POWER AND PERFORMANCE OPTIMIZATION IN GPGPU

A Dissertation

by

AHMAD MAHMOUD MESLEH RADAIDEH

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul Gratz
Committee Members,	Jiang Hu
	Ulisses Braga Neto
	Eun Kim
Head of Department,	Miroslav Begovic

May 2020

Major Subject: Computer Engineering

Copyright 2020 Ahmad Mahmoud Mesleh Radaideh

ABSTRACT

Thread parallel hardware, as the Graphics Processing Units (GPUs), greatly outperform CPUs in providing high compute throughput and memory bandwidth which make them ideal for accelerating various data-parallel applications. These hardware designs provide high performance computing by supporting a massive thread level parallelism (TLP) processing model. Our work focuses on making the thread parallel hardware more power and energy efficient and higher performance. It also focuses on making the simulation of this type of hardware more accurate. Our work is divided into three main parts: (1) We introduce a coalescing-aware register file organization that takes advantage of frequent narrow-width data present in general-purpose applications in order to increase performance and reduce energy consumption in GPU. We present a new design that is capable of combining read and write accesses originated from same or different warps into fewer accesses. Our design reduces the number of register file accesses by 30.5%, achieves IPC speedup of 16.5%, and reduces overall GPU energy by 32.2% on average. (2) We present a low-cost power saving scheme in GPU that dynamically exploits frequent zero data within and across registers in order to gate off register file reads and writes and execution units to reduce dynamic power without impacting performance. Our scheme reduces register file reads and writes on average by 50% and 54%, respectively. The register file and execution unit dynamic power are reduced on average by 27% and 19%, respectively. The reduction in total GPU dynamic power achieved is about 8% on average. (3) For multi-threaded applications, the results taken from full system architecture simulation can often be inconsistent, primarily because of a combination of small input sets and the behavior of the Linux thread scheduler. We propose a simple solution wherein the scheduler is modified to enforce mapping of software threads into available distinct processors that provides consistent runtimes for short-run, multi-thread benchmarks, leading to expected, consistent experimental results.

DEDICATION

To all my family with love.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Paul Gratz for his invaluable guidance and feedback that made this work successful. I would also like to thank my committee members, Dr. Eun Kim, Dr. Jiang Hu, and Dr. Ulisses Braga Neto, for their time and effort reviewing this work. Finally, I would like to thank all my family, my father, brothers, sisters, and my wife and daughter, for their endless love, encouragement, and support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of my advisor Dr. Paul Gratz, Dr. Jiang Hu, and Dr. Ulisses Braga Neto of the Department of Electrical and Computer Engineering and Dr. Eun Kim of the Department of Computer Science and Engineering.

All work conducted for the dissertation was completed by the student, Ahmad Radaideh, independently.

Funding Sources

Graduate study was supported in part by a tuition assistance program from Qualcomm Technologies, Inc.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

TABLE OF CONTENTS vi

LIST OF FIGURES ix

LIST OF TABLES xv

1. INTRODUCTION 1

1.1 Register File Access Coalescing in GPU 2
1.2 Exploiting Zero Data to Reduce Power Consumption in GPU 4
1.3 Architecture Simulation and the Impact of Linux Thread Scheduler 5
1.4 Dissertation Statement 8
1.5 Dissertation Outline 9

2. BACKGROUND 10

2.1 Modern GPU Architecture Model 10
2.1.1 CUDA Overview 10
2.1.2 GPU Chip Layout 11
2.1.3 Warp Scheduler 14
2.1.4 Register File 15
2.1.5 Execution Units 17
2.2 Full System Simulation 18

3. REGISTER FILE ACCESS COALESCING IN GPU 22

3.1 Introduction 22
3.2 Performance Impact of Limited Access Ports 22
3.3 Motivation 25
3.3.1 Register Operands Width 25
3.3.2 Register File Bandwidth 27
3.3.3 Warp Instruction Operands 28

3.4 Promoting Coalescing Opportunities ... 29
3.5 Related Work ... 32
3.5.1 Non-coalescing Techniques ... 32
3.5.2 Register Coalescing Techniques ... 34
3.6 Register File Access Coalescing Design ... 39
3.6.1 Design Overview ... 40
3.6.2 Coalescing-aware Register File Organization ... 43
3.6.2.1 Registers Layout (Register to Bank Mapping) ... 43
3.6.2.2 Register File Bank ... 44
3.6.2.3 Register Alignment ... 46
3.6.2.4 Dual-access Banks ... 49
3.6.2.5 Register File Bank Arbiter ... 52
3.6.2.6 Register File Interconnect ... 55
3.6.2.7 Operand Collector Write ... 58
3.6.2.8 Register Width Detection ... 59
3.6.2.9 Design Overhead ... 61
3.7 Evaluation ... 62
3.7.1 Methodology ... 62
3.7.2 Register File Access Reduction ... 65
3.7.3 Register File Bandwidth Increase ... 67
3.7.4 Register File Coalesced Access ... 70
3.7.5 IPC Performance Speedup ... 72
3.7.6 Dynamic Energy Reduction ... 73
3.7.7 Result Summary ... 75
3.8 Conclusion ... 75

4. EXPLOITING ZERO DATA TO REDUCE REGISTER FILE AND EXECUTION UNIT DYNAMIC POWER CONSUMPTION IN GPU ... 77

4.1 Introduction ... 77
4.2 Motivation ... 77
4.3 Reducing Register File Dynamic Power ... 80
4.3.1 Using the Thread Active Mask ... 81
4.3.2 Using the In-lane Zero Mask ... 82
4.3.3 Using the Cross-lane Zero Mask ... 84
4.3.4 Dynamic Zero Mask Selection ... 85
4.4 Reducing Execution Unit Dynamic Power ... 86
4.4.1 Using the Active Thread Mask ... 86
4.4.2 Using the Operand Zero Masks ... 88
4.5 Evaluation ... 89
4.5.1 Register File Power ... 90
4.5.2 Execution Units Power ... 92
4.5.3 GPGPU total Power ... 94
4.6 Related Work ... 95
4.7 Conclusion ... 97

5. MULTI-PROCESSOR FULL SYSTEM SIMULATION AND THE IMPACT OF LINUX THREAD SCHEDULER 98

5.1 Introduction 98
5.2 Behavior of Thread Scheduler in Full System Simulation 98
5.2.1 Thread Scheduling and Load Imbalance 98
5.2.2 Periodic Load balancing 100
5.2.3 Immediate Load balancing 101
5.3 Proposed Solution 102
5.4 Evaluation 104
5.5 Conclusion 109

6. CONCLUSION 110

REFERENCES 112

LIST OF FIGURES

FIGURE Page

1.1 Actual run-time of 12-thread Canneal benchmark with different input sets on 12-core hardware machine. 6

1.2 Performance speedup for Canneal benchmark using small input set, with 8, 12, 15, and 16 threads under memory speeds of 200MHz and 800MHz in full system simulation with original Linux scheduler. Results are normalized against an 8-thread 800MHz case. 7

2.1 C++ code example for vector addition (a) Serial code that typically runs on CPU. (b) CUDA thread-parallel version of the code that runs on GPU. 11

2.2 CUDA hierarchy of threads that maps to a hierarchy of processing elements on the GPU. 12

2.3 Modern GPU chip layout with 16 Streaming Multiprocessors (SMs), each of which has its own register file, instruction and data caches, and execution units. Reprinted from [1]. 13

2.4 Dual-warp scheduler used in Fermi GPU. Reprinted from [1]. 14

2.5 GPU main register file and execution pipeline. 15

2.6 A 128B warp register entry in one register file bank occupying four 32B sub-bank entries that have the same index. Each 32B sub-bank entry holds data for 8 threads within the warp. 16

2.7 Register-to-bank mapping (layout) with warp registers interleaved across register file banks. 16

2.8 Execution units in a GPU Streaming Multiprocessor (SM) core with 32 Streaming Processing Units (SPUs), 4 Special Functional Units (SFUs), and 16 memory Load/Store Units (LDSTs). Reprinted from [1]. 18

2.9 Full system simulation environment for a Core Multi-Processor (CMP) chip with 16 cores managed by a real Operating System (OS) running a thread scheduler. The full simulation system runs multi-threaded user applications similar to a real multi-core system with an OS. 19

3.1 A limit study on the potential IPC performance speedup of reducing register file banks port conflicts. 23

3.2 Examples of port conflict on register file bank and operand collector accesses (a) Bank conflict between two write requests. (b) Bank conflict between a read and a write request. (c) Bank conflict between two read requests. (d) Port conflict on two writes to an operand collector unit. 24

3.3 Width distribution of GPU register file warp accesses classified into two groups: accesses that require 4-byte per thread (full width) and accesses that require less than 4-byte per thread (narrow width) (a) source operands width distribution (b) destination operands width distribution. 26

3.4 Unused register file bandwidth on (a) register file banks access (b) collector units write access. 27

3.5 Percentage of the number of source register operands in warp instructions. A given warp instruction can have one, two, or three source register operands. ... 28

3.6 Examples of register coalescing on register file bank and operand collector accesses (a) Access coalescing of two write requests. (b) Access coalescing of a read and a write request. (c) Access coalescing of two read requests. (d) Access coalescing of two writes to an operand collector unit. 30

3.7 CORF design overview (a) limited CORF with read coalescing support within the same physical register entry (b) enhanced CORF++ with read coalescing support across two register entries within a bank. Reprinted from [2]. 35

3.8 GPU register file design used in CORF. Reprinted from [2]. 36

3.9 Our coalescing-aware GPU register file design. 40

3.10 Registers to banks mapping (register layouts): (a) all registers belonging to the same warp are mapped into one register file bank and (b) warp registers are interleaved across register file banks. 43

3.11 Data format of a 128B warp register within an RF bank entry: (a) Byte-interleaved format: each 32B sub-bank entry holds data for 8 threads in the warp and (b) Thread-interleaved format (supports register coalescing): each 32B sub-bank entry holds 1-byte (same byte number) for every thread in the warp. 45

3.12 Register data representation within the register file and operand collectors and outside. Switching data from one format to the other is done through wiring bytes into different byte-position (no logic cost). 45

3.13 Warp register data alignment: (a) Default right-alignment with byte 0 for all 32 threads map to sub-bank 0 and (b) Left alignment using intra-thread byte-swap MUX with byte 0 for all 32 threads map to sub-bank 3. 47

3.14 Data alignment of warp registers within a register file bank based on even/odd register entry number. Even registers are right aligned with byte 0 of all threads map to sub-bank 0. Odd registers are left aligned (byte swapped) with byte 0 of all threads map to sub-bank 3. 48

3.15 Comparison between basline register file bank and our coalescing-aware bank: (a) Baseline bank with single-access support (b) Our dual-access bank with left and right requests that can access two register entries with different data alignments in non-overlapping sub-banks. 50

3.16 Register file request matrix: (a) Request matrix for baseline arbiter. Up to four requests can be granted access at the same time (b) Request matrix for our coalescing-aware arbiter. Each bank can have left or right requests with each request having a 4-bit mask to indicate the sub-banks it needs to access. Up to eight requests (four coalesced requests) can be granted access at the same time. 53

3.17 A wrapped wavefront arbitration scheme (WWFA) used in GPU register file matrix arbiter. Four priority diagonals are used $P0$–$P3$ with a priority wave initially starting at $P0$ and propagating from one diagonal to the next every cycle. 54

3.18 Register file interconnect: (a) Baseline 4×4 crossbar with 128B ports that connects register file banks to the operand collector units. (b) New crossbar structure used in our design, which supports register coalescing at no extra cost, with four 4×4 crossbars each of which has narrow 32B ports and connects a particular sub-bank (from all four banks) to the operand collector units. 56

3.19 Coalescing-aware operand collector unit with 32B write ports. Coalesced read data is naturally unpacked into the destined source operands buffering space. 58

3.20 Width detection and sub-mask generation logic for a warp result in WB stage. The 4-bit sub-bank mask is saved in a buffer in a 2-bit encoded form along with the sign-bit. 60

3.21 Percentage of register file access reduction with a *wid_ layout* register file and using different register alignment schemes: fixed alignment based on register number (*reg_ alignment*), write interleaved alignment (*wr_ alignement*), and an ideal alignment (*ideal_ alignment*). 66

3.22 Percentage of operand collectors write reduction with a *wid_ layout* register file and using different register alignment schemes: fixed alignment based on register entry number (*reg_ alignment*), register interleaved alignment on writes (*wr_ alignement*), and an ideal alignment (*ideal_ alignment*). 66

3.23 Percentage of register file access reduction comparison between two different register file layouts: *wid_layout* and *wshift_layout* using two register alignment schemes: fixed alignment based on register number (*reg_ **) and write interleaved alignment (*wr_ **). 67

3.24 Percentage of register file bandwidth increase with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register number (*reg_alignment*), write interleaved alignment (*wr_alignement*), and an ideal alignment (*ideal_alignment*). 68

3.25 Percentage of operand collectors bandwidth increase with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register entry number (*reg_alignment*), register interleaved alignment on writes (*wr_alignement*), and an ideal alignment (*ideal_alignment*). 68

3.26 Percentage of used register file bandwidth showing the bandwidth increase from register coalescing using a fixed register alignment (*reg_bw_inc*) and an ideal register alignment (*ideal_bw_inc*) over the baseline bandwidth (*baseline_bw*). 69

3.27 Percentage of used operand collector bandwidth showing the bandwidth increase from register coalescing using a fixed register alignment (*reg_bw_inc*) and an ideal register alignment (*ideal_bw_inc*) over the baseline bandwidth (*baseline_bw*). 69

3.28 Percentage of register file bandwidth increase comparison between two different register file layouts: *wid_layout* and *wshift_layout* using two register alignment schemes: fixed alignment based on register number (*reg_ **) and write interleaved alignment (*wr_ **). 70

3.29 Breakdown of register file coalesced accesses with a *wid_layout* register file: Read-Read coalesced accesses (*rd-rd*), Write-Write coalesced accesses (*wr-wr*), and Read-Write coalesced accesses (*rd-wr*). 71

3.30 Breakdown of register file coalesced accesses with a *wshift_layout* register file: Read-Read coalesced accesses (*rd-rd*), Write-Write coalesced accesses (*wr-wr*), and Read-Write coalesced accesses (*rd-wr*). 71

3.31 Percentage of overall IPC speedup in GPU with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register number (*reg_alignment*), write interleaved alignment (*wr_alignement*), and an ideal alignment (*ideal_alignment*) compared to the upper bound (*upper_bound*). 72

3.32 Percentage of overall IPC speedup comparison between two different register file layouts: *wid_layout* and *wshift_layout* using two register alignment schemes: fixed alignment based on register number (*reg_ **) and write interleaved alignment (*wr_ **). . . . 73

3.33 Percentage of overall dynamic energy reduction in GPU with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register entry number (*reg_alignment*), register interleaved alignment on writes (*wr_alignement*), and an ideal alignment (*ideal_alignment*) compared to the upper bound (*upper_bound*). . . . 74

3.34 Percentage of overall dynamic energy reduction comparison between two different register file layouts: *wid_layout* and *wshift_layout* using two register alignment schemes: fixed alignment based on register number (*reg_ **) and write interleaved alignment (*wr_ **). . . . 74

4.1 GPGPU application warp thread statistics: (a) Percentage of warp threads in an inactive state, active and writing to register file, and active but not writing to register file. (b) Percentage of warp thread results with zero and non-zero values that can be represented with 8, 16, 24, and 32 bits. . . . 78

4.2 GPGPU main register file and execution pipeline with added components for power reduction highlighted. . . . 80

4.3 Usign the thread active mask and operand zero mask to gate off register file read access to the first four threads in a warp register. . . . 81

4.4 Generating the zero mask from data result produced by four execution lanes. . . 83

4.5 Data re-ordered in byte-position form to take advantage of low dynamic range values for power reduction. . . . 85

4.6 Power reduction for one execution lane using thread active mask bit and operands zero mask bits. . . . 87

4.7 Register file access reduction for read requests using power savings techniques: access reduction using threads active mask (inactive_gating), in-lane operands zero masks (zero_gating(in-lane)), and operands zero masks with data re-ordering(zero_gating (cross-lane)). . . . 90

4.8 Register file access reduction for write requests using power savings techniques: access reduction using threads active mask (inactive_gating), in-lane operands zero masks (zero_gating(in-lane)), and operands zero masks with data re-ordering(zero_gating (cross-lane)). . . . 91

4.9 Dynamic power reduction in GPGPU register file contributed by power reduction techniques using threads active mask (inactive_gating), in-lane operands zero masks (zero_gating (in-lane)), and operands zero masks with data reordering (zero_gating (cross-lane)). 92

4.10 Dynamic power reduction in GPGPU execution units contributed by power reduction techniques using threads active mask (inactive_gating) and trivial operations handling using operands zero masks (zero_gating). 93

4.11 Dynamic power reduction in total GPGPU chip power contributed by power reduction techniques applied to the register file (RF gating) and the techniques applied to the execution units (EX gating). 94

5.1 Behavior of Linux scheduler for a multi-thread benchmark running on architecture simulator. 99

5.2 Periodic load balancing done by Linux scheduler for a multi-thread benchmark running on architecture simulator. 100

5.3 Behavior of Linux scheduler with immediate load balancing for a multi-thread benchmark running on architecture simulator. 101

5.4 Mapping of new thread and core status update with the patched scheduler. 104

5.5 Behavior of patched Linux scheduler for a multi-thread benchmark running on architecture simulator. 105

5.6 Percentage of workload per core for 12-thread Canneal benchmark with different input sets run on 12-core hardware machine. 107

5.7 Performance speedup for Canneal benchmark using small input set, with 8, 12, 15, and 16 threads under memory speeds of 200MHz and 800MHz in full system simulation with the patched Linux scheduler. Results are normalized against an 8-thread 800MHz case. 107

5.8 Normalized performance speedup for PARSEC benchmarks runs with two memory speed settings using current and patched Linux schedulers in full system simulation. 108

LIST OF TABLES

TABLE Page

3.1 Design overhead of CORF compared with our low-cost design. 37

3.2 Register coalescing support in CORF compared with our design. 39

3.3 Comparison between CORF dual-address banks and our new dual-access banks. 51

3.4 Design overhead cost for our coalescing-aware design compared with CORF design. 62

3.5 Configuration parameters for our GPU design with register coalescing support. 63

3.6 GPGPU application benchmarks from Rodinia general-purpose suite. 63

3.7 Summary of results achieved by our design compared with CORF design. 75

4.1 GPU configuration parameters. 89

5.1 Configurations of the hardware machine used. 105

5.2 Configurations of the multi-core simulator used. 106

1. INTRODUCTION

In the single-core era, increasing performance primarily was obtained through transistor and clock frequency scaling while a constant power envelop was maintained due to Dennard scaling [3]. Dennard *scaling law* states that voltage and current of a digital integrated circuit scale with transistor dimensions, and therefore, power consumption is proportional to the circuit area. With the recent breakdown of Dennard scaling, the subsequent power consumption and heat dissipation constraints [4, 5], and the consequent inability to increase clock frequency significantly have forced the computer industry to rely upon core-count (and particularly thread level parallelism) scaling as the way forward to improve performance with increasing transistor density.

Thread parallel hardware significantly outperform single-core CPUs in both computational and memory bandwidth capabilities and became an ideal accelerator for multi-threaded and data-parallel applications. Graphics processing units (GPUs) are thread parallel processors that concurrently run thousands of hardware threads for graphics applications. General-purpose GPUs (GPGPUs) achieve high compute throughput and remarkable performance speedups leveraging GPUs to run more general compute applications. The increasing computational complexity of general purpose applications demands for higher compute capabilities which have been primarily accomplished by integrating more compute resources and promoting higher number of parallel threads in the GPU.

To support massive thread level parallelism (TLP) and fast context switching between active threads, GPUs provide a large register file to hold execution state (context) of each thread and a large number of execution units to execute threads in parallel. The size of the register file in the GPU has been almost doubling for every new generation of the Nvidia GPUs, recently reaching 20MB in Tesla VG100 [6]. The number of execution units have been also increasing as the number of units in Tesla VG100 is eleven times the number found in Fermi GTX480 [1]. Prior power analysis showed that the register file and the execution

units are the largest dynamic power consumers in the GPU and both contribute to about 40% of the total chip dynamic power consumption [7].

To avoid the high cost of multi-ported register file design, GPUs deploy a multi-banked structure with physical banks built using 6T SRAM arrays having a single read/write access port. The 6T arrays have a significant area benefit over 8T arrays at the cost of reducing the number of access ports from two to one. In addition, GPUs use single-ported operand collector units to capture the read data out of the register file banks. The register file banks operate in parallel to support high access demand and provide high bandwidth. However, due to access port limitation on the banks as well as the collector units, multiple access requests that target the same shared resource at the same time experience *port conflicts* and their access is serialized. As a result, register file access latency increases and negatively impacts overall GPU performance and energy efficiency.

1.1 Register File Access Coalescing in GPU

As mentioned earlier, port serialization on the GPU register file banks and operand collector units increase the register file access latency and negatively impact overall all performance and energy efficiency. In this work, we present a new register file design that supports read and write access coalescing in order to improve performance and reduce overall energy in GPGPU. Access coalescing has been used in memory system to combine multiple access requests to contiguous memory space into a single request in order to reduce memory traffic and improve bandwidth utilization. We applied the concept of access coalescing on GPU register file to combine multiple bank accesses that target different registers within the bank into fewer accesses. In addition, we also supported coalescing registers reads from different register file banks that target the same operand collector unit. Coalescing opportunities arise from the frequent narrow-width data found in general purpose compute applications that are read from and written into the register file. Read and write requests of narrow-width data that target register file banks can be combined to reduce the number of bank accesses, yield higher bandwidth utilization for register file banks and operand collectors, reduce register

file and operand collectors ports contention, and as a result, improve overall performance and energy efficiency in GPU.

Our design seeks to support all possible coalescing opportunities between access requests that contend on the limited access ports available on the register file banks and the operand collectors. The register file bank is physically built with multiple sub-banks each of which holds a slice of every physical register entry. In our design, a narrow-width read or write request into a register file bank is arranged in a way that only a subset of the sub-banks within the bank is accessed. This allows for read or write requests to different register entries that access non-overlapping sub-banks in a given bank to be coalesced into a single access. To help reduce contention on the collector unit write port, our design also supports coalescing read requests from the same warp instruction across different banks given that the read requests access non-overlapping sub-banks (across the banks) to allow their read data to be packed into a single operand collector write. With these access coalescing capabilities, our design supports coalescing the following register file requests into a single physical access: two read requests from same or different warps accessing the same bank, two write requests into the same bank, a read and write requests accessing the same bank, two read requests from the same warp accessing different banks.

In this work, we made the following main contributions:

- We present a new register file organization that supports coalescing across different register entries within register file banks for read and write requests and combine them into a single bank access. It also supports coalescing read accesses across different banks that target the same operand collector by combining their read data into a single operand collector write.

- We provide a hardware-only solution to support register file access coalescing with low overhead and complexity. Our design requires minimal addition of micro-architectural states and small combinational logic that do not require extra pipelining.

- We support a coalescing-aware register file that place no restrictions on how physical registers are arranged among the register file banks. And we also do not require accessed registers to be packed in the same physical register entry for their access to be coalesced as we freely support coalescing across physical register entries within a bank.

1.2 Exploiting Zero Data to Reduce Power Consumption in GPU

As mentioned earlier, the main register file and execution units are the largest dynamic power consumers in the GPU. In this work, we focus on reducing dynamic power for these two power-hungry components without impacting performance for GPGPU applications by proposing gating techniques that support the following power savings opportunities:

- **Inactive threads:** A warp represents the unit of execution in GPGPU and it consists of 32 threads executing in a lock-step in a single instruction multiple data (SIMD) execution pipeline. Divergent flow presented in general-purpose compute applications are causing warps to be under-utilized. In other words, some of the threads in a warp are inactive due to control divergence and need not be executed. We take advantage of this program attribute to reduce dynamic power of the main register file and execution units by gating off inactive threads during warp execution.

- **In-lane zero data:** Each thread in a warp executes in one 32-bit execution lane. Data operands or results that are specific to a given thread, which we refer to as an in-lane data, can have a zero value sometimes. The presence of zero value gives an opportunity to reduce threads dynamic power consumption by having the zero information for every architectural register saved in a separate state. This allows for unnecessary access to the power-hungry register file to read or write zero values to be avoided. It also allows for avoiding unnecessary execution of certain instructions that perform trivial operations when one or more of their source operands having a zero value.

- **Small dynamic range data (cross-lane zero data):** As the execution lane in

GPGPU is 32-bits wide, a thread in a given lane reads or writes 32-bit data values from the register file. The register file is typically comprised of 32 $\times$ 32-bit vector registers that supply operands to 32 threads within a warp. Per-thread data values used in compute applications varies in size and some values can be represented by only 8, 16, or 24 bits with the upper most-significant bits being zeros but these values are still being treated as 32-bit values when read and written into the register file. To take advantage of such small dynamic range of program values, in-lane data produced by adjacent execution lanes can be ordered (cross lanes) in a way to group the upper zero bytes together and potentially forming 32-bit zero values that can be captured in a separate zero state. This way the register file access for these 32-bit zero values can be gated to further reduce the register file dynamic power.

In this work, we propose a power reduction scheme that has low area and power overheads and has no performance impact. The proposed scheme takes advantage of the high percentage of zero data that exist in general purpose GPU applications to reduce dynamic power for the following power-hungry GPU components:

- **Register file:** avoid reading and writing zero data (in-lane or cross-lane) from the register file by capturing the zero information in a separate low area state.
- **Execution unit:** use the captured zero information to detect trivial operations which have one or more zero operands and avoid their execution by generating their trivial results directly.

1.3 Architecture Simulation and the Impact of Linux Thread Scheduler

As we have mentioned, the computer industry as well as architecture research have moved to multi-core systems with the end of Dennard scaling. Exploring new microarchitectures often requires simulation to do quantitative analyses of the performance and other metrics of these new designs, since implementing them in hardware is often prohibitively expensive.

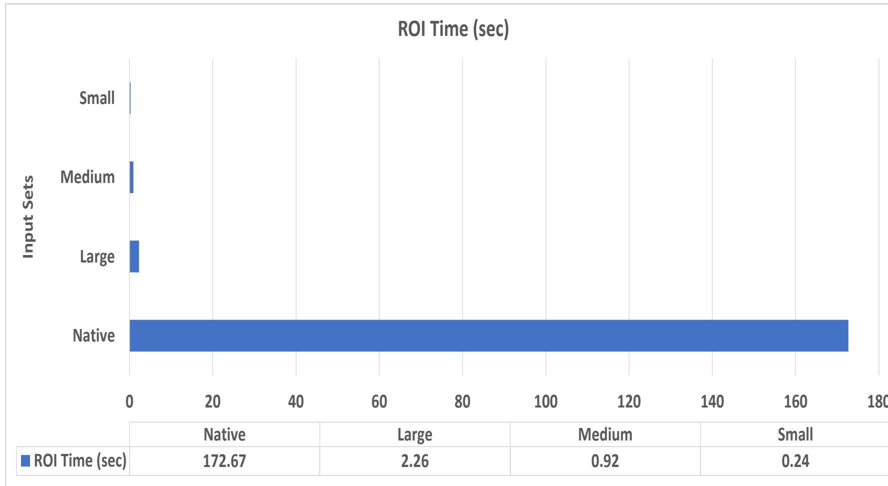


Figure 1.1: Actual run-time of 12-thread Canneal benchmark with different input sets on 12-core hardware machine.

Unfortunately, simulating an architecture incurs huge overheads in terms of simulated cycles per second versus the machine being simulated [8, 9, 10]. Typically, the slowdown of simulation versus real hardware is on the order of 10,000-100,000:1. As a result, architecture simulation experiments usually run scaled down versions of real applications where input data set and run iterations are chosen such that the runtime is reasonably short, while the performance characteristics of the full/native run of the benchmark are maintained. Fig. 1.1, which illustrates this point, shows the execution time of a 12-thread Canneal benchmark (from PARSEC suite [11]) with different input data sets when run on a real machine. In the figure, the *Small*, *Medium*, and *Large* represent the runtimes of input sets designed for architecture research, as compared to the *Native* input set which would be a typical production input for the application. As we see, the architecture research input sets are between 70-700x smaller than the native set. We note, that this same small input set for this benchmark takes approximately 6 hours from start to end, in full system simulation on the gem5 simulation toolkit [12].

Unlike the single threaded benchmarks of the previous era, wherein simulators could

simply emulate the operating system, architecture research on multi-threaded applications requires "full system" simulation. In full system simulation, the simulated system boots a real operating system then launches the multi-threaded application under test. Achieving scaling performance with core count in multi-threaded applications, according to Amdahl's Law [13], critically requires that the OS balance the workload across cores effectively. Architecture researchers rely on the fact that the OS is providing maximum utilization of the core resources and system software has no impact on their experimental results relative to a real system running the same program. Unfortunately, we have found that, for the short input sets typically used in architectural research, the behavior of the OS thread scheduler is often not as expected.

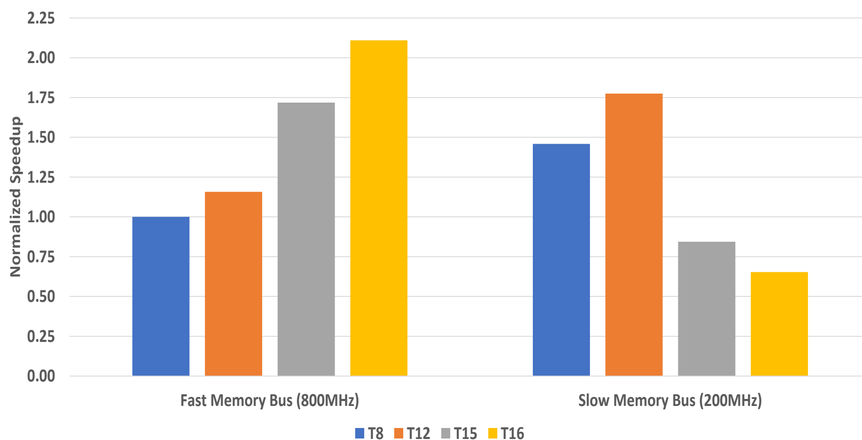


Figure 1.2: Performance speedup for Canneal benchmark using small input set, with 8, 12, 15, and 16 threads under memory speeds of 200MHz and 800MHz in full system simulation with original Linux scheduler. Results are normalized against an 8-thread 800MHz case.

Fig. 1.2 illustrates how the short runtimes used in architecture research, together with the thread-scheduling/balancing of typical OSes interact to produce inconsistent and incorrect results from simulation. The Figure shows normalized speedups of Canneal benchmark with the small input set and with different number of software threads under two memory-bus

speed settings. The benchmark was run under full system simulation of a (gem5 [12]) 16-core system with Linux OS. Looking at the results we see several inconsistencies. First, we see that increasing thread-count with the 200MHz memory bus has a seemingly random effect on runtime, sometimes lowering it, sometimes raising it. Second, we see that the 8-thread, 200MHz bus counter-intuitively produces better performance than the 8-thread and 12-thread version with an 800MHz bus. As we will show the main source of these inconsistencies is due to the behavior of the current OS thread scheduler when it is used for short-lived simulation experiments.

This work focuses on the impact of system software on the behavior and correctness of simulation experiments performed in full system simulation with real OSes. We show that, for the short runtimes used in architecture research, the scheduler does not behave as expected to provide global load balance and fully utilize the simulated multi-core system. We characterize why and how this effect occurs. Finally, we propose a simple patch for the OS scheduler, for use in architecture research, to improve the consistency and correctness of multi-threaded applications when used for architecture research.

1.4 Dissertation Statement

As thread parallel computing became essential for accelerating a variety of general purpose applications in use today, our goal is to make the thread parallel hardware more power and energy efficient, higher performance, and its simulation to be more accurate. To this end, our work covers the following topics: (1) We significantly improved overall performance and dynamic energy efficiency on the GPU by introducing a new register file organization that supports access coalescing of narrow-width registers frequently found in general purpose applications. (2) We improved dynamic power consumption of the main register file and execution units in the GPU by introducing power saving techniques that take advantage of programs attributes, inactive threads and zero data, frequently found in general purpose applications. (3) We also addressed inaccuracies in full system simulation environment with a real operating system (Linux kernel) that is primarily used in architecture research on

multi-threaded applications.

1.5 Dissertation Outline

Chapter 2 provides a background on modern GPU architectures and full system simulation environment. In Chapter 3, we present our new coalescing-aware register file organization to improve performance and energy efficiency in GPGPU. We start the chapter with a brief introduction in Section 3.1 followed by performance limitations in GPU register file that we aim to overcome in Section 3.2. Our motivations for register file coalescing is presented in Section 3.3. Section 3.4 presents illustrative examples of register access coalescing supported by our design. In Section 3.5, we present related work on GPU power optimization as well as prior work done on register file coalescing. The new coalescing-aware register file design is presented in Section 3.6 and is evaluated in Section 3.7. Then, we conclude this chapter in Section 3.8.

In Chapter 4, we propose power optimization techniques in GPU by exploiting zero data that exist in general-purpose applications. Section 4.1 gives a brief introduction followed by statistical measurements that show our motivations in Section 4.2. The design of the proposed techniques are presented in Section 4.3 and in Section 4.4 for register file and execution units, respectively. The power saving techniques are evaluated in Section 4.5. Section 4.6 covers related work on GPU power optimization and we then conclude this chapter in Section 4.7.

In Chapter 5, we present the negative impact of system software behavior on multi-threaded applications running on full system architecture simulation when small input sets are used with a brief introduction given in Section 5.1. Section 5.2 demonstrates the behavior of Linux thread scheduler when running multi-threaded application with small input set and show the negative impact on the correctness of simulation results. We propose a solution to the unexpected behavior of the Linux scheduler in full system simulation in Section 5.3 and evaluate our solution in Section 5.4. We conclude this chapter in Section 5.5. Finally, an overall summary of our research work is given in Chapter 6.

2.1 Modern GPU Architecture Model

2.1.1 CUDA Overview

Compute Unified Device Architecture (CUDA) is the software platform that enables Nvidia GPUs to execute programs written in C, C++, or other languages [14]. A serial C++ program that performs a vector addition as in Fig. 2.1a can be accelerated to run on the GPU by creating a CUDA equivalent version of the program as in Fig. 2.1b. Each function defined on the device, as the *add* function, is called a program kernel. A CUDA program can have one or more kernels which are called and executed in parallel. The *add* kernel executes in parallel across a set of parallel threads. The program specifies the number of threads needed to execute the kernel and organizes the threads into thread blocks and grids of thread blocks. Each thread within a thread block executes an instance of the kernel and has its own thread identifier. CUDA thread hierarchy maps to a hardware hierarchy of multi-processors in the GPU.

