data, resulting in a Single Program Multiple Data (SPMD) programming model. Figure 2.1 ③ depicts how the block and thread identifiers are used within the kernel to index data specific to a thread from GPU memory. In the next section, we describe how the three level of thread abstractions: the grid, CTAs and threads, are scheduled on the GPU hardware. We encourage the reader to refer to [54, 47] for more details regarding the programing model.

2.2 Overview of the Hardware Architecture and Scheduling in GPGPU

In this section, we provide an overview of the GPU hardware architecture, with a focus on the flow of scheduling in GPGPU. Fig. 2.3 depicts a simplistic block diagram of a single GPU system, with two GPGPU applications executing on the CPU. Applications 1 and 2 launch two and one kernels respectively. Launching of the kernels, as well as other GPU compute APIs (refer to Sect. 2.1) are executed via the GPU driver. Data is transfered between the CPU and GPU using the PCI express or an equivalent bus [52].

At a high level, GPUs have a manycore architecture with several in-order SIMD cores called SMs, or Streaming Multiprocessors (Fig. 2.3). Each SM consists of several arithmetic vector execution units (single and double precision floating point units, integer math units, load/store units and special function units used for operations like sine and cosine). Threads launched on a

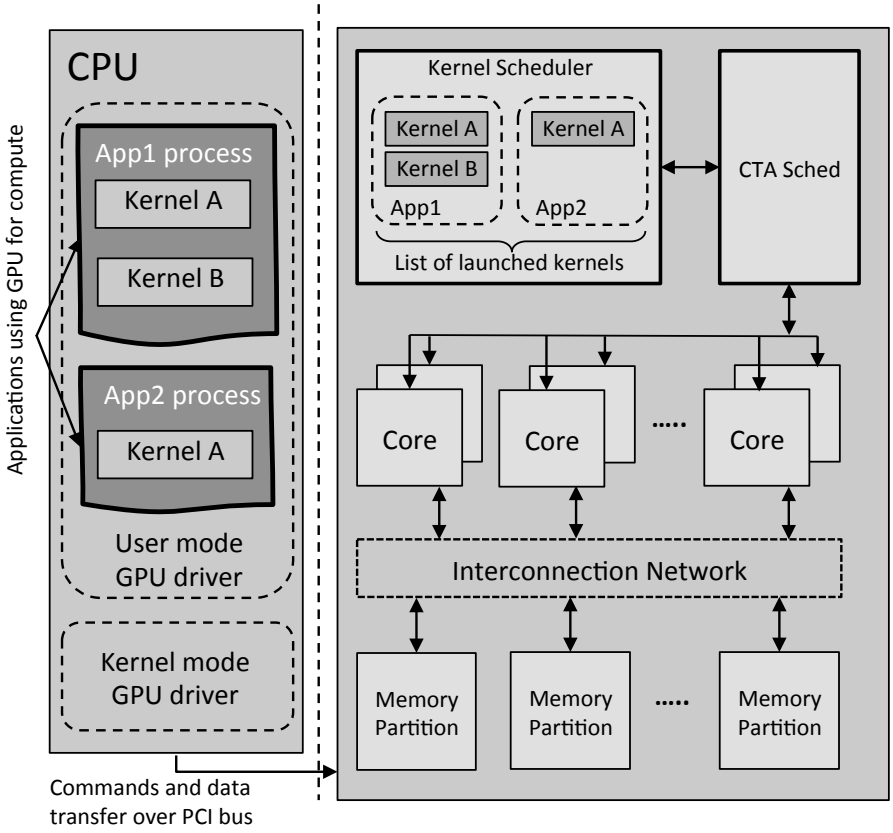


Figure 2.3: Block diagram depicting the flow of GPU compute work launch.

GPU core execute instructions on the vector units in Single Instruction Multiple Data (SIMD) fashion. For example, the current HPC chip from NVIDIA, the V100, has 80 SMs, each with four 16-wide SIMD datapaths with 512 bit wide execution units [55]. As context of a all threads launched on the SMs is kept active, each SM has a large register file. Contrary to the CPU architecture, GPUs have a larger register file capacity compared to data cache. For example, the P100 and V100 GPUs have 256 KB of register file per SM, and a total of 14 MB and 20 MB of register file capacity across the chip. In comparison, they have an L2 data cache of 4 MB and 6 MB respectively [52, 55]. Additionally, each SM has a L1 data cache and a high-bandwidth on-chip memory that can be shared by threads from the same thread block. If a memory request misses the L1 data cache, it is routed via the interconnection network to a one of the memory partitions. Each memory partition has a L2 cache bank and a memory controller. GPU architectures typically provide extremely high memory bandwidths. For example, the Fermi and Kepler architecture configurations used in our work have a peak bandwidth of 177

GB/s and 250 GB/s respectively [53, 3]. The more recent P100 and V100 architectures support peak memory bandwidths of 715 GB/s and 877 GB/s respectively [52, 55].

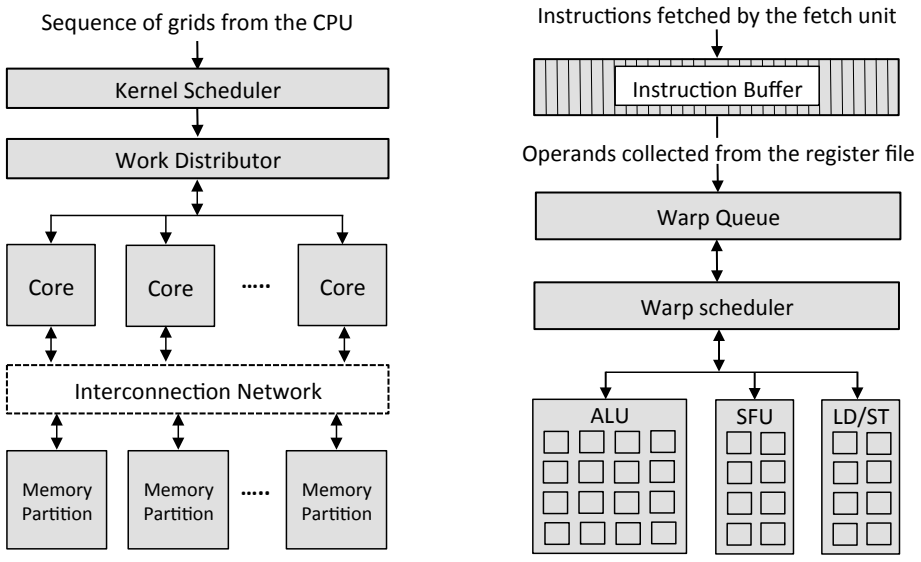
In the previous section, we described how the GPGPU programming model divides the workload of a GPU kernel into three distinct granularities: the gird, thread block and threads. Consequently, scheduling of threads is performed on the GPU hardware at three levels of granularities as well.

2.2.1 Kernel Scheduler

Fig. 2.3 depicts two hardware units on the GPU that manage launching of work from the kernels that are active on the chip: the kernel scheduler, and the CTA scheduler. As mentioned in Sect. 2.1, GPU compute applications launch parts of the computation that are data-parallel as kernels on the GPU. The kernel level scheduler maintains a list of all the kernels currently active on the chip, and manages the assignment of cores to each kernel. It also manages the interactions with the CPU side driver. Traditionally, context of only one kernel was kept active on the GPU at a given time. More recently, as GPUs have become larger, there has been a growing interest in executing multiple kernels concurrently. In these scenarios, the kernel level scheduler also performs resource partitioning across kernels that are concurrently active on the chip. Previous works have analyzed resource partitioning extensively, and have proposed runtime techniques to find optimal resource partitioning across concurrently active kernels [59, 72, 77].

2.2.2 CTA Scheduler

Once the kernel-to-core mappings have been assigned by the kernel scheduler, the CTA scheduler manages launching of work from the active kernels onto the cores they have been assigned (Fig. 2.4(a)). As described in Sect. 2.1, threads in a grid are divided into groups called CTAs. As threads within a CTA can share data using the on-core SRAM, all threads in a CTA are scheduled on the same core. Additionally, as threads in a CTA can use synchronization primitives, all threads in a CTA have to be issued at the same time. Consequently, thread blocks or CTAs are the smallest granularity at which work is issued to a GPU core.



(a) Chip level block diagram. (b) Core block diagram.

Figure 2.4: Overview of GPU architecture.

The primary goal of the CTA scheduler is to keep each core as full as possible, while maintaining a balanced workload across all cores. The scheduling policy used by the CTA scheduler is round-robin: CTA 1 is issued to SM 1, CTA 2 to SM 2, and so on. The CTA scheduler launches blocks on cores one by one, until none of the cores have enough resources to support an entire thread block. Four resources are checked: number of registers used per thread, shared memory used per thread, number of available thread slots and number of available thread block slots. Once each core is executing the maximum possible number of thread blocks, the CTA scheduler stalls and waits for a core to complete a CTA. It launches the next CTA in the kernel grid on any core that completes a CTA first. As the programming model does not guarantee the scheduling order of threads that belong to different blocks, blocks are assigned to cores in any order. More details on the design of the CTA scheduler can be found in [8, 32, 40, 79].

2.2.3 Warp Scheduler

Thread blocks launched on the core are further partitioned into groups, of typically 32 threads, called warps. Warps are entities used for execution by the hardware units on the core.

Fig. 2.4(b) shows a depiction of some of the units on a GPU core.

The fetch unit fetches instructions for each warp, from the instruction cache into the instruction buffer. The instruction buffer has a separate entry for each warp, allowing each warp to be at its own instruction in the kernel. Once the current instruction is decoded and its operands are collected from the register file, the warp is placed into the Warp-Queue. The warp scheduler selects a warp from this queue and dispatches it for execution on the vectorized functional units. Each thread within a warp executes the same instruction on its data, in Single Instruction Multiple Data (SIMD) fashion. Indeed, the various hardware units on the GPU core (arithmetic units, shared memory, register file) are vectorized and indexed at the warp abstraction level, because of this SIMD form of execution adopted by the GPU architecture.

Context for a warp is kept live in the register file until it completes the entire kernel. The warp scheduler keeps track of all the warps active on the core, and interleaves their execution to overlap memory latency and arithmetic computation. Initial warp schedulers used a single queue to store all the warps that are active on the core. This is depicted as the Warp-Queue in Fig. 2.4(b). We refer to such schedulers as single level schedulers in this thesis. As GPU cores became bigger and could support a higher number of warps (current architectures support up to 64), arbitrating among all the active warps every cycle became less energy efficient, and hierarchical warp scheduling policies were proposed [48, 22]. Warps are grouped into smaller subsets called fetch groups, and the warp scheduler arbitrates only among warps in a fetch group until they stall on memory access; at which point the warps are put in the larger warp queue, and warps with the next highest priority become the next fetch group. We refer to such schedulers as two level schedulers in this thesis. Warp scheduling policies have been studied extensively in the academic research community and several optimizations have been proposed. For further details on the design of warp scheduling polices, please refer to [21, 20, 48, 22, 63, 9]

In the next three chapters, we look at scheduling at each of the three levels: kernel, thread blocks, and warps, in more detail. At each level of scheduling, we describe the current state of the art, outline the issues being addressed by this research work, propose novel techniques to mitigate those issues and analyze the benefits and overheads of the proposed techniques through experiments on a cycle level GPU architecture simulator [10].

CHAPTER 3. PHASE-AWARE WARP SCHEDULER

3.1 Abstract

GPU architectures interleave execution of SIMD threads (warps) via cycle-level hardware multithreading to hide the arithmetic pipeline, and memory access latencies. The Two-Level Round Robin (TLRR) and Greedy Then Oldest (GTO) warp scheduling policies are widely accepted as state of the art due to their simplicity and applicability to a wide range of workloads. In this work, we show that the two policies do not scale with the same performance across different applications. The disparity regarding which scheduling policy works better for a given workload, depends on the characteristics of opcodes in different regions of the kernel instructions (phases). We identify phases at compile time and design a warp scheduling policy that uses information regarding them to make scheduling decisions. By mitigating the adverse effects of application phase behavior, our policy always performs closer to the better of the two existing policies for each application. We evaluate performance of the warp schedulers on 35 kernels from the Rodinia and CUDA SDK benchmark suites. For workloads that have a better performance with the GTO scheduler, our warp scheduler matches the performance of GTO, and achieves a speedup of 6.31% over TLRR. Similarly, for workloads that perform better with the TLRR scheduler, performance of the phase-aware warp scheduler matches that of TLRR and achieves a speedup of 6.65% over GTO.

3.2 Introduction

Graphics Processing Units (GPU) architectures are designed for high floating point throughput and streaming memory bandwidth. Consequently, GPU dedicate a significantly larger portion of their die area to arithmetic units compared to CPU architectures. As a trade-off, GPUs do

not include the hardware units used for latency hiding in CPUs. For example, GPUs use a non-speculative in-order pipeline, do not have branch predictors, and have much smaller caches compared to CPUs.

The mechanism used in GPUs to hide the pipeline and memory access latencies is to perform massive multithreading in hardware. Each core executes a very high number of threads in parallel, and a hardware scheduler interleaves their execution to hide latency. Specifically, groups of threads are executed as SIMD units called warps [54] (or wavefronts [47]), and the scheduler overlaps the latency of warps waiting on long latency operations with computation from other warps. The policy used by the warp scheduler is pivotal in being able to achieve a throughput which is close to peak, as it largely affects how well the latencies and computations are overlapped.

There is a large body of previous work on warp scheduling techniques. A majority of the initial works focused on mitigating warp-divergence, a problem that occurs when threads within a warp diverge in their execution flow [21, 20, 22, 48] Other works have designed techniques that focus on a subset of applications that exhibit specific characteristics. For example, authors in [63] focus on workloads that are sensitive to L1 data cache, authors in [45] focus on workloads with varying levels of memory divergence, while authors in [42] focus on applications with irregular workloads. However, the underlying policy which selects the next warp to be dispatched for execution has received rather less attention.

Two underlying policies have been widely adopted, namely round-robin (RR) and greedy then oldest (GTO). The RR policy rotates the priority of warps in round robin order after each selection. The GTO policy on the other hand, always gives a higher priority to warps that are launched earlier. Authors in [48, 22] proposed hierarchical implementations of these policies for improving increasing energy efficiency. A smaller set of warps (typically 6 to 8), from all the active warps (typically 48 to 64), referred to as a fetch group, is kept in a separate queue called the ready queue (Refer to Fig. 3.1). The scheduler only selects warps from the ready queue for execution, which makes the warp selection and scoreboarding logic simpler and more energy efficient. A warp in the ready queue is replaced only when it arrives at a long latency operation, such as a memory request. It has been shown that warps in a fetch group have

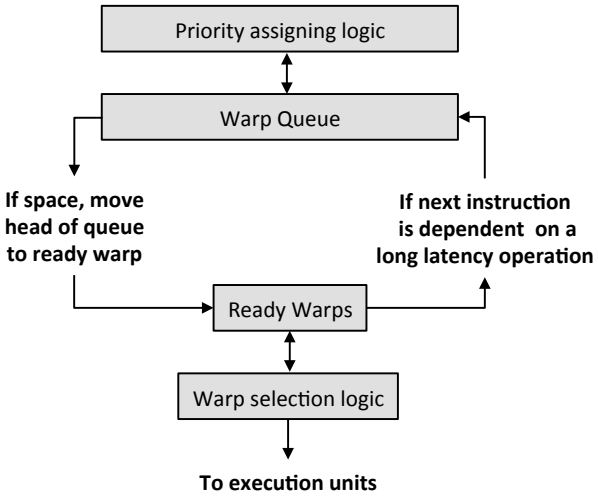


Figure 3.1: Block diagram of the two level warp scheduler. An equivalent block diagram of the single level scheduler would not have the Ready Warps queue.

enough parallelism to hide the shorter ALU latencies [48]. Moreover, prioritizing execution of subsets of warps spaces out the requests to main memory in time, which results in a better overlap of memory latency and computation [48]. Both the RR and GTO policies can be used for assigning priority to the fetch groups as well as warps within the fetch group.

In this work, we show that while the GTO policy performs better for some applications, some applications show better performance when RR is used, while others have comparable performance with either scheduler. To understand the disparity between performance of warp schedulers for different applications, we analyze how the warps progress through the programs instructions. The set of warps within a fetch group proceed together through the program instructions, at approximately the same pace, until they arrive at an instruction that depends on a long latency operation. When selected the next time, they again proceed until the next long latency operation, and so on. In this way, the program is essentially divided into regions of computations, separated by instructions that depend on long latency operations. We refer to these regions as ***phases***.

Our results clearly indicate that the length and arrangement of phases have a direct impact on the performance of warp scheduling policies. We show that the disparity regarding which policy performs better for a particular application can be explained by understanding how warps progress through the different phases of the application. With this understanding, we

then design a warp scheduling policy that uses information regarding phases that is embedded in the programs instructions by the compiler. Our policy results in a performance which is always closer to the better performing policy for the respective application. Our contributions can be summarized as follows:

1. We characterize phase behavior in GPGPU workloads, and via case studies of real-world applications show how phase characteristics impact the performance of the RR and GTO scheduling policies.

2. We describe how the phase information can be inserted at compile time and design a hardware warp scheduler that uses this information to be more robust to application phase behavior.

The remainder of this chapter is organized as follows: In the next section, we formally define program phases and provide an illustrative example. We analyze two real-world kernels in detail, and show how phase behavior affects the performance of warp schedulers for them. In Sect. 3.4, we describe our phase-aware scheduling policy, and its software and hardware implementation. In Sect. 3.5 , we provide experimental results that compare our phase-aware scheduler to the RR and GTO warp schedulers. In Sect. 3.6 we discuss related work, and in Sect. 3.7 we provide our conclusions from this work, and discuss possible future directions for compiler assisted warp scheduling techniques.

3.3 Phase Behavior in GPGPU Kernels

In this section, we formally define phases in the context of GPGPU kernels. We then illustrate the impact of phase characteristics on warp scheduling policies using examples of two CUDA kernels.

3.3.1 Definition of Kernel Phases

Fig. 3.2 shows an example of a simple CUDA kernel that adds two vectors. Three high level sections are shown in the code. First, the index for a thread is calculated using special

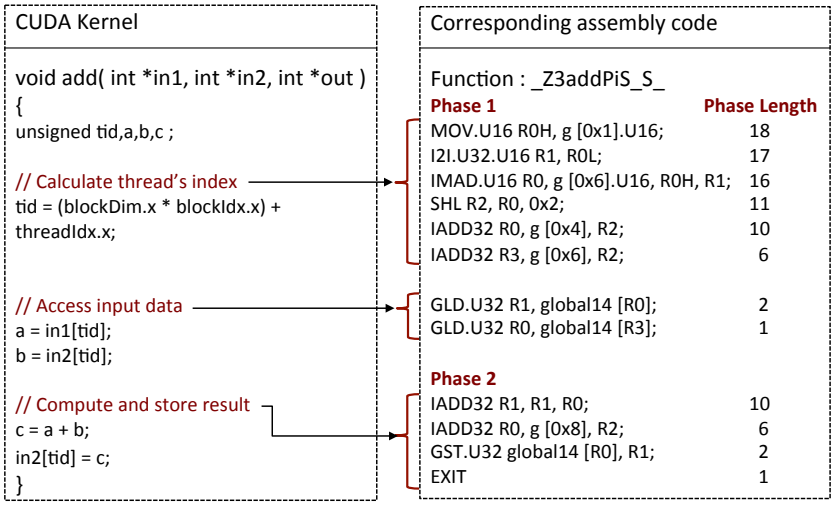


Figure 3.2: CUDA kernel for vector addition and its corresponding assembly code showing phases and phase length.

variables that store the thread and block identifiers (`threadIdx.x` and `blockIdx.x`), and the thread block dimension (`blockDim.x`). The index is then used as an offset from the base addresses (passed as arguments to kernel call) to load data for that thread. The computation is performed and result is stored back in the third section. The right box in the figure shows the corresponding assembly code.

We define a ***phase*** as: *A set of consecutive instructions such that, no instruction in the set has any input operands that are produced by a long latency instruction from any other instruction in the set.*

Observe that this kernel has two phases (refer to Fig. 3.2). Phase 2 begins at instruction: `IADD32 R1,R1,R0`. The input operands `R0` and `R1` are produced by memory load instructions (`GLD`), which belongs to the set of long latency instructions[1]. It should be noted that in real-world application kernels, contrary to the example shown, it is common to have multiple instructions in a phase that depend on long latency instructions from the previous phase. A new phase begins only if an instruction depends on a long latency instruction from the current phase. We show a simple example here for brevity. The structure of a phase is often as follows: multiple memory loads at the beginning of a phase (initial loads), followed by computations

[1]In addition to load and store, conditional and unconditional branches, and synchronization instructions are also considered as long latency instructions.

that depend on loads from the previous phase, and then an instruction that depends on one of the initial loads (this would begin a new phase). For the kernels we studied for this work, the number of phases in a kernel vary from 3 to 45.

3.3.2 Effect of Kernel Phases on Warp Schedulers

In the two level warp scheduler, a warp is moved from the Ready Warps queue to the larger pool of all warps, when it arrives at an instruction that depends on a long latency operation. As such instructions lie at phase boundaries, warps proceed through the kernel instructions one phase at a time. When a warp reaches the end of a phase, it is put back in the larger warp pool and a different fetch group gets a chance to execute. As the fetch group maintains its priority until it reaches the end of a phase, one of the main factors that affects the performance of the warp schedulers is phase length (refer to Fig. 3.2). Phase length is computed by summing the instruction latencies, from the last to the first instruction of a phase. It is an approximation of the minimum number of cycles that a warp would take to reach the end of a phase. The lengths of phases 1 and 2 in Fig. 3.2 are 18 and 10 respectively.

Fig. 3.3 is a depiction of the effect of phase length on performance of the warp schedulers. For simplicity, the illustration assumes that all warps in a fetch group (FG) arrive at the end of a phase simultaneously. Fig. 3.3(a) is an example of an application where the GTO scheduler performs better than RR. It has a medium length phase, followed by a short phase and then phase of long length. Notice that the computation from medium length phase of FGs 2 and 3 are enough to hide the memory latency of FG 1. At this time, the RR scheduler selects FG 4 ①, while the GTO scheduler selects FG 1 ②. Observe that, with RR scheduling all the FGs arrive at the short length phase at the same time. As computation from three short length phases is not enough to hide the memory latency, some of the latency is exposed ③. In contrast, as the GTO scheduler selects FG 1 at ②, FG 1 arrives at the long length phase earlier ④. This long phase is then used to hide the latency which was exposed in the case of RR scheduling. In general, for kernels that have shorter length phases in the middle of the kernel, the GTO scheduler performs better than RR.

Fig. 3.3(b) is an example of an application where the RR scheduler has a better performance

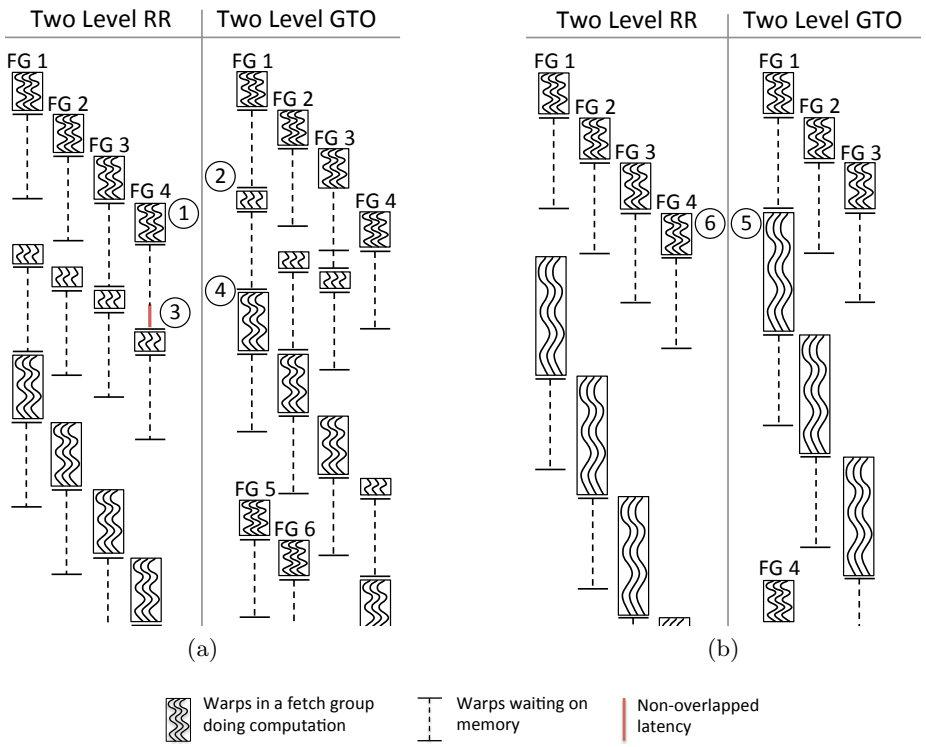


Figure 3.3: Impact of phase length on warp scheduling.

compared to GTO. Notice that the application has a medium phase, followed by a phase of long length. Similar to the example of Fig. 3.3(a), computation from three FGs is enough to hide the memory latency of FG 1, and the GTO scheduler switches back to FG 1 ⑤. As this phase is long enough to overlap the latency of outstanding memory requests, the load on the memory subsystem is reduced. On the contrary, the RR scheduler selects FG 4 ⑥. This results in sending out more requests to memory, and thus better utilization of the memory bandwidth while executing the long length phase. We can observe that around the time when the GTO scheduler begins fetch group 4, the RR scheduler has already executed a significant portion of it. In general, for kernels that have extremely long phases in the middle of the kernel, the RR scheduler performs better than GTO.

3.3.3 Illustrative Applications

The effects of phase length on warp scheduling policies can be summarized as follows:

1. Performance of the RR policy is adversely affected in kernels that have shorter length phases,

due to all the warps arriving at these phases at the same time.

2. Performance of the GTO policy is adversely affected in kernels that have long length phases, as the scheduler keeps choosing warps from these phases, thereby under-utilizing the memory bandwidth.

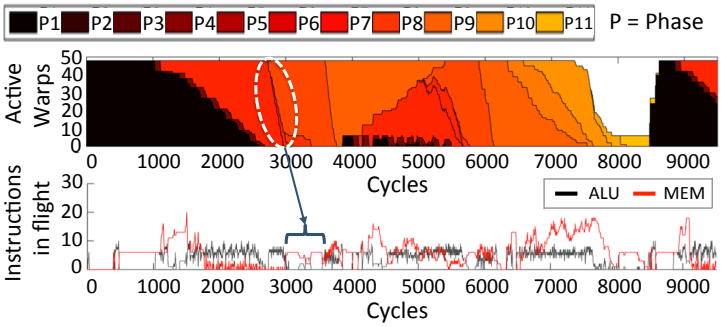
In this section, we explore two real-world application kernels that clearly demonstrate these effects. The top figures in each of the subplots of Fig. 3.4 and Fig. 3.5 plot the total number of warps in each phase at a given cycle. The bottom figure plots the total the number of ALU instructions and memory requests that are in flight.

