UNIVERSITY OF CALIFORNIA
RIVERSIDE

Enhanced Register Data-Flow Techniques for High-Performance, Energy-Efficient GPUs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Hodjat Asghari Esfeden

June 2021

The Dissertation of Hodjat Asghari Esfeden is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This thesis is the result of many years of hard work that simply would not have been possible without the continuous support of many people. Undoubtedly, I could not have done this without the unwavering support and mentorship from my advisor Professor Nael Abu-Ghazaleh. Nael gave me the freedom to pursue any research direction I was passionate about. His motivation, immense knowledge, and great insight into research inspired me to overcome all difficulties in the past six years. I will forever be thankful to him for his endless support. I really appreciate the effort and time you invested in helping me to achieve my full potential in life, Nael. It is my great pleasure and honor to have known you and worked with you. I would also like to thank all of my collaborators and mentors, especially Farzad Khorasani from Tesla as well as Amin Farmahini-Farahani from Google. Finally, I would like to acknowledge my committee members, Professors Laxmi Bhuyan, Rajiv Gupta, and Daniel Wong for their help and support throughout my PhD.

The past six years at Riverside have been some of the most wonderful time in my life, working with the talented and friendly colleagues in our research lab and spending time with my supportive friends. I would like to thank my labmates: Khaled Khasawneh, Fatemah Alharbi, Sankha Dutta, Shafiur Rahman, Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, Ahmed Abdo, Sakib Md Bin Malek, Jason Zellmer, Abdulrahman Bin Rabiah, and Shirin Haji Amin Shirazi. I would like to thank my dearest friends, Fatemeh Ganjisaffar, Mohammad Bakhshalipour (Soltan), Majid Tanha (Yakka), Nahid Kazemi, Ehsan Faghih, Fazel Arasteh (Alex), Hanieh Jafarzadeh, Amirhossein Mirhosseini, Alireza Ramezani, Arezoo Etesamirad, Amirmahdi Mohammadzadeh, Saba & Saghi Baraghani, Hadi Mardani, Kimia

Zamiri, Reza Mokhtari, Niloufar Ghavi. Thank you for being there whenever I have needed you. Without you, I would not experience this fascinating PhD life.

Finally, and also most importantly, the completion of my dissertation would not have been possible without the support and nurturing of my parents, Heidar Asghari Esfeden and Fatemeh Ramezani. I feel so honored and blessed to have you as my parents. Thank you for instilling me with a strong passion for learning and for doing everything possible to put me on the path to greatness. I am also extremely grateful to my brothers, Javad and Sadjad, my sister, Nadjmeh, and my sister in law, Masoumeh for their significant support and encouragement, throughout my life. No matter how far you are and how long I was not able to visit you all, I feel you next to me every single day. Although this period was no easier for you than it was for me, you remained more than supportive all the time and gave me hope and encouragement every time I was about to give up.

To my beloved parents, for their endless love, support, and encouragement.

ABSTRACT OF THE DISSERTATION

Enhanced Register Data-Flow Techniques for High-Performance, Energy-Efficient GPUs

by

Hodjat Asghari Esfeden

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2021
Prof. Nael Abu-Ghazaleh, Chairperson

To avoid immoderate power consumption, the chip industry has shifted away from high performance single threaded designs to high throughput multi-threaded designs. Graphic Processing Unit (GPU) is a great example of such high throughput multi-threaded designs. GPUs have emerged as an important computational platform for data-intensive applications in a plethora of application domains. They are commonly integrated in computing platforms at all scales, from mobile devices and embedded systems, to high performance enterprise-level cloud servers.

GPUs use a massively multi-threaded architecture that exploits fine-grained switching between executing groups of threads to hide the latency of data accesses. In order to support this fast context switching at scale, GPUs invest in large Register Files (RF) to allow each thread to maintain its context in hardware. RF is a critical structure in GPUs responsible for a large portion of the area and power; the frequent accesses to the register file during kernel execution incur a sizable overhead in GPU power consumption, and introduce delays as accesses are serialized when port conflicts occur. This dissertation presents novel synergistic compiler/microarchitecture techniques for enabling high-performance and energy-efficient GPUs.

Our first technique, CORF, is a compiler-assisted Coalescing Operand Register File which performs register coalescing by combining reads to multiple registers required by a single instruction, into a single physical read. To enable register coalescing, CORF utilizes register packing to co-locate narrow-width operands in the same physical register. Our proposed design uses compiler hints to identify which register pairs are commonly accessed together. This novel technique simultaneously reduces the leakage and dynamic access power, while improving the overall performance of the GPU.

The second technique, Breathing Operand Windows to exploit bypassing in GPUs (BOW), is motivated by the observation that there is a high degree of temporal locality in accesses to the registers: within short instruction windows, the same registers are often accessed repeatedly. To exploit this opportunity, we propose an enhanced GPU pipeline and operand collector organization that supports bypassing register file accesses and instead passes values directly between instructions within the same window. To further arise bypassing opportunities, we introduce compiler optimizations to help guide the write-back destination of operands depending on whether they will be reused to further reduce the write traffic. Our results show that BOW can shield the register file from unnecessary register file accesses, which improves performance and reduces the energy consumption.

In our third study, inspired by the fact that registers are the fastest and simultaneously the most expensive kind of memory available to GPU threads, we propose Register Mutual Exclusion (RegMutex). RegMutex a software-hardware co-mechanism to enable sharing a subset of physical registers between warps during the GPU kernel execution. With RegMutex, the compiler divides the architected register set into a base register set and an extended register set. While physical registers corresponding to the base register set are statically and exclusively assigned to the warp, the hardware time-shares the remaining physical registers across warps to provision their extended register set. Therefore, the GPU programs can sustain approximately the same performance with the lower number of registers hence yielding higher performance per dollar.

One of the most critical performance and design hurdles in today's computing challenges is operating on a large volume of data. Large data not only impedes performance by imposing long-latency memory accesses, but also makes the processor design more costly by having the design to overprovision the on-chip memory size to afford the data. In our last study, we proposed another novel register sharing mechanism and also a warp scheduling scheme for GPUs to resolve these issues.Instead of modifying workloads to apply advanced algorithms or changing the GPU architecture significantly, our proposed locality-aware register file (LARF) and locality-aware scheduler (LAS) effectively reduce off-chip memory accesses and enable data sharing across warps in timely manner. We exploited the unique data sharing patterns of big data workloads such as deep learning and matrix multiply algorithms and have the warps opportunistically share data in register file. In our studies, we have observed a lot of cases where the amount of parallelism was limited largely by register shortage. With our proposed LARF, the register usage is also effectively reduced by having warps to share one physical copy of the register.

Contents

List of Figures **xii**

List of Tables **xv**

1 Introduction **1**

2 CORF: Coalescing Operand Register File for GPUs **7**
2.1 Introduction 7
2.2 Background 10
2.3 The Virtues of Register Coalescing 14
2.4 CORF: Coalescing Operands in Register File 17
2.4.1 CORF Overview 18
2.4.2 Generating Compiler-assisted Hints 19
2.4.3 CORF Run-time Operation 20
2.5 CORF++: Re-architected RF 23
2.5.1 Compiler-assisted Register Allocation 23
2.5.2 Coalescing-aware RF Organization 25
2.5.3 CORF++ Run-time Operation 29
2.6 Additional Implementation Details 31
2.7 Performance/Power Evaluation 34
2.8 Hardware/Software Overheads 40
2.9 Related Work 42

3 BOW: Breathing Operand Windows to Exploit Bypassing in GPUs **45**
3.1 Introduction 45
3.2 Background 52
3.3 Motivation 55
3.4 Breathing Operand Windows 59
3.4.1 BOW Architecture Overview 59
3.4.2 BOW-WR: Compiler-guided writeback 63
3.4.3 Reducing the Bypassing Storage Space 70
3.5 Evaluation 73

3.5.1 Performance/Energy Evaluation 74
3.6 Related Work 82

4 RegMutex: Inter-Warp GPU Register Time-Sharing 85
4.1 Introduction 85
4.2 Motivation 88
4.3 RegMutex: Inter-Warp Register Time-Sharing 92
4.3.1 Compiler Support 95
4.3.2 Architecture Support 101
4.3.3 Paired-Warps Specialization 106
4.4 Experimental Evaluation 107
4.4.1 Kernel Occupancy Boost Analysis 109
4.4.2 Register File Size Reduction Analysis 111
4.4.3 Performance Comparison with Related Work 112
4.4.4 Extended Set Size Sensitivity Analysis 115
4.4.5 Paired-Warps Specialization Performance Analysis 116
4.5 Related Work 119

5 LARF: Locality-Aware Register File for GPUs 122
5.1 Introduction 122
5.2 Background and Related Work 126
5.2.1 On-chip Inter-thread Data Sharing 126
5.3 Locality-Aware Register File 127
5.3.1 Perfect Sharing 128
5.3.2 Partial Sharing 130
5.4 Locality-Aware Warp Scheduler 131
5.5 Architectural Modification 135
5.5.1 Dynamic Register Mapping 135
5.5.2 Address Mapping Table and Mapping Controller 136
5.5.3 LAS Support 143
5.6 Evaluation 144
5.6.1 Performance 146
5.6.2 Register File Utilization 151
5.6.3 Synergy With Existing Optimizations 152
5.6.4 Area 153

6 Conclusions 155

Bibliography 158

List of Figures

1.1 On-chip memory components size in NVIDIA GPUs (from 2010–2018). . . . 3

2.1 Baseline GPU register file design with proposed enhancements (in dark purple) for register packing [43, 139]. CU0-CU3 are operand collector units. 12
2.2 Width distribution of registers accessed from RF 14
2.3 Unused RF bandwidth (also proportional to wasted dynamic energy). . . . 15
2.4 Instructions with coalesceable register reads; first fit is clearly weak in promoting coalescing. 16
2.5 CORF overview. Compiler-generated register pairs guide register allocation to create coalescing opportunities 18
2.6 Percentage of successful combinations of compiler identified register pairs for CORF 21
2.7 CORF++ overview. At compile time, we identify which registers should be left-aligning, or right-aligning through graph coloring algorithm, so that we can maximize coalescing opportunities. This information will then guide register allocation in our coalescing-aware register file. 24
2.8 CORF++ register assignment heuristic example 24
2.9 Modified register to bank mapping where all registers belonging to a warp maps to the same bank. 26
2.10 Baseline register sub-bank organization shown in Ⓐ. Sub-bank organization when packing R1 w/ R3, and R2 w/ R4 Ⓑ. Coalescing-aware sub-bank organization Ⓒ enables coalescing across different physical registers with non-overlapping sub-banks. 27
2.11 Dual address register file. 29
2.12 Illustrative Example of CORF++ register allocation (Ⓑ–Ⓓ) and read coalescing (Ⓔ, Ⓕ). 30
2.13 Coalesced instructions: CORF and CORF++ significantly increases the amount of coalescing opportunities. 33
2.14 Reduction in number of accesses to register file. 35
2.15 IPC Improvement. 36
2.16 Reduction in allocated physical registers. 37
2.17 Normalized RF dynamic energy 38

2.18 Normalized RF leakage energy 39
2.19 Static code size increase. 42

3.1 On-chip memory components size in NVIDIA GPUs (from 2010–2018). . . . 46
3.2 Conventional GPU register file architecture (OCU0-OCU31 are Operand Collector Units). 53
3.3 Eliminated read (top) and write (bottom) requests through operand bypassing. 54
3.4 Average time taken by operand collection stage for memory vs. non-memory instructions. 58
3.5 (a) An overview of BOW. BOCX is Bypassing Operand Collector assigned to Warp X; (b) Baseline operand collector unit (left) compared to the proposed wider Bypassing Operand Collector (BOC) unit with forwarding logic support (right). 60
3.6 Code snippet from BTREE application illustrating bypassing operation in BOW. 63
3.7 Distribution of write destinations in BOW-WR 67
3.8 Operand collector units' occupancy. 72
3.9 BOC occupancy with a window 3: half of the entries are unused. 73
3.10 IPC improvement. 76
3.11 IPC increase with 6-entry BOC (half-size). 77
3.12 Cycles spent in OC stage for different window sizes (normalized to the baseline). 78
3.13 Normalized RF dynamic energy 79

4.1 The utilization of a sample thread's allocated register set during kernel execution. X axis shows the number of instructions executed by the thread and Y axis shows the percentage of live registers with respect to allocated registers. Results are extracted using our extension to *GPGPU-Sim* [17]. Applications are from Rodinia [31] and Parboil [124]. 90
4.2 Example of two warps A and B executing identical code with and without RegMutex. Base register set size is 16 registers (per thread) as well as the Shared Register Pool (SRP) size. The architecture is assumed to have 48 hardware registers per thread. 94
4.3 A GPU code sample from DWT2D application and its static register liveness. 97
4.4 The baseline design from *GPGPU-sim* [17] (top) and RegMutex's added storage structures (bottom). Specified sizes are in bits. 103
4.5 Acquire/release procedure implementation in RegMutex. 104
4.6 Architected to physical register mapping design in the Operand Collector Unit. X is the architected register index and Y is the resulted physical register index. 106
4.7 The performance improvement enabled by RegMutex over the baseline. . . 110
4.8 The performance of applications with and without RegMutex on an architecture with half the baseline's register file size. 111
4.9 RegMutex performance comparison with Register File Virtualization (RFV) [62] and the work of Jatala *et al.* [59], which we refer to it as OWF. 113
4.10 The sensitivity of kernel performance to variations in the extended set size with RegMutex. Columns with diagonal stripes are our heuristic's pick. . . 114

4.11 The variations in the theoretical kernel occupancy and the ratio of successful acquires with respect to changes in the extended set size. Columns with diagonal stripes are our heuristic's pick. 116
4.12 The effect of RegMutex's paired-warps specialization on the execution cycle and the occupancy of kernels. 117
4.13 Acquire instruction success rate in RegMutex with and without paired-warps specialization. The results for the 8 leftmost applications are reported on the baseline architecture, and the rest, on the architecture with half of the baseline register file size. 119

5.1 Input data sharing among three neurons that are vertically and horizontally neighboring in a conv layer 123
5.2 Proposed Data Sharing (an example of CNN) 128
5.3 Instruction Sequences of Global Memory Accesses in GPUs 133
5.4 Architectural Modification 136
5.5 Example Scenarios with LARF 139
5.6 Scheduler Impact on CifarNet Execution: Cv: convolution layer, Pl: pooling layer, FC: fully-connected layer 147
5.7 Per-Layer Speedup and Global Memory Accesses of MobileNet: Numbers in parenthesis are the address mapping table entry counts. DWC: depth-wise convolution layer, Cv: 3D convolution layer, PWC: point-wise convolution layer 148
5.8 Per-Layer Speedup and Global Memory Accesses of CifarNet 149
5.9 End-to-End Speedup and Global Memory Accesses of Remaining Workloads 150
5.10 Utilization of Compiler Allocated Registers 151
5.11 Speedup and Register Usage of 128×128 MatrixMul 153

List of Tables

2.1 Summary of CORF, CORF++, and register packing (and register virtualization). All values normalized to the baseline GPU register file. 40
2.2 Renaming table overheads in 40nm technology 41

3.1 Number of write operations to the register file for code snippet shown in Figure 3.6. 66
3.2 Nvidia TITAN X (Pascal Arch.) Configuration 74
3.3 List of used benchmarks 75
3.4 BOC overheads in 28nm technology 81

4.1 Workloads used in experiments. The number of registers per thread and RegMutex's base register set size are shown for each kernel. 109

Chapter 1

Introduction

Graphics Processing Units (GPUs) have emerged as an important computational platform for data-intensive applications in a plethora of application domains. They are commonly integrated in computing platforms at all scales, from mobile devices and embedded systems, to high-performance enterprise-level cloud servers. GPUs use a massively multi-threaded architecture that exploits fine-grained switching between executing groups of threads to hide the latency of data accesses. In order to support this fast context switching at scale, GPUs invest in large Register Files (RF) to allow each thread to maintain its context in hardware. The amount of parallelism available on a GPU (e.g., number of streaming multiprocessors, or SMs) has been steadily increasing as GPUs continue to grow in performance and size, which in turn increases the number of concurrent thread contexts needed to keep these units utilized [15, 61, 72, 73, 100, 103, 133].