The CUDA program for the vector add example requires 1024 threads where each thread performs the addition operation on a single instance (element) of the input vectors. Threads are organized into a grid of two thread blocks with each block has 512 of the threads as shown in Fig. 2.2. A grid is an array of thread blocks that perform the same kernel and execute in parallel on different GPU Streaming Multiprocessors (SMs) or cores. A thread block, of 512 threads in this example, is assigned into an SM core and has concurrently executing threads that execute in groups of 32 threads called warps (also known as wavefronts). A warp of 32 threads is the the amount of work an SM core can initiate in a cycle. The warp scheduler within each SM picks an active warp every cycle and threads within the warp execute the same instruction on different data elements concurrently in a lock-step in the

```
//C++ serial version
void add (int N, int* A, int* B, int* C) {
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}

//invoke add function
int N = 1024;
add(N, A, B, C);
```

(a)

```
//Compute Unified Device Architecture (CUDA) version
__device__
void add (int N, int* A, int* B, int* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}

//invoke add function
__host__
int N = 1024;
add<<<2,512>>>(N, A, B, C);
```

(b)

Figure 2.1: C++ code example for vector addition (a) Serial code that typically runs on CPU. (b) CUDA thread-parallel version of the code that runs on GPU.

Single Instruction Multiple Data (SIMD) execution pipeline.

2.1.2 GPU Chip Layout

Fig. 2.3 shows a modern GPU chip of the Nvidia Fermi family [1]. The GPU consists of 16 Streaming Multiprocessor units (SMs) with an on-chip shared level-two (L2) cache.

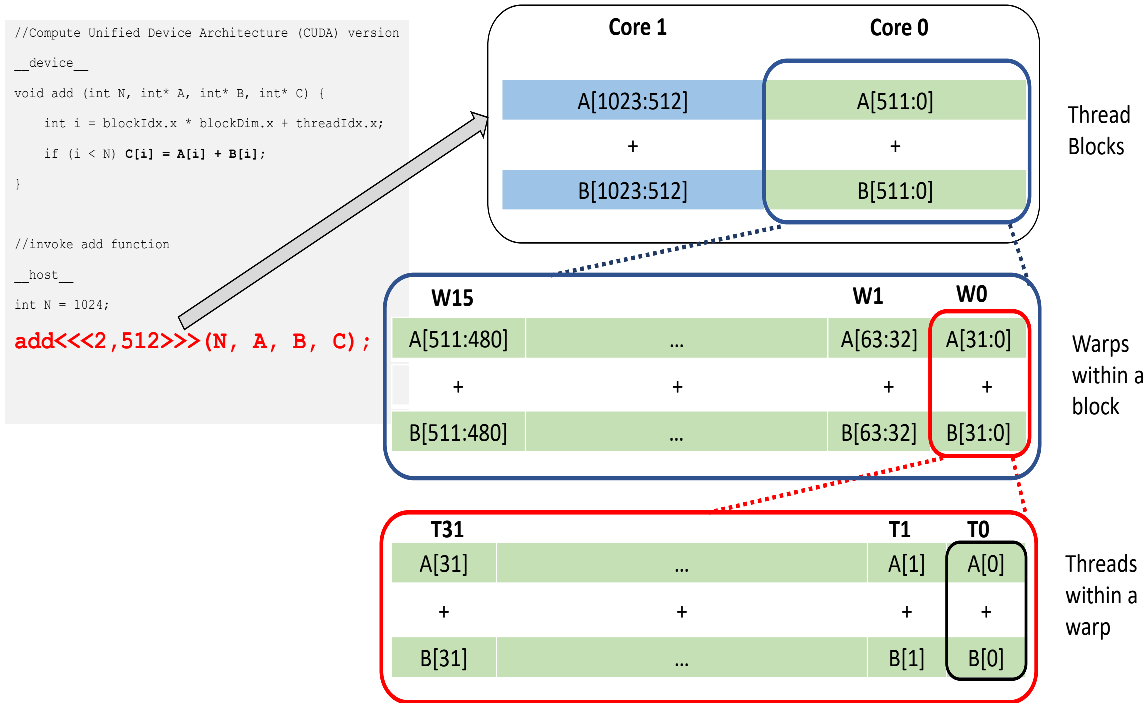


Figure 2.2: CUDA hierarchy of threads that maps to a hierarchy of processing elements on the GPU.

The SMs access the L2 cache and the external Dynamic Random Access Memory (DRAM) using an interconnection network which is usually referred to as Network On Chip (NOC). Each SM has a single execution core that consists of level-one (L1) instruction and data caches, warp schedulers, main register file, multiple execution units, and a shared memory. Threads within the thread block executing on the SM core communicate through the local shared memory and each thread in the thread block has its own private register file entries to save its architectural state (context). The register file in each SM has a total size of 128KB and is organized into multiple banks. The SM execution units are of three types: (1) Streaming Processing Units (SPUs) to execute integer and floating-point arithmetic and logical instructions. (2) Special Functional Units (SFUs) for executing special functions like sin and cosine. (3) Load and Store units (LDSTs) to perform memory accesses. The SM has

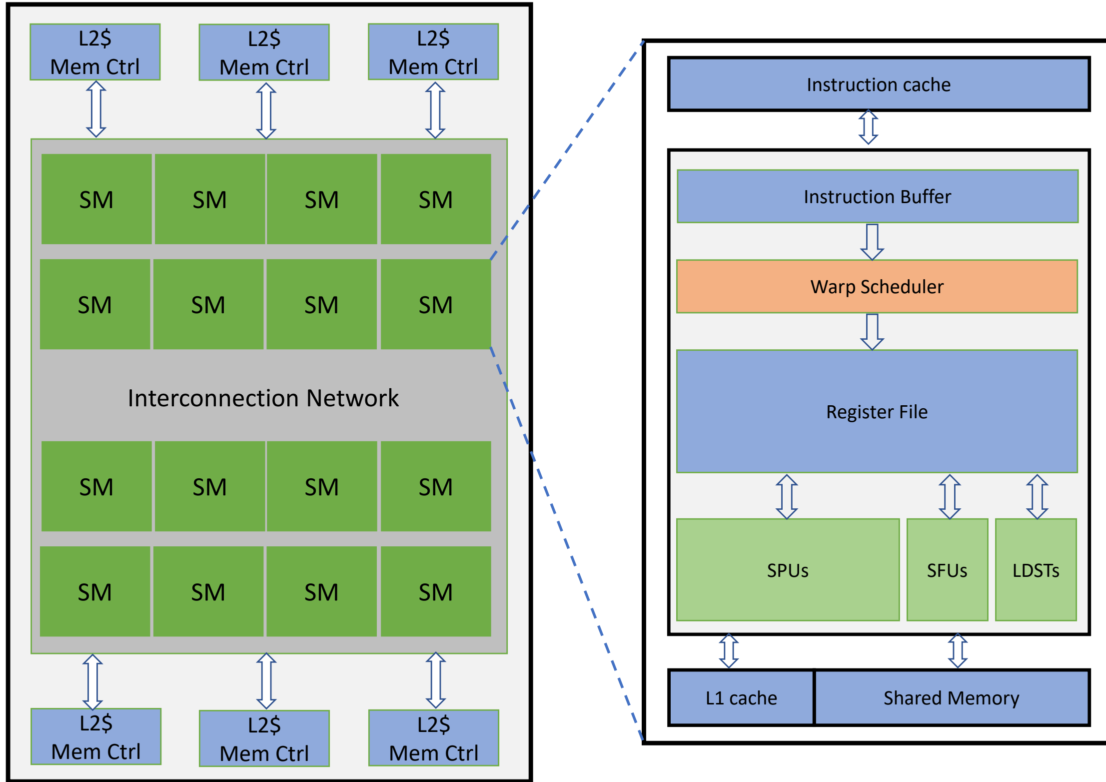


Figure 2.3: Modern GPU chip layout with 16 Streaming Multiprocessors (SMs), each of which has its own register file, instruction and data caches, and execution units. Reprinted from [1].

a configurable partitioning of L1 data cache and shared memory space with a total size of 64KB which can be configured as 48KB shared memory and 16KB cache or as 16KB shared memory and 48KB cache.

A program kernel, as the vector add example mentioned earlier and shown in Fig. 2.1, is divided up into thread blocks (also known as Concurrent Thread Arrays (CTAs)) and each block gets allocated into one of the SMs to be worked on. Threads within the thread block are divide up into groups of 32-thread warps (or wavefronts) that get issued by the warp scheduler once every cycle. Each issued warp requires an Operand Collector Unit (OCU) to read all needed source operands from the register file before it gets dispatched into the execution pipeline. Threads in a warp execute in a SIMD fashion in a lock-step where each

thread executes the same instruction on a 32-bit slice of the operands data. The 32-bit wide execution pipeline for a single thread is referred to as an execution unit or (lane).

2.1.3 Warp Scheduler

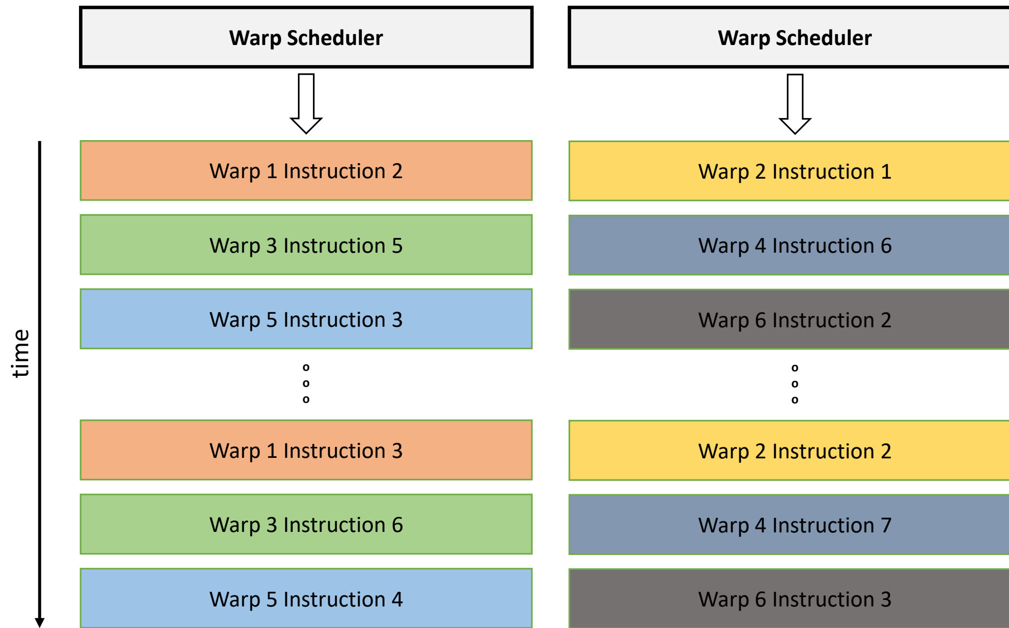


Figure 2.4: Dual-warp scheduler used in Fermi GPU. Reprinted from [1].

The GPU relies on the warp scheduler to maintain high utilization of the compute resources available. Warps within a thread block, as shown in Fig. 2.2, are organized by the warp scheduler, as in the two-level scheduler [15], into two groups: (1) pending warps that are waiting on long-latency memory access and (2) warps that are active. The active warps are also organized into two groups:(1) warps that have dependency on older executing warps that is either a Read-After-Write (RAW) or a Write-After-Write (WAW) dependency and (2) warps that have no data dependencies and are *ready* to be issued. The warp scheduler selects one of the *ready* warps to issue every cycle using an arbitration policy, such as

Round-robin (RR) or Least Recently Used (LRU), to provide fair arbitration among available warps. Fig. 2.4 shows the dual-warp scheduler used in the Fermi GPU. An issued warp is assigned an available operand collector unit to read its needed source register operands from the register file before it gets dispatched into the execution units pipeline.

2.1.4 Register File

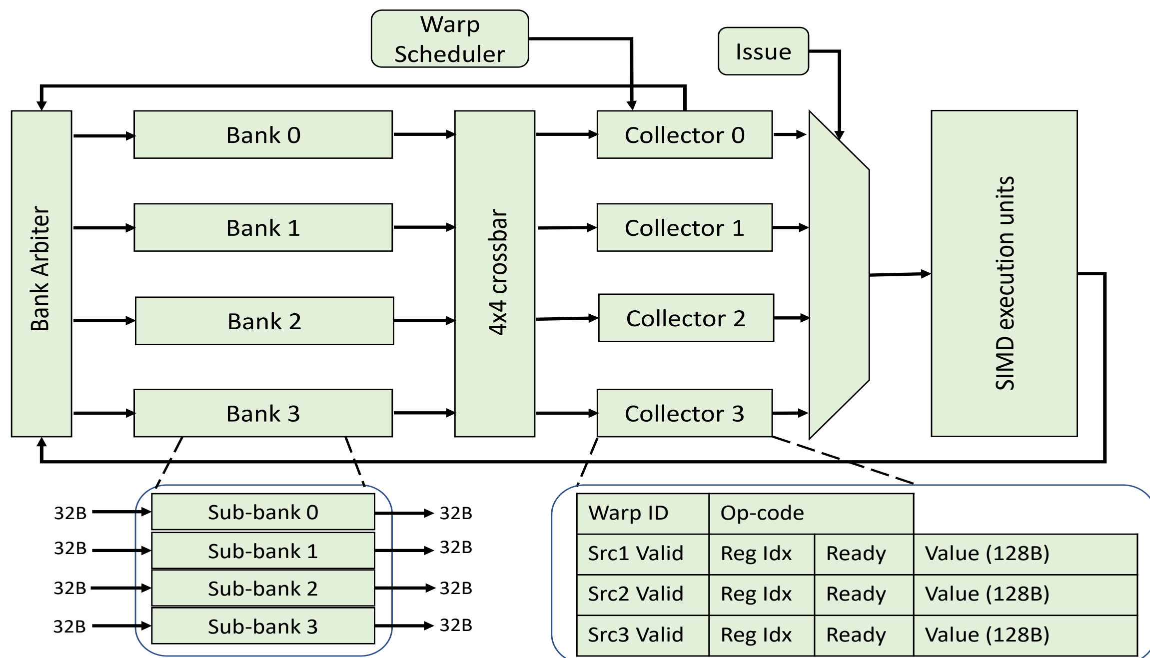


Figure 2.5: GPU main register file and execution pipeline.

Fig. 2.5 shows the register file organization similar to the Fermi family of Nvidia GPUs [1]. To avoid the area cost of multi-ported design, GPUs adopt for a multi-banked register file organization built with single read-write port SRAM banks to provide large access bandwidth. Register file banks operate in parallel to serve read and write requests where each of these requests can target only one bank. Multiple requests that target the same bank experience a *bank conflict* and their access is serialized due to access port limitation.

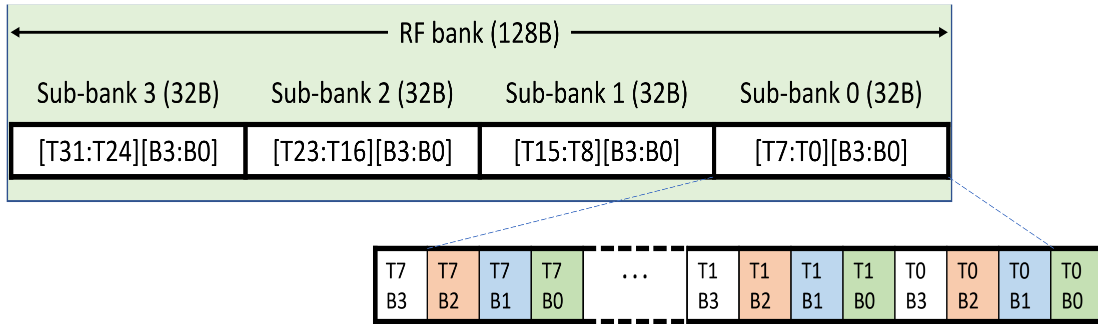


Figure 2.6: A 128B warp register entry in one register file bank occupying four 32B sub-bank entries that have the same index. Each 32B sub-bank entry holds data for 8 threads within the warp.

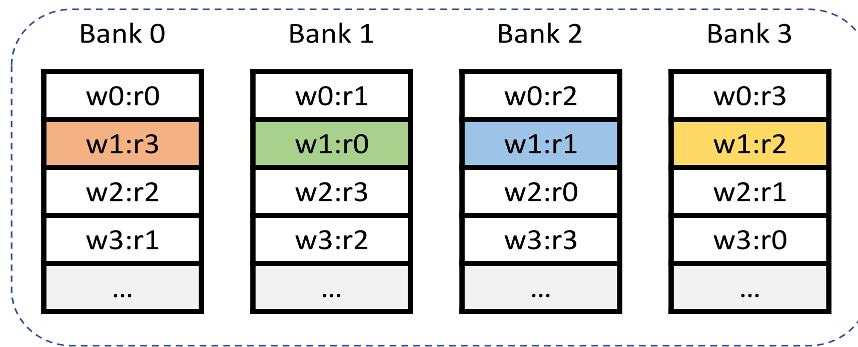


Figure 2.7: Register-to-bank mapping (layout) with warp registers interleaved across register file banks.

A register file bank is built using multiple narrower sub-banks each of which holds a 32B slice of all register entries in the bank. A read or write request to a given bank accesses all its sub-banks with the same index at the same time. Every warp in the GPU has a dedicated set of 128B registers that are indexed using the warp number. As shown in Fig. 2.6, the 32 thread-registers within a warp form a single bank entry which is split across the sub-banks and are accessed with the same warp register index. Data in the warp register entry is represented in a byte-interleaved format where the four bytes (byte 0, 1, 2, and 3) of thread 0 is presented in the least significant position, followed by the four bytes of thread 1, and so

on.

There is a one-to-one mapping between logical registers and physical registers. Warp registers are mapped to the register file banks based on the layout chosen with one possible layout is to map all registers for a given warp into the same bank. Another layout, used to reduce bank conflicts between warps, has registers belonging to the same warp interleaved across the banks as shown in Fig. 2.7.

Operand collector units are used to buffer warps operands data as they are read from the register file, over multiple cycles, with one collector unit used per warp instruction. The number of register operands to read for a given warp varies by instruction type with a maximum of three operands needed for a fused-multiply-add (FMA) instruction. The number of write ports on an operand collector is limited to only one 128B wide port that can accept read data from one of the register file banks at a time. Multiple bank reads that target the same operand collector experience a *port conflict* and are serialized due to the single-port limitation on the collector unit. Routing read data from the banks into the operand collector units is done using a crossbar interconnection network.

2.1.5 Execution Units

Each SM has three different types of SIMD execution pipelines as shown in Fig. 2.8. An arithmetic/logic Streaming Processing Unit pipeline (SPU) is used to execute integers and floating-point instructions, a Special Functions Unit (SFU) pipeline is used to execute special functions such as sin/cosine and square root operations, and a Load/Store Unit (LDST) pipeline used to perform memory loads and stores.

A 32-thread warp instruction is issued into an execution pipeline when all its operands are marked ready in the operand collector and is also found the oldest among other ready warps. Each thread within a warp requires a single 32-bit lane (pipeline unit) to execute the warp instruction. 32-thread warp instructions are dispatched every cycle into the SPUs, every 8 cycles into the SFUs, and every 2 cycles into the LDST units. With a dual-warp scheduler, two warp instructions can be dispatched into the the SPUs at the same time were

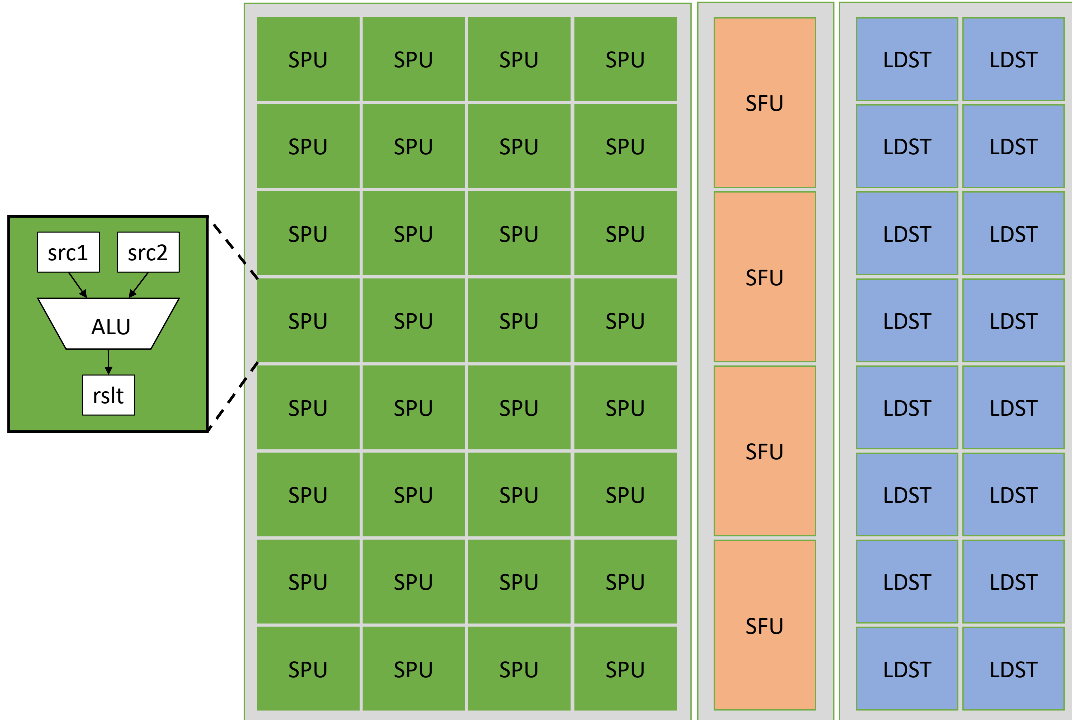


Figure 2.8: Execution units in a GPU Streaming Multiprocessor (SM) core with 32 Streaming Processing Units (SPUs), 4 Special Functional Units (SFUs), and 16 memory Load/Store Units (LDSTs). Reprinted from [1].

each warp uses half the number of the SPUs and dispatched over two cycles. Once a warp is dispatched, the operand collector assigned to the warp is freed and can be immediately used by one of the younger warps. The three pipelines can operate on different warp instructions in parallel and each may write up to one result back into the register file.

2.2 Full System Simulation

As previously discussed, architecture simulators are used by the research community to validate their new ideas and proposed solutions. Some simulators, such as gem5 [12], can run the full Linux kernel[1] within the simulation environment in *full system* simulation experiments. Fig. 2.9 shows a full system simulation environment with 16 single-threaded cores managed by an operating system kernel running a thread scheduler. Just as in a real

[1] We focus on Linux here, as it is the OS typically used in architecture research.

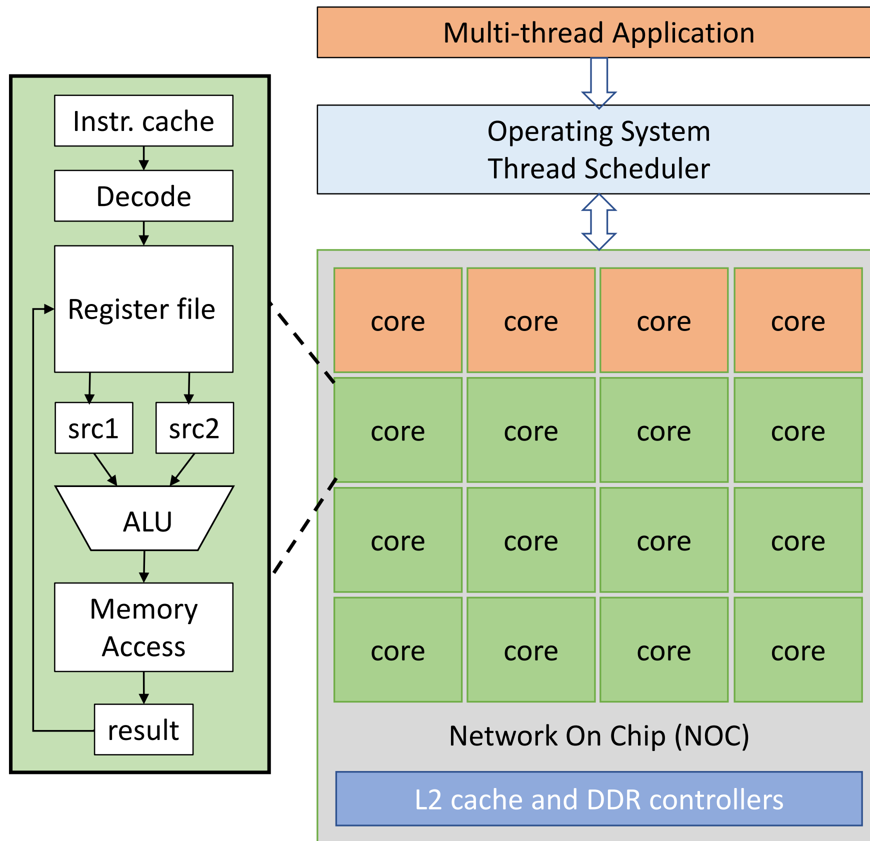


Figure 2.9: Full system simulation environment for a Core Multi-Processor (CMP) chip with 16 cores managed by a real Operating System (OS) running a thread scheduler. The full simulation system runs multi-threaded user applications similar to a real multi-core system with an OS.

system, the kernel is booted on one of the cores in the simulated system before user-level code can be executed. Once the kernel is up and running on the simulator, multi-threaded application benchmarks (*e.g.* the PARSEC benchmarks [11]) can be run on the system. These benchmarks use the pthread run-time libraries to fork software threads and manage communication/synchronization between those threads. Ultimately the pthreads library is a wrapper around OS calls to complete these tasks. Similarly, the OS handles the scheduling of threads; it is expected to be performed in a way that provides high performance and fair execution among running threads. Further, thread-to-core mapping is another important

job of the kernel and is expected to be performed in a way to fully utilize the multi-core system and achieve good global load balance.

Linux scheduler has evolved over time to support different platforms such as desktops and servers. Early versions of the Linux scheduler only supported simple, uniprocessor systems with no multi-threading or multi-processing. Starting with version 1.2, the scheduler used a circular buffer, enforcing a round-robin policy to provide fairness among software threads regardless of their type or class. Later in version 2.2, scheduling classes were added to provide different polices for real-time and none-real-time tasks. This version also had the first support for symmetric multi-processing (SMP). With the introduction of SMP, the scheduler's job became more complicated as it needs to provide fair scheduling among running tasks and also provides global load balancing over the available cores in the multi-core system. The scheduler has been an active research topic and been evolving to improve fairness and reduce run-time complexity. The current scheduler in use today is the "completely fair scheduler" (CFS) [16] which followed O(1) scheduler [17] in version 2.6.23. This scheduler's goal is provide better fairness among running threads and enhance applications performance compared to previous schedulers.

While it is important to achieve fairness among threads running on a given core, achieving global fairness is highly important for multi-thread applications running on a multi-core system. CFS made some improvements for global load balancing in version 2.6.24 among them the introduction of scheduling domains [18, 19]. Each scheduling domain spans a number of cores in the system and domains are built in a hierarchical fashion. Cores within a scheduling domain are organized into groups where the union of the groups is the span of the domain and the intersection between any two groups is an empty set. Load balancing within a scheduling domain happens between groups. Each group is considered an entity with a load equal to the sum of loads of all cores in the group. Tasks are moved from one group to another when imbalance condition is detected. In SMP mode, all cores in the multi-core system belong to one parent scheduling domain where each group within the domain

has only one core.

A multi-thread application running on a multi-core system relies on the global load balance provided by the scheduler in order to achieve scaling performance with core count. When a new software thread is forked, the scheduler performs a minimum search among the available cores in the system to find a candidate core to run the thread. The scheduler relies on current core status information to select the first idle or otherwise least busy core found during the search to run the thread. Critically, this search starts from the same core ID each time, taking into no account whether that core has already had a given application's thread mapped to it in the recent past, only whether or not that core is currently idle.

Lacking an application-level view can lead the scheduler to map more threads to some cores over others in the event that those threads are currently idling, causing load imbalance in the multi-core system. To address the adapting load per core, the OS periodically (approximately once every 30 milliseconds) performs a load balancing operation on all cores to incrementally reduce the degree of load imbalance and enhance applications performance. In this operation, a single core searches for the busiest other core in the system, and performs a thread migration when a high load imbalance is detected between the two cores. The heavy-weight system-wide search is initiated by one core at a time in a sequential order to reduce contention and avoid ordering complexity among cores performing the rebalance. Thus, one full iteration of the core balancing requires $30 \times N$ milliseconds, where N is the number of cores in the system. Over the long haul this system will generally find an optimal thread-core mapping balance, however it can often take many iterations of this search to do so. For example ten full iterations of core rebalancing on a 16-core system could take as long as 5 seconds to find an optimal balance of threads, much longer than the runtime of the benchmarks used in architecture research (see Fig. 1.1).

3. REGISTER FILE ACCESS COALESCING IN GPU

3.1 Introduction

As we mentioned in Chapter 1, a multi-banked register file structure with limited access ports has been used in GPUs to mainly reduce its area cost. With limited number of ports, register file accesses are serialized when accessing individual banks which impacts access latency and overall IPC performance for the GPUs. In this chapter, we focus on improving IPC performance and energy utilization for the GPU by proposing a new register file organization that support register file access coalescing. Access coalescing provides combining different narrow-width read and write requests that contend on available bank ports to form a single physical request which in turn reduce the overall number of bank accesses and register file pressure, improve bandwidth utilization, reduce access latency which improves overall IPC performance and energy efficiency in GPU.

3.2 Performance Impact of Limited Access Ports

As we mentioned in Section 2.1, the warp scheduler in each SM core maintains a pool of ready warp instructions and selects one of these warp instructions to issue in a cycle. The issued warp instruction is assigned an available operand collector unit (OCU) to read its source operands before it starts executing. Each operand collector may fetch up to three source register operands from the register file by sending read requests to the banks where the registers are located. Write requests targeting the banks can be generated from an arithmetic instruction or a memory load instruction when their results are ready. As illustrated in Fig. 3.2, with limited number of access ports, read or write requests can experience one of the following types of *port conflicts* and their access is serialized:

1. Write-Write Conflict: As shown in Fig. 3.2a, a memory write request is blocked by a write request from the execution pipeline which has a higher priority accessing the same bank.

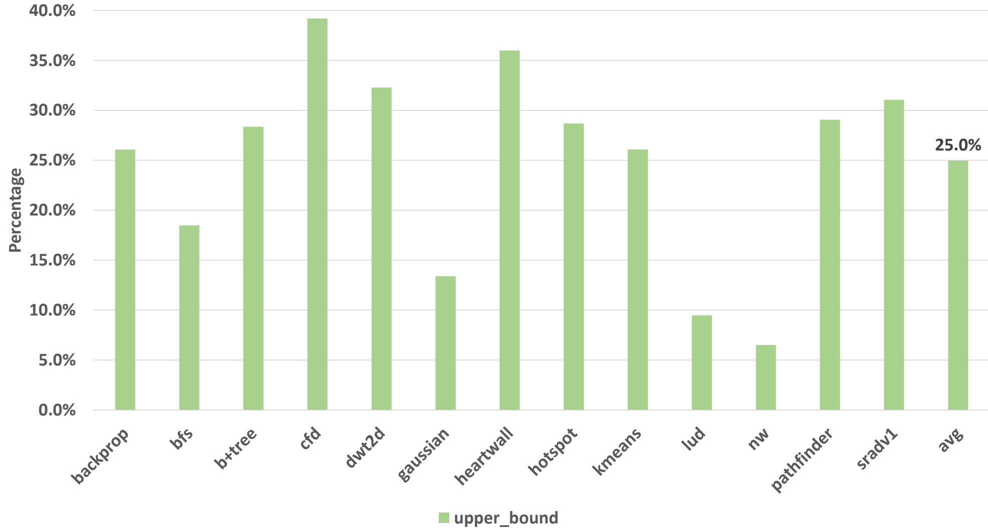


Figure 3.1: A limit study on the potential IPC performance speedup of reducing register file banks port conflicts.

2. Read-Write Conflict: As shown in Fig. 3.2b, a read request is blocked by a higher priority write request accessing the same bank.
3. Read-Read Conflict: As shown in Fig. 3.2c, a read request is blocked by another read request that won the bank arbitration and granted access to the bank.
4. OC Write Conflict: As shown in Fig. 3.2d, a read request is blocked by another read request accessing a different bank that targets the same operand collector unit and won the arbitration and granted access to the bank as well as the collector unit.

