Hardware and Software Optimizations for GPU Resource Management

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

by
Vishwesh Jatala

to the
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR, INDIA

December, 2018

CERTIFICATE

It is certified that the work contained in the thesis entitled "***Hardware and Software Optimizations for GPU Resource Management***", by ***Vishwesh Jatala***, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 9/2/2018

February, 2018

Dr. Amey Karkare
Associate Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur,
Kanpur-208016, INDIA.

Abstract

Graphics Processing Units (GPUs) are widely adopted across various domains due to their massive thread level parallelism (TLP). The TLP that is present in the GPUs is limited by the number of resident threads, which in turn depends on the available resources in the GPUs – such as registers and scratchpad memory. Recent GPUs aim to improve the TLP, and consequently the throughput, by increasing the number of resources. Further, the improvements in semiconductor fabrication enable smaller feature sizes. However, for smaller feature sizes, the leakage power is a significant part of the total power consumption. In this thesis, we provide hardware and software solutions that aim towards the two problems of GPU design: improving throughput and reducing leakage energy.

In the first work of the thesis, we focus on improving the performance of GPUs by effective resource management. In GPUs, resources (registers and scratchpad memory) are allocated at thread block level granularity, as a result, some of the resources may not be used up completely and hence will be wasted. We propose an approach that shares the resources of SM to utilize the wasted resources by launching more thread blocks in each SM. We show the effectiveness of our approach for two resources: registers and scratchpad memory (shared memory). On evaluating our approach experimentally with 19 kernels from several benchmark suites, we observed

that kernels that underutilize register resource show an average improvement of 11% with register sharing. Similarly, the kernels that underutilize scratchpad resource show an average improvement of 12.5% with scratchpad sharing.

For scratchpad sharing, we observe that the performance is limited by the availability of the part of scratchpad memory that is shared among thread blocks. To address this, we propose compiler optimizations to improve the availability of shared scratchpad memory. We describe an allocation scheme that helps in allocating scratchpad variables such that shared scratchpad is accessed for short duration. We introduce a new hardware instruction, *relssp*, that when executed releases the shared scratchpad memory. Finally, we describe an analysis for optimal placement of *relssp* instructions, such that shared scratchpad memory is released as early as possible, but only after its last use, along every execution path. We evaluated the proposed compiler optimizations on 19 kernels from various benchmarks suites. We observe that the kernels that under-utilize scratchpad memory show an average improvement of 19% when compared to the baseline approach, without affecting the performance of the kernels that are not limited by scratchpad memory.

In the second work of the thesis, we focus on reducing the leakage power of the register file. Recent trends in the GPU increase the number of on-chip registers to increase the TLP. However, with the increase the register file size, the leakage power increases. Also, with the technology advances, the leakage power component has increased and has become an important consideration for the manufacturing process. The leakage power of a register file can be reduced by turning infrequently used registers into low power (drowsy or off) state after accessing them. A major challenge in doing so is the lack of runtime register access information. To address this, we propose a system called **GREENER**. It employs a compiler analysis that

determines the power status of the registers, i.e., which registers can be switched off or placed in drowsy state at each program point and encodes this information in program instructions. Further, it uses a runtime optimization that increases the accuracy of power status of registers. We implemented the proposed ideas using GPGPU-Sim simulator and evaluated them 21 kernels from several benchmarks suites. We observe that **GReEneR** shows an average reduction of register leakage energy by 46.96% when compared to baseline approach with a negligible number of simulation cycles overhead (0.53%).

Acknowledgements

This work would not have been made possible without the help of many people.

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Amey Karkare for his continuous support and guidance. He is a great mentor and teacher; he not only supported me academically but also personally throughput my Ph.D. program. He always understands the student's capabilities and focuses his vision towards the student's career, and I am very fortunate to have supervised by him. He is always open to any kind of discussion – starting from formulating the ideas, implementing the concepts, analyzing results, and reviewing manuscripts. His constructive feedback at every stage of my thesis helped me in improving the quality of research. Thanks to him for always allocating me time, even during his busy schedule, whenever I required his guidance. I am also grateful to him for all his administrative support and recommendations. Under his guidance, I not only developed research skills but also learned several transferable skills that make me handle difficult problems in life.

I also would like to thank Late. Prof. Sanjeev K. Aggarwal, who jointly supervised me during the initial stages of my research work. He introduced me to the world of Graphics Processing Unit (GPU) architectures and compilers for GPUs. I am greatly inspired by his administrative and abstract thinking skills.

I am grateful to Dr. Jayvant Anatpur at Mentor Graphics India Pvt. Ltd. I have been collaborating with him during the entire period of my research. Many thanks to my advisor for introducing him to me during the beginning stages of the research work. His vast experience in the areas of GPU architecture has helped to shape my research work in a good direction. He is a wonderful mentor and always approachable. I had a very fruitful discussion with him on every part of the thesis.

I would like to thank the department of computer science at IIT Kanpur for providing all the resources and facilities to conduct research. Also, I want to thank all the faculty members in my department for teaching me various courses. Special thanks to Research-I Foundation for providing additional scholarship and travel grants during the first few years of my research. I am grateful to my institute for giving me a platform to pursue my research career and providing a beautiful environment. Several inspiring sessions and cultural events conducted by the institute kept refreshing my life during my stay at IIT Kanpur.

I have been financially supported by Tata Consultancy Services (TCS) through TCS Research Scholarship program since 2014. I am deeply grateful to TCS for providing me a regular monthly fellowship, contingency grant, and travel assistance to attend the conferences held at various national and international venues.

Many thanks to all my friends who made my life enjoyable at IIT Kanpur. Especially, I would like to thank all the M.Tech 2011 friends for giving me a memorable experience during the first two years of my Ph.D. Also, I would like to thank all my lab mates, Ph.D. colleagues, and seniors for their discussions on a wide range of technical and non-technical topics that helped me in improving my social skills.

I am grateful to all my family members for their unconditional love and support throughout my life. Special thanks to my wife, Nagajyothi, for her co-operation throughout my research. My biggest strengths are their moral support and encouragement, which made me cross all the hard stages of my life. Without their support, it would have been very difficult to achieve this.

Finally, I thank god for blessing me and giving me inner strength to make my path up to this stage.

Vishwesh Jatala

Dedicated To

My Family Members and Teachers

Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic Logic Unit
CACTI	Cache Access and Cycle Time Model
CFG	Control Flow Graph
CMOS	Complementary Metal-Oxide-Semiconductor
CTA	Co-operative Thread Array
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random-Access Memory
DVFS	Dynamic Voltage Frequency Scaling
FCFS	First-Come-First-Served
FR-FCFS	First-Ready, First-Come-First-Served
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GREENER	GPU REgister file ENErgy Reducer
GTO	Greedy Then Old
IPC	Instructions Per Cycle
ITRS	International Technology Roadmap for Semiconductors
LRR	Loose Round Robin
McPAT	Multicore Power, Area, and Timing
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OWF	Owner Warp First
PTX	Parallel Thread Execution
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessors

SP	Stream Processor
SRAM	Static Random-Access Memory
TLP	Thread Level Parallelism

Chapter 1

Introduction

For many decades, the processor vendors kept increasing the performance of the single-core processor by increasing the core clock frequency and exploiting instruction-level-parallelism in every generation [13]. Moreover, with the advent of Moore's law [73], the density of transistors on a chip increased, as a result, computer architects leveraged the additional transistors to improve the performance further. However, by the mid-to late-1990s, these techniques became less effective due to power constraints and heat dissipation [45]. Consequently, the chip designers shifted their designs towards having multiple processors on a single chip rather than having a single-core processor. This has started the development of chip multiprocessors and accelerators such as Graphics Processing Units (GPUs).

The initial development of the GPUs was driven by the graphics applications to render the graphics quickly on a screen [76]. The graphics applications typically require performing the same operation on a large number of pixels at the same time. To meet this requirement, the architecture of the GPU is designed differently compared to that of a traditional CPU. GPUs expose more parallelism and achieve high throughput by adopting single instruction and multiple threaded (SIMT) execution model, in which several threads execute the same instruction at the same time in parallel. During the early stages, GPUs were less programmable. Later, they have become more featured and programming interfaces, such as OpenGL [77], were developed to program graphics applications on GPUs.

In the early days, GPUs were targeted towards graphics applications only. With several architectural innovations, they are now being used for general purpose applications. Programming languages such as CUDA [1] and OpenCL [5] were developed to write programs for general purpose computation on GPUs. Several data parallel applications are leveraging GPUs by processing a huge amount of data to improve performance [33, 41, 66].

One of the key factor to achieve high throughput in the GPU is the ability to hide the long latencies by using thread-level-parallelism (TLP). In the past decade, several studies [46, 90, 92, 96] focus on improving the throughput of GPUs by exploiting the TLP. They propose static and run-time techniques to improve the performance of GPUs by managing the GPU resources effectively.

1.1 Problem Description

The amount of TLP utilized by a GPU depends on the number of resident threads, and hence the number of resident thread blocks, in each of its streaming multiprocessor (SM). However, the number of thread blocks that can be launched on an SM depends on the resource usage of the thread blocks–e.g. the number of registers, the amount of shared memory. Since the allocation of threads to an SM is at the thread block granularity, some of the resources may not be used up completely and hence remain underutilized. Theoretically, resources can get underutilized from [0, 50%) when the applications are limited by the resource. On evaluating experimentally for several benchmark applications, we observed that resource allocation at thread block level granularity shows up to ∼25% resource underutilization (details are discussed in Chapter 2). Moreover, when the applications are limited by the resources, the number of resident thread blocks get limited, as a result, the TLP in the SM also gets limited.

To keep improving the TLP, and consequently the throughput, GPU architects increase the maximum number of resident threads and the on-chip resources such as register file in every generation. For instance, NVIDIA Fermi GF100, which was released in 2010, has 128KB register file and allows up to 1536 resident threads. While the next generation NVIDIA Kepler GK110, released in 2012, has 256KB register file and allows maximum 2048 resident threads [4].

However, with the increase in the number of on-chip resources and with the increase in technology, the leakage power dissipation has become a serious concern for GPU manufacturing process. Kim et al. [50] discussed the growing importance of leakage power dissipation with the decrease in the feature size of the semiconductor devices. Also, Lim et al. [65] has observed that the leakage power dissipated by the GPUs is more than 50% of the total power for several workloads. Moreover, earlier studies [59, 67] show that register files in GPUs consume around 15% of the total power. Also, we observed that the registers that are allocated to a thread are accessed for a very short duration (on an average $< 2\%$ of the total simulation cycles of its warp) during the entire life time of its warp (details are described in Chapter 4), but they continue to dissipate leakage power throughout the entire execution of its warp.

To summarize, improving GPU throughput and reducing leakage energy have become the two crucial factors of GPU design. In this thesis, we address these problems by managing the GPU resources (registers and scratchpad memory) effectively. To improve GPU performance, we propose hardware and software solutions that increase the TLP by utilizing the wasted on-chip registers and scratchpad memory present in the SMs. To minimize the leakage energy of register file, we propose hardware and software solutions by taking into account register access patterns.

1.2 Thesis Contributions

In the first part of the thesis, we focus on improving the GPU performance. We propose an approach called *Resource Sharing* that exploits underutilized registers and scratchpad memory of the GPU to improve the throughput. To improve the performance further, we propose compiler optimizations that improve the availability of scratchpad memory that is present with resource sharing approach. In the second part, we propose a system called **GReEneR** that employs static and run-time techniques to reduce the leakage energy register file by turning the registers into low power states.

1.2.1 Resource Sharing

To improve the TLP and the resource utilization, we propose *Resource Sharing* that launches additional thread blocks in each SM. These thread blocks use the unutilized resources and also share the resources with other resident blocks. The additional thread blocks help in improving the throughput by hiding long execution latency cycles. We evaluated the effectiveness of the resource sharing for two resources, i.e., registers and scratchpad memory. We further propose three optimizations, viz., *Owner Warp First (OWF)*, *Unrolling and Reordering of Register Declaration* and *Dynamic Warp Execution* that help in managing the additional thread blocks effectively.

We implemented resource sharing in GPGPU-Sim simulator [3] and experimentally validated it on 19 applications from 4 different benchmark suites: GPGPU-Sim [14], Rodinia [19], CUDA-SDK [2], and Parboil [7]. We observed that applications that underutilize register resource show a maximum improvement of 24% and an average improvement of 11% with register sharing. Similarly, the applications that underutilize scratchpad resource show a maximum improvement of 30% and an average improvement of 12.5% with scratchpad sharing. The remaining applications, whose number of resident thread blocks are not limited by any resources, perform similar to the baseline approach.

1.2.2 Compiler Optimizations for Scratchpad Sharing

In GPUs, resources allocated to a thread block are released only after all the threads of a thread block finish their execution even though the resources are not accessed till the end of program execution. This mechanism affects of availability of shared scratchpad memory that is associated with scratchpad sharing since scratchpad memory may not accessed until the program execution. Thus, it can reduce the amount of TLP with the scratchpad sharing approach.

To improve the availability of shared scratchpad memory, we propose compiler optimizations. We describe an allocation scheme that helps in allocating scratchpad variables such that shared scratchpad is accessed for short duration. We introduce a new hardware instruction, *relssp*, that when executed, releases the shared scratchpad memory. Finally, we describe an analysis for optimal placement of *relssp* instructions such that shared scratchpad memory is released as early as possible, but only after its last use, along every execution path.

We integrated the *relssp* instruction in the GPGPU-Sim simulator, and implemented the compiler optimizations in Ocelot [21] framework. We evaluated the effectiveness of our approach on 19 kernels from 3 benchmarks suites: CUDA-SDK, GPGPU-Sim, and Rodinia. We observe that the kernels that under-utilize scratchpad memory show an average improvement of 19% and maximum improvement of 92.17% with scratchpad sharing including compiler optimizations when compared to the baseline approach.

1.2.3 Reducing Leakage Energy of Register File

The leakage energy of a register file can be minimized by turning the register into low power states [8]. However, having the precise knowledge of register access pattern will make it more efficient.

In this thesis, we propose a system, **GREENER**, to reduce the leakage energy of GPU register file. It uses a compile-time analysis to determine the power state of the registers (OFF, SLEEP, or ON) after each instruction by estimating the register usage information. **GREENER** transforms an input assembly language by encoding the power state information at each instruction to make it energy efficient. The static analysis makes safe approximations while computing power state of the registers, therefore, the choice of the state can be suboptimal at run-time. Hence, to improve the accuracy and energy efficiency, **GREENER** provides a run-time optimization that dynamically corrects the power state of registers of each instruction.

We implemented **GREENER** using GPGPU-Sim simulator. We integrated GPUWattch [59] with CACTI-P [63] version to enable power saving mechanism. We evaluated our implementation on wide range of kernels from different benchmark suites: CUDA-SDK, GPGPU-Sim, Parboil, and Rodinia. We observe a reduction in the register leakage energy by an average of 46.96% and maximum of 57.57% with a negligible number of simulation cycles overhead.

1.3 Overview of GPU Architecture

Our proposed ideas involve implementing hardware and software strategies to improve GPU performance and energy efficiency. This section briefs the required background on GPU architecture, programming models, and power models.

1.3.1 GPU Architecture and Programming Model

A typical NVIDIA GPU [1] consists of a set of Streaming Multiprocessors (SMs). Each multiprocessor contains a large number of execution units such as Arithmetic Logic Units (ALUs), Special Function Units (SFUs), and Load/Store units. GPUs achieve high throughput because they can hide long memory execution latencies with massive thread level parallelism. Each SM in a GPU maintains on-chip resources such as register file and scratchpad memory. The register file allows the resident threads to maintain their contexts, and hence can have faster context switching. To reduce the access latency, the register file is divided into multiple banks. The registers from different banks can be accessed in parallel. A bank conflict occurs whenever multiple registers need to be accessed from the same bank, and these registers need to be accessed in serial.

A programmer can parallelize an application on GPU by using programming languages such as CUDA [1] and OpenCL [5]. The region of a program which is to be parallelized is specified using a function called kernel. A program written in CUDA can be compiled using *nvcc* compiler. The compiler translates the program into PTX, which is a pseudo-assembly instruction set supported by NVIDIA. PTX does not include the optimizations, such as register allocation and strength reduction, also it is not executed on the real hardware. However, NVIDIA executes SASS instruction set, which is assembled by *ptxas* from PTX code. NVIDIA provides a tool, *cuobjdump*, to disassemble the executable into SASS assembly language. GPGPU-Sim simulator [3] does not simulate SASS directly but simulates PTXPlus, which is a one to one mapping of SASS and has similar syntax as that of PTX.

A kernel written in CUDA is invoked with the configuration specifying the number of thread blocks and number of threads in a thread block as <<<`#ThreadBlocks, #Threads`>>>. The number of thread blocks that can reside on an SM depends on: (a) the number of registers used by a thread block and the number of registers available in the SM, (b) the amount of scratchpad memory used by a thread block and the amount of scratchpad memory available in the SM, (c) the maximum number of threads allowed per SM, and (d) the maximum number of thread blocks allowed per SM.

The thread blocks that are launched in the GPUs are executed independently [1].

This allows the thread blocks to be scheduled in any order on to the SMs. The threads that are launched the SM are further divided into a set of consecutive 32 threads called Warp. Each SM contains one or more warp schedulers which schedule a ready warp every cycle from a pool of ready warps. All threads in a warp execute the same instruction. Warp schedulers schedule instructions in-order and so, when the current instruction of a warp can not be issued, the warp is not considered to be ready. If no warp can be scheduled in a cycle, then that is a stall cycle. As the number of stall cycles increases, the run time goes up and the throughput decreases. The resource sharing approach proposed in this thesis (discussed in Chapter 2) aims to improve the GPU performance by increasing the number of resident thread blocks, which help in hiding long latency cycles.

1.3.2 Power Model

GPUWattch [59] framework uses the simulation statistics of GPGPU-Sim to measure the power of each component in the GPUs. The framework is built on McPAT [62], which internally uses CACTI [18]. McPAT models the register files as memory arrays to measure the register power. CACTI divides memory arrays into set of banks, which are finally divided into subarrays (collection of memory cells).

Our proposed system **GReEneR** (discussed in Chapter 4) optimizes the PTX-Plus code to make it energy efficient by reducing the leakage power of the register files. In the experiments, we use GPUWattch to measure the leakage power.

1.4 Thesis Organization

The rest of the thesis is organized as follows. The details of resource sharing approach is discussed Chapter 2. The compiler optimizations to improve the scratch-pad sharing is described in Chapter 3. Chapter 4 presents the system **GReEneR** to reduce the leakage of register file. Chapter 5 discusses the related work. Chapter 6 concludes the thesis and discusses future research directions to improve the performance and energy efficiency of GPUs.

Chapter 2

Improving GPU Performance Through Resource Sharing

2.1 Introduction

General-Purpose Graphics Processing Unit (GPGPU) applications exploit on-chip resources like registers and scratchpad memory available in GPUs to improve their performance. The throughput achieved by a GPU depends on the amount of thread level parallelism (TLP) utilized by it. However, the TLP that is present in the GPU is limited by the number of resident threads, which in turn depends on the availability of resources in its streaming multiprocessor. A programmer interested in parallelizing an application in GPU invokes a function, called *kernel*, with a configuration consisting of the number of thread blocks and the number of threads in each thread block. The maximum number of thread blocks, and hence the number of threads, that can be launched in an SM depends on the number of available resources (such as registers and scratchpad memory) in it. For instance, if an SM has R resources and each thread block requires R_{tb} resources, then $\lfloor R/R_{tb} \rfloor$ number of thread blocks can be launched in each SM. Thus utilizing $R_{tb} * \lfloor R/R_{tb} \rfloor$ units of resources present in the SM; the remaining $R \bmod R_{tb}$ resources are wasted.

Table 2.1: Set-1: Benchmarks Limited by Registers

Benchmark	Application	Kernel	Block Size	Registers per Thread
Rodinia [19]	backprop	bpnn_adjust_weights_cuda	256	24
Rodinia	b+tree	findRangeK	508	24
Rodinia	hotspot	calculate_temp	256	36
GPGPU-Sim [14]	LIB	Pathcalc_Portfolio_KernelGPU	192	36
GPGPU-Sim	MUM	mummergpuKernel	256	28
Parboil [7]	mri-q	ComputeQ_GPU	256	24
Parboil	sgemm	mysgemmNT	128	48
Parboil	stencil	block2D_hybrid_coarsen_x	512	28

Table 2.2: Set-2: Benchmarks Limited by Scratchpad Memory

Benchmark	Application	Kernel	Block Size	Scratchpad Size (in bytes)
CUDA-SDK [2]	convolutionSeparable (CONV1)	convolution RowsKernel	64	2560
CUDA-SDK	convolutionSeparable (CONV2)	convolution ColumnsKernel	128	5184
Rodinia	lavaMD	kernel_gpu_cuda	128	7200
Rodinia	nw (NW1)	needle_cuda_shared_1	16	2180
Rodinia	nw (NW2)	needle_cuda_shared_2	16	2180
Rodinia	srad_v2 (SRAD1)	srad_cuda_1	256	6144
Rodinia	srad_v2 (SRAD2)	srad_cuda_2	256	5120

2.1.1 Resource Underutilization in GPUs

To quantify the amount of resource underutilization, we analyzed several benchmark applications using the GPGPU-Sim [3] simulator. The GPU configuration used for the experiments is shown in Table 2.3. The benchmark details are given in Table 2.1 and Table 2.2. For applications that are limited by register resource, we show the number of resident thread blocks per SM in Figure 2.1(a), and we show the percentage of registers that are unutilized per SM in Figure 2.1(b).

Example 2.1.1. *Consider the application* hotspot*. Each thread for this benchmark needs 36 registers, and there are 256 threads in each block, so the number of registers required per thread block is 9216 (36 * 256). According to the configuration*

Table 2.3: GPGPU-Sim Architecture

Resource	GPU Configuration
Number of SMs	14
Max Num of TBs per SM	8
Max Num of Threads per SM	1536
Number of Registers (32 bit) per SM	32768
Scratchpad Memory per SM	48KB for register sharing
	16KB for scratchpad sharing
Warp Scheduling	LRR
L1-Cache per SM	16KB
DRAM Scheduler	FR-FCFS
GDDR3 Timings	$t_{RRD} = 6, t_{WR} = 12, t_{RCD} = 12, t_{RAS} = 28,$
	$t_{RP} = 12, t_{RC} = 40, t_{CDLR} = 5, t_{CL} = 12$

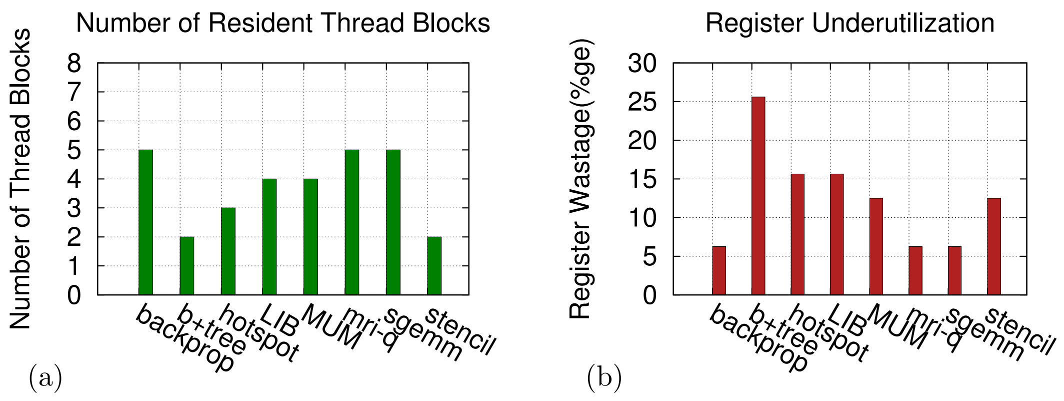
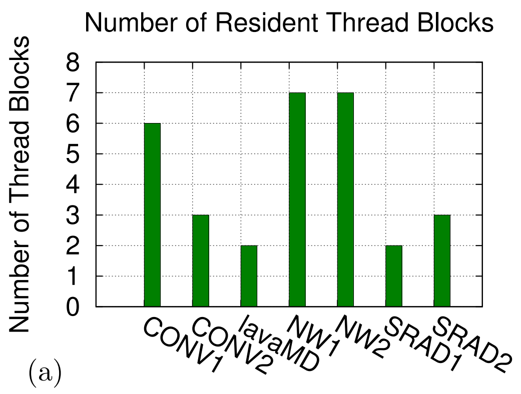


Figure 2.1: (a) Number of Resident Thread Blocks with Limited Registers (b) Underutilization of Registers

(Table 2.3), the number of registers available on an SM is 32768, so an SM can fit only 3 threads blocks ($\lfloor \frac{32768}{9216} \rfloor$)*. This results in wastage of 5120 registers per SM.* □

Similarly, in Figure 2.2(a) we show the number of resident thread blocks per SM for the applications that are limited by scratchpad resource, and in Figure 2.2(b) we show the percentage of scratchpad memory that remains unutilized per SM.

Example 2.1.2. *Consider the application* lavaMD*. Each thread block for this benchmark needs 7200 bytes of scratchpad memory. According to the configuration in Table 2.3, the amount of scratchpad memory available per SM is 16384 bytes, hence*



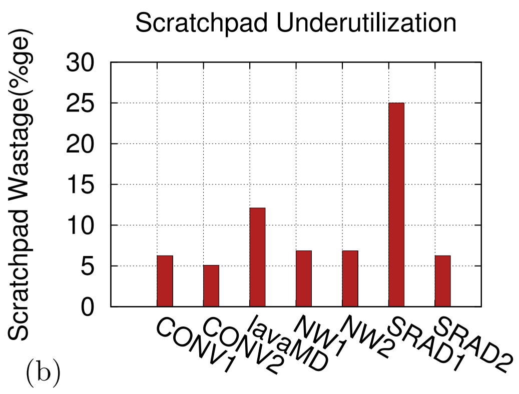


Figure 2.2: (a) Number of Resident Thread Blocks with Limited Scratchpad Memory (b) Underutilization of Scratchpad Memory

an SM can fit 2 thread blocks. This results in 1984 bytes of scratchpad memory per SM remaining unutilized. Similar behavior is observed for other applications as well.

□

To summarize, applications that are constrained by their resource requirements may not only have low residency, but also waste resources of GPU.

2.1.2 Our Solution: Resource Sharing

To improve the resource utilization and thread-level-parallelism, we propose a mechanism, *Resource Sharing*, to share resources of SM and launch more thread blocks, effectively reducing resource wastage. In particular, we show how sharing of registers and sharing of scratchpad improves the throughput of SMs. It is observed [46] that increasing the number of threads benefits compute-bound applications, but may result in increased L1/L2 cache misses for memory-bound applications, thereby decreasing their performance. To overcome this, we propose an optimization, called *Owner Warp First (OWF)* that schedules the extra thread blocks and their constituent warps effectively. For the register sharing approach, we further propose two optimizations, viz., *Unrolling and Reordering of Register Declaration* and *Dynamic Warp Execution* that improves register utilization and minimizes the number of stall cycles observed by the additional thread blocks respectively.

To summarize, this chapter describes the following contributions of the thesis.

1. To utilize the resources of GPUs effectively, we propose a novel resource sharing mechanism that enables launching of more thread blocks per SM.

2. We implemented our approach for two resources, i.e., registers and scratchpad. We propose optimizations to further improve the throughput of applications.

3. We implemented our approach using GPGPU-Sim and evaluated on 19 applications from GPGPU-Sim [14], Rodinia [19], CUDA-SDK [2], and Parboil [7] benchmarks. We observe that 8 of the applications, which underutilize the register resource, show an average improvement of 11% with register sharing approach. Similarly 7 applications, which underutilize the scratchpad resource, show an average improvement of 12.5% with scratchpad sharing. While the remaining 4 applications perform comparable to the baseline approach.

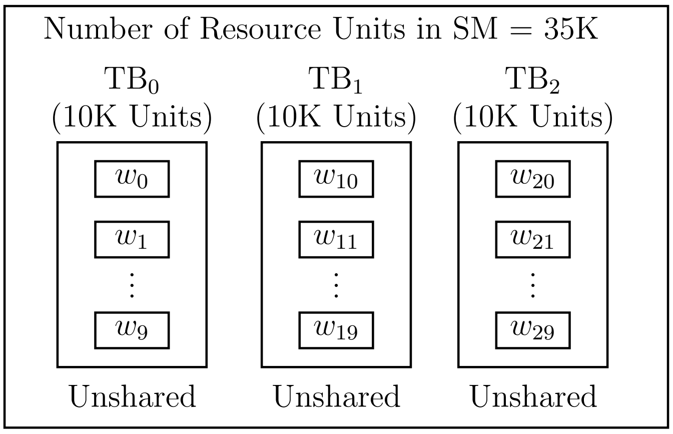
In the rest of the chapter, Sections 2.2 and 3.4.1.4 present the details of resource sharing and optimizations respectively. Section 2.4 discusses hardware overhead for implementing resource sharing. Section 2.5 describes the experimental evaluation, and Section 2.6 summarizes the chapter.

2.2 Resource Sharing

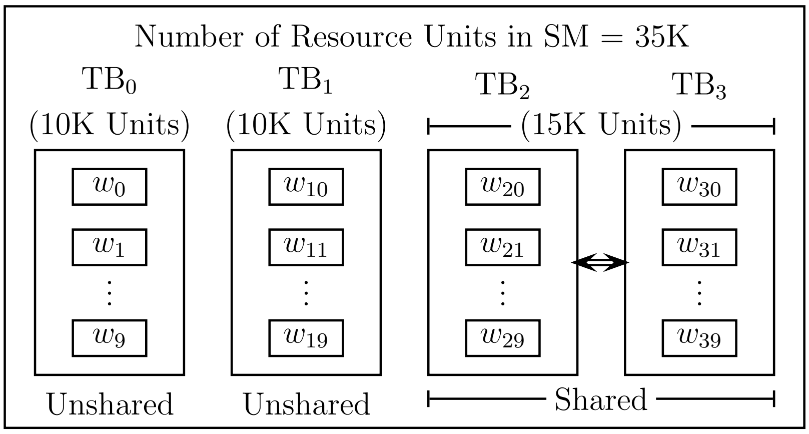
The following example illustrates how we can increase the number of thread blocks in an SM by allowing two thread blocks to share resources.

Example 2.2.1. *Consider an application that has thread blocks of size 10 warps (320 threads), and a thread block requires 10K resource units to complete its execution. If an SM has 35K resource units, at most 3 thread blocks can be resident on each SM by utilizing 30K resource units; the remaining 5K units are wasted. The schematic of this approach (baseline) is shown in Figure 2.3(a), where thread blocks TB_0, TB_1, and TB_2 are scheduled on an SM.*

In order to reduce the wastage of resources, our approach allocates one more thread block (TB_3) in sharing mode with TB_2 Figure 2.3(b). Instead of allocating 10K resource units separately to each of the thread blocks TB_2 and TB_3, a total of 15K units for the two blocks are allocated as follows: each of TB_2 and TB_3 is allocated 5K units exclusively (Private or Unshared Resource), while the remaining



(a) Default Approach



(b) Resource Sharing

Figure 2.3: Approaches to Resource Allocation

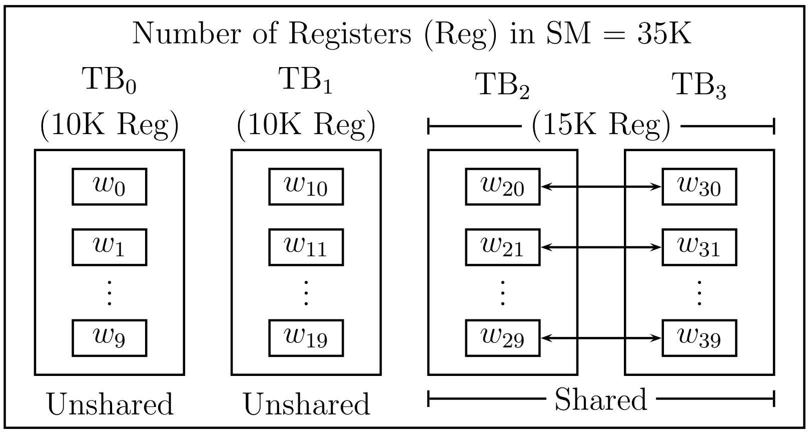


Figure 2.4: Resource Allocation with Register Sharing

5K units (Shared Resource) are all allocated to TB$_2$ or TB$_3$ whoever needs any one of these resources first*. The other thread block (which did not get the ownership of shared resources), when it needs* any *of the shared resources, waits till the owner block finishes.* □

We refer to any two thread blocks as *Shared Blocks* when they share resources exclusively (for example TB$_2$ and TB$_3$ in Figure 2.3(b)), and the warps of such thread blocks as *Shared Warps*. Thread blocks (warps) that do not participate in sharing are referred to as *Unshared Blocks* (*Unshared Warps*). We describe in detail our sharing approach for two types of resources (a) Registers, and (b) Scratchpad.

2.2.1 Register Sharing

Figure 2.4 shows an example of register allocation scheme, in which we allocate 10K registers to each thread block TB$_0$ and TB$_1$. The remaining 15K registers are shared between thread blocks TB$_2$ and TB$_3$ such that each pair of warps in these thread blocks are allocated 1.5K registers as described next. We refer to TB$_0$ and TB$_1$ as unshared thread blocks, whereas, TB$_2$ and TB$_3$ as shared thread blocks.

Example 2.2.2. *Consider the pair of warps W_{20} and W_{30} that participate in sharing. We allocate 0.5K registers (private or unshared registers) each to W_{20} and W_{30}. The remaining 0.5K registers are shared registers, that are allocated to these warps* together *in a shared but exclusive manner, i.e., only one of them can access the*

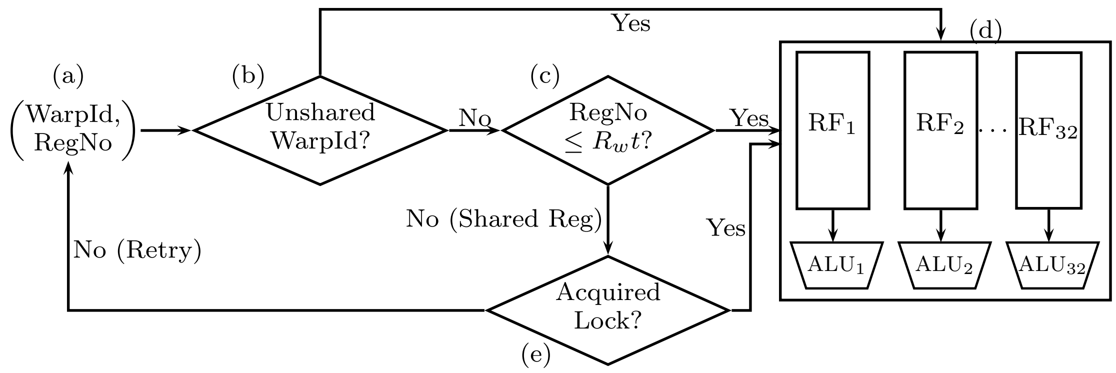


Figure 2.5: Register Access Mechanism

pool of shared registers at a time. For example, if warp W_{20} accesses any of the shared registers first, exclusive access to all the 0.5K shared registers is given to W_{20}, while W_{30} is prevented from accessing any of those 0.5K shared registers till W_{20} finishes. This implies, W_{30} can continue its execution until its first access to any of the 0.5K shared registers and waits until the shared registers are released. Only after W_{20} finishes execution, W_{30} can access the shared registers and continue. This way, additional warps make some *progress, which helps in hiding execution latencies.* □

To generalize this idea and to compute the increase in number of thread blocks, we will consider a GPU that provides R registers per SM. Also, consider a thread block that requires R_{tb} registers, and each warp in the thread block requires R_w registers to complete its execution. To increase the number of thread blocks that share registers with other existing thread blocks in the SM, we allocate $R_{tb}(1+t)$ (for any threshold $0 < \text{t} < 1$) registers to each pair of shared thread blocks, instead of allocating $2R_{tb}$ registers to them (in Figure 2.4, t is 0.5). Equivalently we allocate $R_w(1+t)$ registers per two warps from these thread blocks (i.e., one warp from each shared thread block in the pair), such that each of these warps can access $R_w t$ unshared registers independently, and they can access the remaining $R_w(1-t)$ shared registers only when granted access.