3.3.3.1 B+Tree

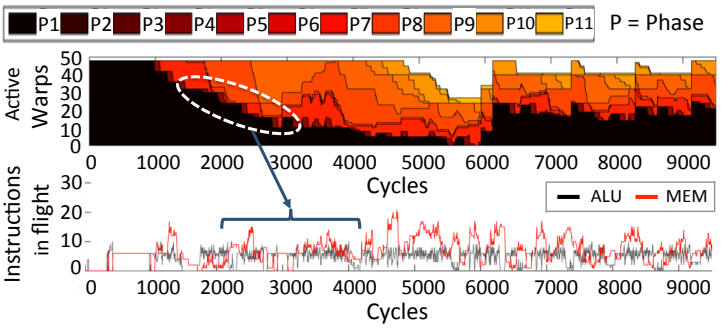
B+Tree is an example of a kernel for which the GTO scheduler achieves a better performance compared to RR. It launches 48 warps on each core, as six thread blocks of eight warps each. The fetch group size is six.

Observe in Fig. 3.4(a) that, when warps from the first fetch group (FG) complete the first phase, warps in the next FG execute. Warps completing a phase can be seen in the plot when the number of warps in a particular phase decreases. This is followed by the third FG, and so on. Thus, due to RR scheduling warps proceed through the kernel instructions one phase at a time. As mentioned in Sect. 3.3.2, this trait of the RR scheduler becomes an issue in kernels which have short length phases. Observe that phase 4 is extremely short (marked oval). All the warps finish this phase and arrive at the start of phase 5 at around the same time. Observe that during this time, the memory load is high and ALU load becomes almost zero. This is because all warps are waiting for the memory requests issued in phase 4. The phase problem alleviates a little after cycle 4000 due to some warps branching back to phases 1 and 3. However, only a small portion of the runtime is shown here. The application grid is of 65535 blocks and this pattern keeps repeating throughout the kernel execution after every 6^{th} block.

In contrast, observe in Fig. 3.4(b), that after three fetch groups complete, the GTO scheduler selects FG 1. This is observed in the figure when the number of warps in phase 1 stop decreasing and the number of warps in phase 2 start to decrease. Due to this decision, only a small set of warps arrive at the short length phase 4, at the same time. As the other warps are in the



(a) Two Level Round Robin



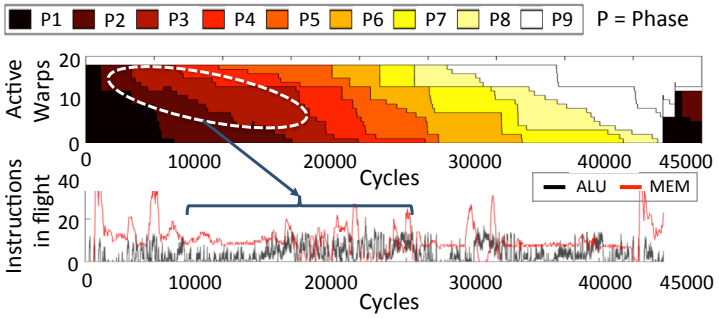
(b) Two Level GTO

Figure 3.4: B+Tree Phase Graphs

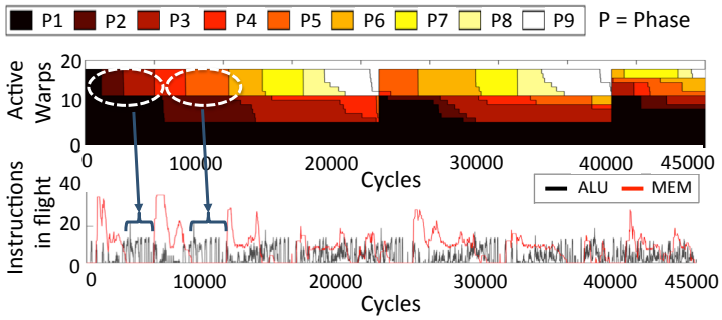
longer length phases, the scheduler can switch to them to hide the memory latency. Observe in the lower plot of Fig. 3.4(b), the amount of time when the ALU and memory load becomes zero has reduced as compared to the RR scheduler. Moreover, observe in the area plot that the distribution of warps across different phases (marked oval) is much better in GTO as compared to RR.

3.3.3.2 Computational Fluid Dynamics (CFD)

CFD is an example of a kernel for which RR scheduling achieves better performance as compared to GTO. It launches 18 warps on each core, as three thread blocks of 6 warps each. Observe in Fig. 3.5(b) that the first fetch group (FG) of 6 warps starts executing ahead and reaches phase 2. Notice that the GTO scheduler selects FG 1 again (first marked oval). This is because, as phase 1 is a long length phase, the memory requests sent near its beginning are completed by the time warps reach phase 2. As the next two phases are long as well, the warps in FG 1 become pending only when they reach phase 4. At this time fetch group 2 gets to



(a) Two Level Round Robin



(b) Two Level GTO

Figure 3.5: CFD Phase Graphs

execute. Observe that when warps from FG 2 reach phase 2 and become pending, FG 1 gets selected again and executes until it reaches phase 6 (second marked oval).

In this manner, due to phases of long length, a group of warps keeps getting the priority. Consequently, other warps do not get a chance to execute and issue their memory requests. Observe in the lower plot of Fig. 3.5(b) that the memory load becomes zero in the duration when warps from FG 1 are in the long compute phases. In comparison, the RR scheduler switches priority to the next fetch group at phase boundaries. Due to this, warps that were in the shorter length phases are able to execute and send the memory requests. We can observe in Fig. 3.5(a) that the warp distribution across different phases is much better. Also, notice that the memory load is more regular and has lesser fluctuations as compared to GTO. The average memory load for this kernel with the two level RR scheduler was 4% higher as compared to the two level GTO.

3.4 Phase Aware Warp Scheduling

In this section, we first propose a dynamic scheduling policy based on phase length that can mitigate the negative effects of phases outlined in the previous section. We then provide details of the compiler frontend that adds the required phase information in program instructions and hardware implementation of the warp scheduler that uses this information at runtime.

3.4.1 Scheduling Policy

In the previous section, we showed that performance of the RR scheduler is affected if the kernel has phases of short length and all the warps arrive at such phases simultaneously. On the other hand, performance of the GTO scheduler is adversely affected when the kernel has long phases and the scheduler keeps selecting the set of warps that are in the long phase.

Our policy is based the following simple observation: *Adverse effects of the RR and GTO scheduling policies can be mitigated by always choosing the warp that is at the shortest next phase.*

Consider a kernel with two phases P_i and P_j, such that P_j is shorter than P_i.

Case (1): The RR policy chooses a warp in phase P_i. This implies that warps in P_j were selected before this.

(a): P_i is before P_j in program order. This is the more common case given that warps in P_j were selected earlier. Selecting warps in phase P_i would get all warps to the shorter phase P_j, leading to a possibility of non-overlapped memory latency. Hence, it would be ideal to first select warps in P_j.

(b): P_i is after P_j in program order. This would happen if warps (currently in P_j) executed before this and branched back to an earlier phase. Selecting warps in phase P_j would get all warps to the longer phase P_i. Note that in this case, selecting warps from P_i would harm performance only if they branch back to phase P_j. Nonetheless, selecting warps in P_j would not negatively impact performance.

Case (2): The GTO policy chooses a warp in phase P_i. This implies that warps in P_i were launched before warps in P_j.

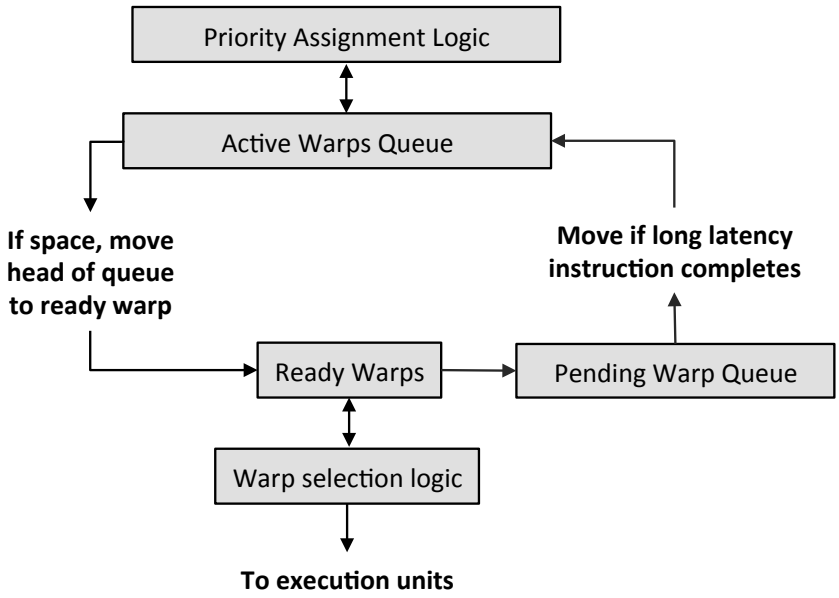


Figure 3.6: Block diagram of our two level warp scheduler.

(a): P_i is after P_j in program order. This is the common case as GTO gives priority to older warps, causing them to be ahead in the program. If phase P_i is extremely long, choosing a warp from phase P_i might under-utilize the memory bandwidth. Hence, it would be better to first execute warps in P_j and then overlap the memory latency using warps in phase P_i.
(b): P_i is before P_j in program order. This would happen if warps with the lower index have branched back to P_i. Again, selecting warps in phase P_j would issue memory requests which can then be overlapped by warps from P_i. Note that in this case, selecting warps from P_i would have the same effect, if warps in P_j also branch back to P_i. Nonetheless, selecting warps which are in phase P_j would not negatively impact performance.

3.4.2 Implementation

3.4.2.1 Front-end

The phase length information is computed at compile time and included in the kernel instructions. As the CUDA ISA is not open-source, we use PTX-Plus [10]. PTX is an intermediate assembly language generated during the compilation of CUDA kernels. PTX-Plus is a modified version of PTX which closely matches the ISA that executes on the hardware. We

chose to work with PTX-Plus because we observed that PTX is generated before the compiler has performed instruction scheduling. Due to this, the memory load and store instructions are scheduled very close to their dependent instructions, resulting in the instruction sequence being fragmented into several phases.

A long-op register is defined as a register that is a destination operand of a long latency instruction. To create phases, a set of long-op registers (empty at initialization) is maintained. The PTX-Plus assembly code is parsed at compile time from top to bottom and long-op registers are added to the set one instruction at a time. The first instruction that consumes any register currently in the set marks the start of a new phase. At the start of each phase, the set is cleared. This assumes that long latency instructions issued close to each other would complete around the same time, and avoids creating several short phases. In addition, each basic block starts a new phase.

Once the phases are created, the code is traversed backwards to calculate phase lengths. Within each phase, instruction latencies are accumulated from the last to the first instruction as mentioned in Sect. 3.3.1. This simultaneously calculates two pieces of information. Each instruction is assigned a phase distance, which approximates the number of cycles a warp executing this instruction would take to reach the end of the phase. Additionally, each phase is assigned a phase length, which approximates the number of cycles a warp takes to execute the phase.

3.4.2.2 Hardware Implementation

The schedulers were implemented in GPGPU-Sim v3.2 [10], a cycle-accurate GPU architecture simulator. The phase distance mentioned in the previous subsection is used by the phase-aware single level scheduler, while the phase length is used by the phase-aware two level scheduler.

Single Level Schedulers: As mentioned in Sect. 3.2, single level schedulers maintain a queue of all warps that are active on a core. All the warps in the active queue are checked every cycle and the one with the highest priority is chosen. The priority assigning policy is the key difference between the different schedulers. The RR scheduler rotates the priority after each selection, while the GTO scheduler always assigns the highest priority to the oldest warp. The

phase aware scheduler compares the phase distance of the current instruction for each warp. Warp at an instruction with the lowest distance from the next phase is chosen. For warps that are at the same distance, the oldest warp is chosen first. This requires the distance information to be added for each instruction. In real GPU implementations, this can be achieved via opcode extensions. Our experiments showed that a phase distance of 512 covers more than 98% of all kernels[2]. Considering instruction size of 8 bytes and L1 cache line size of 128 bytes, adding phase distance would increase the static instruction size by 12.5%.

Two Level Schedulers: In the baseline two level scheduler (refer to Fig. 3.1), when a warp in the ready queue arrives at a long latency operation, it is put in the active queue and replaced by the warp from the head of the queue. This implementation works well if priority of the warps in the active queue is rotated in a round robin order. However, notice that this is an issue for any scheme that requires warps to be prioritized. If a single queue is used to store the active warps, as well as the warps waiting on long latency operations, all the warps would need to be checked when replacing a warp from the ready queue. To solve this, we use an additional pending queue to store warps that are waiting on long latency instructions.

Fig. 3.6 shows a block diagram of our implementation of the two level scheduler. A warp waiting on a long latency instruction is first moved to the pending queue. When all the long latency instructions complete, it is moved to the tail of the active queue. The priority assignment logic is triggered at this point and warps in the active queue are sorted. This design effectively allows implementation of different scheduling policies by modifying the policy of the priority assignment logic. The RR scheduler does not sort the warps, while the GTO scheduler sorts the warps in launch order. The phase-aware scheduler uses the phase distance of the next phase to sort the warps. Consequently, contrary to the single level scheduler, length information is required only once for an entire phase. Our experiments show that adding phase length increases the static instruction size by less than 1%. After sorting, warp at the head of the active queue is moved to the ready queue.

[2]For longer phases, multiple opcode extensions can be used.

Table 3.1: GPGPU-Sim configuration used for evaluating phase aware warp scheduler

Chip configuration	
Number of cores	16
Core frequency	1300 MHz
DRAM clock frequency	1850 MHz
Peak SP / DP floating point throughput	1330 / 650 GFLOPs
Peak DRAM bandwidth	177 GB/sec
Core configuration	
Maximum thread blocks per core	8
Maximum warps supported per core	48
Execution units per core	32 ALUs, 4 SFUs 16 LD/ST units
Scheduler configuration	
Warp schedulers per core	2
Instruction dispatch throughput per scheduler	1 instruction every 2 cycles
Ready warps queue size	6

3.5 Experimental Results

3.5.1 Methodology

We configured the simulator [10] to match the architecture of NVIDIA Tesla M2090 GPU [53] (refer to Table 3.1). To perform our evaluations we chose kernels from the CUDA SDK [51] and the Rodinia benchmark suites [14]. The SDK has 49 applications, while Rodinia has 19; with each application having multiple kernels. We pruned our workload list by omitting the applications provided in the SDK for hardware profiling and demonstrating interoperability with graphics APIs. We also omitted kernels that did not have a grid size large enough to fill all the cores, and the size could not be increased without significantly changing the application code. Our final workload list had 13 kernels from 11 applications of the SDK and 17 kernels from 12 applications of Rodinia (refer to Table 3.2 and Table 3.3). For brevity, we discuss results of kernels for which either the RR or the GTO scheduling policy showed a better performance. For the remaining kernels, the GTO, RR and phase-aware scheduling policies had comparable performance (within 1% of each other).

Table 3.2: Workloads from the Rodinia benchmark suite [14] used for evaluating phase aware warp scheduler

Name	Back Propagation		B+Tree Search		Heart Wall	K-means Clustering	LU Decom-position	Speckle Reducing Anisotropic Diff.	Comp.Fluid Dynamics
Abbreviation	BP - K1	BP - K2	B+T - K1	B+T - K2	Heart	KM	LUD	SRAD	CFD
Total Thread Blocks	65535	65535	65535	65355	56	841	16129	16384	1817
Threads / Block	256	256	256	256	256	256	256	256	192
Thread Blocks / Core	6	5	5	6	4	6	6	6	3
Total Phases	5	5	16	11	45	2	3	4	9
Longest Phase	206	99	40	32	212	48	150	218	985
Shortest Phase	26	14	5	3	41	14	7	31	86

Table 3.3: Workloads from the CUDA SDK [51]

Name	Discrete Wavelet Transform	DXT Compression	Fast Walsh Transform	Histogram	Monte Carlo Option Pricing
Abbreviation	DWT	DXTC	FWT	HIST	MCO
Total Thread Blocks	4096	4096	4096	256	2048
Threads / Block	512	256	512	192	256
Thread Blocks / Core	3	4	3	6	5
Total Phases	2	3	2	2	2
Longest Phase	144	1159	174	44	284
Shortest Phase	44	24	45	24	38

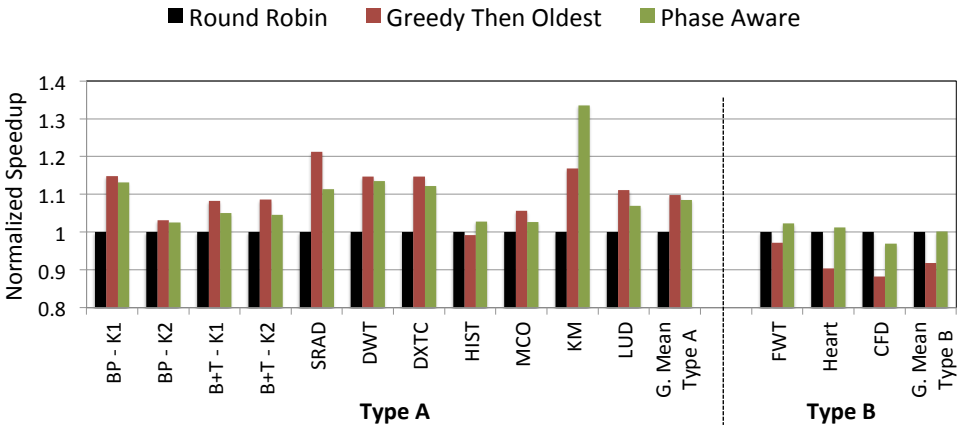
3.5.2 Impact on Performance

Fig. 3.7 plots the speedup of the Greedy Then Oldest (GTO) and phase-aware schedulers normalized to the Round Robin (RR) scheduler. The single level schedulers have an advantage of selecting from all the warps on the core. Hence, we compare the performance of the single level and two level schedulers separately. We have grouped the kernels into two types. For kernels grouped under **Type A**, the GTO scheduler achieves a better performance compared to RR, while for kernels grouped under **Type B**, the RR scheduler performs better than GTO.

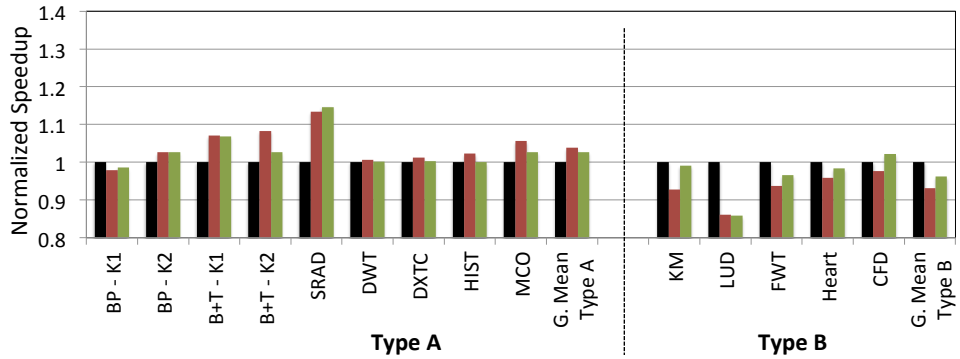
3.5.2.1 Type A Kernels

Kernels for which the GTO policy performs better (grouped under type A), have phases of extremely short length. As mentioned in subsection 3.3.3.1, our simulations showed that when the RR policy is used, warps get accumulated in these short phases at around the same time, causing the core to become idle.

Eight kernels always perform better with the GTO policy (refer to Fig. 3.7(a)). The BP-K2, DWT, HIST and MCO kernels have their shortest phase at the beginning of the kernel. The SRAD kernel has two short phases in the middle of the kernel code, while the B+Tree kernels have phases of length shorter than 20 cycles intermixed throughout the code. Notice



(a) Single Level Schedulers



(b) Two Level Schedulers

Figure 3.7: Performance comparison. **Type A** - GTO performs better than RR. **Type B** - RR performs better than GTO.

in Fig. 3.7 that the BP-K1, LUD and KM kernels have a better performance with the GTO policy for single level implementation and with the RR policy for two level implementation. This happens due to a phenomenon we refer to as intra-block tail effect, which we will explain in detail in subsection 3.5.5.

For the type A kernels, the single level GTO has a speedup of 10% over single level RR on average. The performance of the phase-aware scheduler is close to that of GTO for all these kernels and it achieves a speedup of 9% on average over RR. A similar performance trend was observed for the two level schedulers. However, as warps progress through the kernel in fetch groups, all the warps do not arrive at the short phases simultaneously. This reduces the negative effect on performance of the RR scheduler. Notice in Fig. 3.7(b) that for the DWT, DXTC and HIST kernels RR scheduling now performs comparable to GTO and phase-

aware. Consequently, the speedup of GTO over RR is lesser with an average of 4%. Again, the performance of the phase-aware scheduler is close to that of GTO and it achieves a speedup of 3% on average over RR.

3.5.2.2 Type B Kernels

Kernels for which the RR policy performs better (grouped under type B) typically have extremely long length phases. Heartwall is a very long kernel with 1523 instructions and 45 phases, with several long length phases of over 100 cycles. CFD has pairs of medium and long length phases, with an average phase length of 290 cycles, and FWT has just two phases of length 45 and 174. As discussed in subsection 3.3.3.2, memory operations issued at the beginning of these long phases complete before the phase ends. Consequently, the GTO policy keeps prioritizing a small set of warps causing it to under-utilize the memory bandwidth.

For the kernels in our benchmark suite, RR scheduling performs better than GTO only for three kernels for the single level implementation and has an average speedup of 9.2% over GTO. Notice in Fig. 3.7(a) that the phase-aware scheduler now performs closer to the RR policy. Performance of the phase-aware scheduler is comparable to RR and it achieves a speedup of 9% over GTO. For the two level implementation, similar to the type A kernels, the impact of phase behavior on performance (now on the GTO policy) reduces. The RR policy achieves a better performance for six kernels, with an average speedup of 7% over GTO. The phase-aware scheduler matches the performance of RR for all kernels except LUD and achieves a speedup of 4% over GTO. The reason for the negative impact on performance of the LUD kernel is explained in subsection 3.5.5.

3.5.2.3 Summary of Performance Impact

Observe in Fig. 3.7(a) and Fig. 3.7(b), that for kernels grouped under type A, performance of the phase-aware scheduler is always closer to that of the GTO scheduler. On average, its performance is within 99% of GTO for both the single level and two level implementations, and it achieves a speedup of 9% and 3% respectively over RR. For the kernels grouped under type B, the phase-aware scheduler now performs closer to RR. It matches the performance of RR

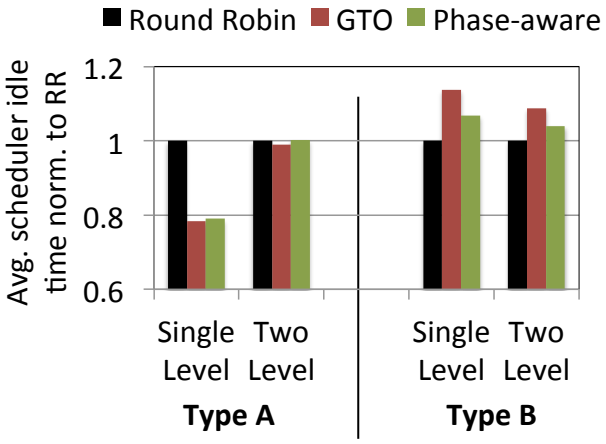


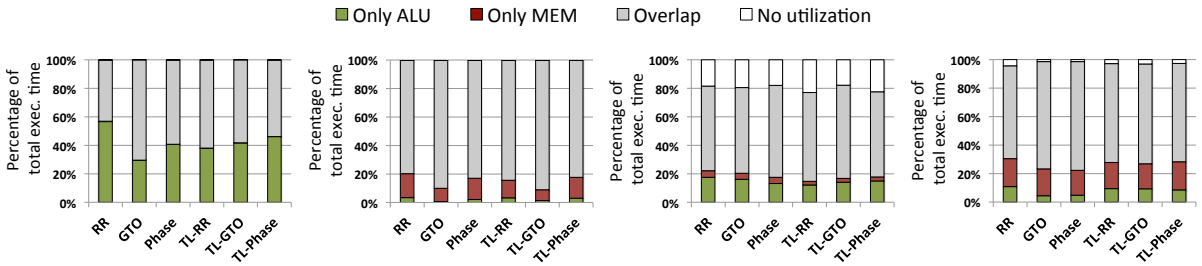
Figure 3.8: Comparison of average scheduler idle time.

for the single level implementation and performs within 96% for the two level implementation. Consequently, it achieves a speedup of 8.9% and 4% over the GTO policy. These results indicate that application phase behavior has an impact on performance of the GTO and RR scheduling policies. As application type is not known a priory, performance can be negatively impacted if static policies are used. For example, if the RR policy is used, we would incur performance loss for type-A kernels, and similarly for type-B if the policy was GTO. On the contrary, the phase-aware policy always performs closer to the better performing policy for any kernel. Hence, by utilizing information regarding program phases, our scheduling policy becomes applicable to a wider range of applications.

3.5.3 Impact on Scheduler Idle Time

At any given cycle during kernel execution, the scheduler can be in either one of the four states: 1 - no warps have a new instruction, 2 - all warps that have an instruction are not ready (waiting for an operand from a previous instruction), 3 - all warps that have a ready instruction cannot issue due to a pipeline stall, or 4 - at least one warp can issue an instruction. We refer to the sum of the cycles spent in the first three states as scheduler idle time. As scheduler idle time is the cycles when no new instructions are issued, it is a good indicator of the performance trend.