The large register file accounts for an increasingly larger fraction of on-chip storage, as shown in Figure 1.1. For example, in NVIDIA Pascal GPU, register file size is 14 MB, which accounts for around 63% of the on-chip storage area. Due to frequent accesses to the RF, it is a crucial microarchitectural component whose architecture substantially impacts the performance and energy-efficiency of GPUs. For example, port conflicts (in register file banks as well as operand collector units that collect the register operands) cause delays in issuing instructions as register values are read in preparation for execution. Number of available physical registers is also a potential factor that could affect overall performance. In addition, the RF has a large energy consumption footprint, since it is the largest SRAM structure that serves a large number of data accesses from the working threads. Earlier studies estimate that the register file is responsible for 18% of the total power consumption on a GPU chip [81], a percentage that has most likely increased as the size of RFs has continued to grow. In this thesis, we first show the existing inefficiencies in register allocation policies in current design. We classified the existing inefficiencies in two classes: *spatial inefficiency* and *temporal inefficiency*. Then, we propose novel synergistic compiler-microarchitecture techniques to resolve those inefficiencies and enable high-performance, energy efficient GPUs.

Spatial inefficiency: In GPUs, registers are allocated in a very conservative way. the allocation of physical registers to architected registers in the kernel binary is static, i.e., the maximum number of live registers at any given point determines the kernel's physical register demand, and is exclusive, i.e., a warp's physical registers are solely its own for the lifetime of the thread-block containing the warp. This allocation scheme carves a portion of the physical registers for the warp regardless of the fluctuations in the register usage by the warp. Also, it is always assumed that all values stored in individual registers during a kernel's lifetime need a fixed number of bits to be represented (32 bits in today's GPUs).

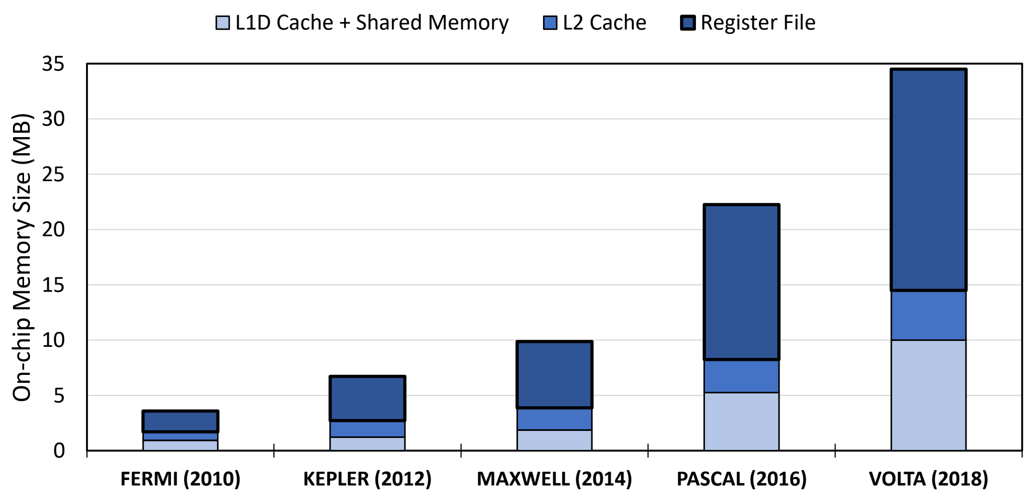


Figure 1.1: On-chip memory components size in NVIDIA GPUs (from 2010–2018).

However, in this thesis, we will show that such conservative choices are not required. CORF, RegMutex, and LARF seek to resolve the spatial inefficiency.

Temporal inefficiency: In our reference architecture, regardless of temporal register reuse opportunities, all operands stage through the register file component, meaning that per each register source operand, one read request is sent to the corresponding register file bank to fetch the data. In BOW, we show that there is a high degree of temporal locality in accesses to the registers across a window of few instructions belong to the same warp.

In chapter 2, we present CORF, a compiler-assisted Coalescing Operand Register File which performs register coalescing by combining reads to multiple registers required by a single instruction, into a single physical read. To enable register coalescing, CORF utilizes register packing to co-locate narrow-width operands in the same physical register. CORF uses compiler hints to identify which register pairs are commonly accessed together. CORF saves dynamic energy by reducing the number of physical register file accesses, and improves performance by combining read operations, as well as by reducing pressure on the register file. To increase the coalescing opportunities, we re-architect the physical register file to allow coalescing reads across different physical registers that reside in mutually exclusive sub-banks; we call this design CORF++. The compiler analysis for register allocation for CORF++ becomes a form of graph coloring called the bipartite edge frustration problem. CORF++ reduces the dynamic energy of the RF by 17%, and improves IPC by 9%.

In chapter 3, we observe that there is a high degree of temporal locality in accesses to the registers: within short instruction windows, the same registers are often accessed repeatedly. We characterize the opportunities to reduce register accesses as a function of the size of the instruction window considered, and establish that there are many recurring reads and updates of the same register operands in most GPU computations. To exploit this opportunity, we propose Breathing Operand Windows (BOW), an enhanced GPU pipeline and operand collector organization that supports bypassing register file accesses and instead passes values directly between instructions within the same window. Our baseline design can only bypass register reads; we introduce an improved design capable of also bypassing unnecessary write operations to the RF. We introduce compiler optimizations to help guide the write-back destination of operands depending on whether they will be reused to further reduce the write traffic. To reduce the storage overhead, we analyze the occupancy of the bypass buffers and discover that we can significantly down size them without losing performance. BOW along with optimizations reduces dynamic energy consumption of the register file by 55% and increases the performance by 11%, with a modest overhead of 12KB increase in the size of the operand collectors (4% of the register file size).

In chapter 4, we propose a software-hardware co-mechanism named RegMutex (Register Mutual Exclusion) to share a subset of physical registers between warps during the GPU kernel execution. With RegMutex, the compiler divides the architected register set into a base register set and an extended register set. While physical registers corresponding to the base register set are statically and exclusively assigned to the warp, the hardware time-shares the remaining physical registers across warps to provision their extended register set. Therefore, the GPU programs can sustain approximately the same performance with the lower number of registers hence yielding higher performance per dollar. For programs that require a large number of registers for execution, RegMutex will enable a higher number of concurrent warps to be resident in the hardware via sharing their register allocations with each other, leading to a higher device occupancy. Since some aspects of register sharing orchestration are being offloaded to the compiler, RegMutex introduces lower hardware complexity compared to existing approaches. Our experiments show that RegMutex improves the register utilization and reduces the number of execution cycles by up to 23% for kernels demanding a high number of registers.

In chapter 5, we propose a new register sharing mechanism and a warp scheduling scheme for GPUs to share physical registers across different warps to reduce the memory traffic. Instead of modifying workloads to apply advanced algorithms or changing the GPU architecture significantly, our proposed locality-aware register file (LARF) and locality-aware scheduler (LAS) effectively reduce off-chip memory accesses and enable data sharing across warps in timely manner. We exploited the unique data sharing patterns of big data workloads such as deep learning and matrix multiply algorithms and have the warps opportunistically share data in register file. Though register file is the largest on-chip memory, due to the excessive usage by state-of-the-art optimizations such as register file blocking, we observed that the parallelism was limited largely by register shortage. With our proposed LARF, the register usage is also effectively reduced by having warps to share one physical copy of register. To the best of our knowledge, this is the first approach that tackles both memory accesses and on-chip memory shortage. Our experimental results on various deep learning and matrix multiply workloads show that the combination of LARF and LAS improves individual layer performance up to 3.5× and end-to-end deep learning performance up to 10%, and reduces the number of global memory loads by up to 80%.

Chapter 2

CORF: Coalescing Operand Register File for GPUs

2.1 Introduction

Over the past decade, GPUs have continued to grow in terms of performance and size. The number of execution units has been steadily increasing, which in turn increases the number of concurrent thread contexts needed to keep these units utilized [72, 73, 78, 97, 100, 103, 115]. In order to support fast context switching between large groups of active threads, GPUs invest in large register files to allow each thread to maintain its context. This design enables fine-grained switching between executing groups of threads, which is necessary to hide the latency of data accesses. For example, the Nvidia Volta GPU has 80 streaming multiprocessors each with a 256KB register file (64K registers, each 32-bit wide) for a total of 20MB of register file space. Due to its continuous access, the register file is a critical structure for sustaining performance. The register file is the largest SRAM structure on the die and one of the most power-hungry components on the GPU. In 2013, it was estimated that the register file is responsible for 18% of the total power consumption on a GPU chip [81], a percentage that is likely to have increased as the size of the RF has continued to grow.

In this chapter, we seek to improve the performance and energy efficiency of GPU register files by introducing *register coalescing*[1]. Similar to memory coalescing where contiguous memory accesses are combined into a single memory request, register coalescing combines multiple register reads from the same instruction into a single physical register read, provided these registers are stored in the same physical register entry. Specifically, register coalescing opportunities are possible when we use register packing [43, 139], where multiple narrow-width registers are stored into the same physical register. In contrast to register packing, which requires one separate read access for each architectural register read, register coalescing allows combining of read operations to multiple architectural registers that are stored together in the same physical register entry. Register coalescing reduces dynamic access energy, improves register file bandwidth, reduces contention for register file and operand collector ports, and therefore improves overall performance.

We propose a Coalescing Operand Register File (CORF) to take advantage of register coalescing opportunities through a combination of compiler-guided register allocation and coalescing-aware register file organization. The key to increasing register coalescing opportunities is to ensure that *related registers*—registers that show up as source operands

[1] "Register coalescing" is analogous to memory coalescing where requests are coalesced [18], and distinct from register coalescing in compiler register allocation which is used to eliminate copy instructions [29,47,107].

in the same instruction—are stored together in the same physical register entry. CORF first identifies exclusive *common pairs* of registers that are most frequently accessed together *within the same instruction.* If both common pair registers are narrow-width and are packed together into the same physical register entry, then accesses to these registers (in the same instruction) can be coalesced. CORF reduce physical register accesses, resulting in ˜8.5% reduction in register file dynamic energy, and ˜4% increase in IPC.

A limitation of CORF is that each register may only be coalesced exclusively with one other register, which limits the opportunities for coalescing registers that are frequently read with several other registers. To further increase register coalescing opportunities, we present CORF++ which presents a re-architected coalescing-aware register file organization that enables coalescing reads from non-overlapping sub-banks across different physical register entries. Thus, reads to any two registers that reside in non-overlapping sub-banks, even if they reside in different physical register entries, can be coalesced. To maximize the opportunities for coalescing, we introduce a compiler-guided run-time register allocation policy which takes advantage of this re-organization. In particular, we show that the compiler must solve a graph coloring variant called the bipartite edge frustration problem to optimize allocation. Since the problem is NP-hard, we use a heuristic to determine how to allocate the registers effectively. CORF++ is able to substantially improve register coalescing opportunities, leading to a reduction in dynamic register file energy by 17% and an IPC improvement of ˜9% over the baseline.

As a secondary contribution, we show that CORF can be combined seamlessly with register file virtualization [62] to further reduce the overall effective register file size, resulting in an overall reduction of over 50%. In particular, both register file packing and register virtualization are orthogonal and combine in benefit, where both utilize indirection using a renaming table, amortizing this common overhead. This reduction in register file size can be leveraged for other optimizations, such as power gating unused registers to save static power [7], or enabling more kernel blocks/threads to be supported using the same register file to improve performance [139].

This chapter makes the following contributions:

- We introduce the idea of register read coalescing, enabling the combination of multiple register reads into a single physical read. CORF implements coalescing with the aid of compiler-guided hints to identify commonly occurring register pairs.
- We propose CORF++, consisting of a re-organized register file to enable coalescing across different physical registers, and a compiler-guided allocation policy that optimizes allocation against this new register file. This new policy relies on compile-time graph coloring analysis, solving the bipartite edge frustration problem.
- We combine CORF++ and register file virtualization, observing that their benefits add up (CORF++ optimizes in space, while virtualization optimizes in time), but their overheads do not (both share a renaming table), resulting in the smallest known effective register file size among register compression proposals.

2.2 Background

In this section, we first overview the organization of modern GPU register files as well as its impact on performance and power. Next, we discuss the concept of register packing [43, 139], a well-known microarchitectural technique, from which register coalescing opportunities arise.

GPU Register File: Modern GPUs consist of a number of Streaming Multiprocessors (SMs), each of which has its own register file, and a number of integer, floating point, and specialized computational cores. A GPU kernel, i.e. program, is decomposed into one or more Cooperative Thread Arrays (CTAs, also known as thread blocks) that are scheduled to the SMs. The threads within a block are grouped together into *warps*, or *wavefronts*, typically of size 32. The threads within a warp execute together in lockstep, following a Single Instruction Multiple Thread (SIMT) programming model. Each warp is assigned to a warp scheduler that issues instructions from its pool of ready warps to the operand collection unit (OC) and then to the GPU computational cores.

Each warp has its own set of dedicated architectural registers indexed by the warp index. There is a one-to-one mapping between architectural registers and physical registers [82]. To provide large bandwidth without the complexity of providing a large number of ports, the register file is constructed with multiple single-ported register banks that operate in parallel. A banked design allows multiple concurrent operations, provided that they target different banks. When multiple operations target registers in the same bank, a *bank conflict* occurs and the operations are serialized.

Figure 2.1 shows our baseline register file organization for the Fermi generation of Nvidia GPUs. It has a register file size of 128 KB per SM split across four banks. A bank is made up of 8 sub-banks that are 128 bits wide each. All 32 registers belonging to the 32 threads in the same warp are statically allocated to consecutive sub-banks (in a single bank)

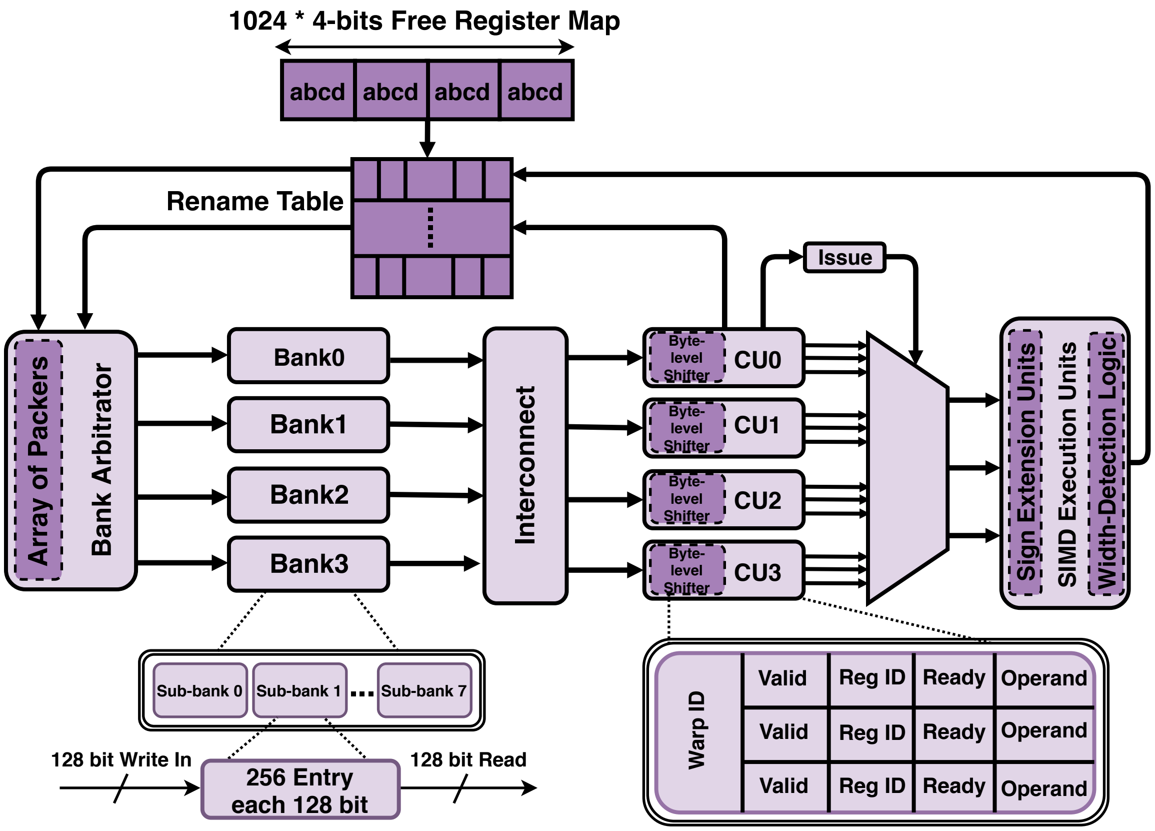


Figure 2.1: Baseline GPU register file design with proposed enhancements (in dark purple) for register packing [43, 139]. CU0-CU3 are operand collector units.

with the same entry index. Thus, a full register for all the threads within a warp can be striped using one entry of one bank, allowing it to be operated on in a single cycle. Each bank can store up to 256 warp-registers.