We allocate registers to a warp dynamically when it requires to access the registers on its first usage, and we deallocate them from the register file after the warp has finished its execution, as described in GPGPU-Sim [3]. Every unshared register is allocated as per the request, but the shared registers are allocated to only

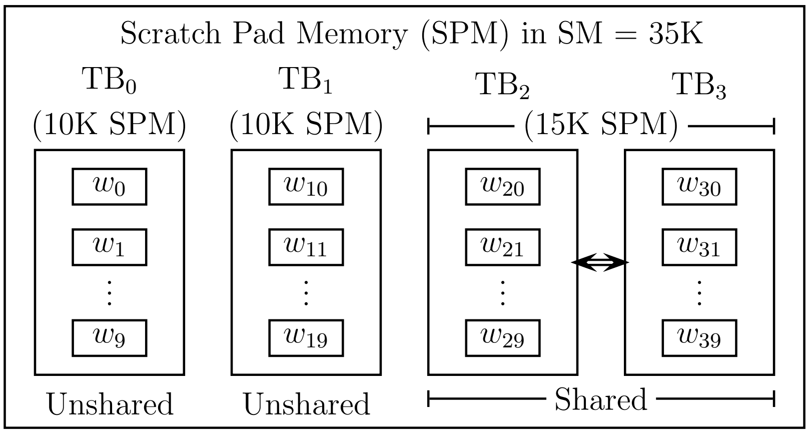


Figure 2.6: Resource Allocation with Scratchpad Sharing

a warp that has obtained an exclusive access to the shared register. To detect a register accessed by a warp as shared or unshared, and to efficiently access it from the register file unit, we modify the existing register file access mechanism as shown in Figure 2.5. When a warp (WarpId) needs to access a register (RegNo), we first check if the warp is an unshared warp, i.e., if it belongs to an unshared thread block (Figure 2.5, Step (b)). If it is an unshared warp, it can directly access the register from register file using a combination of (WarpId, RegNo). If WarpId is a shared warp, the accessed register is an unshared register if RegNo $\leq R_w t$ (Step (c)). This is because $R_w t$ number of unshared registers are allocated to each warp. If RegNo $> R_w t$, we treat the register as a shared register. A warp can access an unshared register directly from the register file, but it can access a shared register only when it gets exclusive access by acquiring a lock (Step (e)), otherwise it retries the access in another cycle[1].

2.2.2 Scratchpad Sharing

Figure 2.6 shows an example of *Scratchpad Sharing*, where we consider a GPU that has 35K units of scratchpad memory per SM, and each thread block requires 10K units. To increase number of resident thread blocks with scratchpad sharing, we allocate 10K units to each TB_0 and TB_1; the remaining 15K scratchpad units are allocated together for thread blocks TB_2 and TB_3 such that each one gets 5K units

[1]The details of required additional storage units are described in Section 2.4.

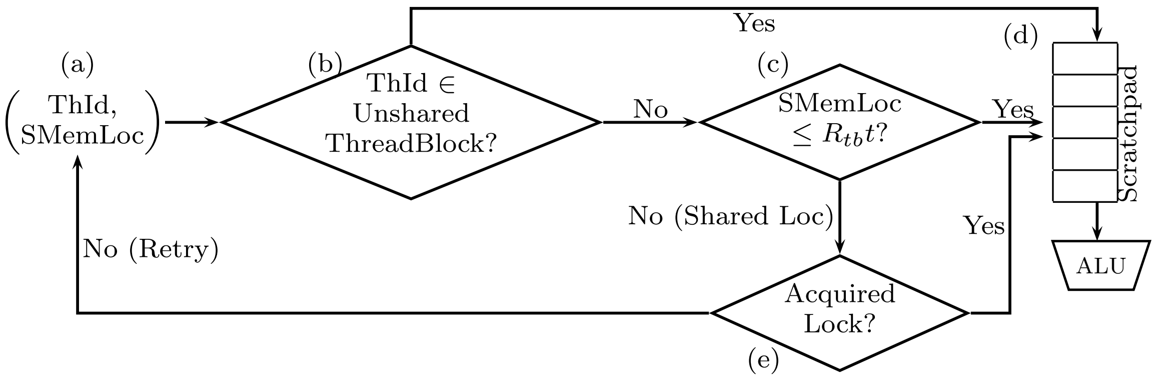


Figure 2.7: Scratchpad Access Mechanism

in private mode and the remaining 5K units are accessed in exclusive mode, i.e., only one thread block can access it at a time. Unlike register sharing, we can not distribute 1.5K scratchpad memory to each pair of warps because any thread within a thread block can access any scratchpad location allocated for that block. Similar to register sharing approach, we refer to TB_0 and TB_1 as unshared thread blocks, whereas, TB_2 and TB_3 as shared thread blocks.

When a thread from the shared thread block (say TB_2) needs to access a memory location from shared scratchpad, it gains an exclusive access by acquiring a lock. As long as TB_2 is running, no thread from TB_3 can access the shared scratchpad locations and hence the corresponding warps of TB_3 will have to wait for TB_2 to finish before they can proceed further. But warps of TB_3 that do not access the shared scratchpad locations can continue execution.

The implementation to support scratchpad sharing in GPGPU-Sim is shown in Figure 2.7. The steps for the shared scratchpad access follow the rules similar to the shared register access. When a thread (Thread Id: $ThId$) needs to access a scratchpad location ($SMemLoc$), we need to check if it is from an unshared thread block. If it belongs to an unshared thread block, it can access the location directly from scratchpad memory (Figure 2.7 Step (b)). Otherwise, we need to make another check if it accesses unshared scratchpad location (Step (c)). The thread accesses unshared scratchpad location if $SMemLoc < R_{tb}t$ because we allocate $R_{tb}t$ units of scratchpad memory to each of the shared thread blocks. Otherwise, we treat the location as shared scratchpad location. A thread can access unshared scratchpad location directly, however it can access the shared scratchpad location only after

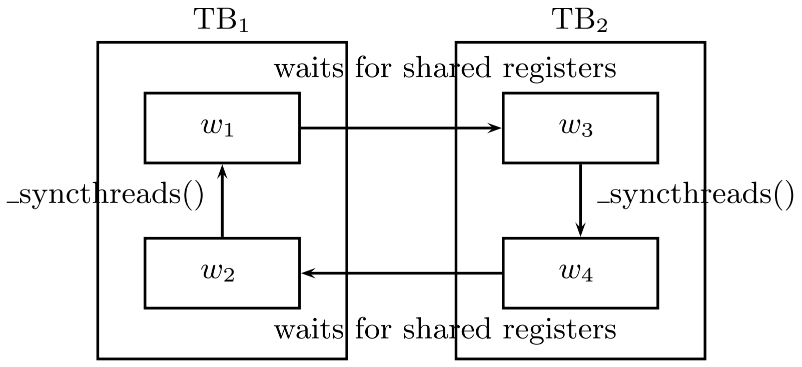


Figure 2.8: Deadlock in the Presence of Barrier Instructions

acquiring the exclusive lock as (Step (e)). Otherwise, it retries the access in the next cycle.

2.2.3 Can Resource Sharing Cause a Deadlock?

Example 2.2.3. *Consider a scenario shown in Figure 2.8 with register sharing, where two thread blocks TB_1 and TB_2 are in shared mode. Assume that W_2 and W_3 have already acquired locks for accessing shared registers. Also, assume that the warps W_2, W_3 are waiting for warps W_1 and W_4 respectively, to arrive at a barrier instruction (_syncthreads()). Now, if warp W_1 tries to acquire a lock to access shared registers from W_3, and W_4 tries to acquire a lock to access shared registers from W_2, then a deadlock occurs.* □

To avoid deadlock in the register sharing approach, we always ensure that if thread blocks TB$_1$ and TB$_2$ share registers, then a warp from TB$_1$ (TB$_2$) can acquire a lock only when either (a) none of the warps from TB$_2$ (TB$_1$) have acquired a lock for the shared registers, or (b) the warps from TB$_2$ (TB$_1$) that have acquired exclusive access to the shared registers have finished their execution. For the above example, if warp W_3 already has acquired a lock, W_2 can not acquire a lock, avoiding the deadlock.

A deadlock can never occur with scratchpad sharing. Consider two thread blocks TB$_1$ and TB$_2$ that share scratchpad. When a warp from shared thread block (say TB$_1$) acquires a lock, no other warp from TB$_2$ is given access to the shared scratchpad region until TB$_1$ finishes its execution. So, only the warps from TB$_2$ that require accessing the shared resources wait for TB_1 to finish. Warps from TB_1 never wait for TB_2 to finish. Hence there is no deadlock cycle.

2.2.4 Computing the Number of Effective Thread Blocks

A naive method of sharing, where each thread block is sharing resources with some other thread block, may launch more thread blocks as compared to default (non-sharing) approach. However, the number of thread blocks that make progress (*effective thread blocks*) per SM can be less than that for non-sharing.

Example 2.2.4. *Consider a configuration where each SM has 36 resource units, and each thread block requires 10 units of resource. Without any sharing, 3 thread blocks are resident in each SM. With sharing (say t=0.8), it is possible to launch 4 resident thread blocks in each SM, such that block 1 shares resources with block 2 and together get 18 ((1+0.8)*10) resource units. Similarly, block 3 with shares block 4 and gets 18 resource units. In this case, it can happen that block 2 and 4 start accessing shared resources causing blocks 1 and 3 to wait. Effectively only two thread blocks (blocks 2 and 4) will make progress in this naive sharing approach, whereas all 3 blocks can make progress in the non-sharing approach.*

To avoid this, we describe a method to compute the total number of thread blocks (Shared + Unshared) to be launched per SM such that the number of effective thread blocks using sharing approach is no less than that of non-sharing approach. We use the following notations:

1. R: Number of units of resource available per SM,
2. R_{tb}: Number of resource units required by a thread block,
3. S: Number of pairs of thread blocks that are to be launched per SM in shared mode,
4. U: Number of thread blocks to be launched in an SM that do not share resources with any other thread block,
5. M: Maximum number of thread blocks to be launched in an SM,

6. t: Threshold for computing the number of resources that a thread block shares with another thread block. For a given threshold value t $(0 < t < 1)$ we allocate $(1+t)R_{tb}$ resource units per two shared thread blocks, in which $(1-t)R_{tb}$ resource units are shared.

Without sharing, we can launch up to $\lfloor R/R_{tb} \rfloor$ thread blocks in an SM, and all of them make progress. Whereas in our approach, if two thread blocks are launched in sharing mode, at least one thread block always makes progress. So, when S shared pairs are launched in an SM, at least S thread blocks always make progress. Also, if U unshared thread blocks are launched in the SM, they always make progress. Therefore, at least $S+U$ thread blocks always make progress with our approach. In order to keep the number of effective thread blocks in our approach to be same as that of no-sharing approach, we need the following relation to hold:

$$S + U = \left\lfloor \frac{R}{R_{tb}} \right\rfloor \tag{2.1}$$

For each shared pair of thread blocks, we allocate $R_{tb}(1+t)$ resource units and for each unshared thread block, we allocate R_{tb} resource units. Since the total number of resource units available in the SM is R, we have:

$$UR_{tb} + SR_{tb}(1+t) \leq R \tag{2.2}$$

The total number of thread blocks that can be launched in sharing approach is equal to the number of unshared thread blocks plus twice the number of shared pairs, i.e.,

$$M = U + 2S \tag{2.3}$$

Using Equations 2.1, 2.2, and 2.3,

$$M = \left\lfloor \frac{R}{R_{tb}} \right\rfloor + \frac{1}{t}\left(\frac{R}{R_{tb}} - \left\lfloor \frac{R}{R_{tb}} \right\rfloor\right) \tag{2.4}$$

Since the actual number of thread blocks that can reside in an SM also depends on other factors, such as (a) maximum number of resident threads per SM, and (b) maximum number of resident thread blocks per SM; the number of thread blocks that are launched in an SM by our approach is the minimum of values obtained

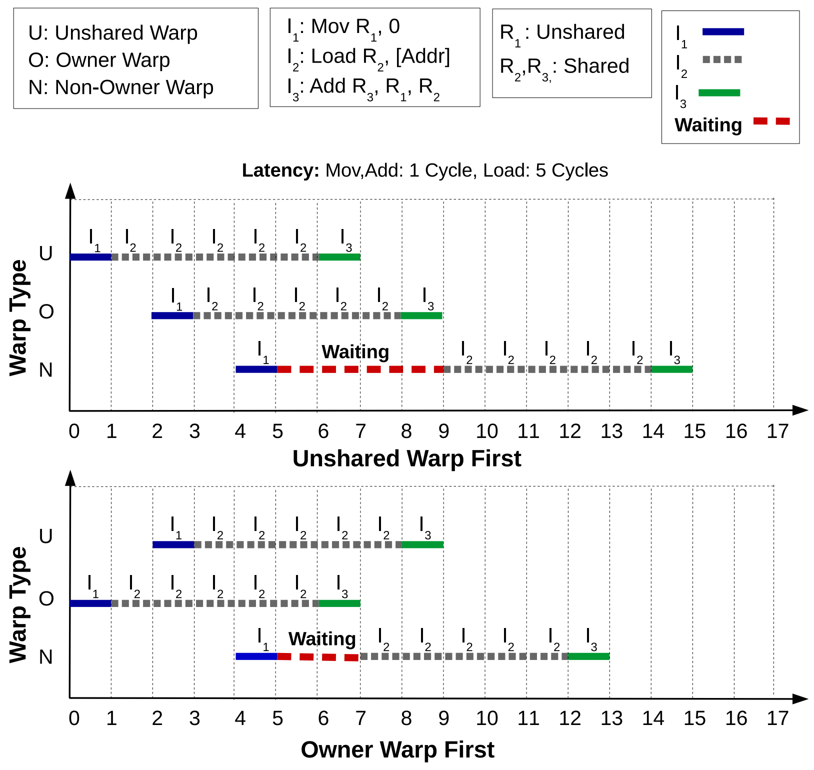


Figure 2.9: Warp Scheduling

using the factors (a), (b), and the value M. When the computed number of thread blocks launched by our approach is more than that of baseline approach (i.e., $\left\lfloor \frac{R}{R_{tb}} \right\rfloor$), we enable our resource sharing approach; otherwise, we launch all the thread blocks in unsharing mode.

2.3 Optimizations

With the proposed resource sharing approach, each SM has unshared and shared warps, and scheduling these warps plays a very important role in determining the performance of applications. We propose an optimization called "Owner Warp First (OWF)" to schedule these warps effectively. If two thread blocks TB$_i$ and TB$_j$ are a shared pair, and at least one of the warps of TB$_i$ waits for shared resources from TB$_j$, we call TB$_j$ as *Owner Block*, and the warps that belong to TB$_j$ are called *Owner Warps*. TB$_i$ is called *Non-Owner Block* and the warps of TB$_i$ are called

Non-Owner Warps. As soon as the owner thread block finishes its execution, it transfers its ownership to the non-owner thread block (i.e., the non-owner thread block becomes the owner), and a new non-owner thread block gets launched.

2.3.1 Scheduling Owner Warp First (OWF)

A warp scheduler in the SM issues a warp every cycle from a pool of ready warps. With our solution, the warps can be categorized into three types viz., unshared, shared owner and shared non-owner. In register sharing, shared non-owner warps depend on the corresponding shared-owner warps to release registers, before they can make progress. Similarly, with scratchpad sharing, warps from non-owner thread blocks wait for owner thread blocks to complete their execution. Hence scheduling of the warps plays a role in improving the performance of applications.

Example 2.3.1. *Consider a scenario shown in Figure 2.9. Assume that an SM contains 3 warps: unshared (U), shared owner (O), shared non-owner (N) warps, and each warp needs to execute three instructions (I_1, I_2, and I_3) as indicated in the figure. Assume that latency of Mov and Add instruction is 1 cycle, and the latency of Load instruction is 5 cycles. Also, assume that register R_1 is an unshared resource and R_2, R_3 are shared resources. If unshared warp is prioritized over owner warp (shown as* Unshared Warp First*), the unshared warp executes I_1 in the first cycle, and it starts execution of I_2 in the 2nd cycle. However, it can not start I_3 in the 3rd cycle because register R_2 of I_3 is dependent on the instruction I_2, and I_2 takes 5 cycles to complete the execution. If owner warp is prioritized over non-owner warp, it can start execution in the 3rd cycle. The non-owner warp which has the least priority can start its execution I_1 at the 5th cycle. However, it can not execute I_2 in the 6th cycle because it needs to acquire access to the shared resource R_2, which is held by its owner warp. Hence, it waits until the owner warp releases the shared resources (i.e., till the 9th cycle). The non-owner warp can resume its execution in the 10th cycle and can finish in 15 cycles.* □

To minimize the waiting time of the non-owner warps, we propose an algorithm, *Owner Warp First (OWF)*, that prioritizes warps in the order: shared owner, unshared, and shared non-owner. Giving the highest priority to shared owner warps helps finish them sooner, and hence the dependent shared non-owner warps can make progress. Since non-owner warps depend on their corresponding owner warps for shared resources, giving them low priority helps in hiding stalls when other types of warps are not ready to run.

Example 2.3.2. *As seen in Figure 2.9, with OWF approach, owner warp can finish sooner, i.e., in 7 cycles. Similarly unshared warp, with second priority, can finish in 9 cycles. Since the non-owner warp has low priority, it can start executing I_1 in the 5th cycle. It can overlap the execution of I_2 with the unshared warp in the 8th cycle because its owner warp has released the shared resources. Further, it can finish the execution in 13 cycles, thus improving the overall performance.* □

2.3.2 Unrolling and Reordering of Register Declarations

In register sharing, non-owner warps need to wait for owner warps when they try to access shared registers. If the very first instruction issued by a non-owner warp uses a shared register, then the warp has to wait and can not start its execution until corresponding owner warp has released the shared register. In order to allow the non-owner warps to execute as many instructions as possible before stalling due to unavailability of shared registers, we unroll and reorder the register declarations.

Example 2.3.3. *Consider the PTXPlus [3] code shown in Figure 2.10(a), which is generated by GPGPU-Sim [3] for the sgemm application from Parboil Suite [7]. The first instruction of the code accesses registers p0 and r124, which get the register sequence numbers as 31 and 35 according to the declaration. These registers are part of the shared registers for a certain threshold value t. Hence, a non-owner warp has to wait until the registers are released. To delay accessing the shared registers, we unroll and rearrange the order of the register declarations so that p0, r124 become unshared registers (i.e., they get the register sequence numbers as 1 and 3, as shown in Figure 2.10(b)). Hence the non-owner warps get to execute more number of instructions before they start accessing shared registers.*

To implement this optimization, we converted the assembly code (PTXPlus) produced by GPGPU-Sim into an optimized assembly code. To achieve this, we first find an order of registers according to their first usage. Further, to ensure

```
.reg .u32 $r<27>;
.reg .u32 $ofs<3>;
.reg .pred $p<4>;
.reg .u32 $r124;
.reg .u32 $o127;
...........
...........
...........
...........

set.le.s32.s32 $p0/$o127,
        s[0x003c], $r124;
mov.u32 $r16, $r124;
mov.u32 $r17, $r124;
mov.u32 $r9, $r124;
mov.u32 $r18, $r124;
mov.u32 $r10, $r124;
/* Code here */
```

(a) Normal Declarations

```
.reg .pred $p0;
.reg .u32 $o127;
.reg .u32 $r124;
.reg .u32 $r16;
.reg .u32 $r17;
.reg .u32 $r9;
.reg .u32 $r18;
.reg .u32 $r10;
...........
...........
set.le.s32.s32 $p0/$o127,
        s[0x003c], $r124;
mov.u32 $r16, $r124;
mov.u32 $r17, $r124;
mov.u32 $r9, $r124;
mov.u32 $r18, $r124;
mov.u32 $r10, $r124;
/* Code here */
```

(b) Unrolled Declarations

Figure 2.10: Unrolling and Reordering of Register Declarations

that unshared registers are used before shared registers, we modify the register declarations so that a register that has been used first is declared first. Finally, we modified the GPGPU simulator to use optimized PTXPlus code for simulating instructions. This optimization can be easily integrated at assembly level using CUDA compiler.

2.3.3 Dynamic Warp Execution

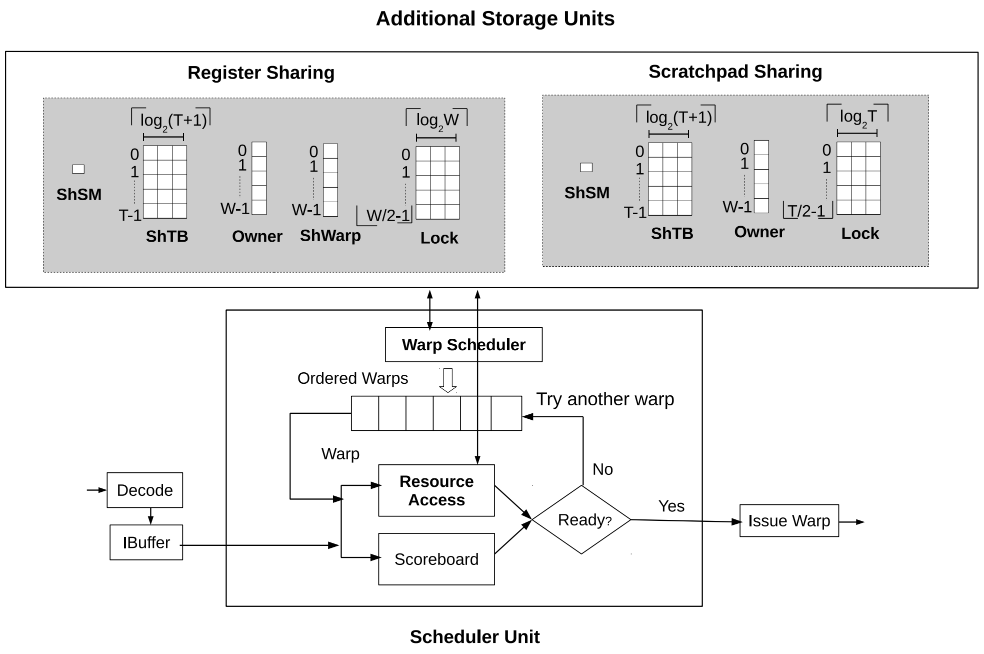
A study by Kayiran et. al. [46] shows that the performance of memory-bound applications can degrade with increase in the number of resident thread blocks. Executing additional thread blocks can increase L1/L2 cache misses, which leads to increase in the stall cycles. In register sharing, the additional warp (non-owner warp) resumes its execution as soon as its corresponding owner warp finishes, while in scratchpad sharing non-owner warps wait until its corresponding owner thread block finishes. In order to reduce the number of additional stalls due to the execution of non-owner warps in register sharing, we propose an optimization that can dynamically enable or disable execution of long latency instructions (memory) issued by the non-owner warps.

To control the execution of memory instructions from the non-owner warps, we monitor the number of stall cycles for each SM. When executing memory instructions from non-owner warp leads to increase in the number of stalls, we decrease the probability of executing further memory instructions from the non-owner warps. To illustrate this, consider a GPU that has N SMs, all in sharing mode. Our approach disables execution of memory instructions for the non-owner warps, only on a specific SM (e.g. SM_0). Every other SM, SM_i for $i \in \{1 \ldots N-1\}$, allows execution of memory instructions for the non-owner warps, and compares its stall cycles periodically with the stalls on SM_0. If stalls observed in the SM_i are more than the stalls appearing in SM_0, then the probability of executing memory instructions on SM_i from the non-owner warps is decreased by a predetermined value p. If the stalls in SM_i are less than that in SM_0, then the probability of executing memory instructions on SM_i from the non-owner warps is increased by the same value p. Thus, we reduce the number of stall cycles by controlling the execution of memory instructions.

After running several experiments, we selected the periodicity of monitoring to be 1000 cycles, which is to ensure that (a) the monitoring overhead is not high, and (b) sufficient number of stall cycles are observed. In our experiments, initially all the SMs (except SM_0) are allowed to execute all memory instructions, i.e., the probability of executing memory instructions from non-owner warp is 1. Depending on the stall cycles observed for an SM_i ($i \in \{1 \ldots N-1\}$), this probability for SM_i is decreased or increased by $p = 0.1$, but is kept within interval $[0, 1]$ as a saturating counter.

2.4 Hardware Requirement

Figure 2.11 shows the modified architecture to implement our proposed resource sharing approach. There are mainly two changes in the scheduling logic. The first change is that the warp scheduler uses OWF policy to prioritize warps, using the owner information. The second change is the inclusion of resource access check. A warp is considered to be ready for issuing only when it can access the required resources (resource access check) and has all its operands available (scoreboard check). The resource access unit follows the resource access mechanism (Figure 2.5 and Fig-



In the additional storage units, X dimension of each table refers to the number of bits, and Y dimension refers to the number of entries.

Figure 2.11: Modified Architecture for Resource Sharing

ure 2.7), and uses some additional storage units (shown in grey color in Figure 2.11) to determine the access to resources.

2.4.1 Storage Units Required for Register Sharing

1. Each SM requires a bit (shown as *ShSM* in Figure 2.11 in Additional Storage Units corresponding to Register Sharing) to specify whether sharing mode is enabled for it. This bit will be set when the number of thread blocks assigned to the SM using resource sharing is more than the default number of thread blocks per SM.

2. Each resident thread block stores its shared thread block id in the *ShTB* table, shown in the figure. If a thread block is in unsharing mode, its corresponding value is set to -1. For T thread blocks, $T\lceil\log_2(T+1)\rceil$ (assuming ids 0 to T-1 for T thread blocks, we can use id T to represent -1) bits are required per SM.

3. Each warp requires a bit for specifying the owner information, which is stored

in *Owner* table in the figure. This bit is set only when the warp is an owner warp. Hence, for W warps, W bits are needed.

4. Each warp requires a bit to specify whether it is in sharing or unsharing mode (shown as *ShWarp* table in the figure). A warp is set to be in sharing mode, when its corresponding thread block is in sharing mode. For W warps in an SM, W bits are required. For a warp in shared mode, its corresponding shared warp can be identified using the sharer thread block id of its thread block and its relative position in the thread block.

5. Each pair of shared warps uses a lock variable to access the shared registers exclusively. The lock variable is set to the id of the warp which has gained access to the shared registers. This is maintained in the *Lock* table in the figure. If an SM has W warps, there can be a maximum of $\lfloor W/2 \rfloor$ shared pairs of warps in the SM. Hence, we need a total of $\lfloor W/2 \rfloor \lceil \log_2 W \rceil$ bits per SM.

2.4.2 Storage Units Required for Scratchpad Sharing

1. Similar to register sharing, scratchpad sharing approach also requires *ShSM*, *ShTB*, and *Owner* tables as described above. These tables are shown in Figure 2.11 in Additional Storage Units corresponding to Scratchpad Sharing.

2. Each pair of shared thread blocks uses a lock variable to access the shared locations exclusively. The lock variable is set to the id of the thread block which has gained access to the shared scratchpad region. If an SM has T thread blocks, there can be a maximum of $\lfloor T/2 \rfloor$ shared pairs of thread blocks in the SM. Hence, we need a total of $\lfloor T/2 \rfloor \lceil \log_2 T \rceil$ bits per SM. Similar to register sharing, these values are maintained in the *Lock* table, shown in Figure 2.11 for scratchpad sharing.

The total amount of storage required (in bits) for a GPU with N SMs for implementing register sharing is:

$$(1 + T\lceil \log_2(T+1) \rceil + 2W + \lfloor W/2 \rfloor \lceil \log_2 W \rceil) * N$$

Table 2.4: Set-3: Benchmarks Limited by Threads or Blocks

Benchmark	Application	Kernel	Limited by
Rodinia	backprop	bpnn_layer forward_CUDA	Threads
GPGPU-Sim	BFS	Kernel	Threads
Rodinia	gaussian	FAN2	Blocks
GPGPU-Sim	NN	executeSecondLayer	Blocks

and for implementing scratchpad sharing is:

$$(1 + T\lceil \log_2(T+1)\rceil + W + \lfloor T/2 \rfloor \lceil \log_2 T \rceil) * N$$

For the architecture shown in Table 2.3, the additional storage required per SM is 273 bits for register sharing and 93 bits for scratchpad sharing.

In addition to storage units, the resource access unit requires two comparator circuits to implement the steps (b) and (c) shown in Figures 2.5 and 2.7. Similarly, it requires an arithmetic circuit to set the lock as shown in step (e).

2.5 Experiments and Analysis

We implemented our approach using GPGPU-Sim V3.X [3]. Table 2.3 shows the baseline architecture used for comparison. We evaluated our approach on several applications from GPGPU-Sim [14], Rodinia [19], CUDA-SDK [2], and Parboil [7] benchmarks. Depending on the resource requirement of applications, we divided the benchmarks into three sets. Set-1 (Table 2.1) consists of applications whose number of thread blocks per SM are limited by registers, and hence register sharing approach is applicable to them. Set-2 (Table 2.2), which is applicable to scratchpad sharing, has applications that are limited by scratchpad memory. For Set-1 applications (for register sharing), we set the scratchpad memory per SM to 48KB so that their number of resident thread blocks are not limited by scratchpad memory after applying register sharing. Set-3 (Table 2.4) has applications that are limited neither by registers nor by scratchpad memory (i.e., they are limited either by the number of resident threads or the number of resident thread blocks). We choose Set-3 applications to ensure that our approach does not degrade the performance of applications that are neither limited by registers nor scratchpad memory. For each application in Table 2.1 and Table 2.2, we show names of the kernels used for evaluation and the number of threads per thread block. In Table 2.1, we report the number of registers per thread for each kernel, which GPGPU-Sim uses to compute the number of resident thread blocks, and in Table 2.2 we show the amount of scratchpad memory used by each thread block. We simulated all the applications using PTXPlus assembly language, because (1) this chapter deals with register resource, also PTXPlus uses more optimal number of registers since it is directly obtained from SASS (2) this chapter require implementing more hardware features (which require GPGPU-Sim framework) and less software features.

We use the value of threshold (t) to configure the percentage of resource sharing. For example, if each thread block requires R_{tb} units of resource, and we choose $t = 0.1$, then we allocate $1.1 * R_{tb}$ resource units per two shared thread blocks, which means 90% of resource units (R_{tb}) are used as shared resource units. So for a given threshold t, we can compute the percentage of resource sharing as $(1-t)*100$. We analyzed the performance of our approach for each application by varying t and chose the threshold value as 0.1 (i.e., 90% resource sharing) for our results (Details are discussed in Section 2.5.1.8).

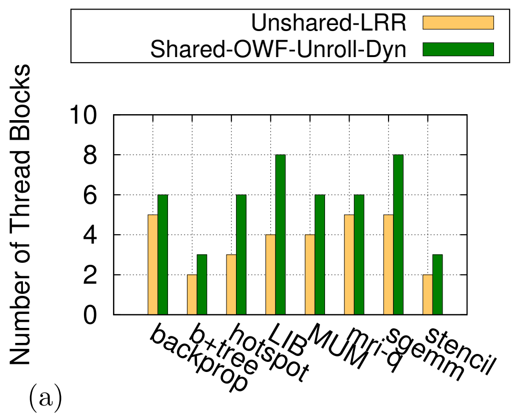
We measure the performance of our approach using the following metrics, which are reported by GPGPU-Sim [3]:

1. The Number of Resident Thread Blocks: It indicates the number of thread blocks that are launched in an SM. We choose this metric to compare the amount of TLP that is present in an SM.

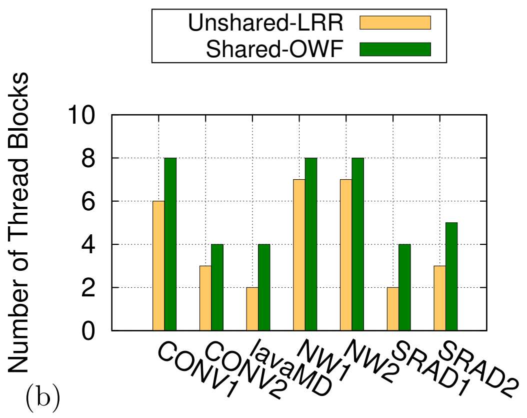
2. Instructions Per Cycle (IPC): It is the number of instructions that are simulated per core clock cycle. We use it to measure the performance of our modified GPU architecture with respect to benchmark applications.

3. Simulation Cycles: It is the number of cycles that a kernel takes to complete its simulation. We use this metric to measure performance of benchmarks applications with our modified GPU architecture.

4. Pipeline Stall Cycle: It is the cycle in which no warp can execute an instruction because the execution units are busy. This is to show that our approach can



(a)



(b)

Figure 2.12: Comparing Number of Resident Thread Blocks of Baseline Approach with (a) Register Sharing (b) Scratchpad Sharing

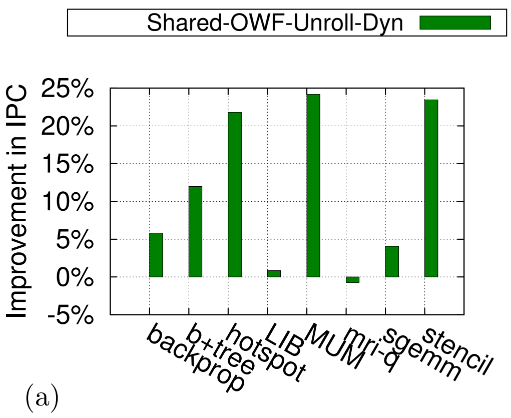
help in hiding the long latency instructions.

5. Idle Cycle: It is the cycle in which no warp is ready to execute next instruction. We choose this metric to show that the additional thread blocks launched by our approach help in minimizing the cycles in which SMs are idle.

2.5.1 Analyzing Benchmarks that are Applicable to Resource Sharing

2.5.1.1 Increase in the Number of Thread Blocks

Figures 2.12(a) and (b) show that resource sharing helps in increasing the number of thread blocks launched for the applications. Figure 2.12(a) compares the effective number of thread blocks launched by register sharing approach (denoted as Shared-OWF-Unroll-Dyn) with that of baseline implementation (denoted as Unshared-LRR). For applications *MUM*, *backprop*, *hotspot*, and *mri-q* our approach is able to launch 6 thread blocks (i.e., 1536 threads), which is the limit on the number of resident threads per SM. Applications *stencil* and *b+tree* launch 3 thread blocks per SM, compared to 2 in the baseline approach. For applications *LIB* and *sgemm* our approach is able to launch 8 thread blocks per SM, which is the limit on the number of resident thread blocks.