Fig. 3.8 plots the scheduler idle time as a percentage of the total execution time. For brevity,



(a) BP - K1 (Type A - Comp. intensive)

(b) B+T - K2 (Type A - Mem. intensive)

(c) Heartwall (Type B - Comp. intensive)

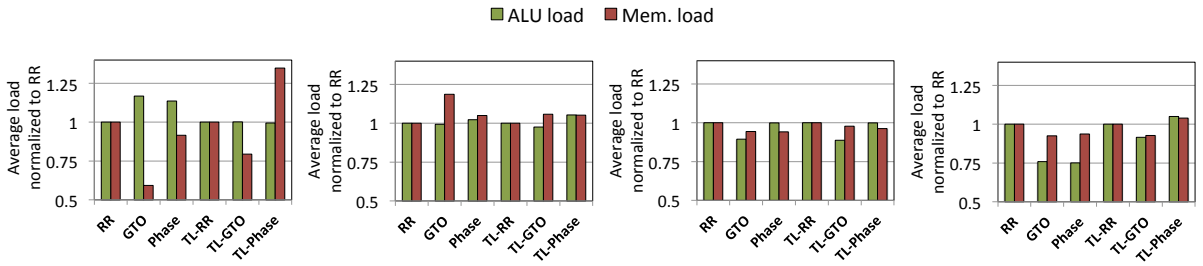
(d) CFD (Type B - Mem. intensive)

Figure 3.9: Breakdown of execution time for two kernels of type A and type B.

the data is averaged across kernels of the same type and normalized to RR. For type A kernels, the GTO and phase-aware policies had a 21.56% and 21.03% lower idle time compared to RR for the single level scheduler, while the average idle time was comparable to RR for the two level scheduler. As expected, for the kernels grouped under type B, the average scheduler idle time for GTO was higher than the RR policy. It was higher by 13.66% for the single level and by 8.73% for the two level scheduler. As the phase-aware scheduler performs closer to RR, its idle time was lower than GTO, but higher than RR by 6.8% for the single level and 3.9% for the two level. Notice that similar to the performance trend, the difference in the idle time between schedulers for a given type of kernels is lower in the two level implementations. This again shows that the impact of phase behavior is reduced with two level scheduling. Also, the idle time for the phase-aware warp scheduler is always closer to the better performing policy for each kernel type.

3.5.4 Impact on Functional Unit Load

The total runtime of a kernel can be broken down as 'Idle cycles' + 'Total Computation Cycles' + 'Total Memory Access Cycles' − 'Cycles of Overlap of Computation and Memory Access'. The amount of time spent by a kernel doing only arithmetic or only memory operations indicates whether the kernel is compute or memory intensive. To measure this, we instrumented the simulator and monitored the ALU pipeline and the memory system. The kernel is considered as performing computation (or memory access), if there is at least one ALU (or memory)



(a) BP - K1 (Type A - Comp. intensive)

(b) B+T - K2 (Type A - Mem. intensive)

(c) Heartwall (Type B - Comp. intensive)

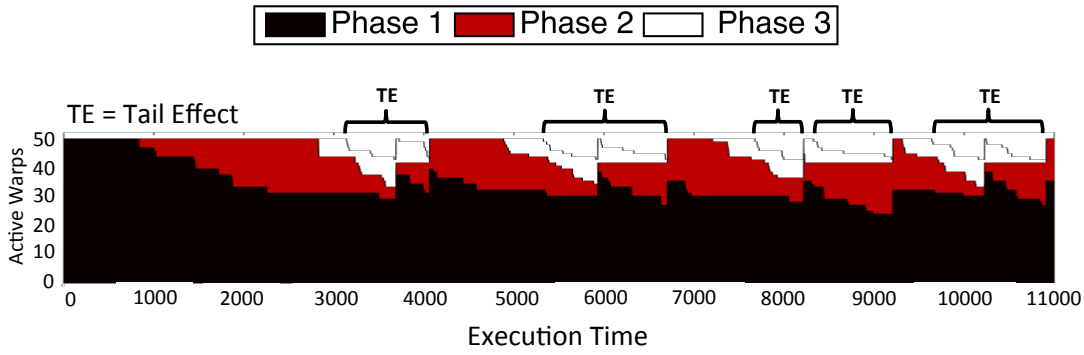
(d) CFD (Type B - Mem. intensive)

Figure 3.10: Average ALU and memory load of two kernels of type A and type B.

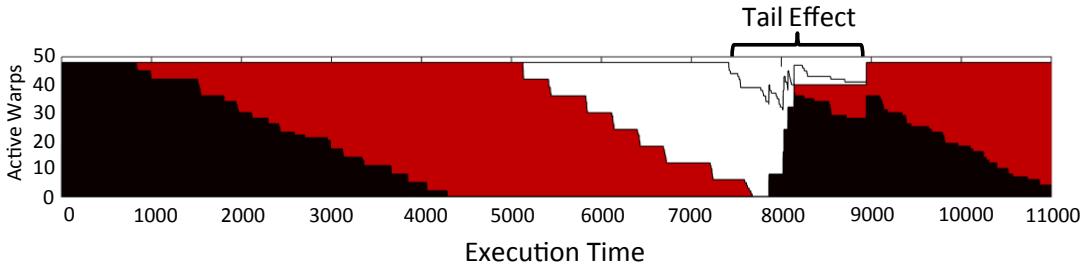
instruction in flight at that cycle. Load on the ALU and memory pipelines is defined as the total number of instructions that are in flight during that cycle. Fig. 3.9 shows the breakdown of execution time of two kernels each, of type A and type B, and Fig. 3.10 plots their respective ALU and memory loads.

BP-K1 is an example of a type A kernel that is compute intensive. A significant portion of the execution time is spent performing only ALU operations (refer to Fig. 3.9(a)). Notice in Fig. 3.10(a) that the single level GTO and phase-aware schedulers maintain a lower load on the memory subsystem. However, as the average ALU load achieved is higher, they perform better than RR. Also, notice that for the two level implementation, the RR scheduler maintains a ALU load that is similar to the GTO and phase-aware schedulers. Consequently, the GTO and phase-aware scheduler do not achieve a speedup over RR for the two level implementation (refer to Fig. 3.7(b)). B+Tree-K2 is a type A kernel that is memory intensive (refer to Fig. 3.9(b). Again, observe in Fig. 3.10(b) that although the ALU load is comparable for all the schedulers, the GTO and phase-aware schedulers maintain a higher memory load, and consequently achieve a better performance compared to RR.

Similarly, Figs. 3.9(c) and 3.9(d) are examples of type B kernels (RR performs better than GTO), which are compute and memory intensive respectively. Around 20% of the execution time of Heartwall is spend performing ALU operations by all schedulers (refer to Fig. 3.9(c)). Notice in Fig. 3.10(c) that the average ALU load achieved by the GTO policy is lower than that of RR and phase-aware schedulers, thereby achieving a lower performance. CFD is a type



(a) Two-level GTO



(b) Two-level Round-robin

Figure 3.11: Intra-block tail effect of the LUD kernel.

B kernel that is memory intensive. The GTO scheduler achieves a lower average memory load for both the single and two level implementations, and hence achieves a lower performance as compared to RR. Notice that the single level phase-aware scheduler maintains a lower memory load compared to RR, while the two level implementation achieves a slightly higher load. This is reflected in the performance results (refer to Fig. 3.7), as the single level phase-aware scheduler is 3% slower than RR, while the two level achieves a speedup.

3.5.5 Impact of Intra-Block Tail Effect on Performance

As mentioned in Sect. 3.5.2, performance of the two level GTO and phase-aware policies, for the LUD, KM and BP-K1 kernels, is adversely affected due to intra-block tail effect. Intra-block tail effect is a phenomenon where some warps of a thread block complete the kernel before others. This happens when some warps from a thread block complete their long latency instructions before others and are selected in an earlier fetch group. The LUD, KM and BP-K1 kernels have a short phase of length 7, 14 and 16 respectively at the end. The warps selected in the earlier fetch groups run ahead and complete the last phase. However, a new thread block

is not launched until all warps of a block complete, thus reducing the number of warps on the core. The effect is reduced in the single level schedulers as warps are not scheduled in fetch groups. Moreover, with single level RR, all warps arrive at the last short phase at the same time leading to the GTO and phase-aware policies performing better (refer to Fig.3.7(a)).

Fig. 3.11 depicts the intra-block tail effect for the LUD kernel by plotting the total number of active warps on a core. Notice in Fig. 3.11(a) that 6 warps (size of the fetch group) finish all the three phases and the number of warps reduce to 42 (the first marked TE). However, new warps are not launched until more warps finish phase 3. Notice that the tail effect repeats each time warps in phase 3 complete. On the contrary, as the RR scheduler switches to a different fetch group after each phase (refer to Fig. 3.11(b)), all warps reach phase 3 at around the same time. Notice that the length of the tail for RR is around 2000 cycles for all the 6 blocks, as compared to 5000 cycles for GTO (sum of all TEs in Fig. 3.11(a)). Performance of the phase-aware scheduler is also affected as it chooses the shortest length phase which is the last phase for these kernels. To mitigate this problem, we are currently evaluating implementations for adding thread-block affinity to our two level scheduler. Note that this would increase the complexity of the scheduler considerably and its overhead for the hardware implementation should be carefully considered.

3.6 Related Work

3.6.1 General Warp Scheduling Techniques

Lakshminarayana and Kim [37] analyzed the performance of GPU kernels under various instruction fetch and memory scheduling policies. They showed that applications in which warps have a uniform execution latency, a fairness based warp and DRAM access scheduling approach results in the best performance. The goal of their work was to explore the effects of the scheduling policies on the performance of various applications. In contrast, we try to use information about the application to make scheduling decisions at runtime and perform closer to the better performing policy for each application.

Gebhart et al. [22] and Narasiman et al. [48] proposed the hierarchical warp scheduler that

we used as a baseline for our work. The focus of authors in [22] was making the warp scheduling logic simpler and more energy efficient. Choosing from a smaller set of warps every cycle saves energy spent on scheduling. Their results show that using an active queue size of 6 results in less than 1% performance loss compared to the single level scheduler. The focus of authors in [48] was to use the two level policy to improve latency hiding. As the two level policy makes the warps progress through the kernel instructions at different speeds, it results in a better overlap of memory latency and computation. We showed that although the two level schedulers perform well, the policy to select the fetch group has an effect on their performance. We build upon the schedulers proposed in their works, and make them more robust to application phase behavior.

Our work is closest to the work published by Chen et al. in [15]. They identify the adverse effects of short phases on the two level RR scheduler and propose a scheduler that shifts to the next fetch group only at priority shift instructions. Short phases are identified by the compiler and merged with the previous phase by adding a priority shift instruction after the short phases. This effectively makes the priority selection policy as greedy until all the short phases are completed by a fetch group. Our policy to select the phase with the shortest length has the same result. The problem they identify with the RR scheduler can be mitigated if the GTO policy is always used. Instead, our work identifies the scenarios when the GTO policy has a lower performance as well, and proposes a more robust scheduling policy that would be applicable to a wider range of application types.

3.6.2 Scheduling Techniques to Mitigate Warp Divergence

In addition to the generic techniques mentioned in the previous subsection, there is a significant amount of published work that has focused on techniques to mitigate warp divergence. Warp divergence occurs when threads within a warp execute different program paths due to branches. Traditional GPUs execute each path sequentially with lesser number of threads, thereby under-utilizing the GPU core's computational throughput. Fung et al. [21] proposed Dynamic Warp Formation (DWF) which creates new warps from threads that fall on the same program path after the divergence point. Their technique suspends a warp when it reaches an instruction that causes threads in a warp to diverge. When other warps arrive at the same

instruction, new warps are created and all warps proceed from that point. To reduce the synchronization overhead due to warps from different thread blocks forming a warp, Fung et al. [20] later proposed Thread Block Compaction (TBC), which adds thread block affinity to DWF. TBC creates new warps from diverged warps of the same thread block, similar to the Large Warps technique proposed in [48].

Meng et al. [46] proposed Dynamic Warp Subdivision (DWS) which divides a warp into warp splits on branch divergence. The focus of DWS is to improve latency hiding in scenarios when one warp split encounters a cache miss. The latency of memory access can be overlapped by executing the other warp split. Their technique creates warp splits to improve latency hiding upon memory divergence as well. Memory divergence occurs when some threads of a warp hit in the cache and others miss. The DWS technique creates a warp split with threads that hit and lets them continue. This allows those threads to reach the next memory request and possibly prefetch for the warp split that missed the first memory request. A similar technique of dividing warps into separate scheduling entities was proposed by Steffen and Zambreno [68]. They divide the kernel instructions, which are potential program paths after divergence, into smaller instruction blocks called μ-kernels. On warp divergence, threads wait in a partial warp pool. When there are enough threads to make a complete warp, the new warp is allowed to execute the μ-kernel.

Our phase scheduler can be used along with the warp divergence techniques mentioned above. Our technique marks each basic block as a new phase. Thus on warp divergence warps would be first put into the pending pool. Each of the techniques mentioned above creates new scheduling entities at this point. The new entities can then be scheduled by our phase scheduler using the phase length information.

3.6.3 Thread Throttling

There is a body of work that focuses on throttling the amount of thread level parallelism available on the core. Although all of these are not at the warp level, changing the number of active warps on a core has an effect on warp scheduler performance. Guz et al. [25] designed an analytical model to study the impact of amount of TLP on performance of highly multi-

threaded architectures. They showed that performance increases initially due to increase in TLP, and then starts to reduce once the total working set of the threads does not fit in the cache. Performance continues to decrease if more threads are launched until the TLP is high enough to hide the increased memory latency. The region of performance dip is referred to as the "performance valley". The authors in [63, 40, 32, 8] effectively detect at runtime when the number of active threads causes the GPU to get into the performance valley.

Rogers et al. [63] detect scenarios when the L1 data cache is trashed. They monitor the cache lines to detect warps that have lost intra-warp locality because of other warps evicting data that would have been used by them. A scoring system increases the score of such warps. Warps below a certain score are not selected by the scheduler, thereby reducing the number of active warps. Kayiran et al. [32] monitor the amount of time spent waiting for memory. The number of threads is reduced if the time is more than an empirically found threshold. Lee et al. [40] find the *optimal* amount of TLP by launching the maximum number of warps initially and then using a greedy scheduling policy. After the first thread block completes, the optimal number of blocks is estimated using the number of instructions that have been completed until then. Awatramani et al. [8] detect the optimal thread block count by comparing the pipeline stalls at different block counts. They launch half the maximum number of warps and then use history information of the previous block counts to guide the scheduler. Lee et al. [41] identify a problem similar to the tail effect mentioned in Sect. 3.5.5 for workloads that have varying warps execution times. They throttle warp execution by assigning a time slice to each warp, proportional to its the execution time with the RR scheduler. The tail effect is reduced by giving a larger time slice to longer running warps.

Each of the above techniques reduces the number of warps on a core to a smaller set. However, the underlying policy to select warps from this reduced set to dispatch to the execution units is either RR or GTO. Hence, our phase-aware warp scheduler can be easily used in conjunction with the above-mentioned techniques.

3.7 Conclusion

In this work we analyze phases in GPGPU kernels and show that the performance of warp schedulers depends on characteristics of these phases. Using real-world application examples, we show that an efficient warp scheduling policy can be designed by understanding how warps progress through these kernel phases. Based on these observations, we propose a novel phase-aware warp scheduling policy that uses information provided by the compiler to make scheduling decisions. We implement this scheduler in a GPU simulator and demonstrate that it is more robust to phase behavior as compared to the static policies like round-robin and GTO. As more and more applications are being ported to the GPU for acceleration, we believe that to cater to the wide array of workloads, application-aware warp scheduling policies will become more relevant in the near future.

CHAPTER 4. WORKLOAD AWARE THREAD BLOCK SCHEDULING

4.1 Abstract

Exploiting thread level parallelism is the primary technique used in Graphics Processing Units (GPUs) to achieve latency tolerance. Each core interleaves execution of several SIMD threads to overlap long latency operations with floating point computations. With each technology node, as the floating point throughput supported by GPUs has continued to increase, GPU architectures have supported execution of an increasing number of concurrently active threads. The thread block (or CTA) scheduler tries to maximize the number of active threads on each core by launching CTAs until core resources are exhausted. The rationale is, more threads would give the warp scheduler on the core more opportunities to hide memory latency and thus result in better performance. In this work, we show that executing the maximum number of threads is not always required to achieve peak performance. GPGPU workloads have an optimal value for number of active threads at which the performance saturates. Increasing the number of threads beyond this value results in no better, and sometimes worse performance. To this end, we design two techniques.

First, we develop Perf-Sat: a mechanism to detect the optimal number of threads to be executed on each core to achieve peak performance. Perf-Sat is integrated into the CTA scheduler and guides it at runtime, to either increase or decrease the number of active threads. We evaluate the performance impact of Perf-Sat on two GPU architecture generations and show that it scales well to different applications as well as architectures. With performance loss of less than 1%, Perf-Sat is able to achieve core resource savings of 18.32% on average.

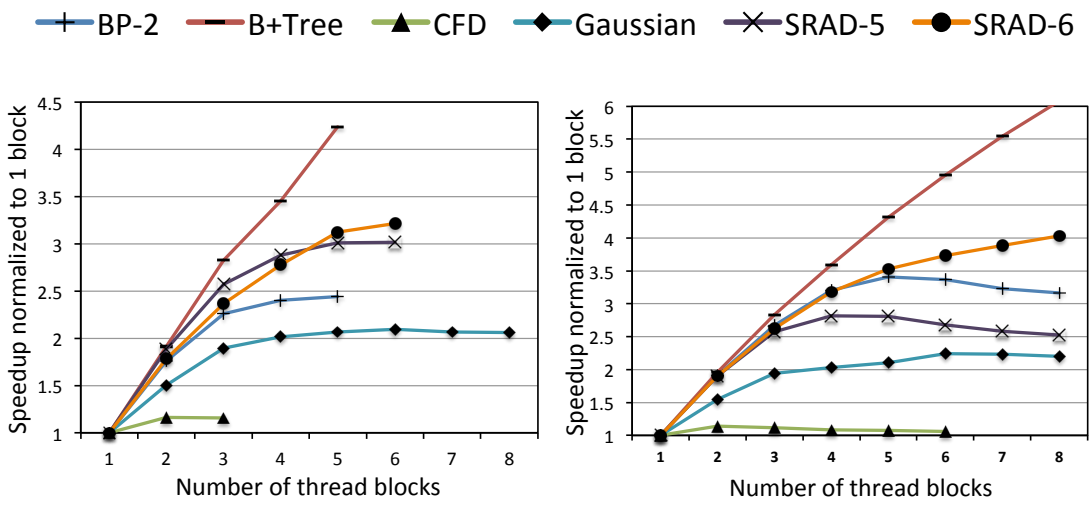
An alternate approach is to execute the workload with maximum number of threads, but on fewer than all cores on the chip. We refer to the number of cores at which performance of

a workload saturates as the optimal number of active cores (Nopt). We propose executing the workload on Nopt cores, and power gating the unused cores to reduce static power consumption. We design ONAC (Optimal Number of Active Cores detector), a hardware unit that detects Nopt at kernel runtime using a novel estimation model. For memory-intensive workloads, ONAC reduces the detection time compared to sequential detection by 45%, and reduces average energy consumption by 20%, with less than 2% performance overhead.

4.2 Introduction

GPGPU programming models use the thread block (CTA) abstraction to enable kernels written for a previous generation GPU architectures to scale to the increased computation resources provided by future generation architectures. The number of CTAs that can be active simultaneously on a GPU core depends on the core's resources, like the register file and shared memory capacity, and number of warp and CTA slots. The peak floating point throughput and memory bandwidth of GPU architectures has been increasing at a steady pace. Consequently, newer generation GPUs provide more resources, thereby supporting a higher number of concurrently active CTAs.

In this work, we show that performance of all GPGPU workloads does not scale uniformly with number of active CTAs. While higher number of concurrently active CTAs benefit if performance of the workload is latency limited, performance of throughput limited workloads either saturates, or can degrade with higher number of active CTAs. Each workload has an optimal active thread count, beyond which increasing the number of threads does not result in further improvement in performance. We design two orthogonal hardware techniques to detect the optimal active thread count of a workload at runtime: executing optimal number of CTAs per core, and executing the workload with an optimal number of active cores. In the next two sections, we describe each of the two techniques in detail.



(a) M2090 (Fermi) (b) K20X (Kepler)

Figure 4.1: Impact of thread count on performance when simulated on configurations matching the NVIDIA Tesla M2090 (a) and K20X (b) GPUs.

4.3 Perf-Sat: Runtime Detection of Performance Saturation for GPGPU Workloads

To observe the effect of number of threads on performance, we simulated various GPGPU kernels with an increasing number of CTAs on GPGPU-Sim, a cycle-accurate GPU architecture simulator [10]. The configurations were chosen to match the NVIDIA Tesla M2090 (Fermi) and K20X (Kepler) GPUs, as they are designed specifically for high performance general purpose computing. Fig. 4.1 depicts the results for six out of the 16 kernels that we used for this work. Three types of workloads are shown in Fig. 4.1(a). One type of workload (SRAD-6 and B+Tree) exhibits continuous improvement in performance as the number of CTAs are increased. For the second type (SRAD-5 and BP-2) performance improves initially and then saturates. Behavior of the third type (CFD and Gaussian) is similar through the saturation point, except the performance drops off as the number of CTAs is increased further. Interestingly, the kernels for which performance saturated on the M2090 (SRAD and BP-2) behave like the third type of workload on the K20X (Fig. 4.1(b)) as now they execute a higher number of thread blocks.

Prior studies have made similar observations [63, 32, 40]. Cache Conscious Wavefront Scheduling [63] proposes a warp scheduling policy that reduces the number of active warps,

when threads from different warps contend for the same L1 data cache set and increase evictions of possibly useful cache lines. DYNCTA [32] tries to find the optimal number of thread blocks by monitoring the core idle and memory wait cycles. However, both techniques use thresholds that are determined heuristically and hence require re-executing the kernels to re-tune their parameter for a different GPU architecture.

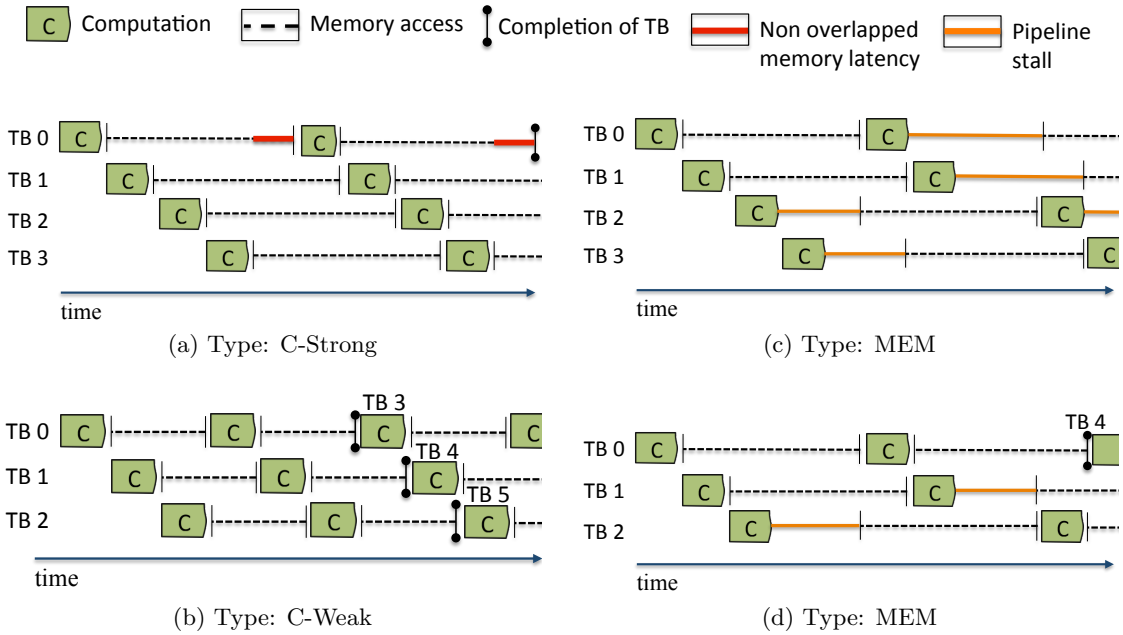
Even for the same GPU architecture, our experiments show that it is not possible to use the same threshold across different workloads. Thus, we propose Perf-Sat: a hardware mechanism that detects the optimal thread block count at runtime without relying on empirically determined thresholds. Per-Sat uses information about core activity and makes decisions based on relative performance at different CTA counts. The thread block count detected by Perf-Sat is used by the thread block scheduler to limit the number of active blocks on each core. The unused core resources opens up opportunities to execute threads from a concurrently active kernel on the same core [7], thereby increasing overall system throughput without having a significant overhead on performance of the first kernel.

4.3.1 Motivation

4.3.1.1 The Need for Thread Level Parallelism

Graphics processing units are designed for high throughput rather than low latency. Hence, they trade-off the optimizations used in CPUs for reducing latency, such as large caches, out of order execution and branch prediction, to provide more die area to computational units. To account for the overhead of increased latency of memory accesses, GPU cores use a technique called Single Instruction Multiple Threads (SIMT) execution [53]. SIMT interleaves execution of multiple SIMD groups, each of which can be at a different instruction.

As mentioned in chapter 2, a SIMD group of threads is called a warp. Warps within one thread block are executed in round-robin order, until all of them stall due to memory access. At this point, warps from the oldest thread block that was scheduled are selected for execution and so on. In this manner, the warp scheduler tries overlap memory access latency of the stalled threads with computation of the active threads. This scheduling technique is known as



(a) Type: C-Strong

(c) Type: MEM

(b) Type: C-Weak

(d) Type: MEM

Figure 4.2: Depiction of memory latency tolerance for different kernel categories.

Greedy Then Oldest. Many other warp scheduling policies exist such as Round Robin, Two-level, etc. We do not describe them in detail here as the work reported in this chapter is neutral with respect to the warp scheduling policy. Further details regarding different warp scheduling policies can be found in chapter 3.

4.3.1.2 Optimal Thread Block Count

The thread block scheduler launches CTAs on a core until this maximum count is reached (N_{max}). The rationale is, more threads would provide the warp scheduler with more opportunities to overlap memory access latency with computation. However, as shown in Fig. 4.1, launching the maximum number of CTAs (and hence threads) does not result in the best performance for all workloads. For some workloads, increasing the number of threads beyond a point does not result in further performance improvement, and for others it might even reduce performance. We call this point, the ***optimal*** thread block count.