These conflicts are due to the limited access ports on register file banks and operand collectors as the area cost of adding a port is very high given the large width of warp registers. The bank arbiter is responsible for prioritizing the read requests that are conflicting on either accessing the same bank or accessing the same operand collector. The port serialization on register file banks and operand collector units directly impacts overall GPU performance as it may lead to one of the following situations:

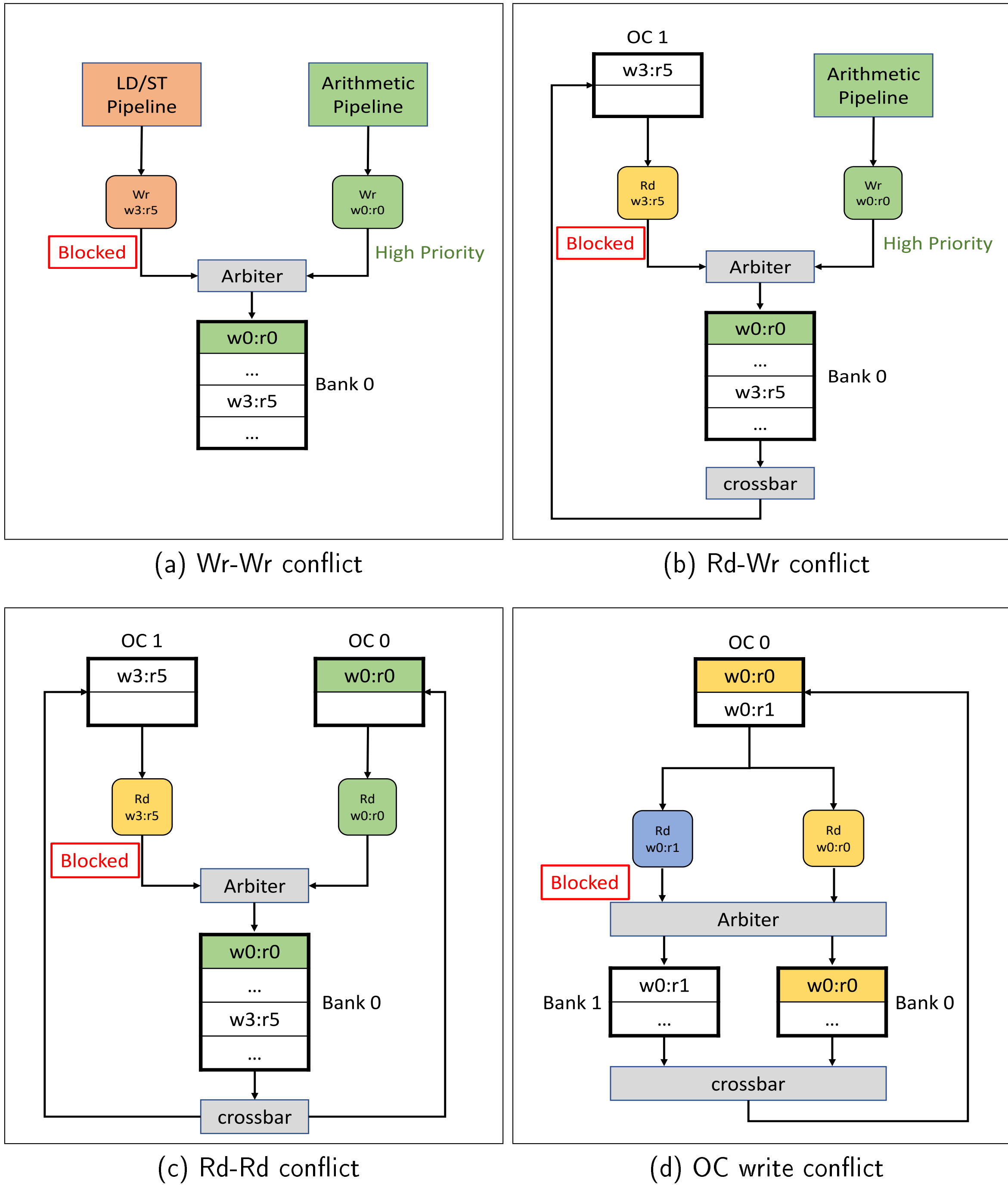


Figure 3.2: Examples of port conflict on register file bank and operand collector accesses (a) Bank conflict between two write requests. (b) Bank conflict between a read and a write request. (c) Bank conflict between two read requests. (d) Port conflict on two writes to an operand collector unit.

- Delay the execution of a warp instruction.
- Cause a dependent instruction to wait longer before it can get issued.
- Delay freeing up an operand collector unit which prevents new warps from getting issued.

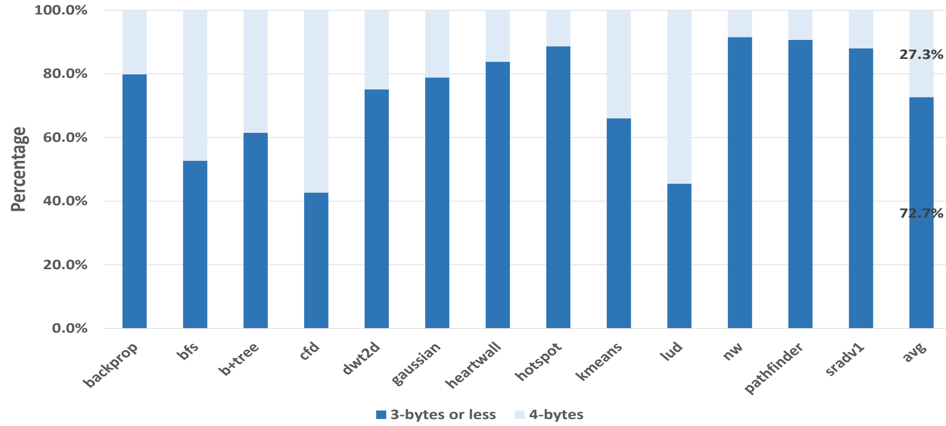
Fig. 3.1 shows the potential benefit of reducing register file bank conflicts which can provide an IPC performance speedup of up to 25% on average. In this work, we present a new register file organization that can combine conflicting narrow-width requests, reads or writes, into a single coalesced request for register file banks as well as operand collector units. With access coalescing, the number of register file requests and the access latency are reduced which lead to improving overall performance and energy utilization in GPU.

3.3 Motivation

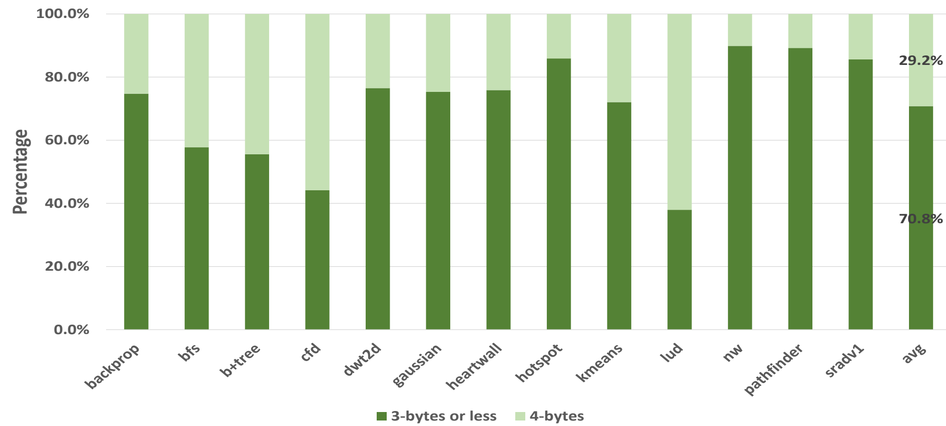
In this section, we present results from compute-intensive (GPGPU) benchmarks in order to show our motivation behind the coalescing-aware register file design we are proposing in this work that enables access coalescing on register file banks and operand collectors. For the experiments we show in this section, we examined GPGPU benchmarks from the general-purpose Rodinia benchmark suite [20] v3.1 and obtained the experimental results by running those benchmarks on the GPGPU-sim v3.2 simulator [21] modeling a Fermi GPU with design parameters shown in Section 3.7.

3.3.1 Register Operands Width

Register file coalescing opportunities arise from the presence of narrow-width register operands in general-purpose compute applications. Fig. 3.3 shows the frequent narrow-width source operands (Fig. 3.3a) and narrow-width destination operands (Fig. 3.3b) found in the GPU benchmark suite we used. We classified the operands into either full-width that require 4-Bytes per warp thread or narrow-width operands that effectively require 3-Bytes or less per thread to correctly represent their data without any loss of information (the rest



(a)

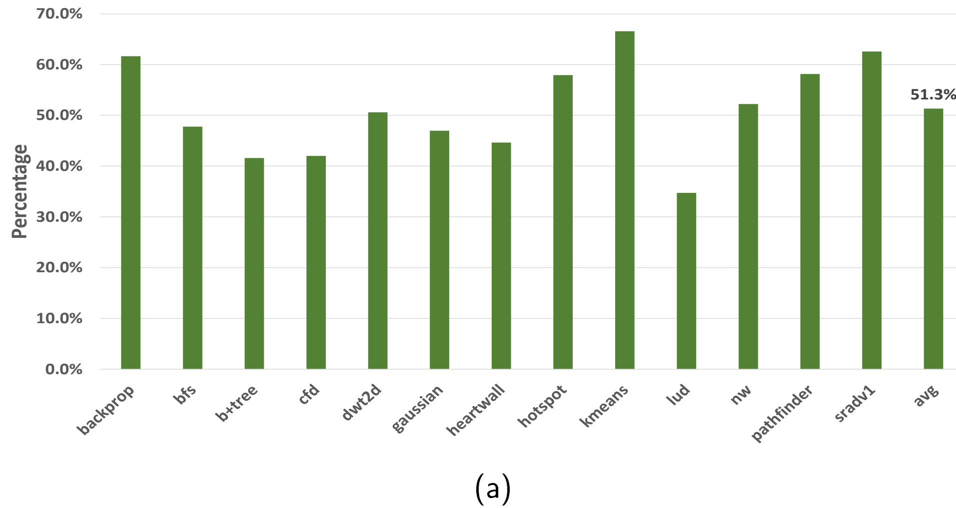


(b)

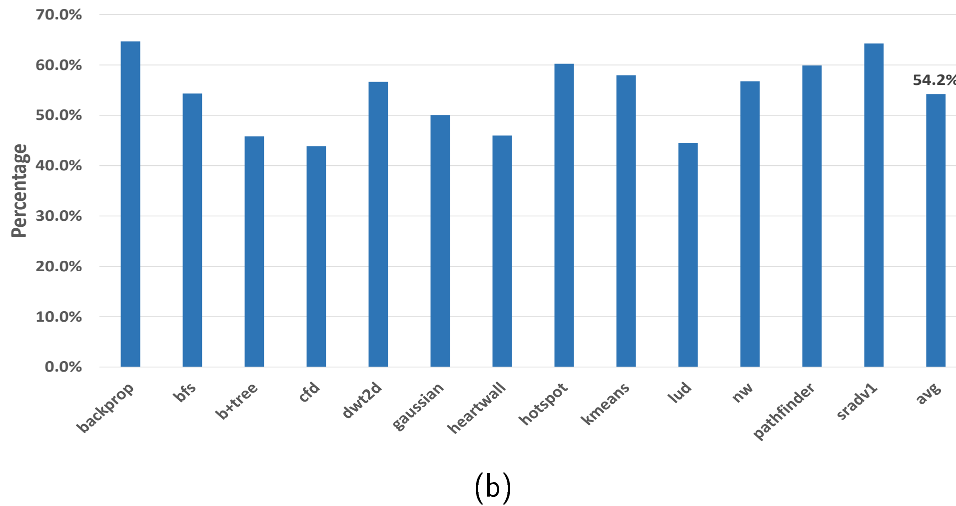
Figure 3.3: Width distribution of GPU register file warp accesses classified into two groups: accesses that require 4-byte per thread (full width) and accesses that require less than 4-byte per thread (narrow width) (a) source operands width distribution (b) destination operands width distribution.

of the bytes having either all zeros or all ones). Across the benchmarks we used, only 27.3% of the source operands and 29.2% of the destination operands in executing warp instructions require a full register width on average.

This shows that there exists a significant amount of narrow-width source and destination operands that are subject to register file access coalescing. Our proposed design enables two requests targeting the same register file bank or operand collector unit to be coalesced into a single request if the combined size of the two requests is no more than a full register



(a)



(b)

Figure 3.4: Unused register file bandwidth on (a) register file banks access (b) collector units write access.

width. The result also motivates access coalescing support not only among narrow-width read requests but also among write requests, or a mix of read and write requests as both reads and writes have a significant number of narrow-width requests that may be serialized due to limited access ports on register file banks and operand collector units.

3.3.2 Register File Bandwidth

Narrow-width read and write accesses do not fully utilize the register file bandwidth as they carry unneeded bits in their value. Fig. 3.4 shows the wasted bandwidth for register

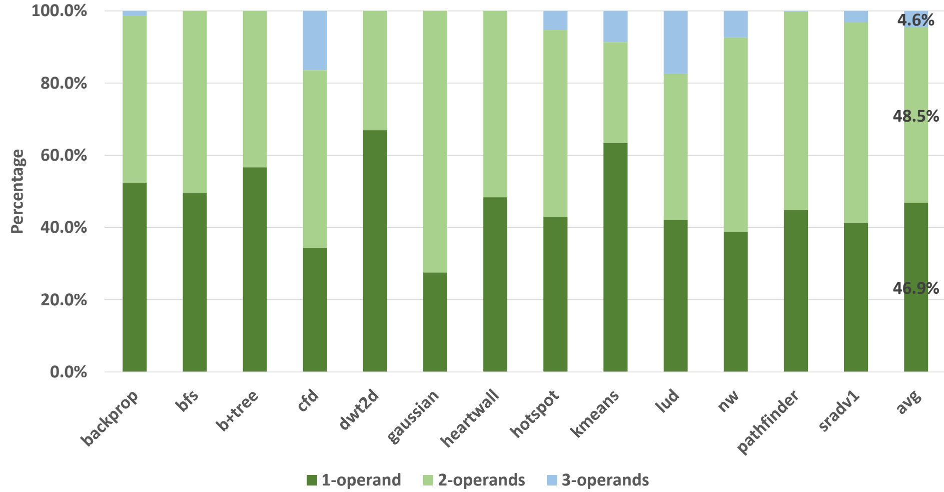


Figure 3.5: Percentage of the number of source register operands in warp instructions. A given warp instruction can have one, two, or three source register operands.

file banks (Fig. 3.4a) and operand collectors (Fig. 3.4b) due to narrow-width accesses. In addition, the unneeded bits in the narrow-width values waste dynamic energy on every read and write to the register file and when they propagate down into the operand collector units.

Our proposed design aims at utilizing the register file bandwidth more efficiently by coalescing multiple narrow-width accesses targeting a register file bank or an operand collector unit into a single physical access that would improve bandwidth utilization and reduce the register file pressure. Our design is not restricted to only support coalescing read accesses from the same wrap instruction as we take advantage of all coalescing opportunities on read and write requests initiated from same or different warp instructions to achieve high bandwidth efficiency.

3.3.3 Warp Instruction Operands

Warp instructions collect their needed operands from the register file over multiple cycles before they can be dispatched into the Single Instruction Multiple Data (SIMD) execution pipeline. A warp instruction may require one, two, or three source operands to read from the register file depending on its type or class. Fig. 3.5 shows the percentage of warp instructions

with different number of source operands. The result shows a significant percentage of warp instructions that require a single register source operand. Examples of such instructions are memory stores, data copy or permutation, and arithmetic or logical operations with immediate values.

Register coalescing that is restricted to only coalesce read accesses from the same warp instruction overlooks a significant percentage of register accesses initiated from single-operand warp instructions that can be coalescable targets. The high percentage of single-operand warp instructions motivates register coalescing among different warp instructions targeting the same register file bank to further reduce the number of bank accesses and register file pressure which results in performance and energy efficiency improvements. Our design supports register access coalescing for read as well as write requests originated from the same or different warp instructions with no restrictions.

3.4 Promoting Coalescing Opportunities

Our design takes advantage of the presence of frequent narrow-width data in general-purpose compute applications and provides coalescing support in order to reduce the number of register file accesses, improve performance, and energy efficiency in the GPU. A narrow-width operand requires fewer bytes per 4-byte thread to be fully represented without any loss of information with one or more of the upper bytes carrying only the sign information (most significant bits are all zeros or all ones). We call a narrow-width operand for a given warp as an N-byte operand if every thread within the warp requires no more than N-byte to be represented. Access coalescing arise when two narrow-width requests having a size of N-byte and M-byte, for instance, target the same shared resource and their total size $N + M <= 4$. That is, their combined size is no more than a full-sized operand of 128B or, in consistent term, a 4-byte operand.

Coalescing these narrow-width requests targeting the same bank can be made possible only if two requests can target different sub-banks within the bank and also having each sub-bank to be controlled separately. In this case, the N-byte request would target N number

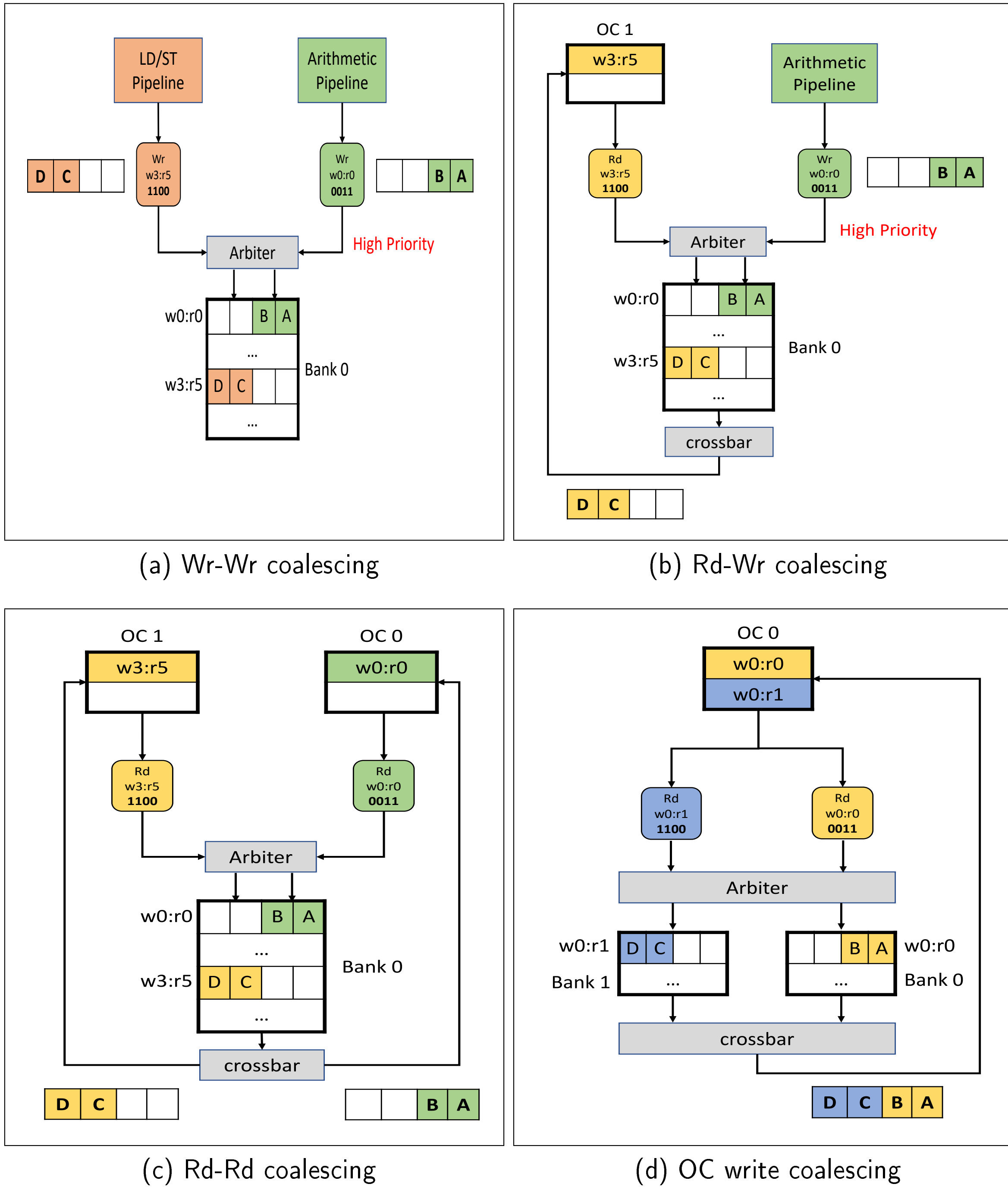


Figure 3.6: Examples of register coalescing on register file bank and operand collector accesses (a) Access coalescing of two write requests. (b) Access coalescing of a read and a write request. (c) Access coalescing of two read requests. (d) Access coalescing of two writes to an operand collector unit.

of the sub-banks in one register entry and the M-byte request would target M of the sub-banks in another register entry. And the two, N and M, subsets of sub-banks across the two entries are mutually exclusive. In other words, both requests target non-overlapping sub-banks within a bank across different entries. This implies that the needed narrow-width data for read requests is available in a subset of sub-banks in the targeted register entry. It also implies that register entries in a given bank are aligned differently to have one entry, for example, occupies the lower N sub-banks and another entry occupies the upper M sub-banks.

Coalescing two read requests from one operand collector that target two different banks can also be made possible, with low design cost, if the two requests target non-overlapping sub-banks across the two banks. For instance, if an N-byte request targets bank 0 and an M-byte request targets bank 1 at the same time, the read data from each bank can be simply combined, with no permutations and needing extra MUXing, and guaranteed to fit into a single 4-byte output data that can be written into the operand collector in the same cycle. This can be made possible by having separate control for every 1-byte of input data to the collector unit instead of controlling the 4-byte input data the same way such that each 1-byte can be selected from any of the 4 banks.

Recall that multiple requests targeting a register file bank at the same time experience *port conflict* and their access is serialized as shown in Fig. 3.2. In Fig. 3.6, we illustrate access coalescing examples supported by our design for narrow-width read and write requests that can access a register file bank or an operand collector unit at the same time and eliminate port conflicts and access serialization between those requests. Fig. 3.6a shows two narrow-width write requests access a bank at the same time each of which writes to different register entry within the bank. Fig. 3.6b shows narrow-width read and write requests accessing a bank at the same time and Fig. 3.6c shows two narrow-width read requests from different warp instructions accessing a bank at the same time. In Fig. 3.6d, two narrow-width read requests targeting different banks were able to write their data into an operand collector unit at the same time.

Such access coalescing improves performance and energy efficiency in the GPU in different ways including:

1. Improve memory writebank bandwidth.

2. Enable dispatching ready instructions into execution pipeline sooner.

3. Help freeing up operand collector units sooner and increasing warp instruction issue rate.

4. Reduce data dependency stalls and allows new warp instructions to issue faster.

5. Reduce register file pressure and improves access latency.

3.5 Related Work

Significant prior work exists on improving performance or energy efficiency for GPU register file. In this section, we first highlight different techniques proposed to improve performance or reduce power consumption for GPU register file and then present related work on register file coalescing in GPU.

3.5.1 Non-coalescing Techniques

Gebhart et al. proposed a small register file cache (RFC) to capture short-lived registers which would reduce read and write accesses to the main register file and reduce its dynamic power consumption with a small impact on performance [15]. Sadrosadati et al. proposed using an RFC with software managed register prefetching to tolerate access latency of a larger register file and improve overall performance at the cost of using higher power register file [22].

A compile-time managed hierarchical register file is proposed by Gebhart et al. with the aim of reducing dynamic power consumption [23]. In this work, the register file is partitioned into multiple levels and the compiler is used to leverage its knowledge of registers usage to determine where to allocate values across the register file hierarchy. Similar work that used

both a register file cache and a hierarchical register file is proposed by Bailey et al. [24]. A unified local memory structure with partitioning of capacity among register file, data cache, and scratchpad memory is proposed by Gebhart et al [25].

A partitioned register file is proposed by Abdel-Majeed et al. where less frequent accessed registers are placed in a slow register file that operates in a lower voltage and frequently accessed registers are placed in a small and fast register file [26]. The technique targeted both dynamic and leakage power reduction and both compile-time and run-time profiling had to be used to collect register access statistics needed. Abdel-Majeed et al. also aimed at reducing leakage power by operating the register file in different power modes [27]. They proposed using an active mask gating on the register file to reduce dynamic power consumption which is done on 128B entries using Divided Word Line (DWL) approach previously proposed by Yoshimoto et al. [28].

Jeon et al. proposed register file virtualization to reduce the number of physical register file entries used and gate off unused entries to reduce power consumption [29]. Kloosterman et al. proposed replacing the main register file with a smaller size operand staging unit to reduce power consumption while providing similar performance[30]. Operands are allocated space in the staging unit using compiler annotations that determine future registers usage. Registers are all kept in memory and fetched into the staging unit when needed.

Data compression has been proposed by Lee et al. for GPU register files to reduce dynamic power by gating off unused sub-banks [31]. This work used the Delta-Base-Immediate (DBI) compression technique that Pekhimenko et al. proposed for data caches [32]. Applying the DBI mechanism on the GPU register file incurs high area and power overheads as it requires adding a vector-wide adder-subtractor units to compress and de-compress operands data. Another form of data compression is proposed by Liu et al. to handle scalar execution in GPGPU where duplicate values in thread registers are captured in a separate scalar buffer to save access power [33] at the cost of small performance loss.

Register file packing technique for GPUs has been proposed by Wang et al. [34] and

Ergin et al. [35] to take advantage of narrow-width data to reduce the number of physical register entries used and gate off unused entries. In this technique, two narrow-width registers can be combined and placed in a single physical register entry which requires register renaming. Although it reduces the physical size of the register file, the proposed register packing techniques do not coalesce register accesses. Each register, packed or non-packed, still requires a separate read access which does not provide any performance benefit.

Data-path slicing is proposed by Gilani et al. to take advantage of low dynamic range values that can be represented by 16 bits [36]. The 32-bit thread registers are also split into low and high halves and controlled separately to reduce the dynamic access power. The main objective of this work was to increase the warp issue rate by issuing two warps with 16-bit data in same cycle which required modifying the warp scheduler and register file banking scheme. Khorasani et al. proposed time-sharing a subset of physical registers between executing warps to improve performance [37]. Oh et al. proposed increasing the number of concurrent thread blocks to provide performance improvement. The register file is partitioned into two regions, one for active thread blocks to use and another region for pending thread blocks.

All of these proposed power reduction and performance improvement techniques are orthogonal and potentially complementary to register coalescing technique we propose in this work which takes advantage of the frequent narrow-width data to provide performance and energy efficiency improvements in GPUs.

3.5.2 Register Coalescing Techniques

Similar to the concept of memory coalescing where multiple memory requests targeting contiguous memory locations can be combined into a single request, Esfeden et al. [2] have recently introduced a Coalescing Operand Register File (CORF) for GPUs to combine read requests from the same warp instruction into a single bank request. This work is the first and the only one available in literature, as of today, on GPU register file coalescing. We will present this prior work in more details and highlight the many design limitations and

overheads it has.

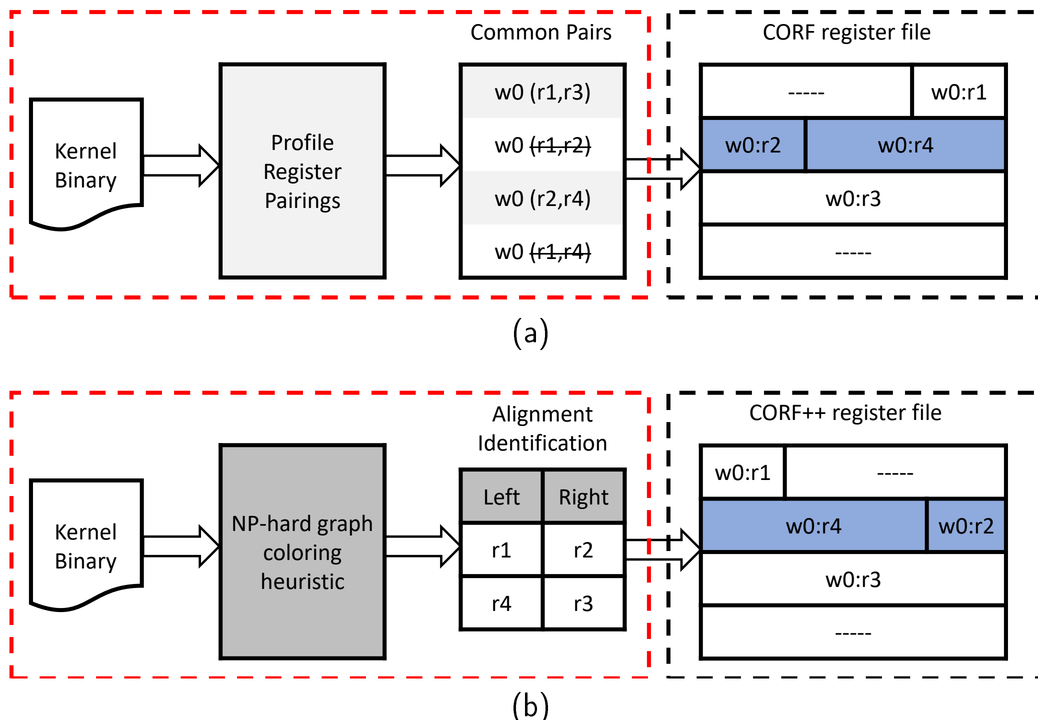


Figure 3.7: CORF design overview (a) limited CORF with read coalescing support within the same physical register entry (b) enhanced CORF++ with read coalescing support across two register entries within a bank. Reprinted from [2].

The prior work introduced two register coalescing design flavors, a limited CORF and an enhanced CORF++ design, illustrated in Fig. 3.7, in order to improve overall performance and energy efficiency in GPUs. CORF designs have the following specifications:

- Register coalescing is built on top of register packing [34, 35] which allows two narrow-width registers to coexists in a single physical register entry.
- Use compiler assistance to guide register allocation decisions to help promote coalescing opportunities. Common register pairs used for a given warp instructions are identified through register profiling and passed to the hardware to guide allocating each common pair within the same physical entry (CORF). An NP-hard graph coloring heuristic is

used to hint register alignment (left or right alignment) within the physical register entry to help guide register allocation to promote read coalescing across two physical entries.

- Support coalescing of two read requests from the same warp instruction that target the same bank and are located in the same physical entry (CORF) or cross two physical entries in non-overlapping sub-banks (CORF++).
- Register file virtualization (renaming) is necessary to support register read coalescing.

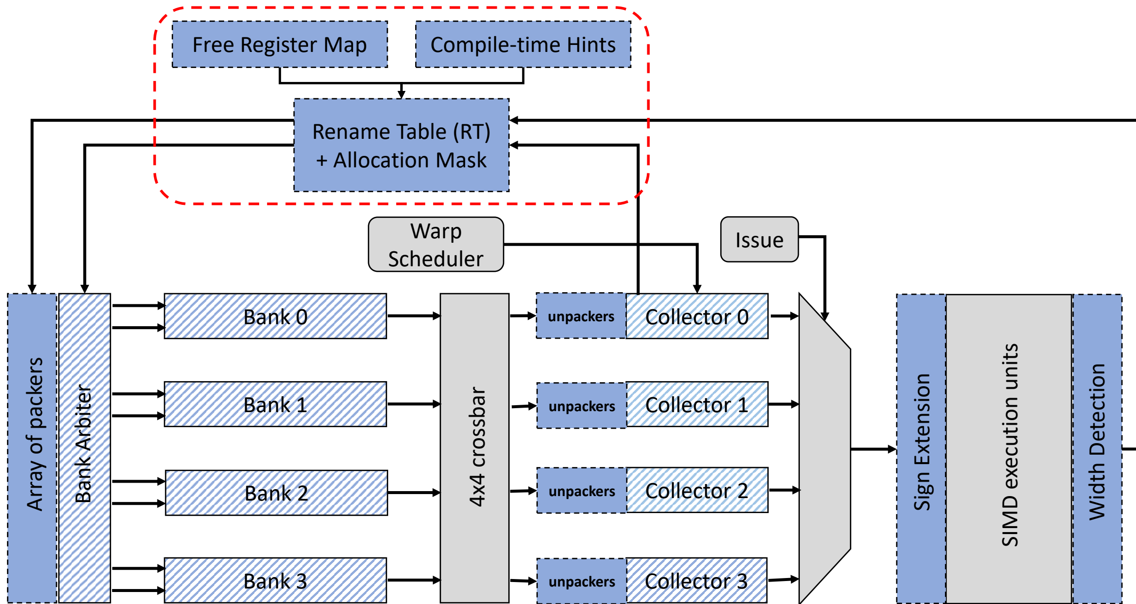


Figure 3.8: GPU register file design used in CORF. Reprinted from [2].

Fig. 3.8 shows the GPU register file design used in CORF. In Table 3.1, we summarize the design overheads for CORF in comparison to our low-cost design. We propose a hardware-only design that support register access coalescing without the need for the complexity of compile-time hints used in CORF design for the following reasons:

Design Overhead	CORF/CORF++	Our Design
Require register packing	Yes	No
Require register renaming	Yes	No
Require compile-time assistance	Yes	No
Require byte-level vector-wide shifters	Yes	No
Require specific register file layout	Yes	No

Table 3.1: Design overhead of CORF compared with our low-cost design.

1. The compile-time approach is lacking information about registers widths, and therefore, the hints provided may or may not be useful.

2. Estimating the dynamic frequency of occurrence for each warp instruction is a difficult problem at compile time due to loops that may not be resolvable, and therefore, the hints are approximated based on heuristics.

3. Finding the common register pairs or proper register alignment are graph coloring problems that are difficult to solve and only heuristics can be given.

4. Registers width is subject to change during the course of kernal execution and the initial register allocation may have to change.

5. Our design supports cross warps access coalescing for reads as well as write requests and providing compile-time hints for these cases is impractical due to dynamic behavior of warps at run-time.

Our proposed design also avoids the complexity and overhead of register packing and virtualization (renaming) needed in CORF design for the following reasons:

1. It does not restrict register coalescing to be within a physical register entry. It supports coalescing across *any* two register entries.

2. It does not restrict register coalescing to only accesses from the same warp instruction.

It supports coalescing across *any* two register entries whether they belong to the same warp or different warps.

3. It does not restrict coalescing to only read requests. It supports coalescing a mix of read and write requests targeting *any* two register entries.

4. It does not restrict read coalescing to be within a register file bank. It supports coalescing reads from different banks for the same warp targeting the same operand collector unit.

In addition, our design avoids using variable 128B-wide shifters for register alignments used in CORF design, which are needed on the read and write sides of the register file, as they have high cost and require additional pipeline stages to be added due to their timing impact. Instead, we use a much cheaper thread-local one-level multiplexers to align warp registers. CORF design also requires a specific register file layout with all warp registers located in the same bank, as shown if Fig. 3.10a, in order to support read coalescing. In general, this layout causes more bank conflicts and negatively impacts overall IPC performance compared to another layout shown in Fig. 3.10b. Our proposed design addresses this limitation and provides register coalescing capabilities with no restriction on the register file layout used.

Despite the fact that it has high design overhead, CORF is very limited and only capable of coalescing read requests for the same warp instruction that target the same register file bank. Coalescing was only limited within the same physical register entry in the limited edition of CORF and then extended to cover coalescing across two physical register entries in CORF++. Table 3.2 compares coalescing capabilities between CORF and our proposed design. Our design is capable of supporting many register coalescing opportunities with low-cost that CORF can not support. In Section 3.7 we will compare the overhead cost of CORF compared to our design as well as compare the performance and energy results achieved.

Coalescing Support	Bank	Warp Instr.	CORF/CORF++	Our Design
2 reads	Same	Same	Yes	Yes
2 reads	Same	Different	No	Yes
2 writes	Same	—	No	Yes
1 read and 1 write	Same	—	No	Yes
2 reads	different	Same	No	Yes

Table 3.2: Register coalescing support in CORF compared with our design.