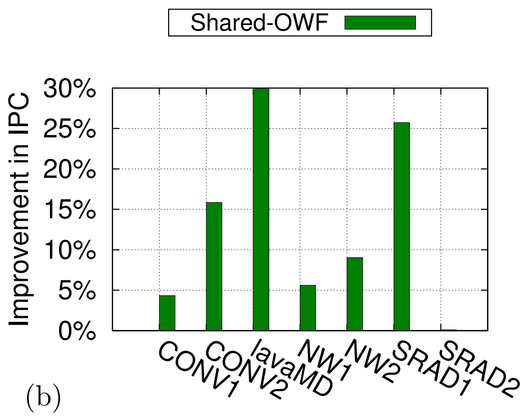
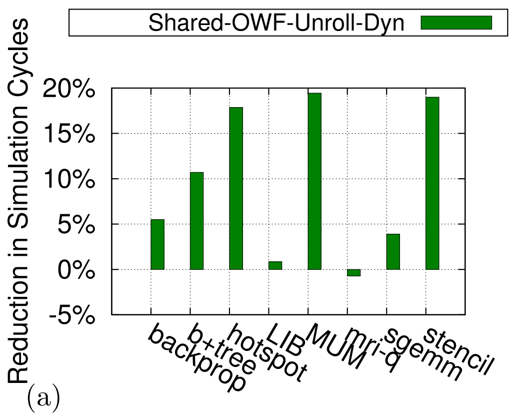


Figure 2.13: Performance Comparison of (a) Register Sharing (b) Scratchpad Sharing with Baseline Approach

In Figure 2.12(b), we compare the number of resident thread blocks launched by scratchpad sharing (labeled as Shared-OWF) with baseline approach. For applications *CONV1*, *NW1*, and *NW2*, we launch 8 thread blocks per SM, which is the limit on the number of resident thread blocks.

2.5.1.2 Performance Analysis

Figure 2.13(a) shows the improvement in IPC with register sharing over baseline LRR (Loose Round Robin) implementation. We observe that applications show an average (Geometric Mean) improvement of 11% with register sharing. Applications *b+tree*, *hotspot*, *MUM*, and *stencil* achieve significant speedups of 11.98%, 21.76%, 24.14%, and 23.45% respectively. Similarly Figure 2.13(b) shows the performance improvement in IPC with scratchpad sharing. We observe that applications show an average improvement of 12.5% with scratchpad sharing. *CONV2*, *lavaMD*, and *SRAD1* achieve speedups of 15.85%, 29.96%, and 25.73% respectively. These applications leverage all our optimizations to perform better. The performance improvement in IPC for *lavaMD* is due to two reasons: (1) The number of resident thread blocks launched by our approach is twice that of baseline approach (2) No instruction that uses scratchpad memory location falls into shared scratchpad. Note that this is the run-time behavior of the application. In this case, all the additional thread blocks execute instructions without waiting for shared thread blocks. Though *LIB* launches 8 thread blocks per SM with register sharing, it improves



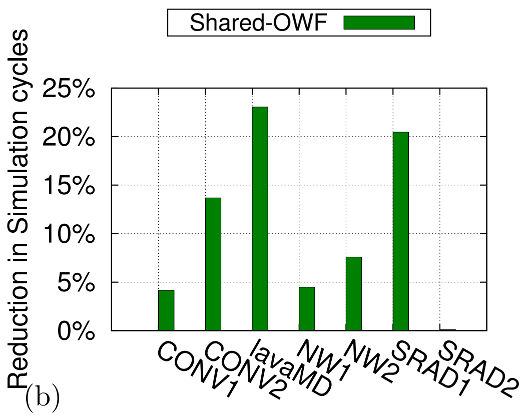


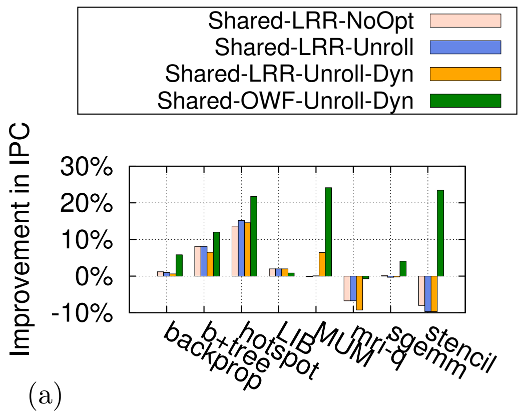
Figure 2.14: Percentage Decrease in Simulation Cycles (a) Register Sharing (b) Scratchpad Sharing

only by 0.84%. It is due to increase in L2 cache misses caused by additional shared blocks. The benchmarks *backprop* and *sgemm* achieve modest improvements of 5.82% and 4.06% respectively with register sharing. Similarly, *CONV1*, *NW1*, and *NW2* show improvements of 4.33%, 5.62%, and 9.03% respectively with scratchpad sharing. *mri-q* slows down by 0.72% because additional shared blocks increase L1 cache misses and hence increase the number of stalls. *SRAD2* shows improvement only up to 0.1% because a barrier instruction placed next to shared scratchpad access limits the progress of shared threads that do not access any shared scratchpad location.

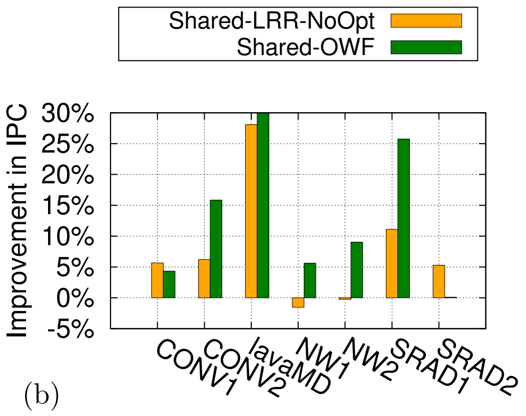
In Figures 2.14(a) and (b), we show the percentage decrease in the number of simulation cycles with register and scratchpad sharing when compared to baseline approach. Since the number of instructions executed in our approach is same as that of the baseline approach, all the applications that show improvement in IPC in Figures 2.13(a) and (b) will take less the number of simulation cycles for completing their execution using our approach. That is why Figure 2.13 and Figure 2.14 show similar trend.

2.5.1.3 Effectiveness of Optimizations

Figure 2.15(a) compares register sharing optimizations with baseline approach. We compare the results of register sharing when we do not use any optimization and



(a)



(b)

Figure 2.15: Performance Analysis of Optimizations for (a) Register Sharing (b) Scratchpad Sharing

use the existing baseline LRR scheduling policy (labeled Shared-LRR-NoOpt). Consider the application *hotspot*, it achieves a speedup of 13.65% even without using any optimization because the additional thread blocks launched by our approach help in hiding execution latencies. With register unrolling optimization (labeled Shared-LRR-Unrolled), we further see an improvement up to 15.18% because register unrolling enables threads to execute more instructions before they start accessing shared registers. Hence it can execute more instructions before it accesses shared registers. When we enable the dynamic warp execution (labeled Shared-LRR-Unrolled-Dyn), we see an improvement only up to 14.58% because it limits the execution of memory instructions from non-owner warps. However when we apply the OWF optimization (labeled Shared-OWF-Unrolled-Dyn), the application speeds further up to 21.76%. With OWF optimization, the priority of non-owner warps decreases compared to the other warps. Hence the memory instructions issued by non-owner warps do not interfere with the other warps, which minimizes the L1/L2 cache misses. We see that *b+tree* behaves similarly to *hotspot* in terms of performance gain by varying the optimizations.

MUM slows down by 0.15% when we do not use any optimization. We observe that increase in the resident thread blocks leads to increase in the number of memory instructions issued by non-owner warps, increasing L1 and L2 cache misses. Though we see an increase in the L1/L2 cache misses, the other instructions issued by the non-owner warps help in minimizing the stall cycles. With register unrolling optimization, we see a slight improvement (0.08%). When we apply the dynamic warp execution, it shows a speed up of 6.45%. From this, we analyze that dynamic warp execution reduces the additional stall cycles produced by issuing memory instructions from the non-owner warps. Further with OWF optimization, performance improves up to 24.14% because of the decrease in interference from non-owner warps.

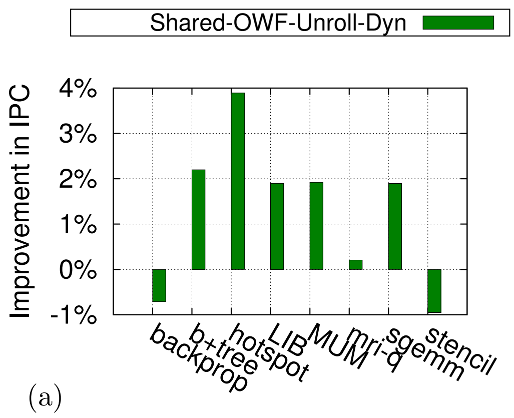
LIB shows an improvement of 2% using sharing with no optimizations. We observe the same performance even with unrolling optimization because the number of instructions that use unshared registers before they start accessing shared registers is exactly the same as without optimization. With dynamic warp execution, we still observe the same since in this application all the owner warps have completed executing all instructions before any non-owner warp starts issuing any memory instructions. With OWF optimization, we observe a small degradation because of increase in the number of stall cycles compared to the LRR policy.

The benchmarks *sgemm*, *backprop*, and *stencil* achieve good improvements only when OWF optimization is enabled. Since instructions issued by non-owner warps execute with the least priority, they do not interfere with other warps and hence minimize L1/L2 cache misses. We do not see any performance improvement with *mri-q* because the additional thread blocks increase L1 cache misses with our approach. However the slow down was reduced to 0.72% in the presence of all the optimizations.

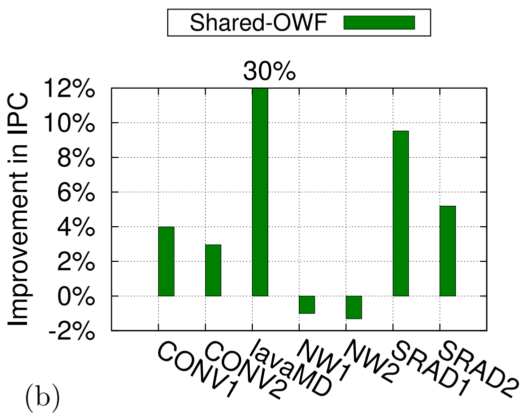
To summarize, memory-bound applications, like *MUM*, take advantage of our sharing approach in the presence of dynamic warp execution and OWF optimizations. Whereas, compute-bound applications, like *hotspot*, perform better even without any optimizations, and they further improve with OWF optimization.

In Figure 2.15(b), we show the effect of OWF optimization on scratchpad sharing. *lavaMD* shows an improvement of 28% even without any optimization (labeled shared-LRR-NoOpt). It is because additional thread blocks do not access any memory location which belongs to shared scratchpad memory. *CONV1*, *CONV2*, *SRAD1*, and *SRAD2* applications show improvements of 5.68%, 6.21%, 11.1%, and 5.28% respectively without applying optimization, which is due to additional thread blocks that help in hiding the latencies.

With OWF optimization, *CONV2*, *NW1*, *NW2*, and *SRAD1* applications im-



(a)



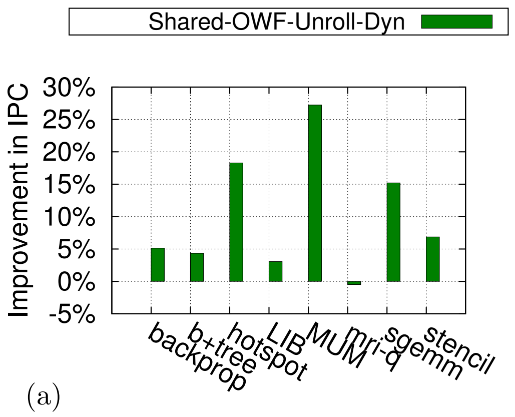
(b)

Figure 2.16: Performance Comparison of (a) Register Sharing (b) Scratchpad Sharing with GTO (Baseline) Scheduler

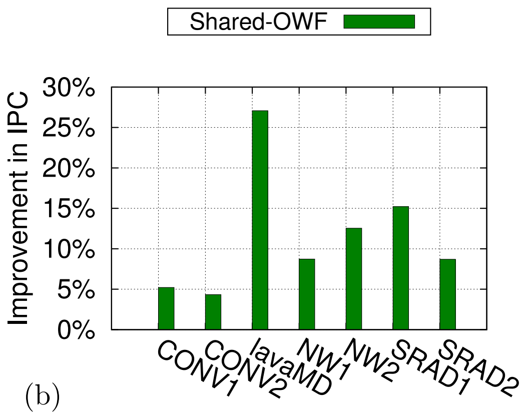
prove up to 15.85%, 5.62%, 9.03%, and 25.73% respectively. Since OWF optimization schedules the owner warps efficiently, it helps in minimizing stall cycles thus improving IPC value. *lavaMD* improves up to 30% since it has more benefit with sharing than OWF optimization. *CONV1* and *SRAD2* perform better when no optimization is applied because these applications go through extra cache misses (L1 and L2) and extra stall cycles with OWF optimization when compared to no optimization.

2.5.1.4 Comparison with Other Schedulers

In Figures 2.16(a) and (b), we show the performance improvement in the register sharing and the scratchpad sharing, respectively, over GTO (Greedy Then Old) scheduler. We observe that our approach shows an improvement up to 3.9% with register sharing and shows an improvement up to 30% with scratchpad sharing. *backprop* shows the same number L2 misses as the baseline GTO, but it has more L1 misses with our approach. In *stencil*, we observe extra L2 misses with our approach. *NW1* and *NW2* degrade with our approach because they have less number of stall cycles with GTO scheduler than our approach. Further, as shown in Figure 2.17(a) and (b) we observe an improvement up to 27.22% with register sharing and up to 27.08% with scratchpad sharing over the two-level scheduling policy.



(a)

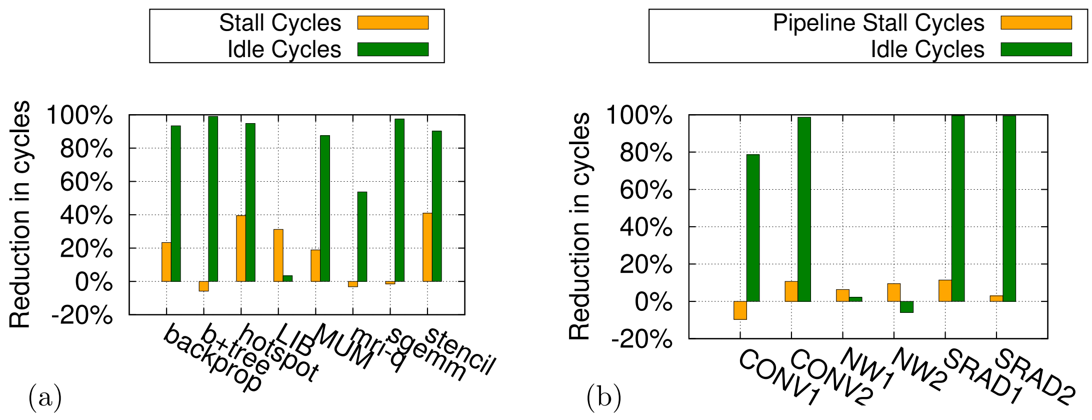


(b)

Figure 2.17: Performance Comparison of (a) Register Sharing (b) Scratchpad Sharing with Two-Level (Baseline) Scheduler

2.5.1.5 Reduction in Idle and Stall Cycles

In Figures 2.18(a) and (b), we report percentage decrease in the number of idle cycles and pipeline stall cycles when compared to the baseline approach. We observe that, all applications but one show reduction in the number of idle cycles (up to 99%). This is expected because with the increase in the number of thread blocks, number of instructions that are ready to execute also increase. For *MUM*, *LIB*, *backprop*, *hotspot*, and *stencil* the stall cycles also reduce with register sharing. Similarly for *CONV2*, *NW1*, *NW2*, *SRAD1*, and *SRAD2* applications, number of stall cycles reduce with scratchpad sharing. It indicates the additional thread blocks launched with our approach hide the long execution latencies in a better way. We observe an increase in the stall cycles for applications *b+tree* and *sgemm*. However, since the number of idle cycles have significantly reduced, overall we see a benefit with our approach. For *mri-q*, the number of stall cycles increases with our approach due to the increase in L1 cache misses. *lavaMD* shows an increase of 259 stall cycles because the additional threads wait for execution units (SP units) to become ready. For *CONV1* we see an increase in number of stalls with our approach due to L1 cache misses.



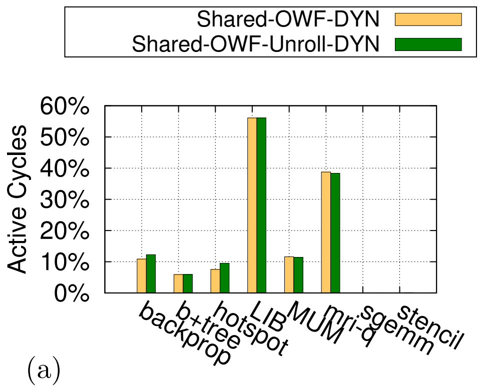
Note that lavaMD is not shown in (b) as it has zero stall cycles in baseline approach and 259 cycles in shared-OWF approach. It shows 49.5% decrease in idle cycles.

Figure 2.18: Percentage Decrease in Stalls and Idle Cycles for (a) Register Sharing (b) Scratchpad Sharing

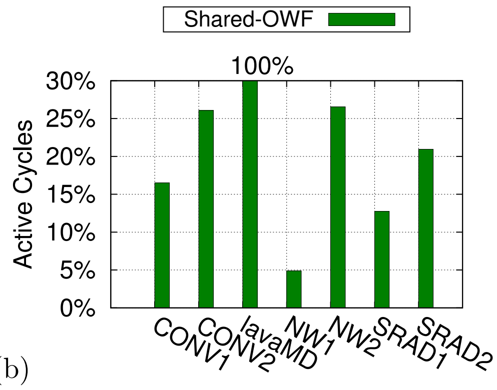
2.5.1.6 The Progress of Additional Warps

Figure 2.19 shows the effectiveness of resource sharing by measuring the progress made by the additional warps launched in our approach. In the figure, we report the percentage of simulation cycles in which an additional warp made progress till it becomes an owner warp. In the figure, the average over all additional warps is shown, denoted as *Active Cycles*. For register sharing, an additional warp makes progress in two phases: (1) from the time it is launched till it starts waiting for the shared resources from its corresponding shared warp (2) from the time its shared warp has finished the execution till its corresponding shared thread block has finished the execution (where the warp becomes the owner). In case of scratchpad sharing, an additional warp can make progress from the time it is launched till it needs to wait for shared resources.

The Figures 2.19(a) and (b), show the results for register sharing and scratchpad sharing respectively. From Figure 2.19(a), we observe the additional warps could make progress up to 56% with registers sharing. We find that *sgemm* and *stencil* make very less progress because their additional warps need to start accessing the shared resources early, and the resources are released by their shared warps very late. For *LIB*, we observe that some additional warps have a short life time, and



(a)



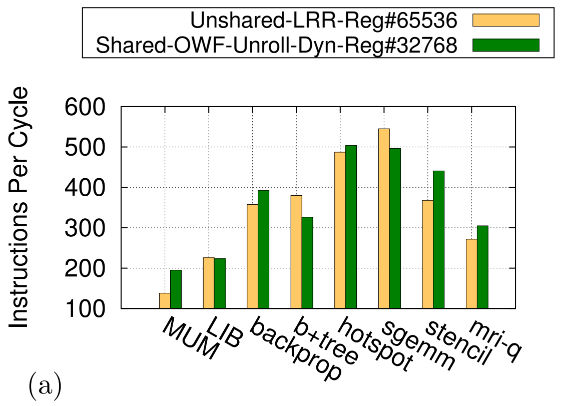
(b)

Figure 2.19: Effectivenss of the Additional Thread Blocks with (a) Register Sharing (b) Scratchpad Sharing

their shared resources are released early. Hence, these warps could make more progress in their life time. Figure 2.19(a) also shows that unrolling and reordering of register declarations is less effective since the additional warps access the shared resources early even after the optimization. From Figure 2.19(b), we notice that additional warps progress up to 100% with scratchpad sharing. Further, results of scratchpad sharing are more effective than that of register sharing because the additional warps in the Set-2 applications spend significant time before they start accessing any scartchpad memory.

2.5.1.7 Resource Savings

We also compare our approach against LRR Scheduler that uses twice the number of resources. In Figure 2.20(a), the baseline approach (labeled as Unshared-LRR-Reg#65536) uses 64K registers, whereas our approach uses only 32K registers. Even with an increase in the number of registers and hence an increase in the number of resident thread blocks in the baseline approach, our approach performs better in 5 out of 8 applications. *MUM* performs better with our approach, even though the number of thread blocks is same (6) in both the approaches because dynamic warp execution optimization helps minimizing the stalls produced by the additional thread blocks. *sgemm*, *b+tree*, and *LIB* perform better with the baseline approach due to an increase in the number of resident thread blocks and hence an increase in the



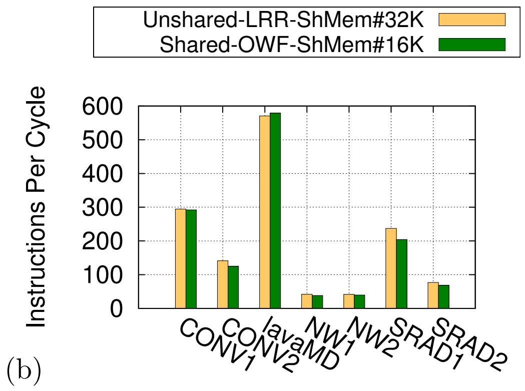


Figure 2.20: Comparison with LRR that Uses Twice the Number of (a) Registers (b) Scratchpad

number of active warps. In Figure 2.20(b), we compare scratchpad sharing approach that uses 16K bytes of memory with that of baseline approach that uses 32K byes of memory. From the figure we observe that, performance of *CONV1*, *NW1*, and *NW2* is comparable to that of baseline approach because our approach can launch the same number of thread blocks as the baseline approach. *lavaMD* performs better than baseline approach because sharing helps in minimizing latencies. *CONV2*, *SRAD1*, and *SRAD2* degrade with our approach because number of resident thread blocks in our approach is less, and number of stall cycles in our approach is more compared to baseline approach.

2.5.1.8 Effect of Sharing on Performance

In Table 2.5 and Table 2.7, we analyze the performance of resource sharing approach with the amount of sharing. From the results we observe that, most of the applications perform better when the amount of sharing is 90%. It is because, as shown in Tables 2.6 and 2.8, with increase in the amount of resource sharing, the number of resident thread blocks will increase. These resident thread blocks help in hiding long latencies and hence help in achieving high throughput. From the Tables 2.5 and 2.7, we also notice that, all applications behave same at 0% and 10% sharing. At these percentages of sharing, the number of resident thread blocks that are launched with our approach is same as that of baseline implementation. However, the perfor-

Table 2.5: Effect on IPC with Register Sharing

% Sharing	0%	10%	30%	50%	70%	90%
backprop	389.9	389.9	389.9	389.9	394.1	392.8
b+tree	318.5	318.5	318.5	323.3	326.1	326.1
hotspot	489.5	489.5	489.5	475.2	476.9	503.59
LIB	218.0	218.0	203.0	203.0	216.3	223.3
MUM	190.5	190.5	190.5	192.1	192.4	194.9
mri-q	303.7	303.7	303.7	303.7	305.3	305.0
sgemm	490.6	490.6	490.6	490.6	446.3	496.7
stencil	448.2	448.2	448.2	448.2	448.2	440.8

Table 2.6: Effect on Resident Thread Blocks with Register Sharing

% Sharing	0%	10%	30%	50%	70%	90%
backprop	5	5	5	5	6	6
b+tree	2	2	2	3	3	3
hotspot	3	3	3	4	4	6
LIB	4	4	5	5	6	8
MUM	4	4	4	5	5	6
mri-q	5	5	5	5	6	6
sgemm	5	5	5	5	6	8
stencil	2	2	2	2	2	3

mance is not same as baseline approach since in these cases we leverage our OWF optimization. With OWF, the warps are arranged according to the priorities of the owner, the unshared, and the non-owner warps. Since there are no additional thread blocks, all the thread blocks are in unshared mode. Hence, OWF optimization sorts the warps according to their dynamic warp ids. *SRAD2* (Table 2.7) performs better at 50%, because at this sharing, the number of instructions that get executed before they start enter the shared scratchpad memory region is more than that at 90%. Also at 50% sharing, these extra instructions belong to loop statements, hence we observe more IPC values.

Table 2.7: Effect on IPC with Scratchpad Sharing

% Sharing	0%	10%	30%	50%	70%	90%
CONV1	280.33	280.33	280.33	280.33	288.82	292.24
CONV2	119.29	119.29	119.29	119.29	119.02	124.6
lavaMD	452.29	452.29	452.29	452.29	452.29	578.85
NW1	39.96	39.96	39.96	38.67	38.37	38.37
NW2	41.93	41.93	41.93	42.14	40.54	39.72
SRAD1	188.13	188.13	188.13	229.38	208.27	204.32
SRAD2	63.48	63.48	63.48	63.52	63.62	68.29

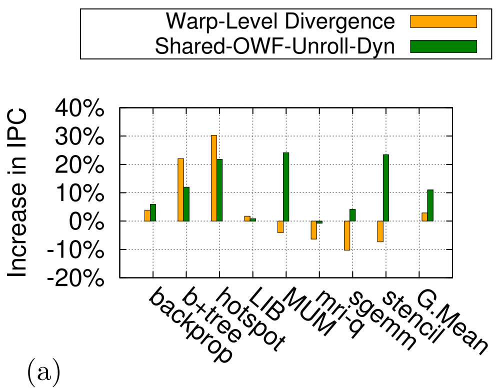
Table 2.8: Effect on Resident Thread Blocks with Scratchpad Sharing

% Sharing	0%	10%	30%	50%	70%	90%
CONV1	6	6	6	6	7	8
CONV2	3	3	3	3	3	4
lavaMD	2	2	2	2	2	4
NW1	7	7	7	8	8	8
NW2	7	7	7	8	8	8
SRAD1	2	2	2	3	4	4
SRAD2	3	3	3	3	3	5

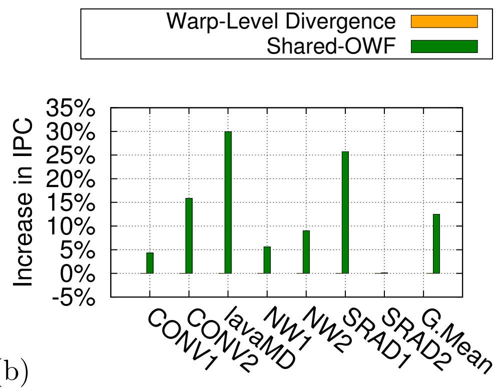
2.5.1.9 Performance Comparison with Warp-Level Divergence [92]

Figure 2.21 shows the performance comparison of resource sharing with the warp-level divergence [92] approach proposed by Xiang et al. We implemented their approach and evaluated on our experimental setup shown in Table 2.3. In Figure 2.21(a), we report the percentage increase in the IPC of applications with register sharing w.r.t *Unshared-LRR* and compare it with that of warp-level divergence. Similarly, we show the increase in IPC with scratchpad sharing w.r.t and *Unshared-LRR* and compare it with that of warp-level divergence in Figure 2.21(b).

From Figure 2.21(a), we observe that applications show an average improvement (shown as *G.Mean*) of 2.87% with warp-level divergence, whereas, with register sharing, they show an average improvement of 10.99%. Consider the applications *b+tree* and *hotspot*. They perform better with warp-level divergence than with register sharing. This behavior can be expected because the warps from the additional partial thread block launched by their approach can progress without waiting for



(a)



(b)

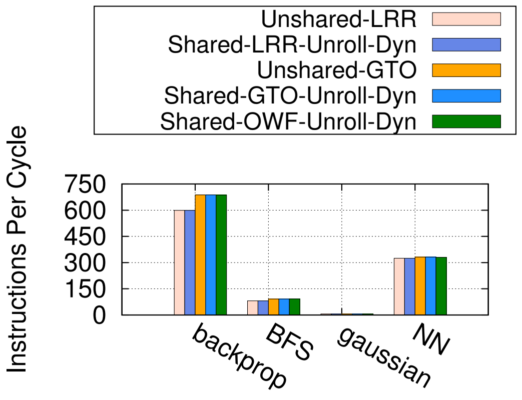
Figure 2.21: Performance comparison of (a) Register Sharing (b) Scratchpad Sharing with Warp-Level Divergence [92].

any other warps. Also, as discussed in Section 2.5.1.3, these applications benefit from the increase in the number of thread blocks without any further optimizations. However for *backprop, MUM, mri-q, sgemm*, and *stencil* applications, register sharing performs better than warp-level divergence. This is because register sharing not only increases the number additional thread blocks but also manages the their warps effectively using our optimizations. For *LIB*, register sharing is comparable to that of warp-level divergence.

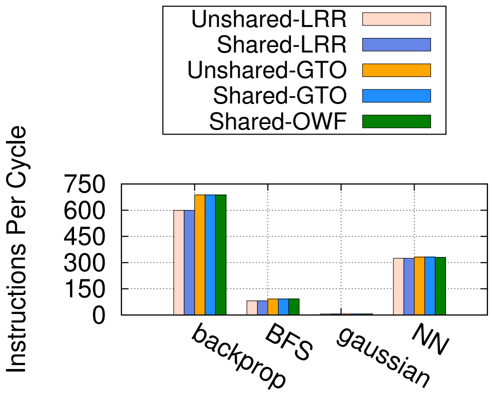
Figure 2.21(b) shows the performance comparison of scratchpad sharing and warp-level divergence. All the applications shown in the figure are limited by scratchpad memory. Hence, warp-level divergence can not increase the number of thread blocks/warps, consequently, it does not affect the performance of applications. However, with scratchpad sharing, we can launch additional thread blocks by exploiting unutilized scratchpad memory and improve the performance (on an average 12.47%).

2.5.2 Analyzing Benchmarks that are not Applicable to Resource Sharing

The performance of register sharing and scratchpad sharing approach for the Set-3 applications (Table 2.4) is presented in Figures 2.22(a) and (b) respectively. As discussed earlier, these applications are not limited by the number of available re-



(a)



(b)

Figure 2.22: Performance Analysis of Set-3 Applications for (a) Register Sharing (b) Scratchpad Sharing

sources but due to other factors such as the number of threads or thread blocks. We measure their performance when our approach uses (1) LRR scheduling policy, (2) GTO scheduling policy, and (3) OWF scheduling policy[1]. From Figures 2.22(a) and (b), we observe that our proposed resource sharing approach when used with LRR scheduling (labeled as Shared-LRR-Unroll-Dyn) performs exactly same as the baseline LRR scheduling (Unshared-LRR). Since the number of thread blocks launched by the applications are not limited by the resources, our approach does not launch any additional thread blocks, and all the thread blocks are in unsharing mode. Hence, it behaves exactly similar to the baseline approach. Similarly, our approach when used with the GTO scheduling policy (Shared-GTO-Unroll-Dyn), performs exactly same as the baseline approach that uses GTO scheduling policy without sharing (Unshared-GTO). Finally, we observe that with OWF scheduling policy (Shown as Shared-OWF-Unroll-Dyn), our approach is comparable to that of Unshared-GTO implementation. In OWF optimization, the warps are arranged according to the priorities of the owner, the unshared, and the non-owner warps. Since in this case, we do not launch any additional thread blocks, all the thread blocks are in unshared mode. Hence all the unshared warps are sorted according to their dynamic warp id. So the performance of Shared-OWF-Unroll-Dyn is similar to that of Unshared-GTO

[1]We do not use two-level scheduling policy because it cannot be directly integrated with our sharing approach.

implementation.

From the results of Set-1, Set-2, and Set-3 benchmark applications we can say that, if the number of thread blocks launched by an application is limited either by registers or by scratchpad memory (as shown in Set-1 and Set-2), then they can leverage our sharing approach to improve their performance. When they are limited either by the number of resident threads or by the number of thread blocks, our approach does not launch any additional thread blocks, and they perform comparable to the baseline approach.

2.6 Summary

This chapter proposed sharing of some resources of SM to minimize their wastage by launching additional thread blocks in each SM. For effective utilization of these additional thread blocks, we proposed optimizations which further help in reducing the stalls produced in the system. We validated our approach for register sharing and for scratchpad sharing on several applications that underutilize register and scratchpad memory and showed improvements up to maximum 24% and average 11% with register sharing, and maximum 30% and average 12.5% with scratchpad sharing. For the remaining applications, our approach performs comparable to the baseline approach.

Chapter 3

Improving Scratchpad Sharing with Compiler Optimizations

3.1 Introduction

In Chapter 2, we presented the resource sharing approach which improves the GPU performance by increasing the TLP. It launches additional thread blocks in each SM that use wasted resources of the SM and share some resources with other resident thread blocks in the SM. However, note that, the performance achieved by the resource sharing technique is limited by the amount of the TLP that is exhibited the additional thread blocks. In other words, the performance gain depends on the progress made by the shared thread blocks (non-owner) while accessing the unshared resource without waiting for shared resource. A non-owner shared thread block can make more progress if the shared resources are held for minimum time. Hence, to improve the TLP, it is important to allocate the resources into shared and unshared parts such that access to the shared resource has a short duration.

Moreover in GPUs, the resources held by a thread block are released only after the thread block finishes its execution. As a result, when a shared thread block acquires a shared resource, it is released only after the shared thread block finishes its execution. This limits the amount of thread-level-parallelism that is exhibited a non-owner thread block because it can access the shared resource only after its owner shared thread block has finished its execution even though the shared resources need

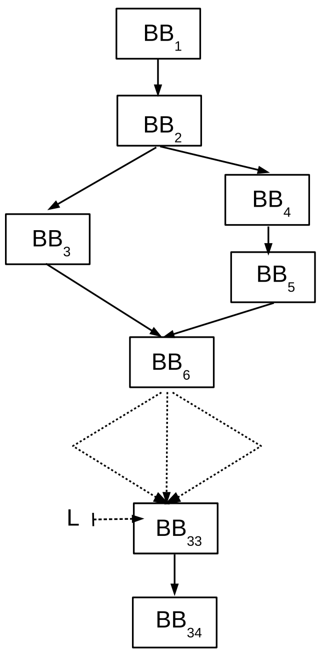


Figure 3.1: Release of Shared Scratchpad

not be accessed till towards the end of execution. This limitation is observed more in case of scratchpad memory, where a thread block may not access scratchpad memory till the end of execution.

We explain these limitations with suitable examples in the context of scratchpad memory and motivate the need for compile time optimizations to improve the TLP further.

3.1.1 The Need for Compiler Optimizations

In scratchpad sharing, when two thread blocks (say, TB_0 and TB_1) are launched in shared mode, one of them accesses the shared scratchpad region at a time. As soon as one thread block, say TB_0, starts accessing the shared scratchpad region, the other thread block, TB_1, can not access the shared scratchpad region and hence may have wait until TB_0 finishes execution.

Example 3.1.1. *Consider the CFG in Figure 3.1, which is obtained for* SRAD1 *application [19]. In the figure, the program point marked* **L** *corresponds to the last access to the shared scratchpad. In this case, the shared scratchpad region will be*

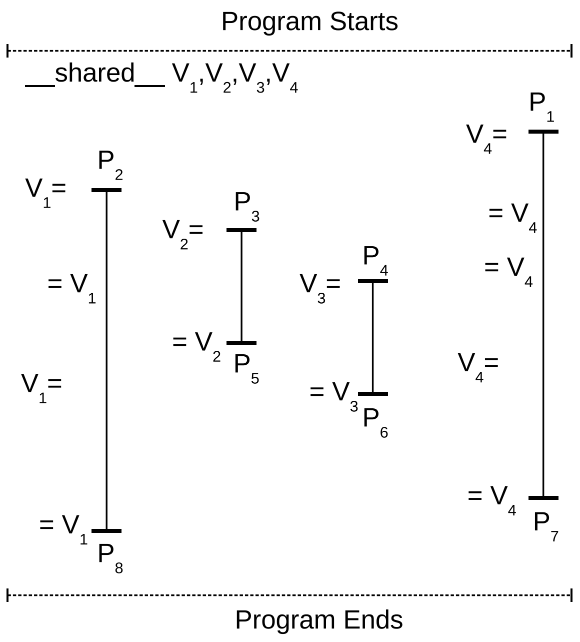


Figure 3.2: Access Ranges of Scratchpad Variables

*released only at the end of the last basic block (*Exit *node of CFG) even though it is never accessed after L.* □

Further in scratchpad sharing, the allocation of scratchpad variables into shared and unshared scratchpad can affect the availability of the shared scratchpad region, and hence the effectiveness of sharing.