In our experiments, we observe three categories of workloads. The effect of thread block count on each category is illustrated in Fig. 4.2. The workloads depicted in the figure have a N_{max} values of four. In Fig. 4.2(a), memory requests of warps from the first block are not

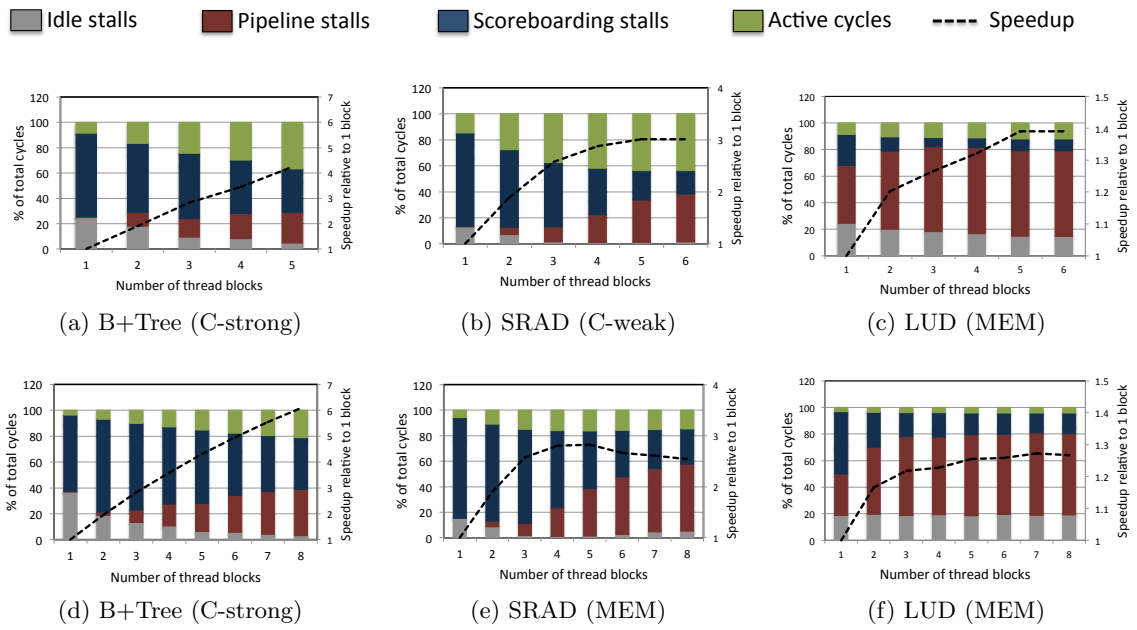
completed even after all the other blocks have completed their computations. The performance of this category of workloads would scale if additional CTAs could be issued. They are referred to as **C-strong**. The workloads typically have un-overlapped memory latency even when executing the maximum number of CTAs. Consequently, the optimal CTA count for these workloads is N_{max}. Fig. 4.2(b), depicts the execution of the second type of workload: **C-weak**. Notice that three blocks are sufficient to overlap all the memory latency. Launching the fourth thread block would not result in a better performance and hence the optimal count is three. Such workloads typically become compute throughput or memory bandwidth limited once the optimal block count is reached. For the third category of workloads (refer Fig. 4.2(c) and 4.2(d)), performance starts to decrease if the number of active blocks is more than the optimal count. In the figures, we can observe that the memory system can support requests from only two blocks. Issuing more blocks, starts increasing pipeline stalls. The optimal count is the number of blocks that can issue memory requests simultaneously, plus one. In this example, it would be three. Such applications are categorized as **MEM**. In the next section, we demonstrate that this behavior of the three types of workloads can be understood by analyzing the breakdown of core activity.

4.3.1.3 Effect of Workload Characteristics on Optimal Thread Block Count

To understand the effect of number of thread blocks on performance, we simulated kernels form a wide range of applications from the Rodinia benchmark suite [14] on a cycle accurate GPU microarchitecture simulator [10]. The thread block scheduler was modified to limit the number of blocks it launches on each core. Kernels were executed from one to N_{max} active blocks and the following architectural parameters were monitored on each core:

Scoreboarding stalls: Cycles when all the warps are stalled due to dependency from a previous instruction. A stall is categorized as either ALU or memory depending on whether the previous instruction is an arithmetic or a memory operation.

Pipeline stalls: Cycles when all the warps are stalled because of insufficient hardware resources. Again, a stall is categorized as ALU if the resource is a computational unit and



(a) B+Tree (C-strong) (b) SRAD (C-weak) (c) LUD (MEM)

(d) B+Tree (C-strong) (e) SRAD (MEM) (f) LUD (MEM)

Figure 4.3: Breakdown of core activity for three types of workloads on the NVIDIA Fermi (above) and Kepler (below) architectures.

memory if the resource is required for a memory operation (e.g. MSHR entry, memory access queue, etc.).

Idle stalls: Cycles when the warp scheduler does not have any warps to issue.

Active: Cycles when the core is performing computations.

Fig. 4.3 illustrates the breakdown of runtime for a workload from each of the three categories at different active CTA counts. Data plotted for scoreboarding and pipeline stalls are attributed to both ALU and memory. For simplicity, we do not distinguish between these in the figure.

C-strong: We can observe in Fig. 4.3(a) that the number of scoreboarding stalls is very high with one block. This is because there are not enough threads to overlap the memory access and ALU latencies. As the number of thread blocks increases, the scoreboarding stalls decrease while the pipeline stalls increase. Notice that the decrease in scoreboarding stalls is always greater than the increase in pipeline stalls, which results in an overall increase in the number of active cycles. This trend continues until the thread block limit is reached. Hence, the optimal count is the same as N_{max} for these workloads. Similar behavior is observed on the Kepler architecture (Fig. 4.3(d)). It should be noted that, while the kernels had a lesser

execution time on the Kepler architecture, the ratio for which the core was active also reduced.

C-weak: Fig. 4.3(b) shows that the performance of SRAD scales initially, but starts to plateau at five blocks. Observe that after five blocks, the decrease in scoreboarding stalls is comparable to the increase in pipeline stalls. This is because, as the number of active threads increases, the warp scheduler has a higher number of ready threads to choose from, which decreases the number of scoreboarding stalls. However, this also increases the contention for hardware resources, which results in an increase in the number of pipeline stalls. Due to this opposing effect, the number of active cycles does not increase, and the performance saturates after the optimal count value.

MEM: Similar to the C-weak category of workloads, the performance of workloads in this category also saturates when the decrease in scoreboarding stalls becomes comparable to the increase in pipeline stalls. However, beyond this point the performance starts to degrade. Observe in Fig. 4.3(c) that performance of LUD drops off when the number of blocks is increased beyond five. We observed that this is due to an increase in L1 data cache contention among warps, which causes an increase in the L1 data cache miss rates. The effect is more prominent on the Kepler configuration, as the L1 cache size is the same and the number of warps executing simultaneously increases. This causes SRAD, which is a C-weak type of workload on Fermi, to becomes a MEM workload on the Kepler configuration (Fig. 4.3(e)).

Our work is motivated by these observations. Detecting the optimal thread block count for C-weak and MEM workloads can reduce the amount of core resources required by a kernel, without sacrificing performance. These resources can either be power-gated or utilized for issuing threads from other workloads in concurrent kernel execution scenarios. In the next section, we explain how Perf-Sat utilizes the information regarding core stalls to detect the optimal thread block count.

4.3.2 Perf-Sat - Underlying Principles and Design Details

4.3.2.1 Hypothesis

The design of Perf-Sat is based on the behavior presented in the previous section. The key observation is that, performance improves until the increase in the number of pipeline stalls is lesser than the decrease in the number of scoreboard related stalls. Lets consider a workload that is executing with N active blocks. Using N+1 active blocks would be better if,

$$Pipeline_{N+1} - Pipeline_N < Scoreboard_N - Scoreboard_{N+1}$$

Thus, N+1 active blocks are a better choice, if the sum of the pipeline and scoreboarding stalls is lower than at N active blocks. We refer to this sum as the stalled cycle count in the rest of this section. Our thread block scheduler issues $\lceil N_{max}/2 \rceil$ CTAs on each core at initialization. At each sample, the number of active CTAs is adjusted until the stalled cycle count is greater than at the previous CTA count.

4.3.2.2 Detection Algorithm

The detection algorithm has three phases:

1. Sample rate detection: As our scheduler makes decisions based on samples of the core activity at runtime, its accuracy is sensitive to the sampling rate. For example, a sample of the core activity when the workload was very compute intensive should not be compared against one that had many memory access instructions. As discussed in chapter 3, GPGPU kernels typically exhibit such phase behavior within warps. The effect of workload phases gets averaged across multiple thread blocks. Thus, we set the sampling period to approximately the number of cycles it would take for N_{max} thread blocks to complete. When work from a new kernel is started on a core, the number of cycles that elapsed until the first thread block completes (One_TB_{cycles}) are recorded. The sampling period is then set as $One\_TB_{cycles} * N_{max}$.

Decisions in the next two phases are made according to the state machine shown in Fig. 4.4. The state machine has four states: weak-increase, strong-increase, weak-decrease and strong-decrease. The stalled cycle count for the previous block count is stored (previous sample). At the end of a sample period, Perf-Sat compares the stalled cycle count for the current thread

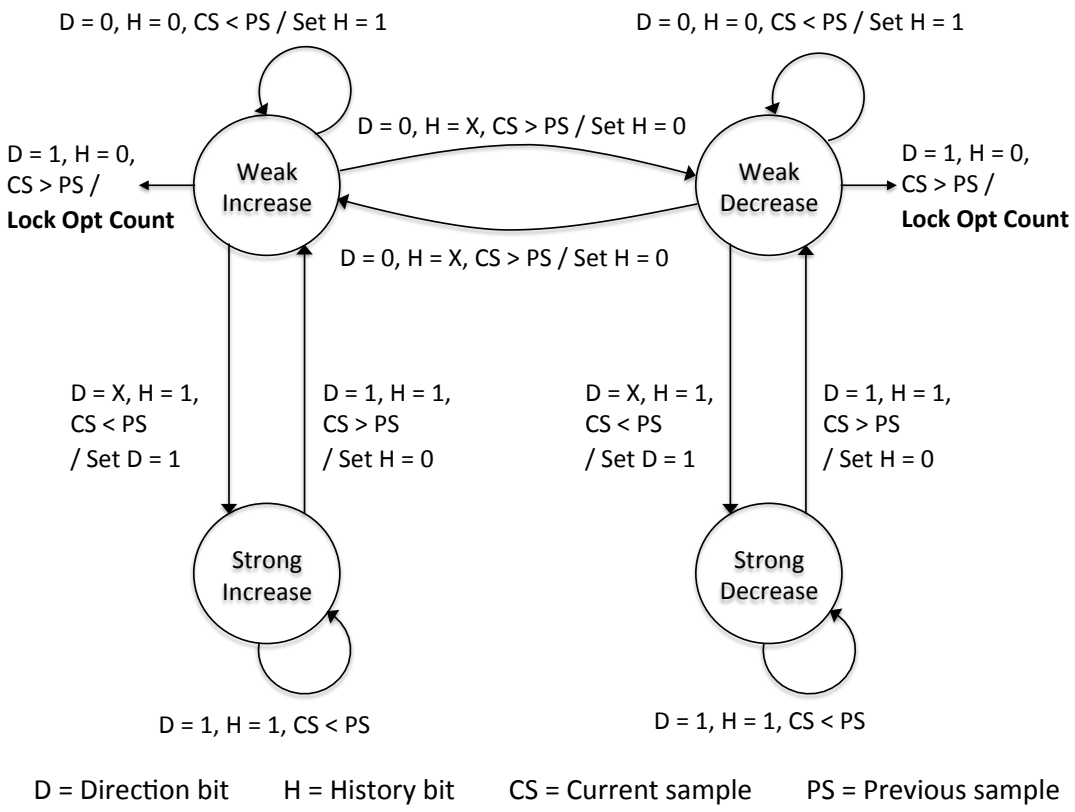


Figure 4.4: State machine used by Perf-Sat

block count (current sample) with the previous sample. A direction bit is used to indicate whether a particular direction of throttle has been decided. A history bit is used to distinguish between the weak and strong states.

2. Direction of throttle: The optimal thread block count can be either higher or lower than the initial value of $\lceil N_{max}/2 \rceil$. Hence, the direction in which we should change the number of active blocks is decided in the second phase. At initialization, the direction bit is set to 0. At the end of the first sample period, the stalled cycle count for $\lceil N_{max}/2 \rceil$ is stored as the previous sample and the number of blocks is increased to $\lceil N_{max}/2 \rceil + 1$. The Perf-Sat state machine goes into the weak-increase state.

At the end of the second sample period, if the stalled cycle count is lesser than the previous sample, it is considered that the decision to increase was correct. Perf-Sat remains in the weak-increase state and the history bit is set to 1. The thread block count remains as $\lceil N_{max}/2 \rceil + 1$. On the other hand, if the stalled cycle count is greater than the first sample, Perf-Sat goes into

the weak-decrease state. The sample corresponding to $\lceil N_{max}/2 + 1 \rceil$ is stored and number of active blocks is reduced to $\lceil N_{max}/2 \rceil$.

At the next sample, if the number of stalled cycles is coherent with the decision taken, Perf-Stat goes into the strong-increase (or strong-decrease) state and the direction bit is set. If not, the history bit is reset again. Thus, the history bit ensures that a direction of throttle is decided only if two consecutive decisions favor the same direction. If the optimal thread count value is close to $\lceil N_{max}/2 \rceil$, Perf-Sat can toggle several times between the weak-increase and weak-decrease states. The optimal count is conservatively set as $\lceil N_{max}/2 + 1 \rceil$, if the state machine toggles more than three times.

3: Detection of optimal value

Once the state machine is in the strong-increase (or strong-decrease) state, Perf-Sat keeps increasing (or decreasing) the number of active blocks at the end of each sample period. The stored sample is now updated at each sample point. This continues until the stalled cycle count of the current sample is more than the previous sample. For example, lets say an application has $N_{max} = 15$ and $N_{optimal} = 12$. Perf-Sat starts with N = 8. It takes two samples at N = 9 to set the direction bit and goes into the strong-increase state. After N = 9, the number of active blocks is increased at every sample point until 12.

The number of stalled cycles at N = 13 would be more than at N = 12. At this point; the N_{13} sample is discarded, Perf-Sat goes into the weak-increase state and number of active blocks are kept as 13. The history bit is set to 0. At the next sample, if the stalled cycle count is again more than the N_{12} sample, the optimal block count is detected as 12. Similar to the problem mentioned in the previous section, just before converging at $N_{optimal} = 12$, Perf-Sat can toggle several times between the weak-increase and strong-increase states. The optimal block count is optimistically set as the previous value (12 in this example) if the state machine toggles more than 3 times.

Table 4.1: Details of GPU configurations used for evaluating Perf-Sat

Resource	M2090(Fermi)	K20X (Kepler)
Register file capacity per SM	128 KB	256 KB
Maximum threads supported per SM	1536	2048
Maximum thread blocks supported per SM	8	16
SP floating point units per SM (total)	32(512)	192(2688)
DP floating point units per SM (total)	16(256)	64(896)
SP floating point performance (peak)	1330 GFLOPS	3935 GFLOPS
DP floating point performance (peak)	665 GFLOPS	1311 GFLOPS
DRAM clock frequency (MHz)	1850	2600
DRAM peak bandwidth (GB/s)	177	250

Table 4.2: Details of CUDA kernels used for evaluating Perf-Sat

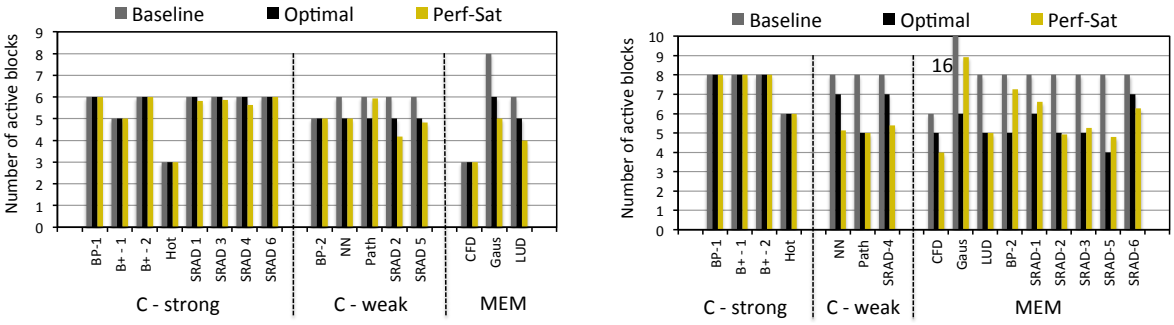
Kernel	Resource per block		M2090(Fermi)			K20X (Kepler)		
	Reg (Bytes)	Threads	Max	Opt	Type	Max	Opt	Type
Backprop - 1 (BP-1)	12228	256	6	6	C-strong	8	8	C-strong
Backprop - 2 (BP-2)	24576	256	5	5	C-weak	8	5	MEM
B+Tree - 1 (B+-1)	24576	256	5	5	C-strong	8	8	C-strong
B+Tree - 2 (B+-2)	20480	256	6	6	C-strong	8	8	C-strong
Comp. Fluid Dyn. (CFD)	39936	192	3	3	MEM	6	5	MEM
Gaussian (Gaus)	1536	16	8	6	MEM	16	6	MEM
LU Decomposition (LUD)	16384	256	6	5	MEM	8	5	MEM
Hotspot (Hot)	36864	256	3	3	C-strong	6	6	C-strong
Pathfinder (Path)	16384	256	6	5	C-weak	8	5	C-weak
Nearest neighbor (NN)	16384	256	6	5	C-weak	8	7	C-weak
SRAD - 1	12288	256	6	6	C-strong	8	6	MEM
SRAD - 2	12288	256	6	5	C-weak	8	5	MEM
SRAD - 3	16384	256	6	6	C-strong	8	3	MEM
SRAD - 4	20480	256	6	6	C-strong	8	7	C-weak
SRAD - 5	20480	256	6	5	C-weak	8	4	MEM
SRAD - 6	12288	256	6	6	C-strong	8	7	MEM

4.3.3 Experimental Results

4.3.3.1 Experimental Setup

Our experiments were performed on GPGPU-Sim v3.1 [10], a cycle accurate GPU microarchitecture simulator. The simulator configurations were chosen to closely match the NVIDIA Tesla M2090, which is based on the Fermi architecture, and the Tesla K20X, which is based on the Kepler architecture. Details of both the configurations are provided in Table 4.1.

We simulated a wide range of CUDA kernels from the Rodinia benchmark suite [14]. We show results for 16 kernels that represents a good mix of the three types of workloads: C-strong, C-weak and MEM. Table 4.2 provides more details about the workloads executed by the kernels. Numbered kernels are from applications that launch more than one kernel.



(a) Tesla M2090 (b) Tesla K20X

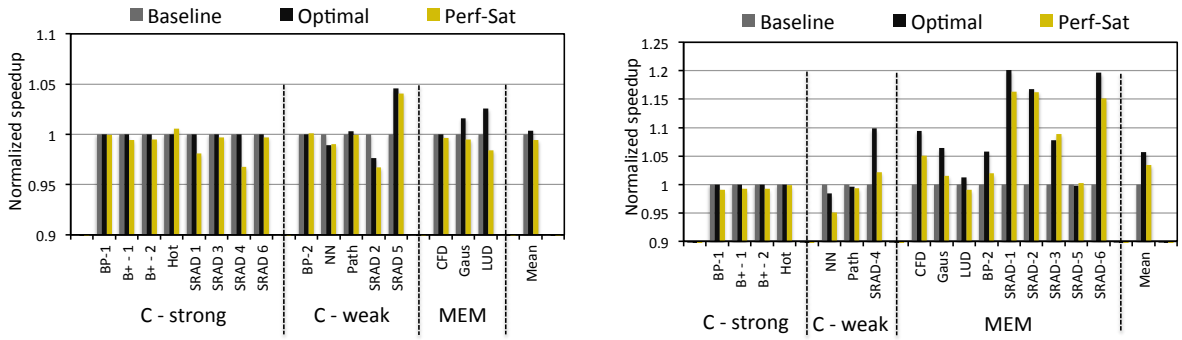
Figure 4.5: Comparison of active thread blocks detected by Perf-Sat, with baseline and empirically found optimal count.

4.3.3.2 Detection Accuracy

To find the optimal thread block count, kernels were executed separately at each count by modifying the thread block limit per core in the simulator. The count after which increasing the number of blocks resulted in less than 2% performance improvement was decided as optimal. Fig. 4.5 compares this empirically determined optimal count with the count detected by Perf-Sat. The baseline thread block scheduler issues the maximum number of blocks supported. As our mechanism operates separately for each core, the number reported in the figure is the average across all cores. Perf-Sat finds the optimal count with an accuracy of 94.25% on the Fermi configuration, and with 85.12% on the Kepler configuration. There are two reasons for the difference between the detected and the empirically found optimal values. Kernels for which the optimal count is close to the maximum, Perf-Sat detects the optimal count as maximum on some cores and lower than the empirically found optimal on other cores. Secondly, for the cores where the optimal count is close to $N_{max}/2$, Perf-Sat conservatively chooses the higher block count value.

4.3.3.3 Performance Impact

Fig. 4.6 compares the performance of Perf-Sat with baseline. The difference between Perf-Sat and the baseline scheduling policy is that Perf-Sat throttles the number of active blocks in the beginning to detect the optimal count, while the baseline scheduler always issues



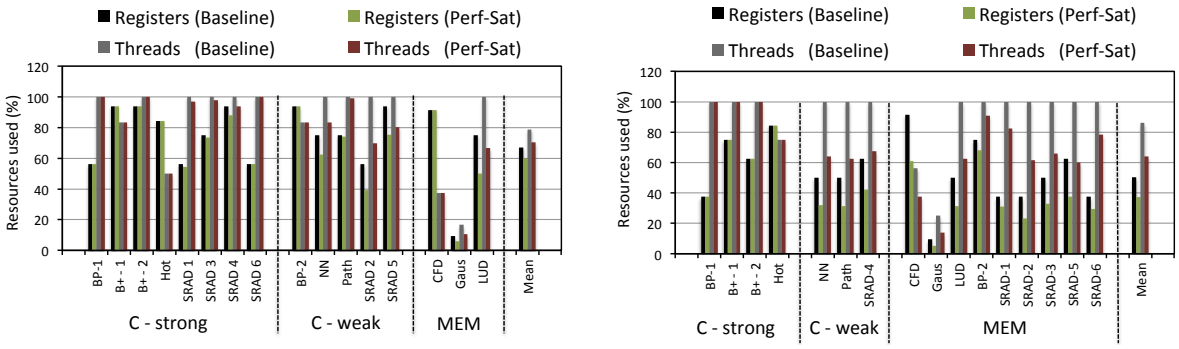
(a) Tesla M2090 (b) Tesla K20X

Figure 4.6: Performance comparison between baseline, optimal and Perf-Sat.

the maximum number of blocks possible. The speedup reported under Optimal is using the execution time when the kernel is executed with the empirically found N_{opt} CTAs. As expected, we incur some loss in performance for the C-strong and C-weak category of kernels. This is because, their optimal block count is close to the maximum. Perf-Sat starts with $N_{max}/2$ number of blocks and takes a few iterations to reach the optimal count. However, the performance loss is only 0.51% on the Fermi configuration and 0.88% on the Kepler configuration. The performance of the MEM category of workloads increases because the optimum count is much less than the maximum. There is a 4.95% performance improvement on average for these workloads across the two configurations.

4.3.3.4 Resource Utilization

Fig. 4.7 shows the percentage of resources utilized by each kernel with the baseline and Perf-Sat scheduling policies. As the maximum block count for all of our kernels was constrained due to either thread slots or registers, results are shown for only these two resources. For the C-strong category kernels, Perf-Sat utilizes only 0.97% lesser resources on average compared to the baseline scheduler. This is expected as the optimal count value is the same as maximum for these workloads. The resource savings are more significant for the other two categories of workloads. Perf-Sat utilizes 14.32% and 35.35% less resources compared to the baseline for the C-weak workloads on the M2090 and K20X configurations respectively. For the MEM category,



(a) Tesla M2090 (b) Tesla K20X

Figure 4.7: Comparison of resource utilization per core.

the reduction in resource usage is 25.31% on the M2090 and 31.48% on the K20X configuration. Across all kernels, the savings are 10.45% on the M2090 and 25.51% on the K20X. Average resource savings across the configurations is 18.32%.

4.3.4 Related Work

This section gives on overview of related work in GPGPU scheduling techniques and workload throttling.

4.3.4.1 Scheduling in GPGPU

There have been several prior works that have proposed scheduling techniques at the warp level. To increase the overlap of memory latency with computation, Narasiman et al. [48] proposed the two-level warp scheduling scheme. They group a fixed number of warps into fetch groups. Warps are scheduled in round-robin order from one fetch unit until all the warps stall on a memory request, and then the next fetch group is brought into the scheduler. Fetch groups are selected in round robin order as well. Gebhart et al. [22] proposed a similar hierarchical scheduling policy, along with a register file cache. The objective of their work was to improve the energy efficiency of the warp scheduler and instruction dispatch. Rogers et al. [63] suggested that using a greedy then oldest (GTO) policy for selecting fetch groups, as well as for selecting warps within a fetch group, performs better than round-robin. Jog et al. [30] used the two level scheduler to improve the efficiency of prefetching for GPUs. They group non-consecutive

warps into one fetch group, which then prefetch for the next group. As warps for which data is being prefetched are scheduled in the next group, the amount of time warps are blocked due to memory access is reduced.

4.3.4.2 Workload Throttling

It has been suggested in a few prior works that increasing number of threads beyond a point can degrade performance for some workloads. Bakhoda et al. [10] varied the amount of core resources from 25% to 200% and observed that two of the workloads they used showed a decrease in performance. In their work on cache conscious warp scheduling [63], Rogers et al. showed that high thread count can degrade performance for cache sensitive workloads. They track L1 data cache lines to detect warps that have lost intra-warp data locality, possibly because of other warps evicting data lines that would have been used by them. A scoring system increases the priority of such warps. Warps that have a score below a certain threshold are not scheduled, thereby reducing the number of active threads. For cache-sensitive workloads, their warp scheduler effectively reduces the active thread count close to the optimal count. We attempt to find this count at the CTA scheduler.

Our work is closest to [32] and [40]. Kayiran et al. [32], monitor the core idle cycles (C_{idle}) and cycles when all threads are waiting for memory access (C_{mem}). They start by issuing $\lceil N_{max} \rceil$ thread blocks. They increase the number of blocks if C_{idle} is higher than a threshold. Once, C_{idle} is less than the threshold, they further increase (or decrease) the block count if C_{mem} is lower (or higher) than another threshold. Values for both the thresholds, as well as the sampling period are found empirically.

In [40] Lee et al. make similar observations as ours. Their technique launches N_{max} blocks at initialization and use a greedy warp scheduling policy to issue work. The number of instructions executed until the first thread block completes are counted. $N_{optimal}$ is estimated as $\lfloor N\_max * (instr.\ in\ 1\ block)/instr.\ executed \rfloor$. The idea is the amount of computation done to overlap latency of the first block should be enough as now a new block can be issued in its slot. Thus, the number of thread blocks required to execute those computations (calculated by the earlier equation) is detected as the optimal active thread block count.