3.6 Register File Access Coalescing Design

In this section, we present the coalescing-aware register file design that we propose to improve overall performance and energy efficiency in GPUs. GPUs deploy a large register file to support massive thread-level parallelism (TLP) execution model by holding state of thousands of threads contexts and allowing fast context switching between threads. Given the large size of the register file and the width of warp registers, a multi-ported register file design is not viable due to the high area and power overheads. To this end, GPUs adapt for multi-banked single-ported register file built with 6T SRAM arrays and combined with single-ported operand collector units to provide high access bandwidth. However, due to access port limitations, multiple accesses targeting the same register file bank or operand collector unit are forced to be serialized due to port conflicts which may negatively impact performance. On the other hand, our access coalescing design is providing the ability to combine two requests targeting a limited shared resource (register file bank port or operand collector port) into a single request and eliminate the access serialization penalty that they would have had due to shared resource conflicts.

Inspired by the CORF design [2], our proposed design is the first to provide register coalescing capabilities for multiple access scenarios including read and write requests, from same or different warp instructions, accessing the same bank as well as read requests to different banks targeting the same operand collector unit. Access coalescing enables performance improvements by reducing the number of read and write operations, reducing overall

register file pressure, reducing access serialization penalties due to port conflicts on register file banks and operand collector units. In addition, access coalescing also reduces register file dynamic power consumption as it reduces the number of read and write operations to the register file and reduces writes to the operand collector units and also improves their bandwidth utilization. As a result to the performance and dynamic power enhancements, overall GPU energy efficiency is improved.

3.6.1 Design Overview

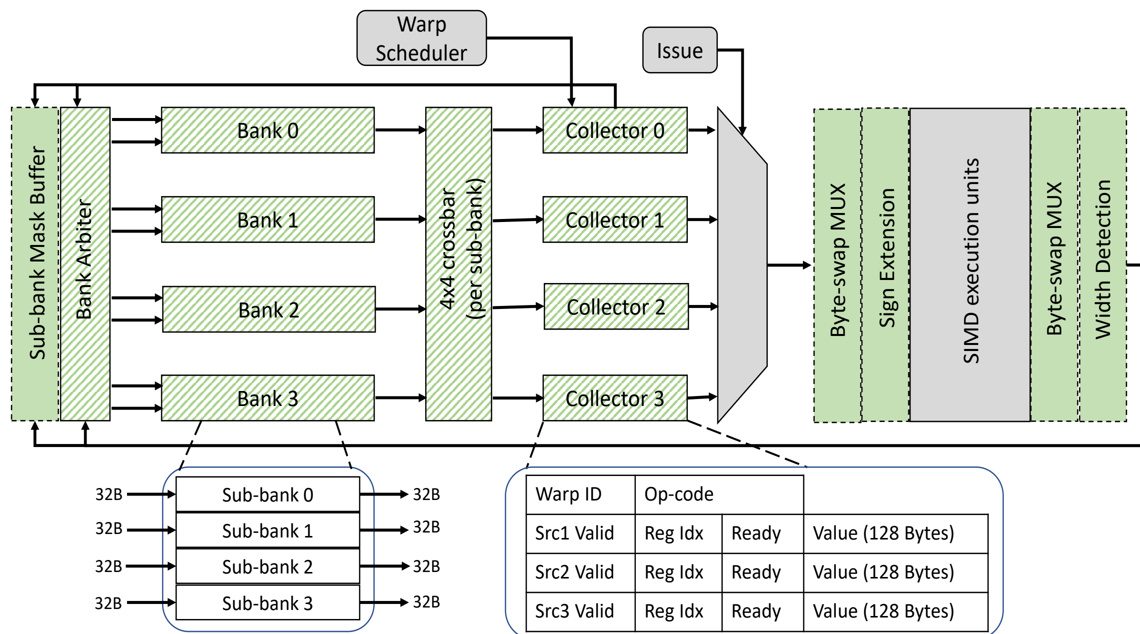


Figure 3.9: Our coalescing-aware GPU register file design.

The baseline register file for the Fermi GPUs has a total size of 128KB per SM and is divided equally into four banks. Each bank has 256 entries and each entry is 128B wide to hold a warp register of this size. Physically, the bank is constructed using four narrow sub-banks where each sub-bank holds a 32B slice of every bank entry. To read out a full 128B

warp register from a given bank, the same entry is indexed in the four physical sub-banks to retrieve the 128B register. Each bank offers a single access port to either read or write a full register in a cycle and has the four sub-banks controlled the same way to either all read or write the same indexed entry. Operand collector units are need in this multi-banked organization as a warp source operands may take multiple cycles to be read out from the register file banks due to port conflicts. Each operand collector offers a single write port that can be used by one bank at a time. The register file banks are connected to the operand collectors using a 4×4 crossbar network with 128B links. A register file arbiter is responsible for prioritizing banks and operand collectors accesses, which may experience conflicts due to limited access ports, in a way that only one bank access is allowed in a cycle and only one collector unit write is allowed in a cycle.

Fig. 3.9 highlights the micro-architectural enhancements for our coalescing-aware register file design. The following is a summary of the enhancements made to support access coalescing:

- Enabled the four sub-banks within a bank to be controlled separately.
- Provided a bank with dual-access controls to target different sub-banks.
- Enabled crossbar controls per 32B of output data.
- Enabled operand collectors to have separate write controls per 32B of input data.
- Extended the register file arbiter to support access coalescing for register banks and collector units.
- Added a register width detection logic and storage.
- Added thread-local multiplexers for register alignment.
- Allowed banks and operand collectors to operate on coalescing-friendly data format.

Recall that our design avoids compile-time hints to (left or right) align narrow-width registers and register packing and renaming as used in CORF design due to the many reasons mentioned previously in Section 3.5.2. Instead, our design supports a low-cost hardware-only solution for register alignment that avoided any need for register packing and renaming and their high design overhead. In our design, warp registers entries have fixed alignment based on their even/odd number where even numbered entries are right aligned and odd numbered entries are left aligned. As we will show in Section 3.7.5, our low-cost fixed alignment scheme is within a 2.2% of performance speedup to an *unrealistic* ideal alignment bound and provides higher performance speedup compared to CORF design as it supports far more coalescing capabilities.

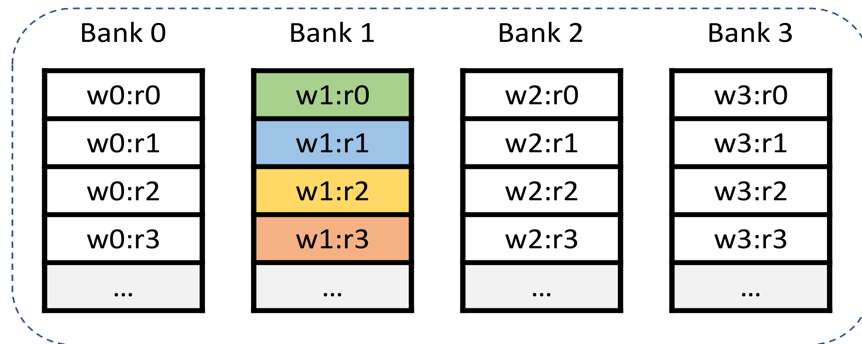
As we mentioned, our coalescing-aware register file design addresses the limitations and avoids the complexity and overhead of the prior CORF design. Our low-cost design provides access coalescing capabilities not only to read requests from same warp instruction targeting the same bank, which is the only scenario CORF is capable of, but also to a variety of access scenarios including:

- Read requests from same warp instruction targeting the same bank.
- Read requests from different warp instructions targeting the same bank.
- Write requests targeting the same bank.
- A mix of read and write requests targeting the same bank.
- Read requests from same warp instruction (same operand collector) targeting different banks.

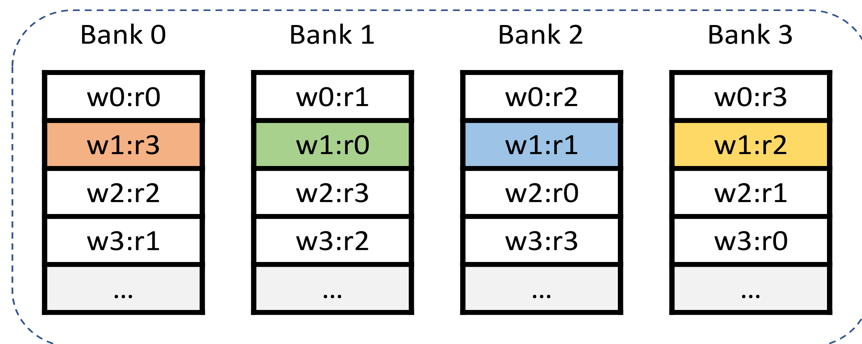
In the following subsections, we present implementation details of our proposed register coalescing design.

3.6.2 Coalescing-aware Register File Organization

3.6.2.1 Registers Layout (Register to Bank Mapping)



(a)



(b)

Figure 3.10: Registers to banks mapping (register layouts): (a) all registers belonging to the same warp are mapped into one register file bank and (b) warp registers are interleaved across register file banks.

Fig. 3.10 shows two possible layouts of mapping warp registers into register file banks. In Fig. 3.10a, all registers for a given warp are mapping into the same bank with a mapping function $bank = warp_id$ and we refer to this layout as *wid_layout*. The other layout in Fig. 3.10b shows registers for a given warp interleaved across the banks using a mapping function $bank = warp_id + reg_id$ and we call this layout *wshift_layout*. The wshift_layout

has been used in GPUs as it minimizes bank conflicts cross warps.

One limitation of CORF design is the requirement of having all registers for a warp map into the same register file bank as in Fig. 3.10a. This limitation is due to the fact that CORF can only support register coalescing for read requests if they belong to the same warp and target the same bank. Therefore, in order for CORF to increase the coalescing opportunities for its limited design, it had to move to a less efficient register file layout.

In our design, we addressed this limitation and supported register coalescing that can work on *any* register file layout. Our design is capable of coalescing requests from same or different warp instructions targeting the same bank and it also capable of coalescing read requests for the same warp instruction (from the same operand collector) targeting two different banks. Even with the register layout used in CORF, the wid_layout, our design provides more coalescing capabilities that CORF can not support.

3.6.2.2 Register File Bank

Recall that the register file is divided up into multiple banks, four banks in the Fermi GPUs, with each one of these banks having 256 entries each of which holds a warp register of size 128B. Each bank is physically divided into four narrow-width sub-banks with each sub-bank having 256 entries of size 32B. A warp register entry is spread cross the four sub-banks and indexed with the same entry number in every sub-bank. A warp register carries data for all 32 threads within the warp where each thread has a 4B slice of the data. In baseline GPU, warp register data is represented in a *byte-interleaved* format. As shown in Fig. 3.11a, warp register data starts with the four bytes of thread 0 (T0.B0, T0.B1, T0.B2, T0.B3), then the four bytes of thread 1 (T1.B0, T1.B1, T1.B2, T1.B3), and so on. Each of the sub-banks holds 32B of the warp register data with sub-bank 0 holds data for the first 8 threads within the warp, sub-bank 1 holds data for the second 8 threads, and so on.

With this data format, a narrow-width register that uses, for instance, only B0 of every warp-thread would require accessing all four sub-banks to read or write the warp register as B0 for the 32 threads are spread across the entire register entry. This implies that, for

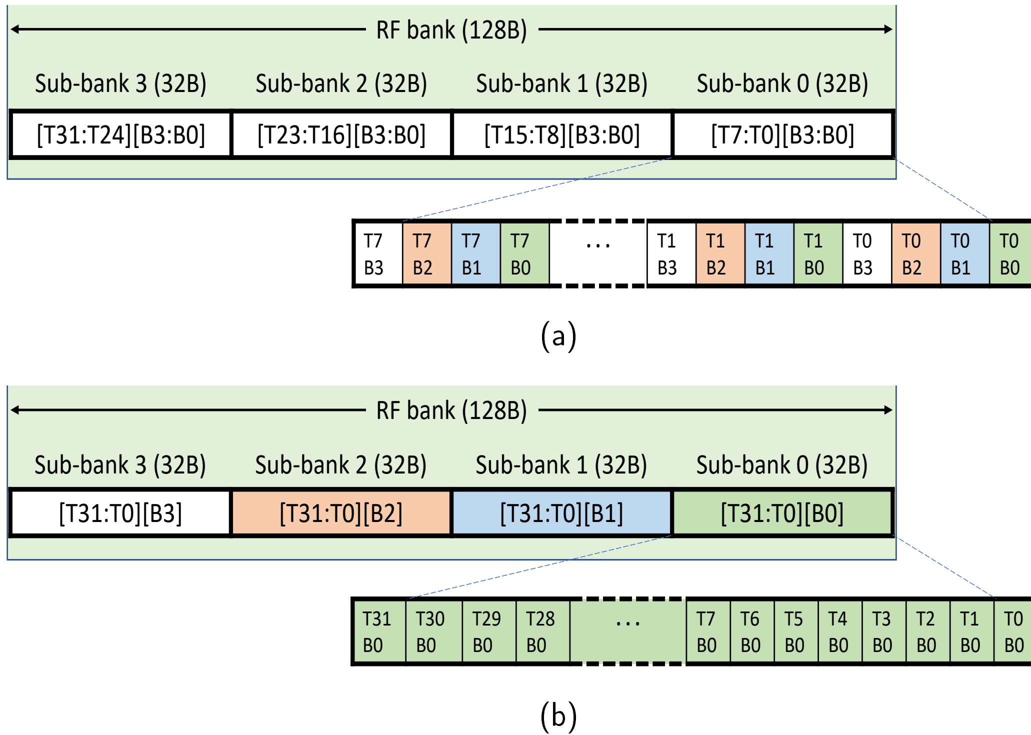


Figure 3.11: Data format of a 128B warp register within an RF bank entry: (a) Byte-interleaved format: each 32B sub-bank entry holds data for 8 threads in the warp and (b) Thread-interleaved format (supports register coalescing): each 32B sub-bank entry holds 1-byte (same byte number) for every thread in the warp.

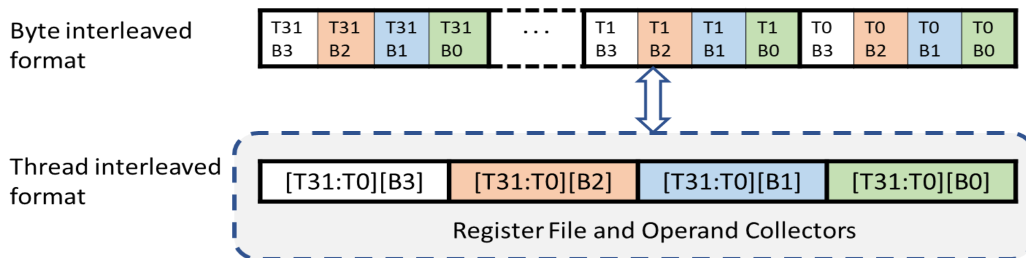


Figure 3.12: Register data representation within the register file and operand collectors and outside. Switching data from one format to the other is done through wiring bytes into different byte-position (no logic cost).

narrow-width data, all four sub-banks resources are fully consumed, however, they are poorly utilized as each sub-bank provides only a portion of its effective bandwidth. Fundamental to

register coalescing, is the ability for two narrow-width requests to be able to access different sub-banks at the same time to reduce the number of requests made to the register file, increase bank bandwidth utilization, and ultimately improve overall performance and energy efficiency in the GPU. Therefore, we changed the data format within warp registers to follow a *thread-interleaved* format instead. With this format, as showin in Fig. 3.11b, warp register data starts with B0 for all the 32 threads in the warp, then B1 for all 32 threads, and so on. Each sub-bank holds 32B of the warp register data with sub-bank 0 holds B0 for all 32 threads, sub-bank 1 holds B1 for all 32 threads, and so on. This allows for narrow-width request of 1-byte per thread, for instance, to consume and fully utilize only one of the four sub-banks which set the other three sub-banks free that we can utilize through register coalescing.

Moving from one data format to the other is done at no cost as it only requires data bytes to be wired into different byte-positions. Fig. 3.12 shows that the register file and operand collectors use thread_interleaved format whereas data outside those components are represented in its original form using byte_interleaved format. That is, data result will change to thread_interleaved format as it is written into the register file and source operands data will change to byte_interleaved format once issued out from an operand collector unit.

3.6.2.3 Register Alignment

With the thread_interleaved data format we used in the register file bank, a narrow-width register has its data spans only a subset of the four sub-banks. For instance, an N-byte register holds its data in only N of the sub-banks. With separate sub-bank control, the remaining sub-banks can be either (1) gated off when reading and writing the warp register to reduce register file access power as we will present in Chapter 4 or (2) utilized by another (coalesced) request to the bank that access another narrow-width register that is fully located in these remaining sub-banks. This implies that, in order to coalesce two requests, the targeted register entries would need to be occupying non-overlapping sub-banks with one entry, lets say, occupies sub-bank 0 and sub-bank 1, and the other entry occupies

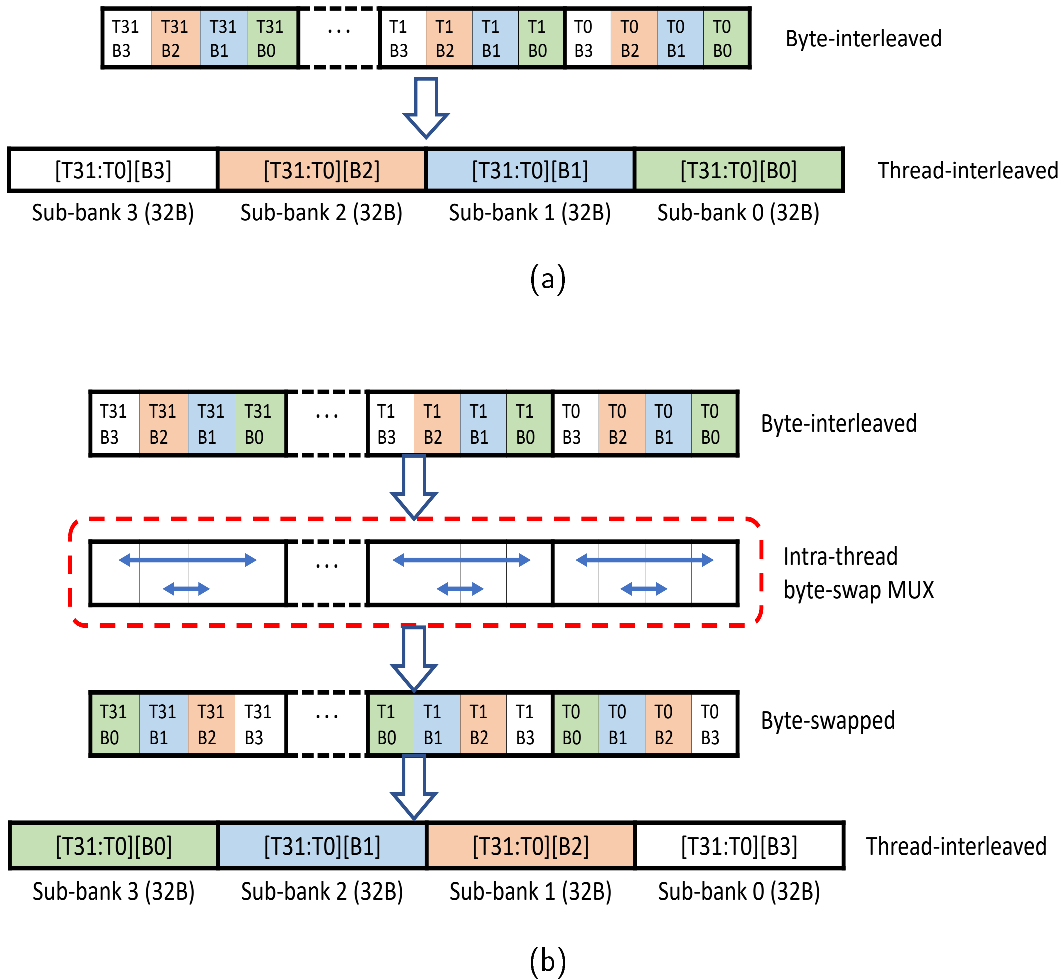


Figure 3.13: Warp register data alignment: (a) Default right-alignment with byte 0 for all 32 threads map to sub-bank 0 and (b) Left alignment using intra-thread byte-swap MUX with byte 0 for all 32 threads map to sub-bank 3.

sub-bank 2 and sub-bank 3.

Representing the narrow-width register data in thread_interleaved format would makes the data to be, *by default*, right-aligned to use only the lower sub-banks within the register entry. For instance, a 1-byte operand will map to sub-bank 0 to be read or written into the register file by default. To support register coalescing, however, other narrow-width registers would need to be left-aligned or map to the upper sub-banks such that the two coalesced requests would access non-overlapping sub-banks across two register entries.

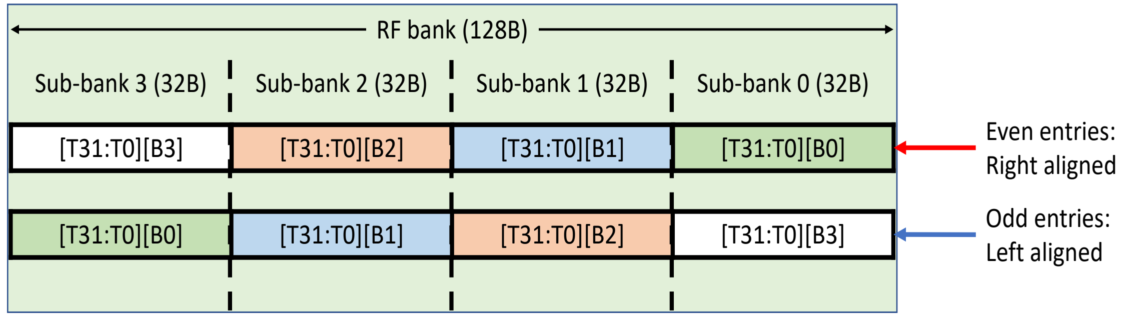


Figure 3.14: Data alignment of warp registers within a register file bank based on even/odd register entry number. Even registers are right aligned with byte 0 of all threads map to sub-bank 0. Odd registers are left aligned (byte swapped) with byte 0 of all threads map to sub-bank 3.

The requirement for data-alignment was addressed in CORF design by shifting up a narrow-width data when writing into the register file to left-align it and re-align the data when it is read out of an operand collector unit by shifting its bytes down. On the write side of the register file, two byte-level shifters were needed to left-align the write data and, on the read side, two byte-level shifters were used to re-align the data for each operand collector unit for a total of ten 128B data shifters used in CORF design. Our design addresses this requirement but in a much cheaper way based on the fact that register file data does not have to be represented in a specific order (*ie.* Little-endian order). Therefore, instead of shifting the data across the warp to, left or right, align it, our design just byte-swaps the narrow-width data such that byte 0 of every thread maps to sub-bank 3, byte 1 of every thread maps to sub-bank 2, and so on.

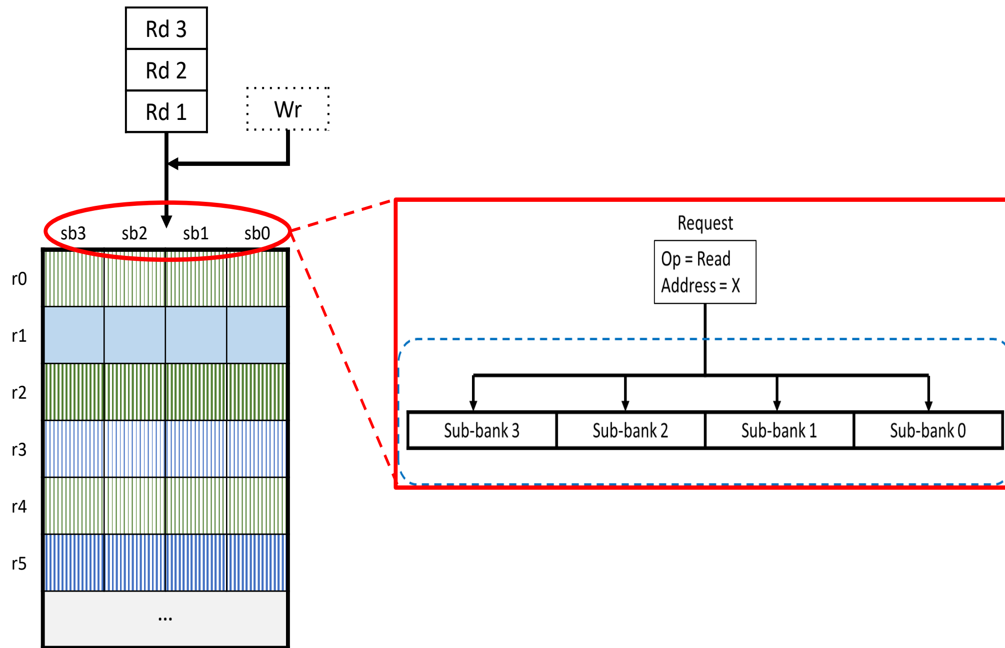
Fig. 3.13 shows a right and left aligned register entries in our design. The warp data is right-aligned by default when represented in thread_interleaved format. To left align the data, we used a single-stage MUX that is local to every thread to swap the bytes within the thread to be in the opposite order with byte 0 in the most significant byte-position and byte 3 in the least significant position. With this intra-thread byte swapping, the warp data

will be left-aligned when represented in thread_interleaved format. Compared to the costly byte-level shifters used in CORF which require a deep MUXing tree and shifting across the entire warp, our approach is much cheaper and only requires a single-stage MUXing local to each thread in the warp with no data movement across the warp threads. We also reduced the cost of data-alignment even further by using fewer byte-swap MUXing on the read side by moving them out of the operand collector units into the first stage of the execution datapath as the timing impact of inserting a single-stage MUX (2 logic-levels) is very minimal. Our design uses a total of five byte-swap MUXes compared to ten byte-level shifters used in CORF design.

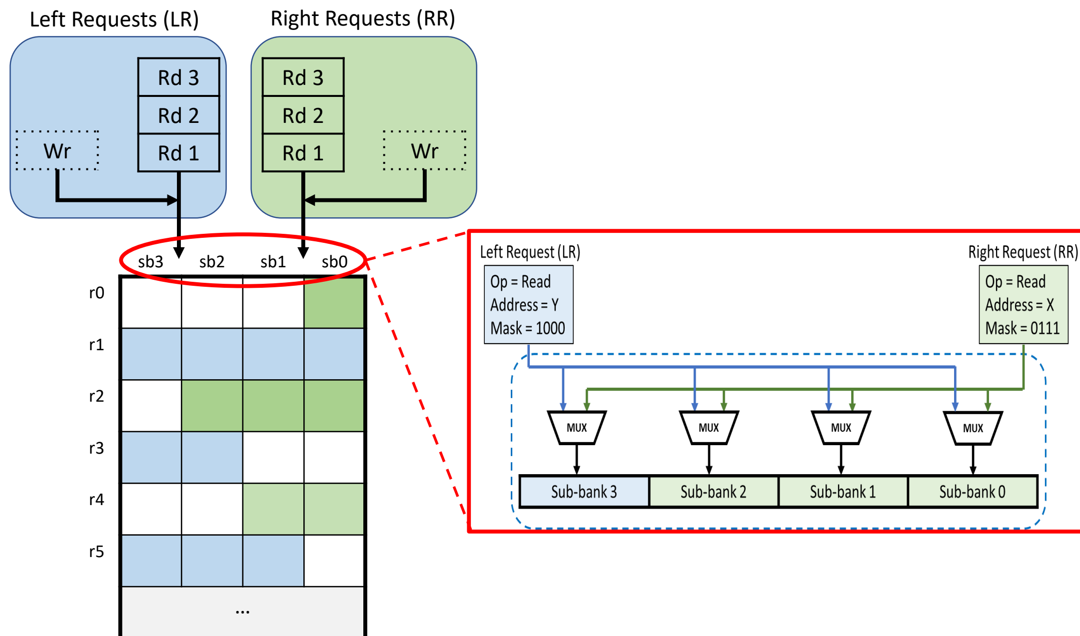
As mentioned earlier, CORF design adopted compile-time hints to guide data alignment for every warp register. Besides the complexity and weaknesses of this approach as we mentioned, it would not provide meaningful hints in our design as we support register coalescing capabilities across different warps as well as for different read and write accesses that are dynamically detected and handled by hardware. Therefore, our design adopted a hardware-only solution to guide data alignment of warp registers based on their register numbers with even registers being right aligned and odd registers being left aligned (byte swapped) as shown in Fig. 3.14. The main advantage of our approach is its very low cost as it only requires a single-stage byte-swap MUX to left align the data for odd registers and avoids all the cost and overhead associated with dynamic allocation and register renaming logic used in CORF design. As we will see in the following Section, our low-cost design provided higher benefits by supporting many register coalescing scenarios whereas CORF had to adapt a high overhead design to improve coalescing opportunities for the very limited register coalescing scenario it supports.

3.6.2.4 Dual-access Banks

With the support of data alignment and formatting, narrow-width registers in a given bank occupy only a subset of the sub-banks with even registers occupying the lower sub-banks and odd registers occupying the higher sub-banks. This made possible for two registers, an



(a)



(b)

Figure 3.15: Comparison between basline register file bank and our coalescing-aware bank: (a) Baseline bank with single-access support (b) Our dual-access bank with left and right requests that can access two register entries with different data alignments in non-overlapping sub-banks.

Dual-address **banks used in CORF**	***Dual-access*** **banks used in our design**
Coalesced requests should be of the same type. Only two read requests are supported	Coalesced requests can be of same or different types. Any combination of two read or write requests are supported
Coalesced requests should belong to same warp instruction	Coalesced requests can be from same or different warp instructions

Table 3.3: Comparison between CORF dual-address banks and our new dual-access banks.

even and odd registers, to be accessed at the same time given that their data reside in non-overlapping sub-banks. For example, a read from R_2 which is a 1-byte register occupying sub-bank 0 and a write to R_3 with 3-byte result that would access sub-banks 1, 2, and 3 can be coalesced and done in the same cycle.

In order to support register coalescing, we provided a *dual-access* support for register file banks with a right and left accesses that can target different sub-banks within a bank as shown in Fig. 3.15b. To improve coalescing opportunities, we made access requests that target even warp registers be steered as right requests (RR) and requests accessing odd registers be steered as left requests (LR) as targeting two even (or odd) registers at the same time would always result in a sub-bank conflict.

A left or right request can be either a read or a write access from any warp instruction that would specify the register entry number being accessed and also the width of the register data to determine which of the four sub-banks to be accessed. The register width information is supplied as a four-bit mask that indicates the sub-banks begin targeted by the request. Two bank requests, a left and a right requests, can be coalesced if the ANDing of their masks is zero which means they do not conflict on any sub-bank access. For example, a right request with $mask = 0111$ and a left request with a mask value $mask = 1100$ can not be coalesced as they conflict on sub-bank 2. Right and left requests are MUXed into every

sub-bank based on the value of their mask bits. Each bank relies on the register file arbiter to resolve conflicts between access requests such that conflicts are not seen on coalesced, right and left, accesses.

Table 3.3 lists the main differences between the *dual-address* register file bank used in CORF design and our *dual-access* bank design.

3.6.2.5 Register File Bank Arbiter

With limited access ports on register file banks and operand collector units, multiple read or write requests that target the same shared resource would be conflicting and a priority scheme would need to be applied to resolve such conflicts. In baseline GPU, a matrix arbiter is used to manage conflicting request cases, as shown in Fig. 3.16a, with the shared resources being the register file banks and operand collector units as each of which can only serve a single request. The arbiter manages two types of conflicts that might exist between register file requests: (1) *a bank conflict* between multiple requests targeting the same bank. (2) *an OC conflict* between multiple requests targeting the same operand collector unit.

In the GPU, the following priority order is applied for accessing register file shared resources:

- Writes from execution units pipeline have higher priority over writes from memory units. Memory unit requests are blocked and wait to get access. This includes writes back from data caches or shared memory.
- Write requests, from any source, have higher priority accessing a shared resource over read requests. Blocked read requests have to wait in a queue destined for the shared resource to get access.
- Read requests *that are not blocked by writes* arbitrate for banks and operand collectors access using the priority scheme used in the matrix arbiter.

The matrix arbiter used in GPUs is similar to the Wrapped WaveFront Arbiter (WWFA) used in network switching nodes [38]. Priority in this arbiter is given to one *wrapped* diagonal

	OC 0	OC 1	OC 2	OC 3
Bank 0		1	1	
Bank 1	1			1
Bank 2	1		1	
Bank 3		1		1

Bank conflict

Operand collector conflict

(a)

		OC 0	OC 1	OC 2	OC 3
Bank 0	L			1100	
	R		0011		
Bank 1	L				1100
	R	0001			
Bank 2	L	1110			
	R			0001	
Bank 3	L		1000		
	R				0011

Coalescable requests

Coalescable requests

(b)

Figure 3.16: Register file request matrix: (a) Request matrix for baseline arbiter. Up to four requests can be granted access at the same time (b) Request matrix for our coalescing-aware arbiter. Each bank can have left or right requests with each request having a 4-bit mask to indicate the sub-banks it needs to access. Up to eight requests (four coalesced requests) can be granted access at the same time.

in the request matrix in a given cycle and then the *priority wave* propagates to the next wrapped diagonal for the next cycle. With four register file banks and four operand collectors, there are a total of four wrapped diagonals in the matrix which are marked with priorities $P0$–$P3$ with $P0$ as the initial priority diagonal as shown in Fig. 3.17.