Example 3.1.2. *Consider the scenario in Figure 3.2, where a kernel function declares four equal sized scratchpad variables V_1 to V_4. The figure also shows the regions of the kernel within which different variables are accessed. If V_1 and V_4 are allocated into shared scratchpad region, then the shared scratchpad region is accessed from program point P_1 to program point P_8. However, when V_2 and V_3 are allocated to shared scratchpad region, the shared region is accessed for a shorter duration, i.e., from program point P_3 to program point P_6.* □

Note that the choice of allocation of scratchpad variables does not affect the correctness of the program.

3.1.2 Contributions

To improve the availability of shared scratchpad memory, we have developed static analysis that helps in allocating scratchpad variables into shared and unshared scratchpad regions such that the shared scratchpad variables are needed only for a short duration. To promote the release of shared scratchpad region before the end of kernel execution, we introduce a new hardware instruction (PTX instruction) called *relssp*. It releases the shared scratchpad memory at run-time when all the threads of a thread block have finished accessing shared scratchpad memory. We describe an algorithm to help compiler in an optimal placement of the *relssp* instruction in a kernel such that the shared scratchpad can be released as early as possible, without causing any conflicts among shared thread blocks. These optimizations improve the availability of shared scratchpad memory.

To summarize, we make the following contributions in this chapter:

1. We present a static analysis to layout scratchpad variables in order to minimize the shared scratchpad region.
2. We introduce a hardware instruction, *relssp*, and an algorithm for optimal placement of *relssp* in the user code to release the shared scratchpad region at the earliest.
3. We used the GPGPU-Sim [14] simulator and the Ocelot [21] compiler framework to implement and evaluate our ideas. On several kernels from various benchmark suites, we achieved an average improvement of 19% and a maximum improvement of 92.17% (with scratchpad sharing) over the baseline approach.

In the rest of the chapter, Section 3.2 provides the details of compiler optimizations. Section 3.3 analyzes the complexity of our approach. Section 3.4 shows the experimental results, and Section 3.5 summarizes the chapter.

3.2 Compiler Optimizations

In this section we describe a compile time memory allocation scheme and an analysis to optimally place *relssp* instructions. The memory allocation scheme allocates scratchpad variables into shared and unshared region such that shared scratchpad variables are accessed only for a small duration during the run-time.

Computing the optimal minimum shared scratchpad region at compile-time is hard. This is due to several unknown factors such as (1) number of iterations of the loop, (2) program control flow, (3) memory access patterns; cache misses, and (4) optimal scheduling policies at various stage of the pipeline etc. However, we can make approximations to reduce to access to shared scratchpad region.

In the thesis, the metric we use to reduce the shared scratchpad region is the number of instructions present in the shared region. This depends on (1) the number of iterations of loops (2) control flow nature of the program. Since it is not possible to statically bound the number of instructions executed at run-time, we approximate loop bounds. Also, in case of branch diverging points, we consider the path that contains maximum number of instructions in it. Any approximation is safe since, as noted earlier, it only affects the effectiveness of sharing, but not the correctness[1].

To simplify the description of the required analyses, we make the following assumptions:

- The control flow graph (CFG) for a function (kernel) has a unique Entry and a unique Exit node.
- There are no *critical* edges in the CFG. A critical edge is an edge whose source node has more than one successor and the destination node has more than one predecessor. The absence of critical edges is required for optimal insertion of relssp instruction (discussed in Section 3.2.3).

These assumptions are not restrictive as any control flow graph can be converted to the desired form using a preprocessing step involving simple graph transformations: adding a source node, adding a sink node, and adding a node to split an edge [43, 47, 74].

3.2.1 Minimizing Shared Scratchpad Region

Consider a GPU that uses scratchpad sharing approach such that two thread blocks involved in sharing can share a fraction $f < 1$ of scratchpad memory. Assume

[1]Profiling and user annotations can help in finding better approximations for the loop bounds. However, we have not used these in our current implementation.

that each SM in the GPU has M bytes of scratchpad memory, the kernel that is to be launched into the SM has N scratchpad variables[1], and each thread block of the kernel requires M_{tb} bytes of scratchpad memory. We allocate a subset S of scratchpad variables into shared scratchpad region such that:

(1) The total size of the scratchpad variables in the set S is equal to the size of shared scratchpad ($f \times M_{tb}$), and

(2) The region of access for variables in S is minimal in terms of the number of instructions.

To compute the region of access for S, we define *access range* for a variable as follows:

Definition 3.2.1. Access Range of a Variable: *A program point π is in the* access range *of a variable v if both the following conditions hold: (1) There is an access (definition or use) of v on some path from* Entry *to π* and *(2) There is an access of v on some path from π to* Exit.

Intuitively, the access range of a variable covers every program point between the first access and the last access of the variable in an execution path. The access range for a variable can contain disjoint regions due to branches in the flow graph.

Definition 3.2.2. Access Range of a Set of Variables: *A program point π is in the* access range *of a set of variable S if both the following conditions hold: (1) There is an access to a variable $v \in S$ on a path from* Entry *to π* and *(2) There is an access to a variable $v' \in S$ on a path from π to* Exit.

Example 3.2.1. *Consider a kernel whose CFG is shown in Figure 3.3. The kernel uses 3 scratchpad variables A, B and C. Variable A is accessed in the region from basic block BB_1 to basic block BB_4. The start of basic block BB_2 is considered in access range of A because there is a path from* Entry *to start of BB_2 that contains an access of A (definition in BB_1) and there is a path from the start of BB_2 to* Exit *that contains the access of A (use in BB_4).*

[1]In this thesis, we only address the scratchpad variables whose memory is allocated at compile time (like variables declared with primitive data types, arrays, but not pointers).

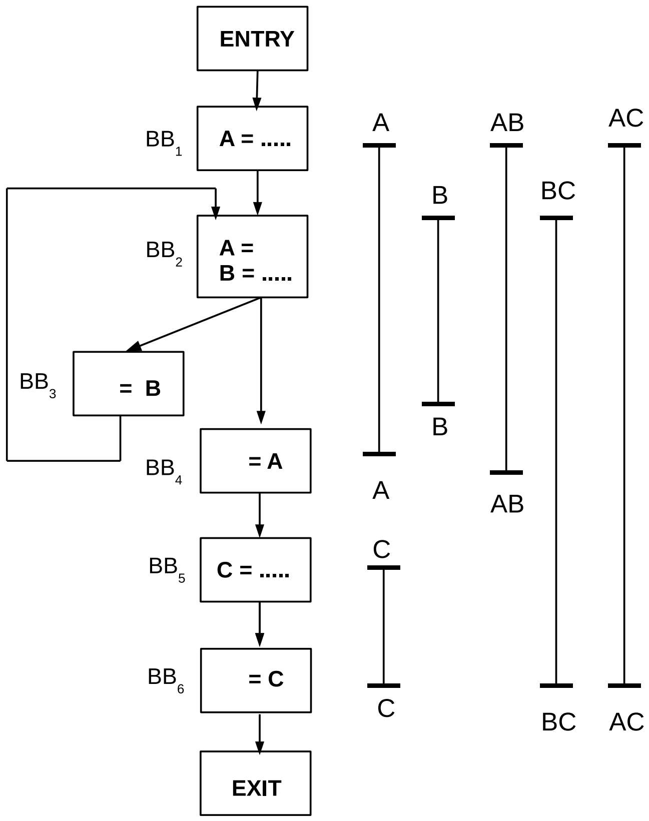


Figure 3.3: Access Ranges of Variables

Consider the set $S = \{B, C\}$. Basic block BB_4 is in access range of S because there is a path from Entry *to BB_4 containing the access of B (definition in BB_2), and there is a path from BB_4 to* Exit *containing the access of the C (use in BB_6).* □

To compute the access ranges for a program, we need a forward analysis to find the first access of the scratchpad variables, and a backward analysis to find the last access of the scratchpad variables. We define these analyses formally using the following notations:

- IN(BB) denotes the program point before the first statement of the basic block BB. OUT(BB) denotes the program point after the last statement of BB.
- PRED(BB) denotes the set of predecessors, and SUCC(BB) denotes the set of successors of BB.

- $\mathsf{PreIN}(v, BB)$ is true if there is an access to variable v before $\mathsf{IN}(BB)$. $\mathsf{PreOUT}(v, BB)$ is true if there is an access to the variable v before $\mathsf{OUT}(BB)$.
- $\mathsf{PostIN}(v, BB)$ is true if there is an access to variable v after $\mathsf{IN}(BB)$. $\mathsf{PostOUT}(v, BB)$ is true if there is a access to variable v after $\mathsf{OUT}(BB)$.
- $\mathsf{AccIN}(S, BB)$ is true if $\mathsf{IN}(BB)$ is in access range of a set of scratchpad variables S. $\mathsf{AccOUT}(S, BB)$ is true if $\mathsf{OUT}(BB)$ is in access range of a set of scratchpad variables S.

The data flow equations to compute the information are:

$$\mathsf{PreOUT}(v, BB) = \begin{cases} \text{true, if } BB \text{ has an access of } v \\ \mathsf{PreIN}(v, BB), \text{ otherwise} \end{cases}$$

$$\mathsf{PreIN}(v, BB) = \begin{cases} \text{false, if } BB \text{ is } \mathsf{Entry} \text{ block} \\ \bigvee_{BP \in \mathsf{PRED}(BB)} \mathsf{PreOUT}(v, BP), \text{ otherwise} \end{cases}$$

$$\mathsf{PostIN}(v, BB) = \begin{cases} \text{true, if } BB \text{ has an access of } v \\ \mathsf{PostOUT}(v, BB), \text{ otherwise} \end{cases}$$

$$\mathsf{PostOUT}(v, BB) = \begin{cases} \text{false, if } BB \text{ is } \mathsf{Exit} \text{ block} \\ \bigvee_{BS \in \mathsf{SUCC}(BB)} \mathsf{PostIN}(v, BS), \text{ otherwise} \end{cases}$$

We decide whether the access range of a set of scratchpad variables S includes the points $\mathsf{IN}(BB)$ and $\mathsf{OUT}(BB)$ as:

$$\mathsf{AccIN}(S, BB) = \Big(\bigvee_{v \in S} \mathsf{PreIN}(v, BB)\Big) \bigwedge \Big(\bigvee_{v \in S} \mathsf{PostIN}(v, BB)\Big)$$
$$\mathsf{AccOUT}(S, BB) = \Big(\bigvee_{v \in S} \mathsf{PreOUT}(v, BB)\Big) \bigwedge \Big(\bigvee_{v \in S} \mathsf{PostOUT}(v, BB)\Big)$$

The above analysis can be extended easily to compute information at any point inside a basic block.

Example 3.2.2. *Table 3.1 shows the program points in the access ranges of scratchpad variables for CFG of Figure 3.3. The table also shows the program points in the access ranges of sets of two scratchpad variables each.* □

Table 3.1: Access Ranges for Scratchpad Variables and Sets.

t denotes true, **f** denotes false. Sets of variables are written as concatenation of variables. For example, AB denotes {A, B}.

	For Variables						For Sets of Variables					
	IN			OUT			IN			OUT		
Block	A	B	C	A	B	C	AB	BC	CA	AB	BC	CA
Entry	f	f	f	f	f	f	f	f	f	f	f	f
BB1	f	f	f	t	f	f	f	f	f	t	f	t
BB2	t	t	f	t	t	f	t	t	t	t	t	t
BB3	t	t	f	t	t	f	t	t	t	t	t	t
BB4	t	f	f	f	f	f	t	t	t	f	t	t
BB5	f	f	f	f	f	t	f	t	t	f	t	t
BB6	f	f	t	f	f	f	f	t	t	f	f	f
Exit	f	f	f	f	f	f	f	f	f	f	f	f

Let SV denote the set of all scratchpad variables. For every subset S of SV having a total size equal to the size of shared scratchpad memory, our analysis counts the total number of instructions in the access range of S. Finally the subset that has the minimum count is selected for allocation in the shared scratchpad memory.

Example 3.2.3. *Consider once again the CFG in Figure 3.3. For simplicity, assume that all the variables have equal sizes, and each basic block contains the same number of instructions. Consider a scratchpad sharing approach that can allocate only two of the variables into the shared scratchpad region. From the CFG, and from Table 3.1, it is clear that when A and B are allocated into shared scratchpad memory, the shared region is smaller, compared to when either* $\{B,C\}$ *or* $\{A,C\}$ *are allocated in the shared region.* □

3.2.2 Implementation of *relssp* Instruction

In scratchpad sharing approach (Section 2.2.2), a shared thread block acquires a lock before accessing shared scratchpad region and unlocks it only after finishing its execution. This causes a delay in releasing the shared scratchpad because the thread block holds the scratchpad memory till the end of its execution, even if it has finished accessing shared region.

```
1 void ReleaseSSP()
2 {
3     static int count=0;
4     ++count;
5     if(count==ACTIVE_THREADS){
6         count=0;
7         UnlockSharedRegion();
8     }
9 }
```

Figure 3.4: Pseudocode of *relssp* Instruction

To minimize the delay in releasing the shared scratchpad, we propose a new instruction, called *relssp*, in PTX assembly language. The semantics of *relssp* instruction is to unlock the shared region only when all active threads within a thread block finished executing the shared region. Here, we use the notion of active threads according to GPGPU-Sim [3] implementation. In GPGPU-Sim, additional threads are padded to a thread block to make the number of threads in the thread block a multiple of the warp size. These additional threads do not execute kernel instructions and are considered to be *inactive*, and the remaining threads are considered to be *active*.

Figure 3.4 shows the pseudo code for *relssp* instruction. The ReleaseSSP() procedure maintains **count**, an integer initialized to zero. When an active thread within a thread block executes a *relssp* instruction, it increments the **count** value. When all active threads of a thread block execute *relssp* instruction (Line 5, when **count** equals `ACTIVE_THREADS`), the shared region is unlocked by invoking UnlockSharedRegion(). The unlock procedure releases the shared scratchpad region by resetting the lock variable. The execution of *relssp* by a thread block that does not access shared scratchpad region has no effect.

It is clear that **count** in Figure 3.4 has to be a shared variable, hence a software implementation will require to manage critical section. The same algorithm, however, can be efficiently implemented in a hardware circuit as shown in Figure 3.5. The i^{th} thread within a thread block is associated with an active mask (A_i) and a release bit (R_i). The mask A_i is set if the i^{th} thread is active. When this thread executes *relssp* instruction, the release bit (R_i) gets set. The shared scratchpad

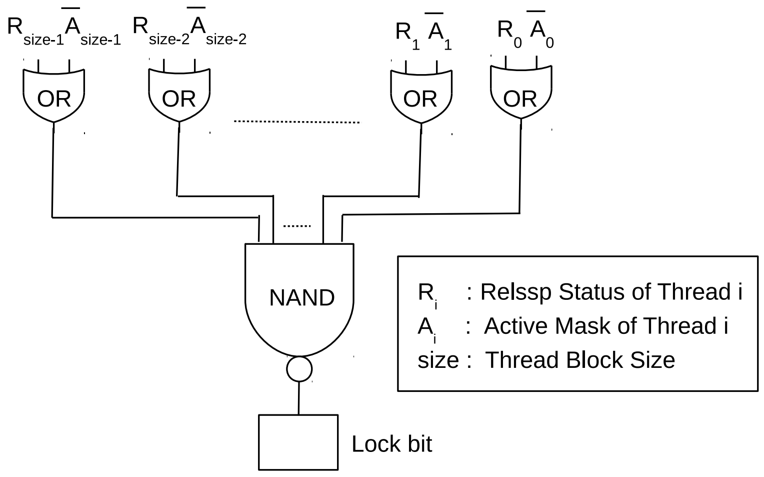


Figure 3.5: Hardware Implementation of *relssp* Instruction

region is unlocked only when all the active threads in a thread block execute *relssp* instruction (the lock bit, i. e. the output of NAND gate becomes 0 in Figure 3.5). In other words, shared scratchpad region is unlocked if $\forall i \; A_i \rightarrow R_i$ is true.

3.2.3 Algorithm for Optimal Placement of *relssp* Instruction

In Section 3.2.2, we introduced a new instruction to release the shared scratchpad memory. In this section, we discuss a compile-time analysis for optimal insertion of *relssp* instruction in the program. We insert a *relssp* instruction at a program point π such that the following conditions are met:

1. **Safety:** The *relssp* instruction must be executed by each active thread within a thread block, and it must be executed after last access to shared scratchpad memory.

2. **Optimality:** The *relssp* instruction must be executed by each active thread exactly once.

Condition (1) (Safety) ensures that shared scratchpad is eventually released by a thread block since the instruction is executed by all the threads of a thread block. Also, it guarantees that shared scratchpad is released only after a thread block has completed using it. Whereas, Condition (2) (Optimality) avoids redundant execution of *relssp* instruction.

In the scratchpad sharing, a thread block releases the shared scratchpad memory after completing its execution, hence it is equivalent to having a *relssp* instruction placed at the end of the program, which guarantees both the conditions, albeit at the cost of delay in releasing the shared scratchpad. A simple improvement that promotes early release of shared scratchpad memory and ensures both the conditions, is to place the *relssp* instruction at a basic block $BB_{postdom}$ where $BB_{postdom}$ is a common post dominator of those basic blocks having the last accesses to the shared scratchpad memory along different paths. Further, $BB_{postdom}$ should dominate Exit, i.e., it should be executed in all possible execution paths. As the following example shows, this strategy, though an improvement over placing *relssp* in Exit, may also result in delaying the release of shared scratchpad memory.

Example 3.2.4. *Consider a CFG shown in Figure 3.6. Assume that L_1, L_2 denote the program points that correspond to the last accesses to shared scratchpad memory. Since relssp instruction is to be executed by all the threads of thread block, it can be placed at the post dominator of the basic blocks BB_3 and BB_9, i.e., program point marked π in BB_{12}, which is visible to all threads. However, this delays the release of shared scratchpad.*

Consider a thread that takes a path along BB_9, it can execute relssp immediately after executing the last access to shared region (shown as OPT_3 in the figure). It executes relssp at program point π. Similarly, when a thread takes a path along the basic block BB_4, it releases the shared scratchpad at π even though it does not access any shared scratchpad in that path. The scratchpad can be released at program point OPT_2 in BB_4. □

As is clear from the above example, placement of *relssp* instruction has an effect on the availability of shared scratchpad memory. Intuitively, a safely placed *relssp* instruction at a program point π can be moved to a previous program point π' in the same basic block provided the intervening instructions do not access shared scratchpad. The movement of *relssp* from a basic block BB to predecessor BB' is possible provided every other successor of BB' also does so.

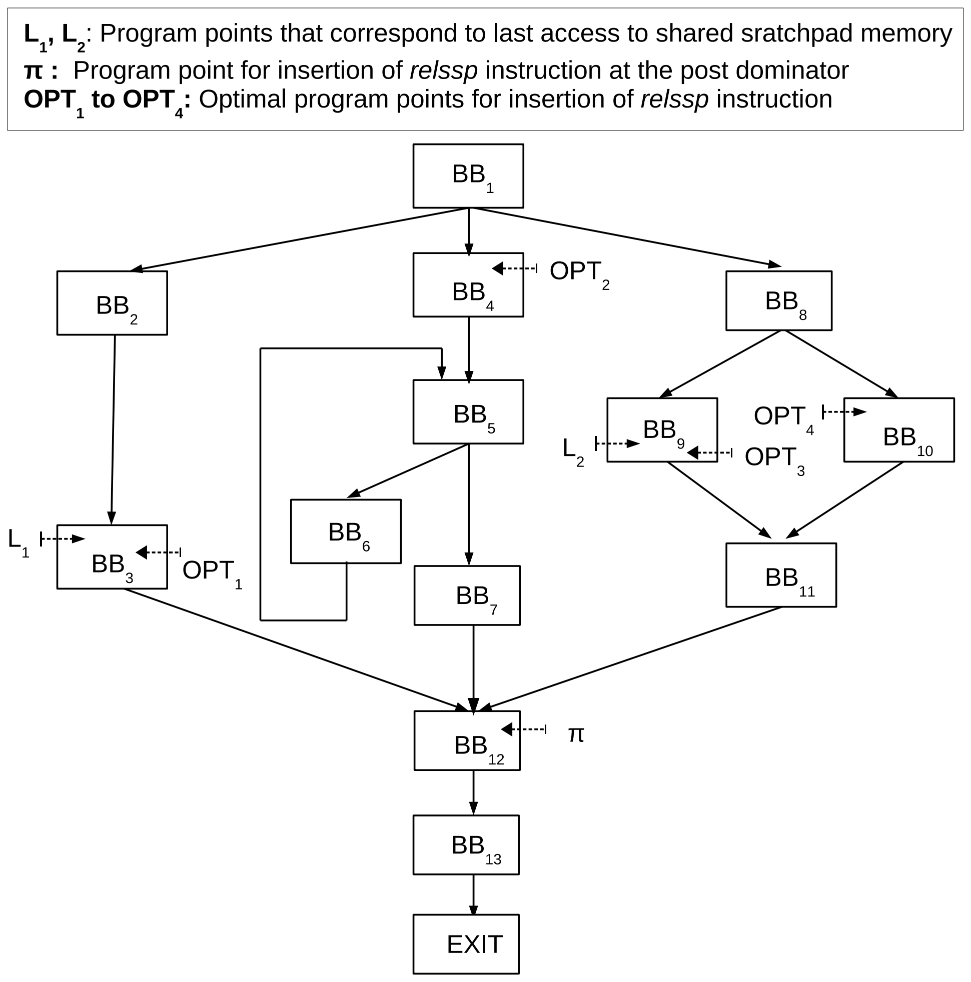

L1, L2: Program points that correspond to last access to shared sratchpad memory
π : Program point for insertion of relssp instruction at the post dominator
OPT1 to OPT4: Optimal program points for insertion of relssp instruction
BB1
BB2
BB4
OPT2
BB8
BB5
BB9
OPT4
BB10
L2
OPT3
L1
BB3
OPT1
BB6
BB7
BB11
BB12
π
BB13
EXIT

Figure 3.6: Possible Insertion Points for *relssp*

Example 3.2.5. *Figure 3.7(a) shows a basic block BB_1, which has the last access to the shared scratchpad memory at L_1. In this block, if the relssp instruction can be placed safely at the program point π_1, then it can be moved to π_2 since there is no access to shared scratchpad memory between π_1 and π_2. However, it can not be moved to the program point π_3 within the same basic block, because it violates safety (Condition 1).*

Consider another scenario shown in Figure 3.7(b), basic block BB_2 has the last access to shared memory at L_2, and basic blocks BB_1, BB_3, and BB_4 do not access any scratchpad memory. If the relssp instruction can be placed safely at π_4 in BB_4, then it can be moved to a program point π_5 and π_7 in the basic blocks BB_2 and BB_3

L_1, L_2: Program points that correspond to last access to shared scratchpad memory
Y: *relssp* can be placed safely
N: *relssp* can not be placed safely

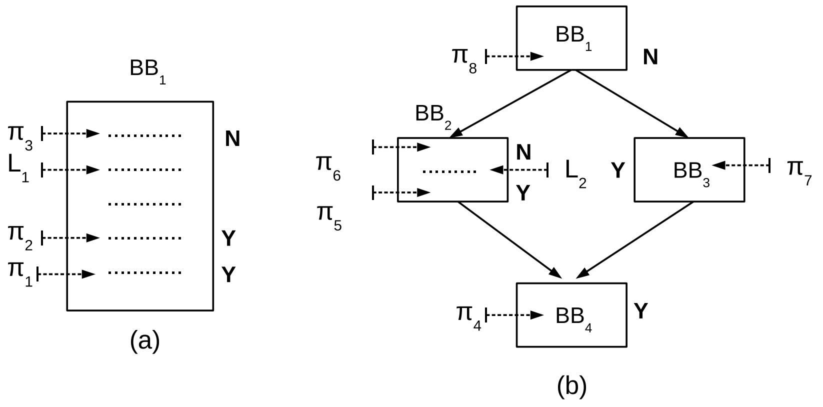


Figure 3.7: Scenarios for Optimal Insertion of *relssp*

respectively. However, it can not be moved to program point π_6 in BB_2 and π_8 in BB_1 since it violates of Condition 1. Also, the relssp instruction can not be moved from π_7 in BB_3 to π_8 in BB_1 since the basic block BB_2, which is a successor of BB_1, does not allow the relssp instruction to be placed at π_8. □

We now formalize these intuitions into a backward data flow analysis. The notations used are:

- $\mathsf{IN}(BB)$ denotes the program point before the first statement of the basic block BB. $\mathsf{OUT}(BB)$ denotes the program point after the last statement of BB.
- $\mathsf{SafeIN}(BB)$ is true if the *relssp* instruction can be safely placed at $\mathsf{IN}(BB)$, and $\mathsf{SafeOUT}(BB)$ is true if the *relssp* instruction can be safely placed at $\mathsf{OUT}(BB)$.
- INS_{π}, if true, denotes that *relssp* will be placed at program point π by the analysis.

The data flow equations are:

$$\mathsf{SafeIN}(BB) = \begin{cases} \text{false, if } BB \text{ has shared scratchpad access} \\ \mathsf{SafeOUT}(BB), \text{ otherwise} \end{cases}$$

$$\mathsf{SafeOUT}(BB) = \begin{cases} \text{true, if } BB \text{ is } \mathsf{Exit} \text{ block} \\ \bigwedge\limits_{BS \in \mathsf{SUCC}(BB)} \mathsf{SafeIN}(BS), \text{ otherwise} \end{cases}$$

The above equations compute the program points where *relssp* can be placed safely. For a basic block BB, $\mathsf{OUT}(BB)$ is an optimal place for *relssp* instruction, if *relssp* can be placed safely at $\mathsf{OUT}(BB)$, and it can not be moved safely to its previous program point in the basic block, i.e., $\mathsf{IN}(BB)$ is false. This is computed as:

$$\mathsf{INS}_{\mathsf{OUT}(BB)} = \mathsf{SafeOUT}(BB) \wedge \neg(\mathsf{SafeIN}(BB)) \tag{3.1}$$

Similarly, $\mathsf{IN}(BB)$ is an optimal point for *relssp* instruction, when the instruction can not be moved to its predecessors (absence of critical edges guarantees that the instruction can either be moved to *all* predecessors or to *none*). This can be computed as:

$$\mathsf{INS}_{\mathsf{IN}(BB)} = \neg\left(\bigwedge_{BP \in \mathsf{PRED}(BB)} \mathsf{SafeOUT}(BP)\right) \wedge \mathsf{SafeIN}(BB) \tag{3.2}$$

Equations (3.1) and (3.2) together, along with the absence of critical edges, ensure the optimality condition that each thread executes the *relssp* instruction exactly once. Thus, our compiler optimizations help in progressing the shared thread blocks that are present in each SM by releasing shared scratchpad and minimizing the access to shared scratchpad region. Hence the shared thread blocks in each SM finish their execution early. Subsequently new shared thread blocks are launched early. Thus a kernel can finish its execution early.

3.3 Analysis of Compiler Optimizations

The dataflow analyses to compute definitions and usages of scratchpad variables (Section 3.2.1) are bit-vector data flow analyses [47]. For a kernel with n scratchpad variables and m nodes (basic blocks) in the flow graph, the worst case complexity is $O(n \times m^2)$ (assuming set operations on n bit-wide vectors take $O(n)$ time).

Table 3.2: GPGPU-Sim Architecture

Resource	Configuration
Number of SMs	14
Scratchpad Memory per SM	16KB
Number of Registers (32 bit) per SM	65536
Max Number of TBs per SM	16
Max Number of Threads per SM	3072
Warp Scheduling	LRR
L1-Cache per SM	16KB
DRAM Scheduler	FR-FCFS

The computation of access ranges for sets of variables may require analyzing all $O(2^n)$ subsets in the worst case, where the largest size of a subset is $O(n)$. Thus, given the usage and definitions at each program point in the kernel, computation of AccIN and AccOUT requires $O(m \times n \times 2^n)$ time. Therefore, the total time complexity is $O(n \times m^2 + m \times n \times 2^n)$. Since the number of scratchpad variables in a kernel function is small (typically, $n \leq 10$), the overhead of the analysis is practical.

Our approach inserts *relssp* instructions in a CFG such that *relssp* is called exactly once along any execution path. In the worst case, all nodes in a CFG (except Entry and Exit blocks) might fall along different paths from Entry to Exit. Hence the worst case number of *relssp* inserted is $O(m)$.

3.4 Experimental Evaluation

We integrated the proposed *relssp* instruction in GPGPU-Sim V3.x [3] simulator. We implemented the compiler optimizations in PTX assembly [6] using Ocelot [21] framework. The baseline architecture that we used for comparing our approach is shown in Table 3.2. Note that, to evaluate our approach on more number of kernels, we increased the number of resident thread blocks, threads, and register file size in each SM in Table 3.2 as compared to the GPU configuration in Table 2.3 (Chapter 2). We evaluated our approach on several kernels from CUDA-SDK [2], GPGPU-Sim [14], and Rodinia [19] benchmark suites.

Depending on the amount and the last usage of the shared scratchpad memory by the applications, we divided the benchmark applications into three sets. Set-1 and

Table 3.3: Benchmark Applications for which the Number of Thread Blocks is Limited by Scratchpad Memory

	Benchmark	Application	Kernel	#Scratchpad Variables	Scratchpad Size (Bytes)	Block Size
			Set-1: Shared scratchpad can be released before the end of the kernel			
1.	RODINIA	backprop	bpnn_layerforward_CUDA	2	9408	256
2.	CUDA-SDK	dct8x8_1 (DCT1)	CUDAkernel2DCT	1	2112	64
3.	CUDA-SDK	dct8x8_2 (DCT2)	CUDAkernel2IDCT	1	2112	64
4.	CUDA-SDK	dct8x8_3 (DCT3)	CUDAkernelShortDCT	1	2176	128
5.	CUDA-SDK	dct8x8_4 (DCT4)	CUDAkernelShortIDCT	1	2176	128
6.	GPGPU-SIM	NQU	solve_nqueen_cuda_kernel	5	10496	64
7.	RODINIA	srad_v2_1 (SRAD1)	srad_cuda_1	6	13824	576
8.	RODINIA	srad_v2_2 (SRAD2)	srad_cuda_2	5	11520	576
			Set-2: Shared scratchpad can not be released before the end of the kernel			
9.	CUDA-SDK	FDTD3d	FiniteDifferencesKernel	1	3840	128
10.	RODINIA	heartwall	kernel	8	11872	128
11.	CUDA-SDK	histogram	histogram256Kernel	1	9216	192
12.	CUDA-SDK	marchingCubes (MC1)	generateTriangles	2	9216	32
13.	RODINIA	NW1	needle_cuda_shared_1	2	8452	32
14.	RODINIA	NW2	needle_cuda_shared_2	2	8452	32

Table 3.4: Set-3 Benchmarks: The Number of Thread Blocks is Not Limited by Scratchpad Memory

Benchmark	Application	Kernel	Limited by
GPGPU-SIM	BFS	Kernel	Threads, Registers
RODINIA	b+tree	findRangeK	Registers
CUDA-SDK	dct8x8_5 (DCT5)	CUDAkernel1DCT	Blocks
RODINIA	gaussian	FAN1	Threads
GPGPU-SIM	NN	executeSecondLayer	Blocks

Set-2 (Table 3.3) consists of applications whose number of resident thread blocks are limited by scratchpad memory. For Set-1, the applications do not access scratchpad memory till towards the end of their execution, while for Set-2, the applications access scratchpad memory till towards the end of their execution. The introduction of *relssp* instruction is expected to give benefit over the scratchpad sharing approach (discussed in Chapter 2) only for Set-1 applications. Set-3 benchmarks (Table 3.4) consist of applications whose number of thread blocks are not limited by scratchpad memory, but by some other parameter. These are included to show that our approach does not negatively affect the performance of applications that are not limited by scratchpad memory.

For each application in Set-1 and Set-2 benchmarks, Table 3.3 shows the kernel that is used for evaluation, the number of the scratchpad variables declared in each kernel, the amount of the scratchpad memory required for each thread block, and the thread block size. Some applications in Set-1 and Set-2 benchmarks are modified to make sure that the number of thread blocks is limited by scratchpad memory, thus making scratchpad sharing approach applicable. These changes increase the scratchpad memory requirement per thread block. For Set-3 benchmarks, Table 3.4 shows the cause of limitation on the number of thread blocks. The causes include the limit on the number of registers, the maximum limit on the number of resident thread blocks, and the maximum limit on the number of resident threads.

We compiled all the applications using CUDA 4.0 and simulated using the GPGPU-Sim simulator. We simulated all the applications using their PTX representation since (1) the compiler optimizations deal with only scratchpad memory and not the registers (2) the chapter requires more static analysis features, which are not currently supported by GPGPU-Sim, but they are supported in Ocelot [21] (a widely used framework for PTX transformations).

CUDA 4.0 is used since GPGPU-Sim and Ocelot do not support CUDA 5.0 and above. We configured the threshold (t) value to 0.1. For any loop with non-constant bounds on the number of iterations of the loop, we assumed a bound of 1024.

We measure the performance of our approach using the following metrics:

1. The **number of the resident thread blocks** launched in the SMs. This is a measure of the amount of thread level parallelism present in the SMs.

2. The **number of instructions executed per shader core clock cycle (IPC)**. This is a measure of the throughput of the GPU architecture.

3. The **number of simulation cycles** that an application takes to complete its execution. This is a measure of the performance of the benchmark applications.

3.4.1 Analyzing Benchmarks that are Limited by Scratchpad Memory

We use *Unshared-LRR* to denote the baseline unsharing approach, *Shared-OWF* to denote our scratchpad sharing approach with OWF scheduler (as described in Chapter 2), and *Shared-OWF-OPT* to denote the scratchpad sharing approach that includes OWF scheduler and compiler optimizations.