4.4 ONAC: Optimal Number of Active Cores Detector for Energy Efficient GPU Computing

Our experiments show that several GPGPU workloads can achieve their peak performance without utilizing all the cores on the chip. Prior works have shown that, memory-intensive workloads can be executed with fewer than maximum supported threads per core without sacrificing performance [8, 40, 32]. Consequently, they proposed techniques that detect the minimum number of threads required on each core to achieve peak performance. Executing fewer than the maximum threads reduces hardware resource utilization, which enables opportunities for reducing energy consumption through power-gating.

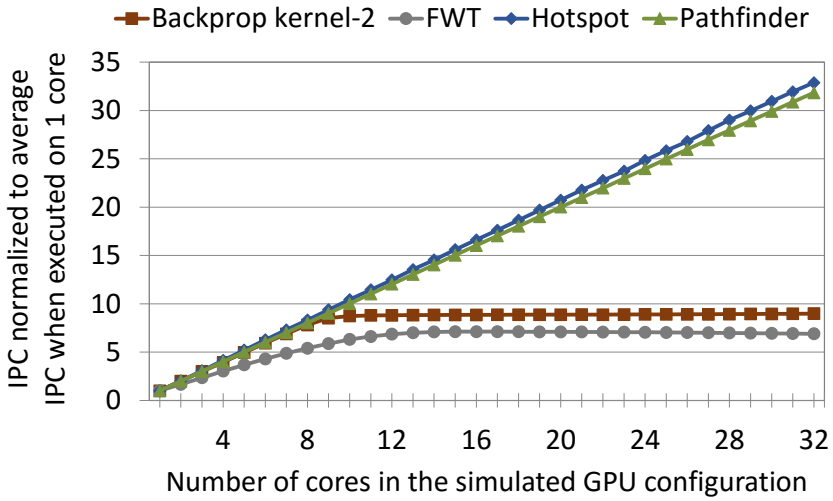


Figure 4.8: The effect of number of cores on performance.

An orthogonal technique to reduce energy consumption is to execute the workload on fewer cores. To observe the effect of number of cores on performance, we executed workloads from the Rodinia benchmark suite [14] and the CUDA SDK [51] with an increasing number of active cores, on a 32-core GPU simulated in GPGPU-Sim [10]. Fig. 4.8 plots the average IPC at each core count for 4 kernels. Two clear performance trends can be observed. The performance of Hotspot and Pathfinder scales linearly with number of cores. Contrary to this, performance of Backprop and FWT scales linearly only for lower core counts. Beyond a certain number of cores, scaling starts to plateau and performance eventually saturates. This shows that for Backprop-K2 and FWT, some of the cores on this chip can be power-gated without adversely

affecting performance.

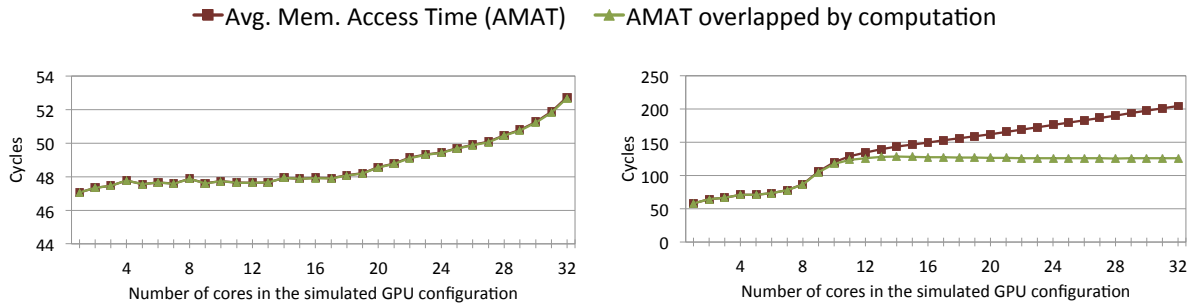
We refer to the number of active cores at which performance of an application saturates as its **optimal active core count**. The authors in [67] propose a mechanism to detect the optimal active core count at runtime. Memory-intensive workloads typically have high average memory latency due to DRAM queuing effects. Their technique marks a core as memory-sensitive if the average memory latency (sampled at runtime) is larger than an empirical threshold. The number of active cores are reduced sequentially (one per sample), until more than half the active cores are not memory-sensitive. Our experiments show that there are two drawbacks of this approach:

1. Reducing the number of active cores one at a time leads to a long detection time for workloads that have a low optimal active core count. As all cores consume static power during the detection period, a long detection time leads to lost opportunity for reducing energy consumption.

2. Memory latency is not a direct indicator of performance. Instead, the amount of latency overlapped by compute instructions is a better indicator. Moreover, the capability to hide memory latency varies across workloads, and consequently a single threshold does not scale across a variety of workloads. We implement the technique in [67] by directly using IPC as an indicator of performance, and comparing samples across cores instead of using a threshold. We refer to our implementation as sequential detection or Seq-Det.

To address these problems, we design ONAC: Optimal Number of Active Cores detector. ONAC uses an estimation model inspired by Roofline [76], and estimates the IPC versus core count curve described in Fig. 4.8. As the optimal active core count is estimated instead of searched by reducing active cores one at a time, ONAC significantly reduces detection time and increases energy saving compared to Seq-Det. The specific contributions of our work are:

- We thoroughly analyze the impact of number of cores on performance via case studies on kernels from two real-world applications. We show that performance saturation at a certain core count can be explained by measuring the amount of memory latency that is overlapped by computation.



(a) Pathfinder (b) Backprop

Figure 4.9: Average Memory Access Time (AMAT) and the amount of AMAT overlapped by computation.

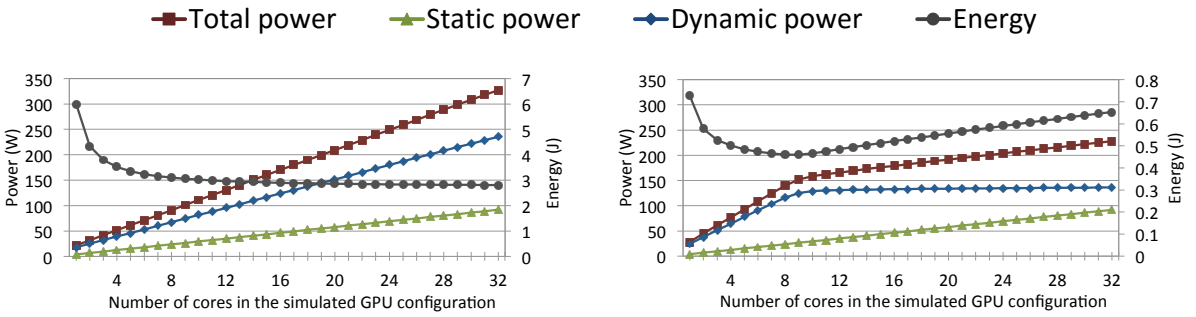
- We propose a novel model to estimate the optimal number of active cores at runtime.
- We implement ONAC and Seq-Det in GPGPU-Sim [10], and analyze their effect on performance, power and energy consumption. Our evaluation shows that for memory-intensive workloads, Seq-Det and ONAC reduce energy consumption by 10% and 20% respectively, with negligible impact on performance.

4.4.1 Effect of Number of Cores on Performance

In this section we analyze the effect of the number of active cores (N_{active}) on performance for kernels from two applications: Pathfinder and Backprop. The Pathfinder kernel is compute-intensive, and reaches peak performance when all cores on the GPU are utilized. On the contrary, the Backprop kernel is memory-intensive, and its performance saturates after a specific number of active cores.

4.4.1.1 Effect on Memory Latency and Performance

The fundamental effect of N_{active} on performance can be deduced by analyzing its effect on non-overlapped memory latency (Fig. 4.9). Average memory access time (AMAT) is typically defined as the average memory latency observed by a memory instruction, irrespective of whether it hits in the L1 or L2 data caches. Fig. 4.9 plots the AMAT observed for the Backprop and Pathfinder kernels, at different core counts, normalized to the run on a configuration with just one core. As expected, as N_{active} increases, the bandwidth available to each core reduces,



(a) Pathfinder (b) Backprop

Figure 4.10: The effect of N_{active} on power and energy.

and consequently AMAT increases. Notice in the figure that AMAT increases for both kernels. Thus, by itself, AMAT is not an indicator of performance.

Fig. 4.9 also plots the amount of AMAT that is overlapped by computation ($AMAT_{overlapped}$). $AMAT_{overlapped}$ is the average number of cycles, for each memory request, when there is at least one arithmetic instruction in flight on its requesting core. Observe in Fig. 4.9(a) that for Pathfinder, $AMAT_{overlapped}$ increases with AMAT. Consequently, performance of Pathfinder keeps scaling until N_{active} equals 32. On the contrary, for the Backprop kernel, the difference between AMAT and $AMAT_{overlapped}$ starts to increase after N_{active} is higher than 9 (Fig. 4.9(b)). This indicates that an increasing amount of memory latency is exposed and contributes to the execution time. Thus, the performance of Backprop starts to plateau around this point and eventually saturates when N_{active} reaches 12 (refer to Fig. 4.8). The threshold used to categorize a kernel's performance as saturated is when it is within 2% of the performance when N_{active} equals 32.

4.4.1.2 Effect on Power and Energy

Fig. 4.10 plots the static and dynamic power consumed by the two kernels with a varying number of active cores. While static power increases linearly with N_{active}, dynamic power consumption scales proportionally to IPC. As discussed in the previous subsection, Pathfinder's IPC continues to scale up until 32 cores. Consequently, dynamic power continues to increase as well. On the contrary, Backprop spends an increasing percentage of the total execution time on

non-overlapped memory latency as N_{active} increases beyond 9 cores. As cores stay idle waiting for data for a larger portion of their execution time, the average dynamic power consumed per core reduces, and becomes almost constant after N_{active} is higher than 12 cores (Fig. 4.10(b)).

The energy consumed by a kernel is directly proportional to average power consumption and inversely proportional to IPC. Hence as N_{active} increases, if the increase in IPC is larger than the increase in power consumption, total energy consumption reduces. Notice in Fig. 4.10(a) that although the average power consumption of the Pathfinder kernel continues to increase until $N_{active} = 32$, energy consumption continues to decrease. This is due to a steady increase in IPC. On the contrary for Backprop (Fig. 4.10(b)), as the average power consumption continues to increase (due to increase in static power), and IPC saturates after $N_{active} = 12$, the total energy increases.

4.4.2 ONAC: Estimation Model and Hardware Implementation

In the previous section, we showed that some GPU kernels can achieve peak performance without using all cores on the chip. The unused cores can be power gated to reduce static power consumption, and thereby energy. In this section we describe ONAC, our hardware technique that detects the optimal core count at runtime. At its crux is our novel estimation model that significantly reduces the detection overhead.

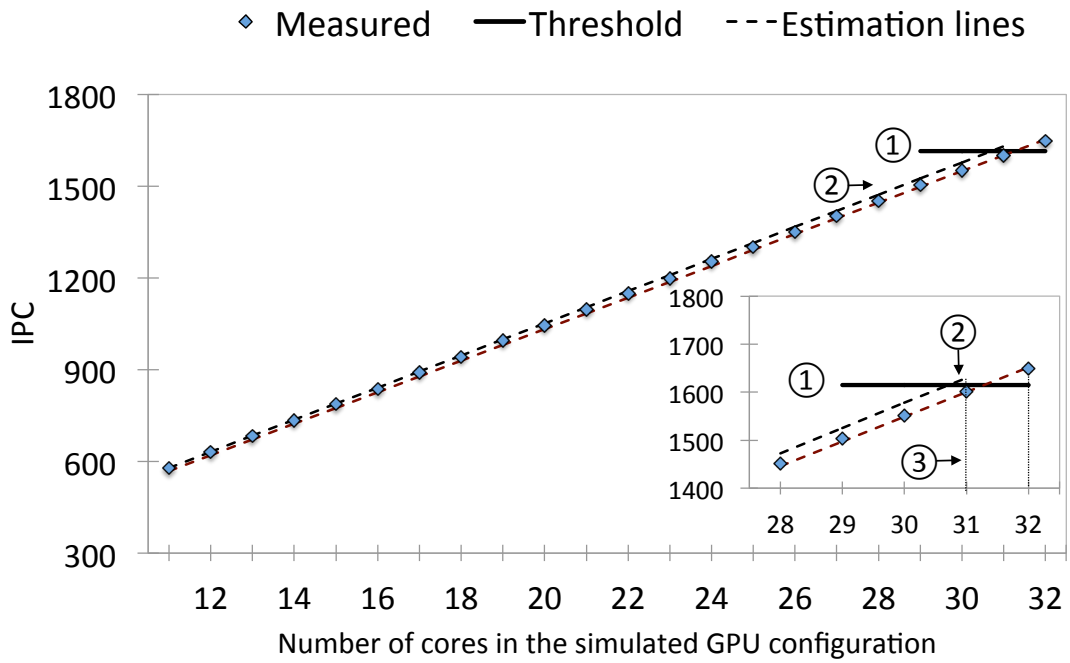
4.4.2.1 Performance Estimation Model

Our estimation model leverages the following observations from the IPC trends described in the previous sections:

Observation 1: IPC achieved when all cores are used (IPC_{all}), is the maximum IPC of the kernel on the current configuration.

Observation 2: Let IPC_1 be the IPC achieved on the 1 core configuration. The IPC at a given core count N, is lower than or equal to N * IPC_1.

Observation 3: As the number of cores increase, IPC per core remains the same or decreases. For a configuration with x cores, let the average IPC per core be $IPC_{slope\,x}$. For any i > j, $IPC_{slope\,i} \leq IPC_{slope\,j}$.



(a)

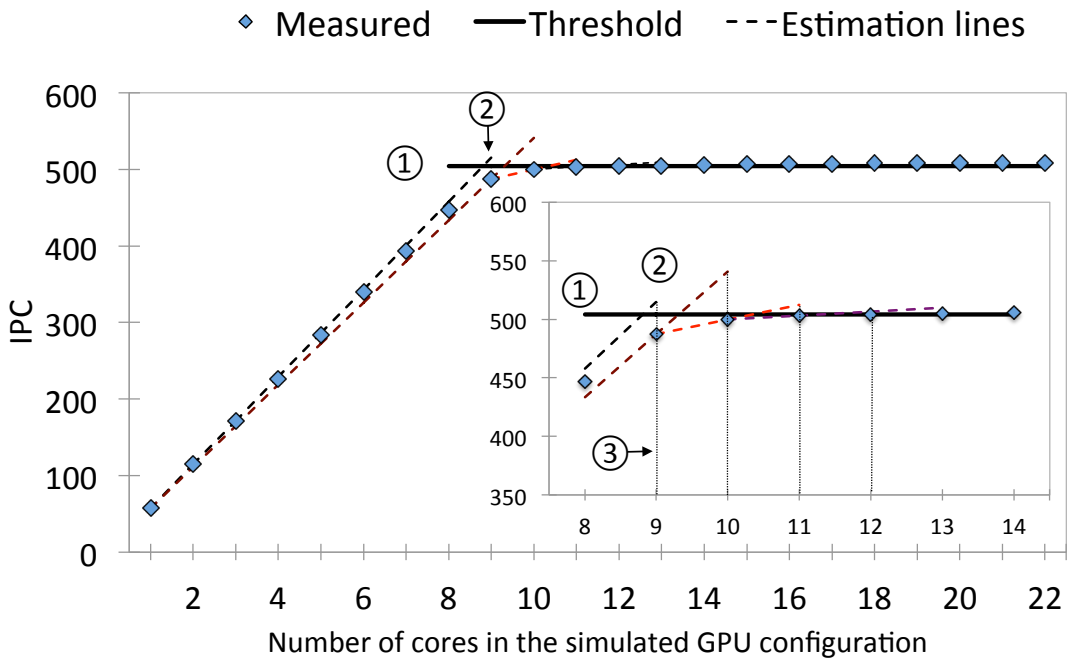
Figure 4.11: Example of optimal active core count detection using our model for the Pathfinder kernel.

In the next subsection, we describe how our model uses these observations and illustrate the steps it takes to estimate the optimal active core count.

4.4.2.2 Optimal Core Detection Examples

At the beginning of a kernel's execution, a sample of IPC with all the cores active is taken (IPC_{all}). Following **observation 1**, our model assumes IPC_{all} to be the maximum achievable IPC for this kernel. Its objective is to find the minimum number of cores required to achieve an average IPC within a certain threshold of IPC_{all}. We refer to this as $IPC_{threshold}$, and is depicted by a solid line in Fig. 4.11 ①.

Next, all but 1 cores are put in a *paused* state and a sample of the IPC is taken with just 1 core active. The details of pausing cores and sampling are described in the next section. Following **observation 2**, the maximum performance at each core count can be projected by a line through the origin with a slope of IPC_1 ②. The intersection of this line with the constant line through $IPC_{threshold}$ is the least number of cores required to achieve peak IPC. This gives



(a)

Figure 4.12: Example of optimal active core count detection using our model for the Backprop kernel.

the model a first estimate of the optimal number of active cores. The first estimate calculated for the Pathfinder and Backprop kernels is 31 and 9 cores respectively (refer to Fig. 4.11 and Fig. 4.12 ③).

An estimate of the optimal active core count is referred to as N_{est}. At this point, a sample of the average IPC is taken with N_{est} cores active (IPC_{est}). If the IPC_{est} is greater than or equal to $IPC_{threshold}$, the model concludes that the optimal number of cores has been detected. As expected, the first estimate is very optimistic. The IPC of Backprop with 9 cores and Pathfinder with 31 cores do not meet the required performance threshold.

All the remaining estimates are calculated using **observation 3**. The model stores two values: a sample of the IPC at the current (IPC_{cur}) and previous (IPC_{prev}) estimates. The next estimate is calculated as the point at which a line passing through IPC_{prev} and IPC_{cur} intersects with $IPC_{threshold}$. For Backprop, a line through IPC_1 and IPC_9 intersects $IPC_{threshold}$ at 10 cores. The IPC sampled at 10 cores is still lower than $IPC_{threshold}$, and a new estimate is calculated using IPC_9 and IPC_{10}. In this way, the model keeps updating IPC_{cur} and IPC_{prev}

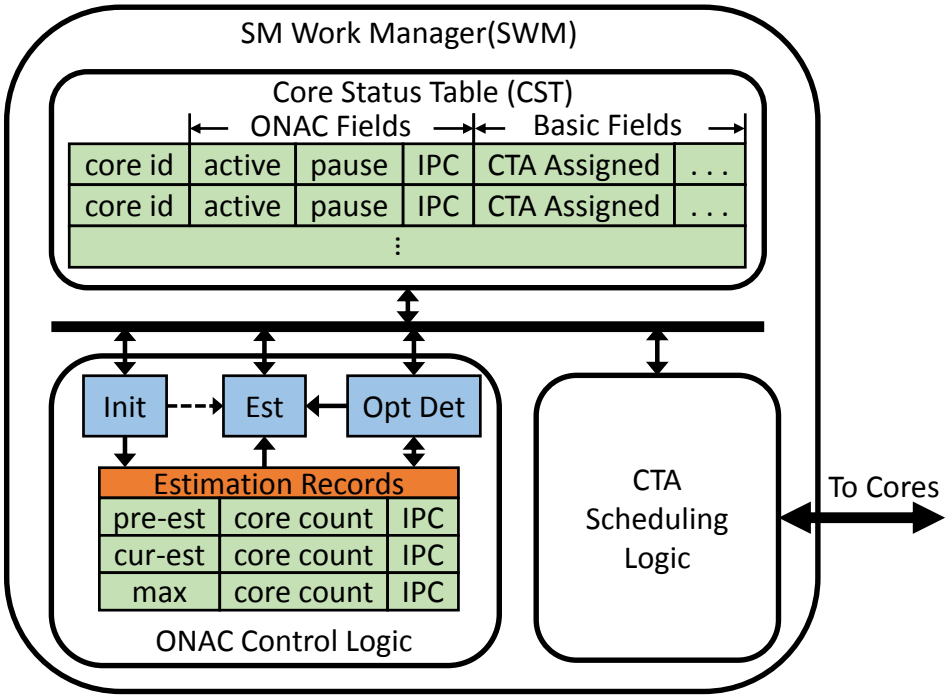


Figure 4.13: Block diagram of ONAC's hardware implementation.

until the IPC at the current estimate satisfies $IPC_{threshold}$. Our model converges at the fourth estimate for Backprop, and correctly detects the optimal core count as 12 cores. For Pathfinder, the optimal active core count is correctly detected as 32 cores at the second estimate.

4.4.2.3 Hardware Implementation

We implemented ONAC inside the kernel scheduler described in chapter 2. We refer to the kernel scheduler unit as the SM Work Manager (SWM) in this section. Fig. 4.13 illustrates a simplified block diagram. The unit inside the SWM which detects the optimal core count is shown as the ONAC control logic.

Core Status Table The baseline SWM stores information required for scheduling in a structure referred to as the Core Status Table (CST). We add three new fields per core to this structure: active bit, paused bit and a field to record the IPCs sampled at runtime. CTAs are issued only to cores which have the active bit set and paused bit unset.

ONAC Control Logic The control logic in Fig. 4.13 illustrates a state machine representation of the detection algorithm. The Estimation Records (ER) table has two entries each at the

previous estimate, current estimate and all cores active, to store the core count and average IPC. The state machine has three states: Init, Estimate (Est) and Optimal core count detection (Opt Det).

1. Init: At the beginning of a kernel, all cores are active and un-paused. Each core takes a sample of its IPC and stores it in the CST (refer to Sect. 4.4.2.4). When all the cores have taken their samples, the IPC is summed and stored in the ER. Next, the pause flag is set for 31 cores (refer to Sect. 4.4.2.5). Once all cores are paused, a sample of IPC with one core is taken and Init invokes the **Est** logic to calculate the first estimate.

2. Est and Opt Det: The detection state machine stays in either Est or the Opt Det states for the remaining portion of the kernel. **Est** calculates the first estimate (N_{est}) using the samples of the IPC at max and 1, and stores it in the cur-est field of the ER. To get a sample of the IPC, N_{est} - 1 cores are un-paused by resetting their pause bits. Once the sample with N_{est} active cores is collected, **Opt Det** compares it with IPC for max and checks if the performance threshold is satisfied. If true, N_{est} is set as the optimal number of active cores and the detection process is terminated. If false, pre-est and cur-est entries are updated and **Est** calculates a new estimate.

4.4.2.4 Sampling

At the start of a sample, the SWM sends a request to all cores that are active but not paused. The requested cores wait for the instructions in-flight to complete, and then take a sample of the IPC over the next sampling period. Our experiments show that sampling period is a significant factor that affects the accuracy of the IPC samples, and consequently the accuracy of detection.

Sampling period: The total number of CTAs concurrently active on the GPU is referred to as a CTA wave. GPU compute kernels typically execute a similar set of instructions on a grid of input data, organized into CTAs. Consequently, workload of a GPGPU kernel can be approximated using a sequence of CTA waves. However, as all CTAs in a wave do not start and finish at the same time, the IPC fluctuates over the execution of a CTA wave. Our experiments

show that the average over a set of four CTA waves is a good sampling period and results in a stable IPC value that is close to the kernel's average IPC.

4.4.2.5 Core Pausing

During the detection period, the cores that are inactive are put in a paused state. To pause a core, the SWM sends a pause request, and the warp scheduler on the core stops issuing instructions. Once the instructions that are in-flight are completed, the core sends an acknowledgment to the SWM. Once all the required cores have paused, the SWM notifies the active cores to begin the next sample. We pause cores at the beginning to take a sample with 1 core. For the remaining portion, cores are un-paused and put back into the active state as the estimate of optimal core count increases. Pausing the cores instead of draining them, helps reduce the time required to wait before beginning the next sampling period. After the optimal core count is detected, the cores that are paused, are drained and power-gated.

4.4.3 Experimental Results

Two metrics are important in the design of a runtime technique for detecting optimal active core count (N_{opt}):

1: Accuracy: The accuracy of detection is important as overestimating N_{opt} reduces the amount of energy saved, while underestimating it negatively impacts performance.

2: Detection Time: A short detection time is important as it increases the amount of energy saved for memory-intensive kernels. Moreover, a long detection time can negatively impact performance, particularly for compute-intensive kernels.

In this section, we evaluate ONAC and the sequential detection technique (Seq-Det) on the basis of accuracy, detection time and their impact on performance, power and energy.

4.4.3.1 Methodology

We implemented ONAC and Seq-Det in GPGPU-Sim, a cycle level GPU architecture simulator [10]. Table 4.3 provides details of the simulated GPU configuration.

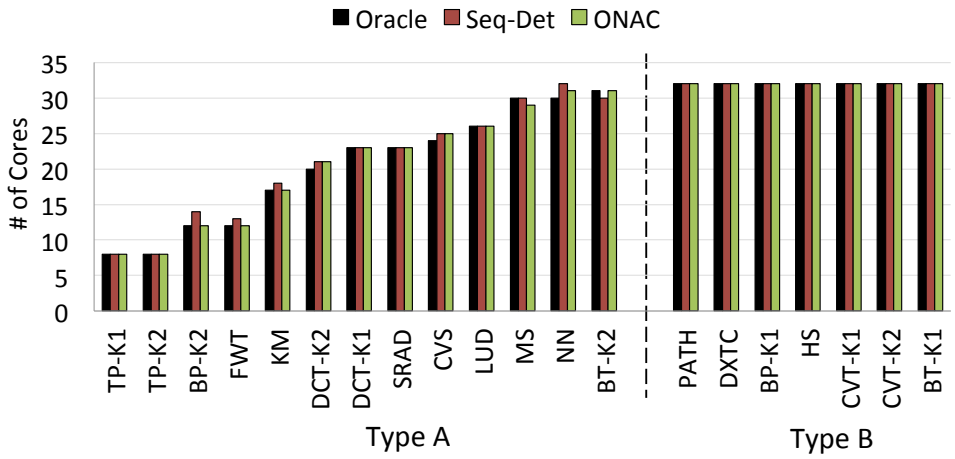
Table 4.3: The GPU configuration used for evaluating ONAC

Number of Cores	32
Warp Size	32
Warp schedulers per core	2, GTO scheduling policy
Execution units per core	32 ALUs, 4 SFUs,16 LD/ST units
Resources/Core	Max. 48 warps, Max. 8 thread blocks, 32768 Registers, 48KB Shared Memory
Core/ICNT/Memory Clock	1300MHz/1300MHz/1848MHz
Number of Mem. Partitions	12
DRAM Chip Model	32bits bus width/Memory Partition, 6 Banks/Memory Partition, GDDR5 timing
Processing Power	Single precision: Max. 2662.4 GFLOPs
Memory Bandwidth	Max. 177.4 GB/s

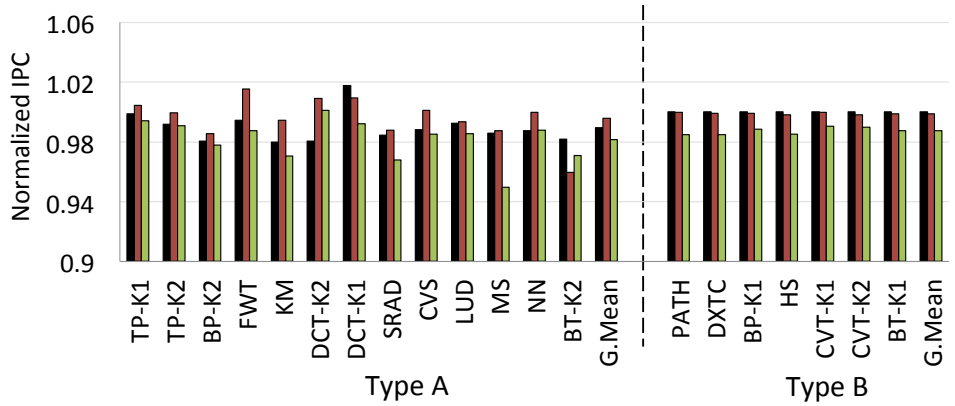
The applications used for our evaluation were chosen from Rodinia [14], an open source benchmark suite for heterogeneous computing and the CUDA SDK [51]. The set of kernels used for our analysis are listed in Table 2 . We broadly group kernels into two categories. Kernels grouped under **type A** have N_{opt} less than 32, and consequently are good candidates for saving energy, while **Type B** kernels have N_{opt} equal to 32.