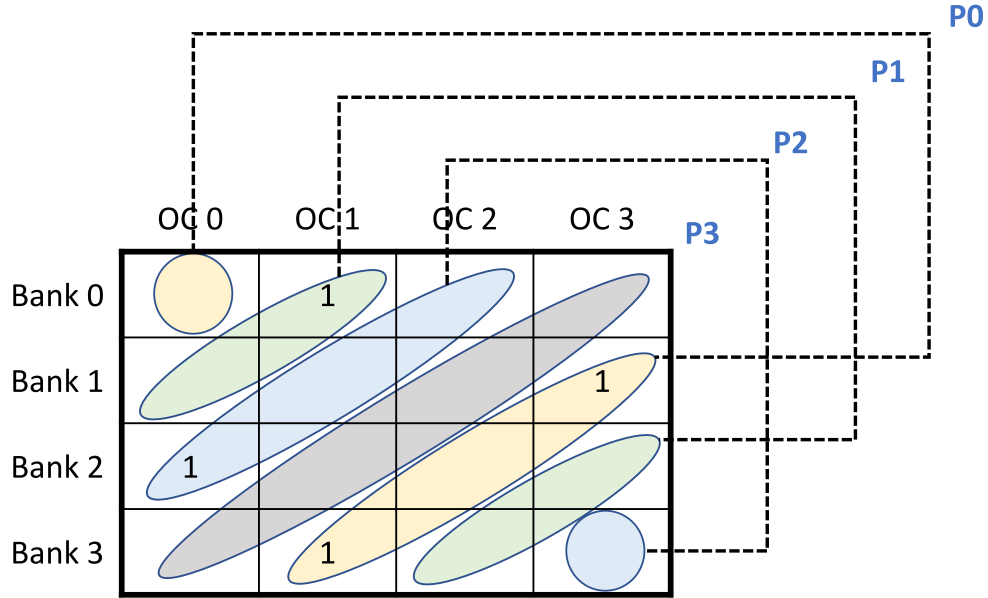


Figure 3.17: A wrapped wavefront arbitration scheme (WWFA) used in GPU register file matrix arbiter. Four priority diagonals are used $P0$–$P3$ with a priority wave initially starting at $P0$ and propagating from one diagonal to the next every cycle.

With the wavefront arbitration scheme, it would take three cycles for the *baseline* arbiter to serve the eight read requests shown in the request matrix in Fig. 3.16a. Assuming priority starts with the $P0$ diagonal, in the first cycle three requests (Bank1, OC3), (Bank2, OC2), and (Bank3, OC1) are granted access. In the second cycle, priority moves to $P1$ diagonal and another three requests (Bank0, OC1), (Bank1, OC0), and (Bank3, OC3) are granted access. Priority then moves to $P2$ diagonal in the third cycle and the remaining two requests (Bank0, OC2) and (Bank2, OC0) are granted access.

The coalescing-aware register file design, we are proposing in this work, addresses the high register file pressure and long access latency issues that negatively impact performance by providing capabilities to coalesce register file accesses into fewer physical accesses that would reduce pressure and access latency and also improve bandwidth utilization for the register file which would ultimately improves overall performance and power utilization in

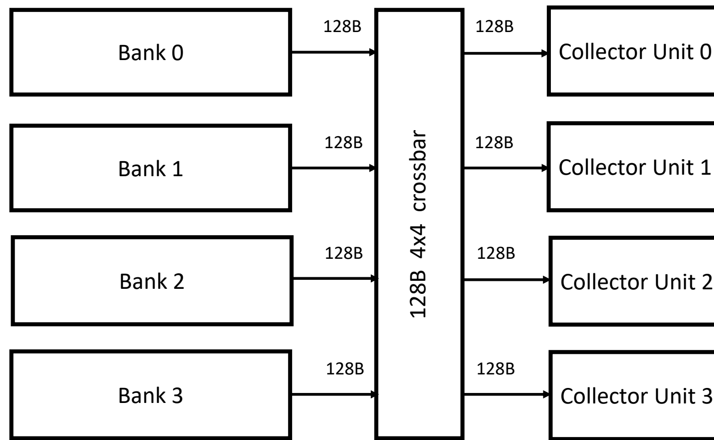
GPU. To enable for such coalescing capabilities, we expanded the matrix arbiter to account for the following: (1) each bank having a left and right accesses. (2) each request having a 4-bit mask that indicates which of the four sub-banks (and the four 32B slices for an operand collector access) are targeted within a bank. In our design, the definition of a shared resource applies to a finer-grained level where, instead of a bank being a shared resource, each sub-bank within a bank is considered a shared resource. Similarly, a 128B operand collector write port is viewed as four 32B shared resources instead of being a single resource.

At this lower-level of resource arbitration, register file requests arbitrate to get access to the available shared resources following the same priority orders and arbitration scheme used in the matrix arbiter. Fig. 3.16b shows the request matrix in our coalescing-aware design with the same eight requests used in the baseline arbiter case earlier but with the additional dual-access banks and sub-bank masks information. In this example, every bank has a left and a right accesses that are *coalescable* since they target non-overlapping sub-banks within the bank (the ANDing of the two sub-bank masks is zero) and every operand collector has two targeting requests that are also *coalescable* across the serving banks (the ANDing of the two sub-bank masks across the banks is zero). Therefore, all of the eight requests can be granted access and proceed in the same cycle.

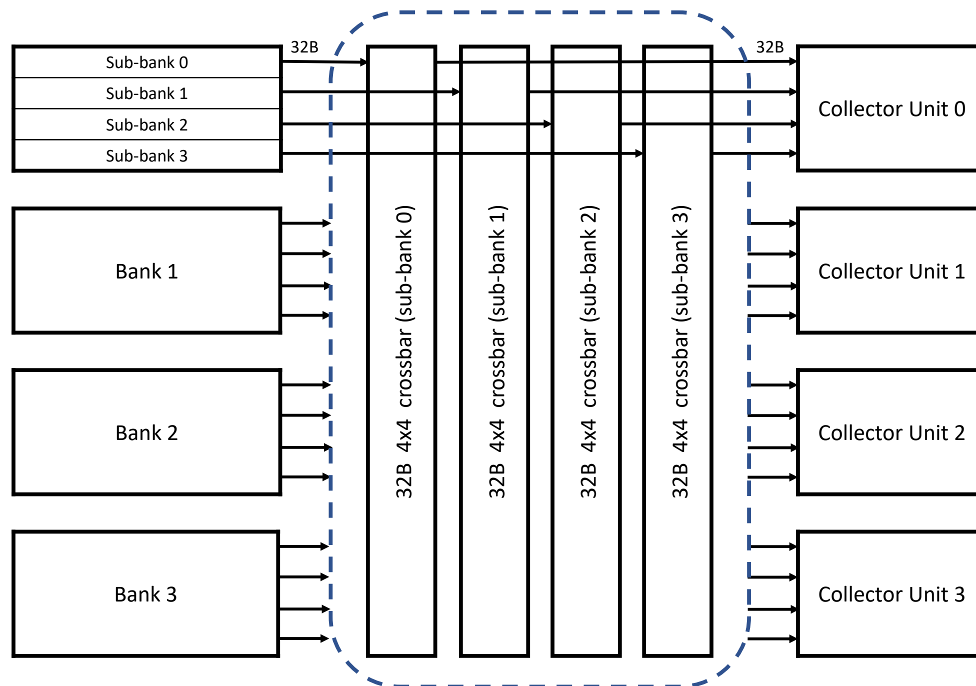
3.6.2.6 Register File Interconnect

When a warp instruction is issued by the warp scheduler, it uses an operand collector unit to read its source operands from the register file banks. A warp instruction may require up to three operands to read which may take multiple cycles due to bank conflicts and operand collector conflicts. The operand collector holding a warp instruction initiates the read requests to the targeted banks. These requests may target same or different banks. Our design provides coalescing capabilities between read requests from different warps targeting the same bank and also support coalescing read requests from the same warp instruction (same operand collector (OC)) targeting different banks.

To support such register coalescing cases, we updated the baseline register file crossbar



(a)



(b)

Figure 3.18: Register file interconnect: (a) Baseline 4×4 crossbar with 128B ports that connects register file banks to the operand collector units. (b) New crossbar structure used in our design, which supports register coalescing at no extra cost, with four 4×4 crossbars each of which has narrow 32B ports and connects a particular sub-bank (from all four banks) to the operand collector units.

that connects the banks to the operand collectors, shown in Fig. 3.18a, to have separate control for every 32B of the switched data. To do so, instead of operating on a 128B data, we split the crossbar to four crossbars each of which operates on a one-fourth of the data (32B) at no extra cost. With this configuration, there is a narrow crossbar corresponding to every sub-bank across the four banks in the register file with its own separate control as shown in Fig. 3.18b. The 32B 4×4 crossbar for sub-bank 0, takes the output of every sub-bank 0 across the four banks and interconnect them into the *lower* 32B of every operand collector unit.

The new crossbar structure supports coalescing read requests from different warps (different operand collectors) targeting the same register bank. To illustrate how this works, consider a read request from OC0 with a $mask = 0001$ and another request from OC1 with $mask = 1100$ that are coalesced at bank 0. The first request reads out from sub-bank 0 and the other request reads out from sub-bank 2 and sub-bank 3 at the same time. The read out data would need to be steered into the two operand collectors correspond to these requests. The output data from each sub-bank within the targeted bank maps directly to a 32B slice of an operand collector write bus with the output of sub-bank 0 maps to the lower 32B data slice and the output of sub-bank 1 maps to the next 32B slice, and so on. In this example, based on the mask values of the two coalesced requests, sub-bank 0 data is steered into OC0 on the lower 32B slice of its write bus and data from sub-bank 2 and sub-bank 3 is into OC1 on the upper 64B of its write bus.

Another important coalescing scenario that is supported with the new crossbar structure is allowing two read requests from the same warp instruction (same operand collector) to access different banks at the same time. For example, OC0 issues two read requests to bank 0 and bank 1 with mask values $mask_1 = 0011$ and $mask_2 = 1100$, respectively. Since these two requests do not conflict on the operand collector write port (ANDing of the two masks is zero), the register file arbiter may grant these two requests access in the same cycle. Read data from bank 0 will be steered into OC0 on the lower 64B of its write bus and read data

from bank 1 will be steered into OC0 on the upper 64B of the write bus and the packed data is written into the operand collector in a single cycle.

3.6.2.7 Operand Collector Write

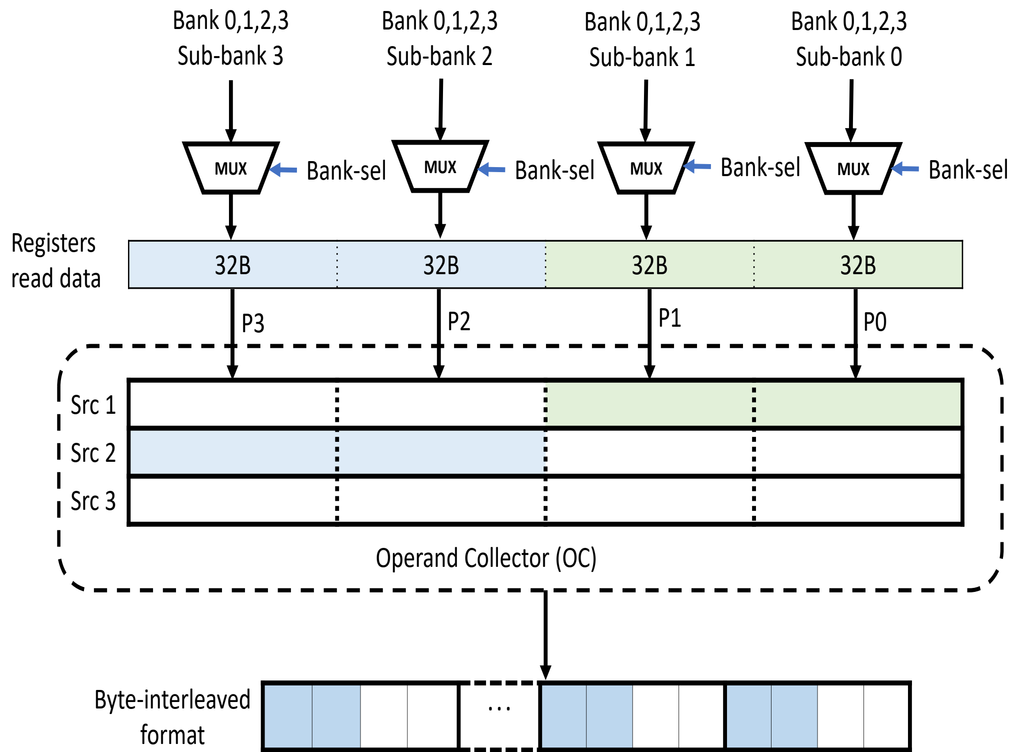


Figure 3.19: Coalescing-aware operand collector unit with 32B write ports. Coalesced read data is naturally unpacked into the destined source operands buffering space.

With separate per sub-bank crossbars, our design is capable of coalescing read requests from different banks targeting the same operand collector. Read data on the collector unit input port can be from a single warp register read or from two coalesced warp registers reads. We made slight modification to the operand collector unit to be able to capture a coalesced data and write it into the corresponding operand registers.

Instead of capturing the coalesced read data in a single register entry inside the operand collector and *unpack* it as it is read out from the collector unit which costs additional data

MUXing (not considering data alignment yet) as done in CORF design, our design unpack the coalesced data as it is written into the operand collector, in a low-cost way, by having separate write controls for every 32B of the operand collector data buffers. The unpacking approach we use falls naturally in our design as the *single* 128B write port on the collector unit behaves as four 32B write ports where each narrow port writes to a 32B slice of the buffer entries inside the operand collector.

Fig. 3.19 highlights the slight update made to the operand collector to handle data for coalesced reads. To illustrate with an example, consider OC0 with two warp registers, $R0$ and $R1$ to read from the register file. Requests for reading the two registers are sent to bank0 and bank1, respectively. The two registers have narrow-width data with $R0$ having a sub-bank mask of value 0001 and $R1$ having a mask of value 1100. The two register are read at the same time and their data is steered into OC0 with $R0$ data appears on the lower 32B and $R1$ data on the upper 64B of the collector unit input port. Considering that the operand collector now has four separately controlled 32B input ports, each 32B port writes its data into a 32B slice of the destined source operand. In this example, the lower 32B port, named as P_0, writes its data into $R0$ data buffer and ports P_2 and P_3 each write its data into the corresponding 32B slice of $R1$ buffer. With this approach, the coalesced read data is naturally *upacked* into its destined registers which would look the same as stored in the register file banks.

When an operand collector is issued into the execution pipeline, every source operand gets represented in byte-interleaved format, by wiring the bytes in different byte-positions, and passes through a byte-swap MUX for its thread-bytes to be re-aligned if necessary and sign-extended.

3.6.2.8 Register Width Detection

Fig. 3.20 shows the width detection logic at the end of the execution pipeline (in writeback stage) that generates a 4-bit sub-bank enables (mask) and captures the sign bit for the result. We detect positive narrow-width values, with most significant bytes being zeros, by ORing

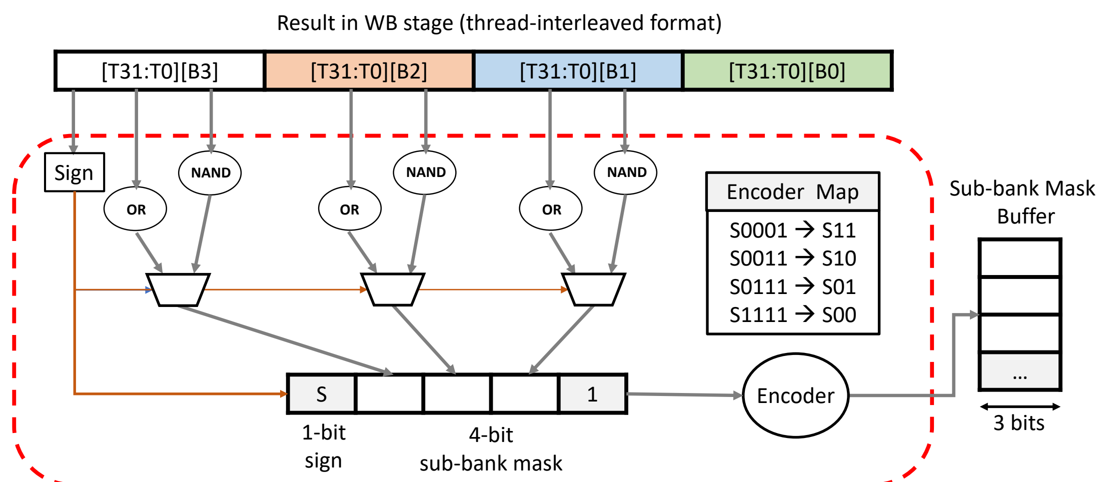


Figure 3.20: Width detection and sub-mask generation logic for a warp result in WB stage. The 4-bit sub-bank mask is saved in a buffer in a 2-bit encoded form along with the sign-bit.

all the bits with same byte-position together. This is applied on bytes 1, 2, and 3 and produces a 3-bit result for the 128B warp data. We also detect if the narrow-width data is negative, with most significant bytes being all ones, by using a reduction NAND gate on all bits with same bye-position, applied on byte 1, 2, and 3, which produces another 3-bit result. The sign bit (the MSB bit) for the warp data is then used to select between the positive and negative 3-bit results to produce the final mask bits that are used to enable/disable register file sub-banks with sub-bank 0 mask-bit is set to 1.

The generated sub-bank mask is saved in a buffer (in an 2-bit encoded form) for the destination warp register being written. We used a simple encoding that maps the 4-bit sub-bank mask into a 2-bit encoded value using the mapping listed in Fig. 3.20. Consecutive reads from the warp register use the saved mask (in a 4-bit decoded form) to access the sub-banks that holds the register data. As we mentioned earlier, the 4-bit sub-bank mask for each register file request is used for arbitrating on fine-grained shared resources (sub-banks access and operand collector 32B ports) and enables register coalescing for read and write requests

targeting the same bank or read requests targeting the same operand collector. Along with the sub-bank mask, we also captured the sign bit of the warp result in the buffer. This bit is then used to recover the full register width of every source operand when a warp instruction is issued into the execution pipeline out of an operand collector unit. Basically, the sign bit would be written to all bits that were striped out of the narrow-width warp register when it was written into the register file.

Our mask generation approach enables the use of fewer register file sub-banks to capture the narrow-width result as compared to the CORF design. In CORF, a 4-bit mask, in its decode form, is captured in a buffer without capturing the sign-bit of the register value. This requires the sign-bit to be present in the narrow-width data, written to the register file, in its MSB bit which effectively reduces the range of the narrow-width value by a factor of two or, equivalently, requires more bytes to be written into the register file to capture the register sign bit. For example, a narrow width value with the two most significant bytes, byte 2 and byte 3, are zeros would require only two sub-banks to capture the register value in our design. Whereas, in CORF design, it may require three of the sub-banks to hold the register value in the case that the MSB of byte 1, which is used as the sign-bit in this case, is found to be different than the leading bits in byte 2 and 3. With a simple encoding scheme, our design is able to capture the sub-bank mask and also the destination register sign-bit in a compact form that only requires 3 bits.

3.6.2.9 Design Overhead

Table 3.4 summaries the cost of the design overhead involved in CORF compared to our low-cost design. Our design avoided, with no loss of coalescing benefits, most of the complexity and cost in CORF design including the compile-time register alignment hints, register renaming and allocation logic, and the costly data shifters used to align warp registers. Our design adopted a low-cost fixed register alignment scheme and requires a small buffer to capture the sign-bit and the sub-bank mask for every warp register in an encoded form. It also requires cheaper intra-thread MUXes to perform warp data alignment. Both designs require

Design Overhead	CORF	Our Design
Rename table	3.7KB	0
Free register map	512B	0
Allocation table (Mask buffer)	1.47KB	384B
Code size increase	1.3% average	0
Data alignment	10 byte-level shifters	5 intra-thread MUXes

Table 3.4: Design overhead cost for our coalescing-aware design compared with CORF design.

a tiny logic in WB stage to detect the size of destination registers, support dual access into register file banks, and perform sign extension when recovering narrow-width registers into their full width.

3.7 Evaluation

3.7.1 Methodology

To evaluate our proposed coalescing-aware register file design, we used GPGPU-Sim version 3.2 simulator [21] and measured dynamic power consumption using GPU-Wattch tool [7]. We used a GPGPU model similar to Nvidia Fermi [1] with configuration parameters listed in table 3.5 and used the Rodinia benchmark suite [20] which consists of general purpose benchmarks covering a wide range of scientific domains that target GPU platforms. We used the standard input set that ships with the benchmark suite. Table 3.6 lists the applications and their applied domains for the Rodinia suite.

Register coalescing opportunities that exist in general-purpose applications are independent of the GPU architecture being use. We conducted simulation experiments and collected data measurements to show the benefits of our proposed coalescing-aware register file design using the following metrics:

- Register file access reduction: we measured the percentage of access reduction we attain from coalescing read and write requests on register file banks and also the reduction percentage we attain from coalescing operand collector writes.

Parameter Name	Value
Number of SMs	16
Number of Warps	48 per SM
SIMD lanes width	32
Register file size	128KB per SM
Register file banks	4 per SM
Register file width	128B
Operand collectors (OCs)	4 per SM
Warp scheduler	2-level, 2 per SM
Streaming Processing Units (SPU)	32 per SM
Special Functional Units (SFU)	4 per SM
Load/Store Units (LDST)	16 per SM
L1 cache size	16KB per SM
Shared memory size	48KB per SM
L2 cache size	768KB
Thread blocks (CTAs)	8 per SM
Load/Store address width	64-bit
Added sub-bank mask buffer	384B per SM

Table 3.5: Configuration parameters for our GPU design with register coalescing support.

Short-name	Application	Domain
backprop	Back Propagation	Pattern Recognition
bfs	Breadth-First Search	Graph Algorithms
b+tree	B+ Tree	Search
cfd	CFD Solver	Fluid Dynamics
dwt2d	GPU DWT	Image/Video Compression
gaussian	Gaussian Elimination	Linear Algebra
heartwall	Heart Wall	Medical Imaging
hotspot	Hot Spot	Physics Simulation
kmeans	Kmeans	Data Mining
lud	LU Decomposition	Linear Algebra
nw	Needleman-Wunsch	Bioinformatics
pathfinder	Path Finder	Grid Traversal
srad	SRAD	Image Processing

Table 3.6: GPGPU application benchmarks from Rodinia general-purpose suite.

- Register file Bandwidth increase: we measured the increase in register file access bandwidth as well as operand collector write bandwidth resulted from supporting register coalescing.

- Register file coalesced requests: we present a breakdown of register file coalesced requests by their type and show their percentage to the overall coalesced requests.

- IPC performance speedup: we measured the overall performance gain we achieved with our coalescing-aware register file design. We measured Instruction Per Cycle (IPC) speedup compared to the baseline design.

- Dynamic energy reduction: we measured the percentage reduction of the overall GPU dynamic energy that we were able to achieve with our register coalescing design compared to the baseline GPU.

We have conducted multiple sets of experiments with different micro-architectural settings to show their impact on register coalescing benefits. We used the following design settings:

- We ran a set of experiments using a register file layout with all warp registers are mapped into the same bank as show in Fig. 3.10a. We refer to this layout as *wid_layout*. In this experiments set, we have simulated our design with the following register alignment schemes:

  1. Fixed alignment (*reg_alignment*): this is primary setting in our design where warp registers have fixed alignment based on their numbers with even registers are right aligned and odd registers are left aligned. All other alignment schemes are simulated to show how they compare with this design setting.

  2. Write interleaved (*wr_alignment*): in this setting, register alignment switches between right and left alignment on every write made by the warp with the goal

of distributing registers for each warp into left and right aligned inside the register file banks.

3. Ideal alignment (*ideal_ alignment*): we mimic ideal register alignment by considering that register file requests do not conflict due to their left/right alignment but still considered that their combined size should be no more than a full register width. This setting would show coalescing limitations due to requests size only.

4. Upper bound (*upper_ bound*): to show the micro-architectural limit of our design, we simulated the design assuming register file requests are all coalescable regardless of their alignment within the banks and the size of their requests.

- We ran another set of experiments to evaluate and compare results with the other register file layout, shown in Fig. 3.10b, where warp registers are distributed across the banks. This layout, which we refer to as *wshift_ layout* is known to reduce the number of bank conflicts and provides a performance improvement over *wid_ layout*. We collected experimental results for the metrics we mentioned earlier on both register layouts to show how they compare in our coalescing-aware register file organization.

3.7.2 Register File Access Reduction

Our coalescing-aware register file design reduces the number of accesses made into register file banks as well as the number of writes on the operand collector units by combining narrow-width registers accesses into fewer physical accesses. Fig. 3.21 and Fig. 3.22 show the percentage of register file banks access reduction and the percentage reduction of operand collectors writes compared the baseline design, respectively. These experiments were conducted with a register file layout (registers to banks mapping) as shown in Fig. 3.10a which we refer to as *wid_ layout*. The figures show access reduction using the fixed register alignment scheme we used in our design (*reg_ alignment*) compared to an interchanged dynamic alignment (*wr_ alignement*) and an *unrealistic* ideal alignment (*ideal_ alignment*) schemes.

With the low-cost fixed alignment scheme, our design reduced register file banks accesses

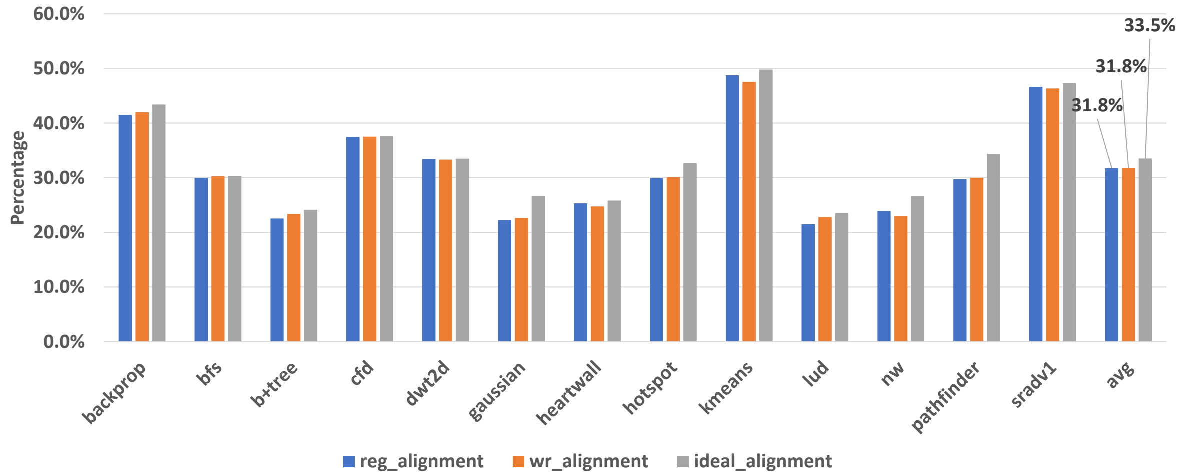


Figure 3.21: Percentage of register file access reduction with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register number (*reg_alignment*), write interleaved alignment (*wr_alignement*), and an ideal alignment (*ideal_alignment*).

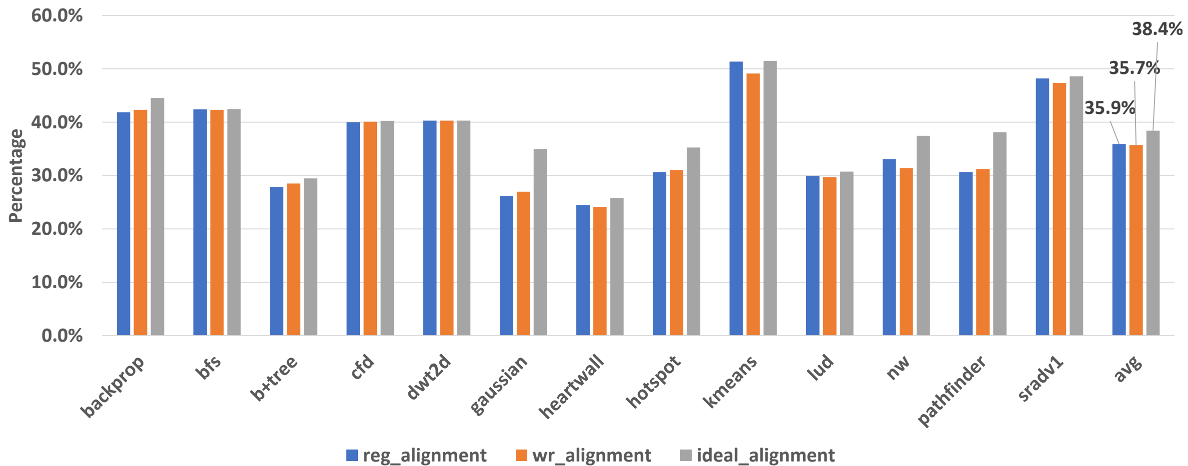


Figure 3.22: Percentage of operand collectors write reduction with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register entry number (*reg_alignment*), register interleaved alignment on writes (*wr_alignement*), and an ideal alignment (*ideal_alignment*).

by 31.8% on average which is about the same reduction achieved using *wr_alignement* and within 2% of what an ideal alignment may have achieved. On the operand collectors, our

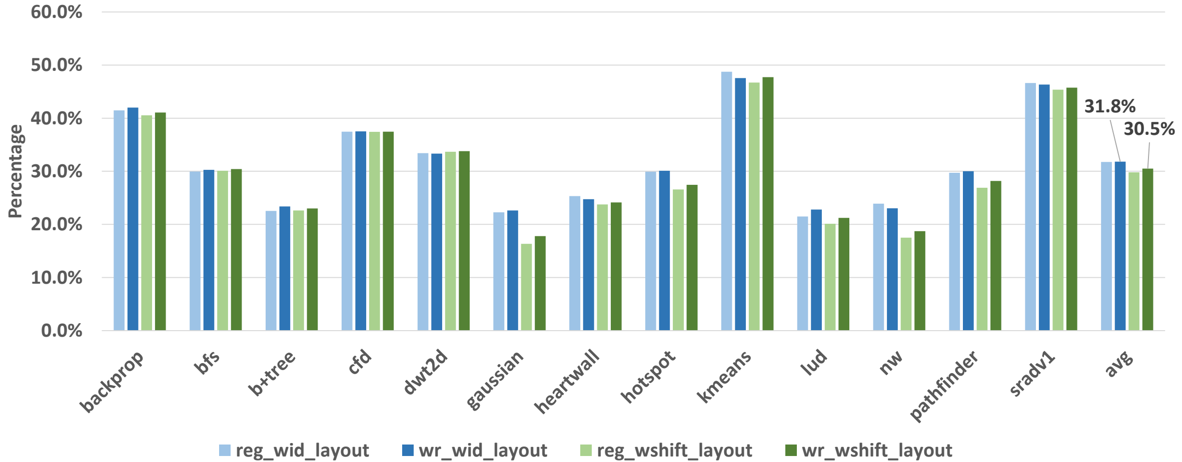


Figure 3.23: Percentage of register file access reduction comparison between two different register file layouts: *wid_layout* and *wshift_layout* using two register alignment schemes: fixed alignment based on register number (*reg_ **) and write interleaved alignment (*wr_ **).

design was able to reduced write accesses by 35.9% on average compared to 35.7% achieved using (*wr_alignement*) and it is within 3% from an ideal register alignment.

Our design is not limited on using a specific register file layout for register coalescing to be supported. In fact, our design can support any register file layout as it supports register coalescing within a register file bank and also across two different banks. In Fig. 3.23 we compare the percentage reduction achieved on register file bank accesses for the two layouts shown Fig. 3.10, *wid_layout* and *wshift_layout*. The figure shows that the two layouts are within 1.5% of each other with the access reduction achieved with *wshift_layout* is about 30.5% on average.

3.7.3 Register File Bandwidth Increase

As our design is capable of combining narrow-width access requests to fewer physical requests, register file and operand collectors bandwidth is consequently improved. Fig. 3.24 and Fig. 3.25 show the percentage of bandwidth improvement on register file banks and operand collectors compared to the baseline, respectively.

With fixed register alignment, our design is able increase the register file bandwidth by

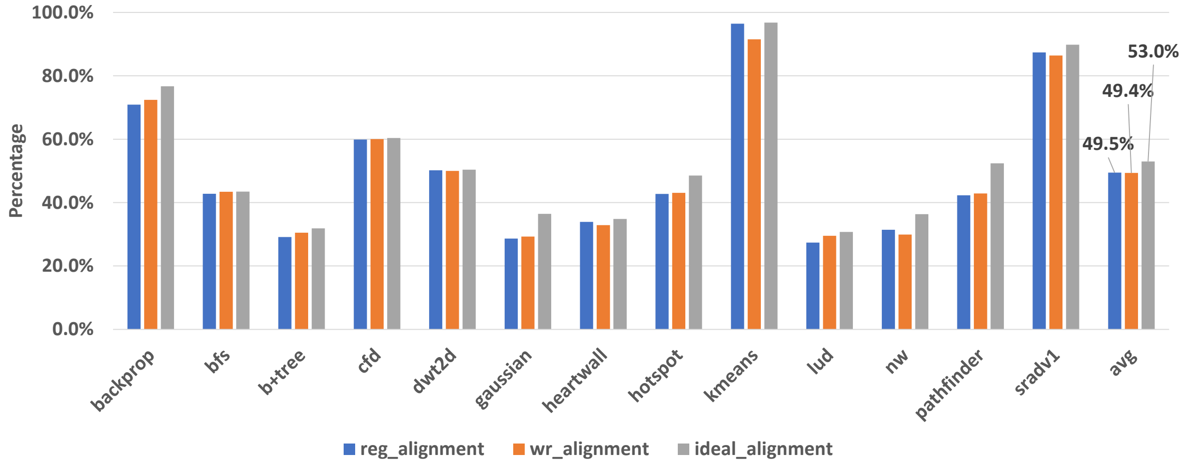


Figure 3.24: Percentage of register file bandwidth increase with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register number (*reg_alignment*), write interleaved alignment (*wr_alignement*), and an ideal alignment (*ideal_alignment*).