Impact of Register File on Performance and Power: When a warp instruction is scheduled by the warp scheduler, an operand collector (OC) unit is assigned to collect its operands for execution. An OC fetches the register operands from the register banks they reside in, bound by the two following constraints: (1) *OC port serialization*: Each

OC has only one port and therefore it has to serialize reads when an instruction has multiple operands (instructions may need up to 3 source operands); and (2) *Register bank conflicts*: While operands from different banks may be concurrently read from different OCs, operands that access the same bank cause bank conflicts and cannot be issued together. The port constraints causing these conflicts are difficult to bypass by increasing the number of ports [19]: the cost of a port is extremely high when considering the width of a warp register. Register coalescing can help with both of these constraints: by coalescing operands, it allows multiple operands to be read by an OC in a single cycle, overcoming port serialization. Moreover, by reducing the overall number of register reads, the pressure on the register file is reduced, potentially reducing register bank conflicts. By reducing the overall number of reads to the RF, energy efficiency is improved. Moreover, improving performance leads to shorter run times, also improving energy efficiency.

Register Packing: Register coalescing opportunities arise when two registers needed by the same instruction are stored in the same physical register entry. This opportunity only exists when we allow multiple registers to be packed in the same physical register entry, a known architectural technique called *register packing* [43, 139]. In particular, register packing maps *narrow-width values* (values which do not need all 32 bits to be represented) of multiple architectural registers to a single physical register.

Since each architectural register read in prior register packing implementations requires a separate uncoalesced physical register read, a greedy *first-fit allocation policy* has been utilized to pack registers. This simple policy is sufficient to achieve the main goal of register packing, which is reducing the effective register file size; enabling unused registers to be power gated, or enabling the register file to be provisioned with a smaller number of physical registers. However, as we will show in the next section, this policy leads to very few register coalescing opportunities. Thus, a key to register coalescing is to pack related registers that are frequently read together, which is the goal of our compiler analysis.

2.3 The Virtues of Register Coalescing

In this section, we motivate register coalescing, and the need to design coalescing-aware register files to maximize the benefits of register coalescing. All experiments are collected with the GPGPU-Sim simulator [17], modeling a Fermi GPU[2]. We utilize benchmarks from Rodinia [31], Parboil [125], NVIDIA CUDA SDK [101], and Tango DNN Benchmark Suite [68]. More details of experimental setup are discussed in Section 2.7.

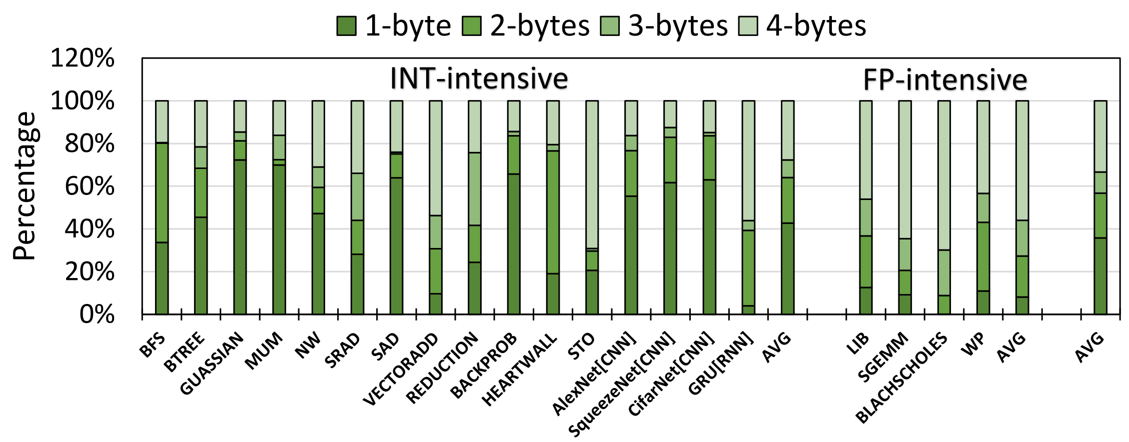


Figure 2.2: Width distribution of registers accessed from RF

Register operand characteristics: Figure 2.2 showcases the prominence of narrow-width values in GPU applications. We classify narrow-width values into four size classes: 1 byte, 2

[2]Register coalescing opportunities are agnostic to hardware architecture.

bytes, 3 bytes, and 4 bytes (full-width). On average, 65% of all register operations contain narrow-width values, with over 33% of operations consuming no more than a single byte. This demonstrates that there exists a significant amount of register operands amenable to register coalescing. For floating point (FP)-intensive benchmarks (such as *sgemm* and *blackscholes*), the percentage of narrow-width values is less than that for integer-intensive benchmarks (such as *bfs* and *btree*). This is due to the IEEE 754 encoding of floating point values, which makes use of all 32 bits.

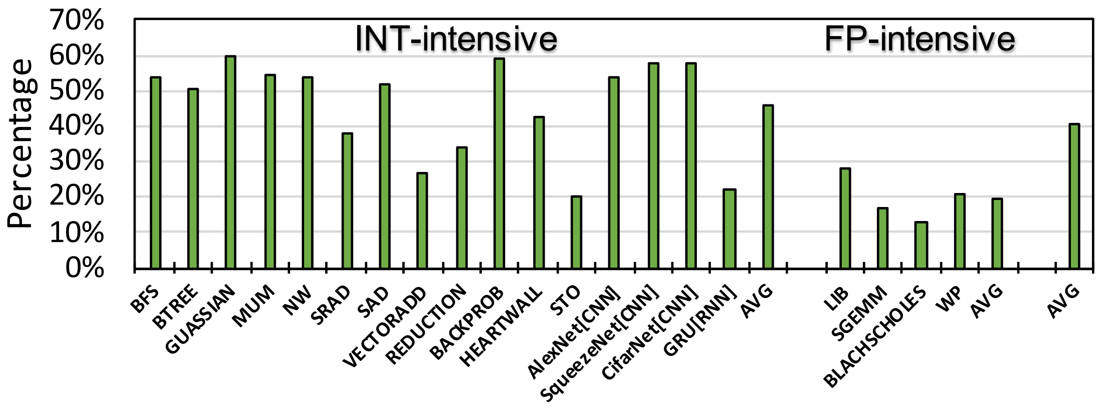


Figure 2.3: Unused RF bandwidth (also proportional to wasted dynamic energy).

Opportunity– Register file bandwidth: Figure 2.3 shows the unused register file bandwidth due to carrying the unneeded bits of narrow-width values. In addition to wasting bandwidth, these unneeded bits also cause wasted dynamic energy, as they are unnecessarily carried through to the operand collector. We observe more wasted bandwidth in integer applications, since narrow-width values are more common in them than in floating point applications.

While register packing is able to reduce the effective size of the register file, each register read still requires a separate physical register read. Therefore, this wasted bandwidth is not recovered with simple register packing. To this end, our proposed register coalescing aims to read multiple related registers used by the same instruction through a single register read operation in order to utilize the register file bandwidth more efficiently.

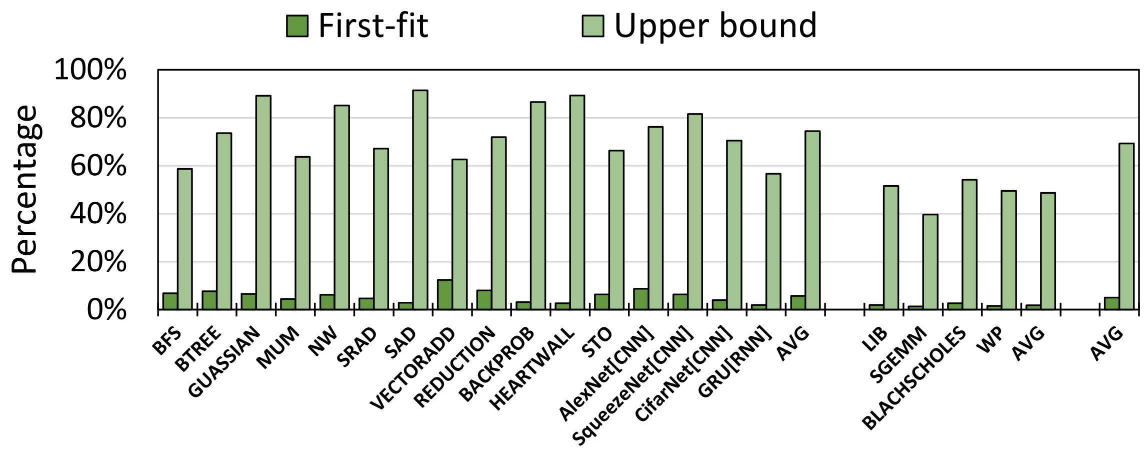


Figure 2.4: Instructions with coalesceable register reads; first fit is clearly weak in promoting coalescing.

Register coalescing opportunity: Figure 2.4 shows the prevalence of operand coalescing opportunities. We profile the register operand values at run-time and measure the fraction of all dynamic instructions which contains two register source operands that are both narrow and can fit together in a single register entry. We consider instructions that have two or more register source operands because they could benefit from coalescing. We find that around 40% of the instructions have two or more register source operands, but more importantly, because they read multiple registers, they are responsible for over 70% of the register file reads. On average, 69% of all dynamic instructions with two or more operands have the potential for operand coalescing, because their register operands can be packed, with up to 91% in some benchmarks like *Sad* and *Gaussian*. Clearly, we have more coalescing opportunities in integer intensive applications compared to floating point.

If we extend register packing to enable coalescing but keep the greedy first-fit register allocation policy, we can only leverage register coalescing opportunities in around 4% of instructions with two or more operands. This is a tiny fraction of the 69% of such instructions where a coalescing opportunity is potentially available! To improve coalescing opportunities, CORF incorporates a compiler-guided register allocation policy to identify pairs of registers commonly read from the same instruction and map them into the same physical register entry. In addition, we propose a coalescing-aware register file sub-bank organization and associated compiler-guided allocation policy (CORF++) which can coalesce register operands that are *not* stored in the same physical register entry, but in non-overlapping byte slices in the sub-bank.

2.4 CORF: Coalescing Operands in Register File

In this section, we present the design of CORF, which coalesces register reads to improve the RF performance. For two reads to be coalesceable, they have to be destined to registers that are packed in the same physical register entry. To improve the opportunity for coalescing, CORF utilizes compiler-assisted hints to pack related registers together. CORF is the first register file optimization technique that simultaneously improves performance and reduces power (both leakage and dynamic power). Coalescing enables higher performance by combining read operations, reducing port serialization of operand collector units and register file port conflicts. Coalescing reduces dynamic power, by decreasing the number of read operations to the register file, and lowers the overall GPU energy consumption because it leads to overall performance improvement that enable programs to finish faster. In Section 2.5, we will present CORF++, which further re-architects the register file organization to create more coalescing opportunities.

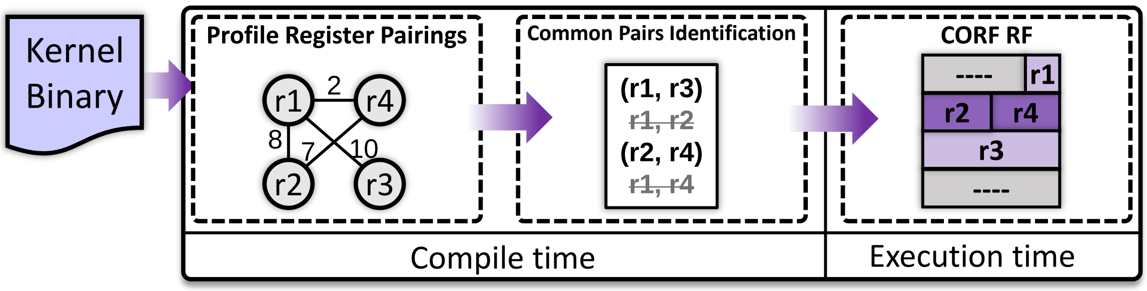


Figure 2.5: CORF overview. Compiler-generated register pairs guide register allocation to create coalescing opportunities

2.4.1 CORF Overview

CORF identifies *register pairs*—registers that are used as source operands in the same instruction—at compile time through static analysis or, alternatively, profiling. For example, in Figure 2.5, we have four registers (`r1`, `r2`, `r3`, `r4`), where register `r1` is read 8 times with `r2`, 10 times with `r3`, and 2 times with `r4`. In this example, we select (`r1`, `r3`) and (`r2`, `r4`) as target exclusive common pairs for coalescing. During run-time, if any of these common pairs happen to be compatible narrow-width values, they will be dynamically packed together. If any instruction requires both `r2` and `r4` as source operands, we can coalesce the operand access using a single read of the register file. However, in this example, during run-time (`r1`, `r3`) could not be packed since their combined size exceeds the size of a physical register entry. Since each register can only be coalesced with at most one other register, we lose opportunities to coalesce operands from instructions with different register pairings, such as (`r1`, `r2`), a limitation which we will target in Section 2.5.

2.4.2 Generating Compiler-assisted Hints

Identifying exclusive common pairs: The first step in identifying common pairs is to profile the frequency of register pairings in order to build a *Register Affinity Graph*, as shown in Figure 2.5. In order to determine the edge weights, we task the compiler to estimate the dynamic frequency of occurrence for each instruction in each kernel. This is, in general, a difficult problem at compile time, which we approximate as follows. For each instruction outside of a loop with two or more operands, we consider every pair of operands to occur once. Inside of loops, if the loop iteration count is statically resolvable, we use that count to increment the edge weight for register pairs that occur in the loop. If the iteration count is not a resolvable constant, we give a fixed weight to each register pair in instructions inside the loop. We use the same approach for nested loops. While these weights are not exact, they serve as a heuristic to assign relative importance to register pairs.

In order to identify exclusive common pairs, we must remove edges of the registers that have more than one edge. Considering only registers with more than one edge, we repeatedly remove the edge with the least weight until we end up with only exclusive pairs of registers. If there are any pair of registers that have all of their edges removed, we check if an edge can be restored between them.

Passing compiler-assisted hints to hardware: The set of exclusive register pairs that are identified by the compiler are annotated in the executable's preamble of a kernel and delivered to the hardware through a metadata instruction. The register pair information is maintained in a small associative structure. Specifically, we use a 64-bit metadata instruction (to be aligned with existing SASS binaries) in the beginning of each kernel in order to carry the compiler hints to the hardware. Consistent with the SASS instruction set that uses 10 bits as opcode for each instruction, we reserved 10 bits as opcode and the remaining bits for storing the common pairs of the registers. Since in Fermi architecture, each thread may have up to 63 registers, we need 6 bits as the register number. Each metadata instruction can carry up to four common pairs. Multiple instructions are used if more than 4 pairs need to be communicated. This design can also be adapted to support newer GPUs with more registers.

2.4.3 CORF Run-time Operation

We complete the description of CORF by explaining how registers are allocated to control the allocation of compiler identified pairs. We will also describe how coalescing opportunities are identified.

CORF register allocation policy: The register allocation policy for CORF attempts to pack the identified register pairs into the same physical register entry to increase coalescing opportunities. A register is allocated for the first time it appears as the destination of an instruction. Additionally, it could be reallocated when its size changes. When an allocation event occurs, we check the register pair information to see if the register belongs to a common pair. If it is, the allocator uses the common pair allocation logic. If the register does not belong to a common pair, it is allocated using the default allocation policy (assumed to be first-fit). We illustrate the common pair allocation using an example. Assume that `r1` and `r2` are identified as a common pair. When the first operand (say `r1`) arrives and is to be allocated, it is identified as a common pair register and mapped to any free full-width physical register. The rationale is to reserve any remaining slices of the physical register for a future allocation of the other register in the pair. When the buddy register (the register complementing the pair, which is `r2` in this example) is allocated, we check to see if it fits the physical availability in the register allocated to `r1`. If it fits, it is allocated to the same physical register. Otherwise, it is mapped using the default policy.