Table 4.4: GPGPU kernels used for evaluating ONAC

Kernel Name	Abbr.	Suite	Type	CTAs
LU Decomposition	LUD	Rodinia	A	65025
K-Nearest Neighbor	NN	Rodinia	A	50115
K-Means	KM	Rodinia	A	25600
B+tree Kernel 2	BT-K2	Rodinia	A	65535
Back Propagation kernel 2	BP-K2	Rodinia	A	65535
Speckle Reducing Anisotropic Diffusion	SRAD	Rodinia	A	32768
Discrete Cosine Transform kernel 1	DCT-K1	CUDA SDK	A	32768
Discrete Cosine Transform kernel 2	DCT-K2	CUDA SDK	A	32768
Transpose No Bank Conflicts	TP-K1	CUDA SDK	A	65536
Transpose Coarse Grained	TP-K2	CUDA SDK	A	65536
Fast Walsh Transform	FWT	CUDA SDK	A	32768
Convolution Separable	CVSEP	CUDA SDK	A	16384
Merge Sort	MS	CUDA SDK	A	24567
Pathfinder	PATH	Rodinia	B	46297
B+tree Kernel 1	BT-K1	Rodinia	B	65535
Hotspot	HS	Rodinia	B	29241
Back Propagation kernel 1	BP-K1	Rodinia	B	65535
Convolution Texture Kernel 1	CVT-K1	CUDA SDK	B	98304
Convolution Texture Kernel 2	CVT-K2	CUDA SDK	B	98304
DXT Compression	DXTC	CUDA SDK	B	16384



(a) Optimal active core count detected



(b) IPC performance

Figure 4.14: Optimal number of active cores (N_{opt}) detected by ONAC and Seq-Det, and the corresponding IPC achieved by the kernels. IPC results are normalized to the IPC achieved when the kernels are executed with all the cores active.

4.4.3.2 Accuracy of Detection and Performance

We detected the optimal number of cores for each kernel by executing them separately at each core count. We refer to this as N_{oracle}. Fig. 4.14(a) compares the N_{opt} detected by ONAC and Seq-Det to N_{oracle}. As expected, the N_{opt} detected at runtime has an impact on the kernel's performance. Consequently in Fig. 4.14(b), we compare the performance achieved by kernels when executed with ONAC and Seq-Det, to when executed on N_{oracle} cores. The results are normalized to the performance achieved on the baseline configuration (referred to as N_{max}).

4.4.3.3 Type A Kernels

13 of the 20 kernels analyzed in our experiments were type A kernels. Observe in Fig. 4.14(a) that the N_{opt} detected by Seq-Det is always higher than or equal to N_{oracle} for all of them, except BT-K2, where N_{opt} is detected inaccurately due to a sampling variation. The sampled IPC taken with 30 cores active, is higher than the average IPC achieved when the kernel is executed on 30 cores, which leads to Seq-Det underestimating N_{opt}. Consequently, notice in Fig. 4.14(b) that the IPC achieved by Seq-Det is always higher than the performance threshold of 2% for all kernels except BT-K2.

For ONAC, the N_{opt} value detected is higher than or equal to N_{oracle} for all type A kernels except MS. This detection inaccuracy is caused by a similar sampling variation problem. Consequently, the IPC achieved by ONAC is 95% of N_{max} for MS, and is close to the performance threshold for others (refer to Fig. 4.14(b)). On average Seq-Det and ONAC achieve 99% and 98% of the performance achieved with N_{max} cores. On the other hand, we show in Sect. 4.4.4 that they reduce the average energy consumption across kernels by 10% and 20% respectively.

4.4.3.4 Type B Kernels

In addition to detection accuracy, detection time also has an effect on performance. Notice in Fig. 4.14(a) that both techniques, Seq-Det and ONAC, detect N_{opt} with 100% accuracy for all type B kernels. However, observe in Fig. 4.14(b) that the IPC achieved by them varies across kernels. As Seq-Det reduces the number of active cores one at a time, it takes one sample at 31 active cores, and switches back to 32. Consequently, the IPC achieved by Seq-Det for all type B kernels is close to that achieved by the baseline. On the other hand, ONAC has to take a sample with one active core, which causes the small impact on performance seen in the figure.

Although ONAC has this overhead for type B kernels, the effect on performance is less than 2% on average. On the other hand, ONAC saves significantly more energy compared to Seq-Det for type A kernels. We analyze the correlation between detection time and energy in detail in the next section. Notice in Fig. 4.14(b) that the effect of detection time on IPC is also observed in a few type A kernels. Although ONAC detects N_{opt} accurately for the BP-K2,

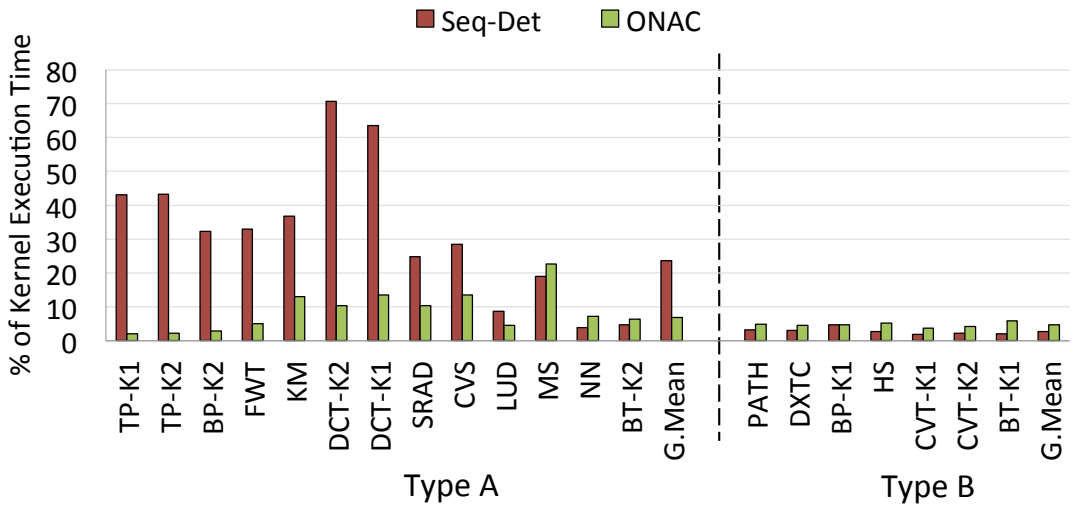


Figure 4.15: The percentage of kernel's total execution time spent on detection by ONAC and Seq-Det.

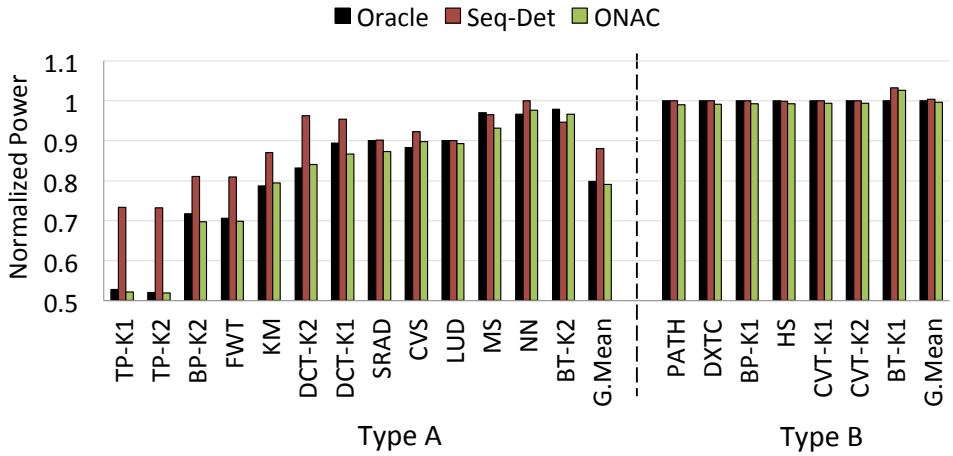
KM, SRAD and BT-K2 kernels, the IPC achieved is a little below the 2% threshold.

4.4.4 Detection Time and its Effect on Power and Energy

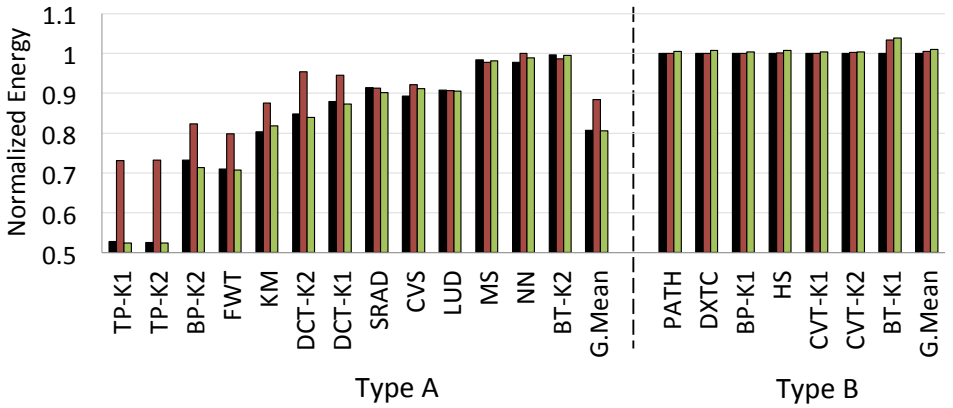
In this section, we analyze the effect of ONAC and Seq-Det's detection time on average power and energy consumption. Fig. 4.15 plots the ratio of detection time to the total execution time, while Fig. 4.16(a) and Fig. 4.16(b) plot the average power and total energy consumed by the kernels. In summary, ONAC and Seq-Det reduce the energy consumption of type A kernels by 20% and 10% respectively, as compared to using all the cores on the chip, with very insignificant overhead on performance. The higher energy saving for ONAC, compared to the sequential detection technique, comes from reducing the detection time by 45% on average across the type A kernels.

4.4.4.1 Type A kernels

The kernels in Fig. 4.15 and Fig. 4.16 are sorted in ascending order of the optimal number of active cores. The optimal core count of type A kernels ranges from 8 (transpose kernels) to 31 (B+Tree K2). Observe in Fig. 4.15 that as N_{opt} increases, the ratio of execution time spent on detection by Seq-Det decreases. The spikes in this trend for the DCT and MS kernels



(a) Average power consumption



(b) Total energy consumption

Figure 4.16: Average power and total energy consumption of the kernels when executed with oracle number of cores, Seq-Det and ONAC. The results are normalized to the power and energy consumption when executed with all the cores active.

are because the total execution time of the kernels is relatively short. Hence, although the detection time of Seq-Det for the DCT kernels is lesser than the KM, FWT, BP-K2 and TP kernels, the ratio is higher. For ONAC, notice that the detection overhead increases as N_{opt} increases. This is because ONAC starts the estimation with 1 core, and increases the number of active cores as it converges to N_{opt}.

For all type A kernels, except MS, NN and BT-K2, Seq-Det takes longer to detect the optimal core count compared to ONAC. Its effect on the average power consumption can be clearly observed in Fig. 4.16(a). For all kernels to the left BT-K2, the average power consumption with Seq-Det is higher compared to ONAC. The difference is larger for lower core

counts. As N_{opt} increases, the detection time of Seq-Det decreases and the difference reduces, until they become comparable for the SRAD, CVS and LUD kernels.

With lower power consumption and comparable performance, the energy consumption with ONAC is lower than Seq-Det when N_{opt} is low. Similar to power, the difference reduces as the optimal core count increases. ONAC consumes a bit more energy compared to Seq-Det for the MS and BT-K2 kernels. Notice that for the MS kernel ONAC consumes lesser power compared to Seq-Det. The loss in IPC is more than the reduction in power consumption, causing the energy consumption to be higher.

4.4.4.2 Type B kernels

As kernels in the type B category have the optimal number of active cores as 32, the detection times for both ONAC and Seq-Det are modest (less than 4% of the execution time). Consequently, the power consumption of both ONAC and Seq-Det is similar to that of N_{max}. As ONAC takes a sample with 1 active core, it has a small impact on the IPC (refer to Fig. 4.14(b)). Consequently, ONAC increases the energy consumption of type A kernels by 2% on average.

4.4.5 Related Work

4.4.5.1 Optimizing thread level parallelism

Kayiran et al. [32] observe that executing the maximal number of CTAs on GPU cores doesn't always achieve the best performance. They propose a mechanism to detect the optimal number of CTAs based on sampling the scheduler state. When the sampled idle time is lower than a threshold and the sampled memory stall time falls within an empirical range, the optimal number of CTAs is captured. The scheme developed by Lee et al. [40] leverages the behavior of greedy warp scheduler. Their scheme measures the number of instructions issued until the first CTA completes, and sets the optimal CTA count as the ratio of number of instructions issued by all CTAs to the number of instructions issued by the first CTA.

In addition to modulating the number of CTAs launched per core, other works focus on

optimizing thread level parallelism at the warp-level. Narasiman et al. [48] and Gebhart et al. [22] propose a warp scheduler that divides the warps executing on a core into two levels, and only issue instructions from the smaller set of warps. Their evaluation demonstrates that energy saving can be achieved by power-gating unused core resources. Rogers et al. [63] design a cache-aware warp scheduling policy that throttles the number of active warps depending on L1 data cache evictions. They demonstrate that workloads that thrash the L1 data cache improve in performance when executed with a lower active thread count.

4.4.5.2 Energy-efficient computing on GPU

Although several works [31, 43] have explored thread-level and core-level optimizations for energy savings in the context of chip multiprocessors (CMP), the techniques cannot be applied directly to GPUs due to two primary differences between CMP and GPU platforms:

- Context switch overhead is much higher on GPUs compared to that of a CMP platform.
- GPU energy efficient techniques cannot rely on complex algorithmic techniques due to lack of OS support.

In [29], Jiao et al. characterize the performance and power consumption of various GPU compute kernels at different GPU core and DRAM frequencies, and show their effect on performance and power consumption. In [39], Lee et al. adjust the number of cores and modulate the core's voltage and frequency to improve throughput under power constraints. While their objective is optimizing performance under given power constraints, we focus on optimizing power consumption under given performance constraints. In [44], Lin et al. utilize software prefetching and dynamic voltage scaling to achieve two objectives: energy optimization under performance constraints and performance optimization under power constraints.

In [27] Hong et al. propose an analytical model to predict power, performance and the optimal number of cores based on kernel's static information. They optimize for performance per watt, which has a side effect of losing performance. Our work is closest to that of Song et al. in [67]. The propose to sample memory latency to each core, and reduce the number of active cores one at a time, until the average latency is lower than an empirically found threshold. Our

implementation of their technique is referred to as sequential detection or Seq-Det, and we thoroughly compare the accuracy, detection time, and impact on power and energy of Seq-Det with ONAC on variety of GPGPU workloads.

4.5 Conclusion

As GPU architectures continue to support higher number of threads with each generation, GPGPU workloads are increasingly becoming throughput limited. In this work, we show that launching maximum number of threads is not always optimal for all GPGPU workloads. We design two techniques that detect the optimal number of threads for a given workload at runtime. First, we present Perf-Sat, a hardware mechanism that detects the optimal thread count on each core at runtime. We identify three categories of workloads, and thoroughly evaluate the effect of number of active threads on their performance. We evaluate the performance of Perf-Sat against the baseline scheduler, and runs with empirically found optimal number of active threads.

Secondly, we show that memory bandwidth limited GPGPU workloads can achieve peak performance without utilizing all cores on the chip. For these workloads, detecting the optimal number of active cores can enable energy savings by power-gating the unused cores. Using detailed analysis of two GPGPU kernels, we demonstrate the effect of number of active cores on performance, power and energy consumption. We then present ONAC, a mechanism to detect the optimal core count at runtime with high accuracy and low detection overhead. We implement ONAC in a cycle level GPU simulator and analyze its efficiency against to a sequential detection technique. Our results show that ONAC reduces the detection time as much as 45% compared to the sequential technique. It reduces energy consumption by 20% on average compared to the baseline (as opposed to 10% with the sequential technique), with less than 2% impact on performance.

CHAPTER 5. WORKLOAD AWARE KERNEL SCHEDULING

5.1 Abstract

The number of active threads required to achieve peak performance for a GPGPU workload depends largely on the ratio of time spent on computation, to the time spent accessing data from memory. While latency limited workloads might require to execute the maximum number of threads supported by the architecture, throughput limited workloads can achieve peak performance with lower than maximum number of threads active. Executing fewer than the maximum threads supported by the architecture frees up hardware resources and enables opportunities for multitasking. In this work, we study the effects of scheduling work from multiple GPGPU workloads on the same GPU core. We show that interleaving workload from different application kernels on the same GPU core can improve the overall utilization, and thus average performance. We analyze two benchmarks that stress different parts of the GPU architecture, and workloads from the Rodinia benchmark suite. Our results show that interleaving workloads that stress complementary parts of the architecture results in extremely promising performance gains. At the same time, workloads that bottleneck on the shared resources do not achieve as high speedups due to interference among threads from concurrently executing kernels.

5.2 Introduction

As discussed in previous chapters, a primary driver for the increasing adoption of GPUs for high throughput computing is the ability to achieve a significantly high floating point throughput and memory bandwidth. For example, the current state of the Nvidia GPU, the Volta V100, supports a peak double precision floating point throughput of 7024 GFLOPs

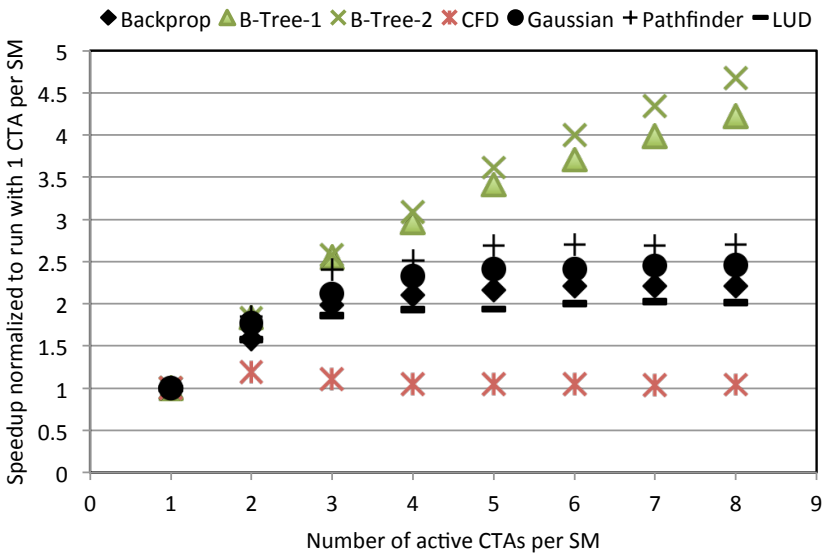


Figure 5.1: The effect of number of CTAs (groups of warps) active per core on performance.

and a peak memory bandwidth of 807 GB/s [55]. To achieve such high throughputs, GPUs use a Single Instruction Multiple Thread (SIMT) programming model. Execution of several SIMD threads (warps) is interleaved to improve the overlap of memory access latency and computation. Consequently, with each new architectural generation, as the compute throughput and memory bandwidths have increased, GPUs have supported an increasing number of warps to increase thread level parallelism. For example, the previous generation NVIDIA Pascal chip, the P100, has a peak FP32 throughput of 4.9 TFLOPs, a peak memory bandwidth of 720 GB/s, and supports concurrent execution of 3584 warps [52]. In comparison, the current generation Volta chip, the V100, supports a peak FP32 throughput of 1.4x, a peak memory bandwidth of 1.2x and supports concurrent execution of 1.4x more warps [55].

While increasing thread level parallelism improves performance of latency-limited workloads, the same is not true if the performance of a workload is throughput-limited. Fig. 5.1 shows the effect of number of active warps on performance of kernels from the Rodinia benchmark suite [14]. It can be observed in Fig. 5.1 that executing the maximum number of warps supported by the chip is not always required to achieve peak performance. Two clear trends can be observed. For some workloads (Backprop, CFD, Gaussian, Pathfinder, and LUD), performance saturates after a certain number of active warps. This happens if either the memory bandwidth, or the

compute throughput is saturated. For the other type of workloads, like the B+Tree kernels, performance continues to improve until the maximum number of threads supported by the architecture has reached. This indicates that there is still some memory latency that is not-overlapped, and the workload might benefit from more threads in flight. This phenomenon has been studied extensively in the academic research community. Authors in [8, 32, 40, 63] have designed runtime techniques that detect the optimal number of warps required to achieve peak performance.

In this work, we focus on the first type of workloads, i.e. workloads for which the performance saturates at lower than the maximum supported thread count. Executing fewer than the maximum number of warps supported by the architecture frees hardware resources, and opens up opportunities for multi-tasking. Concurrent execution of independent kernels has been explored previously in the academic research community. The approaches can be broadly classified into inter-core multi-tasking [5], where threads from concurrent kernels are executed on separate cores, and intra-core multi-tasking [60, 73, 78, 7], where threads from multiple kernels share cores. While inter-core multitasking has shown to be beneficial over sequential execution, these works have shown that intra-core multi-tasking is more beneficial in several cases. This is expected as intra-core multi-tasking has a higher theoretical speedup, as computational units from all cores can be utilized by all kernels executing on the chip. Hence, in this work we solely focus on the intra-core multi-tasking approach.

Sharing GPU core among threads from multiple kernels is a relatively new technique, with its own set of challenges. Previous work have focused mostly on resource partitioning techniques among threads from multiple kernels [60, 73, 78]. In this work, we focus on warp scheduling in the context of intra-core spatial multi-tasking. Authors in [73, 60] introduce two warp scheduling policies based on fairness and issue-rate. However, they touch on them very briefly. In this work, we analyze in more depth, the state of the art warp scheduling policies, namely Two Level Round-robin (TLRR) and Two Level Greedy-then-oldest (TLGTO) with intra-core spatial multi-tasking. We show that interference among threads from different kernels can affect performance of the kernel pair. The specific contributions of this work can be summarized as follows:

1. We implement concurrent kernel scheduling on the same SM in GPGPU-Sim, a cycle level GPU architecture simulator

2. We analyze the runtime characteristics of two GPGPU kernels with concurrent kernel execution, and demonstrate the benefits of intra-core spatial multitasking.

3. We analyze the impact of concurrent kernel execution on performance via two micro-benchmarks, and kernels from the Rodinia benchmark suite.

5.3 Motivation for Concurrent Kernel Execution

To study the benefits of concurrent kernel execution, we analyze the execution of two kernels: `stream_x_words` and `add_x_loops` on GPGPU-Sim [10], a cycle level GPU architecture simulator. Details of the configuration used in our experiments are provided in Table 1. GPGPU-Sim was instrumented to profile the state of warps through a kernel's execution. At a given cycle a warp can be in one of 8 states:

1. Instruction buffer empty: Typically during start of a new warp, as it waits for instruction (inst.) fetch.

2. ALU scoreboard: The warp is at an inst. for which at least one of the operands is waiting on a previous ALU inst.

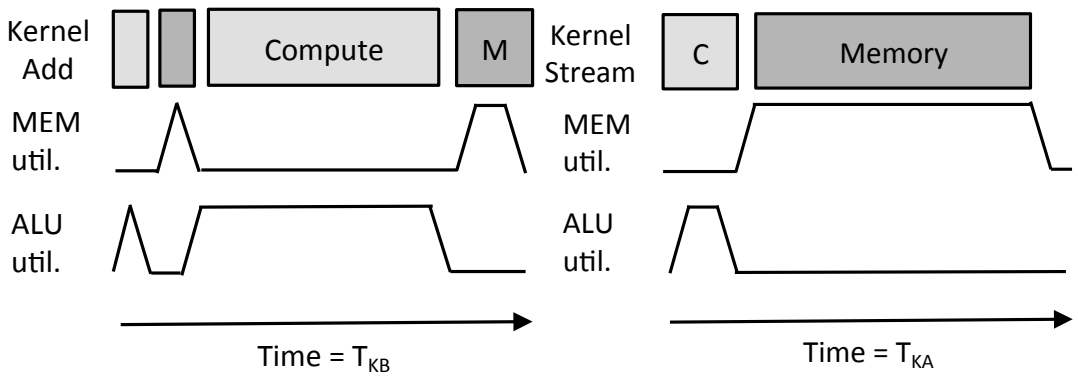
3. MEM scoreboard: The warp is at an inst. for which at least one of the operands is waiting on a MEM inst.

4. ALU stall: The warp is ready to issue an ALU inst., but the execution unit is busy.

5. MEM stall: The warp is ready to issue a MEM inst., but the load / store unit is stalled. This stall can be due a bottleneck anywhere downstream in the memory sub-system.

6. Can issue ALU inst.: The warp can be selected for issue this cycle. It is currently at an ALU inst.

7. Can issue MEM inst.: The warp can be selected for issue this cycle. It is currently at a MEM inst.



(a) Add x loops (b) Stream x words

Figure 5.2: Depiction of the add benchmark used to analyze the benefit of intra-core concurrent kernel execution.

8. Warp completed: The warp has completed execution, and is waiting for other warps in its CTA to complete.

We also profile the number of ALU and memory instructions in flight (ALU and MEM load). In the next subsections, we use these metrics to analyze the workload behavior of two kernels and demonstrate the benefits of concurrent kernel execution.