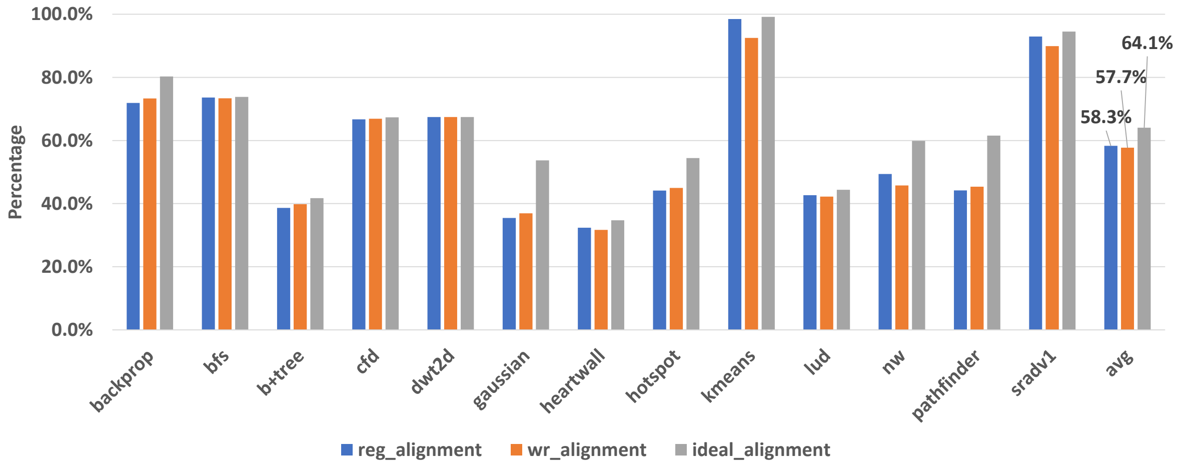


Figure 3.25: Percentage of operand collectors bandwidth increase with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register entry number (*reg_alignment*), register interleaved alignment on writes (*wr_alignement*), and an ideal alignment (*ideal_alignment*).

49.5% and the operand collector bandwidth by 58.3% on average. The figures show that the percentage of bandwidth increase we are able to achieve with low-cost design falls within 4%

and 6% of an ideal register alignment for register file and operand collectors, respectively.

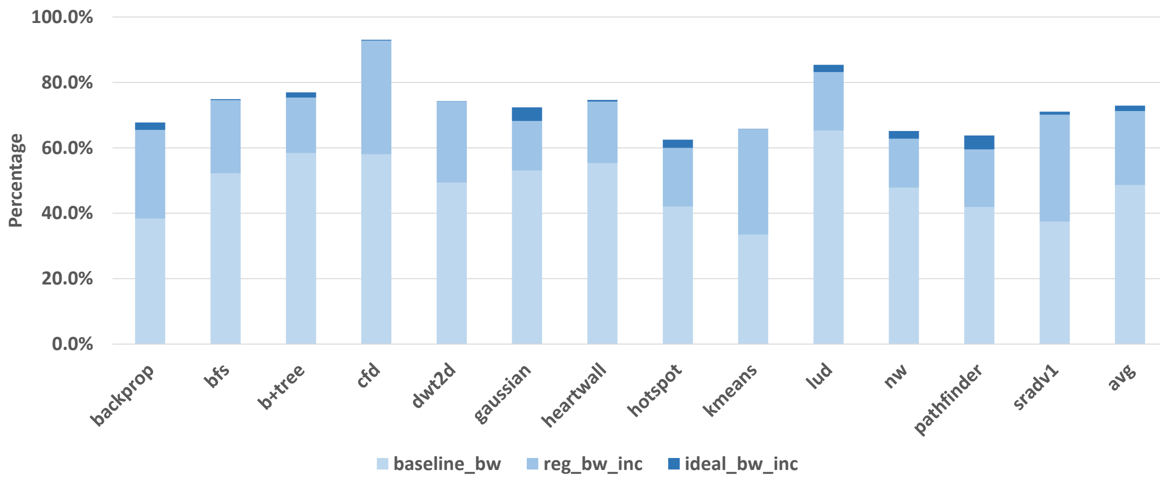


Figure 3.26: Percentage of used register file bandwidth showing the bandwidth increase from register coalescing using a fixed register alignment (*reg_bw_inc*) and an ideal register alignment (*ideal_bw_inc*) over the baseline bandwidth (*baseline_bw*).

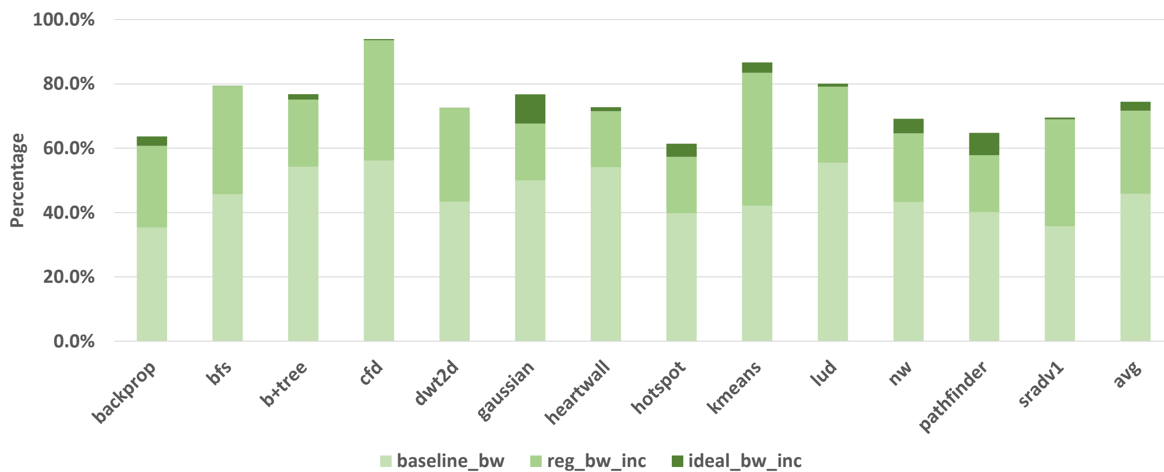


Figure 3.27: Percentage of used operand collector bandwidth showing the bandwidth increase from register coalescing using a fixed register alignment (*reg_bw_inc*) and an ideal register alignment (*ideal_bw_inc*) over the baseline bandwidth (*baseline_bw*).

We provide another view of the bandwidth improvement our design is able to achieve in Fig. 3.26 and Fig. 3.27. The figures show the percentage of *used* bandwidth for register file banks and operand collectors in the baseline design (*baseline_ bw*), the average incremental bandwidth our design is able to achieve (*reg_ bw_ inc*), and the further improvement we may achieve with near-ideal register alignment mechanism (*ideal_ bw_ inc*) which is within 4% to 6% as we just mentioned.

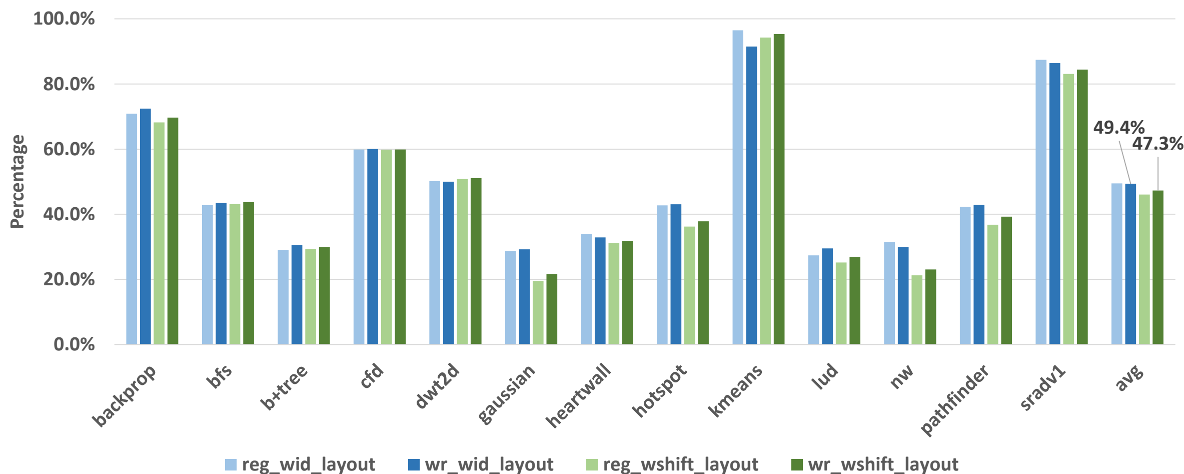


Figure 3.28: Percentage of register file bandwidth increase comparison between two different register file layouts: *wid_ layout* and *wshift_ layout* using two register alignment schemes: fixed alignment based on register number (*reg_ **) and write interleaved alignment (*wr_ **).

Fig. 3.28 compares the bandwidth improvement achieved by our coalescing-aware design for the two register file layouts: *wid_ layout* and *wshift_ layout*. The two layouts are within 2% with bandwidth improvement achieved using *wshift_ layout* averaging about 47.3% compared to the baseline.

3.7.4 Register File Coalesced Access

Our design supports register coalescing for read requests from same or different warp instructions, write requests, and also read and write requests contending on register file

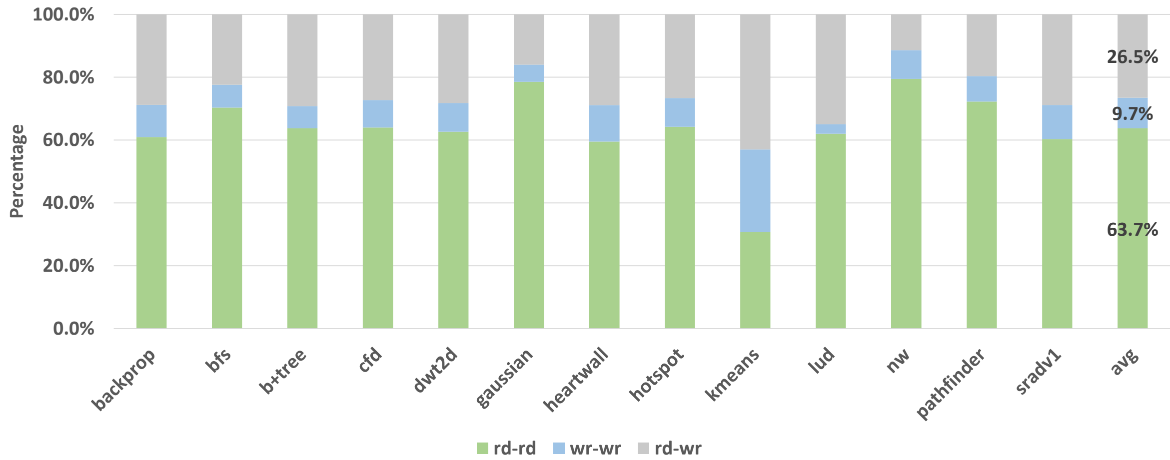


Figure 3.29: Breakdown of register file coalesced accesses with a *wid_layout* register file: Read-Read coalesced accesses (*rd-rd*), Write-Write coalesced accesses (*wr-wr*), and Read-Write coalesced accesses (*rd-wr*).

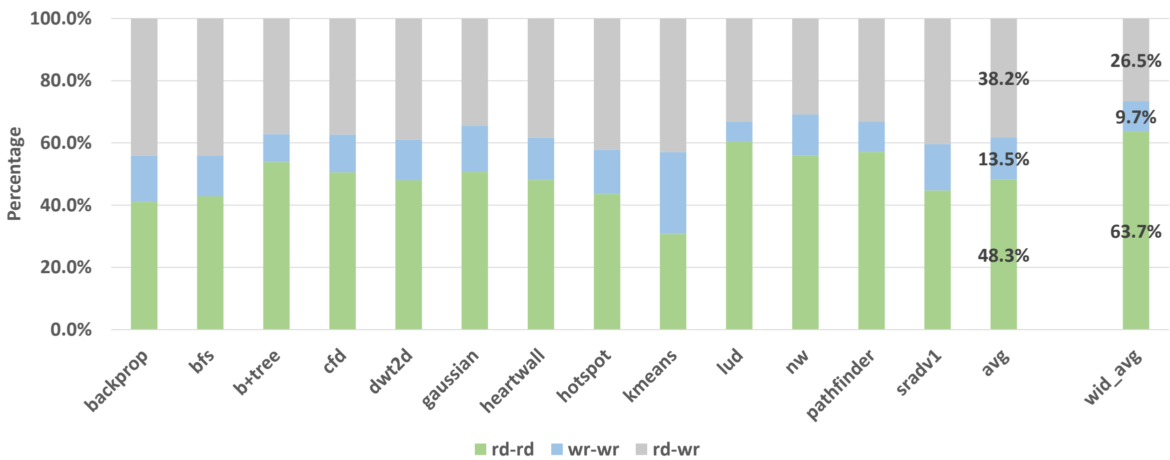


Figure 3.30: Breakdown of register file coalesced accesses with a *wshift_layout* register file: Read-Read coalesced accesses (*rd-rd*), Write-Write coalesced accesses (*wr-wr*), and Read-Write coalesced accesses (*rd-wr*).

banks. Fig. 3.29 shows the breakdown of register coalesced requests served by register file banks. The majority of coalesced accesses are of read type which represents 63.7% of all coalesced accesses. Coalesced writes represents 9.7% and the mixed reads and writes coa-

lescing represents 26.5%. In Fig. 3.30, the same breakdown of coalesced requests is shown for a *wshift_layout* register file. The figure shows, with this layout, more write requests are getting coalesced compared to *wid_layout*. The percentage of write-write coalesced requests is increased to 13.5% and the percentage of read-write coalesced requests significantly increased to 38.2%. As we will show, reducing the number of write requests has higher effect on performance than reducing read requests since it allows for data-dependence requests to be ready and issue faster into the execution pipeline than before.

3.7.5 IPC Performance Speedup

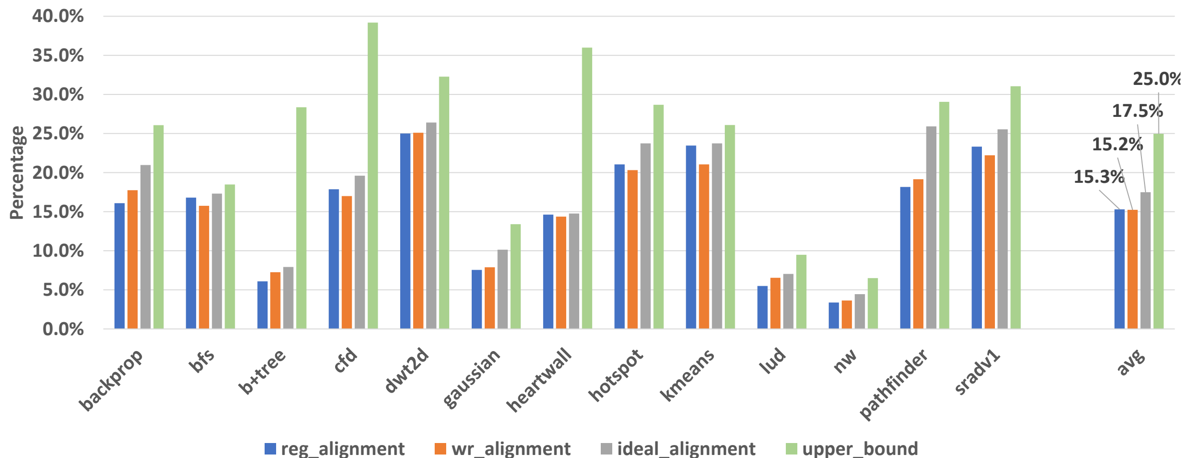


Figure 3.31: Percentage of overall IPC speedup in GPU with a *wid_layout* register file and using different register alignment schemes: fixed alignment based on register number (*reg_alignment*), write interleaved alignment (*wr_alignement*), and an ideal alignment (*ideal_alignment*) compared to the upper bound (*upper_bound*).

One of the primary objectives of our new register file design is to achieve IPC performance speedup for general purpose applications on the GPU. Our design enables register coalescing that reduces the number of register file read and write accesses, reduces register file pressure and port conflicts, and improve access bandwidth which all lead to an overall IPC performance enhancement.

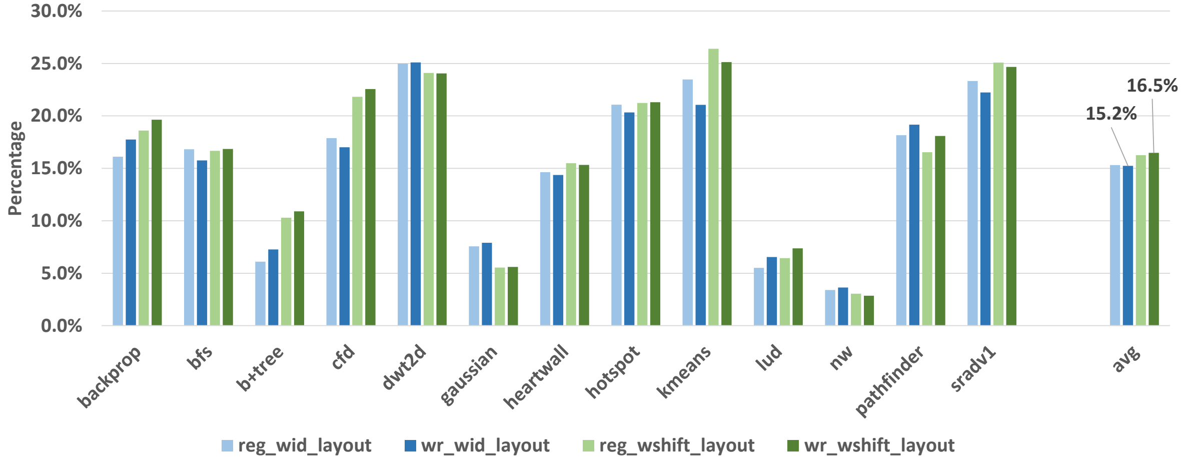


Figure 3.32: Percentage of overall IPC speedup comparison between two different register file layouts: *wid_layout* and *wshift_layout* using two register alignment schemes: fixed alignment based on register number (*reg_ **) and write interleaved alignment (*wr_ **).

Fig. 3.31 shows that the IPC speedup our coalescing-aware register file design is able to achieved using a *wid_layout* and a fixed register alignment (*reg_alignment*) is about 15.3% compared to the baseline. The low-cost alignment scheme used in our design gives a decent speedup compared to an ideal alignment which gives only 2.2% additional speedup. The *upper_bound* value of 25% IPC speedup represents the micro-architectural limit our design can reach given that register file requests are all coalescable and no coalescing opportunity is lost due to registers size or alignment.

Fig. 3.32 shows the IPC seedup our design is able to achieve with a *wshift_layout* compared to *wshift_layout* register file. The figure shows that our design with a *wshift_layout* register file outpaces the other layout and achieves the highest IPC speedup of 16.5% on average. The maximum speedup seen is about 27% on `KMEANS` and the minimum is about 4% on `NW`.

3.7.6 Dynamic Energy Reduction

Improving energy efficiency in the GPU is one of the primary objectives of our new coalescing-aware register file design. With the significant IPC speedups achieved, general-

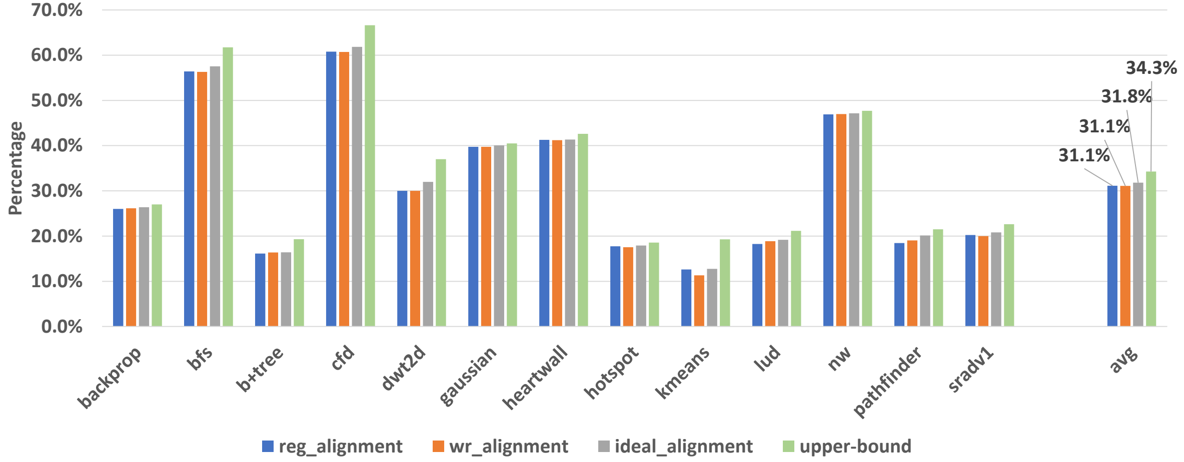


Figure 3.33: Percentage of overall dynamic energy reduction in GPU with a *wid_ layout* register file and using different register alignment schemes: fixed alignment based on register entry number (*reg_ alignment*), register interleaved alignment on writes (*wr_ alignement*), and an ideal alignment (*ideal_ alignment*) compared to the upper bound (*upper_ bound*).

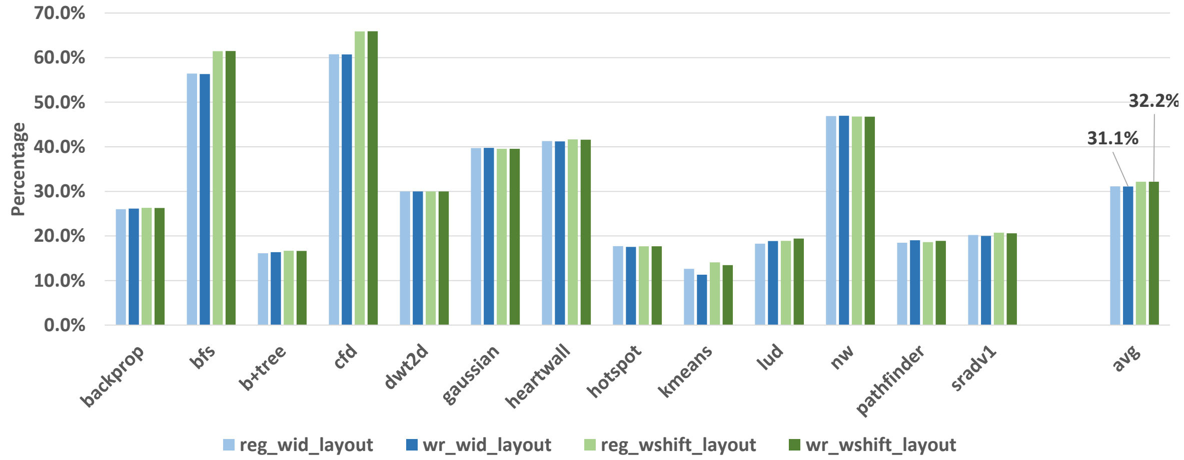


Figure 3.34: Percentage of overall dynamic energy reduction comparison between two different register file layouts: *wid_ layout* and *wshift_ layout* using two register alignment schemes: fixed alignment based on register number (*reg_ **) and write interleaved alignment (*wr_ **).

purpose applications have faster runtime making the GPU more energy efficient. Fig. 3.33 shows the percentage reduction of overall dynamic energy in the GPU using our coalescing-aware register file. With a *wid_ layout* register file and a fixed register alignment scheme

based on register entry number (*reg_ alignment*), our design is able to reduce the GPU dynamic energy by 31.1% on average. The achieved energy reduction with the low-cost register alignment scheme is roughly the same compared to *wr_ alignement* and the *ideal_ alignment* results with an upper limit of 34.3% of energy reduction on average.

Fig. 3.34 shows that our design provides more energy efficiency when the *wshift_ layout* register file is used. The figure shows an average of 1.1% additional energy reduction is achieved over the *wid_ layout* register file which brings the highest reduction achieved to 32.2% on average. The same energy reduction is achieved using the two register alignment schemes: *reg_ alignment* and *wr_ alignement*.

3.7.7 Result Summary

Metric	**CORF**	**CORF++**	**Our Design *Layout-1***	**Our Design *Layout-2***
IPC speedup	4%	9%	15.3%	16.5%
Dynamic energy reduction	8.5%	17%	31.1%	32.2%
RF access reduction	10%	23%	31.8%	30.5%

Table 3.7: Summary of results achieved by our design compared with CORF design.

Table 3.7 summaries the IPC speedups and energy efficiency our design is able to achieve compared to the high cost CORF design. These results along with the overhead comparisons listed in Table 3.4 clearly show that our coalescing-aware design is far superior to CORF design in all aspects including design cost and complexity, register file access reduction, IPC performance speedups, and dynamic energy efficiency.

3.8 Conclusion

In this work, we introduced a new coalescing-aware register file design to improve IPC performance and energy efficiency in GPGPU. Our design supports register coalescing for

narrow-width read and write requests targeting different registers within a register file bank. It also supports coalescing read requests from the same warp instruction accessing different banks. Compared to CORF design, our design provides higher register coalescing capabilities at a much lower cost and complexity. On general-purpose benchmarks from Rodinia suite, our design reduced register file and operand collector accesses on average by 31.8% and 35.9%, respectively. It improved register file and operand collector bandwidth on average by 49.5% and 58.3%, respectively. And provided a 16.5% IPC performance speedup and a 32.2% dynamic energy reduction on average.

4. EXPLOITING ZERO DATA TO REDUCE REGISTER FILE AND EXECUTION UNIT DYNAMIC POWER CONSUMPTION IN GPU

4.1 Introduction

As we mentioned in Chapter 1, earlier power analysis showed that the register file and the execution units are the most power consuming components in GPUs. In this chapter, we focus on reducing dynamic power consumption for these two components without impacting GPGPU applications performance by proposing gating techniques that take advantage of under-utilized warps and the high percentage of zero data that exist in general-purpose applications running on the GPU.

4.2 Motivation

In this section, we present results from compute-intensive (GPGPU) benchmarks in order to show potential dynamic power savings in register file and execution units in GPGPUs. For these experiments we examine GPGPU benchmarks from the Rodinia benchmark suite [20] and obtained the experimental results by running those benchmarks on the GPGPU-sim v3.2 simulator [21].

Inactive threads: In GPUs a warp is considered the smallest amount of work which may be dispatched into the execution pipeline in a given cycle. Warps consists of 32 threads executing in a SIMD pipeline in a lock-step fashion. A warp could either be fully utilized by having all 32 threads executing the same instruction or it may have some inactive threads that do not execute or produce results.

One reason to have warps with inactive threads is the presence of divergent flows in general-purpose workloads. A branch instruction may cause some of the threads in a warp to diverge into the taken path and while the rest of the threads execute the non-taken path. GPUs use predicated execution of threads in a warp on both taken and non-taken paths. Another cause of inactive threads is that general-purpose workloads may always evenly split

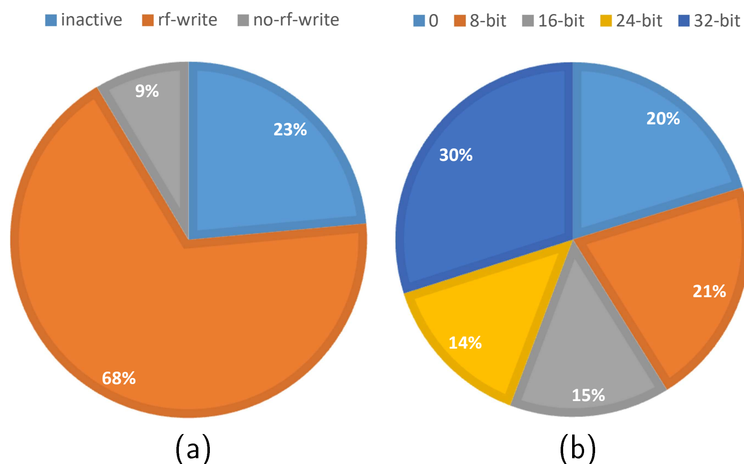


Figure 4.1: GPGPU application warp thread statistics: (a) Percentage of warp threads in an inactive state, active and writing to register file, and active but not writing to register file. (b) Percentage of warp thread results with zero and non-zero values that can be represented with 8, 16, 24, and 32 bits.

into a multiple of 32 threads to fully utilize the SIMD execution pipeline. This limitation would result of having under-utilized warps being dispatched during program execution.

Fig. 4.1a shows statistical measurements on threads during program execution averaged across all benchmarks we used. We found that the percentage of inactive threads in warps that were dispatched for execution is about 23% on average. Note that, each warp still accesses the register file to read operands for all 32 threads and also enables all 32 lanes for threads execution regardless of having some threads in an inactive state.

In-lane zero data: The register file is one of the largest components of the GPU and consumes more than 15% of the total dynamic power. We examined the data values produced in our experimental simulation runs to look for power savings opportunities in the register file. We specifically measured how many times warp threads produced 32-bit results with a zero value. The zero values can be the result of executing arithmetic instructions or performing memory load instructions.

Fig. 4.1b shows that warp threads executed using one or more operands of 32-bit zero data 20% of the time on average across the benchmarks used. This high percentage of zero

data is still written and read from the register file and therefore adds to the register file dynamic power. This high amount of zero data represents an opportunity for energy savings as a 32-bit zero can be represented by a single meta data bit, allowing register file accesses to read and write those 32-bit zero values to be avoided.

The presence of zero data in a program gives another opportunity to save dynamic power in execution units. When a thread executes an arithmetic instruction with one of its operands having a value of zero, simple arithmetic operations such as multiply and add become trivial. By detecting such trivial operations dynamically during program execution, instruction execution for threads having zero operands can be avoided to further save dynamic power.

Low dynamic range: In our experimental study, we also measured the dynamic range of data values produced in each benchmark. The dynamic range gives an indication of how many bits can be used to correctly represent each data value without any loss of information. Fig. 4.1b shows that only 30% of inlane data values produced by warp threads require the full 32-bit range to be represented. The rest of non-zero data values can be effectively represented by fewer bits with upper bits being zero. We used the term low dynamic range in this context to refer to those data values that have zeros in their upper or most significant bits[1].

In our study of the data values often used in GPGPU applications, we found that often when a given execution lane's operands have a low dynamic range, this low range extends to the operands in other, adjacent lanes. The presence of such low dynamic range data values raises an opportunity to further reduce register file access power. As the data result in each 4-byte lane has one or more of its upper bytes as zeros, data from adjacent lanes can be arranged in a way that the zero bytes from each lane are grouped together to form groups of 4-byte zero values.

As we will show, those 4-byte zero values that present after data arrangement can be annotated via a couple bits of metadata, eliding the need to write into the register file,

[1]Note, there may be further gains to be had for low dynamic range data where the lower bits are also not fully utilized, we leave this for future work.

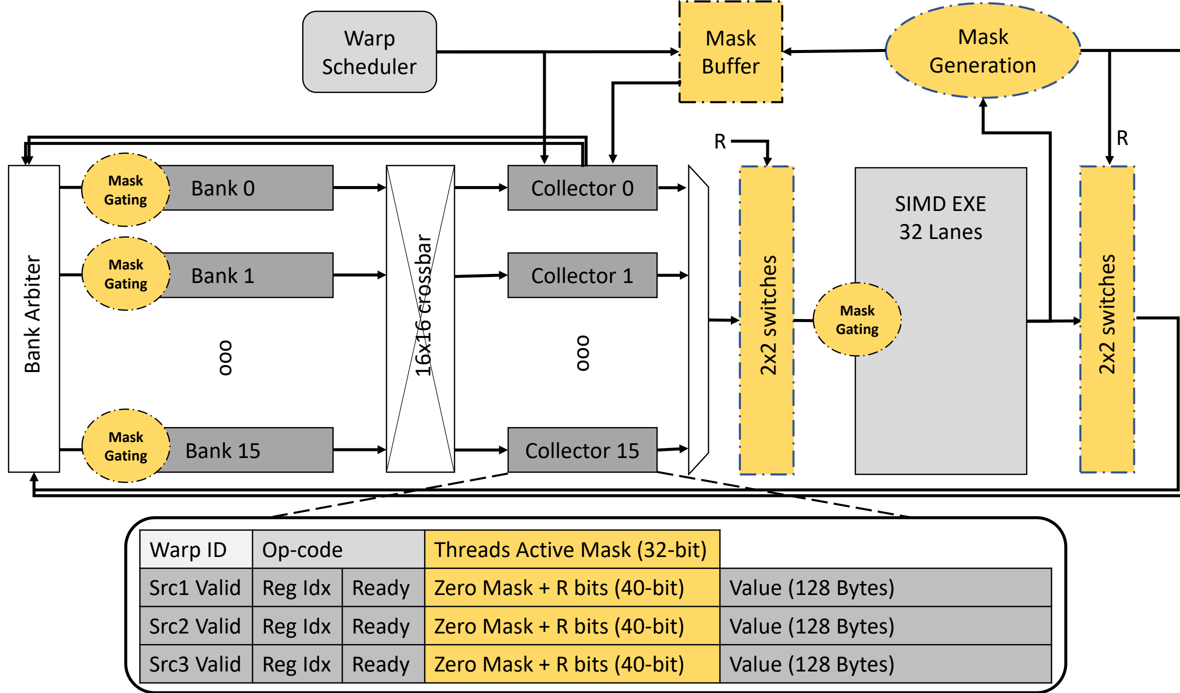


Figure 4.2: GPGPU main register file and execution pipeline with added components for power reduction highlighted.

similar to our approach for the in-lane zero data. Care must be taken, however, as must be reverted back into the original form when it is read by younger instructions.

4.3 Reducing Register File Dynamic Power

As mentioned in previous sections, the register file and its associated logic consume more than 15% of the total dynamic power. The baseline register file, shown in Fig. 4.2, is divided into 16 banks and has arbiter logic to steer read and write requests into the appropriate banks and to resolve bank conflicts among those requests. Each operand collector that holds an issued warp instruction can have up to three operands to read from the register file. The read requests are sent to the arbiter and each of which selects one bank to access. The arbiter prioritizes the read requests such that each bank can serve one read request at a time and each operand collector can capture one operand data at a time. The read data from the 16 banks are sent to the operand collectors using a 16×16 cross bar connecting network.

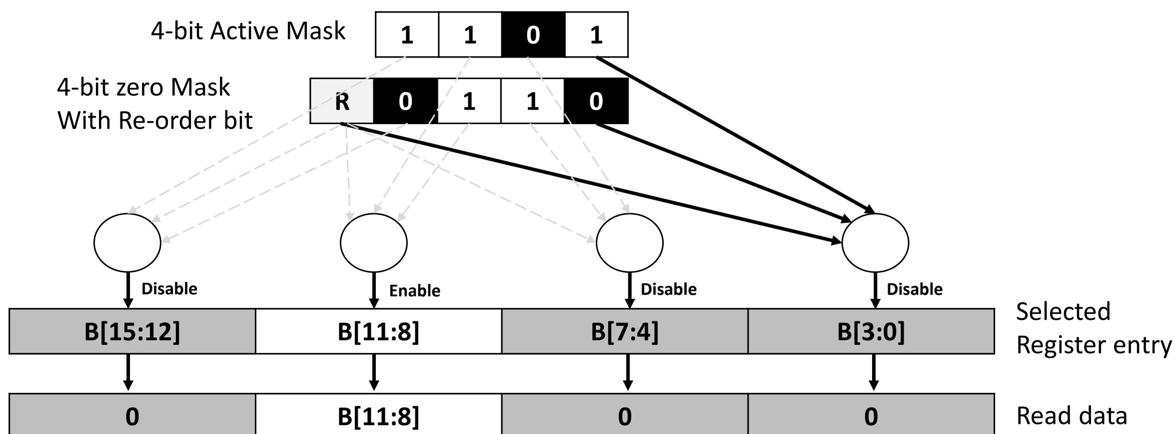


Figure 4.3: Usign the thread active mask and operand zero mask to gate off register file read access to the first four threads in a warp register.