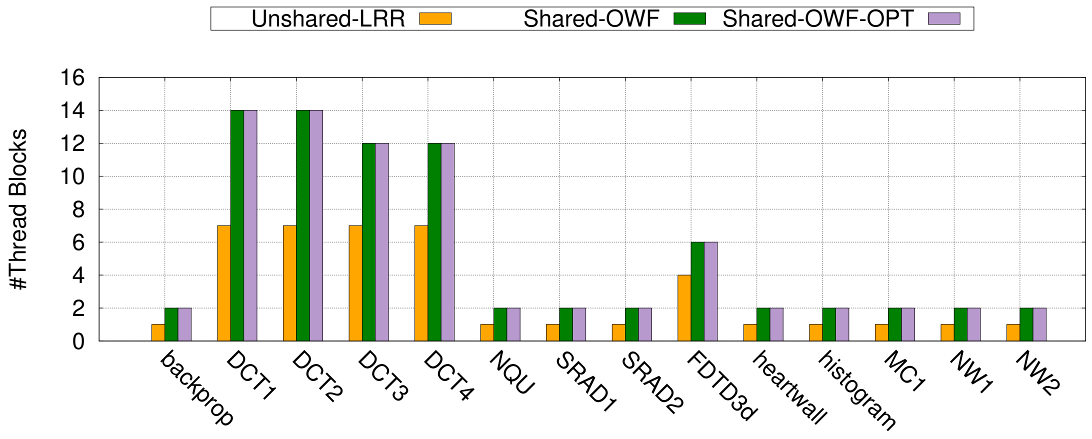


Figure 3.8: Comparing the Resident Thread Blocks

3.4.1.1 Comparing the Number of Resident Thread Blocks

Figure 3.8 shows the number of thread blocks for the three approaches. For applications *DCT1* and *DCT2*, *Unshared-LRR* launches 7 thread blocks in the SM according to the amount of scratchpad memory required by their thread blocks. *Shared-OWF* launches 14 thread blocks in the SM, where each of the 7 additional thread blocks share scratchpad memory with other resident thread blocks. For *DCT3* and *DCT4* applications, *Unshared-LRR* launches 7 thread blocks in the SM, whereas *Shared-OWF* launches 12 thread blocks in the SM such that the additional 5 thread blocks share scratchpad memory with the existing 5 thread blocks; while the remaining 2 existing thread blocks in the SM do not share scratchpad memory with any other thread block. For *FDTD3d*, *Shared-OWF* launches 2 additional thread blocks in the SM when compared to *Unshared-LRR*, which share scratchpad memory with other 2 resident thread blocks. For the remaining applications, *Unshared-LRR* launches 1 thread block, whereas *Shared-OWF* launches 1 additional thread block in the SM which shares scratchpad memory with the existing thread block. Note that the number of thread blocks launched by *Shared-OWF-OPT* is exactly same as that of *Shared-OWF*. This is expected since the number of additional thread blocks launched by scratchpad sharing approach depends on two parameters: (1) the amount of scratchpad sharing, and (2) the amount of scratchpad memory required by a thread block; and our compiler optimizations do not affect either of these parameters.

3.4.1.2 Performance Comparison

Figure 3.9 shows the performance of *Shared-OWF* and *Shared-OWF-OPT*[1] in terms of the number of instructions executed per cycle (IPC) when compared to *Unshared-LRR*. We also present the details of individual optimizations in Section 3.4.1.4. From the figure, we observe a maximum improvement of 92.17% and an average (Geometric Mean, shown as *G. Mean* in the figure) improvement of 19% with *Shared-OWF-OPT* when compared to *Unshared-LRR*. The figure shows that most of the applications benefit with our compiler optimizations. *backprop* shows an improvement of 74.2%, it leverages both scratchpad sharing and the compiler optimizations to perform better. *heartwall* achieves 92.17%, because the additional thread blocks launched by *Shared-OWF-OPT* do not access the shared scratchpad region. Hence all the additional thread blocks make progress without waiting for corresponding shared thread blocks. *MC1* improves by 32.32% because additional thread blocks launched in the SM make significant progress before accessing shared scratchpad region. The improvements in *SRAD1* and *SRAD2* applications are largely due to the compiler optimizations. *FDTD3d* slows down (–2.29%) with *Shared-OWF-OPT* due to more number of L1 and L2 cache misses when compared to *Unshared-LRR*. *histogram* does not benefit from sharing since the thread blocks start accessing shared scratchpad region early in the execution, causing one of the blocks from each sharing pair to wait for the lock.

3.4.1.3 Overhead of *relssp* Instruction

Table 3.5 shows the run-time overhead of inserting *relssp* instruction. We report sum of the number of instructions executed by all threads for *Unshared-LRR*, *Shared-OWF*, and *Shared-OWF-OPT*. We also report the number of threads launched.

From the table, we observe that the number of instructions executed by *Unshared-LRR* and *Shared-OWF* is same. This is because *Shared-OWF* does not insert *relssp* instruction, and hence the input PTX assembly is not altered. *Shared-OWF-OPT* increases number of executed instructions as it inserts *relssp* and, in some cases, *GOTO* instruction to split critical edges. For the applications *DCT1*, *DCT2*,

[1]IPC for *Shared-OWF-OPT* also takes into account the extra instructions inserted by the compiler optimizations.

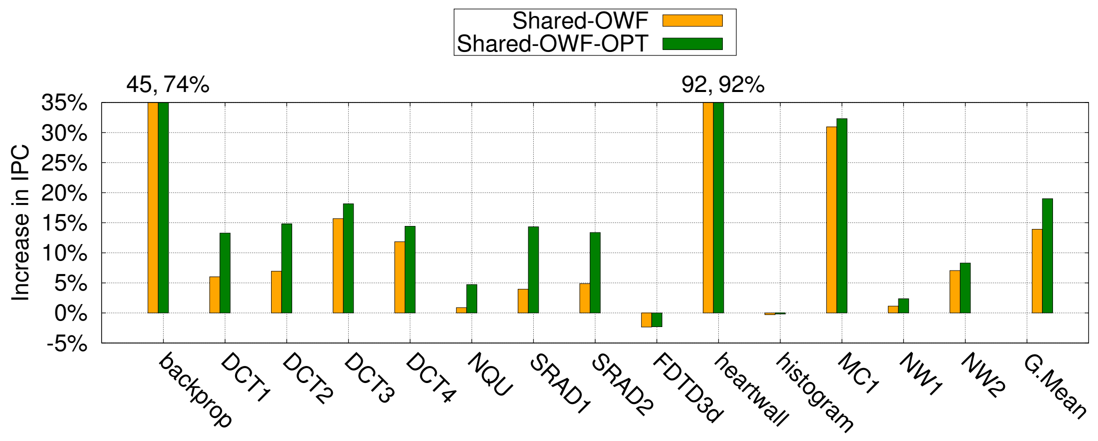


Figure 3.9: Comparing the IPC

SRAD1, *SRAD2*, *NW1*, and *NW2*, the number of additionally executed instructions (shown as *Difference* (SO-U) in the table) is equal to number of threads because *Shared-OWF-OPT* inserts only the *relssp* instruction. Further, each thread executes *relssp* exactly once. For *FDTD3d*, *heartwall*, *histogram*, and *MC1* applications, the number of additional instructions executed by *Shared-OWF-OPT* is twice that of number of threads. For these applications, each thread executes two additional instructions, i.e., one *relssp* instruction, and one *GOTO* instruction for splitting a critical edge. For *backprop*, *DCT3*, *DCT4*, and *NQU* applications, some threads take a path that has two additional instructions (*GOTO* and *relssp*), while other threads take the path which has one additional *relssp* instruction.

3.4.1.4 Effectiveness of Optimizations

Figure 3.10 shows the effectiveness of our optimizations with scratchpad sharing. We observe that all applications, except *FDTD-3d* and *histogram*, show some benefit with scratchpad sharing even without any optimizations (shown as *Shared-NoOpt*). With OWF scheduling (*Shared-OWF*), applications improve further because OWF schedules the resident warps in a way that the non-owner warps help in hiding long execution latencies. For our benchmarks, minimizing shared scratchpad region (shown as *Shared-OWF-Reorder*) does not have any noticeable impact. This is because (a) Most applications declare only a single scratchpad variable (Table 3.3) in their kernel, hence the optimization is not applicable (there is only one possible

Table 3.5: Comparing the Number of Simulated Instructions

Benchmark	Threads	Unshared-LRR (U)	Shared-OWF (S)	Shared-OWF-OPT (SO)	Difference (SO - U)
backprop	1,048,576	131,203,072	131,203,072	133,234,688	2,031,616
DCT1	32,768	9,371,648	9,371,648	9,404,416	32,768
DCT2	32,768	9,502,720	9,502,720	9,535,488	32,768
DCT3	32,768	11,255,808	11,255,808	11,304,960	49,152
DCT4	32,768	11,157,504	11,157,504	11,206,656	49,152
NQU	24,576	1,282,747	1,282,747	1,331,515	48,768
SRAD1	4,161,600	756,433,955	756,433,955	760,595,555	4,161,600
SRAD2	4,161,600	450,077,975	450,077,975	454,239,575	4,161,600
FDTD3d	144,384	5,549,531,392	5,549,531,392	5,549,820,160	288,768
heartwall	17,920	11,280,920	11,280,920	11,316,760	35,840
histogram	46,080	893,769,168	893,769,168	893,861,328	92,160
MC1	3,008	2,881,568	2,881,568	2,887,584	6,016
NW1	3,184	5,580,458	5,580,458	5,583,642	3,184
NW2	3,168	5,561,919	5,561,919	5,565,087	3,168

order of scratchpad variable declarations); and (b) For the remaining applications, the scratchpad declarations are already ordered in the optimal fashion, i.e., the access to shared scratchpad region is already minimal.

The addition of *relssp* instruction at the postdominator and at the optimal places is denoted as *Shared-OWF-PostDom* and *Shared-OWF-OPT* respectively. All Set-1 applications improve with either of these optimizations because the *relssp* instruction helps in releasing the shared scratchpad memory earlier. For *backprop* and *SRAD2* applications, *Shared-OWF-PostDom* is better than *Shared-OWF-OPT* because the threads in *backprop* execute one additional *GOTO* instruction with *Shared-OWF-OPT* (*Shared-OWF-PostDom* does not require critical edge splitting). *SRAD2* has more number of stall cycles with *Shared-OWF-OPT* as compared to *Shared-OWF-PostDom*. For most of the other benchmarks, *Shared-OWF-OPT* performs better as it can push *relssp* instruction earlier than with *Shared-OWF-PostDom*, thus releasing shared scratchpad earlier allowing for more thread level parallelism.

As expected, Set-2 applications do not show much benefit with *Shared-OWF-PostDom* or *Shared-OWF-OPT* since they access shared scratchpad memory till towards the end of their execution. Hence both the optimizations insert *relssp* instruction in the Exit block in the CFGs. The application *heartwall* does not use shared scratchpad memory and hence it shows maximum benefit even without the

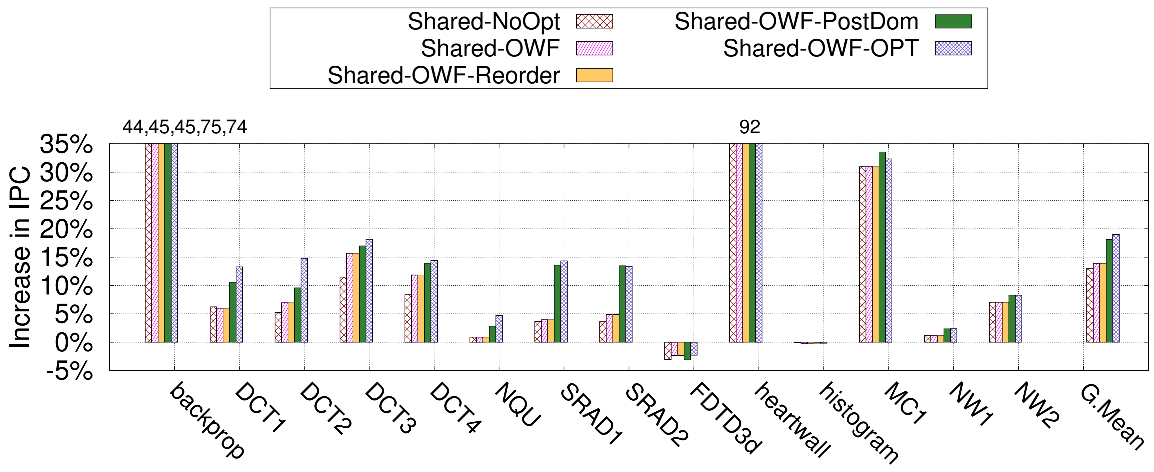


Figure 3.10: Performance Analysis of Optimizations

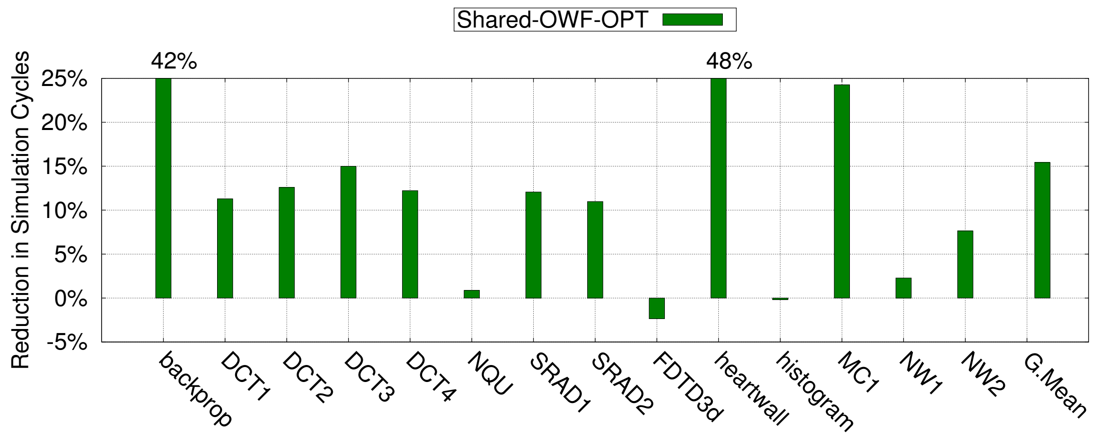


Figure 3.11: Comparing the Simulation Cycles

insertion of *relssp* instruction.

3.4.1.5 Reduction in Simulation Cycles

In Figure 3.11 we observe a maximum reduction of 47.8% and an average reduction of 15.42% in the number of simulation cycles for *Shared-OWF-OPT* when compared to *Unshared-LRR*. Recall that *Shared-OWF-OPT* causes applications to execute more number of instructions (Table 3.5). These extra instructions are also counted while computing the simulation cycles for *Shared-OWF-OPT*.

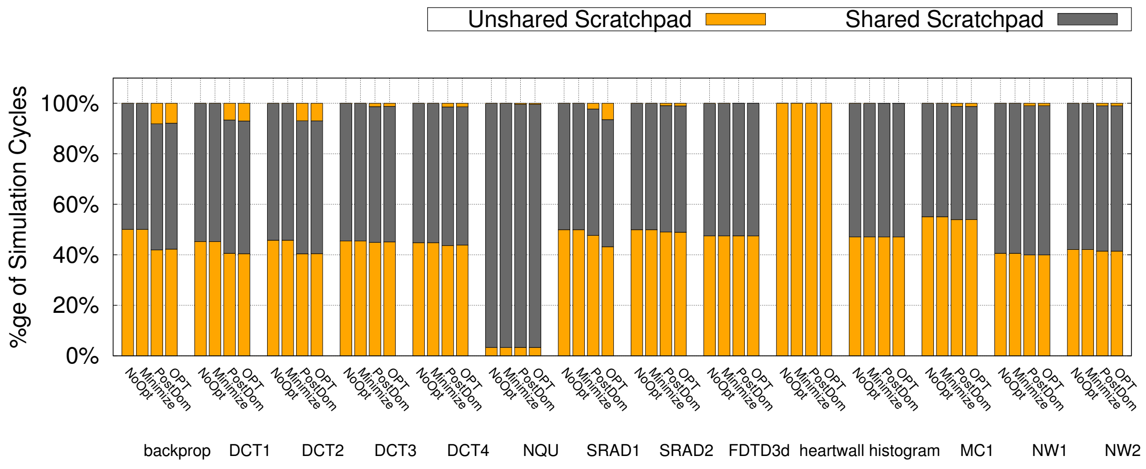


Figure 3.12: Progress of Shared Thread Blocks

3.4.1.6 Progress of Shared Thread Blocks

Figure 3.12 shows the effect of compiler optimizations by analyzing the progress of shared thread blocks through shared and unshared scratchpad regions. In the figure, *NoOpt* denotes the default scratchpad sharing when no optimizations are applied on an input kernel. *Minimize* denotes the scratchpad sharing which executes an input kernel having minimum access to shared scratchpad region. *PostDom* and *OPT* use our modified scratchpad sharing approach that execute an input kernel with additional *relssp* instructions placed at post dominator and optimal places (Section 3.2.3) respectively. In the figure, we show the percentage of simulation cycles spent in unshared scratchpad region (before acquiring shared scratchpad), shared scratchpad region, and unshared scratchpad region again (after releasing the shared scratchpad) respectively. We observe that shared thread blocks in all the applications access unshared scratchpad region before they start accessing shared scratchpad memory. Hence all the shared thread blocks can make some progress without wait. This progress is the main reason for the improvements seen with scratchpad sharing approach.

An interesting case to consider in Figure 3.12 is the application *heartwall*, where none of the shared thread blocks accesses shared scratchpad memory. However, note that this is run-time behavior, specific to the particular inputs used for the benchmark. In this case, there is no scope for compiler optimizations to further improve

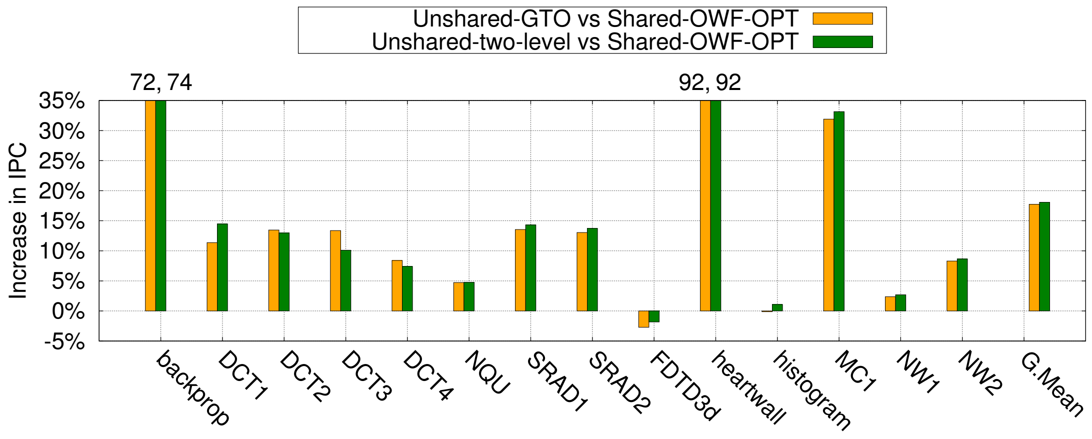


Figure 3.13: Improvement in IPC for *Shared-OWF-OPT* w.r.t. Baseline Having (a) GTO, (b) Two-Level Scheduler

the progress of shared thread blocks. Manual inspection of the code reveals that even though shared scratchpad memory is not accessed for the particular execution, it is used in the program code under an if-then-else condition. Thus it is not possible to eliminate the shared scratchpad at compile-time.

It is clear from the figure that *Minimize* does not affect *DCT1*, *DCT2*, *DCT3*, *DCT4*, *FDTD3d*, *histogram* applications because the kernels in these applications declare single scratchpad variable. For the remaining applications, *Minimize* has same effect as that of *NoOpt*, because the default input PTX kernel already accesses the shared scratchpad variables such that access to shared scratchpad is minimum. We also observe that *PostDom* and *OPT* approaches improve only those applications that spend considerable simulation cycles in unshared scratchpad region *after* last access to shared scratchpad region. We also observe that in *NoOpt* and *Minimize* approaches owner thread blocks keep shared scratchpad locked till the end of their execution. Hence, non-owner thread blocks wait for lock in the unshared scratchpad region for longer periods. However, with *PostDom* and *AllOpt* approaches, owner thread blocks release the shared scratchpad memory early using the *relssp* instruction, hence the non-owner thread blocks can start accessing the shared scratchpad early, minimizing the waiting time. Thus we see an overall performance improvement.

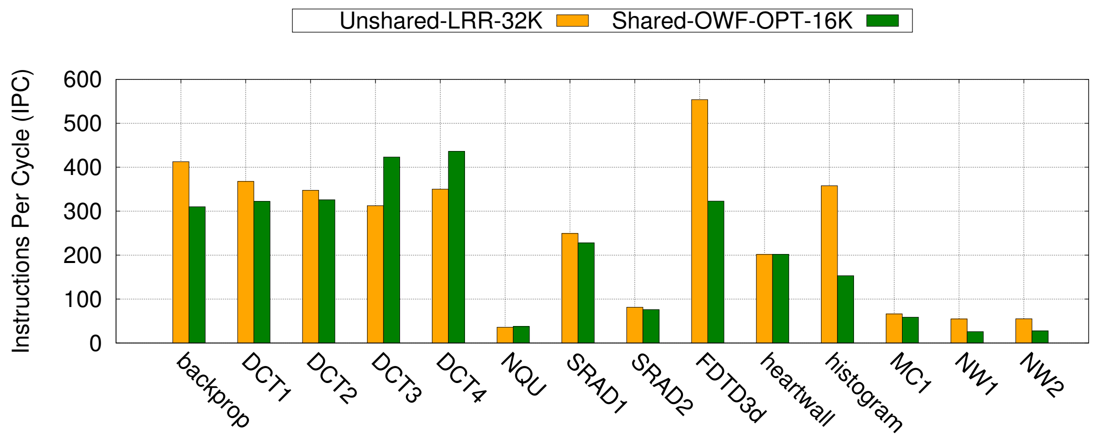


Figure 3.14: Comparison with Unshared-LRR that Uses Twice the Scratchpad Memory

3.4.1.7 Comparison with Different Schedulers

Figure 3.13 shows the effect of using different scheduling policies. The performance of *Shared-OWF-OPT* approach is compared with the baseline unshared implementation that uses greedy then old (GTO) and two-level scheduling policies respectively. We observe that *Shared-OWF-OPT* approach shows an average improvement of 17.73% and 18.08% with respect to unshared GTO and two-level scheduling policies respectively. The application *FDTD3d* degrade with our approach when compared to the baseline with either GTO scheduling or two-level scheduling since it has more number of L1 and L2 cache misses with sharing. The application *histogram* degrades with sharing when compared to the baseline with GTO scheduling because of more number of L1 misses. However *histogram* with sharing performs better than the baseline with two-level policy.

3.4.1.8 Resource Savings

Figure 3.14 compares the IPC of *Shared-OWF-OPT* with *Unshared-LRR* that uses twice the amount of scratchpad memory on GPU. We observe that *DCT3*, *DCT4*, *NQU*, and *heartwall* show improvement with *Shared-OWF-OPT* over *Unshared-LRR* even with half the scratchpad memory. This is because sharing helps in increasing the TLP by launching additional thread blocks in each SM. The applications *DCT1*, *DCT2*, *SRAD1*, *SRAD2*, and *MC1* applications perform comparable with both the

Table 3.6: Other GPGPU-Sim Configurations

Resource/Core	Configuration	
	#1	#2
Scratchpad Memory	48KB	64KB
Max Thread Blocks	16	32
Max Threads	2048	2048
Number of SMs	14	14

Table 3.7: Additional Benchmarks that are Limited by Scratchpad Memory

	Benchmark	Application	Kernel	#Scratchpad Variables	Scratchpad Size (Bytes)	Block Size
			Benchmarks for 48KB / 64KB Scratchpad Memory			
1.	RODINIA	backprop	bpnn_layerforward_CUDA	2	9408	256
2.	CUDA-SDK	DCT1	CUDAkernel2DCT	1	8320	128
3.	CUDA-SDK	DCT2	CUDAkernel2IDCT	1	8320	128
4.	GPGPU-SIM	NQU	solve_nqueen_cuda_kernel	5	10496	64
5.	CUDA-SDK	histogram	histogram256Kernel	1	9216	192
6.	CUDA-SDK	marchingCubes (MC2)	generateTriangles	2	13824	48
7.	RODINIA	NW1	needle_cuda_shared_1	2	8452	32
8.	RODINIA	NW2	needle_cuda_shared_2	2	8452	32
			Benchmarks for 48KB Scratchpad Memory			
9.	CUDA-SDK	FDTD3d	FiniteDifferencesKernel	1	3840	128
10.	RODINIA	heartwall	kernel	8	11872	128
11.	CUDA-SDK	marchingCubes (MC1)	generateTriangles	2	9216	32

approaches. For the remaining applications, *Unshared-LRR* with double scratchpad memory performs better than sharing since more number of thread blocks are able to make progress with the former.

3.4.1.9 Performance Comparison with Other Configurations

To further verify the effectiveness of scratchpad sharing and compiler optimizations, we evaluated them on two GPU configurations that use scratchpad memory of size 48KB and 64KB per SM (Table 3.6). The scratchpad memory and thread block parameters are similar to that of NVIDIA's Kepler and Maxwell architectures respectively. The benchmarks that are used for the evaluation are shown in Table 3.7. The changes in Table 3.7 with respect to those in Table 3.3 are:

- Kernel scratchpad memory size for *DCT1* and *DCT2* is increased from 2112 to 8320. This change ensures that applications are limited by scratchpad memory

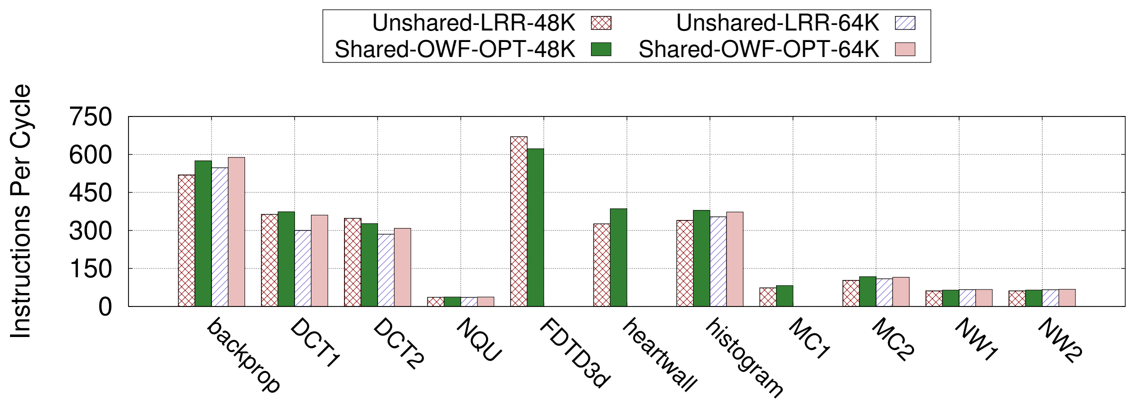


Figure 3.15: Performance Analysis for Various Configurations

for both the configurations.

- A new application, *MC2*, is created based on *MC1*—the only difference being that kernel scratchpad memory size is increased to 13824, this is to enable scratchpad sharing for Configuration-2.
- The applications *DCT3*, *DCT4*, *SRAD1*, and *SRAD2* are dropped as scratchpad sharing could not be made applicable even by increasing the kernel scratchpad memory size. These applications are limited either by the number of thread blocks or by the number of threads.

Note that scratchpad sharing is applicable for *MC1*, *FDTD3d*, and *heartwall* only with Configuration-1, but not with Configuration-2 (Table 3.6).

Figure 3.15 shows the performance comparison of *Shared-OWF-OPT* with the two baseline configurations in terms of number of instructions executed per cycle. In the figure, we use *Unshared-LRR-48K* to denote the baseline approach that uses 48KB scratchpad memory (according to Configuration-1) and *Shared-OWF-OPT-48K* to denote the scratchpad sharing approach (with all optimizations) that use 48KB scratchpad memory. The notations *Unshared-LRR-64K* and *Shared-OWF-OPT-64K*, which use 64KB scratchpad memory configuration, are defined analogously.

We observe that with *Shared-OWF-OPT-48K*, all the applications except *FDTD3d* and *DCT2* show performance improvement when compared to *Unshared-LRR-48*. Similarly with *Shared-OWF-OPT-64K*, all the applications perform better when

Table 3.8: Benchmarks Used for Comparison with Shared Memory Multiplexing [96]

Application	Kernel	#Scratchpad Variables	Scratchpad Size (Bytes)	Block Size
Matrix Vector Multiplication (MV)	mv_shared	1	4224	32
Fast Fourier Transform (FFT)	kfft	1	8704	64
MarchingCubes (MC)	generateTriangles	2	9216	32
ScalarProd (SP)	scalarProdGPU	1	4114	64
Histogram (HG)	histogram256	1	7168	32
Convolution (CV)	convolutionColumnsKernel	1	8256	128

compared to *Unshared-LRR-64K*. Consider the applications *backprop*, *DCT1*, *histogram*, *MC2*, and *NQU*. These applications with *Shared-OWF-OPT-48K* perform better even when compared to *Unshared-LRR-64K*. Also, these applications, when used with *Shared-OWF-OPT-64K*, perform better than *Unshared-LRR-64K* as well. *NW1* and *NW2* show improvement with scratchpad sharing when compared to their respective baseline approaches. Applications *heartwall* and *MC1*, where scratchpad sharing is applicable only with Configuration-1 (Table 3.6), show improvement when compared to *Unshared-LRR-48K*. For *DCT2*, increasing the amount of scratchpad memory from 48KB to 64KB per SM does not improve the performance of *Unshared-LRR* configuration since it increases number of stall cycles in the SM. Hence increase in the number of thread blocks for *Shared-OWF-OPT-48K* does not improve its performance w.r.t *Unshared-LRR-48K*. However, *Shared-OWF-OPT-48K* performs better than *Shared-OWF-OPT-64K* because *Shared-OWF-OPT-48K* has lesser number of thread blocks that make more progress, hence reducing resource contention. *FDTD3d* does not show improvement with scratchpad sharing due to increase in the stall cycles with our approach. To summarize, scratchpad sharing with compiler optimizations helps in improving the performance of the applications even with increase in the size of the scratchpad memory per SM.

3.4.1.10 Performance Comparison with Shared Memory Multiplexing [96]

Figure 3.16 compares the performance of *Shared-OWF-OPT* with the software approaches proposed by Yang et al. [96] in terms of number of simulation cycles. We use their benchmarks (Table 3.8), and simulate them on the GPU configuration shown in Table 3.2. In the figure, *VTB*, *VTB_PIPE*, and *CO_VTB* denote the

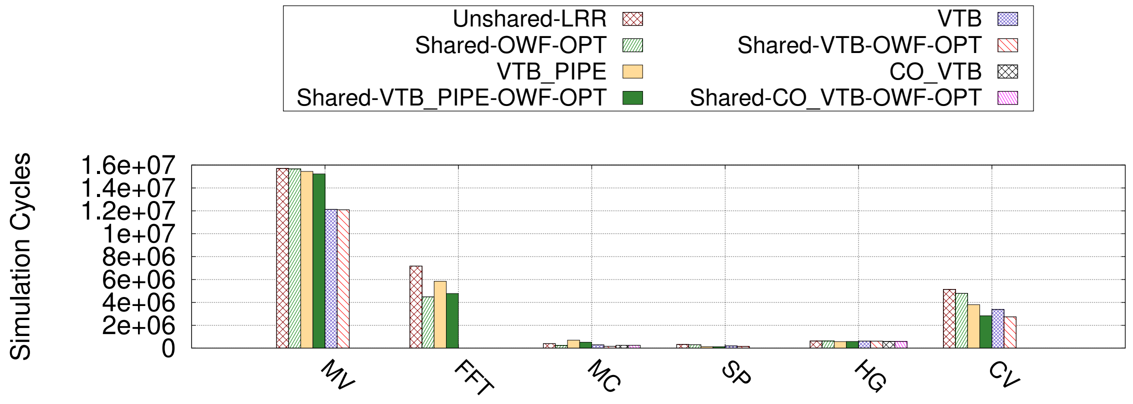


Figure 3.16: Performance Comparison with Other Approaches (Lower Value is Better)

compiler optimizations proposed by Yang et al. [96][1]. Similarly, we use *Shared-VTB-OWF-OPT*, *Shared-VTB_PIPE-OWF-OPT*, and *Shared-CO_VTB-OWF-OPT* to measure performance of scratchpad sharing on the applications that are optimized with *VTB*, *VTB_PIPE*, and *CO_VTB* respectively.

In the experiments we observe a change in the number of executed instructions for *VTB*, *VTB_PIPE*, and *CO_VTB* approaches because they require modifications to the benchmarks. In Figure 3.16, application *MC* performs better (spends fewer simulation cycles) with *Shared-OWF-OPT* than with *Unshared-LLR*, *VTB*, *VTB_PIPE*, and *CO_VTB* approaches. Interestingly, applying *Shared-OWF-OPT* on top of *VTB*, *VTB_PIPE*, or *CO_VTB* improves the performance further. Similarly, *FFT* shows improvement with *Shared-OWF-OPT* when compared to *Unshared-LRR* and *VTB_PIPE*. In this case also *Shared-VTB_PIPE-OWF-OPT* outperforms VTB_PIPE. In contrast, for the application *HG*, sharing does not impact the performance, even on the top of *VTB* and *VTB_PIPE* optimizations. This is because the additional thread blocks launched do not make much progress before they start accessing shared scratchpad. For the same reason, scratchpad sharing does not impact on *MV*. The applications *CV* and *SP* perform better with *VTB_PIPE* than with *Shared-OWF-OPT*. However, the performance is further improved when scratchpad sharing is combined with *VTB* and *VTB_PIPE* approaches. It can be

[1]Note that, as described in [96], *CO_VTB* is suitable only for few workloads (i.e., *MC* and *HG*). Also, for *FFT*, we do not compare *VTB* with *VTB_PIPE* because *VTB* combines 4 thread blocks whereas *VTB_PIPE* combines 2 thread blocks in their implementation.

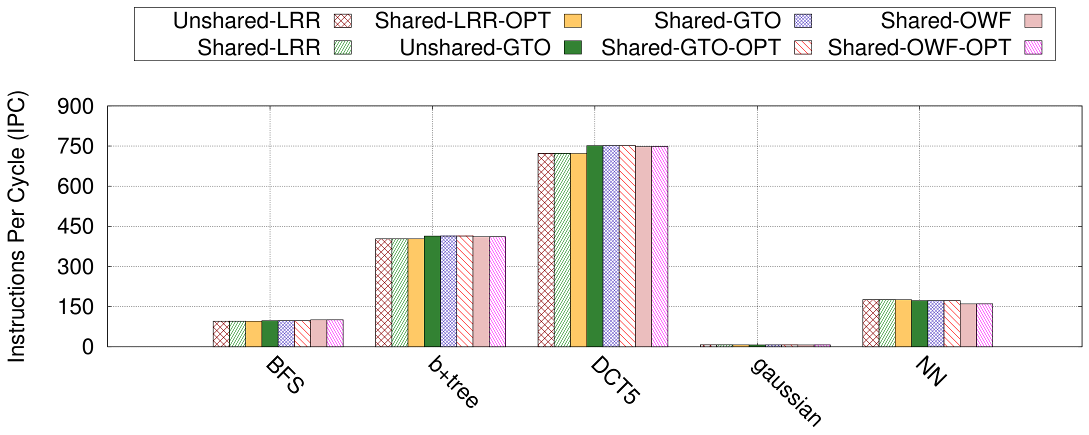


Figure 3.17: IPC Comparison of Set-3 Benchmarks

concluded from these experiments that scratchpad sharing and shared memory multiplexing approaches compliment each other well, and most applications show the best performance when the two approaches are combined.

3.4.2 Analyzing Benchmarks that are not Limited by Scratchpad Memory

Performance analysis of Set-3 benchmarks is shown in the Figure 3.17. Recall that the number of thread blocks launched by these applications is not limited by the scratchpad memory. We observe that the performance of the applications with *Unshared-LRR*, *Shared-LRR*, and *Shared-LRR-OPT* is exactly the same. For Set-3 applications all thread blocks are launched in unsharing mode. Hence *Shared-LRR* behaves exactly same as *Unshared-LRR*. Since these applications do not use any shared scratchpad memory, our compiler optimizations do not insert *relssp* instruction in their PTX code. Hence the number of instructions executed by the *Shared-LRR-OPT* approach is same as that of *Shared-LRR*. Similarly, we see that *Unshared-GTO*, *Shared-GTO*, and *Shared-GTO-OPT* behave exactly the same. However, with OWF optimization, *Shared-OWF* and *Shared-OWF-OPT* is comparable to the *Unshared-GTO* because OWF optimization arranges the resident warps according to the owner. Since all the the thread blocks own their scratchpad memory, they are sorted according to the dynamic warp id. Hence they perform comparable to

Unshared-GTO. The performances with *Shared-OWF* and *Shared-OWF-OPT* are the same because the compiler optimizations do not insert any *relssp* instruction.