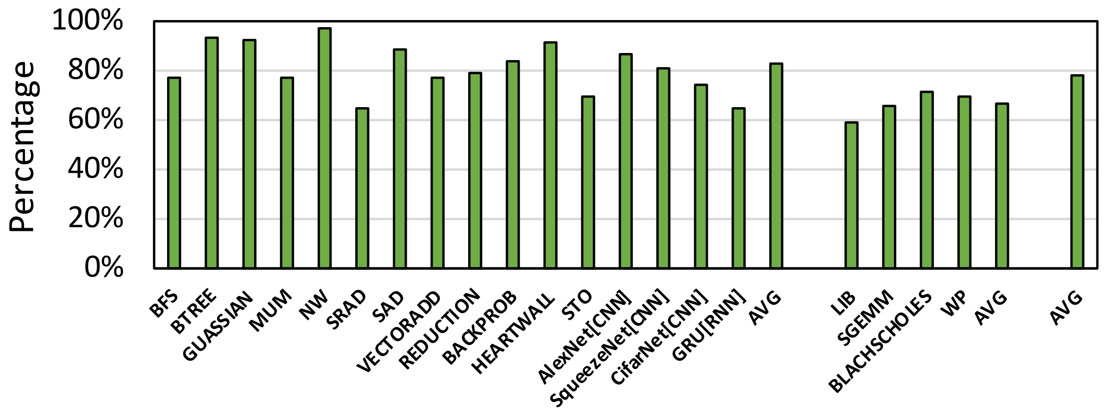


Figure 2.6: Percentage of successful combinations of compiler identified register pairs for CORF

In Figure 2.6, we show that identified common pairs fit together, and are successfully packed in the same register in most of the cases (an average of just under 80%). This is a high percentage despite the fact that we currently carry out no size estimation in the compiler analysis.

Identifying coalescing opportunities: Recall that packing registers in the same physical register is enabled by a *renaming table* (RT) that maps the architectural register to the physical register slice where it is stored. The RT is indexed by a tuple of the warp ID and an architectural register number. Each physical register is split into four 1-byte slices. Thus, each RT entry stores the physical register where this value is stored, and a 4-bit vector called the *allocation mask*, which specifies the bytes in the physical register that the potentially narrow architectural register resides in. We use a *free register map* to keep track of free allocations of physical register slices when making allocation decisions. The free register map is a bit-vector where each bit represents a byte of a physical register (i.e., 4 bits per physical register).

To identify coalescing opportunities as a new instruction is sent to an operand collector unit, we first look it up in the renaming table to determine the physical registers where the operand registers are stored. If the physical registers for two operands match, the reads to these operands are coalesced into a single read to the register file. When the physical register contents are received, the unpacker demultiplexes the two registers and sign-extends them to recover two full-length registers.

Incorporating register virtualization [62]: CORF's implementation seamlessly supports register file virtualization to further reduce the size of the register file. Specifically, we observed that register file virtualization, which releases registers when they are no longer live, can also further reduce the register file size. At the same time, register file virtualization can be directly supported within CORF since it also relies on a renaming table to allocate registers, requiring almost no additional overhead.

2.5 CORF++: Re-architected RF

CORF coalescing opportunities are limited to registers stored within the *same physical register entry.* If a register is commonly accessed with two or more other registers, coalescing is possible with only one of them. To relax this limitation, CORF++ re-organizes the register file to enable more operand coalescing opportunities.

Specifically, CORF++ (Figure 2.7) re-architects the register file to enable coalescing of registers within the *same physical register bank*, provided they reside in *non-overlapping sub-banks.* Recall that each bank consists of eight sub-banks of 16 bytes wide. Since we are no longer restricted to coalescing exclusive pairs of registers packed into the same physical register entry, the compiler's task of guiding register allocation to promote coalescing becomes substantially different. In this section, we overview CORF++. We first present the compiler support to optimize coalescing opportunities in CORF++, then describe the implementation of the coalescing aware register file, and finally discuss its operation during run-time.

2.5.1 Compiler-assisted Register Allocation

CORF++ allows coalescing registers in non-overlapping sub-banks, even if the values reside in two different physical register entries. The main challenge of efficient register allocation in CORF++ is in assigning commonly read register pairs in different sub-banks. We simplify the allocation to a selection of left-aligning and right-aligning assignments; provided that two registers are in separate alignments, they have a chance of being coalesced (subject to their combined size being smaller or equal to 4 bytes).

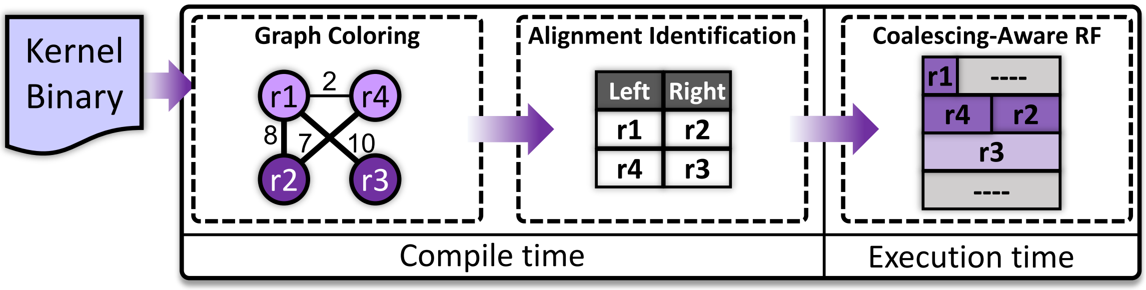


Figure 2.7: CORF++ overview. At compile time, we identify which registers should be left-aligning, or right-aligning through graph coloring algorithm, so that we can maximize coalescing opportunities. This information will then guide register allocation in our coalescing-aware register file.

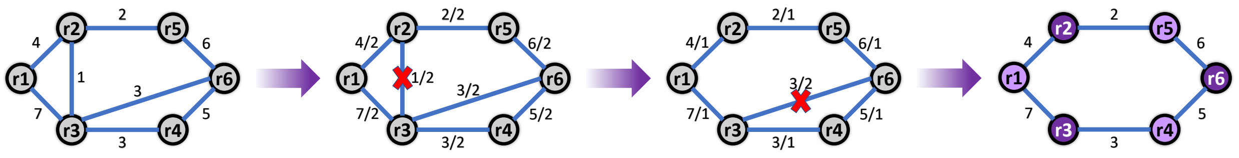


Figure 2.8: CORF++ register assignment heuristic example

Similar to the compiler analysis for CORF, we start by constructing the Register Affinity Graph where edges between registers indicate the expected frequency of reading the two registers together in the same instruction. An optimal assignment maximizes the weight of the edges between registers assigned to alternate alignments. This problem maps to a graph coloring problem variation (where each alignment is a color). We are attempting to remove the minimum edge weight (thus, forsaking the least coalescing opportunities) to enable the graph to be colorable by two colors (left or right). This variation of graph coloring is called the bipartite edge frustration problem, and is NP-hard even with two colors [145].

To derive an efficient heuristic for register mapping, we first observe that any graph with no odd cycles (cycles made up of an odd number of edges) is 2-colorable. Thus, to solve the problem, we should remove the minimum set of edges, considering weight, that will break all odd cycles (to identify odd cycles, we used a modified version of the algorithm in [28]). Since the optimal solution is NP-hard, we develop the following heuristic, as illustrated in Figure 2.8. In the initial graph state (left-most graph), we have four odd cycles: (`r1,r2,r3`), (`r3,r4,r6`), (`r2,r3,r4,r6,r5`), and (`r1,r3,r6,r5,r2`). We assign each edge a weight corresponding to its original weight, divided by the number of odd cycles that removing it would break. We then remove the edge with the minimum weight (among the edges that are part of odd cycles), and update the weights. We repeat this process until all odd cycles are eliminated, enabling us to trivially 2-color the graph.

Similar to CORF, the register allocation information is passed through metadata instructions. We use a metadata instruction to encode the register assignments to either left-aligning, right-aligning, or don't-care. This encoded data is expanded to store 2 bits per register to indicate alignment. This data is stored using a single bit-vector for each kernel, resulting in a storage overhead of 128 bits per kernel. Other designs that reduce or completely remove this overhead are possible, for example, having the compiler preset the register alignments (e.g. all even registers right aligned).

2.5.2 Coalescing-aware RF Organization

Mapping registers to banks: In the baseline register file, registers belonging to the same warp are interleaved across the register banks with the goal of minimizing bank conflicts across warps (Figure 2.9, left side). Since coalescing occurs only within a single instruction of a warp, CORF++ maps all registers belonging to the same warp to a single register bank in order to maximize coalescing opportunities (Figure 2.9, right side). This new mapping ensures that all accesses to registers within the same warp are in the same bank and therefore potentially coalesceable.

Counter-intuitively, our goal is to create more bank conflicts within warps, which gives us more opportunities to convert bank conflicts into beneficial coalescing opportunities. Note that since the operand collector unit can read no more than one register in each cycle, there is no lost opportunity in terms of reading registers from different banks for the same instruction. With respect to conflicts across warps, on average, the new mapping does not increase conflicts, since the probability of two registers from two different warps being in the same bank remains $\frac{1}{n}$, where n is the number of banks. However, with the new mapping, two warps either always conflict (because they are mapped to the same bank) or they never do (mapped to different banks) and there is a possibility for pathologies arising, for example, from two active warps being mapped to the same bank. However, we did not observe any such behavior in our experiments.

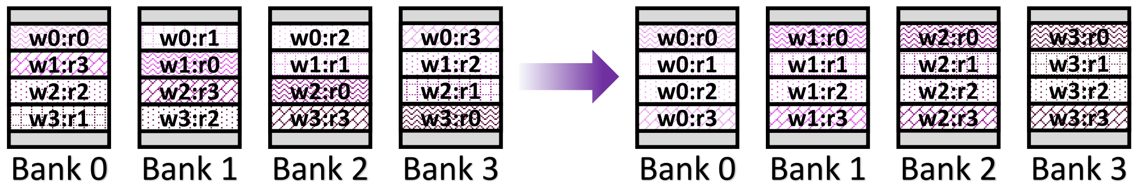


Figure 2.9: Modified register to bank mapping where all registers belonging to a warp maps to the same bank.

Sub-bank organization: CORF++ allows multiple read operations to registers that reside in non-overlapping sub-banks to be coalesced. To support this functionality, we change the mapping of the registers to sub-banks. For clarity, we denote the bytes of a 32-bit register values as $B_3B_2B_1B_0$.

In Figure 2.10 **A** we show how registers are organized across the 8 sub-banks in current GPUs. A register is stored across all 8 sub-banks, where each sub-bank is 128 bits wide. Each sub-bank stores a 32-bit register value for 4 threads. For example, sub-bank 0 stores the register values for threads 0 - 3 in sequential order, where the first 4 contiguous bytes are from thread 0, the next 4 bytes are from thread 1, and so on.

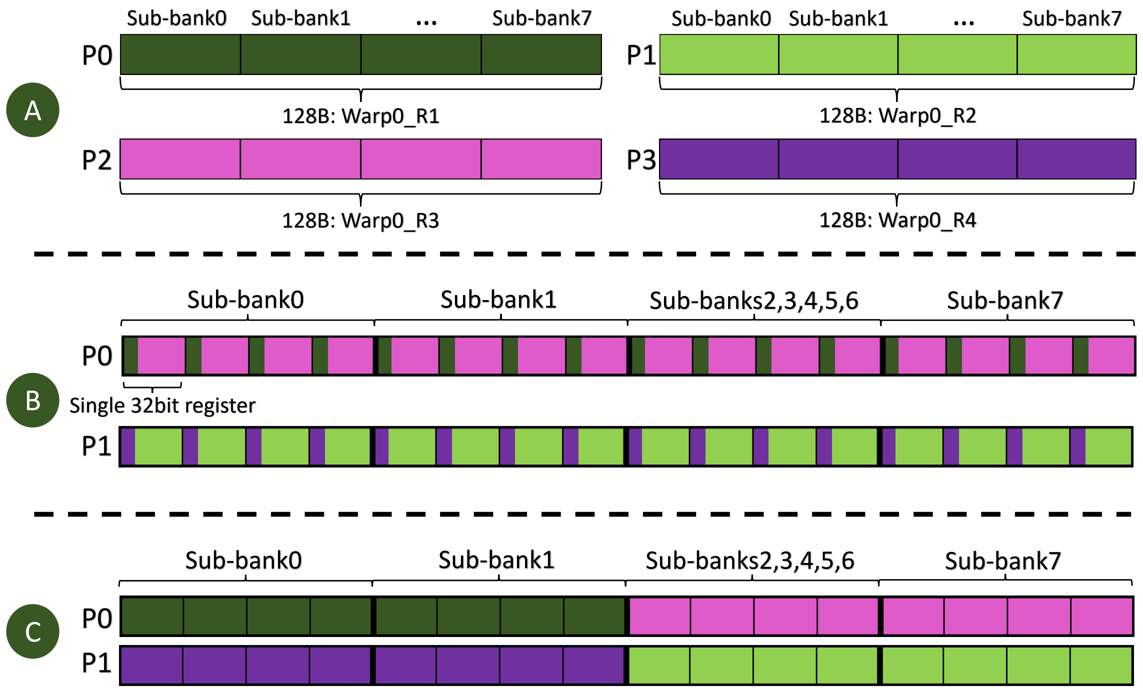


Figure 2.10: Baseline register sub-bank organization shown in A. Sub-bank organization when packing R1 w/ R3, and R2 w/ R4 B. Coalescing-aware sub-bank organization C enables coalescing across different physical registers with non-overlapping sub-banks.

Now let us assume that `r1` and `r4` are 1-byte narrow values, and `r2` and `r3` are 3-byte narrow values. Figure 2.10 Ⓑ shows how these four architectural registers are stored after they are packed into two physical registers. For example, in physical register $P0$, `r1` and `r3` are packed together. In this example, since `r3` is 3-bytes, `r3` will only utilize the 3 least significant bytes (B_{2-0}). This mapping leaves the most significant byte (B_3) available, which is packed with `r1`. `r2` and `r4` are also packed similarly. In this scenario, we can only coalesce reads if they require `r1` and `r3`, or `r2` and `r4`, as these pairs reside in the same physical register entry. Here we lose coalescing opportunities for other compatible pairs, such as `r1` and `r2`, or `r3` and `r4` since parts of every register are spread across all sub-banks.

To address this limitation, we present a re-organized sub-bank mapping, as shown in Figure 2.10 Ⓒ. Instead of storing registers in sequential ordering of the entire 32-bit register value, we will instead interleave the storage of register values across the sub-banks. In this scenario, we first store the most significant bytes (B_3) of threads 0 - 31 consecutively, then store the next significant bytes (B_2) of threads 0 - 31, etc. In this organization, B_3 is stored in sub-banks 0 and 1, B_2 is stored in sub-banks 2 and 3, and so on.

When storing packed values in CORF++, we store the narrow registers as either *left-aligning*, or *right-aligning*. In the case of `r1` and `r3`, `r1` is stored into $P0$ as left-aligning, and `r3` is stored as right-aligning. In this new sub-bank organization, we are able to coalesce `r1` and `r3`, and `r2` and `r4`. Note that if each sub-bank can address different physical register addresses, then it would also be possible to coalesce registers in non-overlapping sub-banks. For example, `r1` and `r2`, as well as `r3` and `r4` would be coalesceable.

Dual-addressable banks: To support coalescing across different physical register entries, we introduce dual-addressable banks (Figure 2.11). We add additional MUXes to pick between $Address1$ and $Address2$, which represent a left-aligning and a right-aligning register being coalesced. If we wish to coalesce `r1` and `r2`, then $P1$ would be sent to $Address1$, and $P0$ to $Address2$. By default, the MUXes select $Address1$, and utilize the 4-bit allocation mask from $Address2$'s entry in the renaming table as the selector. In this scenario, we use `r1`'s allocation mask, which would be 1000.