The arbiter is also used to steer write requests from the execution units into the appropriate banks and resolve any conflicts that may occur among those requests.

In the remainder of this section we present the following mechanisms to reduce register file dynamic power without incurring any performance loss in GPUs for GPGPU applications.

4.3.1 Using the Thread Active Mask

When a new warp is issued by the warp scheduler, it is assigned an operand collector unit to collect all valid operands from the register file that the warp instruction requires before it starts to execute on those operands. Depending on the instruction type, the warp instruction may request to read up to three operands from the register file and each operand is 32×4-byte long to feed the 32 threads in the warp where each thread gets a 4-byte slice of the operand data. In general purpose compute applications, control divergence causes warps to be under-utilized with some of their threads in an inactive state. The inactive threads are considered do-not care as their results are not captured in the register file, however, they are still consuming dynamic power when accessing the register file to read their source operands.

To reduce register file dynamic power, we used the built-in thread active mask associated

with every warp to control its access to the register file. The active mask has one bit per thread to indicate whether the thread is active, and therefore, needs to execute or it is turned off due to control divergence and would be discarded. The thread active mask is captured in the operand collector along with other decoded information about the warp instruction such as operation type and valid operands to read from the register file as seen in Fig. 4.2.

On every read access to the register file, the thread active mask is used to gate off accesses made by inactive threads. In our register file organization, the 128-byte warp register consists of 32 thread registers and each thread register is controlled separately using one bit from the thread active mask associated with that thread. A thread register read is performed when the active bit for a given thread is non-zero. Otherwise, the output bus will have a zero value by default. Fig. 4.3 shows how the active mask is used on threads 0 to 3 to control their read access. All thread registers within a warp register represents a single entry in one of the register file banks and they are all accessed at the same clock cycle. The read output is then routed into the requesting operand collector through a cross bar switch and the valid operand is marked ready in the operand collector.

4.3.2 Using the In-lane Zero Mask

As mentioned in previous sections, there is a high percentage of zero data produced by warp threads in the general purpose computer applications we studied. In our baseline model, every thread in a committing warp writes its result back into the main register file in the write-back stage without any consideration of whether the value being written is zero or not. The 32-bit zero values produced during program execution add to the register file dynamic power on every write or read made by warp threads.

We took advantage of the inlane zero values found in general purpose applications to reduce register file dynamic power by capturing the zeros produced by warp threads in a separate buffer structure and avoid writing them into the register file. The 32-bit zero value can be represented via a single bit of metadata without any loss of information, and therefore, the buffer to capture those zero values would be small in size. For a committing warp with

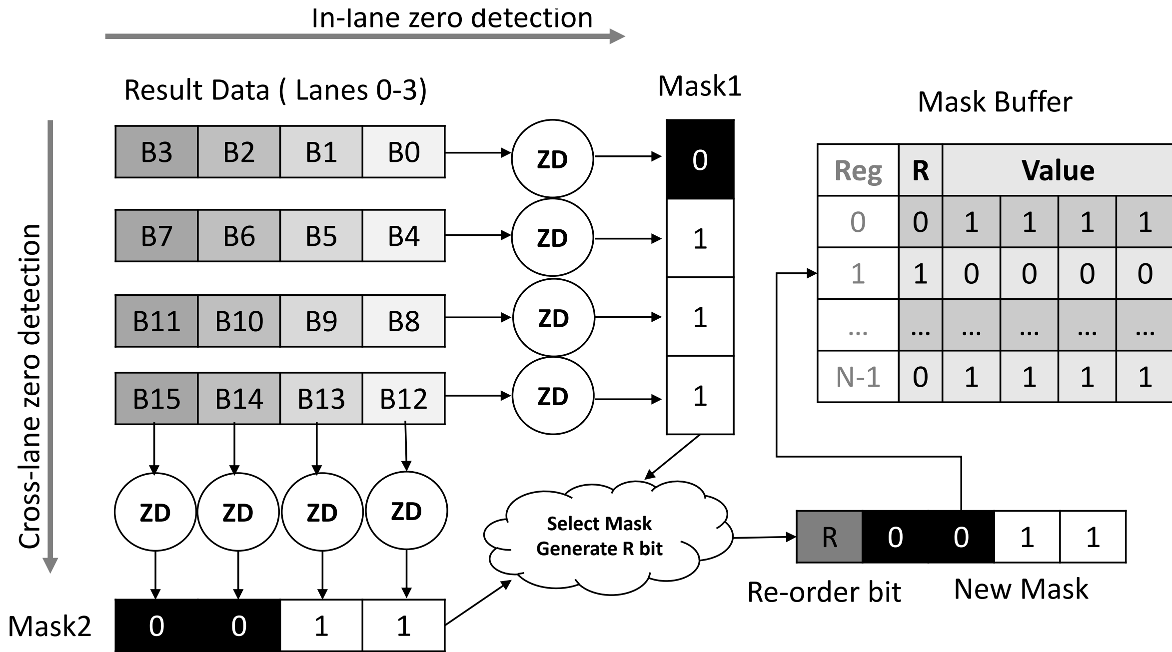


Figure 4.4: Generating the zero mask from data result produced by four execution lanes.

32 threads, each thread will have a single bit to indicate whether the thread's result is zero or not. Those bits from the 32 warp threads form a 32-bit zero mask that is captured in a separate buffer we called the *Mask Buffer*.

Fig. 4.4 shows the zero mask generation using in-lane data produced by four execution lanes. In write-back stage, we added a zero detection logic (ZD) to generate a mask bit for every 32-bit result produced by warp threads. When writing the warp result into the main register file, the generated 32-bit zero mask is used to gate off threads from writing their zero data into the register file. As the zero data is now captured in a compact form in the zero mask, the mask needs to be saved as it carries architectural states. Therefor, the zero mask is written into a per-warp buffer indexed by the warp destination register number.

When a new warp is issued into a collector unit, the mask buffer is accessed for every source operand to retrieve the zero mask value for that operand. As shown in 4.2, the zero mask is captured in the operand collector along with the operand index. When the register file read is performed for a given warp, the zero mask is used to gate off reads from thread

registers within the accessed warp register that would otherwise have a zero value written to them as shown in 4.3. We made our design such that the gated thread register reads would return zero by default. The read value for all threads within the warp including the gated ones is written into the operand collector and the source operand is then marked as ready.

Using the zero mask can be considered a form of data compression where a 32-bit zero data is captured by only 1-bit. Capturing this compressed information in a separate low-power structure allowed us to reduce the register file access power that is consumed to read and write zero data.

4.3.3 Using the Cross-lane Zero Mask

The execution pipeline for every warp thread in a GPU is 32-bits wide and is referred to as an execution lane. In each lane, a thread operates on a number of 32-bit wide operands and produces a 32-bit result. In general purpose applications, data values produced by warp threads sometimes have a small dynamic range which means that those data values can be represented by fewer than 32 bits with no loss of information. A 4-byte thread result may have one or more of the most significant bytes as zeros. In the case that the resulted 4-byte data for a given thread are all zeros, the result will not be written into the register file but instead the zero information is captured by the zero mask as presented in the previous section. For other non-zero result cases, despite having some of the upper bytes in a thread result as zeros, the whole 4-byte thread result from one execution lane is still written into the register file.

The same zero masking mechanism we applied on the register file earlier to reduce its dynamic power consumption is leveraged for the case of small dynamic range results. To do so, we arranged data bytes within groups of four execution lanes such that bytes that have the same byte-position in each group result are placed together. Using this technique, the presence of small dynamic range values in the original result may produce a re-ordered result having four zero-bytes that can be captured only by using the zero mask which would avoid writing those zeros into the register file. With this data re-ordering technique, we

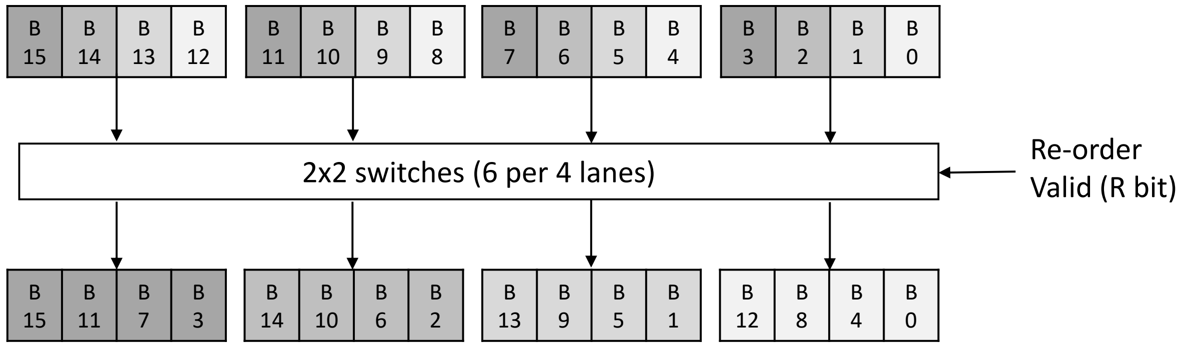


Figure 4.5: Data re-ordered in byte-position form to take advantage of low dynamic range values for power reduction.

have covered more data values produced in general purpose applications to help reduce the register file dynamic power even further. The zero mask generation using cross-lane data is presented in Fig. 4.4.

We clustered execution lanes into multiple groups with each group having 4 lanes. In each group, data bytes are ordered such that each four bytes that have the same byte-position are grouped together. That is, the least significant byte (byte number 0) in each and every lane will form the first 4-byte of the ordered result, bytes number 1, when grouped together, form the second 4-byte of the result and so forth. Notice that this allows the most significant bytes, which are zeros in small dynamic range results, to be grouped together to form 4-byte of zero values which are used to gate off register file access and help reduce its dynamic power.

4.3.4 Dynamic Zero Mask Selection

To support both in-lane and cross-lane zero mask generation, we integrated the two techniques, as shown in Fig. 4.4, such that the resulted zero mask is selected dynamically from one of the two techniques based on which of their generated zero masks has a higher number of zeros, and therefore, provides more power savings on register file access. To do this, we used a zero detection logic on both in-lane and cross-lane data and added a counter

to count how many zeros each mask has.

A zero mask is dynamically selected based on the counters result. Each group of four lanes makes this selection process independently. If the cross-lane zero mask is selected, the result data would need to be arranged based on byte-position order as explained in the prior section. We added six 2×2 switches that are controlled by a bit we called the *Reorder Bit* or the *R*-bit. This bit is set if the cross-lane mask is selected and the result data needs to be re-ordered. Otherwise, the result data stays in same order as in its original form. Fig. 4.5 shows how data is re-ordered when the *R*-bit is set. The *R*-bit is saved in the mask buffer along with the zero mask for the destination register being written. The result data, after passing through the re-ordering switch, is written into the register file. We used the same zero masking technique to gate off register file accesses that supports both in-lane zero masks in which we detect 4-byte zero result within execution lanes and the cross-lane masks in which we detect zero bytes in data result across a group of adjacent lanes.

4.4 Reducing Execution Unit Dynamic Power

As mentioned earlier, one of the largest components of GPU dynamic power comes from the SIMD execution units. To help reduce dynamic power consumption, we used a mask gating technique similar to that which we incorporated into the register file, which has no performance impact. As the divergent flow present in general purpose (GPGPU) programs forms under-utilized warps, the thread active mask can be used to gate off execution lanes for inactive threads found in a warp. The presence of zero data in source operands for a given thread gives an opportunity to gate off the thread execution for cases which result in *Trivial Operations*. Fig. 4.6 shows the mechanism used in gating the execution units.

4.4.1 Using the Active Thread Mask

The control divergence statements in an application program such as *if-else* statements cause under-utilized warps to be formed. On either path of the control divergence (taken or not-taken paths), only threads that pass the divergence condition will execute and the

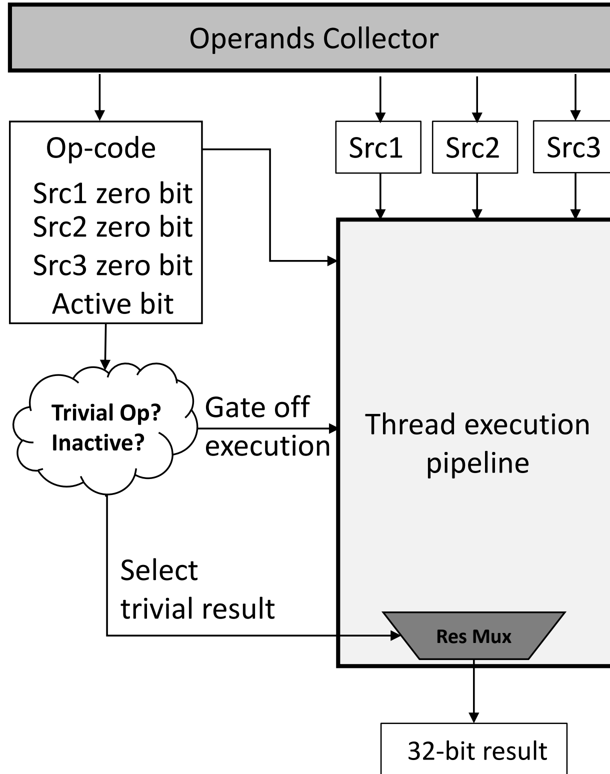


Figure 4.6: Power reduction for one execution lane using thread active mask bit and operands zero mask bits.

remaining threads will be treated as do-not care and their results will be discarded. For every warp, we captured its thread active mask in the collector unit along with source operands used and instruction decoded signals. When all operands are read from their register file and marked ready in the collector unit, the warp arbitrates with other ready warps to be issued into the SIMD execution pipeline. In our baseline design, an arbiter is used to select the oldest ready warp to issue next into the execution pipeline.

When a warp is issued into the execution pipeline, we used the thread active mask to gate off execution lanes for inactive threads in the warp. This involves gating off pipeline staging for all operands and results except for control signals such as the threads active mask. When the warp reaches the write-back stage, the zero mask will be generated by performing the zero detection on the warp result generated by all 32 execution lanes. However, for inactive

threads, the zero mask bits corresponding to those threads will not be updated since those threads will not modify the data that already exist in their destination registers. Active threads, on the other hand, will have their zero mask bits updated based on their result which control their write into the main register file.

4.4.2 Using the Operand Zero Masks

Some operations done by warp threads are trivial, particularly when performing arithmetic or logical operations on zero operand values. In this work, we categorize an operation as trivial if at least one of its operands is zero and its final result is either zero or equal to one of its source operands. As an example, multiplying or adding a zero operand with a non-zero operand, the result would be zero for the multiply operation and equals to the non-zero operand for the addition. The presence of such trivial operations in general purpose compute applications can be geared toward reducing execution unit dynamic power by avoiding their execution.

To detect if a thread has a zero operand, we used the zero mask bit corresponding to that thread which is captured in the operands collector unit. When a warp is selected to be issued into the execution pipeline, we used the zero mask bits for each thread operand and the instruction type (or class) to determine whether a trivial operation is to be performed by that thread or not. We also generate a result selection signal for every thread in the warp to select the final result of the trivial operation which would equal to either a zero value or one of the thread instruction operands.

When a trivial operation is detected for a given thread, the execution pipeline will be gated off except for the control signals used for that thread. This is similar to the inactive thread gating mechanism mentioned earlier except that in this case a result is still need to be generated by the thread that is turned off. In the last stage of the execution pipeline, the generated result selection signal is used to select the final result of that thread operation from either a zero or one of the source operands. The final result muxing in the last pipeline stage is expanded to support such trivial operations.

4.5 Evaluation

To evaluate our proposed techniques for register file and execution units dynamic power savings, we used GPGPU-Sim version 3.2 [21] and measured dynamic power consumption using GPU-Wattch [7]. We used a GPGPU model similar to Nvidia Maxwell GTX980 [39] with configuration parameters listed in table 4.1 and used the Rodinia benchmark suite [20] which consists of general purpose benchmarks that target GPU platforms.

Parameter Name	**Value**
Number of SMs	16
Number of partitions (cores)	4 per SM
Number of Warps	64 per SM
SIMD lanes width	32
Register file size	256 KB per SM
Register file banks	16
Register file width	128 B
Operand collectors (OCs)	16 per SM core
Warp scheduler	2-level, 1 per SM core
Streaming Processing Units (SPU)	32 per SM core
Special Functional Units (SFU)	8 per SM core
Load/Store Units (LDST)	8 per SM core
L1 cache size	16 KB per SM
Shared memory size	96 KB per SM
Added mask buffer size	2.25 KB per SM core
Added mask fields in OC	152 bits per OC

Table 4.1: GPU configuration parameters.

The proposed power reduction techniques have low area and power overhead. For zero mask generation, we added a zero detection logic which produces a bit mask for every 32-bit execution lane. The size of the mask buffer, which is used to save the generated 32-bit zero masks, is about 1/32 of the size of the register file. We added new fields in each operand collector to capture the threads active mask and the operands zero masks. We also added

2×2 switches for data ordering to support cross-lane zero gating. We estimated the power overhead of the proposed techniques to cost about 3% of the register file dynamic power and about 1% of the total GPGPU power.

4.5.1 Register File Power

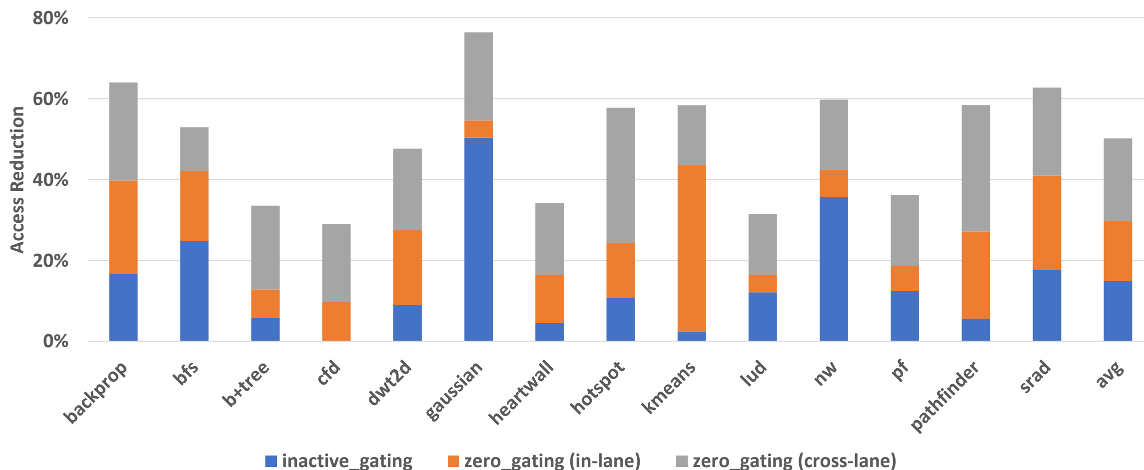


Figure 4.7: Register file access reduction for read requests using power savings techniques: access reduction using threads active mask (inactive_gating), in-lane operands zero masks (zero_gating(in-lane)), and operands zero masks with data re-ordering(zero_gating (cross-lane)).

In this section we present the dynamic power reduction in the GPGPU register file that we were able to achieve using the power reduction techniques presented earlier in Section 4.3. In the first technique, we applied the threads active mask to gate off reads and writes for inactive threads in a warp. The second technique involves using the generated zero mask to a void reading and writing zero data from the register file which instead is captured in the zero mask buffer. In the last technique, we expanded the use of the zero mask gating to cover low dynamic range data by re-ordering the data bytes in a warp result to group bytes with same byte-position together to potentially form a number of 4-byte zeros that can be suppressed from writing into the register file.

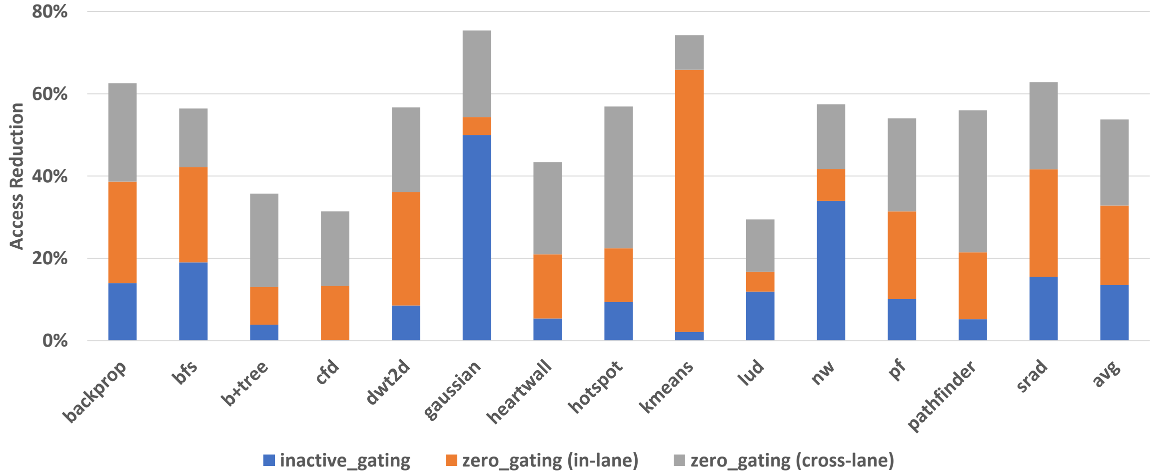


Figure 4.8: Register file access reduction for write requests using power savings techniques: access reduction using threads active mask (inactive_gating), in-lane operands zero masks (zero_gating(in-lane)), and operands zero masks with data re-ordering(zero_gating (cross-lane)).

Fig. 4.7 and Fig. 4.8 show the reduction in register file read and write accesses when applying the power saving techniques we just mentioned, respectively. Using only threads active mask to gate off register file accesses, the number of reads and writes made into the register file were reduced by an average of 15% and 14% of the baseline values, respectively. For the zero mask gating techniques, we show the access reductions from in-lane and cross-lane zero mask gating separately. The in-lane zero gating technique reduced the register file access further by 15% for reads and 19% for writes, on average. The cross-lane zero mask gating which involves data reordering provides more reductions on register file accesses. Using the techniques combined, the overall reduction in register file accesses achieved were 50% for reads and 54% for writes of the baseline values. The highest access reductions were seen on `GAUSSIAN` and `KMEANS` benchmarks due to high control divergence in the former and high percentage of zeros in the latter.

Fig. 4.9 shows the dynamic power savings in the register file when applying the aforementioned power saving techniques. The overall register file power reduced to about 73% of the baseline on average. From the 27% dynamic power reduction, the active mask gating

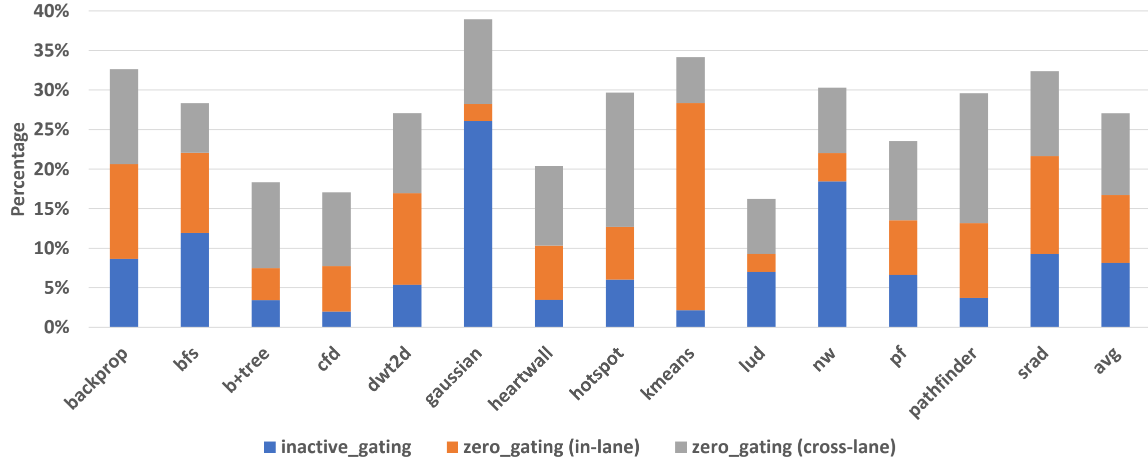


Figure 4.9: Dynamic power reduction in GPGPU register file contributed by power reduction techniques using threads active mask (inactive_gating), in-lane operands zero masks (zero_gating (in-lane)), and operands zero masks with data re-ordering (zero_gating (cross-lane)).

contributed to about 8%, the in-lane zero mask gating achieved an additional 9%, and finally the cross-lane zero gating added 10% more power savings. Note that these power reduction techniques were applied on the register file macros which consumes about 70-75% of the total register file power. Other components in the register file such as the cross bar network and the operand collector units did not benefit from such power gating techniques.

The power reduction from gating inactive threads varies between benchmarks with the highest reductions found in `GAUSSIAN` and `NW` benchmarks due to high thread divergence available in these benchmarks. In turn, this makes the contributions of zero gating in these high divergent benchmarks, which only applies to active threads, much less compared to low divergent benchmarks as in `KMEANS` and `PATHFINDER` which benefited most by the zero gating techniques.

4.5.2 Execution Units Power

As mentioned earlier, the execution units are considered one of the largest components in GPGPU and consumes about 20% of the total dynamic power. To help reduce their dynamic

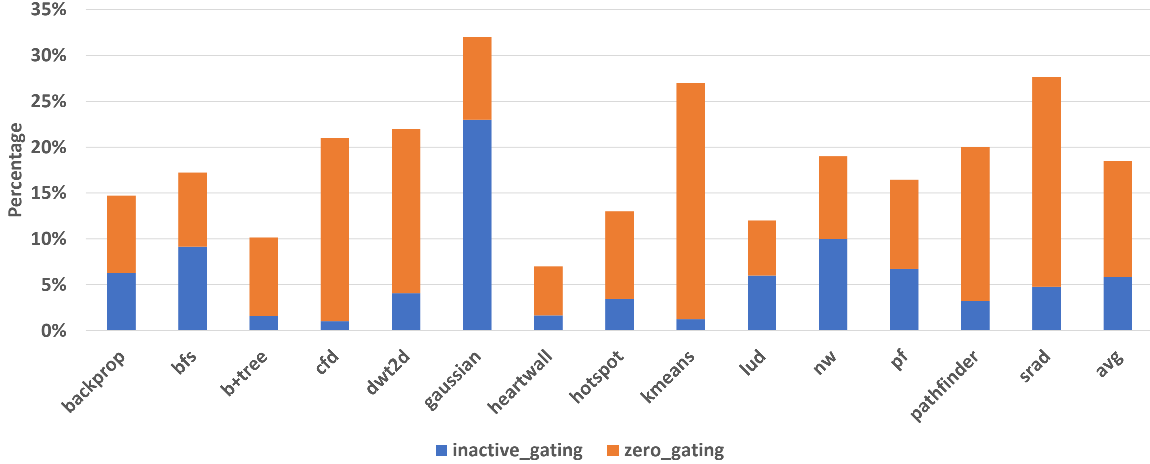


Figure 4.10: Dynamic power reduction in GPGPU execution units contributed by power reduction techniques using threads active mask (inactive_gating) and trivial operations handling using operands zero masks (zero_gating).

power consumption, we proposed two power saving techniques. The first technique uses the threads active mask to gate off execution pipelines for inactive threads found in a warp. And the second technique uses the operands zero masks to identify trivial operations and directly supply their final results without executing them to save dynamic power. We limited the scope of the trivial operations to the ones that have zero operands as we re-used the zero masks that are available in the operand collector. Also the generation of the final result of such trivial operations is simplified to only a multiplexer logic that is easily integrated in the execution data path. We accounted for the power of the added gating logic and data multiplexing in our results.

In Fig. 4.10, we show the dynamic power reduction in execution units using the power saving techniques we proposed. Gating off execution pipelines for inactive threads found in warps reduced dynamic power consumption by about 6% compared to our baseline which has no power gating techniques enabled. Disabling trivial operations through the use of operand zero masks achieved an additional 13% power reduction. Overall, by enabling both power saving techniques we proposed, the dynamic power for the execution units were reduced to

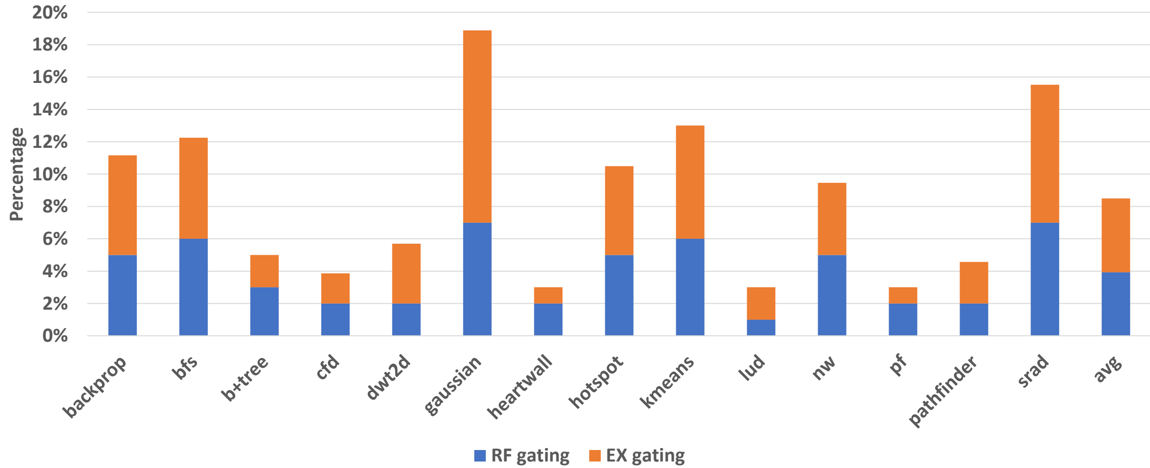


Figure 4.11: Dynamic power reduction in total GPGPU chip power contributed by power reduction techniques applied to the register file (RF gating) and the techniques applied to the execution units (EX gating).

about 81% of the baseline measured values on average across all benchmarks used.

As we applied our gating techniques on arithmetic execution units, the power reduction benefits may vary in benchmarks as the percentage of arithmetic type instructions available in these benchmarks varies. The highest power reduction in execution units seen is 68% for the `GAUSSIAN` benchmark which benefited most from the inactive threads gating. Avoiding execution of trivial operations gave significant power reductions in `SRAD`, `CFD`, and `KMEANS` benchmarks.

4.5.3 GPGPU total Power

In this work, we focused on two large components in GPGPU which consume over one-third of the total chip dynamic power. We proposed practical power reduction techniques that have no performance impact to help reduce their dynamic power consumption. As shown earlier, we were able to reduce the register file power by about 27% and the execution units power by about 19%. In Fig. 4.11 we show the contribution of power savings made on each component on the overall GPGPU dynamic power. The register file power savings techniques were able to achieve a 4% average reduction of the total chip dynamic power. The

techniques we used in execution units contributed to about the same amount of power savings to bring the total power reduction in the whole GPGPU chip to about 8% on average.

4.6 Related Work

Significant prior work exists in register file power reduction for GPUs. Gebhart et al proposed a small register file cache (RFC) to capture short-lived registers which would reduce read and write accesses to the main register file [15]. Similar work that used an RFC with software managed register prefetching is proposed by Sadrosadati et al [22]. A compile-time managed register file is proposed by Gebhart et al [23] which aims at reducing dynamic power consumption. In this work, the register file is partitioned into multiple levels and the compiler is used to leverage its knowledge of registers usage to determine where to allocate values across the register file hierarchy. Similar work that used both a register file cache and a hierarchical register file is proposed by Bailey et al [24]. A partitioned register file is proposed by Abdel-Majeed et al where less frequent accessed registers are placed in a slow register file that operates in a lower voltage and frequently accessed registers are placed in a small and fast register file [26]. The technique targeted both dynamic and leakage power reduction and both compile-time and run-time profiling had to be made to collect register access statistics needed. A unified local memory structure with partitioning of capacity among register file, data cache, and scratchpad memory is proposed by Gebhart et al [25]. Kloosterman et al proposed replacing the main register file with a lower power operand staging unit [30]. Operands are allocated space in the staging unit using compiler annotations that determine future registers usage. Registers are all kept in memory and fetched into the staging unit when needed. Register renaming is done by Jeon et al to reduce the physical register file size and its power consumption [29]. All of these proposed power reduction techniques are orthogonal and potentially complimentary to the zero gating techniques we proposed in this work which rely on data values produced during programs execution as well as the status of executing warps.

Data compression has been proposed by Lee et al for GPU register files to reduce dynamic

power [31]. This work used the Delta-Base-Immediate (DBI) compression technique that Pekhimenko et al proposed for data caches [32]. The DBI mechanism requires adding a vector-wide adder-subtractor unit to compress data before writing to the register file and another adder-subtractor to de-compress the read data out of the register file which has a high area and power overhead. Another form of data compression is proposed by Liu et al to handle scalar execution in GPGPU where duplicate values in thread registers are captured in a separate scalar buffer to save access power [33]. In this technique, data bytes in a result are compared against a selected base value and bytes that are equal to the base are not written into the register file. This compression technique requires a large buffer to save the base values with a size of over 1/4 of the register file and it also has a performance penalty of about 2%. In contrast, the proposed techniques in this work have no performance overhead and the size of the mask buffer used is only 1/32 of the size of the register file. Dusser et al proposed a form of zero value compression for CPU data caches which added a Zero-Content Augmented cache (ZCA) to capture null blocks presented in application programs to save on storage resources available [40]. Abdel-Majeed et al aimed at reducing leakage power by operating the register file in different power modes [27]. They also proposed using an active mask gating on the register file to reduce dynamic power consumption which is done on 128B entries using Divided Word Line (DWL) approach previously proposed by Yoshimoto et al [28]. This is similar to the active mask gating technique we used on the register file. However, our proposed technique does not require any structural changes inside the register file macros. Also our masking technique is applied on both register file and execution units. Data-path slicing is proposed by Gilani et al to take advantage of low dynamic range values that can be represented by 16 bits [36]. The 32-bit thread registers are also split into low and high halves to be controlled separately which would reduce the dynamic access power. The main objective of this work was to increase the warp issue rate by issuing two warps with 16-bit data in same cycle which required modifying the warp scheduler and register file banking scheme. In our proposed techniques, we took advantage of 8, 16, and 24-bit values

to reduce register file power without any updates made to the warp scheduler or how the register banks are statically arranged.