5.3.1 Compute Benchmark: Add Kernel

`Add_x_loops` is a compute-intensive benchmark. Each thread reads two elements, multiplies each element with a floating point number in a loop, and stores back the sum. It has a short compute phase in the beginning for calculating load addresses, followed by a short memory phase to read the elements (refer to Fig. 5.2). Next, there is a long compute phase where the elements are multiplied in a loop, and a memory phase at the end to store the sum. The `x` parameter controls the number of iterations of the loop, and hence length of the compute phase.

Fig. 5.3 plots a summary of the warp states over time, for a run of the add_20_loops and stream_3_words kernels. The kernels were executed using the left-over policy [60], in which kernels are executed sequentially, and the next kernel does not begin until all CTAs of the current kernel have been launched. Each kernel executes a total of 640 CTAs[1], with each core

[1]A grid of 640 CTAs results in 5 CTA waves: 8 CTAs per core on a 16 core GPU configuration.

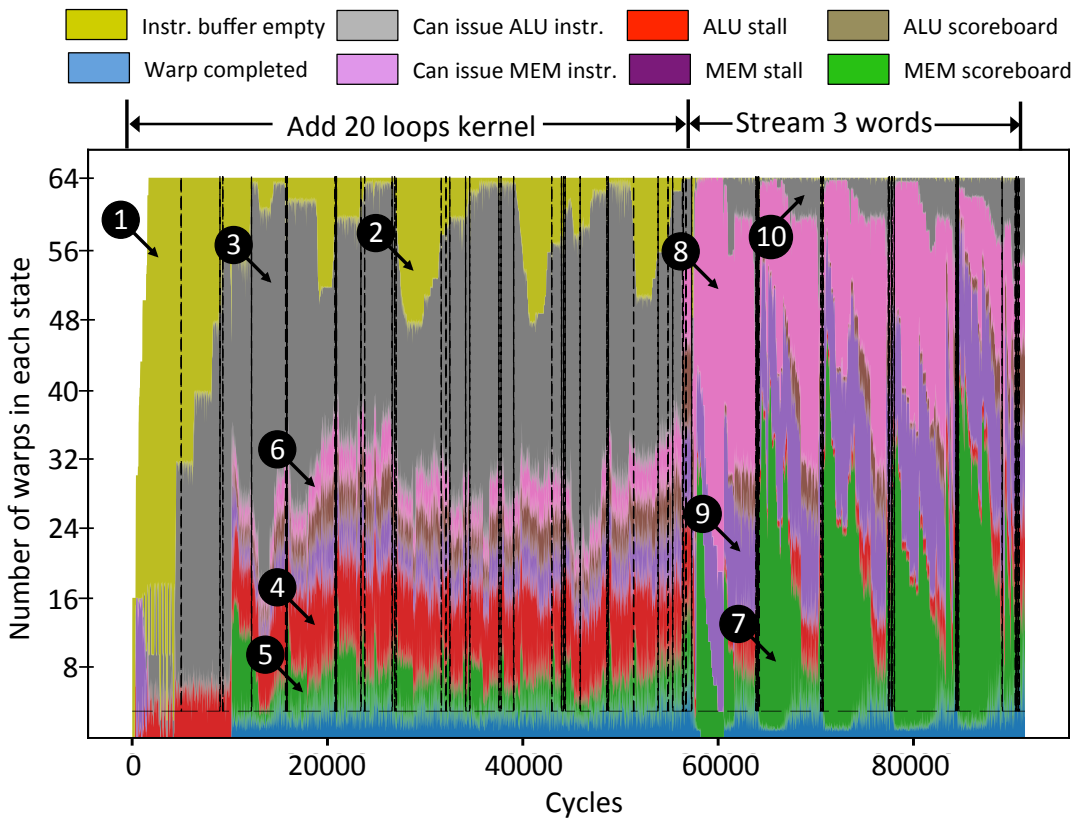


Figure 5.3: A summary of the warp states over time on one core when executing the add_20_loops and stream_3_words kernels sequentially.

concurrently executing a maximum of 64 warps (8 CTAs of 8 warps each). Fig. 5.3 plots the total number of warps in each state on one core. The workload characteristics were identical on the other 15 cores as well. The dashed lines in Fig. 5.3 are cycles when a CTA completes, and new CTA is launched.

We observe that at the beginning of add_20_loops kernel's execution, most warps are waiting for instructions to be fetched ①. This is also observed when new CTAs are launched ②. The figure clearly demonstrates the compute-intensive behavior of the kernel. Most warps are either ready to issue an ALU instruction ③, or hit an ALU stall due to the execution units being busy ④. A small portion of the warps are in the initial memory phase waiting for the elements to be fetched from memory ⑤, ⑥.

Fig. 5.4 plots the measured ALU and memory load over time for the add 20 loops kernel for the execution run shown in Fig. 5.3. The plot clearly demonstrates the workload behavior

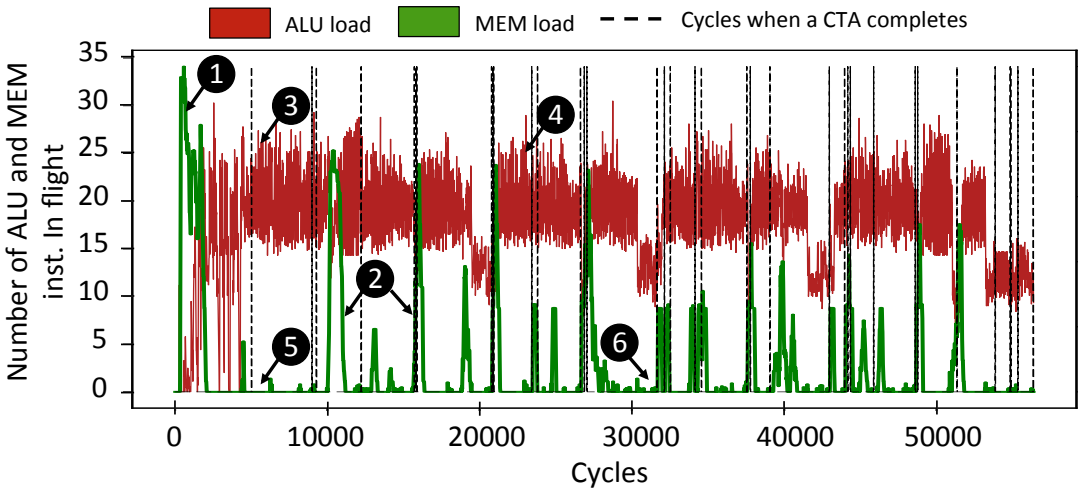


Figure 5.4: ALU and memory utilization over time on one core for the add 20 loops kernel execution in Fig. 5.3.

depicted in Fig. 5.2. We observe in Fig. 5.4 that the add kernel has a high memory utilization only at the beginning of the kernel, and when new CTAs are launched ①, ②, and the workload is highly compute-intensive in the remaining portion of the kernel ③, ④. Moreover, we observe that the memory subsystem has very low utilization during the compute-intensive phases ⑤, ⑥, which opens up opportunity to concurrently schedule threads from a memory-intensive workload.

5.3.2 Memory Benchmark: Stream kernel

`Stream_x_words` is a memory-intensive benchmark, where the x parameter controls the number of elements read by each thread. Fig. 5.2(b) depicts a sketch of the workload of one thread. The kernel has two phases: a compute phase (C) for calculating the thread index, and load and store addresses, followed by a memory phase (M) where each thread loads x elements, and stores them back. The length of the memory phase depends on dynamic memory load latency, and thus increases with x. Our experiments show that loading 3 or more elements per thread saturates the memory bandwidth. Thus, we use stream_3_words as the memory-intensive benchmark for the remainder of this section.

For brevity, we reuse Fig. 5.3 to show the summary of warp states during an execution of the stream_3_words kernel. As mentioned previously, the add and stream kernels were executed

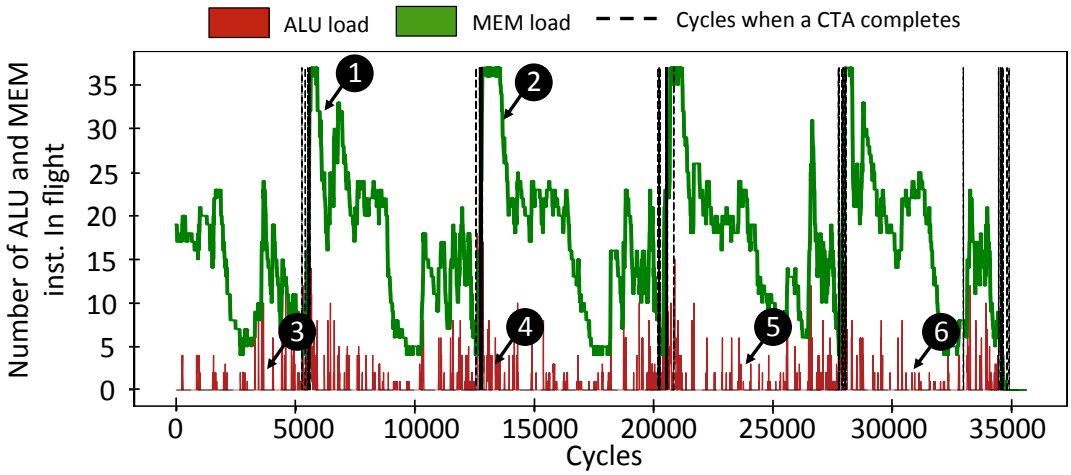


Figure 5.5: ALU and memory utilization over time on one core for the stream 3 words kernel execution in Fig. 5.3.

sequentially using the left-over policy. Similar to the add kernel, the stream kernel executes 5 CTA waves. Fig. 5.3 shows the total warps in each state on one core. We observe that the workload behavior of the stream kernel is complementary to the add kernel, and is very memory-intensive. Most warps are either waiting to issue a memory instruction ⑧ or are stalled as the load / store unit is busy ⑨. Several warps that have issued memory loads are waiting for data from memory ⑦. We also observe a small portion of warps in the short compute phase for calculating thread indexes and memory addresses ⑩.

Fig. 5.5 plots the measured ALU and memory load over the execution of the stream_3_words kernel for the execution run from Fig. 5.3. We observe that, contrary to the add kernel, the workload characteristics of the stream kernel is very memory-intensive. The memory load is high throughout the execution of the kernel ②, while the ALU load is high only in the beginning portion of the kernel ③, and when new CTAs are launched ④. Fig. 5.5 clearly shows that there is sufficient opportunity for utilizing the compute units during the memory-intensive phase of the kernel ⑤, ⑥.

5.4 Intra-core Concurrent Kernel Execution

The thread block scheduler performs two primary tasks: calculating the kernel-to-core mappings, and scheduling thread blocks from active kernels on the mapped cores. In this

Valid	Kernel Index	Grid Dimension	Next Block Dimension	Resources		Num. Active Cores
				Reg	SMEM	
1	9	(128,128,1)	(19,18,1)	4096	128	9
1	10	(256,1,1)	(182,1,1)	2048	256	7
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	0	0	0	0	0	0

Figure 5.6: The kernel status block: Kernel 9 was launched first. 7 cores have started executing kernel 10. Core 16 is still executing kernel 9 (Refer Fig. 5.7).

section, we describe how the baseline scheduler and the intra-core concurrent kernel (interleaved) scheduler perform these tasks.

5.4.1 Kernel to Core Mapping

Kernel to core mapping is calculated each time a new kernel is launched or a kernel completes execution. When a new kernel is launched, its information is added in the **kernel status block** (Fig. 5.6) at a location that has the valid bit reset. If all the entries have their valid bit set, the GPU is executing the maximum number of kernels supported and the kernel launch fails.

The baseline scheduler maps kernels to cores in a round robin order. The index of the mapped kernel is stored in the assigned kernel field of the core in the **core status block** (Fig. 5.7). Launch of new thread blocks is stalled on cores for which the active kernel is different than the assigned kernel. When threads of the active kernel complete, the assigned kernel is made active and launching of thread blocks is resumed.

The interleaved scheduler maps each kernel to all the cores. The assigned kernel field is replaced by a queue with an entry for each active kernel. When the mapping changes, the maximum thread block limit of each kernel in the queue is adjusted. Interleaved scheduling changes the number of thread blocks active per core. Hence, the register and shared memory requirements of each kernel are stored to calculate the occupancy for each kernel.

Full	Core ID	Active Kernel	Assigned Kernel	Maximum Blocks	Num. Active Blocks
1	1	9	9	8	8
1	2	10	10	6	6
⋮	⋮	⋮	⋮	⋮	⋮
0	16	9	10	8	2

Figure 5.7: Core status block: Cores 1 and 2 are executing kernels 9 and 10 respectively. 16 is mapped to kernel 10, but is finishing threads of kernel 9.

5.4.2 Thread Block Scheduling

The baseline scheduler performs two checks before launching a thread block. The active and assigned kernel fields are compared to verify that mapping has not changed. Secondly, the number of active blocks and maximum blocks of the kernel are compared to verify that core has sufficient resources (Fig. 5.7). If the two checks pass, the thread block is launched and the active blocks field is incremented.

The interleaved scheduler updates the kernel to core mapping each time a new kernel is launched or a kernel exits. Depending on the outcome, the maximum blocks field of all kernels within each core is updated. Only kernels having fewer active thread blocks than the maximum are scheduled. This technique ensures that even if kernel-to-core mappings are updated, the CTA scheduler has a finer control on how many CTAs are issued for each kernel. It also handles cases where the mapping changes at runtime from spatial to interleaved or vice versa.

5.4.3 Handling Kernel Exits

In both schedulers, when a new thread block is launched, next block field for that kernel is incremented in the Kernel Status Block. If the next block value exceeds the grid dimensions, the active bit is set to 0 and the active cores field is decremented. When active cores field becomes zero, the valid bit for that kernel is reset. An interrupt is sent to the host interface and the kernel to core mapping is recalculated as well.

Table 5.1: GPU configuration used for evaluating concurrent kernel execution

Chip configuration	
Number of cores	16
Core frequency	1300 MHz
DRAM clock frequency	1850 MHz
Peak SP / DP floating point throughput	1330 / 650 GFLOPs
Peak DRAM bandwidth	177 GB/sec
Core configuration	
Maximum thread blocks per core	8
Maximum warps supported per core	48
Execution units per core	32 ALUs, 4 SFUs 16 LD/ST units
Scheduler configuration	
Warp schedulers per core	2
Instruction dispatch throughput per scheduler	1 instruction every 2 cycles
Ready warps queue size	6

5.5 Experimental Results

In this section, we demonstrate the benefit of concurrent kernel scheduling via concurrent execution runs of the two benchmarks discussed in the previous section, `add_x_loops` and `stream_x_words`. We then present initial results of concurrent execution of kernels from applications from the Rodinia benchmark suite [14] presented in Fig. 5.1. All experiments for this work were conducted on GPGPU-Sim, a cycle level GPU architecture simulator [10]. We configured GPGPU-Sim to match the architecture of Nvidia Tesla M2090 GPU [53] (refer to Table 5.1).

5.5.1 Case Study: Concurrent Execution of Add and Stream

In this section, we continue our investigation of the add and stream kernels discussed in the Sect. 5.3, and analyze the workload behavior when executed with intra-core concurrent kernel execution. In our design of the kernel scheduler, resources within each core are first partitioned to execute both the kernels. Once, all the CTAs of the shorter kernel have been issued, the kernel scheduler starts to change the core partitioning such that more resources are assigned to the longer executing kernel. Consequently, the maximum speedup we can achieve from concurrent kernel execution depends on the relative duration of the two kernels.

Table 5.2: Experimental results of executing the add and stream benchmarks with different workload sizes

Kernel 1	Kernel 2	Kernel 1 runtime	Kernel 2 runtime	Sequential execution (cycles)	Interleaved execution (cycles)	Max speedup	Achieved speedup	Interleaved efficiency
Add 10 loops	Stream 1 word	133529	28104	161633	141230	1.21	1.14	0.94
Add 10 loops	Stream 2 word	133201	55210	188411	154321	1.41	1.22	0.86
Add 10 loops	Stream 3 word	134321	95109	229430	154291	1.71	1.49	0.87
Add 10 loops	Stream 4 word	133412	126053	259465	165207	1.94	1.57	0.80
Add 20 loops	Stream 1 word	265107	26132	291239	273053	1.10	1.07	0.97
Add 20 loops	Stream 2 word	265148	53718	318866	285118	1.20	1.12	0.92
Add 20 loops	Stream 3 word	265103	95162	360265	287276	1.36	1.25	0.92
Add 20 loops	Stream 4 word	265853	122034	387887	300281	1.46	1.29	0.88

The stream kernel simply reads a data from one location in memory into a register, and stores it to another location in memory. It is designed to specifically stress the streaming memory bandwidth of the chip. The efficiency achieved by the workload depends on the total I/O size, and the number of bytes read per thread. Our experiments show that the kernel achieves good memory utilization if the total I/O size is larger than 100KB, and when each thread streams 3 or more words. The add kernel reads two FP32 values from memory, and performs computation on them in a loop. The time spent in the loop is when it has low memory bandwidth utilization. Consequently, as we increase the loop count, there is a greater opportunity to benefit from concurrently executing a memory limited workload.

In Table 5.2, we list the results of executing different workloads with the add and stream kernel. As we increase the number of elements read per thread for the stream kernel, its stress on the memory bandwidth increases. We scale the total I/O size proportionally. We show results with 1 to 4 elements read per thread. For the add kernel, we show results for workload with 10 and 20 loops.

Observe in Table 5.2, that as we increase the number of loops for the add kernel, the runtime scales proportionally. This indicates that the performance of the add kernel is limited on the execution of the compute intensive loop. Similarly, runtime of the stream kernel scales with the number of elements read per thread. Notice that the runtime does not scale linearly with the number of elements. This is because increasing the number of elements per thread improves the memory controller efficiency.

The sequential execution column lists the runtime from runs on the baseline scheduler with the left-over policy, where CTAs of the second kernel are not issued until all CTAs of the first

kernel have been issued. Consequently, runtime with sequential execution is simply the sum of the execution times of the two kernels. The table also lists the maximum possible speedup with concurrent kernel execution, which is ratio of runtime of the longer kernel to that of the runtime with sequential execution. In this ideal scenario, the shorter kernel's execution is completely overlapped by that of the longer kernel, without any performance overhead on the longer executing kernel. The table lists interleaved execution efficiency as ratio of the observed speedup with concurrent kernel execution to the maximum possible speedup.

There are three key observations that can be made about expected speedups from concurrent kernel execution from the data presented in Table 5.2. First, the maximum possible speedup, or the opportunity with concurrent kernel execution depends on the relative runtime of the two kernels. As the runtime of the stream kernel increases, there is a larger opportunity to overlap its use of the memory bandwidth with the compute intensive loop of the add kernel. Second, as the average load on the memory system increases, the effect of concurrent kernel execution on the performance of the add kernel increases. We can observe that as the number of elements read and written per thread is increased, the interleaved execution efficiency decreases. Lastly, the effect on performance degradation of the longer kernel, depends on the ratio of its total execution time spent in interleaved execution mode. We can observe that interleaved execution efficiency of the add 20 loops kernel is higher than that of the add 10 loops kernel when executed with respective stream kernels. This is because the add 10 loops kernel spends larger portion of its execution time sharing the memory bandwidth with the stream kernel compared to the add 20 loops kernel, thereby reducing the interference of the stream kernel.

5.5.2 Analysis of Runtime Workload Characteristics with Concurrent Kernel Execution

In this section, we present the effect on performance with concurrent kernel execution in greater detail by analyzing the runtime workload characteristics of an interleaved execution run of the add_20_loops and stream_3_words kernel. Fig. 5.8 describes a summary of the warp states over time, by plotting the state of each warp on one SM. The workload characteristics were identical on the other 15 cores as well. Dashed lines in the figure are cycles when a CTA completes, and new CTA is launched. Each kernel executes 4 CTAs of 8 warps each. Fig .5.9(b)

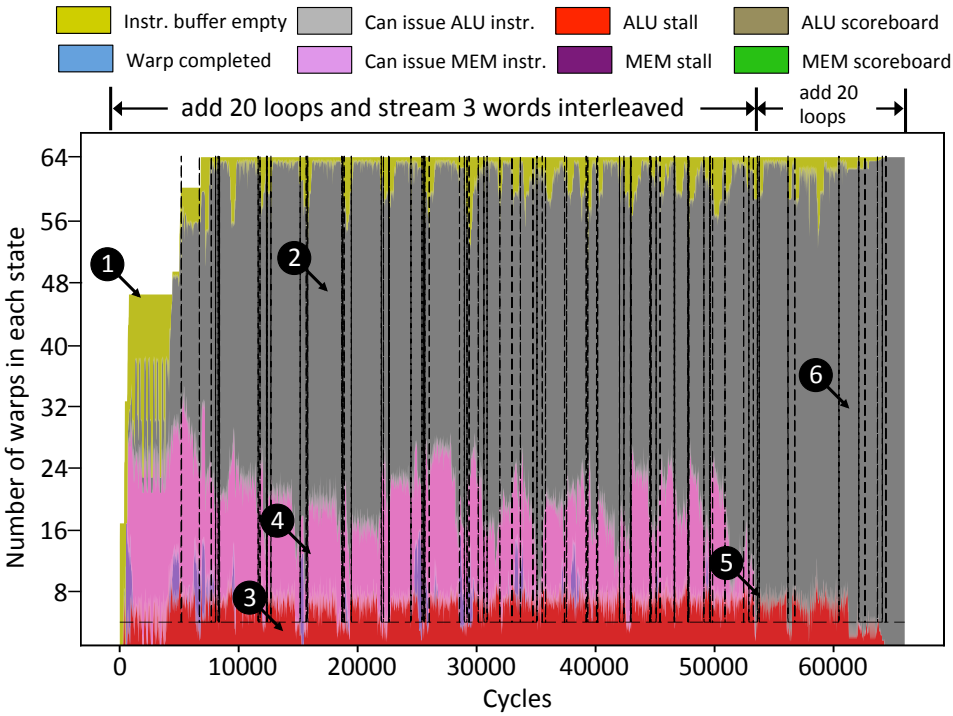
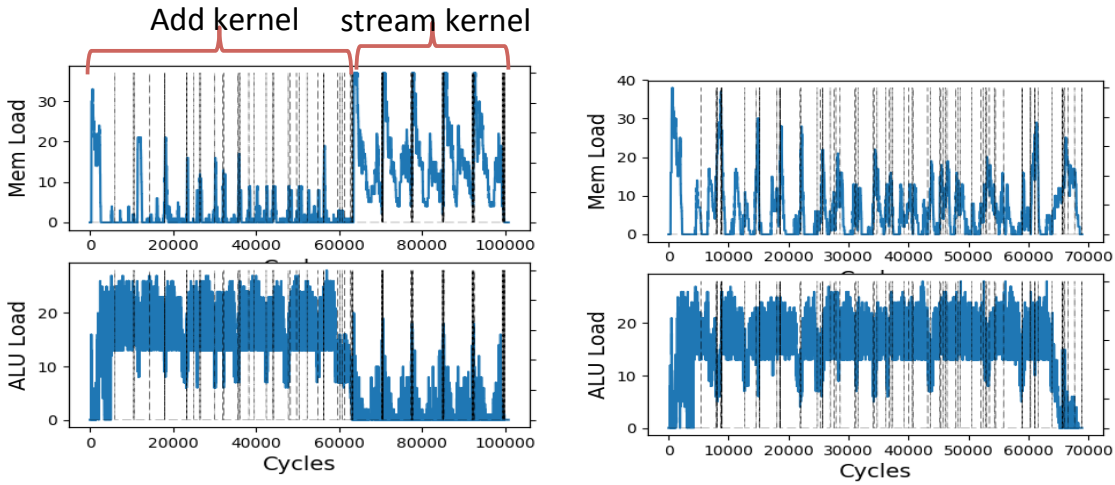


Figure 5.8: A summary of the warp states over time on one core when executing the add_20_loops and stream_3_words kernels with intra-core concurrent kernel execution.

plots the measured ALU and memory load over time during the interleaved execution run.

We observe that similar to sequential execution (refer to Fig. 5.3) most warps are waiting for instructions to be fetched at the beginning of the run ①. Some of this effect is also observed when new CTAs are launched. Three key observations can be made by comparing the plots in the Fig. 5.8 and Fig. 5.9.

1: Average ALU and MEM load is higher in concurrent execution: In the baseline execution run of the `add_x_loops` kernel (and the `stream_x_words` kernel), most warps were either ready to issue an ALU (memory) instruction, or hit a ALU (memory) stall (refer to Sect. 5.3). The number of warps ready to issue memory (ALU) instructions is relatively low. In comparison, during the interleaved execution (Fig. 5.8), there are enough warps ready to issue both ALU ② and memory instructions ④. We observe that this leads to increased ALU and MEM load throughout the interleaved execution run (Fig. 5.9(b)). In comparison, in the sequential execution run (Fig. 5.9(a)), ALU load is high and MEM load is low during execution of the add kernel, and vice versa during execution of the stream kernel.



(a) Baseline - Add and stream executed sequentially.

(b) Interleaved - Add and stream executed concurrently on all cores.

Figure 5.9: ALU and memory utilization of the stream and add kernel profiled on GPGPU-Sim [10].

2: Stalls reduce in concurrent execution: When we compare the warp states in Fig. 5.3 and Fig. 5.8, we notice that the number of stalls decrease in the concurrent execution run. As discussed in chapter 4, the number of stalls increase after a kernel achieves optimal thread count. It is interesting to observe that number of warp hitting memory stalls reduce significantly in the interleaved run, which indicates that 4 CTAs, or 32 warps is the optimal warp count for the stream_3_kernel on this architecture configuration. We observe that the interleaved execution run does encounter ALU stalls ③ (although lower than the baseline case). We would like to note that this does not indicate the optimal warp count of the add kernel is lower than 4 CTAs.

3: Benefit on concurrent kernel execution ceases after the short kernel exits: Although this is an expected result, we can observe this effect in the plots in the Fig. 5.8 and Fig. 5.9(b). Once the stream kernel completes its execution (Fig. 5.8 ⑤), the number of warps ready to issue memory instructions reduce significantly, and most warps are now waiting to issue ALU instructions ⑥. The MEM loads reduces around cycle 55K. The drop in ALU load towards the end of the execution is due to the warps of the add kernel completing, which can be observed in baseline run as well. The burst of MEM load towards the end of kernel execution is due to the stores from the stream kernel that are in pipeline being flushed to memory.