3.5 Summary

In this chapter, we propose compiler optimizations to improve the availability of shared scratchpad memory. Experiments with various benchmarks help us conclude that if the number of resident thread blocks launched by an application is limited by scratchpad (Table 3.3), scratchpad sharing (with the compiler optimizations) improves the performance. On the other hand, for other applications where the number of thread blocks is not limited by scratchpad (Table 3.4), the hardware and software changes do not negatively impact the run-time.

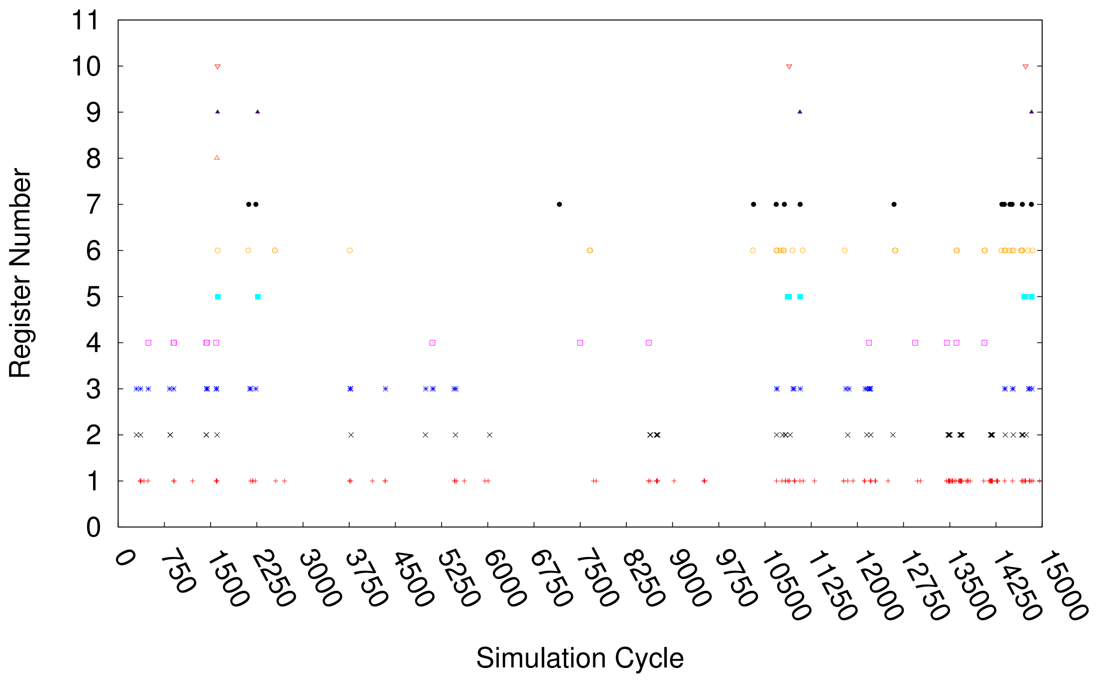
Chapter 4

Optimizations for Reducing Register File Leakage Energy

4.1 Introduction

GPUs maintain a large register file in each of their streaming multiprocessor (SM) to improve the TLP. They allow a large number of resident threads [4] in each SM, and the resident threads can store their thread context in the register file, which facilitates faster context switching of the threads. The threads that are launched in each SM are grouped into sets of 32 threads (warps), and they execute the instructions in a single instruction, multiple threaded (SIMT) manner.

To keep improving the TLP of the GPUs, GPU architects increase the maximum number of resident threads and register file sizes in every generation. However, increase in the register file size comes at a price. Earlier studies [38, 59] show that register files in the GPU consume a significant amount of power (about 15% of the total power). With decrease in the feature size of semiconductor devices, the leakage power has become a crucial factor for the manufacturing process [45, 50]. Also, Lim et al. [65] has shown that the leakage power dissipated by the GPUs is more than 50% of the total power for several benchmark applications. We also observe that registers in the GPU continue to dissipate leakage power throughout the entire execution of its warp even when they are not accessed by the warp.



Each data point shows the access of a register (Y-axis) during a cycle (X-axis).

Figure 4.1: Register Access Pattern for MUM [14]

4.1.1 Opportunities to Reduce Register Leakage Energy

To understand the severity of leakage power dissipation by register file, consider Figure 4.1 which shows the access patterns of some registers of warp 0 during the execution of *MUM* application [14]. We use the access patterns of the registers of a single warp as a representative since all the warps of a kernel typically show similar behavior during execution [9]. We make the following observations:

- Register 10 is accessed very infrequently—it is accessed for only 3 cycles out of 15000 cycles shown in the figure. In fact, it is accessed for only 7 cycles during the complete execution (life time) of the warp (29614 cycles).

- Register 1 is the most frequently accessed register during the warp execution. Even it is accessed for only 330 cycles ($\sim 1.11\%$) during the life time of the warp.

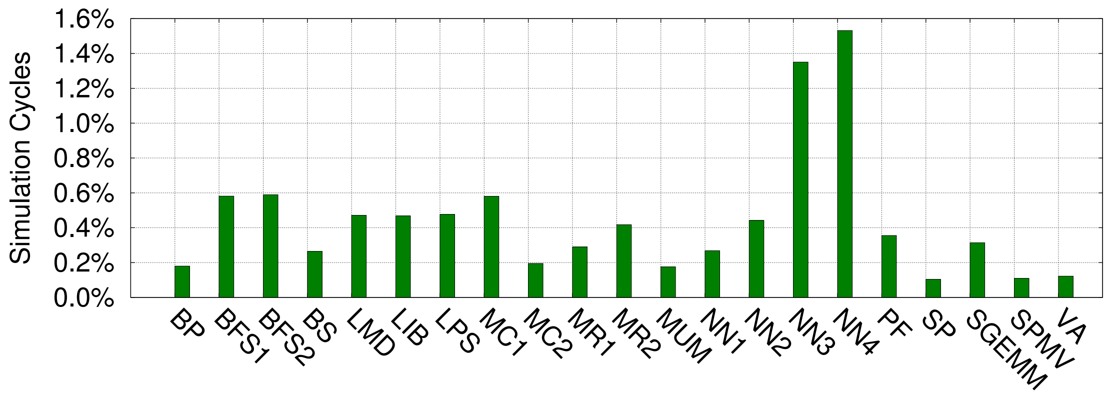


Figure 4.2: Percentage of Simulation Cycles Spent by a Register (Averaged Over all the Registers)

This shows that registers are accessed for a very short duration during the warp life time. However, they continue to dissipate leakage power for the entire life time of the warp. Figure 4.2 shows that the behavior is not specific to *MUM*, but is seen across a wide range of applications. The figure shows the percentage of simulation cycles spent in register accesses (averaged over all the registers in all the warps) for several applications. We observe that registers on an average spend $< 2\%$ of the simulation cycles during the warp execution while leaking power during the entire execution.

One solution to reduce the leakage power of the registers is by putting the registers into low power states [45]. Several studies use different notations for denoting the power state of a register. This thesis uses the following notations.

- **ON**: The default power state, which does not achieve any power savings.
- **SLEEP (Drowsy):** Drowsy [8, 22] and SLEEP [62] states refer to the same low power data preserving states. This thesis uses the term SLEEP.
- **OFF:** It denotes the low power data destroying state, however, it saves more power when compared to SLEEP state.

Warped Register File [8] reduces the leakage power of register file by putting the register into SLEEP state *immediately* after the registers of an instruction are accessed. However, this can have run-time overhead whenever there are frequent wake up signals to the sleeping register. Consider Figure 4.1 again:

- Putting register 10 to SLEEP state immediately after its accesses saves significant power because there are gaps of several thousands of cycles between consecutive accesses.

- In contrast, register 1 is accessed very frequently. If it is put to SLEEP after every access, it will have a high overhead of wake up signals.

- The access pattern of register 7 changes during the warp execution. It is accessed frequently for some duration (for example, between cycles 10500–11250), and not accessed frequently for other duration (between cycles 3000–7500). To optimize energy as well as run-time, the register needs to be kept ON whenever it is frequently accessed, and put to SLEEP otherwise.

- The last access to register 8 is at cycle 1602. The register can be turned OFF after its last access to save more power.

To summarize, the knowledge of registers' access patterns is required to improve energy efficiency without impacting the run-time adversely.

4.1.2 Our Solution: GReEneR

To reduce the leakage energy of the register file, we have developed **GReEneR** for a given assembly program[1]. **GReEneR** uses a compile-time analysis to determine the power state of the registers (OFF, SLEEP, or ON) for each instruction by estimating the register usage information. It then transforms the input assembly program by encoding the power state information at each instruction to make it energy efficient. The static analysis makes safe approximations while computing power state of the registers, therefore, the choice of the state can be suboptimal at run-time. Hence, to improve the accuracy and energy efficiency, **GReEneR** provides a

[1]Note that the techniques [45] to reduce leakage power using low power states address the subthreshold leakage power. Hence, in **GReEneR** the savings on leakage energy refer to savings on subthreshold leakage energy.

run-time optimization that dynamically corrects the power state of registers of each instruction.

To summarize, this chapter describes following contributions of the thesis.

1. We introduce a new instruction format that supports the power states for the instruction registers (Section 4.2.2).

2. We propose a compile-time analysis that determines the power state of the registers at each program point and transforms an input assembly language into a power optimized assembly language (Section 4.2.1 and 4.2.2).

3. We give a run-time optimization to reduce the penalty of suboptimal (but safe) choices made by static analysis (Section 4.2.3).

4. We implemented the proposed compile-time and run-time optimizations using GPGPU-Sim simulator [3]. We integrated GPUWattch [59] with CACT-P [63] version to enable power saving mechanism (Section 4.3).

5. We evaluated our implementation on wide range of kernels from different benchmark suites: CUDASDK [2], GPGPU-SIM [14], Parboil [7], and Rodinia [19]. We are able to reduce the register leakage energy by an average of 46.96% and maximum of 57.57% (Section 4.3).

The rest of the chapter is organized as follows. Section 4.2 describes the details of **GREENER**. Section 4.3 shows the experimental results, and Section 4.4 summarizes the chapter.

4.2 GREENER

To understand the working of **GREENER**, we need to understand the different access patterns of a register and their effect on the wake up penalty incurred. Let W (threshold) denotes the minimum number of program instructions (not the number of clock cycles) that are required to offset the wake-up penalty incurred when a register state is switched from OFF or SLEEP state to ON state. Consider a program that accesses some register R in a statement S during execution. The future accesses of R in this execution govern its power state. The following scenarios exist:

1. The next access (either read or write) to R is by an instruction S' and there are no more than W instructions between S and S'. In this case, since the two accesses to R are very close, it should be kept ON to avoid any wake-up penalty associated with SLEEP or OFF state.

2. The next access to R is a read access by an instruction S' and there are more than W instructions between S and S'. In this case, since the value stored in R is used by S', we can not switch R to OFF state as it will cause the loss of its value. However, we can put R in SLEEP state.

3. The next access to R is a write access by an instruction S' and there are more than W instructions between S and S'. In this case, since the value stored in R is being overwritten by S', we can put R in OFF state.

4. There is no further access to R in the program. In this case also, register R can be safely turned OFF.

We now describe the compiler analysis used by **GReEneR** to capture these scenarios.

4.2.1 Compiler Analysis

To compute power state of registers at each instruction, we perform compiler analysis at the instruction level. Determining the power state of each register requires knowing the life time of registers as well as the distance between the consecutive accesses to the registers. We use the following notations.

- $\mathsf{IN}(S)$ denotes the program point before the instruction S. $\mathsf{OUT}(S)$ denotes the program point after the instruction S.

- $\mathsf{SUCC}(S)$ denotes the set of successors of the instruction S. An instruction I is said to be successor of S if the control may transfer to I after executing the instruction S.

- $\mathsf{isLive}(\pi, R)$ is true if there is some path from π to Exit that contains a use of R not preceded by its definition.

- $\mathsf{Dist}(\pi, R)$ denotes the distance in terms of number of instructions from program point π till the next access to R. $\mathsf{Dist}(\pi, R)$ is set to ∞ when it exceeds the threshold W.
- $\mathsf{SleepOff}(\pi, R)$ is true if the register R can be put into SLEEP or OFF state at program point π.
- $\mathsf{Power}(\pi, R)$ denotes the power state of the register R at program point π.

The liveness information of each register, $\mathsf{isLive}(\pi, R)$, can be computed using traditional liveness analysis [47]. The data flow equations to compute the $\mathsf{Dist}(\mathsf{IN}(S), R)$ and $\mathsf{Dist}(\mathsf{OUT}(S), R)$ are as follows:

$$\begin{aligned}
\mathsf{Dist}(\mathsf{IN}(S), R) &= \begin{cases} 1, \text{ if } S \text{ accesses } R \\ \mathsf{INC}(\mathsf{Dist}(\mathsf{OUT}(S), R)), \text{otherwise} \end{cases} \\
\mathsf{INC}(x) &= \begin{cases} \infty, \text{ if } x \text{ is } W \text{ or } \infty \\ x + 1, \text{otherwise} \end{cases} \\
\mathsf{Dist}(\mathsf{OUT}(S), R) &= \begin{cases} \infty, \text{ if } S \text{ is } \mathsf{Exit} \\ \max\limits_{SS \in \mathsf{SUCC}(S)} \mathsf{Dist}(\mathsf{IN}(SS), R), \text{ otherwise} \end{cases}
\end{aligned}$$

Note that $\mathsf{INC}(x)$ is a saturating increment operator. Since our analysis aims to reduce the power consumption, we compute $\mathsf{Dist}(\mathsf{OUT}(S), R)$ as the maximum value of $\mathsf{Dist}(\mathsf{IN}(SS), R)$ over the successors SS of S. A register R can potentially be put into SLEEP or OFF state at a program point π if it is not accessed within the distance window W on some path:

$$\mathsf{SleepOff}(\pi, R) = (\mathsf{Dist}(\pi, R) == \infty)$$

The power state of each register at each program point can be computed according to Table 4.1.

4.2.2 Encoding Power States

The power state (*Power_State*) of a register can be one of the three states: **OFF**, **SLEEP**, or **ON**. Thus, it requires two bits to represent *Power_State* of one register.

Table 4.1: Computing Power State of a Register R at a Program Point π

$\mathsf{isLive}(\pi, R)$	$\mathsf{SleepOff}(\pi, R)$	$\mathsf{Power}(\pi, R)$
true	true	SLEEP
true	false	ON
false	true	OFF
false	false	ON

Since the power state can change after every instruction at run-time, we need to encode the *Power_State* of the operand registers of an instruction in the instruction itself.

PTXPlus instructions [3] can support up to 4 source and 4 destination registers. Encoding *Power_State* of all the registers will require 16 bits. We observed that in our benchmarks, most instructions use only up to 2 source registers and 1 destination register. Therefore, to reduce the number of bits required to encode *Power_State* in each instruction, we encode information only for 2 source registers and 1 destination register. For instructions having more registers, *Power_State* of the remaining registers is assumed to be **SLEEP** to enable power saving. The modified instructions have the format:

<Opcode> <Options> <Operand_List> <***Power_State_List*** >

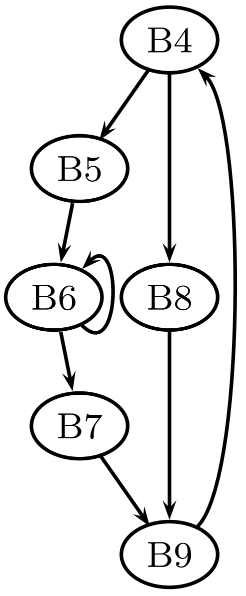
Note that *Power_State* encoded for a register R for an instruction S is given by $\mathsf{Power}(\mathsf{OUT}(S), R)$.

Example 4.2.1. *Figure 4.3(a) shows a snippet of power optimized PTXPlus code, which is generated for* SP *benchmark using a threshold value (W) 7. The control flow graph (CFG) corresponding to the snippet is shown in Figure 4.3(b). Note that the CFG is shown with respect to traditional basic block level to show it in compact. In Figure 4.3(a), explicit branch addresses have been replaced by block labels for ease of understanding. The shaded text in the instructions indicates the power states inserted by* **GReEneR**.

*The instruction at Line-1 uses 2 source registers (r8, r0) and 2 destination registers (p2, o127). As discussed, our analysis inserts the power states only for 2 source registers and 1 destination register. In this case, the power states **ON, SLEEP, ON** correspond to the registers p2, r8, and r0 respectively. The power state of o127*

```
 1  B4:set.le.s32.s32 $p2/$o127, $r8, $r0, ON, SLEEP, ON;
 2     ssy 0x00000110;
 3     mov.u32 $r1, $r0, SLEEP, ON;
 4     $p2.ne bra B8;
 5  B5:shl.u32 $r10, $r0, 0x00000002, ON, SLEEP;
 6     mov.u32 $r12, $r124, ON, SLEEP;
 7     add.half.u32 $r11, s[0x0018], $r10, ON, ON;
 8     add.half.u32 $r10, s[0x0020], $r10, ON, ON;
 9  B6:ld.global.u32 $r14, [$r11], ON;
10     ld.global.u32 $r13, [$r10], ON;
11     mad.f32 $r12, $r14, $r13, $r12, SLEEP, OFF, OFF;
12     add.u32 $r1, $r1, 0x00000400, ON, ON;
13     set.gt.s32.s32 $p2/$o127, $r8, $r1, ON, SLEEP, SLEEP;
14     add.u32 $r10, $r10, 0x00001000, SLEEP, SLEEP;
15     add.u32 $r11, $r11, 0x00001000, SLEEP, SLEEP;
16     $p2.ne bra B6;
17  B7:bra B9;
18  B8:mov.u32 $r12, $r124, ON, SLEEP;
19  B9:add.u32 $r0, $r0, $r5, ON, ON, SLEEP;
20     shl.b32 $ofs1, $r9, 0x0, ON, ON;
21     set.le.s32.s32 $p2/$o127, $r0, $r6, ON, SLEEP, SLEEP;
22     mov.u32 s[$ofs1+0x0000], $r12, OFF;
23     add.u32 $r9, $r9, $r7, SLEEP, SLEEP, SLEEP;
24     $p2.ne bra B4;
```

(a) Power Optimized PTXPlus.



(b) CFG

The shaded text in part (a) denotes the power states inserted by **GReEneR**

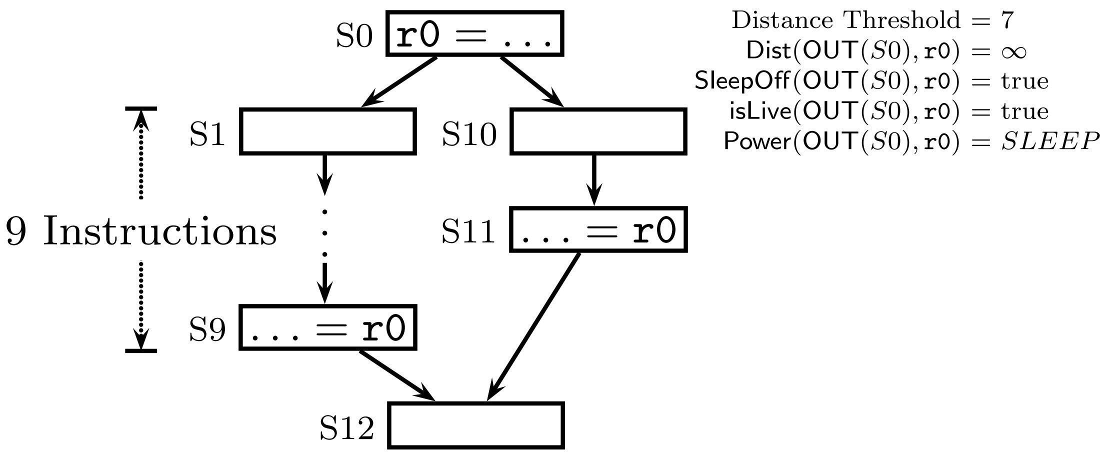
Figure 4.3: A Snippet of the Program and its CFG for *SP* Benchmark [2]

register (the fourth register in the instruction) is set to SLEEP state after accessing the register.

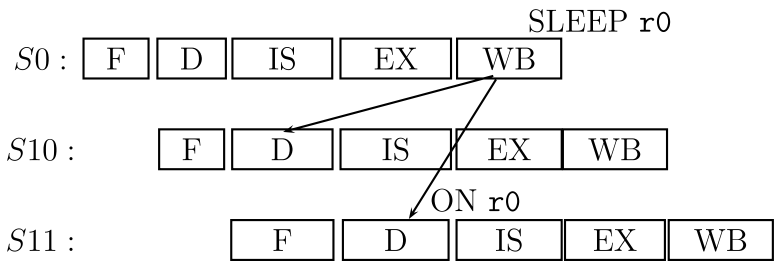
For register r0 of the instruction, the next access to the register occurs at Line-3 (at distance 2, less than the threshold value 7). Hence, the compiler inserts the power state as ON. Register p2 is also kept in ON state for a similar reason. For register r8 of the same instruction, the next access occurs along two paths. One of the paths has a use at a distance of 8 (along B5 at Line-13, $>$ 7), and the other has a definition after B9 (not shown in the figure). **GReEneR** *keeps the register in SLEEP state since there is a path along which the next access happens after a distance > 7.*

Finally, consider register r13 accessed by the instruction at Line-11. There is no further access of r13 along any path in the program. Therefore, the power state of r13 is set to OFF to save power. □

At run-time, power state of the source registers are set after the register contents



(a) Computing Distance at Branch Divergence



(b) Correcting Power State at Run-time. The pipeline phases are: Fetch (F), Decode (D), Issue (IS), Execute (EX), and Writeback (WB)

Figure 4.4: Example for Run-time Optimization

have been read, i.e., in the read operands phase in the GPU pipeline, and the power state of the destination registers are set after the register contents have been written, i.e., in the write back stage of the pipeline. The details of the hardware implementation are discussed in Section 4.2.4.

4.2.3 Run-time Optimization

Recall that the compiler analysis described in Section 4.2.1 computes $\mathsf{Dist}(\mathsf{OUT}(S), R)$ as the maximum distance value over all successors when $\mathsf{OUT}(S)$ is a branch point. This decision increases the chances of power savings, but it can be suboptimal at run-time as shown by the following example.

Example 4.2.2. *Consider the CFG in Figure 4.4(a) for a hypothetical benchmark. Assume the threshold value of 7 for* **GREENER***. Instruction S0 defines a register r0. The next access to r0 occurs along two paths: the path along S10 has a use at a distance of 2, and the other (along S1) has a use in S9 at a distance of* ∞ *(>7).* **GREENER** *computes* $\mathsf{Dist}(\mathsf{OUT}(S0), r0)$ *as* ∞*, the maximum of the distances along the successors. Further, the state* $\mathsf{Power}(\mathsf{OUT}(S0), r0)$ *is computed as SLEEP. When the program executes along the path along S1, power is saved. However, if the program executes the path along S10, then the register needs an immediate wake up, causing an overhead.* □

GREENER's compile-time decision can be corrected at run-time by looking at near future accesses of a register in the pipeline. The hardware is modified to check in the pipeline if any instruction from the same warp has been decoded that accesses a register whose power state is being changed to SLEEP or OFF (Section 4.2.4). If so, then the register power is kept ON. This avoids the wake up latencies for instructions that access the same register within a short duration, thereby avoiding the performance penalty.

Example 4.2.3. *Figure 4.4(b) shows a possible execution sequence of a program whose CFG is shown in Figure 4.4(a). The instruction* $S0$ *writes to register r0. After writing the register value in write back stage (WB), the register needs to be put into SLEEP state. Assume that the program takes the path along S10 and decodes the instruction* $S11$ *before the write back stage of* $S0$*. Our run-time optimization*

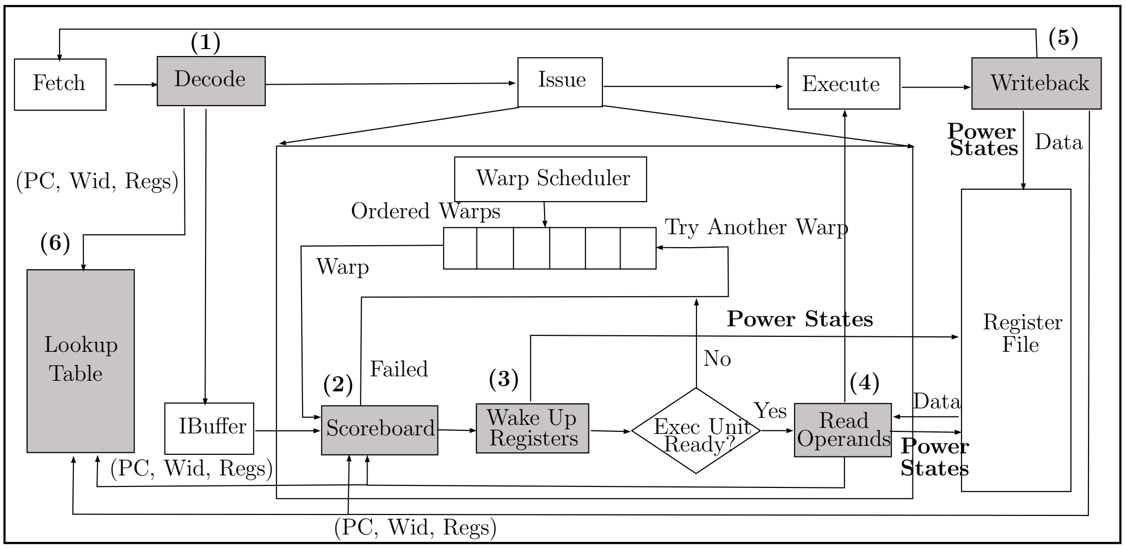


Figure 4.5: Modifications to GPU Pipeline

detects the future access to r0 by S11, and keeps the register in ON state instead of putting it into SLEEP state to avoid additional wake up latencies. On the other hand, if the program takes the path along S1, then the instruction present in the S9 would appear much later in the pipeline (after WB stage of S0). The register r0 will be set to SLEEP state. □

4.2.4 Hardware Support

Figure 4.5 shows the modified pipeline of GPU Architecture that supports our proposed ideas, with the modified components shaded and labeled. The changes are described below and the corresponding overheads are quantified in Section 4.3.5.

1. To support the new instruction format (Section 4.2.2), we modify the decode unit to extract the power states of the registers from the instruction (Label (1) in Figure 4.5).

2. The scoreboard unit (Label (2)) is modified to track RAR (Read After Read) and WAR (Write after Read) dependencies in addition to RAW (Read After Write) and WAW (Write after Write) dependencies. This is done by adding instruction's source registers in the scoreboard table. It is because an instruction can change the power state of a register to SLEEP or OFF after reading

the registers. Hence, the subsequent instructions that read/write the same register need to wait until the power state is modified.

3. The registers in SLEEP or OFF state are woken up by sending a wake up signal to the register file (Label (3)). A warp is considered ready for issuing its current instruction only when all its operand registers are in ON state.

4. The read operands phase (Label (4)) is modified (a) to set the power state of source registers after they have been read and (b) to release the source registers of the instruction which were reserved by the scoreboard unit.

5. The write back stage (Label (5)) includes the logic to set the power state of the destination registers after the registers are written.

6. The run-time optimization is implemented by adding a lookup table (Label (6)) to keep track of the registers accessed by an instruction. For an instruction having program counter PC and warp id Wid, the lookup table is indexed by Wid. When an instruction is decoded, the decode unit inserts the instruction's operand registers into the lookup table. When a warp (Wid) needs to set the power state of a register (R) of an instruction (PC) to SLEEP or OFF, it searches the lookup table for another instruction (a different PC) with the same Wid and accessing R. If a match is found, then the power state of R is kept ON, otherwise, it is changed. After an instruction completes its writeback stage, the corresponding entry is removed from the lookup table.

Each entry for a warp in the look up table stores instruction's PC, and its register numbers. The number of entries required for each warp is determined by the pipeline depth, which can be large. However, in practice, the number of entries required per each warp is less, and experimentally we found that the average number of entries per warp is less than 2. If an SM allows maximum W resident warps, stores w entries per each warp, supports r operand registers for each instruction, and allows maximum R registers per each thread, then the size of look up table (in bits) is $W * w * (sizeof(PC) + (log_2(R) * r))$.

Table 4.2: GPGPU-Sim Configuration

Resource	Configuration
Architecture	NVIDIA Tesla K20x
Number of SMs	14
Shader Core Clock	732 MHz
Technology Node	22nm
Register File Size per SM	256KB
Number of Register Banks	32
Max Number of TBs per SM	16
Max Number of Threads per SM	2048
Warp Scheduling	LRR
Number of Schedulers per SM	4

4.3 Experimental Analysis

We implemented the proposed hardware changes and compiler optimizations in GPGPU-Sim V3.x [3]. The modified instruction format is implemented by extending the PTXParser provided by GPGPU-Sim. The GPGPU-Sim configuration used for the experiments is shown in Table 4.2. We used GPUWattch [59] to measure the power consumption of register file.

Note that GPUWattch internally uses CACTI [18] to measure the power dissipation that does not support leakage power saving mechanism. Therefore, we modified GPUWattch to use CACTI-P [63] that provides power gating technique, which can minimize the leakage power by setting the SRAM cells into low power (SLEEP or OFF) state. It uses minimum data retention voltage so that SRAM cells can enter into SLEEP state without losing their data. We chose SRAM$_{vccmin}$ to be the default value (provided by CACTI-P depending on the technology node, 22nm for this case). To put SRAM cells in OFF state, we configured SRAM$_{vccmin}$ to 0 V. After running several experiments, we chose the threshold value (W) as 3, which achieves lowest energy for maximum number of kernels. We used the latency to wake up a register from SLEEP to ON state to be 1 cycle as reported in [63], and the latency to wake up a register from OFF to ON state be twice (i.e., 2 cycles) [62], except for Section 4.3.6 where we consider the effect of other values for the wake up latencies on performance and energy consumption. We report these latency and energy overheads in Section 4.3.5 and also include these overheads throughout our

Table 4.3: Benchmarks Used for Evaluation

Sr. No.	Benchmark	Application	Notation	Kernel
1	RODINIA	backprop	BP	bpnn_adjustweights_cuda
2	RODINIA	bfs	BFS1	Kernel
3	RODINIA	bfs	BFS2	Kernel2
4	CUDA-SDK	Blackscholes	BS	BlackScholesGPU
5	GPGPU-SIM	LIB	LIB	Pathcalc_Portfolio_KernelGPU
6	RODINIA	lavaMD	LMD	kernel_gpu_cuda
7	GPGPU-SIM	LPS	LPS	GPU_laplace3d
8	CUDA-SDK	MonteCarlo	MC1	inverseCNDKernel
9	CUDA-SDK	MonteCarlo	MC2	MonteCarloOneBlockPerOption
10	PARBOIL	mri-q	MR1	ComputePhiMag_GPU
11	PARBOIL	mri-q	MR2	ComputeQ_GPU
12	GPGPU-SIM	MUM	MUM	mummergpuKernel
13	GPGPU-SIM	NN	NN1	executeFirstLayer
14	GPGPU-SIM	NN	NN2	executeSecondLayer
15	GPGPU-SIM	NN	NN3	executeThirdLayer
16	GPGPU-SIM	NN	NN4	executeFourthLayer
17	RODINIA	pathfinder	PF	dynproc_kernel
18	PARBOIL	sgemm	SGEMM	mysgemmNT
19	CUDA-SDK	scalarProd	SP	scalarProdGPU
20	PARBOIL	spmv	SPMV	spmv_jds
21	CUDA-SDK	vectorAdd	VA	VecAdd

results.

We evaluated **GReEneR** on several applications from the benchmark suites CUDA-SDK [2], GPGPU-SIM [14], Parboil [7], and Rodinia [19]. Table 4.3 shows the list of applications and kernel that is simulated for each application. We compiled all the applications using CUDA-4.0[1] and simulated them using GPGPU-Sim simulator. As discussed in Section 2.5, we simulated all the applications using their PTXPlus representations because this chapter deals with the register resource and PTXPlus uses more optimal number of registers.

We measured the effectiveness of our approach using the following metrics: (1) Power, (2) Energy, (3) Simulation Cycles.

We use *Baseline* to denote the default GPGPU-Sim implementation that does

[1]GPGPU-Sim does not support above CUDA 4.0

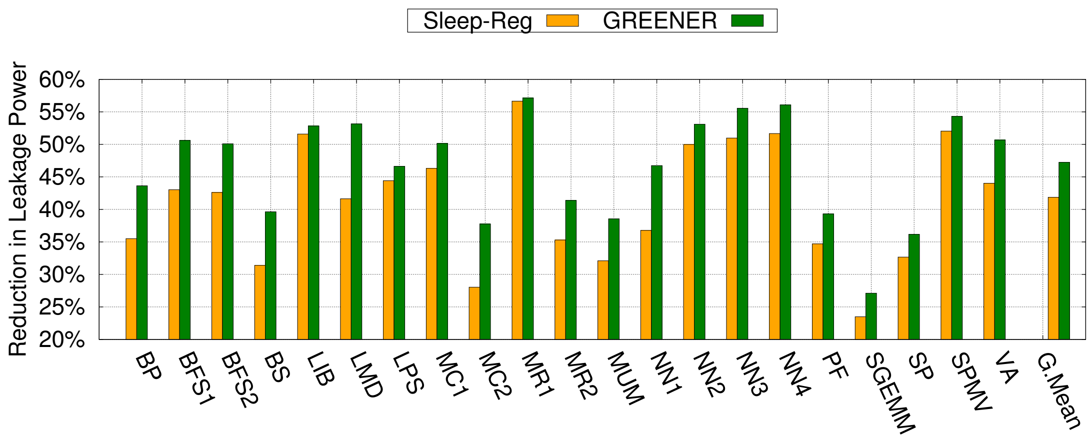


Figure 4.6: Comparing Register Leakage Power

not use any leakage power saving mechanisms. We denote *Sleep-Reg* for the approach that optimizes the baseline approach by (1) turning OFF the unallocated registers and (2) turning the allocated registers into SLEEP state immediately after the registers are accessed as described in [8].

4.3.1 Comparing Register Leakage Power

Figure 4.6 shows the effectiveness of **GREENER** and *Sleep-Reg* by measuring the reduction in leakage power with respect to *Baseline*. From the figure, we observe that **GREENER** shows an average (Geometric Mean denoted as *G.Mean*) reduction of leakage power by 47.24% when compared to the *Baseline*. It shows the **GREENER** is effective in turning the instruction registers into lower power state, such as SLEEP or OFF state depending on the behavior of the registers. The *Baseline* does not provide any mechanism to save the leakage power, as a result, the registers of a warp continue to consume leakage power throughout the warp execution. We also expect a reduction in leakage power, because two instructions that are separated by smaller distance at compile time can be scheduled at different times, with a large number of simulation cycles gap between their execution.