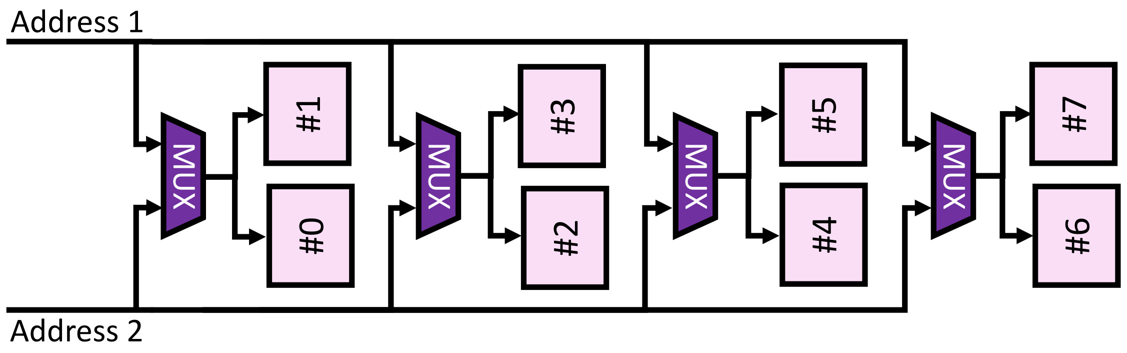


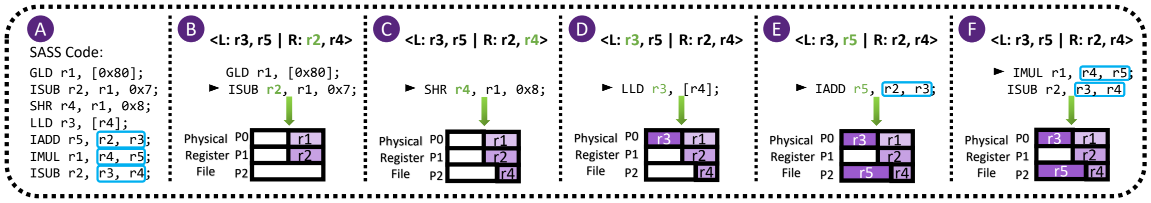
Figure 2.11: Dual address register file.

2.5.3 CORF++ Run-time Operation

Next, we explain the run-time operation of CORF++ through an illustrative example to demonstrate register allocation and coalescing.

CORF++ register allocation: When an allocation event occurs (e.g., writing into `r2` in Figure 2.12 Ⓑ), we check the register alignment to see if it is a right-aligned or left-aligned register. For don't-care registers, we default to the first-fit allocation.

Identifying coalescing opportunities: Similar to CORF, to identify coalescing opportunities as a new instruction is sent to an operand collector unit, we look up the allocation



```
GLD r1, [0x80];
ISUB r2, r1, 0x7;
SHR r4, r1, 0x8;
LLD r3, [r4];
IADD r5, r2, r3;
IMUL r1, r4, r5;
ISUB r2, r3, r4;
```

Figure 2.12: Illustrative Example of CORF++ register allocation (Ⓑ–Ⓓ) and read coalescing (Ⓔ, Ⓕ).

mask in the renaming table for the source operands. Any two source operands could be coalesced if the `AND` of their allocation masks becomes 0000.

Figure 2.12 shows an illustrative example of CORF++ with three physical registers. Ⓐ shows a piece of SASS code. The value loaded in `r1` in Ⓑ is detected by a *width detection unit* as a narrow-width value that needs 2 bytes, and since `r1` is an unallocated don't-care register, we map it to the first available spot (using first-fit policy). The next instruction writes into `r2` which is right-aligned, so we map it to the first available *right* part of a physical register. In Ⓒ, the instruction writes into `r4` and is allocated to the first available *right* part of a physical register. Ⓓ shows a local load into `r3`, so we map it to the first available *left* spot (which is *P0*). In Ⓔ, we first coalesce the read operation for `r2` and `r3` and then write into `r5`, so the allocator maps it to the first available *left* spot. Finally, in Ⓕ, CORF++ coalesces the read operations for `r4` and `r5` and later `r3` and `r4`. In this example, we were able to coalesce all available opportunities. In contrast, CORF is not able to coalesce read operations for `r3` and `r4` because we can only pick *exclusive common pairs.*

2.6 Additional Implementation Details

CORF assumes as a starting point a register file that implements register packing RF [43, 139] and extends it in three important ways: (1) It supports operand coalescing: the ability to identify opportunities for reading registers that are packed in the same physical register (CORF) or in mutually exclusive sub-banks (CORF++), and the support to read them together and unpack them; (2) It receives compiler hints to guide register allocation decisions and uses them to guide allocation to promote coalescing; and (3) It also supports register virtualization [62], allowing it to free registers when they cease to be live. Additionally, CORF++ rearchitects the register file to enable coalescing reads from mutually exclusive sub-banks as we described in the previous section. In this section, we describe additional important components of CORF and CORF++.

Renaming Table (RT): The renaming table is a table indexed by a tuple of the warp ID and an architectural register number. Each entry stores the physical register where this value is stored, and a 4-bit allocation mask. The table consists of ($max\_num\_of\_warps\_per\_SM \times max\_regs\_per\_thread$) entry, which is $48 \times 63 = 3024$ in our reference register file. Each entry has a width of 14 bits (10 bits to represent the physical register number, and the 4-bit allocation mask).

The renaming table needs to be accessed on register reads to resolve the mapping to the physical register. The number of ports needed must at least match the number of read ports on the register file to keep port conflicts from becoming a bottleneck. The renaming table can be implemented as a general multi-ported table. However, to reduce complexity, we implement it as a dual-ported sub-banked structure. We use two ports to allow fast lookup of potentially coalesceable registers. We use a design with a separate bank for each register file bank in the corresponding register file.

Allocation Unit: A small structure that guides the allocation policy using information provided by the compiler. We designed and synthesized this structure in detail for CORF++. It holds an allocation vector that carries the alignment for each register (left, right or don't-care). We store 128 bits per each kernel, for a maximum storage size of 128 bytes per SM (please note that we may have up to 8 concurrent kernels running on each SM). The allocation vector is consulted during allocation in conjunction with a free map that keeps track of the available physical registers (and register slices). The allocator logic uses the alignment preference as it consults the free map to identify a target register for allocation. Note that the renaming logic, free map, and the allocation logic are present in baseline register packing [43, 139]; our allocation unit adds the compiler hints and changes the allocation logic to use them.

Impact on pipeline: Although the RT access latency is low (0.38ns according to CACTI [121], which is well below the cycle time of modern GPUs), we want to avoid combining the RT lookup, coalescing logic, and the register file read in the same cycle. We note that once the scoreboard marks an instruction to be ready to issue, we need at least one cycle to find a free operand collector and move the instruction to it. Thus, we use this cycle to initiate access to the renaming table to avoid trying to fit the renaming table access and the register file access in the same cycle. The RT is dual-ported and sub-banked; however, in the event of a port conflict, the arbitrator (which resolves conflicts for the register file) is extended to delay the register read while the renaming table read is resolved. We extended the pipeline

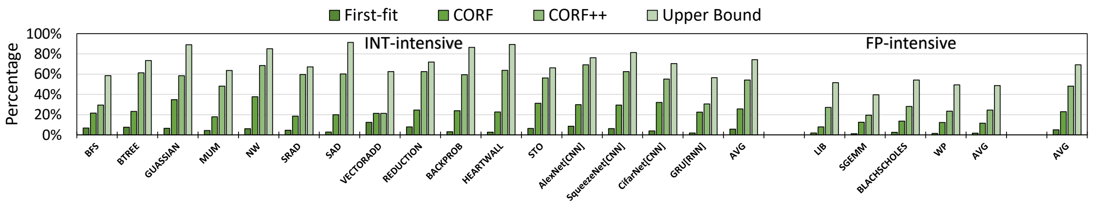


Figure 2.13: Coalesced instructions: CORF and CORF++ significantly increases the amount of coalescing opportunities.

in the simulator to model these effects.

Control divergence: When control divergence occurs, only a subset of SIMT lanes of a warp are active. CORF operation continues unchanged under divergence but considering all registers (whether belonging to active or inactive threads) for all operations (importantly for width determination).

Size changes: If a packed narrow-value register size increases during runtime, we reassign it to another physical register entry using the same process as the initial assignment. The original mapping is then cleared. Size change events which require reallocation are rare (less than 0.3% of writes), which makes these extra accesses to the RT have negligible effects. In case of a size decrease, we keep the old mapping and adjust only the size in the renaming table.

Packers and unpackers: Packers and unpackers are placed as shown in Figure 2.1 so that packed values only exist in the register file and operand collection pipeline stage. Registers are packed as they are written to the register file by first aligning them into the slice they will be written to, and writing only that slice of the physical register. Conversely, when registers are read, they are unpacked by shifting down (if necessary) and sign-extending such that the registers are recovered to full width. Our unpackers are designed to be able to unpack two values in the case of coalesced reads. The number of packers required matches the pipeline width for writing (in our case, two packers). To unpack coalesced registers, we have two unpackers working in parallel in each operand collector, for a total of 8 unpackers per SM.

Width detection units: The register width detection units are embedded into the final stage of SIMD execution units in order to detect the width of produced outputs. This is a combinational circuit: it ORs the 7 least significant bits for each of the three most significant bytes for every register in addition to the most significant bit of the byte before it (to ensure that narrow positive numbers always start with a 0 in the MSB). For example, for byte 1 which spans bits 8 to 15, we OR together bits 7 to 14 to identify whether the byte is 0 or not. This produces a 3-bit output for each register. Moreover, another 3 bits are obtained by NAND-ing together the same bits of each byte to track the width of negative numbers. Again, this ensures that any shortened negative number has 1 in the MSB. We use the most significant bit of the register to multiplex out either the OR outputs (for positive values) or the NAND outputs (for negative values). A second stage ORs the 3 bits output of the MUX per register across all 32 registers in the warp producing a single 3-bit output to capture the maximum width. This 3-bit sequence is used to determine the overall size of the register.

2.7 Performance/Power Evaluation

We have implemented CORF and CORF++ in GPGPU-Sim v3.2.1 [17], based on an Nvidia Fermi-like GPU configuration with 15 SMs. Each SM has a 128 KB register file organized into four banks, and each bank consists of eight sub-banks, as detailed in Figure 2.1. We enabled PTXPlus for all of our evaluations. Since GPGPU-Sim provides a detailed PTX code parser, we modified the parser to carry out our compiler optimizations. Each SM also has two warp schedulers configured to use a two-level warp scheduler.

In all experiments, we use 20 benchmarks selected from Rodinia [31], Parboil [125], NVIDIA CUDA SDK [101], and Tango [68] benchmark suites. The benchmarks cover a range of behaviors and operand mixes (integer/floating point).

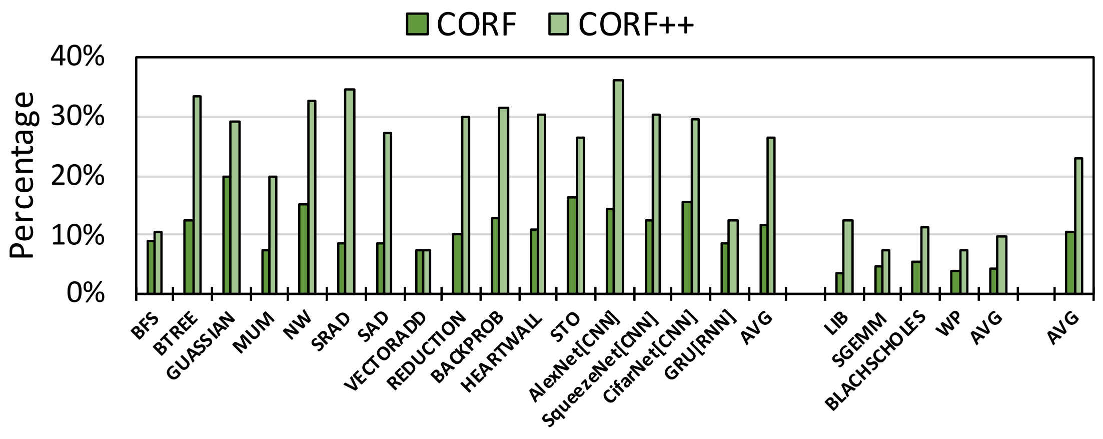


Figure 2.14: Reduction in number of accesses to register file.

Coalescing success: Figure 2.14 shows the reduction in register file accesses due to operand coalescing in CORF and CORF++. CORF reduces the overall number of register file accesses, by 12% for integer applications, 4.5% for floating point applications, and 10% of all applications. This reduction percentage is computed against all accesses (including writes, and instructions with a single register operand, which cannot be coalesced). CORF++ is able to reduce even more accesses (by 2.3x) because of increased coalescing opportunities. Specifically, CORF++ reduces register access of integer applications by 27%, floating point applications by 9.9%, and 23% overall. Figure 2.13 shows the impact of compiler optimizations on the success of coalescing. While first-fit allocation policy results in coalescing only 4% of the instructions with multiple register operands, CORF and CORF++ are able to coalesce 23% and 48%, respectively.

Performance: As a result of the reduced register accesses, performance is improved. Figure 2.15 shows the performance impact of CORF and CORF++. Notably, we observe IPC improvement across all benchmarks. On average, CORF improves IPC by 4.9% for integer benchmarks and 1.7% for floating point benchmarks (harmonic mean across all applications is 4%). For fairness, the IPC computation does not count metadata instructions since they do not further the computation (but we include their cost). CORF++ is able to improve IPC for integer benchmarks by 10.5%, floating point ones by 3.6%, resulting in a harmonic mean of 9%.

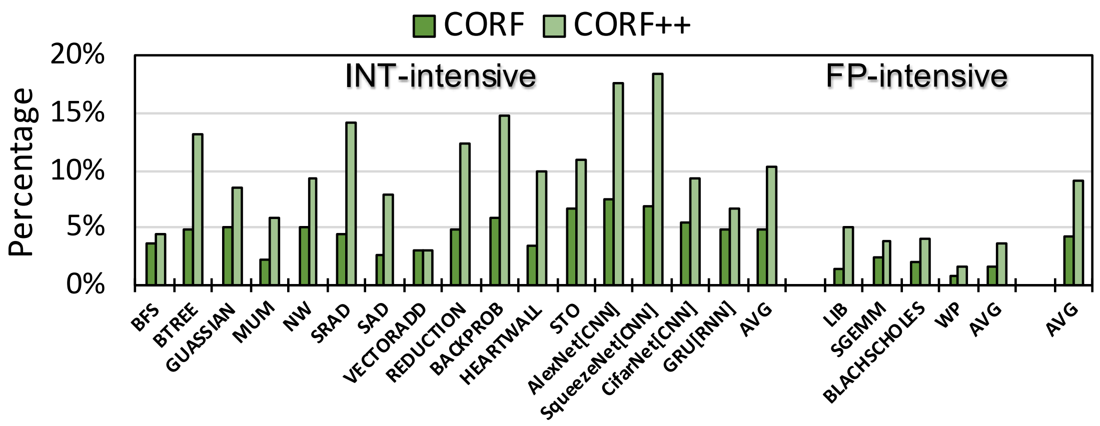


Figure 2.15: IPC Improvement.

Register file size: A secondary contribution of CORF is that we combine register packing and register virtualization to reduce the overall register file size beyond either of these techniques alone. Virtualization is essentially obtained for free since it primarily relies on a renaming table such as the one we already use. Figure 2.16 shows the reduction in the number of allocated physical registers using register packing, register file virtualization (RF-Virtualization) [62], and when combined together. We tracked the number of allocated physical registers (each potentially packing several architectural registers) as a fraction of the total number of architectural registers averaged over the benchmarks' execution. Register packing reduced physical-register allocation by 34%, register file virtualization alone reduced it by 35%, while both together reduced it by 54%. When combined, packing compresses spatially, and RF-Virtualization temporally, leading to synergistic improvements [20, 21]. This is the highest compression ratio achieved by techniques that attempt to compress the register file size [62, 80, 139]. The reduction in effective register file size can be exploited either: (1) by gating unused registers to save power; (2) by reducing the register file size while maintaining performance; or (3) by enabling more threads to be active to improve performance. We demonstrate the advantage using the first option.