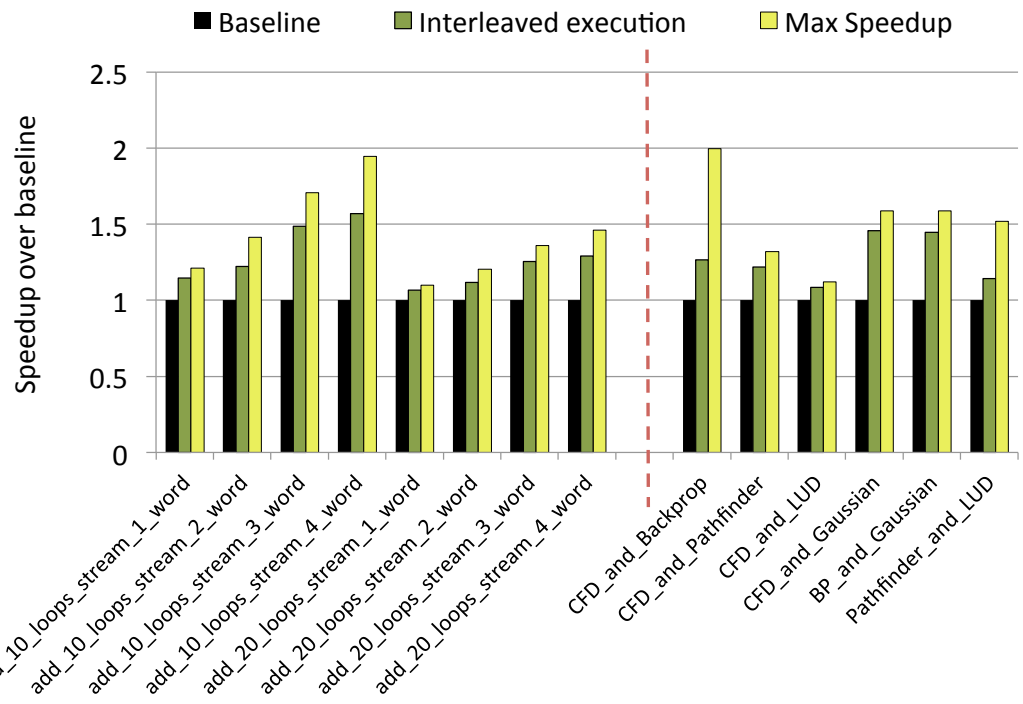


Figure 5.10: Performance comparison of kernel pairs executed with concurrent kernel execution, against execution with the baseline left over policy and the maximum possible theoretical speedup.

5.5.3 Performance Impact with Concurrent Kernel Execution

In this section we present an analysis of the impact of concurrent kernel execution on performance. We further analyze the impact on performance of the add and stream workload pairs presented in 5.2. We also present results of concurrent kernel execution of kernel pairs from the Rodinia benchmark suite presented in Fig. 5.1. As shown in Fig. 5.1, we executed each kernel by varying the number of active CTAs from 1 to the hardware limit, and found their optimal active CTA count. We then execute kernel pairs with optimal number of CTA of each kernel. We omit concurrent kernel execution of the B+Tree kernels as their performance does not saturate even with the maximum number of active threads.

Fig. 5.10 plots the performance of kernel pairs executed with concurrent kernel execution normalized to the baseline left over policy. It also plots the theoretical maximum performance gain possible, which is the runtime of the longer of the two kernels normalized to the runtime with the baseline policy. As discussed in Sect. 5.5.1, the speedup with concurrent kernel

execution increases with increase in the overlap of the kernel pair. We can observe that, as we increase the number of elements streamed per thread, the speedup of concurrent execution with the add kernels increase. The maximum possible speedup increases as well. Notice that concurrent kernel execution achieves closer to the maximum possible speedup for the kernel pairs with the add_20_loops kernel compared to the add_10_loops kernel. This indicates that interference from the stream kernel with the add kernel reduces in the add_20_loops case.

For kernels from the Rodinia benchmark suite, we executed the CFD and Gaussian kernels with Backprop, LUD and Pathfinder. We chose the CFD and Gaussian kernels due to their low IPC (refer to Fig. 5.11). The pairs of the CFD kernel with Pathfinder and LUD achieve performance close to the maximum possible. However, we observe that when CFD is executed with the Backprop kernel, the achieved speedup is not as high. We suspect this is due to the interference across kernels mentioned previously. The kernel pairs with Gaussian are presented as they show a unique behavior. We discuss this in more detail in the next subsections. It is also interesting that we observe speedup with executing the Pathfinder and LUD pair, as they both are compute intensive. We suspect the kernels have high compute and memory intensive phases that overlap nicely with concurrent kernel execution.

5.5.4 Impact on ALU and Memory Utilization

Fig. 5.11 and Fig. 5.12 plot the ratio of the peak floating point throughput and memory bandwidth achieved with concurrent kernel execution, and when each kernel is executed by itself on the GPU with the optimal number by of active threads. As expected, we observe that the IPC achieved by the add_20_loops kernel is higher than, and memory bandwidth achieved is lower than, the add_10_loops kernel. Both kernels achieve close to peak floating point throughput, and low memory bandwidth utilization. Contrary to add, the stream kernels achieve high memory bandwidth, and low floating point throughput utilization. The bandwidth utilization achieved increases as we increase the number of elements streamed per thread, and maxes out around 70%.

The average IPC and memory bandwidth achieved significantly increases during concurrent kernel execution. Notice that the average memory bandwidth utilization increases, and the

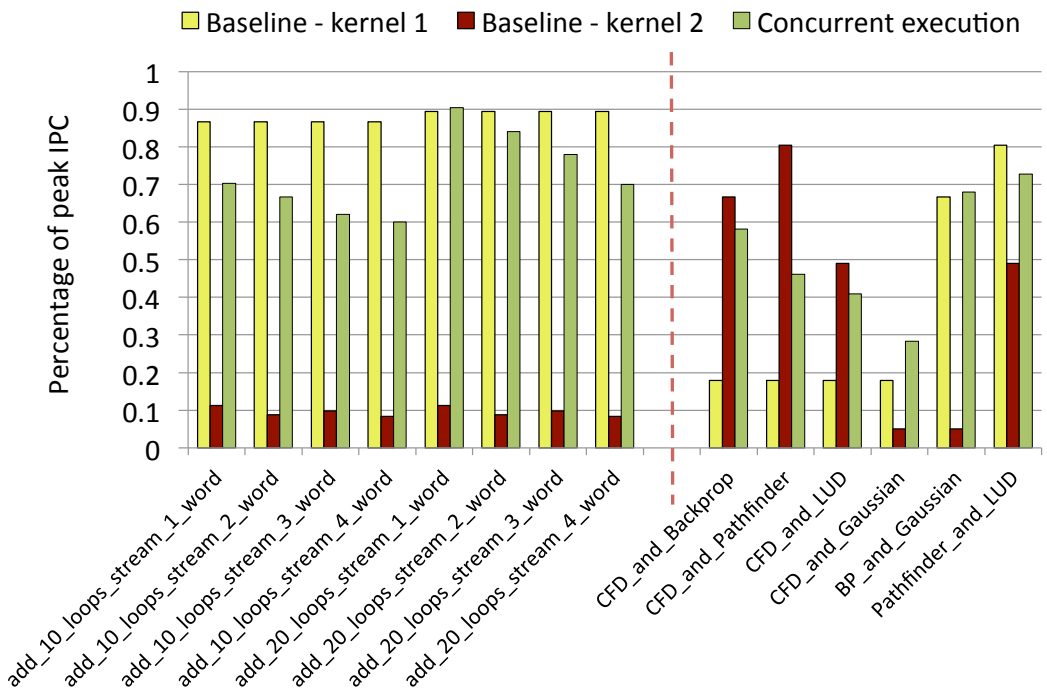


Figure 5.11: Comparison of compute throughput utilization achieved with concurrent kernel execution to when the kernels occupy the entire GPU.

average IPC decreases, as we increase the number of elements streamed per thread. As the number of elements streamed per thread increases, the runtime of the stream kernel increases, which increases the time spent in overlap. During the overlap, the IPC of the pair is lower than that of the add kernel executing alone, which cause the average IPC to decrease. Notice that the decrease in IPC is lower when executed with the add_20_loops kernels, as the ratio of time spent in overlap mode to that of the runtime of the add kernel reduces. We would like to note that, even with this effect on IPC, the speedup with longer running stream kernel increases (refer to Fig. 5.10). The drop in average IPC is another way of looking at the difference between the maximum possible and achieved speedups. This again points to the interference issue mentioned in the previous subsection.

For kernels from the Rodinia benchmark, we observe that none of the kernels were able to achieve good memory bandwidth utilization (refer to Fig. 5.12). Consequently, we ran experiments with the CFD and Gaussian kernels as they have low IPC - kernel pairs where both kernels have high IPC do not result in performance speedup from concurrent kernel

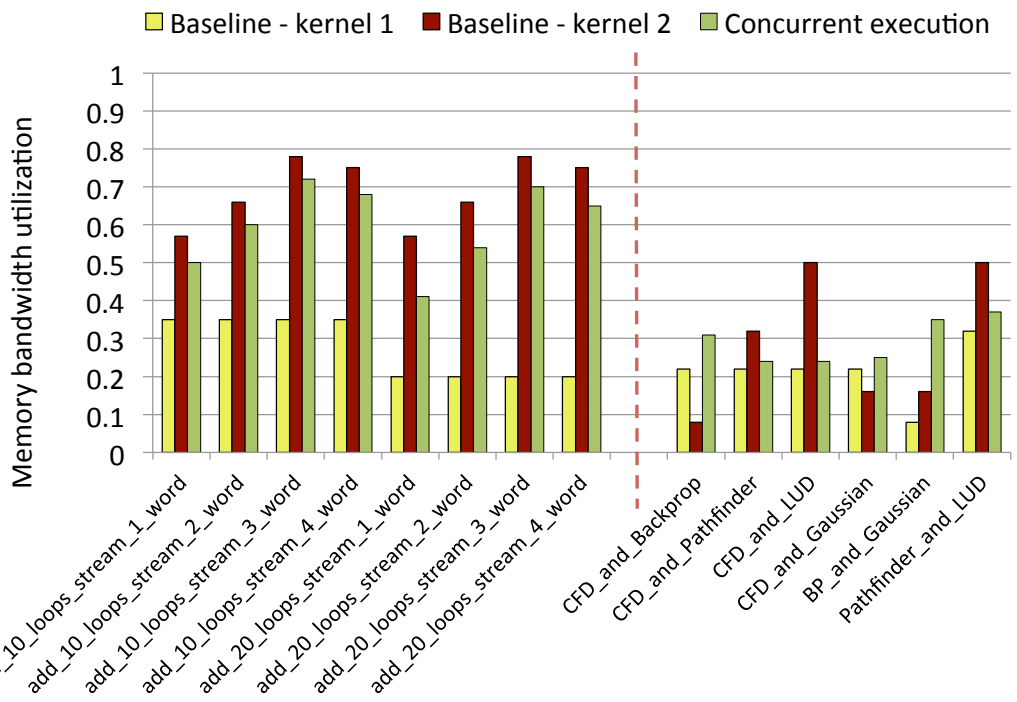


Figure 5.12: Comparison of memory bandwidth utilization achieved with concurrent kernel execution to when the kernels occupy the entire GPU.

execution due to the floating point throughput limit.

As discussed in the previous subsection, concurrent execution of the CFD kernel with Pathfinder and LUD kernels shows speedup over the baseline. This is interesting as memory utilization of both the Pathfinder and LUD kernels is close to that of CFD. Another interesting behavior observed is that the Gaussian kernel, when executed with both CFD and Backprop, increases the average IPC and bandwidth utilization. This is counter-intuitive, as it indicates that executing with the Gaussian kernel is increasing the utilization of the other kernel in the pair. A third interesting behavior observed is that performance of concurrent execution of Backprop and LUD achieves a speedup. Both kernels achieve quite high floating point performance, as well as memory bandwidth when executed by themselves. This indicates that with concurrent kernel execution, their compute and memory intensive phases execute in complementary schedule.

5.6 Related Work

There has been an interest in issuing concurrent work from multiple kernels on the GPU hardware for several years. As initial GPGPU hardware did not support execution of concurrent kernels, most initial works focused on enabling and optimizing launching of GPU compute work from multiple processes. The benefits of concurrent kernel execution were first demonstrated by Guevara et al. [24]. They made a keen observation that some workloads do not have enough data parallelism to completely occupy all the cores on a GPU. They proposed a software technique, called `cusub`, that maintains an issue queue, and manages launching of kernels to the CUDA driver. `cusub` intercepts memory copy and kernel launch APIs from applications compiled with it, and merges the kernels that don't have enough thread blocks to occupy 1 CTA wave of work.

To enable task parallelism for irregular applications on GPUs Chen et al. [16] introduce the persistent kernel approach. In this technique, a kernel with size equal to the maximum number of thread blocks supported by the GPU is launched, and serves as an application that manages launching of user kernels. The persistent kernel executes for the entire lifetime of the application and communicates with the host application via a queue stored in the GPU memory. They use a producer-consumer model, wherein the host (*producer*) enqueues new work in the queue. Thread blocks of the persistent kernel (*consumer*) poll the queue and deques new work when found. The goal of these works was to implement support for concurrent kernel execution on GPUs. Starting from CUDA version 4.0 and higher these functionalities are provided via CUDA streams and CUDA MPS.

Another body of work focuses on efficient sharing of the GPU among multiple host processes. Wang et al. [71] introduce the context funneling approach, where different host threads funnel their work into an application, which then sends it to the GPU queue. They compare their manual funneling approach with the one provided by CUDA and then do a comparison of context funneling with context switching. Wende et al. [75] show that dependencies between kernels can break concurrency. They develop a software layer which intercepts kernel launch functions from multiple host application threads and reorders the kernels to maximize concurrency.

The motivation of work done by Gregg et al. in [23] is the closest to that of ours. Their goal was also to improve average system throughput by concurrently executing compute and memory bound kernels. They use the persistent kernel approach used in [16] and develop a software runtime layer to launch thread blocks from concurrent applications into thread blocks of the persistent kernel. However, they do not discuss how the GPU hardware schedules the thread blocks on the cores.

More recent works have focused on resource partitioning of the GPU across concurrently executing kernels. The approaches can be broadly classified into inter-core multi-tasking [5] and intra-core multi-tasking [60, 73, 78, 7]. In inter-core multi-tasking, thread blocks from different kernels are launched on different SMs. In other words, the GPU resources are divided among concurrent kernels at core granularity. On the other hand, in intra-core multi-tasking, threads from multiple kernels are executed on the same core, thereby sharing the resources on GPU cores. In [5], the authors introduce the term spatial multi-tasking, which is now commonly used to refer to the technique of launching CTAs from concurrent kernels on separate GPU cores. Their focus is on resource partitioning the SMs across multiple kernels that would not otherwise occupy the entire GPU. They implement fairness and round robin based schemes, and show performance improvement of concurrent kernel execution over sequential execution.

Our work is closest to the recent works published in [73, 60, 78]. In [73], authors identify that GPGPU kernels are often limited on different SM resources. We make a similar observation in [7]. They implement a fairness based algorithm in the CTA scheduler that tries to maximize resource usage of each kernel using a heuristic that keeps track of the limiting resource of each kernel. The kernel scheduler used in our implementation achieves a similar goal by statically partitioning the resources among concurrent kernel execution pairs. Authors in [73] also recognize that baseline warp scheduling policies are not sufficient for concurrent kernel execution scenarios. However, they analyze it briefly, and propose a simple scheme that tries to allocate scheduling slots proportional to the number of thread blocks executed by the kernel.

Similar to the work of authors in [73], Park et al. in [60] focus on resource partitioning among thread blocks from concurrently executing kernels. The authors introduce GPUMaestro, which tries to find the optimal resource partitioning among thread block from concurrently

executing kernels by using both inter-core, and intra-core resource partitioning. They divide the cores into dedicated cores, follower cores and reserve two cores as trial cores for collecting performance metrics. The dedicated cores execute CTAs from only one kernel, the follower cores execute equal number of CTAs from both kernels, while the trial cores execute one more and one less CTA compared to the follower cores. After every epoch of 50K instructions, the resource partitioning with better performance of the two trial cores is chosen for the follower cores, and new resource partitioning is tried on the trial cores. They acknowledge that using the baseline GTO and RR warp scheduling policies can cause an issue with fairness among threads of concurrently executing kernels. Consequently, they implement a quota based scheme similar to [73]. However, their results show that for the workloads they tested, the round robin scheme outperformed the quota based scheme.

Authors in [78] also design an algorithm to achieve resource partitioning across CTAs from multiple kernels. They make similar observations as made by us in [7] and [8]. Specifically, they point out that occupancy of different GPGPU kernels is limited by different resources [7], and thus by concurrently executing CTAs from different kernels, there is an opportunity to increase total number of active CTAs. They also make the observation that performance of all GPU workloads does not scale with the number of active threads [8], and there is an optimal thread count for each kernel. They leverage the two observations, and find a resource partitioning which minimizes the average performance degradation across the kernel pairs. To find the optimal performing resource partitioning statically, they execute each kernel with different CTA counts offline, and use analytical equations to scale the performance metrics to a concurrent kernel execution run.

CHAPTER 6. CONCLUSION

In the last decade, there has been a wide scale adoption of Graphics Processing Units (GPUs) as accelerators for high throughput general purpose computing. This trend is expected to continue as throughput demands of applications continue to increase, both in the high performance scientific computing domain, and commercial data centers due to the rise of machine learning applications. We envision that as the number of applications being ported to GPUs continue to rise, workload aware scheduling techniques will be required to achieve a high throughput utilization across a wide array of applications.

We present novel techniques at each of the three levels of scheduling in GPU hardware. First, at the warp scheduling level, we demonstrate that the workload agnostic Greedy Then Oldest (GTO) and Round Robin (RR) policies do not work uniformly for different workloads. We analyze the runtime workload behavior of GPGPU kernels, and show that kernels typically have phase behavior that can cause performance bottlenecks when executed with the baseline policies. We then present a warp scheduling policy that uses information inserted into the kernel instructions by the compiler to make scheduling decisions, and reduces the performance impact of imbalanced workload phase behavior.

At the thread block scheduling level, we make two observations regarding the effect of number of thread blocks on performance. First, we recognize that as GPU architectures are becoming larger and supporting more threads, the amount of floating point throughput and memory bandwidth available per thread is reducing. We show that performance of all kernels does not scale with increase in the number of active threads, and there is an optimal thread count for each kernel. We demonstrate the effect of number of active threads on the states of the warp issue stage, and use this observation to design a simple scheme that detects the optimal active thread count at runtime. Secondly, we recognize that if performance of a kernel is limited

by memory bandwidth, it does not need all cores on the chip to achieve peak performance. We design a hardware technique that detects the optimal number of active cores required at runtime. Our technique uses linear estimation model to project the optimal active core count, and thus converges much faster than a sequential detection technique. Once the number of optimal cores are detected, we power gate the unused cores to reduce static power consumption. Our technique reduces energy consumption by an average of 20% with less than 2% impact of performance.

At the highest level of scheduling, we implement a concurrent kernel scheduler than interleaves work from multiple kernels on the same GPU core. Continuing on our work from the thread block scheduler level, for pairs of kernels for which the optimal thread count is lower than the maximum number of threads supported by the architecture, we demonstrate that interleaving execution of threads from multiple kernels results in significant speedups. The amount of speedup achieved with concurrent execution of a kernel pair depends on the workload characteristics of the two kernels. Consequently, we thoroughly analyze the effect of runtime workload characteristics on the efficiency of concurrent kernel execution via analysis of the warp states over time, and the stress they put on different parts of the architecture. We demonstrate that interference from threads of concurrently executing kernels has a performance overhead, and leads to lower than the ideal speedup that could be achieved from concurrent kernel execution.

BIBLIOGRAPHY

[1] Cuda developer zone.

[2] Technical brief: Nvidia geforce 8800 gpu architecture overview.

[3] *The NVIDIA Tesla K80 GPU Architecture for Servers*, 2014 (accessed November 17, 2014).

[4] The top 500 supercomputers list, 2017.

[5] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The case for gpgpu spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[6] Joshua A Anderson, Chris D Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[7] M. Awatramani, J. Zambreno, and D. Rover. Increasing gpu throughput using kernel interleaved thread block scheduling. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 503–506, Oct 2013.

[8] M. Awatramani, J. Zambreno, and D. Rover. Perf-sat: Runtime detection of performance saturation for gpgpu applications. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 1–8, Sept 2014.

[9] M. Awatramani, X. Zhu, J. Zambreno, and D. Rover. Phase aware warp scheduling: Mitigating effects of phase behavior in gpgpu applications. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 1–12, Oct 2015.

[10] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proc. of the IEEE Int. Symp. on Performance Analysis of Systems and Software*, pages 163–174, 2009.

[11] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[12] Ian Buck. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*, page 6. ACM, 2007.

[13] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Sokadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE Int. Symp. on Workload Characterization*, October 2009.

[15] Jianmin Chen, Xi Tao, Zhen Yang, Jih-Kwon Peir, Xiaoyuan Li, and Shih-Lien Lu. Guided Region-Based GPU Scheduling: Utilizing Multi-thread Parallelism to Hide Memory Latency. In *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), 2013*, pages 441–451. IEEE, 2013.

[16] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-gpu systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[17] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[18] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[19] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 47–47. IEEE, 2004.

[20] Wilson WL Fung and Tor M Aamodt. Thread block compaction for efficient simt control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 25–36. IEEE, 2011.

[21] Wilson WL Fung, Ivan Sham, George Yuan, and Tor 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.

[22] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. A hierarchical thread scheduler and register file for energy-efficient throughput processors. *ACM Transactions on Computer Systems (TOCS)*, 30(2):8, 2012.

[23] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, 2012.

[24] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, 2009.

[25] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. Many-core vs. Many-Thread Machines: Stay Away From the Valley. *IEEE Computer Architecture Letters*, pages 25–28, 2009.

[26] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[27] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *Proc. of the 37th Annu. Int. Symp. on Computer Architecture*, volume 38, pages 280–289, 2010.

[28] Intel Corporation. The intel xeon e7 v4 product family.

[29] Yang Jiao, Heshan Lin, Pavan Balaji, and Wuchun Feng. Power and performance characterization of computational kernels on the GPU. In *Proc. of 2010 IEEE/ACM Int. Conf. on Green Comput. and Commun. & Int. Conf. on Cyber, Physical and Social Comput.*, pages 221–228, 2010.

[30] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 332–343, New York, NY, USA, 2013. ACM.

[31] Changhee Jung, Daeseob Lim, Jaejin Lee, and SangYong Han. Adaptive execution techniques for smt multiprocessor architectures. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 236–246, 2005.

[32] O. Kayran, 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, Sept 2013.

[33] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011.

[34] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[35] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2017.

[36] Argonne National Laboratory. The coral aurora supercomputer.

[37] Nagesh B. Lakshminarayana and Hyesoon Kim. Effect of Instruction Fetch and Memory Scheduling on GPU Performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010.

[38] Lawrence Livermore National Laboratory. The coral sierra supercomputer.

[39] Jungseob Lee, Vijay Sathisha, Michael Schulte, Katherine Compton, and Nam Sung Kim. Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling. In *Proc. of the 20th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–120, 2011.

[40] 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, Feb 2014.

[41] Shin-Ying Lee and Carole-Jean Wu. Caws: Criticality-aware warp scheduling for gpgpu workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 175–186, New York, NY, USA, 2014. ACM.

[42] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic gpu cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 67–77. ACM, 2015.

[43] Jian Li and Jose F Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proceedings of the 12th IEEE International Symposium on High-Performance Computer Architecture*, pages 77–87, 2006.

[44] Yisong Lin, Tao Tang, and Guibin Wang. Power optimization for GPU programs based on software prefetching. In *Proc. of the 10th Int. Conf. on Trust, Security and Privacy in Comput. and Commun.*, pages 1339–1346, 2011.

[45] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated

branch and memory divergence tolerance. *ACM SIGARCH Computer Architecture News*, 38(3):235–246, 2010.

[46] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, New York, NY, USA, 2010. ACM.

[47] A. Munsi. The OpenCL C 2.0 Specification. 2013.

[48] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale 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.

[49] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[50] John Nickolls and William J Dally. The GPU Computing Era. *IEEE micro*, 30(2), 2010.

[51] Nvidia Corporation. *Nvidia CUDA SDK*.

[52] NVIDIA Corporation. NVIDIA Tesla P100: GP100 pascal whitepaper.

[53] NVIDIA Corporation. NVIDIA Fermi Whitepaper, 2010.

[54] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2016. Version 8.0.

[55] NVIDIA Corporation. NVIDIA Tesla V100: GV100 Volta Whitepaper, 2017.

[56] Oak Ridge National Laboratory. The coral summit supercomputer.

[57] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[58] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A Survey of General-purpose Computation on Graphics

Hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[59] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 527–540, New York, NY, USA, 2017. ACM.

[60] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 527–540. ACM, 2017.

[61] Guillem Pratx and Lei Xing. Gpu computing in medical physics: A review. *Medical physics*, 38(5):2685–2697, 2011.

[62] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468–4477, 2009.

[63] Timothy G Rogers, Mike O'Connor, and Tor 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.

[64] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[65] Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC bioinformatics*, 8(1):474, 2007.

[66] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for gpu computing. In *Graphics hardware*, volume 2007, pages 97–106, 2007.

[67] Seokwoo Song, Minseok Lee, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Energy-efficient scheduling for memory-intensive gpgpu workloads. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[68] Michael Steffen and Joseph Zambreno. Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 237–248. IEEE Computer Society, 2010.

[69] John E Stone, James C Phillips, Peter L Freddolino, David J Hardy, Leonardo G Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of computational chemistry*, 28(16):2618–2640, 2007.

[70] Sam S Stone, Justin P Haldar, Stephanie C Tsao, BP Sutton, Z-P Liang, et al. Accelerating advanced mri reconstructions on gpus. *Journal of Parallel and Distributed Computing*, 68(10):1307–1318, 2008.

[71] Lingyuan Wang, Miaoqing Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *2011 International Conference on High Performance Computing and Simulation (HPCS)*.

[72] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369, March 2016.

[73] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 358–369. IEEE, 2016.

[74] W Hwu Wen-Mei. *GPU computing gems emerald edition*. Elsevier, 2011.

[75] F. Wende, F. Cordes, and T. Steinke. On improving the performance of multi-threaded cuda applications with concurrent kernel execution by kernel reordering. In *2012 Symposium on Application Accelerators in High Performance Computing (SAAHPC)*.

[76] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[77] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 230–242, June 2016.

[78] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 230–242. IEEE Press, 2016.

[79] X. Zhu, M. Awatramani, D. Rover, and J. Zambreno. Onac: Optimal number of active cores detector for energy efficient gpu computing. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 512–519, Oct 2016.