Figure 4.6 also shows that *Sleep-Reg* approach reduces the register leakage power by 41.86% when compared to *Baseline*, however, **GREENER** is more power efficient than *Sleep-Reg*. It is because *Sleep-Reg* approach reduces the leakage power by turning the instruction registers into SLEEP state immediately after the instruction

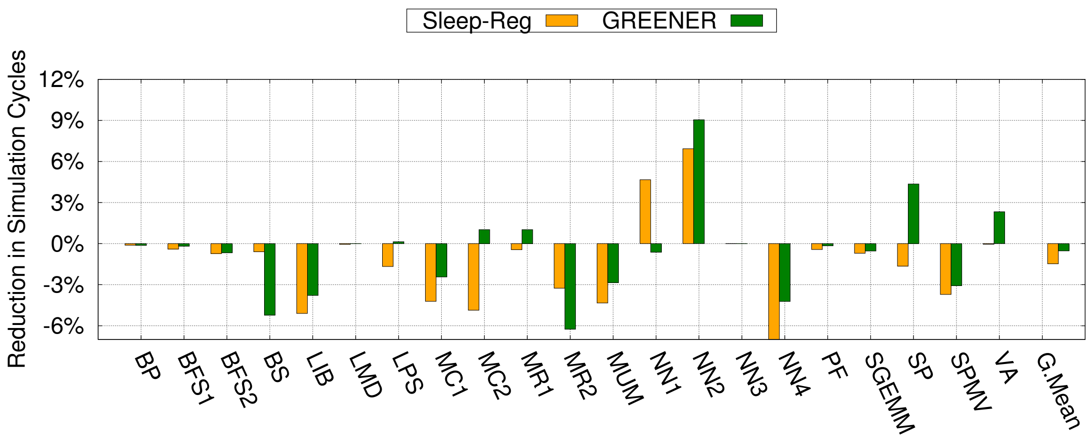


Figure 4.7: Comparing Performance in terms of Simulation Cycles

operands are accessed, without considering the access pattern of the registers. If a register needs an immediate access, then keeping the register into SLEEP instead of ON state requires additional latency cycles to wake up the register, and during these additional cycles, the registers consume power. Further, **GREENER** saves more leakage power compared to *Sleep-Reg* by turning the registers into OFF state when there is no future use of the register, whereas *Sleep-Reg* turns the register into only SLEEP state irrespective of its further usage.

4.3.2 Performance Overhead Using Simulation Cycles

Figure 4.7 shows the performance overheads of **GREENER** and *Sleep-Reg* approaches in terms of the number of simulation cycles with respect to *Baseline*. On an average, the applications show a negligible performance overhead of 0.53% with respect to *Baseline*. A slowdown is expected because **GREENER** turns the registers into SLEEP or OFF states to enable power savings, and these registers are turned back to ON state (woken up) when they need to be accessed. This wake up process takes few additional latency cycles which leads to increase in the number of simulation cycles. The overhead is not high, because the additional latency cycles are hidden by using the thread-level-parallelism and warp scheduling policy.

Interestingly, some applications (*LPS, MC2, MR1, NN2, SP*, and *VA*) show improvement in their performance. This occurs due to the change in the issuing order of the instructions. The warps that require their registers to be woken up

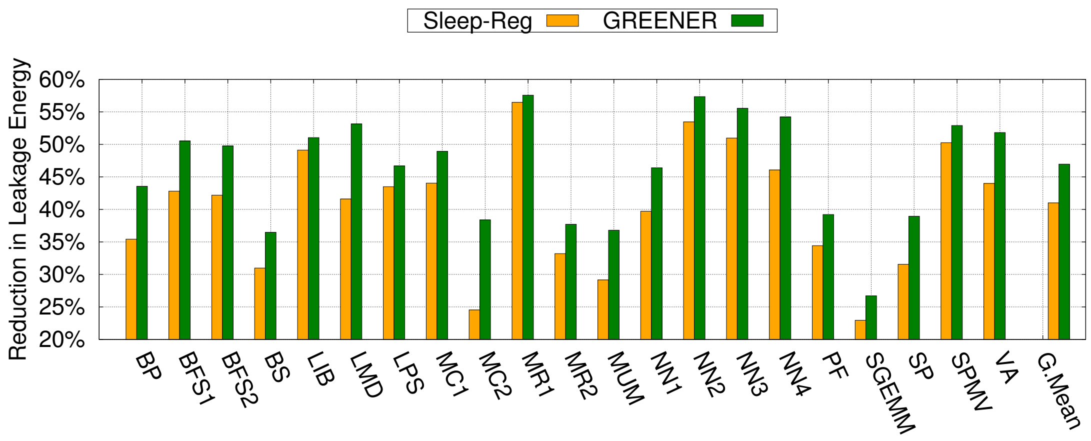


Figure 4.8: Comparing Register Leakage Energy

can not be issued in its current cycle, instead other resident warps that are ready can be issued. This change in the issue order leads to change in the memory access patterns, which in turns changes L1 and L2 cache misses etc. In case of *LPS, MC2*, and *NN2* applications, we observe an improvement in the performance due to less number of pipeline stall cycles with **GReEneR** when compared to *Baseline*. *MR1* shows less number of scoreboard stall cycles with **GReEneR** when compared to *Baseline*. Though *SP* and *VA* applications have same number L1 and L2 cache misses with **GReEneR** and *Baseline* approach, **GReEneR** shows less number of pipeline stall cycles when compared to *Baseline*.

Figure 4.7 also shows that *Sleep-Reg* has an average performance degradation of 1.48% when compared to the *Baseline* approach. This degradation is more when compared to **GReEneR** because *Sleep-Reg* turns all the instruction registers into SLEEP state after the instruction operands are accessed, irrespective of their usage pattern. If a register in SLEEP state is accessed in near future, it needs to be turned on, this incurs additional wake up latencies with *Sleep-Reg*. Whereas, our approach minimizes these additional wake up latency cycles by retaining such registers in the ON state. However, *MR2* performs better with *Sleep-Reg* because it shows less number of scoreboard and idle cycles than that of **GReEneR**. Also, *Sleep-Reg* performs better with *BS* and *NN1* since it has less number of stall cycles when compared to **GReEneR**.

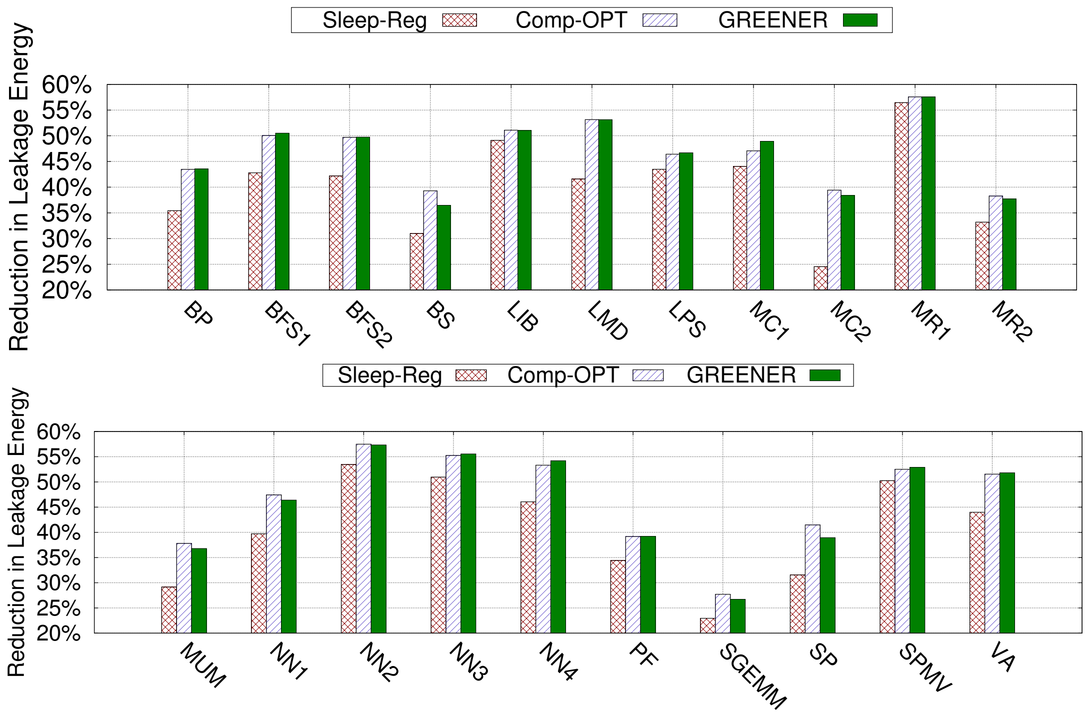


Figure 4.9: Comparing Effectiveness of Individual Optimizations

4.3.3 Comparing Register Leakage Energy

Figure 4.8 compares the total energy savings of **GREENER** and *Sleep-Reg* w.r.t. *Baseline*. The results show that **GREENER** achieves an average reduction of register leakage energy by 46.96% and 10.1% when compared to *Baseline* and *Sleep-Reg* respectively. From Figures 4.6 and 4.7, we see that **GREENER** shows more leakage power saving, also has negligible performance overhead with respect to the *Baseline*, hence we achieve a significant reduction in leakage energy. Also, the applications that exhibit more power savings and improve their performance with **GREENER**, further show more leakage energy savings. Similarly, the applications that show leakage power savings but has more performance overhead will reduce their leakage energy savings accordingly when compared to *Baseline* and *Sleep-Reg* approaches.

4.3.4 Effectiveness of Optimizations

We show the effectiveness of the proposed optimizations in Figure 4.9. Note that the figure has been split into two parts for better readability. From the figure, we analyze that the compiler optimization (discussed in Section 4.2.1, and denoted as *Comp-OPT*) saves more energy (average 47.14%) when compared to *Sleep-Reg* (41.0%). This shows that turning the registers into low power states (SLEEP or OFF state) with the knowledge of register access pattern is more effective than turning the registers into SLEEP state after accessing them.

The run-time optimization (discussed in Section 4.2.3) is evaluated by combining it with *Comp-OPT*, and we denote them as **GReEneR** in the figure. From the results, we observe that, for most of the applications, **GReEneR** show minor improvements when compared to *Comp-OPT* respectively. This is because the run-time optimization helps only in correcting power state of a register by turning to ON state when it detects the future access to the register at run-time. However, if the register is not found to be accessed in the near future at run-time, it does not modify and retains the power state as directed by the *Comp-OPT*. For some applications (e.g. *NN2*), **GReEneR** is less efficient when compared to *Comp-OPT*. It occurs when a register that is determined to be accessed in the near future does not get accessed due to reasons such as scheduling order, scoreboard stalls, or the unavailability of the corresponding execution unit. In those cases, keeping the register into low power states (SLEEP or OFF) can save more energy instead of keeping it in ON state. Note that the effectiveness of run-time optimization depends on the application behavior at the branch divergence points.

4.3.5 Analyzing Hardware Overheads

To support leakage power saving, CACTI-P [63] introduces additional sleep transistors into the SRAM structures. These transistors enable us to put the registers into low power states (SLEEP or OFF) after accessing the operands (discussed in Section 4.2.4), also they enable us to wake up the registers from lower power states before accessing the operands. For the configuration used in our experiments, Table 4.4 shows the additional area, latency, and energy associated with the additional sleep transistors circuitry. Note that in our experiments, we conservatively consider

Table 4.4: Hardware Overheads for Sleep Transistor Circuitry

Parameter	Overhead
Area	0.04709 mm^2
Wake up Latency (SLEEP to ON)	0.005759 ns ($<$ 1 clock cycle)
Wake up Latency (OFF to ON state)	0.01749 ns ($<$ 1 clock cycle)
Energy (SLEEP to ON and vice versa)	0.0373 nJ
Energy (OFF to ON and vice versa)	0.11774 nJ

the latency overhead to change the power state of a register from OFF to ON state to be 2 cycles.

Recall that **GReEneR** encodes the power state of a register with its instruction, and we require 6 bits to encode the power states of the instruction registers. Currently, NVIDIA does not disclose the machine code format of the instructions. However, we can adopt either of the following two solutions as described in [85]. (1) If the instruction format has 6 unused bits, we can exploit these bits to encode the power states. In this case, the instruction length would not increase, and there is no additional power overhead. (2) If there are no unused bits in the instruction format, we can extend the instruction length by 6 bits to encode the power states. However, this incurs additional storage in the GPU pipeline, such as instruction buffers overhead. We measure the additional overhead using GPUWattch framework by increasing the instruction length by 8 bits (2-bit padding for byte alignment). We observe that adding 8 bits to the instruction has $< 0.0001\%$ area overhead and $< 0.005\%$ leakage power overhead in each SM.

As discussed in Section 4.2.4, we are required to modify scoreboard unit in the scheduler unit to keep track of the read after read dependencies. Currently, GPUWattch does not support a power model for scoreboard unit. However, depending on the following design choices we may require additional overheads. (1) If the power model for scoreboard uses a bit mask to keep track of the registers accessed by a warp, then we do not require any additional storage overhead. We can use the existing bit mask to set the registers that will be read by a warp. (2) Instead, if the scoreboard explicitly maintains the register numbers accessed by each warp, then we need to store up to 4 source register numbers of an instruction. If each SM allows W resident warps, and has R registers per each thread, then the

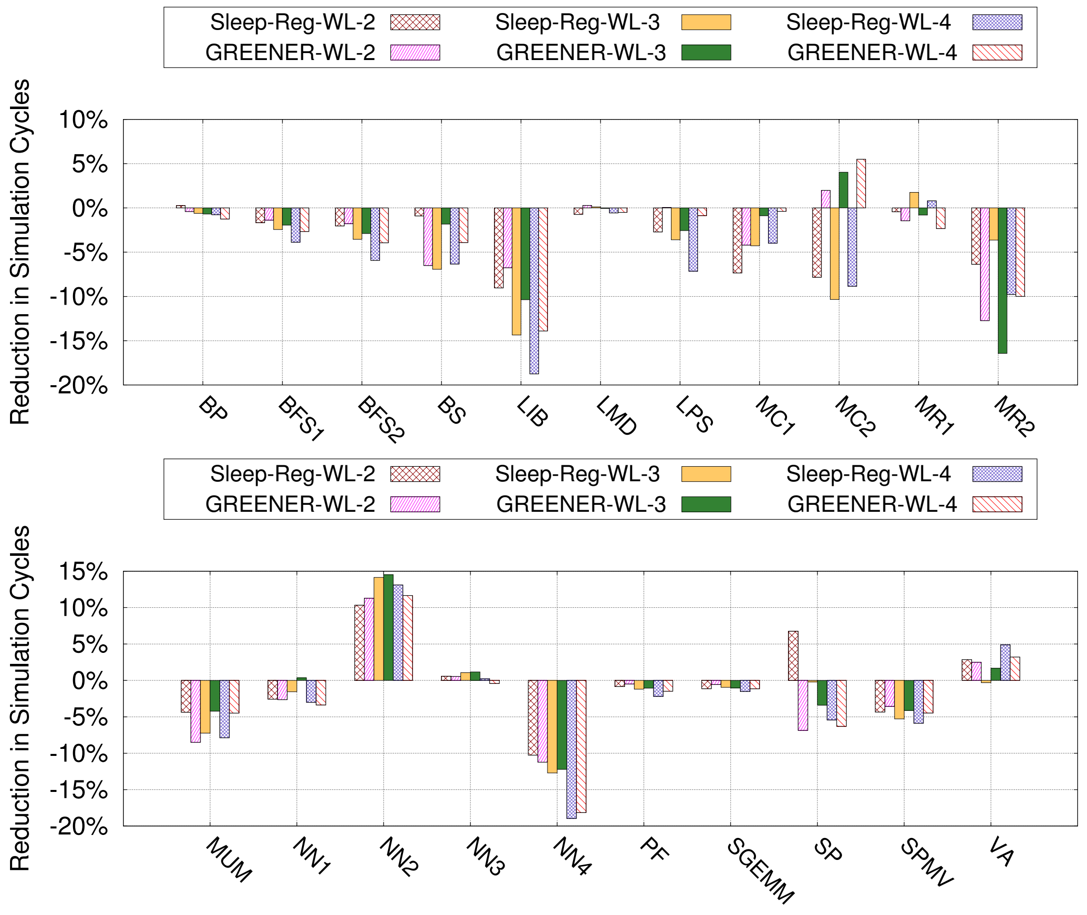


Figure 4.10: Comparing Performance Overhead for Various Wake Up Latencies

additional storage overhead for this scheme is $4 * W * log_2(R)$. For the configuration used in our experiments (i.e., W=64, R=64), the storage overhead is 192 bytes, which is $< 0.1\%$ of register file size. Similarly, to support the run-time optimization, we require a look up table. For our experiments, the additional storage required for lookup table is 1280 bytes ($< 1\%$ of the register file size).

4.3.6 Effect of Wake up Latency

Figure 4.10 compares performance overhead of **GReEneR** and *Sleep-Reg* with the *Baseline* for different values of wake up latencies. In the figure, **GReEneR**-WL-X (X $\in \{2,3,4\}$) denotes the **GReEneR** approach, which considers the wake up latency to change a register state from SLEEP to ON to be X cycles. Whereas,

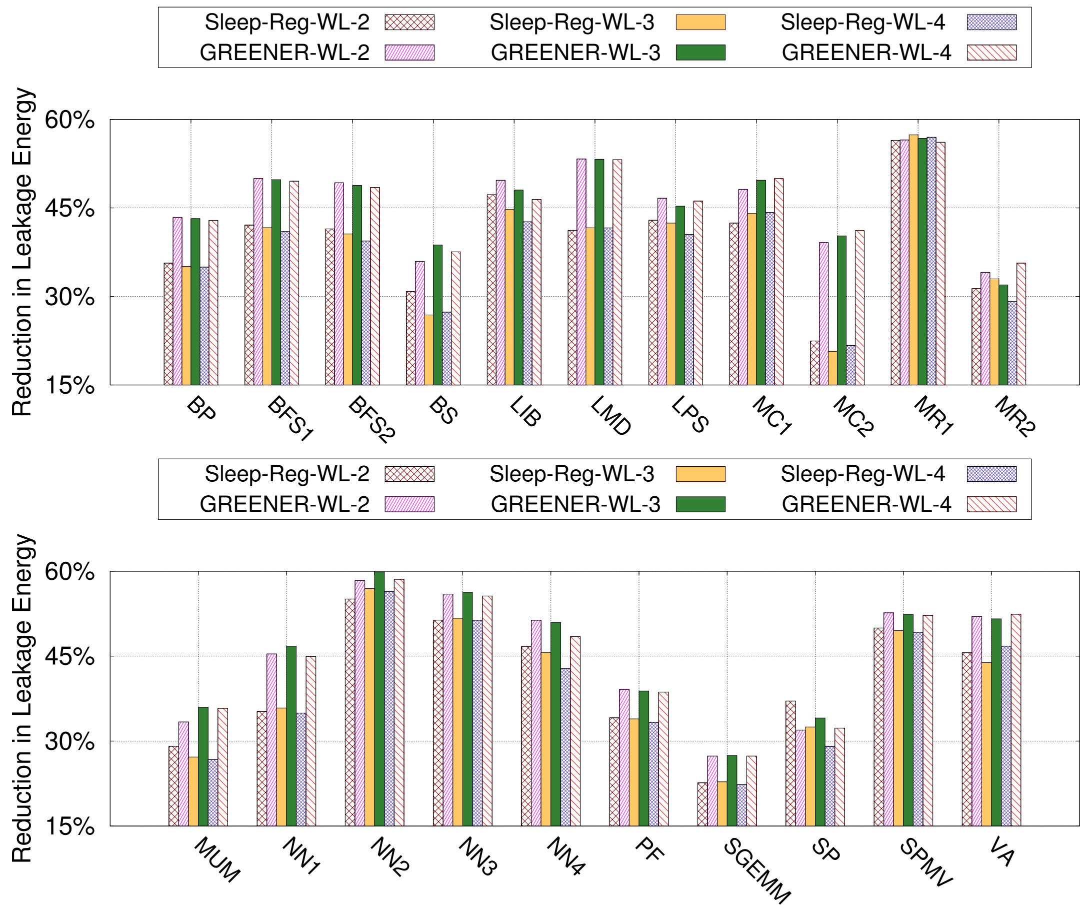


Figure 4.11: Comparing the Leakage Energy for Various Wake Up Latencies

when a register state needs to be changed from OFF to ON, it considers the latency to be 2X cycles. We use the similar notation for *Sleep-Reg* as well.

For most of the applications, **GREENER** and *Sleep-Reg* show performance degradation with the increase in the wake up latency. The increase in the overhead is expected because applications spend additional simulation cycles for changing register's state from OFF or SLEEP state to ON state. Hence, with the increase in the wake up latency, these additional simulation cycles will increase. Interestingly, some applications (*MC1 and MC2*) show performance improvement with the increase in the wake up latency. This is because, as discussed in Section 4.3.2, with the addition of wake up latency, the warps in the SM can get issued in different order, which can change the number of L1-cache misses, L2-cache misses, and stall cycles

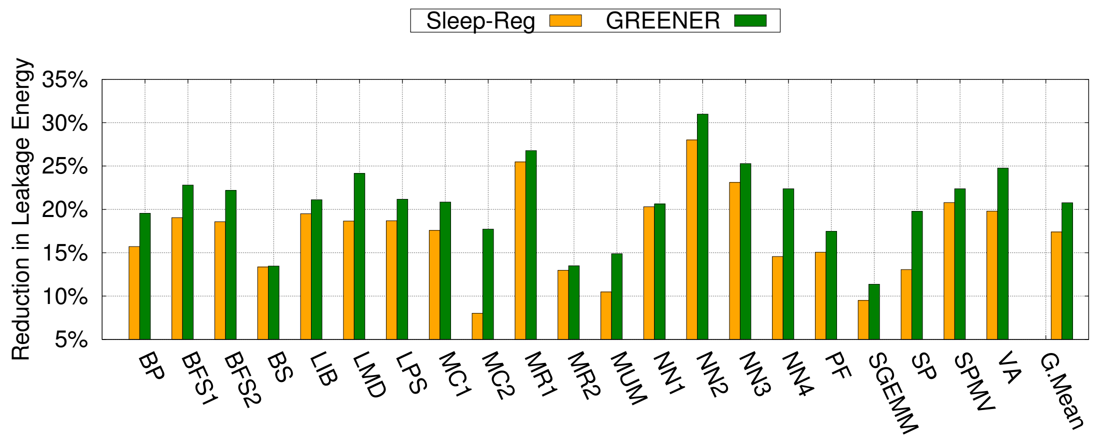


Figure 4.12: Comparing Leakage Energy by Including Routing Energy

etc. For MC1 and MC2, we find that the number pipeline stall cycles decrease with increase in the wake up latencies. Similarly, for *NN2* we observe more number of L1 misses with **GREENER** when used with wake latency 2 cycles than that of 3 cycles, hence **GREENER** performs better with wake up latency 3 cycles. Also, for *NN2*, **GREENER** performs better than *Baseline* for all wake up latencies due to a decrease in the L1 misses when compared to *Baseline*. In case of *SP*, **GREENER**-WL-2 has more stall cycles when compared to **GREENER**-WL-3. Further, for most of the applications **GREENER** performs better than *Sleep-Reg* with various wake up latencies.

We also compare the energy savings by varying the wake up latencies as shown in Figure 4.11. The results indicate that even with varying the wake up latency, the applications show significant reduction in the leakage energy when compared to *Baseline*. Also, the applications show more energy savings with **GREENER** when compared to *Sleep-Reg* for all wake up latencies.

4.3.7 Leakage Energy Savings with Routing

So far we discussed the energy efficiency of registers in a register file, however, GPUs also consume energy for routing of data and address through the register file. While modeling the register file, McPAT uses H-Tree distribution network to route data and address [62]. The H-Tree network spends a constant amount of leakage power, and various organizations can be exploited to reduce this power and to meet routing

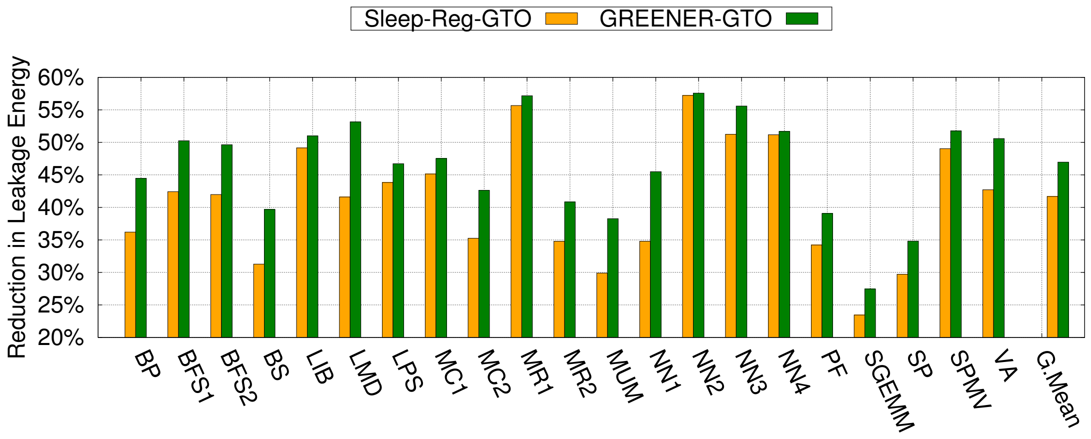


Figure 4.13: Comparing Leakage Energy with GTO Scheduler

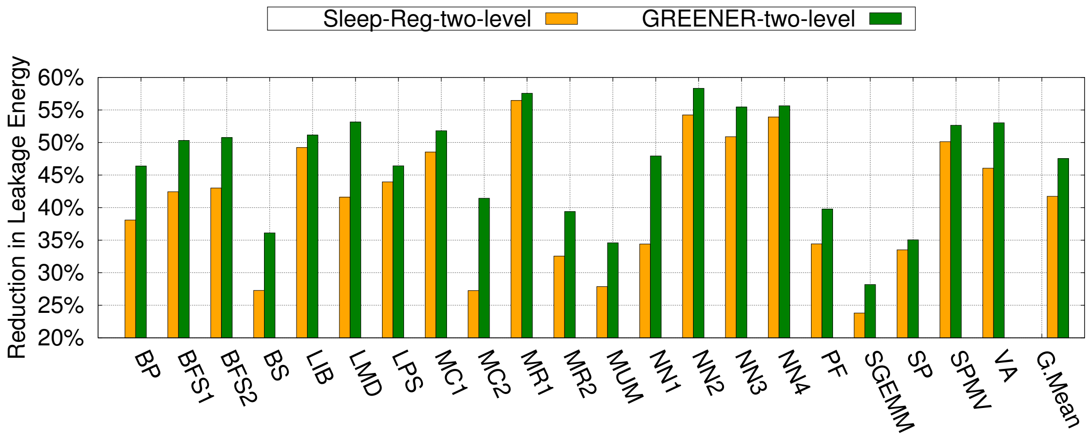


Figure 4.14: Comparing Leakage Energy with Two-Level Scheduler

requirement [18]. Our work focuses only on reducing the leakage power of memory cells of the register file by analyzing the register access patterns, and reducing the routing power is not in this scope. However, we show the effectiveness of **GReEneR** by including the constant routing energy as shown in Figure 4.12. From the results, we observe that **GReEneR** reduces the leakage energy on an average by 20.76% when compared to *Baseline*, which is more than that of *Sleep-Reg* (17.4%). However, the energy savings when including the routing energy are reduced when compared to that of results in Figure 4.8 because **GReEneR** does not provide any mechanism to optimize the routing power, hence the routing power remains unaffected.

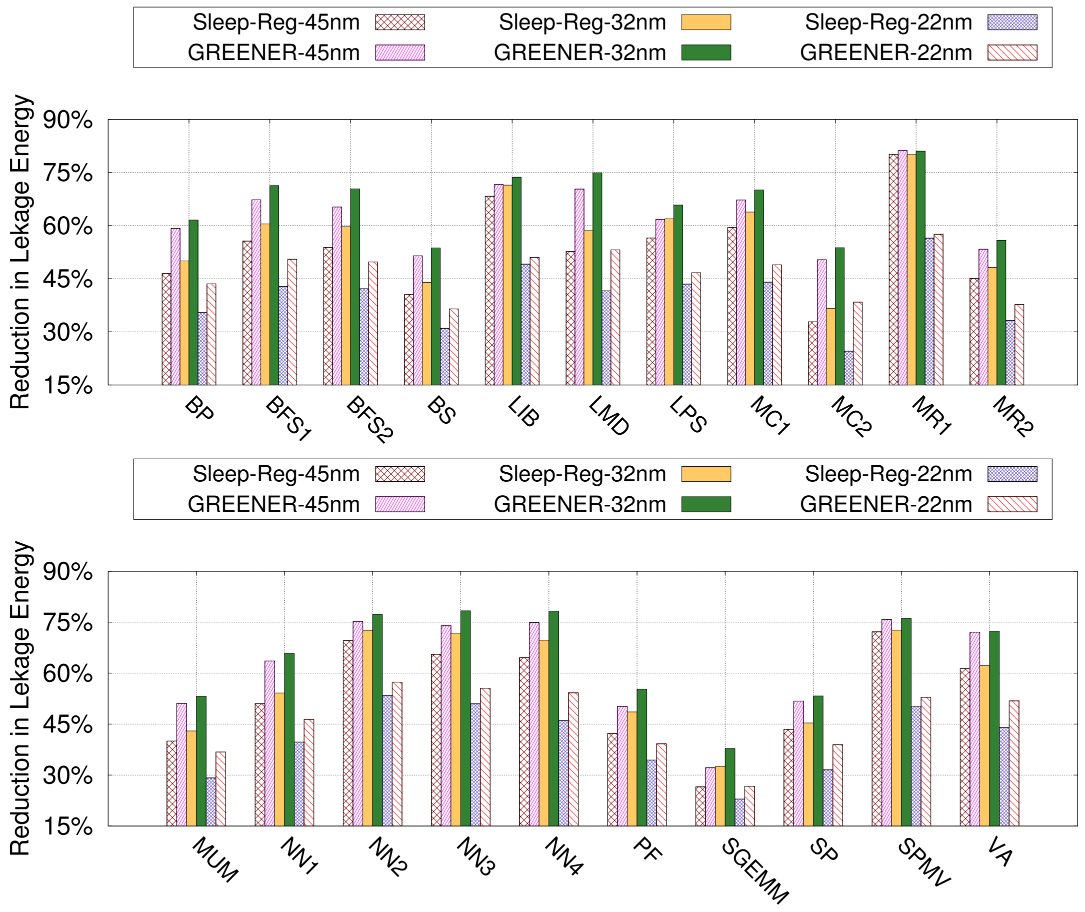


Figure 4.15: Comparing the Leakage Energy for Different Technology Configurations

4.3.8 Leakage Energy Savings with Different Schedulers

Figure 4.13 and 4.14 show the effectiveness of **GREENER** when it is evaluated with GTO and two-level scheduling policies respectively. The figures compare **GREENER** and *Sleep-Reg* with *Baseline* by measuring the reduction in leakage energy for the corresponding scheduling policies. The results show that **GREENER**-GTO and **GREENER**-two-level achieve an average reduction leakage energy by 46.96% and 47.54% with respect to *Baseline-GTO* and *Baseline-two-level* respectively. With different scheduling policies, the warps in the SM have different interleaving patterns, which affect the distance between the two consecutive accesses to a register. Even with the change in these access patterns, **GREENER** shows reduction in leakage energy when compared to *Baseline* and *Sleep-Reg*. We also find that *Baseline-GTO* performs better than *Baseline-two-level* in terms of simulation cycles, hence *Baseline-GTO* relatively consumes less leakage energy when compared to *Baseline-two-level*. However, the average energy savings of **GReEneR** are not affected significantly even with change in the scheduler.

4.3.9 Leakage Energy with Various Technologies

Figure 4.15 shows the effectiveness of **GReEneR** for various technology parameters (45nm, 32nm, and 22nm). The results show a significant reduction in leakage energy with **GReEneR** for all the applications even for various technology nodes. Further, it reduces the energy when compared to *Sleep-Reg*. With transition of technology from 45nm to 32nm, we observe an increase in the leakage energy for the *Baseline* approach, but **GReEneR** shows an increase in the leakage energy savings even with the transition. To model 22nm technology node, McPAT uses double gated technology to reduce the amount of leakage power, even with the advances in technology, **GReEneR** shows a reduction in leakage power when compared to *Baseline*. To summarize, architectural techniques help in reducing the leakage power of a register file, in addition, the knowledge of register access patterns and compiler optimizations further help in reducing the leakage power and energy.

4.3.10 Comparing Leakage Energy for Various Threshold Distances

Table 4.5 shows the effect of threshold distance on register leakage energy for **GReEneR**. The results are collected by varying the threshold distance from 1 to 21. The table shows that keeping the threshold distance to 1 does not benefit with respect to energy consumption, because with the shorter threshold distance, the registers of an instruction are turned to SLEEP or OFF state need to be woken up soon, which leads to increase in the simulation cycles. Whereas, having a high threshold distance also does not help in minimizing energy for all applications, because with longer distance, a register can be turned into SLEEP or OFF only when it is accessed after longer distance, this will result in losing the opportunity to save leakage energy. Hence, having the threshold distance at intermediate level

Table 4.5: Comparing Register Leakage Energy by Varying Threshold Distance

Benchmark/ Threshold	1	3	5	7	9	11	13	15	17	19	21
BP	308.5	308.48	310.53	311.78	311.29	314.01	310.56	313.27	313.27	311.78	311.78
BFS1	52.59	52.03	52.15	52.15	52.15	52.15	52.15	52.15	52.15	52.15	52.15
BFS2	19.92	19.91	19.92	19.92	19.92	19.92	19.92	19.92	19.92	19.92	19.92
BS	2298.66	2283.88	2210.96	2249.95	2231.68	2222.5	2324.12	2231.23	2231.23	2231.23	2185.78
LIB	4055.96	4015.81	3987.68	3986.05	3986.02	3979.52	3970.48	3986.87	3981.64	3991.05	3993.07
LMD	2.99	2.97	2.97	2.97	2.97	2.97	2.97	2.97	2.97	2.97	2.97
LPS	248.26	245.55	248.81	246.51	249.65	247.68	247.68	247.69	247.7	244.8	244.81
MC1	31.58	30.85	30.34	31.65	30.66	31.85	31.85	31.85	31.85	31.85	31.85
MC2	3949.6	3975.91	3978.84	3974.95	3970.92	3966.58	4008.73	4008.74	4008.76	4008.76	4008.76
MR1	2.12	2.08	2.09	2.09	2.09	2.09	2.09	2.09	2.09	2.09	2.09
MR2	2938.98	3030.98	2944.6	2809.69	2809.69	2809.69	2893.19	2903.46	2903.46	2839.47	2848.97
MUM	337.54	335.58	339.29	332.98	341.28	338.97	340.3	335.59	335.59	339.94	340.53
NN1	69.12	67.48	67.44	69.32	66.25	68.31	67.72	67.72	69.03	69.03	69.03
NN2	236.49	231.07	231.28	234.39	231.44	228.27	224.24	234.73	225.9	225.9	225.9
NN3	2560.2	2539.9	2536.1	2532.21	2517.54	2503.09	2498.04	2502.43	2504.65	2504.65	2504.65
NN4	27.09	26.8	26.83	26.7	26.2	28.25	26.69	26.6	26.41	26.41	26.41
PF	273.22	274.27	275.43	276	275.59	275.8	275.82	276.73	276.73	278.27	278.27
SGEMM	5672.12	5660.97	5692.81	5680.86	5684.16	5670.51	5677.64	5672.13	5724.88	5730.11	5730.11
SP	265.82	270.1	259.65	264.32	250.81	287.31	277.71	277.71	277.71	277.71	277.71
SPMV	76.11	75.62	75.26	75.44	75.45	75.85	75.87	76.15	76.28	76.15	76.15
VA	10.85	11.13	11.13	11.13	11.13	11.13	11.13	11.13	11.13	11.13	11.13

can achieve more energy savings.