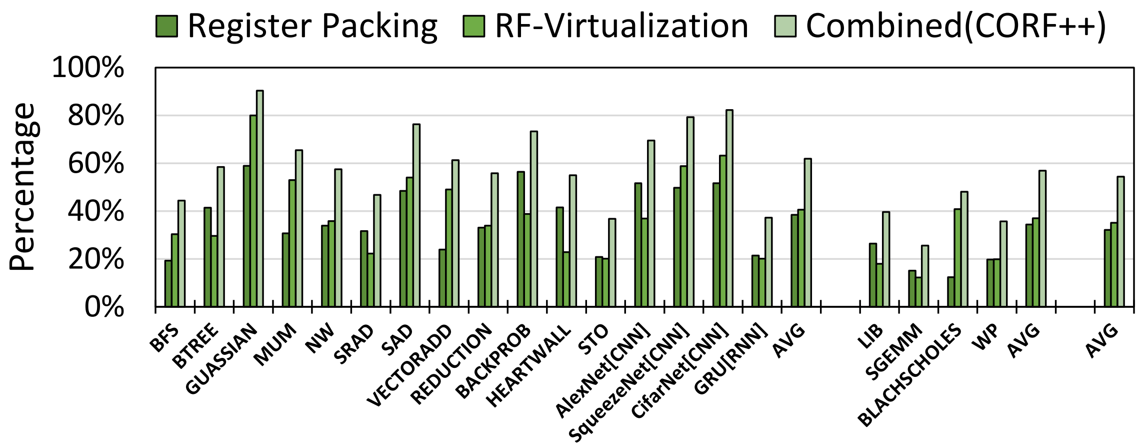
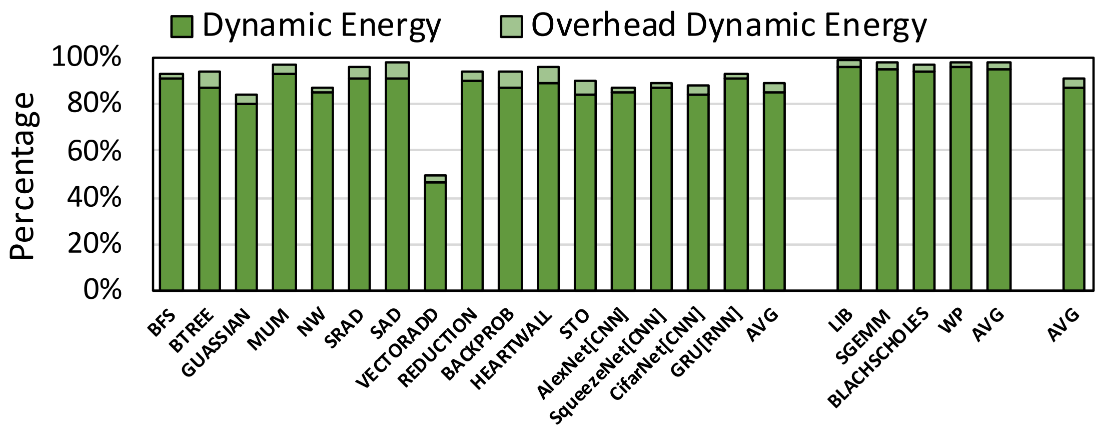
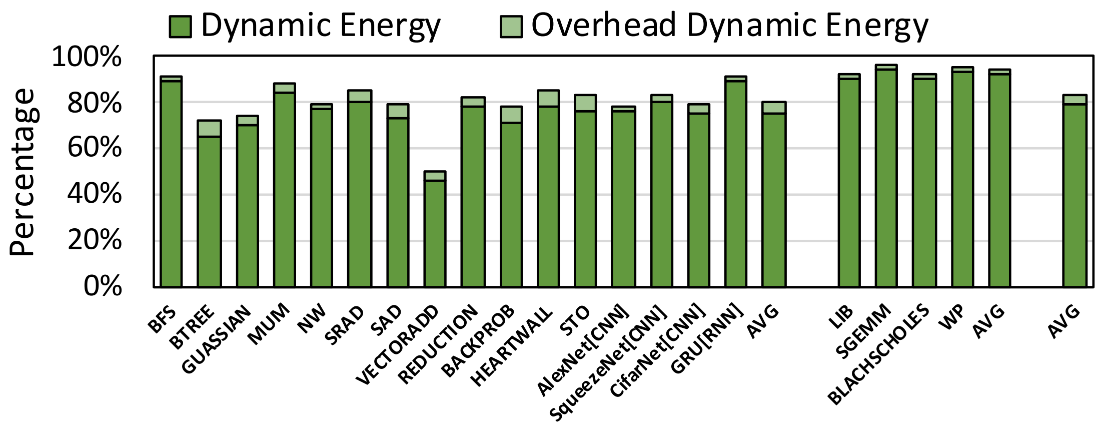


Figure 2.16: Reduction in allocated physical registers.



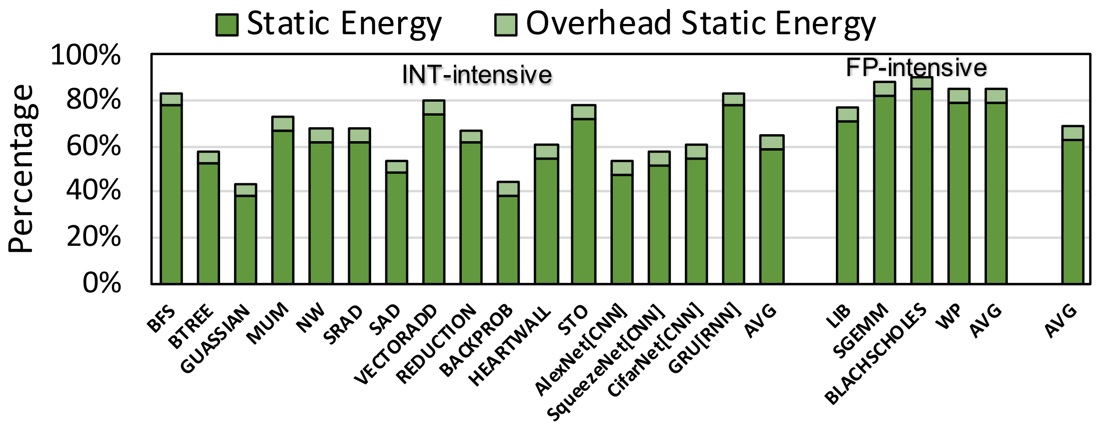
(a) CORF



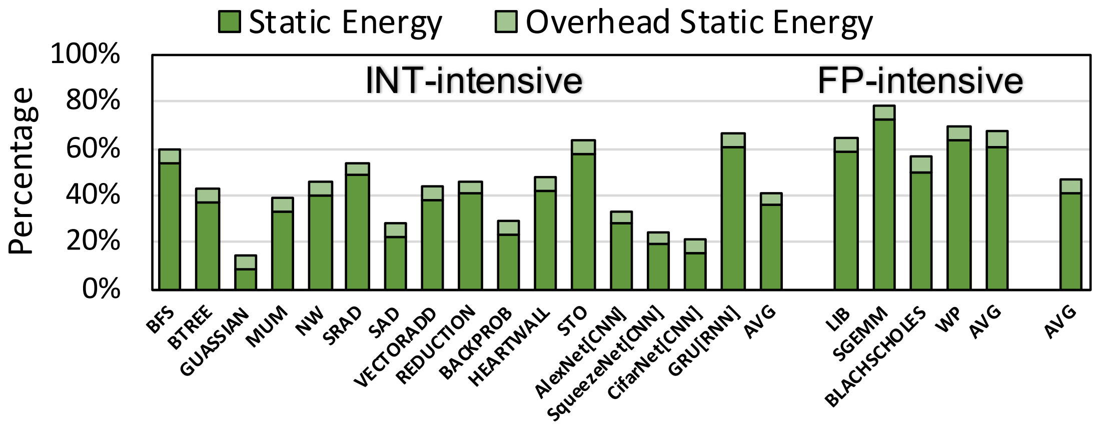
(b) CORF++

Figure 2.17: Normalized RF dynamic energy

RF energy: Figures 2.17 and 2.18 show the dynamic energy and leakage energy impact of our techniques. The small segments on top of each bar represent the overheads of the structures added by CORF/CORF++. Dynamic energy savings in Figure 2.17 are due to the reduced number of accesses to the register file because of operand coalescing. We observed 8.5% and 17% reduction to the overall dynamic energy in CORF and CORF++, respectively, after considering the 3% increase in overheads. The source of dynamic energy overheads include the packers and unpackers, width detection logic, and the accesses to the renaming table.



(a) **Register packing.**



(b) **Combined with virtualization (CORF++).**

Figure 2.18: Normalized RF leakage energy

Figure 2.18 shows the leakage energy for register packing and also the combined register packing and virtualization (CORF++), assuming that we power gate unused

Technique	IPC	Register Reads	RF Dyn. Energy	RF Size
Register Packing	1	1	1	0.65
Register Packing + Virtualization	1	1	1	**0.43**
CORF	**1.04**	**0.9**	**0.92**	**0.43**
CORF++	**1.09**	**0.77**	**0.83**	**0.43**

Table 2.1: Summary of CORF, CORF++, and register packing (and register virtualization). All values normalized to the baseline GPU register file.

registers. Leakage energy is reduced by 33% in register packing (Figure 2.18a), and 52% for the combined with virtualization (CORF++, Figure 2.18b), after accounting for the overheads. On average, the leakage overhead, due to the additional structures (e.g. renaming table, free-register map), is 5.4%, which is easily out-weighed by the leakage energy savings.

We summarize the advantages of CORF/CORF++ compared to register files without coalescing in Table 2.1. Note that Wang et al. [139] evaluate the performance of register packing when they exploit the smaller effective register file to allow more threads to run concurrently per SM. This IPC improvement technique is orthogonal to coalescing and can be combined, therefore we do not include it for comparison.

2.8 Hardware/Software Overheads

Hardware overheads: The largest additional structure in CORF is the renaming table, which is also needed for simple register packing [43, 139]. Each RT entry consists of 14 bits that encodes the physical register and slice to which an architectural register is being mapped. Since our baseline architecture supports up to 48 warps per an SM, and 63 registers per warp, for a total of just over 3000 potential warp architectural registers. Each register has an entry in the table. Therefore, RT total size is $5.16KB$ which is 4% of total $128KB$ register file per each SM. The free register map size is $1024 \times 4 - bits$ or $512bytes$. Supported by the RT, register packing and virtualization reduce the effective register file size to less than half of its original size: *the benefits of shrinking the register file easily offset the overhead, before we even consider coalescing.* We calculate the renaming table and register file power consumption using CACTI v5.3 [121] and report them in Table 2.2.

The overhead of logic, such as the allocation policy logic, coalescing logic, packers, unpackers, and width detection units, was estimated by synthesizing its Verilog HDL description using Synopsys Design Compiler and the NCSU PDK 45nm library. The static and dynamic energy of these logics are also included in our power results. All together, these logic accounts for 57mW of dynamic power, 0.2mW static power, and 0.05mm^2 (or 0.11%) of total on-chip area.

Parameter	Renaming table	Register bank	Percentage
Size	5KB	128KB	3.9%
# Banks	4	4	-
Vdd	0.96V	0.96V	-
Access energy	1.83pJ	149.76pJ	1.2%
Leakage power	5.56mW	89.6mW	6.2%

Table 2.2: Renaming table overheads in 40nm technology

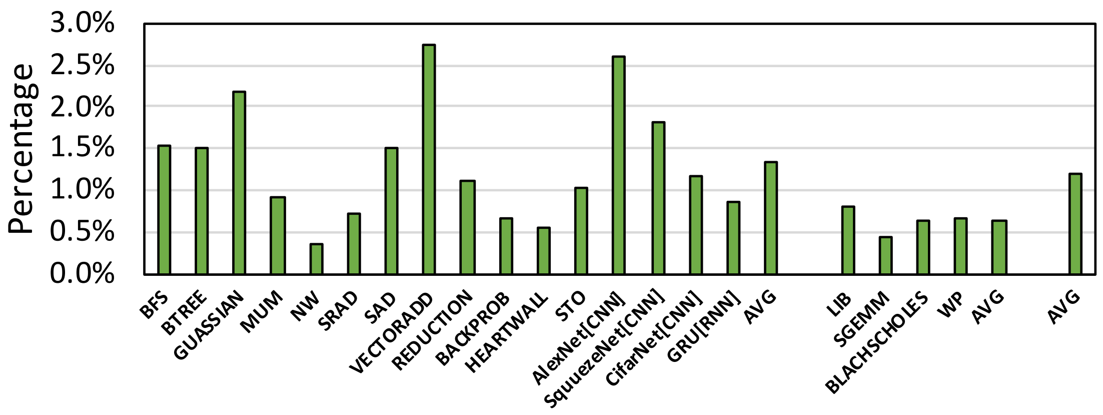


Figure 2.19: Static code size increase.

Software overheads: Figure 2.19 shows the static code increase due to the addition of extra instructions to guide CORF. Overall, CORF only increases the code size by 1.3%. Passing information in CORF++ can be simplified, for example, by having the compiler choose odd register numbers for the left operands, and even numbers for the right operands without explicit metadata instructions. When considering dynamic instruction count, this overhead will be significantly lower.

2.9 Related Work

Energy efficiency of GPU has been an area of increasing importance [6, 7, 9, 10, 12, 30, 70, 83, 84, 88, 91, 114, 116–118, 131, 140–142]. These prior works have explored improving the performance or energy efficiency of GPU register files in a number of ways. In this section, we will highlight works related to GPU register files.

Warped Register File [7] introduces a tri-modal register file structure that enables drowsy mode. Pilot Register File [6] proposed an energy-efficient RF design using FinFETs.

Register File Caching [45, 46] proposed to add a small register file cache to reduce overall RF dynamic power by storing frequently accessed registers in an energy-efficient cache. However, these techniques solely aim to reduce power, with the goal of achieving a negligible performance penalty.

Several works aim to improve the performance of register files. RegMutex [73] improved performance by sharing a subset of physical registers between warps during the GPU kernel execution. FineReg [105] achieved a higher number of concurrent CTAs by partitioning the register file into two regions, one for active CTAs and another for pending CTAs. Register file slicing [49] proposed to split the data path into two 16-bit slices, which enables the register to save power by power gating a slice if storing narrow-values, *or* to improve performance by fetching two 16-bit values. RF slicing fundamentally trades-off between a power-efficient mode, or a performance-enhancing mode.

Another commonly used energy efficiency technique is value compression [13, 80, 92, 108, 109, 119, 136, 144, 148]. Register File Compression [80], utilize base-delta-immediate (BDI) compression to compress data within an entry and power-gate sub-banks. While Register Packing [43, 139] compress narrow values to use less physical register entries, and power gates unallocated entries.

Wang et al. [139] were the first to propose register packing for GPUs. Specifically, they greedily pack narrow-value registers together to reduce register file space. They do not coalesce register reads – each register read still requires a separate physical register file read operation. Register file virtualization [62] reduces the number of allocated physical registers required (and power gate unallocated entries), through register liveness analysis.

While achieving power savings, these techniques do not improve performance. In our work, by combining packing and virtualization and also harnessing coalescing opportunities, we achieve higher compression ratios, power savings, *and* performance improvements.

RegLess [78] replaces the register file with a smaller staging unit with the help of compiler annotations, leveraging the short-lived and long-lived behaviors of the register. RegLess achieves lower power and smaller register storage size while maintaining performance. The Latency-Tolerant Register File (LTRF) [115] similarly uses compiler-analysis to identify registers to move into a register cache, which enables tolerance of large register files. However, this higher performance comes at the cost of a larger, more power-hungry register file.

Chapter 3

BOW: Breathing Operand Windows to Exploit Bypassing in GPUs

3.1 Introduction

Graphics Processing Units (GPUs) have emerged as an important computational platform for data-intensive applications in a plethora of application domains. They are commonly integrated in computing platforms at all scales, from mobile devices and embedded systems, to high-performance enterprise-level cloud servers. GPUs use a massively multi-threaded architecture that exploits fine-grained switching between executing groups of threads to hide the latency of data accesses. In order to support this fast context switching at scale, GPUs invest in large Register Files (RF) to allow each thread to maintain its

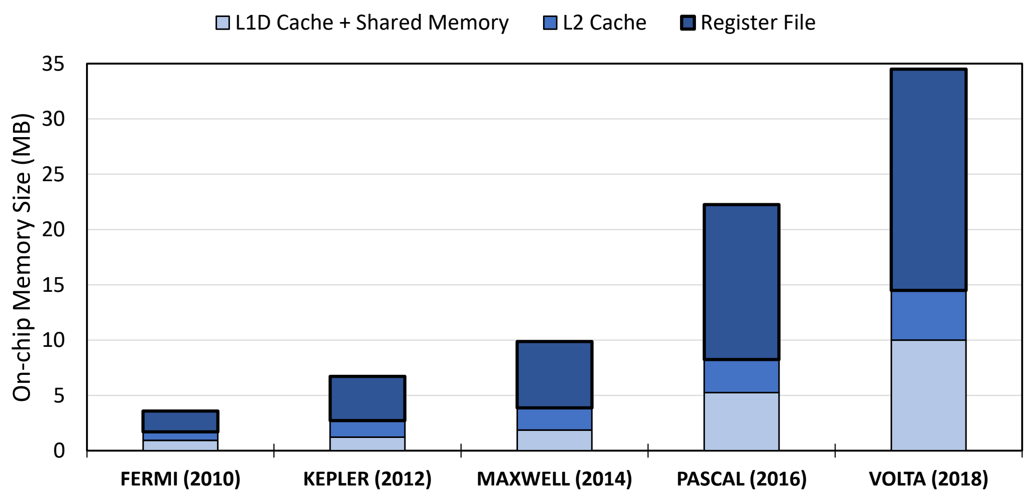


Figure 3.1: On-chip memory components size in NVIDIA GPUs (from 2010–2018).

context in hardware. The amount of parallelism available on a GPU (e.g., number of streaming multiprocessors, or SMs) has been steadily increasing as GPUs continue to grow in performance and size, which in turn increases the number of concurrent thread contexts needed to keep these units utilized [15, 61, 72, 73, 100, 103, 133].