4.7 Conclusion

In this work, we proposed power reduction techniques to help reduce register file and execution units dynamic power consumption in the GPU. Our work takes advantage of the frequent zero data, narrow-width data, and under-utilized warps found in general-purpose compute applications. Two types of gating masks are applied to the register file and execution units: the active-thread bit mask and the zero-operand bit masks. The proposed techniques achieved a 27% reduction in register file power and a 19% reduction in execution units power. The overall power reduction in GPU was ~8% for GPGPU workloads from Rodinia benchmark suite. Our proposed techniques have low design overhead and required minor micro-architectural changes to the GPU that have no performance impact.

5. MULTI-PROCESSOR FULL SYSTEM SIMULATION AND THE IMPACT OF LINUX THREAD SCHEDULER

5.1 Introduction

As we mentioned in Section 1.3, this work focuses on the impact of system software (the Linux kernel) on the behavior and correctness of simulation experiments performed on full system simulation running multi-threaded applications. We show that, for the short runtimes used in architecture research, the operating system thread scheduler does not behave as we expected to provide load balance and fully utilize the simulated multi-core system. The load imbalance in the simulated system leads to unpredictable and inconsistent simulation results. We provide an update to the scheduler, for the use in architecture research, to improve the consistency and correctness of multi-threaded applications that run on the full system simulation environment.

5.2 Behavior of Thread Scheduler in Full System Simulation

As previously discussed, due to the extreme differential between simulation runtime and real system runtime, typically small input sets are used in architecture research. We now focus our attention on the behavior of the current Linux[1] scheduler (the "Completely Fair Scheduler" (CFS) [16] used in version $4.x$) during these short-lived simulation experiments and demonstrate how system software can in fact impact performance and correctness of these experiments. To do so, we depict the behavior of the scheduler when running a multi-threaded Canneal benchmark with a small input set on the gem5 full system simulator.

5.2.1 Thread Scheduling and Load Imbalance

Fig. 5.1 demonstrates the behavior of Linux scheduler when the Canneal benchmark runs on the gem5 simulator. In this benchmark, a parent thread forks a number of child

[1] We focus on Linux here, as it is the OS typically used in architecture research. We note that similar behavior can be found in other OSes.

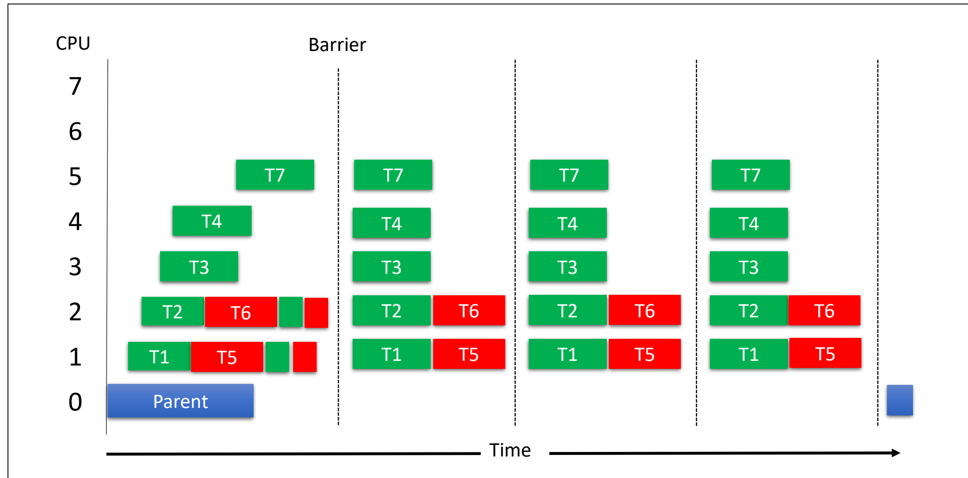


Figure 5.1: Behavior of Linux scheduler for a multi-thread benchmark running on architecture simulator.

threads while running on core 0 from the start of simulation. When all threads are forked, the parent thread goes into the idle state until threads complete their work and all join at the last barrier. Then the parent thread gets re-activated, reports the simulation result, and finally exits. The figure shows a time-line running from left to right, with each row representing a different core within an 8-core system. Running threads are represented by the green or red bars within a row. Since the number of software threads is no more than the number of cores, one would expect each thread to be mapped into a distinct core. We see in this example, however, that core 1 and core 2 have two threads each and the result of this causes the simulation to run twice as long as it should.

There are two scheduling issues in this particular case which cause this load imbalance. First, the scheduler overloads some of the cores with tasks while others are idle. Second, the scheduler does not correct the imbalance in the system after it has happened. After a new thread is initiated, the scheduler sequentially searches, with minimum effort, for an idle or otherwise a light-weight loaded core in the system to map the new thread. Critically, this search starts from the same core ID each time, taking into no account whether that core has already had a given thread mapped to it in the recent past, only whether or not that

core is currently idle. We note that a core with threads in wait mode, perhaps due to a long latency page fault or waiting on a barrier, will be viewed as "idle". In this case, the scheduler would map the new thread into this found-to-be idle core, thereby creating an unintentional load imbalance in the system. In the figure, we see that, because the parent thread is slowly spooling out forked threads, by the time it is ready to fork off thread 5, core 1 has become idle because it hit a page fault. Thus instead of placing thread 5 on core 5, it is placed on core 1. Similarly, thread 6 is placed on core 2.

5.2.2 Periodic Load balancing

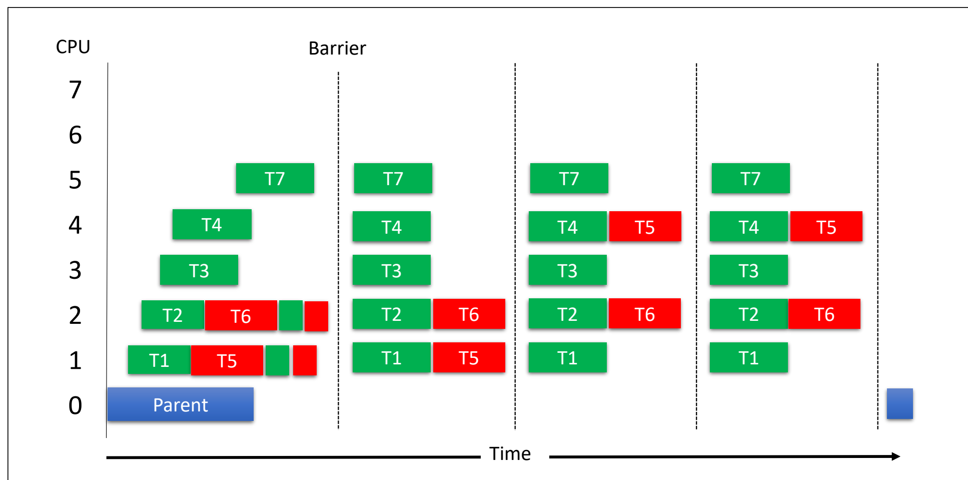


Figure 5.2: Periodic load balancing done by Linux scheduler for a multi-thread benchmark running on architecture simulator.

To address the adapting load per core, the OS periodically (approximately once every 30 ms) performs a load balancing operation on all cores to incrementally reduce the degree of load imbalance and enhance applications performance. The heavy-weight system-wide search is initiated by one core at a time in a sequential order to reduce contention and avoid ordering complexity. Thus, one full iteration of the core balancing requires $30 * N$ ms, where N is the number of cores. Unfortunately, because the system is not balanced it will often

end up migrating a thread from an over-committed core to one that has one thread but is not currently busy because that thread is waiting on a barrier as shown in Fig. 5.2. In the figure, we see that, the scheduler tried to rebalance the load in the system by moving thread 5 from the over-committed core 1 into core 4 since it was found idle during the rebalance attempt. However, core 4 already had thread 4 which was waiting on a barrier and therefore the rebalance attempt done in this case has no value.

Over the long haul, this migration will eventually settle on a balanced configuration of threads to cores, however it can often take many iterations to do so. Since the life-time of typical architecture simulations is very short though, this balance point is typically not found prior to the end of simulation. In fact, ten full iterations of core rebalancing on a 16-core system could take as long as 5 seconds to find an optimal balance, much longer than the runtime of the benchmarks used in architecture research (see Fig. 1.1).

5.2.3 Immediate Load balancing

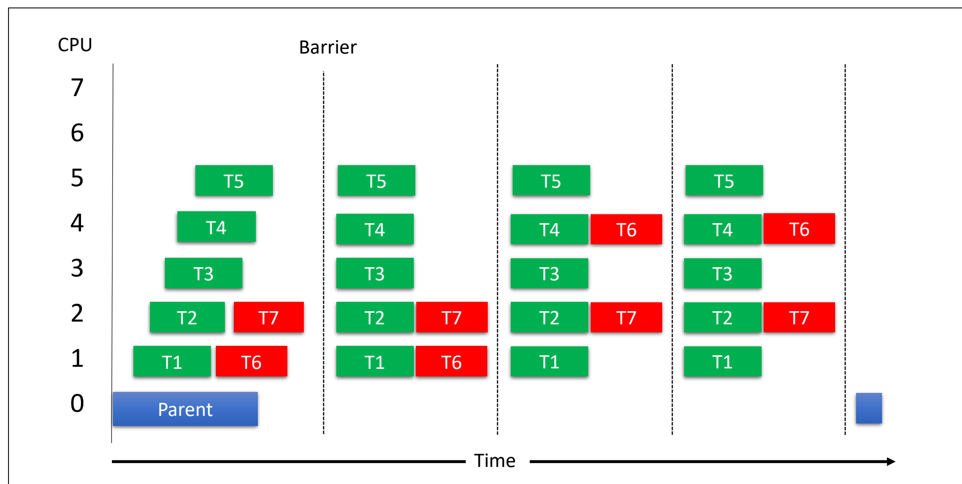


Figure 5.3: Behavior of Linux scheduler with immediate load balancing for a multi-thread benchmark running on architecture simulator.

Linux scheduler can optionally perform an immediate load balancing when new threads

become runnable [41]. This option, *relax_ domain_ level*, is supposed to ensure that as many cores as possible are usefully employed running tasks. Fig. 5.3 shows thread-core mapping for the same experiment with immediate load balancing enabled. The scheduler behavior is a little different than previously shown in terms of thread mapping and we can see threads 1 to 5 get mapped as expected. However, threads 6 and 7 are still mapped to already subscribed cores. The reason this happens is due to core 1, for instance, being idle at the time the scheduler performs the search for a candidate core to map the newly forked thread. The thread that was running before on core 1 finished its first loop iteration and was in sleep mode waiting for other threads to finish their iteration and all join at the barrier. This situation happens primarily in architecture simulation, where benchmarks are run with small input sets and thus they typically complete in less time than it takes for the scheduler to optimally balance all threads across the available cores in the system. The scheduler only uses current core status information with no knowledge of application-level behavior and therefore such load imbalance is likely to happen. Both immediate and periodic load balancing processes were not able to correct the initial mapping of the scheduler and load imbalance remained until the end of simulation.

5.3 Proposed Solution

To avoid the issues mentioned above in the current Linux scheduler specifically in architecture simulation, we propose a patch to the scheduler to enforce mapping of threads into unsubscribed cores in the CMP system in a round robin fashion. The proposed patch does not only avoid the complexity and overhead associated with load balancing in current scheduler but also guarantees correct experimental results when running multi-thread benchmarks in architecture simulation environment. Note that, we chose not to use affinity masks to map threads to cores in user software since using such masks is not a general solution to the problem; it would require knowledge of exactly how many threads a given benchmark will spawn and how many cores there are in the system under test (several benchmarks spawn extra, helper threads which are short running and do not impact performance).

The current thread scheduler views the core resources as in either *active* or *idle* states. Active cores are the ones having threads running on them whereas idle cores are either having no threads mapped to them or the mapped threads are waiting on an exception to return or a barrier to be reached. Thread mapping is performed by the function *sched_balance_self()* in ./linux/kernel/sched.c [19, 42]. The scheduler searches among the group of available cores to find an idle core to serve a newly forked thread. The search is done sequentially starting at core 0 every time. As we mentioned, a core might be found idle and get selected as a candidate to run the new thread even though it has threads in wait or sleep mode which would create an imbalance situation that cannot be resolved in architecture simulation.

Under certain circumstances, the scheduler may decide not to migrate threads off a busy core in the presence of load imbalance in the system due to its impact on performance or power. As an example, a thread that is found to be a cache-hot will not be considered as target for migration due to the high cost of throwing away its private cache data. Another example is when the scheduler is configured in a low power mode it tries to keep threads running on active cores and would rather not to wake up idle cores to balance the load in the system. Therefore, making correct mapping decisions from the start of simulation greatly influence performance and reduce scheduling overhead in architecture simulations.

To avoid the load imbalance issue, we classified the idle cores into two different states; idle cores that have no threads mapped to them (*idle-no-thrd*) and idle cores with mapped threads in wait or sleep mode (*idle-thrd*). Fig. 5.4 shows thread to core mapping with the proposed update. For a new forked thread, the scheduler is forced to first search among the group of cores in idle-no-thrd state to map the thread in a round-robin fashion. If no core is found, it searches among the idle-thrd cores group and then the active group to find a candidate core. Only when the number of software threads exceeds the number of available cores that the idle-thrd and the active core groups are searched in the same way that the original scheduler search is done. Forcing the scheduler to first look for a candidate core among the idle-no-thrd group guarantees a 1-to-1 mapping of threads to cores when the

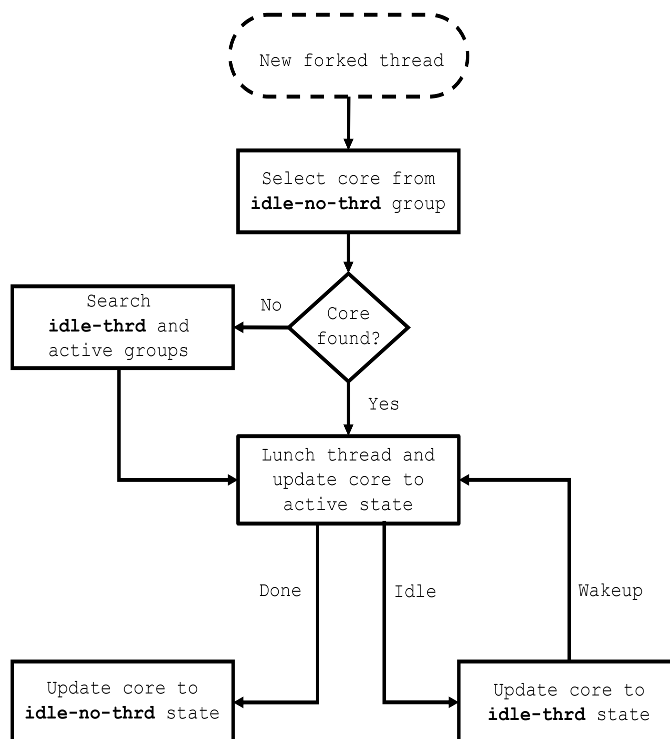


Figure 5.4: Mapping of new thread and core status update with the patched scheduler.

number of threads is no more than the number of available cores in the system and avoids the load imbalance issue from the start of simulation.

Fig. 5.1 in section 5.2 shows the load imbalance issue in a multi-core simulation when the original Linux scheduler is used. We repeated the same experiment with the patched scheduler to examine its behavior. Fig. 5.5 shows the result of running an 8-thread Canneal benchmark on an 8-core simulation system with the proposed scheduler updates. In the figure, we see that every software thread in the benchmark gets mapped to a distinct core in the system and that the load imbalance issue is prevented from the start of simulation.

5.4 Evaluation

In order to see the impact of load imbalance on multi-thread benchmarks with different input sets, we ran the Canneal benchmark from the PARSEC benchmark suite [11] on a hardware machine with native, large, medium, and small input sets. The hardware machine

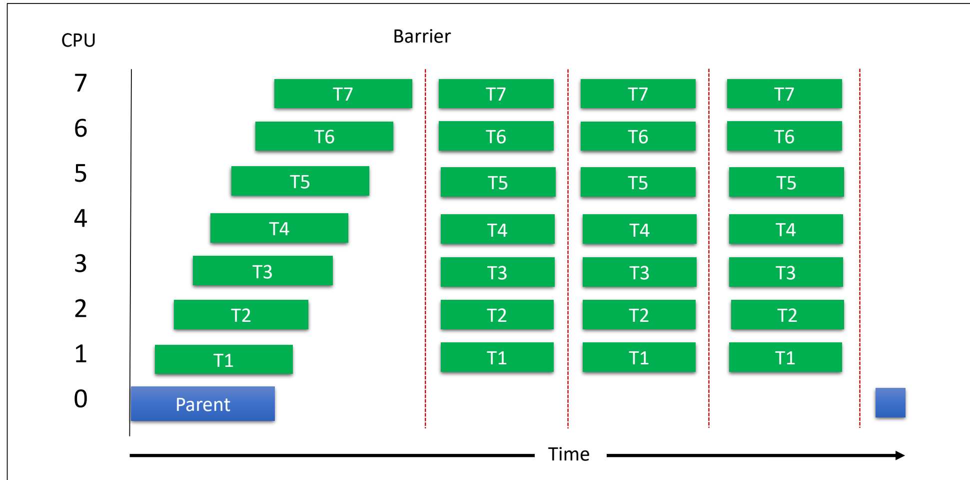


Figure 5.5: Behavior of patched Linux scheduler for a multi-thread benchmark running on architecture simulator.

has 12 cores and each core is single threaded. A list of the machine's configuration obtained using Linux command *lscpu* is shown in Table 5.1.

Parameter	**Configuration**
Architecture	x86_64
Core op-mode(s)	32-bit, 64-bit
Core(s)	12
Core frequency	1900 MHz
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	15360K

Table 5.1: Configurations of the hardware machine used.

Our simulation experiments were performed using the gem5 full system simulation environment [12] in order to show the impact of system software on the validity and correctness of simulation results. Latest Linux with CFS scheduler was used as the baseline OS for the full system simulation environment. We simulated a multi-core CMP system with 16 homoge-

neous cores connected by a network-on-chip (NOC) and with L1 private cache and an L2 split cache. Each core has a single threaded CPU with an out-of-order execution pipeline which ran at 2.66 GHz clock frequency. Table 5.2 lists the system configurations used to perform the simulation experiments. We ran multi-thread benchmarks from PARSEC benchmark suite on the simulated CMP system with memory bus speeds of 200 MHz and 800 MHz. The simulation experiments were repeated on a simulated CMP system with the patched version of Linux to compare against the baseline version and validate our proposed solution.

Parameter	**Configuration**
Cores	16
Threads per core	1
Pipeline	Out of Order
Core frequency	2.66 GHz
Architecture	x86_64
L1 cache	32 KB
L2 cache	256 KB per Core
Network topology	4x4 2D Mesh
Network routing	X-Y DOR
Directories	4
Coherency	MESI protocol

Table 5.2: Configurations of the multi-core simulator used.

To see the impact of short simulation run times on workload balance on a real machine running a modern, Linux operating system, we ran the Canneal benchmark with 12 software threads for each input set on the system described above. The per-core load results from this experiment are shown in Fig. 5.6. In this figure, we see the load is well balanced among the cores when the native input set is used. This indicates that the load imbalance when threads are first mapped to the cores is resolved by the Linux scheduler and has negligible effect on the overall performance for real applications. For input sizes that are appropriate for simulation runs, we see the degree of load imbalance increases as we go with smaller input

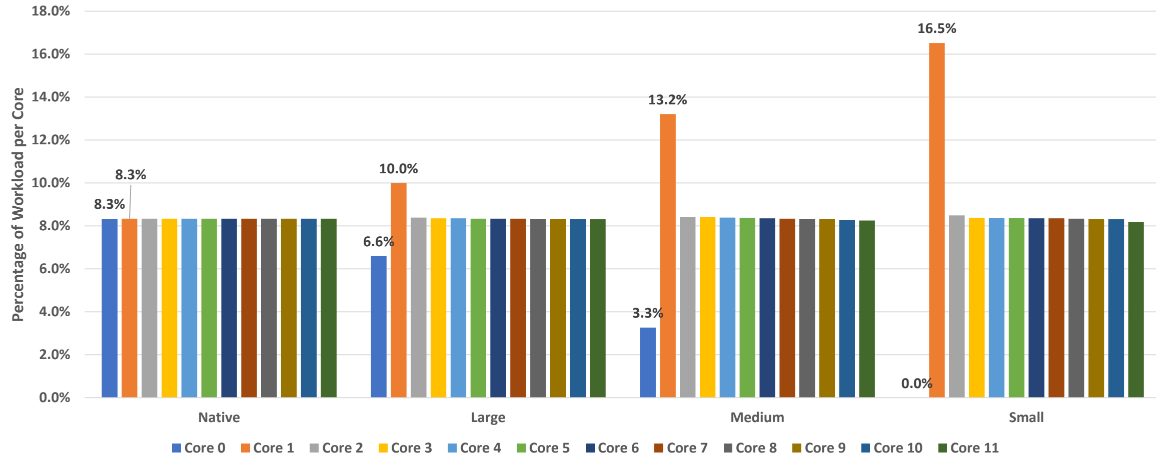


Figure 5.6: Percentage of workload per core for 12-thread Canneal benchmark with different input sets run on 12-core hardware machine.

sets. With small input set, we see that core 1 has about twice the load of the other cores while core 0 remains idle. The proposed patch to the scheduler should avoid load imbalance for small input size in particular and any other input sizes as well as for any arbitrary number of software threads used.

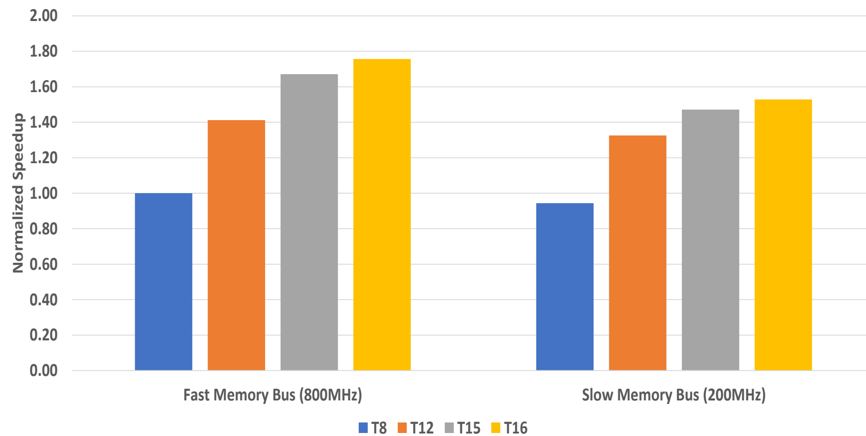


Figure 5.7: Performance speedup for Canneal benchmark using small input set, with 8, 12, 15, and 16 threads under memory speeds of 200MHz and 800MHz in full system simulation with the patched Linux scheduler. Results are normalized against an 8-thread 800MHz case.

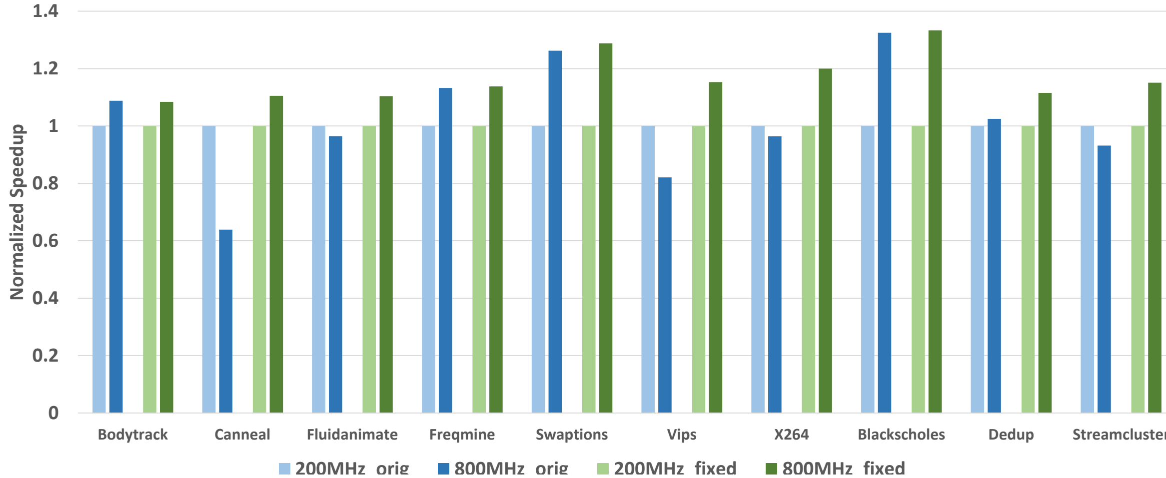


Figure 5.8: Normalized performance speedup for PARSEC benchmarks runs with two memory speed settings using current and patched Linux schedulers in full system simulation.

Fig. 1.2 in Section 1.3, shows the impact of load imbalance caused by the current Linux scheduler in architecture simulation. In that experiment, we ran Canneal with different software thread counts and two memory bus frequencies (200 and 800 MHz). There we found that the thread load imbalance on short runtime loads leads to incorrect results, such as that a faster bus speed yields a slower system performance and that more threads/cores can produce worse runtimes. These discrepancies and incorrect results in simulation experiments are the main reason for having the scheduler patch we propose here. We implemented the Linux kernel changes described in Section 5.3 and reran the same experiment. Fig. 5.7 shows the results of simulation runs using the patched thread scheduler on the same Canneal benchmark with a small input set. In the figure we see a near linear performance scaling with core count from 8 to 16 threads for both 200 MHz and 800 MHz bus frequencies. Further we see the expected (if small) performance improvement going from a 200 MHz to an 800 MHz bus.

Fig. 5.8 shows the effects of the baseline and patched schedulers across all the PARSEC benchmarks for a 200 MHz and an 800 MHz memory bus. In this experiment all benchmarks are run with 16 threads on 16 cores simulated system. Results are normalized to the

baseline scheduler. In the figure we see that the baseline scheduler always produces runtimes significantly longer than the patched scheduler for the 800 MHz runs. Further we see that, while in some cases the 800 MHz bus produces longer runtimes than the 200 MHz bus for the original scheduler (Canneal, Fluidanimate, Vips, X264, and Streamcluster); the faster bus always produces a shorter runtime with the patched scheduler. These results indicate that the patched scheduler produces much more consistent and correct results for multiprocessor benchmarks.

5.5 Conclusion

In this work, we addressed the behavior of Linux thread scheduler on full system architecture simulation when small input sets are used in multi-threaded applications. We focused on how threads are mapped into the available cores in the simulated system and showed that the current scheduler behavior, with small input sets, cased a load imbalance that led to a slower and non-representative performance causing incorrect experimental results. We provided a simple update to the scheduler to fix the undesired behavior in architecture simulations by forcing a round-robin mapping of software threads into available cores in the system and avoid the load imbalance issue from the very beginning. We evaluated our proposed solution using a gem5 full system simulator with a Linux OS and ran multi-threaded benchmarks from PARSEC suite.

6. CONCLUSION

With the increasing importance of thread parallel computations, our work focuses on improving performance and energy efficiency of thread parallel hardware and making its simulation more accurate. We achieved our objectives through the following three parts:

In the first part of our work, we targeted the access port limitation on register file and operand collector units that were causing access serialization and increasing access latency which negatively impact overall IPC performance and energy efficiency in the GPU. We focused on making the GPU more energy efficient and higher performance for general-purpose applications with frequent narrow-width data. Similar to the concept of access coalescing used in memory system, we introduced a new register file organization that supports register access coalescing in the GPU. Our design addresses the many limitations found in CORF design and provides far more register coalescing capabilities with far less design overhead and complexity. Our design is capable of coalescing a combination of narrow-width read and write requests targeting a register file bank. It is also capable of coalescing read requests from the same warp instruction targeting different banks. In addition, our design does not restrict register coalescing to only registers that are packed in the same physical entry or registers that belong to the same warp within a bank. It also does not restrict warp registers to be mapped into the register file banks in a specific order. Besides being superior in terms of performance and energy efficiency to CORF design, our new register file design has low cost and complexity as it does not require register packing, virtualization, nor allocation based on compile-time hints. On general-purpose benchmarks from Rodinia suite, our coalescing-aware design achieved a significant reduction on register file and operand collector accesses that led to a 16.5% IPC performance speedup and a 32.2% dynamic energy reduction in GPU on average.

As power constraints are challenging the compute scaling of future GPGPUs within a

limited power budget. In the second part, we focused on reducing dynamic power consumption for the two most power consuming components in the GPU, the main register file and the execution units. We proposed power saving techniques that collectively take advantage of the presence of under-utilized warps in general purpose compute applications as well as the high percentage of zero data produced in these applications to reduce both register file and the execution units dynamic power. Using the proposed techniques, we were able to achieve a 27% reduction in register file power and a reduction of about 19% in execution units power. The overall power reduction for the GPU chip was ~8% for GPGPU workloads. The proposed techniques required only minor micro-architectural changes to the GPU design and have low area and power overhead. The techniques were integrated into the SIMD data-path without introducing any performance penalty.

In the last part of our work, we addressed inaccuracies in full system simulations that use a Linux kernel to run multi-threaded applications, with small input sets, on multi-core system. We demonstrated the behavior of current Linux scheduler when multi-thread benchmarks are simulated on a multi-core system using gem5 full system simulator. We focused in particular on how the current scheduler maps software threads onto the available cores in the system. We showed that the load imbalance caused by the current scheduler has a significant impact in architecture simulations as they run for a very short duration and the scheduler does not have the time needed to effectively load balance the simulated system. We showed how the imbalance leads to a slower, non-representative performance and causes undesired behavior in simulation experiments, leading to incorrect experimental results versus a real system using a native input. We provided a patch to the scheduler to fix the mapping primarily for architecture simulation by forcing the scheduler to perform a round robin mapping of software threads into available cores in the system to avoid the load imbalance issue and its side effects from the start of simulation.

REFERENCES

[1] Nvidia, "Whitepaper: Nvidia's next generation cuda compute architecture: Fermi," 2009.

[2] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "Corf: Coalescing operand register file for gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 701–714, 2019.

[3] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct 1974.

[4] C. Martin, "Multicore processors: Challenges, opportunities, emerging trends," in *Proceedings Embedded World Conference 2014, Nuremberg, Germany, Design & Elektronik*, February, 2014.

[5] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burgur, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[6] Nvidia, "Whitepaper: Nvidia tesla v100 gpu architecture," 2017.

[7] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 487–498, ACM, 2013.

[8] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 cpus," in *Proceedings of the 48th Design Automation Conference*, DAC '11, (New York, NY, USA), pp. 1050–1055, ACM, 2011.

[9] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev, "Mptlsim: A cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches," *SIGARCH Comput. Archit. News*, vol. 37, pp. 2–9, July 2009.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, Feb. 2002.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[13] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[14] Nvidia, "Nvidia cuda c programming guide," *Compare A Journal Of Comparative Education*, 01 2010.

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

[16] M. T. Jones, "Inside the linux 2.6 completely fair scheduler," in *IBM DeveloperWorks*, December 2009.

[17] J. Aas, "Understanding the linux 2.6.8.1 cpu scheduler," *SGI, 2005. http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf, accessed on August*, vol. 22, p. 05, 2005.

[18] "Linux kernel documentation." https://www.kernel.org/doc/Documentation/scheduler/sched-domains.txt.

[19] D. Bovet and M. Cesati, *Understanding the Linux kernel.* OReilly, 2006.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, IEEE, 2009.

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

[22] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, "Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching," in *ACM SIGPLAN Notices*, vol. 53, pp. 489–502, ACM, 2018.

[23] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 465–476, IEEE, 2011.

[24] J. Bailey, J. Kloosterman, and S. Mahlke, "Scratch that (but cache this): A hybrid register cache/scratchpad for gpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2779–2789, 2018.

[25] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 96–106, IEEE, 2012.

[26] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram, "Pilot register file: Energy efficient partitioned register file for gpus," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 589–600, IEEE, 2017.

[27] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for gpgpus," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 412–423, IEEE, 2013.

[28] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, and T. Nakano, "A divided word-line structure in the static ram and its application to a 64k full cmos ram," *IEEE Journal of Solid-State Circuits*, vol. 18, no. 5, pp. 479–485, 1983.

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

[30] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: Just-in-time operand staging for gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 151–164, ACM, 2017.

[31] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: enabling power efficient gpus through register compression," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 502–514, ACM, 2015.

[32] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 377–388, ACM, 2012.

[33] Z. Liu, S. Gilani, M. Annavaram, and N. S. Kim, "G-scalar: Cost-effective generalized scalar execution architecture for power-efficient gpus," in *2017 IEEE International*

Symposium on High Performance Computer Architecture (HPCA), pp. 601–612, IEEE, 2017.

[34] X. Wang and W. Zhang, "Gpu register packing: Dynamically exploiting narrow-width operands to improve performance," in *2017 IEEE Trustcom/BigDataSE/ICESS*, pp. 745–752, IEEE, 2017.

[35] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, pp. 304–315, IEEE, 2004.

[36] S. Z. Gilani, N. S. Kim, and M. J. Schulte, "Power-efficient computing for compute-intensive gpgpu applications," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 330–341, IEEE, 2013.

[37] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "Reg-mutex: Inter-warp gpu register time-sharing," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 816–828, IEEE, 2018.

[38] Y. Tamir and H.-C. Chi, "Symmetric crossbar arbiters for vlsi communication switches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 13–27, 1993.

[39] Nvidia, "Whitepaper: Nvidia geforce gtx 980," 2014.

[40] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 46–55, ACM, 2009.

[41] "Linux man pages." `http://man7.org/linux/man-pages/man7/cpuset.7.html`.

[42] R. Love, *Linux kernel development*. Pearson India, 2nd ed., 2010.