Consider the application *SGEMM*. It achieves lowest energy at a threshold distance of 3. Further, with an increasing in the distance from 1 to 3, the leakage energy tends to decrease, and with an increasing in the distance beyond 3 tends to increase the energy. The similar behavior can be observed with *BP* as well. However, some applications like *LIB, MR2, NN1, NN3* and *SP* achieve their minimum energy at a threshold distance other than 3. For applications like *BFS1*, the leakage energy does not change beyond particular threshold distance, because a register can be turned into SLEEP or OFF state only up to a certain distance, beyond that the register must be kept in the ON the state even with increasing the threshold distance.

Finally, we chose the threshold distance of 3 for the experiments in the paper, which achieves lowest energy for maximum number of applications. However, this is not restrictive and can be reconfigured at compile time depending on application behavior.

4.4 Summary

This chapter presents a system called **GReEneR** to minimize the leakage energy of register file in the GPUs. It employs a compiler analysis and presents a run-time optimization that help in turning the registers into low power states by analyzing register access patterns. We implemented the proposed system in GPGPU-Sim simulator and evaluated them on several kernels from CUDASDK, GPGPU-SIM, Parboil, and Rodinia benchmark suites. We achieved an average reduction of register leakage energy by 46.96% with a negligible performance overhead when compared to baseline approach.

Chapter 5

Related Work

GPUs are widely adopted for various general purpose applications [27, 33, 41, 66] due to their compute capabilities. Therefore, improving GPU performance and energy efficiency have become the crucial factors of GPU design in the recent years. This thesis focuses on these two problems and proposes hardware and software solutions by managing the on-chip resources effectively. To improve GPU performance, we propose a resource sharing approach that improves the throughput by minimizing the register and scratchpad memory underutilization. To improve the performance further, we propose compiler optimizations that increase the availability of scratchpad memory that is associated with scratchpad sharing. To reduce the leakage energy of GPU register file, we propose **GReEneR** that analyzes the register access patterns and turns the registers into low power states. We discuss several other efforts that focus on achieving these two goals below.

5.1 Improving GPU Performance

GPUs maintain large number of resources, such as registers, scratchpad memory, cache, schedulers, and DRAM to increase the TLP. However, the throughput achieved by the GPUs depends on various factors, such as effective resource utilization and scheduling polices at several stages of the GPU pipeline. In the recent years, several hardware and software techniques were proposed to improve the performance by exploiting various GPU resources and analyzing resource access patterns. We discuss some of these techniques below.

Register and Scratchpad Memory Management in GPUs: Shared memory multiplexing [96] technique comes closest to our resource sharing approach. It provides software and hardware solutions to address the TLP problem caused by limited shared memory. The software approach combines two thread blocks into a single virtual thread block. The two thread blocks in a virtual block can execute instructions in parallel, as long as they do not access shared memory; and become serial when they need to access shared memory. The paper also describes a mechanism (called CO-VTB) that divides the shared memory into private and public part so that the thread blocks in a virtual block can access the private part in parallel and the public part in serial. However, CO-VTB has a high overhead of partitioning the data into private and public part, and is not suitable for all workloads. Also, they need to generate the code manually. The paper also gives a hardware solution to dynamically allocate and deallocate scratchpad memory using the existing barrier instruction. Again, these instructions need to be inserted manually in the code, and nesting of the barrier instructions is not allowed in order to avoid any deadlocks.

In contrast, resource sharing is a hardware solution that allows launching additional thread blocks in each SM. These additional thread blocks use the wasted scratchpad memory, and also share part of the allocated scratchpad memory with other resident thread blocks. The additional thread blocks launched in our approach make progress as long as they do not require shared scratchpad memory, and wait until the shared scratchpad is released by other thread blocks. Our compiler optimizations are fully automatic—the compiler analysis automatically identifies the regions of the shared and unshared scratchpad memory and inserts instruction to release the shared scratchpad as early as possible. Even in the presence of barrier instructions, our approach can not have deadlocks. In addition, we propose a warp scheduling mechanism that effectively schedule these additional warps to hide the long latencies in a better way. Section 3.4.1.10 compares our work with theirs quantitatively, on the same benchmarks.

Warp level divergence [92] improves the TLP by minimizing register underutilization. It launches one additional partial thread block when there are insufficient number of registers for an entire thread block. However, the number of warps in the partial thread block is decided by the number of unutilized registers, and also the partial thread block does not share registers with any other thread blocks. The unified storage approach [26] allocates the resources of SM (such as registers, scratchpad memory, and cache) dynamically as per the application demand. The patented register management [90] uses the concept of virtual registers, which are more than the actual physical registers, and hence can launch more thread blocks than allowed by physical registers. Our compiler optimizations can help in early release of unused registers with this approach. Gomez-Luna et al. [28] describe a mechanism to lock and unlock parts of scratchpad memory. We can reutilize the existing mechanism by defining a custom hash function that maps shared scratchpad memory regions to corresponding lock addresses. For unshared scratch region, the access can be given directly.

Virtual Thread [97] improves the GPU performance when the number of resident thread blocks is limited by the scheduling constraint. In other words, it aims to increase the TLP when the number of resident blocks is limited by the maximum number of resident threads or the maximum number of thread blocks that can fit in an SM. Whereas, our resource approach improves the TLP when the number of resident thread blocks are limited by the on-chip registers or scratchpad memory (denoted as capacity limit in their paper). Both the approaches are complementary to each other – we can improve the performance and the resource utilization of GPUs further when both the approaches are integrated. Kayiran et al. [46] propose a dynamic algorithm to launch the optimal number of thread blocks in an SM to reduce the resource contention. We can combine their techniques with our approach to reduce the increase in the stall cycles that occur with shared thread blocks.

Compiler Optimizations for Efficient Resource Utilization in GPUs: Ma et al. [68] proposed an algorithm for shared memory allocation using integer programming framework. It improves the performance by maximizing the access to shared memory and minimizing access to the device memory. Hayes et al. [32] proposed an on-chip memory allocation scheme for efficient utilization of GPU resources. It alleviates register pressure by spilling registers to scratchpad memory instead of local memory. CRAT [94] introduces a compile time coordinated register allocation scheme to minimize the cost of spilling registers. These schemes do not propose any architectural change to GPUs and are orthogonal to our resource sharing approach. RegMutex [48] presents a compiler-microarchitecture design to time multiplex the registers between warps. It divides the register file into base register set and extended register set. The registers in the base register set are allocated to the resident warps exclusively, whereas the extended registers are allocated based on compiler directed instructions. Their approach, similar to our register sharing, helps in increasing the warps residency. Whereas, we provide additional optimizations that manage the additional warps in an effective manner.

Scheduling Techniques to Improve GPU Performance: The two level warp scheduling algorithm [75] partitions the resident warps into groups and schedules the warps in each group according to LRR policy. CAWS [58] hides the long execution latencies by scheduling critical warps more frequently than other than warps. However, it requires the knowledge of critical warps. To address this problem, CAWA [57] identifies the critical warps at run-time by monitoring the number of instructions and the number of stall cycles. Further, it accelerates the critical warps in the SM using a greedy based critical warp scheduling algorithm. OWL [42] provides a scheduling mechanism to reduce cache contention and to improve DRAM bank level parallelism. Lee et al. [55] propose a lazy thread block scheduling mechanism to reduce the resource contention. In addition, they propose a block level CTA scheduling policy that allocates consecutive CTAs into the same SM to exploit cache locality.

Improving GPU Performance through Memory Management: Several approaches exploit memory hierarchy to improve the performance of GPU applications. Li et al. [60] proposed compiler techniques to efficiently place data onto registers, scratchpad memory, and global memory by analyzing data access patterns. Mascar [88] provides a scheduling policy to improve the performance of memory intensive workloads. It detects memory saturation events and prioritizes the memory requests of a single warp to improve cache hit rate. Priority based cache allocation [61] addresses the cache contention problem which occurs due to increased number of resident threads in an SM. Their approach is alternative to the thread throttling techniques [46, 82, 83].

Improving GPU Performance by Handling Warp Divergence: Dynamic warp formation [24] addresses the limited thread level parallelism that is present due to branch divergence. It dynamically forms new warps based on branch target condition. However, the performance of this approach is limited by the warp scheduling policy. Thread block compaction [23] addresses the limitation of dynamic warp formation that occurs when the new warps require more number of memory accesses. It regroups the new warps at the reconverging points. However in their solution, warps need to wait for other warps to reach the divergent path. Anantpur et al. [12] proposed linearization technique to avoid duplicate execution of instructions that occurs due to branch divergence in GPUs. Similarly, other hardware and software techniques [17, 20, 30, 69, 81] were proposed to handle branch and thread divergence, and these are orthogonal to our approach.

Miscellaneous: Warped pre-execution [49] accelerates a single warp by executing independent instructions when a warp is stalled due to long latency instruction. Further, it improves the GPU performance by hiding the long latency cycles in a better way. Baskaran et al. [15] proposed a compiler framework for optimizing memory access in affine loops. [29, 37] show that several applications are improved by using scratchpad memory instead of using global memory. Li et al. [64] propose a resource virtualization scheme for sharing of GPU resources with multiprocessors. The virtualization layer proposed by them helps in improving the performance by overlapping multiple kernels executions.

5.2 Improving Energy Efficiency

Leakage and dynamic power are the two major sources of power dissipation in CMOS technology. Reducing the leakage and dynamic power has been well studied in the context of CPUs when compared to GPUs. Though **GReEneR** is only for saving leakage power consumption of GPU register files, we describe briefly the techniques to save leakage and dynamic power in the context of both CPUs as well as GPUs. In addition, we discuss the power models that measure the power dissipation for CPU and GPU components. A comprehensive list of architectural techniques to reduce leakage and dynamic power of CPUs are described in [45]. Mittal et al. [70] discuss the state of the art approaches for reducing the power consumption of CPU register file. A survey of methods to reduce GPU power is presented in [71].

CPU Leakage Power Saving Techniques: Powell et al. [79] proposed a state destroying technique, Gated-V_{dd}, to minimize the leakage power of SRAM cells by gating supply voltage. Several methods [44, 95, 98] leverage Gated-V_{dd} technique to reduce the leakage power of cache memory by turning off the inactive cache lines. However, these techniques cannot preserve the state of the memory cells. To maintain the state, Flautner et al. [22] proposed an architectural technique that reduces the leakage power by putting the cache lines into a drowsy state. Other approaches [36, 78] exploit this by using cache access patterns to put cache lines in the drowsy state. As expected, the leakage power savings in this (drowsy) approach are less when compared to Gated-V_{dd} approach.

GPU Leakage Power Saving Techniques: Warped register file [8] reduces leakage power of register files by putting the registers into the drowsy state immediately after accessing them. However, it does not take into account the register access pattern while turning the registers into low power states, hence it can have high overhead whenever there are frequent wake up signals to the drowsy registers. In contrast, **GReEneR** considers register access information and proposes compile-time and run-time optimizations to make the register file energy efficient.

Register file virtualization [38] reduces the register leakage power by reallocating unused registers to another warp. This uses additional meta instructions to turn off the unused registers. However, the meta instructions are inserted at every 18 instructions, which can cause a delay in turning off the registers. **GReEneR** encodes the power saving states of the registers in the same instruction, and hence the registers can be switched to low power state at the earliest. Their approach optimizes power for unused registers only, while **GReEneR** can put even a used register into low power state if the next use is far away in the execution.

Pilot register file [9] partitions the register file into fast and slow register files, and it allocates the registers into these parts depending on the frequency of the register usage. It uses compiler and profiling information to allocate the register into one of these parts. The partition of the registers is done statically. Therefore, if a register is accessed more frequently for some duration, and less frequently for other duration, then allocating the register to either of the partitions can make it less energy efficient. **GReEneR** changes power state during the execution, so it does not suffer from this drawback.

Dynamic Power Saving Techniques for CPU and GPU: In CPUs, dynamic voltage frequency scaling (DVFS) has been widely adopted at the system level [91], compiler level [35, 93], and hardware level [86] to reduce dynamic power consumption. In case of GPUs, equalizer [89] dynamically adjusts the core and memory frequencies depending on the application behavior and the user requirement (i.e., power or performance). Lee et al. [54] propose mechanisms to dynamically adjust the voltage and frequency values to improve the throughput of applications under the power constraints. GPUWattch [59] uses DVFS algorithm to reduce the dynamic power by adjusting the processor frequency depending on the number of stall cycles. Warped compression [56] exploits the register value similarity to reduce effective register file size to minimize the dynamic as well as leakage power. Gebhart et al. [25] propose two complementary techniques to reduce GPU energy. The hierarchical register file proposed by them reduces register file energy by replacing the single register file with a multilevel hierarchical register file. Further, they design a multi level scheduler that partitions warps to active and pending warps and propose mechanisms to schedule these warps to achieve energy efficiency. These techniques mainly focus on reducing the dynamic power of GPUs and are orthogonal to our approach.

Power Models: CACTI [18] is an analytical model for estimating area, power, cycle time of cache and memory units. CACTI-P [63] is an extension of CACTI that introduces power gating logic to minimize the leakage power consumption by adding sleep transistors. Wattch [16] and McPAT [62] are the frameworks for analyzing the CPU power dissipation at cycle level. However, these models lack the support for GPUs. Hong et al. [34] propose integrated power and performance model that supports GPU, but it does not support cycle level power simulation and cannot be configured for different architectures. GPUWattch [59] is built on the top of McPAT to measure the power and area for the GPU architectures that provides cycle level power estimation. This tool has been integrated with GPGPU-Sim simulator to collect the simulation statistics.

Miscellaneous: Seth et al. [87] present algorithmic strategies for insertion of processor idle instructions at various points in the program such that the overall energy is reduced. LTRF [84] employs compiler optimizations to achieve low latency multilevel hierarchical register file while reducing the power consumption of the register file. RegLess [52] reduces the register storage space by using compiler annotations. Warped Gates [10] exploits the idle execution units to reduce the leakage power with a gating aware scheduling policy. This approach is complementary to **GReEneR** and it should be possible to combine the two techniques to further reduce leakage power.

Chapter 6

Conclusions and Future Work

This thesis deals with the two aspects of the GPU design: (1) improving throughput and (2) improving energy efficiency. To improve GPU performance, we propose resource sharing that exploits the on-chip GPU resources (registers and scratchpad memory) by managing them effectively. The key is to increase the number of resident thread blocks, consequently the thread level parallelism. This is achieved by reducing the resources' underutilization by sharing them among the resident thread blocks. To further improve the effectiveness of resource sharing, we propose three optimizations that manage the warps from the additional thread blocks effectively.

Experiments with various benchmarks help us to conclude that the additional thread blocks along with the optimizations help in hiding the long execution latencies, thus improving the throughput. When the number of resident thread blocks launched by an application is limited by registers, register sharing improves the performance. Similarly, if the number of resident thread blocks are limited by scratchpad memory, scratchpad sharing improves the performance. On the other hand, for other applications where the number of thread blocks is not limited by scratchpad or registers, the hardware changes do not negatively impact the run-time. We also observe that resource sharing will be more effective if programs start accessing the shared part of the resource as late as possible. This allows additional thread blocks to increase the TLP by executing more number of instructions before they start accessing shared resource.

Another important observation from the benchmarks was that the resources allocated to a thread block are released only after all the threads of a thread block finish their execution even though the resources are not accessed till the end of program execution. This mechanism not only affects of availability of resources but also reduces the amount of TLP. We propose compiler optimizations to improve the availability of the shared scratchpad memory that is associated with scratchpad sharing. The results of the experiments show that the compiler optimizations help in improving in the throughput by releasing the shared part of the scratchpad sooner.

The other part of the thesis focuses on reducing the leakage power of the register file in GPUs. We discuss various opportunities to save leakage power of the registers by analyzing the access patterns of the registers. We propose a system called **GReEneR** that employs compiler analysis to determines the power state of each register at each program point. To improve the effectiveness further, we introduce a run-time optimization that dynamically corrects the power states determined by the static analysis.

On evaluating **GReEneR** using several applications, we observed that the knowledge of register access patterns and the compiler optimizations help in improving the energy efficiency of register file with a negligible number of simulation cycles overhead. Further, we analyze that run-time optimization is effective when the power state estimated by the static analysis is sub-optimal at run-time.

6.1 Future Directions

In future, we can extend the resource sharing approach to improve the performance further. Also, the current research focuses only on managing the registers and scratchpad memory resources. In future, the research can be extended to other GPU resources, such as shared memory, cache, and DRAM. Several static/run-time techniques can be studied to improve GPU performance and energy efficiency by analyzing resource access patterns. We discuss few short-term and long-term research directions in this context below.

6.1.1 Short-term Research Directions

1. We can incorporate the traditional compiler analysis and optimizations into the resource sharing approach. For example, live range analysis [31, 80] along

with instruction reordering can be used to detect and release registers that are not used beyond a point. Such registers, if shared, can be used by the warp in the other thread block waiting for shared registers.

2. To improve resource sharing approach further, we can study the effect of various techniques such as, increasing the number of registers per thread, allocating temporary variables into available resources, and applying cache replacement policies.

3. Currently, we could not find a kernel whose number of resident thread blocks are limited by both registers and scratchpad memory. However, in future, we can combine both register sharing and scratchpad sharing to improve performance of applications that are limited by both the resources.

4. We can study various techniques for register allocation and reducing the register bank conflicts that not only help in improving the GPU performance but also help in reducing the register file leakage energy.

6.1.2 Long-term Research Directions

1. **Cache and Memory:** In GPUs, the pattern in which memory locations accessed by different threads can affect the performance [88]. For instance, coalesced memory access pattern yields better performance results than uncoalesced access patterns [11, 51]. Moreover, changes in the memory access pattern also affect the number of L1 and L2 cache misses due to change in spatial and temporal locality [39, 40]. This can be addressed by analyzing the possible interleavings of memory accesses and incorporating compiler optimizations that can transform a given program to an optimized program, which not only improves performance but also achieves energy efficiency.

2. **Scheduling Mechanisms:** GPU features different schedulers at various stages of its pipeline, such as thread block scheduler [55], warp scheduler [58, 75], and DRAM scheduler [53]. Several scheduling algorithms were proposed, which schedule job requests by analyzing the application behavior (i.e., memory

bound and computed bound etc.) to improve performance [83, 88]. In contrast, we can develop scheduling algorithms that can achieve both performance improvement and energy efficiency regardless of application behavior.

3. **Exploiting Execution Units for Energy:** GPU maintains large number of execution units such as ALUs, SFUs, and Load/Store units [4]. However, the execution units may not be active throughout the entire execution of an application, as a result, they dissipate a significant amount of power [10, 59]. This can be addressed by using runtime techniques that reduce power consumption of execution units.

4. **CPU-GPU Heterogeneous Architectures:** Stand-alone CPUs and GPUs have their own advantages in improving the performance of various types of applications depending on their behavior. For instance, applications that require more data transfer time compared to execution time can benefit from CPUs rather than GPUs. Similarly, the applications that expose more parallelism and has less branch divergence can benefit from GPUs instead of CPUs.

   To exploit both these processing units, researchers have investigated in collaborating CPU and GPUs. Several architectural, compiler and algorithmic strategies are proposed that utilize both CPU and GPUs to further improve performance, resource utilization, and energy efficiency. Mittal et al. [72] discuss the state-of-the-art techniques for CPU-GPU heterogeneous computing systems that focus on these issues. We plan to explore this area to study the impact of various resource management techniques on performance and energy.

References

[1] CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[2] CUDA-SDK. http://docs.nvidia.com/cuda/cuda-samples.

[3] GPGPU-Sim. http://www.gpgpu-sim.org.

[4] Kepler Architecture. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[5] OpenCL. https://www.khronos.org/opencl/.

[6] Parallel Thread Execution. http://docs.nvidia.com/cuda/parallel-thread-execution/.

[7] Parboil Benchmarks. http://impact.crhc.illinois.edu/Parboil/parboil.aspx.

[8] Abdel-Majeed, M., and Annavaram, M. Warped Register File: A Power Efficient Register File for GPGPUs. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture* (2013), HPCA '13, IEEE Computer Society, pp. 412–423.

[9] Abdel-Majeed, M., Shafaei, A., Jeon, H., Pedram, M., and Annavaram, M. Pilot Register File: Energy Efficient Partitioned Register File for GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017* (2017), pp. 589–600.

[10] Abdel-Majeed, M., Wong, D., and Annavaram, M. Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), MICRO-46, ACM, pp. 111–122.

[11] Amilkanthwar, M., and Balachandran, S. CUPL: A Compile-time Uncoalesced Memory Access Pattern Locator for CUDA. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (2013), ICS '13, ACM, pp. 459–460.

[12] Anantpur, J., and Govindarajan, R. Taming Control Divergence in GPUs through Control Flow Linearization. In *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings* (2014), pp. 133–153.

[13] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. A View of the Parallel Computing Landscape. *Commun. ACM 52*, 10 (Oct. 2009), 56–67.

[14] Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., and Aamodt, T. M. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software* (April 2009), pp. 163–174.

[15] Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22Nd Annual International Conference on Supercomputing* (2008), ICS '08, ACM, pp. 225–234.

[16] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: A Framework for Architectural-level Power Analysis and imizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (2000), ISCA '00, ACM, pp. 83–94.

[17] Brunie, N., Collange, S., and Diamos, G. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012), ISCA '12, IEEE Computer Society, pp. 49–60.

[18] CACTI. `http://www.hpl.hp.com/research/cacti`.

[19] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization* (2009), IISWC '09, IEEE Computer Society, pp. 44–54.

[20] Diamos, G., Ashbaugh, B., Maiyuran, S., Kerr, A., Wu, H., and Yalamanchili, S. SIMD Re-convergence at Thread Frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), MICRO-44, ACM, pp. 477–488.

[21] Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (2010), PACT '10, ACM, pp. 353–364.

[22] Flautner, K., Kim, N. S., Martin, S., Blaauw, D., and Mudge, T. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *SIGARCH Comput. Archit. News 30*, 2 (May 2002), 148–157.

[23] Fung, W. W. L., and Aamodt, T. M. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture* (2011), HPCA '11, IEEE Computer Society, pp. 25–36.

[24] Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (2007), MICRO 40, IEEE Computer Society, pp. 407–420.

[25] Gebhart, M., Johnson, D. R., Tarjan, D., Keckler, S. W., Dally, W. J., Lindholm, E., and Skadron, K. A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors. *ACM Trans. Comput. Syst. 30*, 2 (Apr. 2012), 8:1–8:38.

[26] Gebhart, M., Keckler, S. W., Khailany, B., Krashinsky, R., and Dally, W. J. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), MICRO-45, IEEE Computer Society, pp. 96–106.

[27] GNU Linear Programming Kit. `https://www.gnu.org/software/glpk/`.

[28] Gomez-Luna, J., Gonzalez-Linares, J. M., Benavides Benitez, J. I., and Guil, N. Performance Modeling of Atomic Additions on GPU Scratchpad Memory. *IEEE Trans. Parallel Distrib. Syst. 24*, 11 (Nov. 2013), 2273–2282.

[29] Gutierrez, E., Romero, S., Trenas, M., and Zapata, E. Memory Locality Exploitation Strategies for FFT on the CUDA Architecture. In *International Meeting on High-Performance Computing for Computational Science* (2008), vol. 5336.

[30] Han, T. D., and Abdelrahman, T. S. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (2011), GPGPU-4, ACM, pp. 3:1–3:8.

[31] Harrison, W. H. Compiler Analysis of the Value Ranges for Variables. *IEEE Trans. Softw. Eng. 3*, 3 (May 1977), 243–250.

[32] Hayes, A. B., and Zhang, E. Z. Unified On-chip Memory Allocation for SIMT Architecture. In *Proceedings of the 28th ACM International Conference on Supercomputing* (2014), ICS '14, ACM, pp. 293–302.

[33] He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (2008), PACT '08, ACM, pp. 260–269.

[34] Hong, S., and Kim, H. An Integrated GPU Power and Performance Model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ISCA '10, ACM, pp. 280–289.

[35] Hsu, C.-H., and Kremer, U. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (2003), PLDI '03, ACM, pp. 38–48.

[36] Hu, J. S., Nadgir, A., Vijaykrishnan, N., Irwin, M. J., and Kandemir, M. Exploiting Program Hotspots and Code Sequentiality for Instruction Cache Leakage Management. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design* (2003), ISLPED '03, ACM, pp. 402–407.

[37] Huo, X., Ravi, V. T., Ma, W., and Agrawal, G. Approaches for parallelizing reductions on modern GPUs. In *2010 International Conference on High Performance Computing* (Dec 2010), pp. 1–10.

[38] Jeon, H., Ravi, G. S., Kim, N. S., and Annavaram, M. GPU Register File Virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48, ACM, pp. 420–432.

[39] Jia, W., Shaw, K. A., and Martonosi, M. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing* (2012), ICS '12, ACM, pp. 15–24.

[40] Jia, W., Shaw, K. A., and Martonosi, M. MRPB: Memory request prioritization for massively parallel processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (Feb 2014), pp. 272–283.

[41] Jiang, C., and Snir, M. Automatic Tuning Matrix Multiplication Performance on Graphics Hardware. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques* (2005), PACT '05, IEEE Computer Society, pp. 185–196.

[42] Jog, A., Kayiran, O., Chidambaram Nachiappan, N., Mishra, A. K., Kandemir, M. T., Mutlu, O., Iyer, R., and Das, C. R. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, ACM, pp. 395–406.

[43] Kam, J. B., and Ullman, J. D. Global Data Flow Analysis and Iterative Algorithms. *J. ACM 23*, 1 (Jan. 1976), 158–171.

[44] Kaxiras, S., Hu, Z., and Martonosi, M. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture* (2001), ISCA, ACM, pp. 240–251.

[45] Kaxiras, S., and Martonosi, M. *Computer Architecture Techniques for Power-Efficiency*, 1st ed. Morgan and Claypool publishers, 2008.

[46] Kayiran, O., Jog, A., Kandemir, M. T., and Das, C. R. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques* (2013), PACT '13, IEEE Press, pp. 157–166.

[47] Khedker, U., Sanyal, A., and Karkare, B. *Data Flow Analysis: Theory and Practice*, 1st ed. CRC Press, Inc., 2009.

[48] Khorasani, F., Esfeden, H. A., Farmahini-Farahani, A., Jayasena, N., and Sarkar, V. RegMutex: Inter-Warp GPU Register Time-Sharing. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (June 2018), pp. 816–828.

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

[50] Kim, N. S., Austin, T., Blaauw, D., Mudge, T., Flautner, K., Hu, J. S., Irwin, M. J., Kandemir, M., and Narayanan, V. Leakage Current: Moore's Law Meets Static Power. *Computer 36*, 12 (Dec. 2003), 68–75.

[51] Kim, Y., and Shrivastava, A. CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA. In *Proceedings of the 48th Design Automation Conference* (2011), DAC '11, ACM, pp. 128–133.

[52] Kloosterman, J., Beaumont, J., Jamshidi, D. A., Bailey, J., Mudge, T., and Mahlke, S. Regless: Just-in-time Operand Staging for GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017), MICRO-50 '17, ACM, pp. 151–164.

[53] Lakshminarayana, N. B., Lee, J., Kim, H., and Shin, J. DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function. *Computer Architecture Letters 11*, 2 (2012), 33–36.

[54] Lee, J., Sathisha, V., Schulte, M., Compton, K., and Kim, N. S. Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (2011), PACT '11, IEEE Computer Society, pp. 111–120.

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

[56] Lee, S., Kim, K., Koo, G., Jeon, H., Ro, W. W., and Annavaram, M. Warped-compression: Enabling Power Efficient GPUs Through Register Compression. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (2015), ISCA '15, ACM, pp. 502–514.

[57] Lee, S.-Y., Arunkumar, A., and Wu, C.-J. CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (2015), ISCA '15, ACM, pp. 515–527.

[58] Lee, S.-Y., and Wu, C.-J. CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (2014), PACT '14, ACM, pp. 175–186.

[59] Leng, J., Hetherington, T., ElTantawy, A., Gilani, S., Kim, N. S., Aamodt, T. M., and Reddi, V. J. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ISCA '13, ACM, pp. 487–498.

[60] Li, C., Yang, Y., Lin, Z., and Zhou, H. Automatic Data Placement into GPU On-chip Memory Resources. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2015), CGO '15, IEEE Computer Society, pp. 23–33.

[61] Li, D., Rhu, M., Johnson, D. R., O'Connor, M., Erez, M., Burger, D., Fussell, D. S., and Redder, S. W. Priority-based cache allocation in throughput processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)* (Feb 2015), pp. 89–100.

[62] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. *ACM Trans. Archit. Code im. 10*, 1 (Apr. 2013), 5:1–5:29.

[63] Li, S., Chen, K., Ahn, J. H., Brockman, J. B., and Jouppi, N. P. CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *Proceedings of the International Conference on Computer-Aided Design* (2011), ICCAD '11, IEEE Press, pp. 694–701.

[64] Li, T., Narayana, V. K., El-Araby, E., and El-Ghazawi, T. GPU Resource Sharing and Virtualization on High Performance Computing Systems. In *Proceedings of the 2011 International Conference on Parallel Processing* (2011), ICPP '11, IEEE Computer Society, pp. 733–742.

[65] Lim, J., Lakshminarayana, N. B., Kim, H., Song, W., Yalamanchili, S., and Sung, W. Power Modeling for GPU Architectures Using McPAT. *ACM Trans. Des. Autom. Electron. Syst. 19*, 3 (June 2014), 26:1–26:24.

[66] Liu, W., Schmidt, B., Voss, G., and Muller-Wittig, W. Streaming Algorithms for Biological Sequence Alignment on GPUs. *IEEE Trans. Parallel Distrib. Syst. 18*, 9 (Sept. 2007), 1270–1281.

[67] Lucas, J., Lal, S., Andersch, M., Alvarez-Mesa, M., and Juurlink, B. How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) 00* (2013), 97–106.

[68] Ma, W., and Agrawal, G. An Integer Programming Framework for Optimizing Shared Memory Use on GPUs. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (2010), PACT '10, ACM, pp. 553–554.

[69] Meng, J., Tarjan, D., and Skadron, K. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ISCA '10, ACM, pp. 235–246.

[70] Mittal, S. A survey of techniques for designing and managing CPU register file. *Concurrency and Computation: Practice and Experience 29*, 4 (2017).

[71] Mittal, S., and Vetter, J. S. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Comput. Surv. 47*, 2 (Aug. 2014), 19:1–19:23.

[72] Mittal, S., and Vetter, J. S. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.* (July 2015).

[73] Moore, G. E. Cramming more Components onto Integrated Circuits. *Electronics 38*, 8 (April 1965).

[74] Muchnick, S. S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., 1997.

[75] Narasiman, V., Shebanow, M., Lee, C. J., Miftakhutdinov, R., Mutlu, O., and Patt, Y. N. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), MICRO-44, ACM, pp. 308–317.

[76] Nickolls, J., and Dally, W. J. The GPU Computing Era. *IEEE Micro 30*, 2 (Mar. 2010), 56–69.

[77] Open Graphics Library. `https://www.opengl.org/`.

[78] Petit, S., Sahuquillo, J., Such, J. M., and Kaeli, D. Exploiting Temporal Locality in Drowsy Cache Policies. In *Proceedings of the 2Nd Conference on Computing Frontiers* (2005), CF '05, ACM, pp. 371–377.

[79] Powell, M., Yang, S.-H., Falsafi, B., Roy, K., and Vijaykumar, T. N. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design* (2000), ISLPED '00, ACM, pp. 90–95.

[80] Quintao Pereira, F. M., Rodrigues, R. E., and Sperle Campos, V. H. A Fast and Low-overhead Technique to Secure Programs Against Integer Overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2013), CGO '13, IEEE Computer Society, pp. 1–11.

[81] Rhu, M., and Erez, M. CAPRI: Prediction of Compaction-adequacy for Handling Control-divergence in GPGPU Architectures. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012), ISCA '12, IEEE Computer Society, pp. 61–71.

[82] Rogers, T. G., O'Connor, M., and Aamodt, T. M. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), MICRO-45, IEEE Computer Society, pp. 72–83.

[83] Rogers, T. G., O'Connor, M., and Aamodt, T. M. Divergence-aware Warp Scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), MICRO-46, ACM, pp. 99–110.

[84] Sadrosadati, M., Mirhosseini, A., Ehsani, S. B., Sarbazi-Azad, H., Drumond, M., Falsafi, B., Ausavarungnirun, R., and Mutlu, O. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), ASPLOS '18, ACM, pp. 489–502.

[85] Sami, M., Sciuto, D., Silvano, C., Zaccaria, V., and Zafalon, R. Low-power data forwarding for VLIW embedded architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 10*, 5 (Oct 2002), 614–622.

[86] Semeraro, G., Albonesi, D. H., Dropsho, S. G., Magklis, G., Dwarkadas, S., and Scott, M. L. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture* (2002), MICRO 35, IEEE Computer Society Press, pp. 356–367.

[87] Seth, A., Keskar, R. B., and Venugopal, R. Algorithms for Energy imization Using Processor Instructions. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (2001), CASES '01, ACM, pp. 195–202.

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

[89] Sethia, A., and Mahlke, S. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), MICRO-47, IEEE Computer Society, pp. 647–658.

[90] Tarjan, D., and Skadron, K. On demand register allocation and deallocation for a multithreaded processor, June 30 2011. US Patent App. 12/649,238.

[91] Weiser, M., Welch, B., Demers, A., and Shenker, S. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation* (1994), OSDI '94, USENIX Association.

[92] Xiang, P., Yang, Y., and Zhou, H. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (Feb 2014), pp. 284–295.

[93] Xie, F., Martonosi, M., and Malik, S. Compile-time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (2003), PLDI '03, ACM, pp. 49–62.

[94] Xie, X., Liang, Y., Li, X., Wu, Y., Sun, G., Wang, T., and Fan, D. Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48, ACM, pp. 395–406.

[95] Yang, S.-H., Falsafi, B., Powell, M. D., Roy, K., and Vijaykumar, T. N. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture* (2001), HPCA, IEEE Computer Society, pp. 147–157.

[96] Yang, Y., Xiang, P., Mantor, M., Rubin, N., and Zhou, H. Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (2012), PACT '12, ACM, pp. 283–292.

[97] Yoon, M. K., Kim, K., Lee, S., Ro, W. W., and Annavaram, M. Virtual Thread: Maximizing Thread-level Parallelism Beyond GPU Schedul-

ing Limit. In *Proceedings of the 43rd International Symposium on Computer Architecture* (2016), ISCA '16, IEEE Press, pp. 609–621.

[98] Zhang, M., and Asanović, K. Fine-grain CAM-tag Cache Resizing Using Miss Tags. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design* (2002), ISLPED, ACM, pp. 130–135.

Publications

1. **Vishwesh Jatala**, Jayvant Anantpur, and Amey Karkare. Improving GPU Performance Through Resource Sharing. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), Kyoto, Japan, 2016, ACM, pp. 203-214.

2. **Vishwesh Jatala**, Jayvant Anantpur, and Amey Karkare. Resource Sharing for GPUs. In 14th International Symposium on Code Generation and Optimization (CGO, Poster Track), Barcelona, Spain, 2016.

3. **Vishwesh Jatala**, Jayvant Anantpur, and Amey Karkare. Scratchpad Sharing in GPUs. In ACM Transactions on Architecture and Code Optimization (TACO). 14, 2 (May 2017), 15:1-15:29.

4. **Vishwesh Jatala**, Jayvant Anantpur, and Amey Karkare. GREENER: A Tool for Improving Energy Efficiency of GPU Register File. In 24th IEEE International Conference on High Performance Computing, Data, and Analytics, Student Research Symposium (HiPC, SRS), Jaipur, India, 2017.

5. **Vishwesh Jatala**, Jayvant Anantpur, and Amey Karkare. Reducing GPU Register File Energy. In 24th International European Conference on Parallel and Distributed Computing (Euro-Par), Torino, Italy 2018.