The large register file accounts for an increasingly larger fraction of on-chip storage, as shown in Figure 3.1. For example, in NVIDIA Pascal GPU, register file size is 14 MB, which accounts for around 63% of the on-chip storage area. Due to frequent accesses to the RF, it is a crucial microarchitectural component whose architecture substantially impacts the performance and energy-efficiency of GPUs. For example, port conflicts (in register file banks as well as operand collector units that collect the register operands) cause delays in issuing instructions as register values are read in preparation for execution. In addition, the RF has a large energy consumption footprint, since it is the largest SRAM structure that serves a large number of data accesses from the working threads. Earlier studies estimate that the register file is responsible for 18% of the total power consumption on a GPU chip [81], a percentage that has most likely increased as the size of RFs has continued to grow.

We propose a new GPU architecture technique, *Breathing Operand Windows (BOW)*, exploits the temporal locality of the register accesses to improve *both* the access latency and power consumption of the register file. More specifically, we observe that registers are often accessed multiple times in a short window of instructions, as values are incrementally computed or updated and subsequently used. As a result, a substantial fraction of register read and register write accesses can bypass the register file if mechanisms exist to forward them directly from one instruction to the next. This *operand bypassing* reduces dynamic access energy by eliminating register accesses (both reads and writes) from the RF, and improves overall performance by reducing port contention and other access delays to the register file banks.

BOW re-architects the GPU execution pipeline to take advantage of operand bypassing opportunities. Specifically, in the baseline design we consider operands reused within an instruction window: a key to increasing bypassing opportunities is to select the instruction window size carefully to capture register temporal reuse opportunities while maintaining acceptable overheads for the forwarding. To facilitate bypassing we dedicate an operand collector to each warp so that it can hold the set of active registers for that warp in a simple high performance buffering structure dedicated for each warp. Whenever a register operand is needed by an instruction, BOW first checks if the operand is already buffered so it can use it directly without the need to load it from the RF banks. If the operand is not present in the operand collector unit, a read request will be generated to the

RF, which is sent to the arbitrator unit. In the baseline BOW, after an instruction finishes execution, the computed result is written back to both the operand collector unit as well as the register file (i.e., a write through configuration). This organization supports reuse of operand reads and avoids the need for an additional pathway to enable writing back values from the operand collector to the RF when they slide out of the window. Based on our experiments, BOW with a window size of 3 instructions reduces the physical register read accesses by 59% across all of our benchmarks. However, it does not support write bypassing since every write is still written to the RF; in fact, it increases the overhead for writes which are now written to both Operand Collector and RF.

In order to be able to capitalize on the opportunities for write bypassing, we introduce BOW-WR, an improved design that uses a write-back philosophy to overcome the redundant writes present in BOW. Specifically, the improved design writes any updated register values back to the operand collector only. When an instruction slides outside of the active bypass window its updated register value is written back to the RF *only if it has not been updated again by a subsequent instruction in the window* (in which case that first write has been bypassed since the update was transient). As described, BOW-WR shields the RF from some of the write traffic, but does not capture all write bypassing opportunities, and preserves some redundant and inefficient write behavior. Consider the following two cases: (1) Unnecessary OC writes: When a value will no longer be reused, writing it to the OC first, and then to the RF causes a redundant update. We are better off writing such value directly to the RF; (2) Unnecessary RF writes: When an updated register value is no longer live (i.e., it will not be read again before it is updated), it will be written back to the

RF unnecessarily when the instruction slides out of the active window. In this case, we are better off not writing the value back to the RF.

Unfortunately, it is difficult to capture either of these opportunities directly in the architecture because they depend on the subsequent behavior of the program. Thus, to exploit the opportunity to eliminate these redundant write backs in BOW-WR, we task the compiler to do liveness analysis and classify each destination register to one of these three groups: those that will be written back only to the register file banks (to handle case 1 above); operands that will be written back only to the operand collectors (to handle case 2); and finally operands that first need to reside in operand collector and then due to their longer lifetime need to be written back to the register file banks for later use (this was the default behavior of BOW-WR before the compiler hints). We pass these compiler hints to the architecture by encoding the writeback policy for each instruction using two bits in the instruction. This compiler optimization not only substantially minimizes the amount of write accesses to the register file and fixes the redundant write-back issue, but also reduces the effective size of the register file as a significant portion of register operands are transient, not needed outside the instruction windows (52% with a window size of 3): we avoid allocating registers altogether in the RF for such values.

With respect to implementation, a primary cost incurred by the baseline BOW(and BOW-WR) is the cost of increasing the number of operand collectors (so that there is one dedicated per warp) as well as the size of each operand collector to enable it to hold the register values active in a window. With respect to increasing the number of OCs, we believe that this is in line with current trends in GPUs: While earlier Nvidia GPUs had a smaller number of operand collector units, starting from the Kepler series, the number of their operand collector units have increased. For example, NVIDIA TITAN X GPU (Pascal architecture) has 32 operand collectors which matches the maximum number of in-flight warps on an SM. With respect to the size of each OC, the baseline implementation adds additional entries to each operand collector to hold the operands within the active window (4 registers per instruction in the window). In the baseline implementation, this adds around 36KB of temporary storage for a window size of 3 across all OCs, which is significant (but still only around 14% of the RF size of modern GPUs). In order to reduce this overhead, we observe experimentally that this worst case sizing substantially exceeds the mean effective occupancy of the bypassing buffers. Thus, we provision BOW-WR with smaller buffering structures. However, since the available buffering can be exceeded under the worst case scenarios, we have to redesign the OCs to allow eviction of values when necessary. We also restrict the window size to the predetermined fixed window size and do not bypass instructions beyond the window size even if there is sufficient buffer space in the buffering structure. The reason behind this conservative choice is to facilitate the compiler analysis and tag the writeback target in BOW-WR correctly in the compiler taking into account the available buffer size. Without this simplifying assumption, an entry which is tagged by the compiler for no writeback to the RF may need to be saved if it is evicted before all its reuses happen. We are able to reduce the storage size by 50% with a performance reduction of less than 2% of the baseline BOW-WR. Considering other overheads (such as modified interconnect), BOW requires an area increase of 0.17% of total on-chip area.

Because of the importance of the RF structure on GPUs, a number of prior studies have explored optimizations primarily to reduce its energy footprint. A number of works have explored different approaches to reduce the effective size of the register file [15,62,78,80]. The effect of reducing the register file size is to improve the static energy consumption of the RF, but it does not impact the performance or the dynamic energy consumption of the RF. Most similar to our work, RF caching [45] adds a register file cache to keep the most commonly used data for each active warp, saving dynamic RF energy. This cache is organized like the original RF, but only smaller, and therefore there it improves energy but unlike BOW it does not improve performance.

We compare our work with register file caching and other related works in more detail in Section 3.6.

In summary, the chapter makes the following contributions:

1. Introduces *Operand Bypassing*, a new technique in the context of GPU microarchitecture that capitalizes on the high temporal reuse of GPU register operands to substantially reduce accesses to the register file, improving performance and energy.

2. We leverage compiler liveness analysis to guide destination selection of the write-back register values, substantially reducing unnecessary write traffic. Bypassed transient values are also never allocated in the RF reducing the effective RF size.

3. We carry out occupancy analysis of the forwarding buffers and discover that their utilization is low. We propose to provisioning the operand collectors with smaller buffer structures to substantially reduce storage overhead.

Overall, BOW-WR improves IPC by 11%, and reduces dynamic energy of the RF by 55%, at a modest overhead of 0.17% increase in the total chip area, and 4% increase in storage (compared to the RF size).

3.2 Background

In this section, we overview the organization of modern GPU architecture, with a focus on the register file unit, to provide the necessary background for BOW. In the GPU execution model, a kernel is the unit of work issued typically from the CPU (or directly from another kernel if dynamic parallelism is supported). A kernel is a GPU application function, decomposed by the programmer into a grid of blocks mapped each to a portion of the computation applied to a corresponding portion of a typically large data in parallel. Specifically, the kernel is decomposed into Thread Blocks (TBs, also Cooperative Thread Arrays or CTAs), with each being assigned to process a portion of the data. These TBs are then mapped to streaming multiprocessors (SMs) for execution. The threads executing on an SM are then grouped together into *warps* (or *wavefronts* in AMD terminology) for the purposes of scheduling their issuance and execution. Warp instructions are selected and issued for execution by warp schedulers in the SM (typically 2 or 4 schedulers, depending on the GPU generation). Warps that are assigned to the same warp scheduler compete for the issue bandwidth of that scheduler. In our baseline GPU (NVIDIA Titan X Pascal), there are four schedulers per SM, each able to issue two instructions per cycle to available GPU cores.

All the threads in a warp execute instructions in a lock-step manner (Single Instruction Multiple Thread, or SIMT model). Most GPU instructions use registers as their

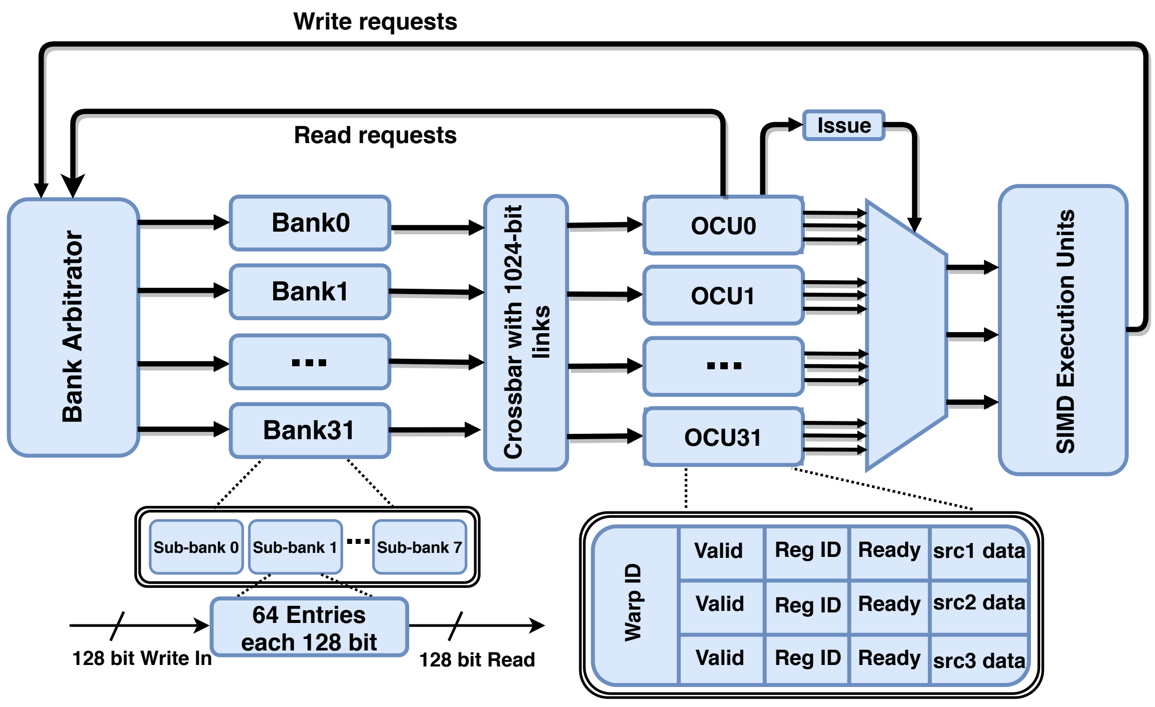


Figure 3.2: Conventional GPU register file architecture (OCU0-OCU31 are Operand Collector Units).

source and/or destination operands. Therefore, an instruction will access the Register File (RF) to load the source operands for all of its threads, and will write back any destination operand after the execution to the RF. The RF in each SM is typically organized into multiple single-ported register banks so as to support a large memory bandwidth without the cost and complexity of a large multi-ported structure. A banked design allows multiple concurrent operations, provided that they target different banks. When multiple operations target registers in the same bank, a *bank conflict* occurs and the operations are serialized, affecting performance.

Figure 3.2 shows the baseline register file organization for the Pascal generation of NVIDIA GPUs, with a size of 256 KB per SM split across 32 banks. A bank is made up of 8 sub-banks that are 128 bits wide each. All 32 registers belonging to the 32 threads in the same warp are statically allocated to consecutive sub-banks (in a single bank) with the same entry index. Thus, a full register for all the threads within a warp can be striped using one entry of one bank, allowing it to be operated on in a single cycle. Each bank can store up to 64 warp-registers.

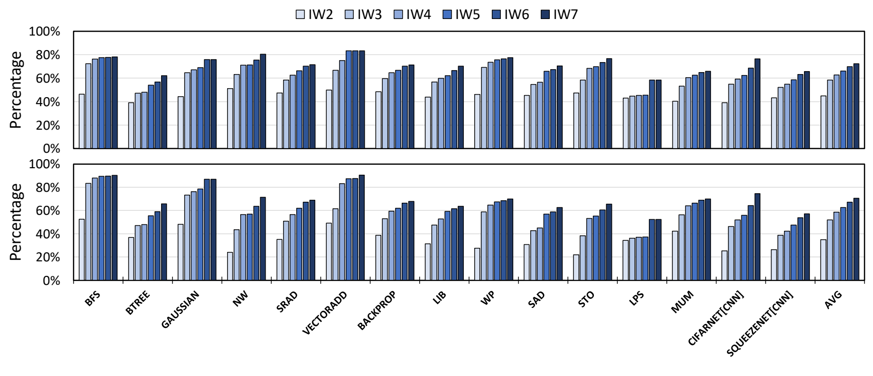


Figure 3.3: Eliminated read (top) and write (bottom) requests through operand bypassing.

When a warp instruction is issued for execution, an Operand Collector Unit (OCU) is assigned to it to collect its source operands values. Assuming 32-thread warps, each source operand (i.e., warp register) is 32 $thread \times 32\ bits = 128B$ in size. A warp's source operands are read from the RF banks and then buffered in the OCU. The operand collector units are not used to eliminate name dependencies through register renaming, but rather are used as a way to space register operand accesses out in time so that no more than one access to a bank occurs in a single cycle. To reduce the interconnect network complexity, operand collectors are designed as single-ported buffers. An OCU fetches the register operands from the register banks they reside in, bound by the two following constraints: (1) *OCU port serialization*: Each OCU has only one port and therefore has to serialize reads when an instruction has multiple operands (NVIDIA GPU's use SASS whose instructions have up to 3 source operands); and (2) *Register bank conflicts*: While operands from different banks may be concurrently read from different OCUs, operands that access the same bank cause bank conflicts and cannot be issued together. The port constraints causing these conflicts are difficult to bypass by increasing the number of ports: the cost of a port is extremely high when considering the width of a warp register (128 Bytes).

Once all the source operands for a warp instruction are collected, it is ready for execution. Since each instruction may have up to three source operands [101], each OCU has three entries, each 128B to hold these operands. After the warp completes the execution, its results are written back to the RF, also competing for bank access with read operations. When this set of operations is performed repeatedly, it will generate many accesses to the large register file, and will incur a significant portion of the power consumed by the GPU. The RF also impacts performance due to the serialization that occurs due to port contention (in both register file banks as well as operand collector units).

3.3 Motivation

In this section, we motivate operand bypassing by studying register reuse patterns within different instruction window sizes. We use the GPGPU-Sim simulator [17], modeling a Pascal GPU. Given the in-order execution of GPUs, repeated accesses on operands within a small window of consecutive instructions are inevitable. Although we show results only for the

Pascal architecture configuration, we repeated the results for Fermi and Volta configurations, which exhibit almost identical reuse statistics confirming that operand reuse patterns are computational properties rather than architecture dependent [18,21]. Experiments in this chapter use benchmarks from Rodinia [31], Parboil [125], NVIDIA CUDA SDK [101], and the Tango DNN Benchmark Suite [68].

Temporal locality in register operand accesses: In a conventional GPU Register File, each operand collector unit sends read requests for the source operands of one instruction (the one which currently resides in the operand collector unit). This read request process repeats independently for each individual instruction, with all register operands fetched from or written to the register file independently for each instruction. Our work is motivated by the observation that there is high temporal locality in the accesses of registers: in other words, the same register values are read and updated by nearby instructions, within a short window of instructions. If this is indeed the case, the traditional execution pattern where these operands are read and written repeatedly through the register file causes redundant expensive operations to the RF increasing both the power consumption of this large structure, as well as the access time due to the increased pressure on the limited ports of the RF.

To characterize the temporal reuse opportunity [19,20,22], we show in Figure 3.3 all bypassing opportunities for read (top) and write (bottom) requests to the register file, for different window instruction sizes and across 15 different benchmarks. An instruction window (IW) refers to a number of consecutive instructions from the same warp: an IW of 2 considers a sliding window of two instructions at a time and examines whether the operands of the first instruction are also needed by the second one. Note that a value that is reused in three consecutive instruction can continue to be bypassed even with an IW of 2 since the instruction window for bypassing is a sliding window. While we can bypass 45% of total read accesses and 35% of total write accesses to the register file with a window of just two instructions, a window of three instructions would eliminate substantially more accesses: 59% of total reads, and 52% of total writes on average. Beyond a window size of three instructions, the reuse opportunities continue to increase slowly, reaching over 70% with an instruction window of 7. Clearly, if we save this portion of register file accesses, we can substantially improve the dynamic energy consumption of the register file (by reducing the number of RF accesses) as well as performance (by reducing access time and port contentions in register file banks). A larger window size increases reuse opportunities, but comes at the price of wider (bigger) operand collectors which increase the area and energy consumption within those components. An effective BOW configuration balances these competing considerations.

Impact of operand collection stage latency on performance: The operand collection stage of the GPU pipeline holds issued instructions until their operands can be collected, typically from the register file. Figure 3.4 shows a breakdown of the percentage of cycles taken on average for memory instructions versus non-memory instructions within the operand collection stage of the pipeline. In our experiments, we excluded the amount of time spent for an instruction to be fetched; total execution time assumed to be from the moment that an instruction is scheduled by one of the warp schedulers until it finishes execution. Overall, about a quarter of the instruction execution time (and up to 47% for benchmarks

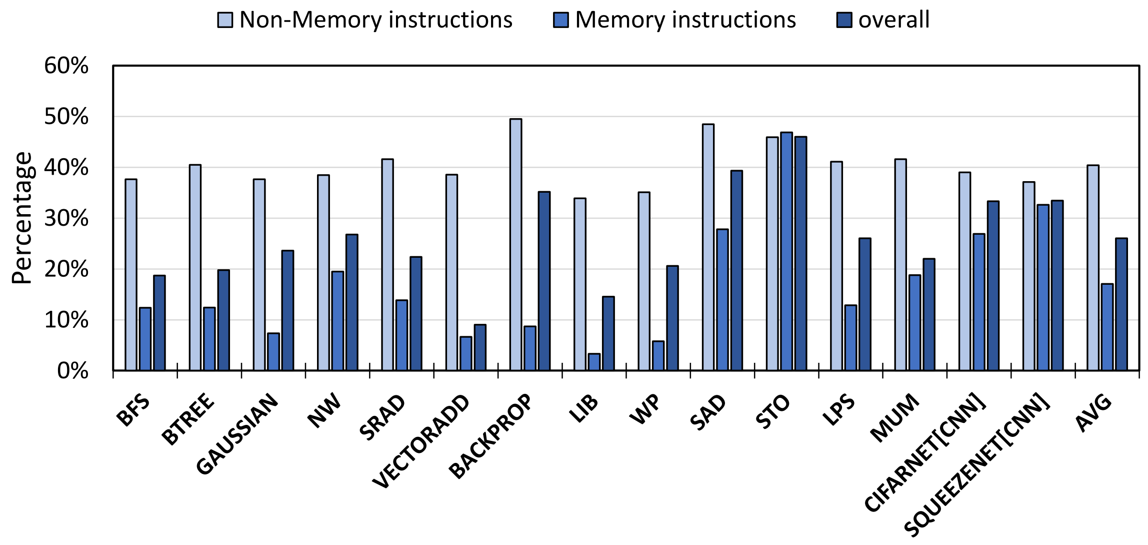


Figure 3.4: Average time taken by operand collection stage for memory vs. non-memory instructions.

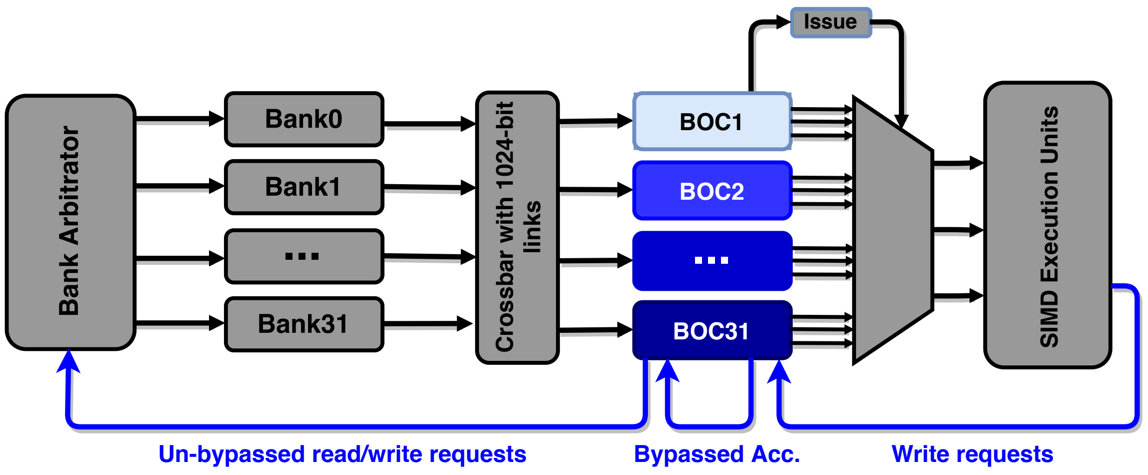
such as STO) is spent in the operand collector unit. We note that this percentage is skewed by memory access instructions which have long execution times as well as a fewer number of operands (especially `global load` and `global store` instructions with cache misses). The operand collector unit consumes a substantial percentage of the execution time of non-memory instructions, as depicted in Figure 3.4. The primary delays in the OC occur while registers are read from the RF and are being collected in the single-ported operand collectors. As discussed previously, register reads for each OC are serialized since it is a single-ported buffer-like structure. Moreover, some reads are delayed due to register bank conflicts. With bypassing, as we decrease RF traffic, and with more operands already available in the OC, we expect the time spent in the OC to significantly decrease, improving overall performance.

3.4 Breathing Operand Windows

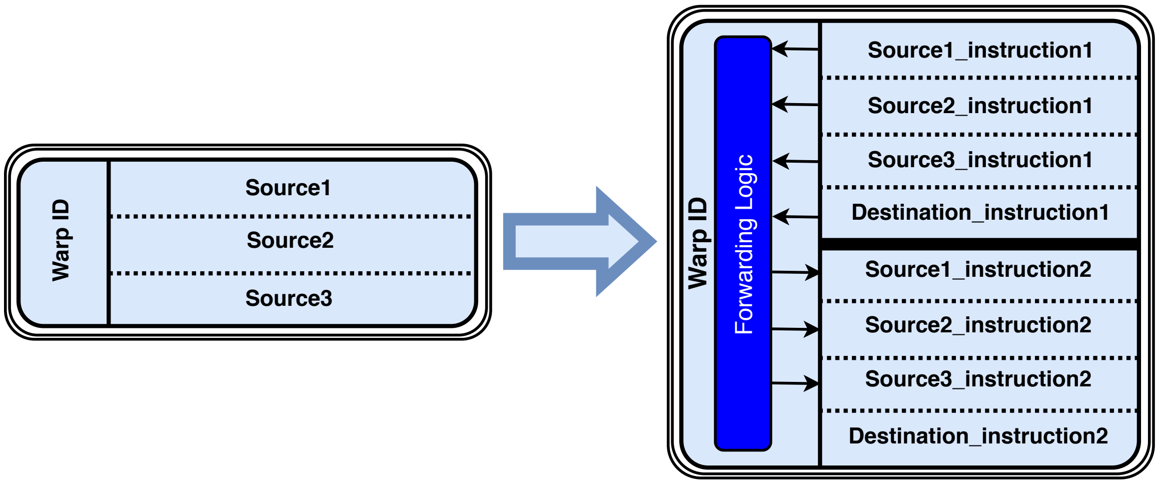
In this section, we overview the design of BOW, the proposed architecture which exploits high temporal operand reuse to bypass having to read and write reused operands to the register file. We also introduce a number of compiler and microarchitectural optimizations to improve reuse opportunities, as well as to reduce overheads. BOW consists of 3 primary components. (1) Bypassing Operand Collector (BOC) augmented with storage for active register operands to enable bypassing among instructions. Each BOC is dedicated to a single warp; this restriction simplifies buffering space management since each buffer is accessed only by a single warp. The sizing of the BOC is determined by the instruction window size within which bypassing is possible; (2) Modified operand collector logic that considers the available register operands and bypasses register reads for available operands (whereas baseline operand collectors fetch all operands from the RF); and (3) Modified write-back pathways and logic which enable directing values produced by the execution units or loaded from memory to the BOCs (to enable future data forwarding from one instruction to another) as well as to the register file (for further uses out of the current active window) in the baseline design. The writeback logic is further optimized with compiler-assisted hints in the improved BOW-WR.

3.4.1 BOW Architecture Overview

Figure 3.5 overviews the proposed architecture highlighting the primary changes and additions. The design centers around new operand collector unit additions, called the Bypassing Operand Collectors (BOC) in our design, that will allow the GPU to bypass



(a)



(b)

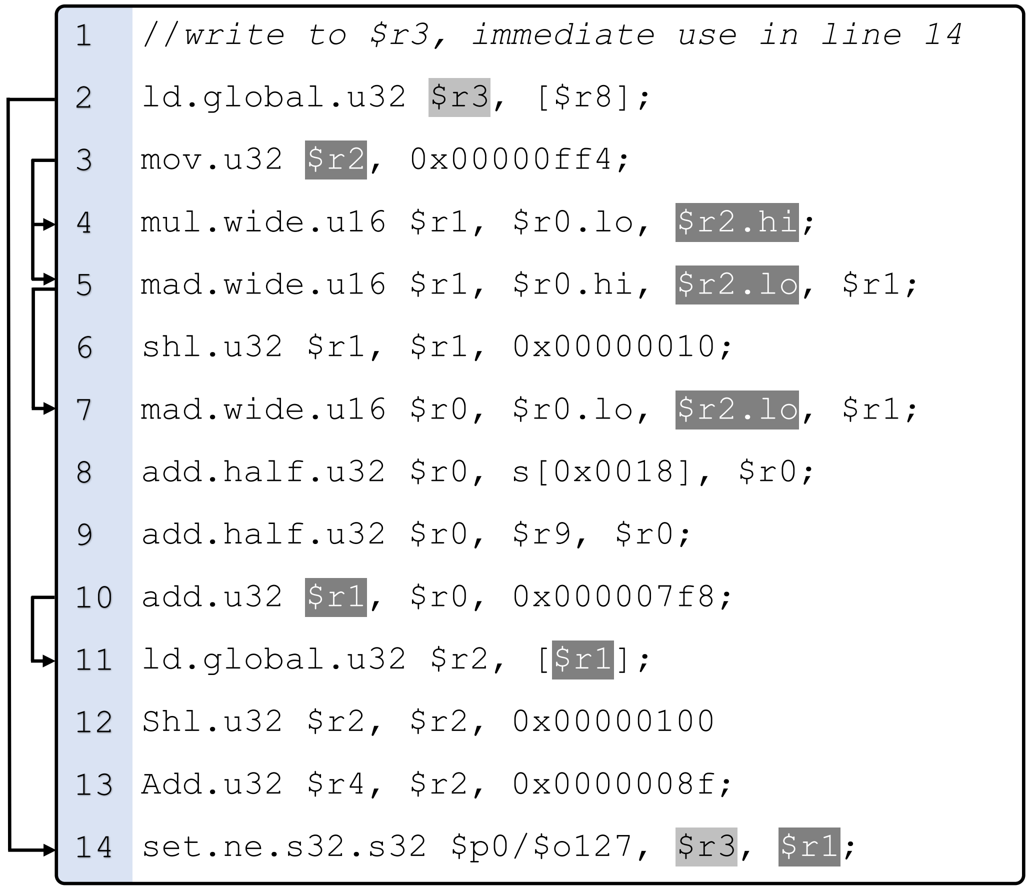
Figure 3.5: (a) An overview of BOW. BOCX is Bypassing Operand Collector assigned to Warp X; (b) Baseline operand collector unit (left) compared to the proposed wider Bypassing Operand Collector (BOC) unit with forwarding logic support (right).

RF accesses. Each BOC is assigned to a single warp (*BOC0-BOC31*) in Figure 3.5(a). While the operand collectors in our baseline architecture have three entries to hold the data of the source operands of a single instruction (Figure 3.5(b), left), BOW widens the operand collectors to enable the storage of source and destination register values for the usage of subsequent instructions (Figure 3.5(b), right). In addition, the forwarding logic in the BOC will check whether the requested operands are already in the BOC so will be sent to the next instruction. Similar to the baseline architecture, and to avoid making the interconnection network more complicated, BOCs have a single port to receive operands coming from the register file banks. However, the forwarding logic within the BOCs allows forwarding multiple operands available in the forwarding buffers when an instruction is issued. In the baseline design, we conservatively reserve four entries per each instruction in the BOC to match the maximum possible number of operands which is three source operands plus one destination. Later we show that such conservative sizing is rarely needed, enabling us to provision the BOC with substantially smaller storage.

Instructions for the same warp are scheduled to the assigned BOC in program order as the instruction window slides through the instructions. When instruction x at the end of the window is inserted into the BOC, the Forwarding Logic checks if any of the required operands by instruction x is already available in the current window, then the oldest instruction (first instruction in the current window) with its operands are evicted from the window to make room for the next instruction, which will become available when the window moves. It is important to note that the instruction window is sliding; every time an operand is used by an instruction it remains active for window size instructions after that. If it is accessed again in this window, its presence in the BOC is extended in what we refer to as the *Extended Instruction Window.* When a branch occurs, the BOC waits until the next instruction is determined.

Instructions from *different* BOCs are issued to the execution units in a round-robin manner. As soon as all the source operands for an instruction are ready (which potentially have been forwarded directly within the active window and without sending read requests to the register file), the instruction is dispatched and sent to the execution unit. When the execution of an instruction ends, its computed result is written back to the assigned BOC (to be used later by next instructions in the window). In the baseline BOW, this value is also written back to the register file (for potential later uses, if any, by an instruction out of the current window). It is worth mentioning that only the pathway from execution units to the BOCs has been added in our design thusfar, as the pathway from execution units to the register file is already established in the baseline architecture. While such simple write-through policy minimizes the complexity, it suffers substantial of redundant write backs (to the BOCs as well as register file); an inefficiency which will be addressed in BOW-WR.

Please note that *two dependent instructions* (where there is a RAW or WAW dependency between them) can never be among the ready to issue instructions within the same BOC. The scoreboard logic checks for this kind of dependencies prior to issue instructions to the operand collection stage (this is actually done when a warp scheduler schedules an instruction). Having an instruction in one of the BOCs means that it has already passed the dependency checks and its register operands exist either in the BOC or the register file. For independent instructions, there is no delay for bypassing: both can start executing, and even finish out-of-order.



```
1   //write to $r3, immediate use in line 14
2   ld.global.u32 $r3, [$r8];
3   mov.u32 $r2, 0x00000ff4;
4   mul.wide.u16 $r1, $r0.lo, $r2.hi;
5   mad.wide.u16 $r1, $r0.hi, $r2.lo, $r1;
6   shl.u32 $r1, $r1, 0x00000010;
7   mad.wide.u16 $r0, $r0.lo, $r2.lo, $r1;
8   add.half.u32 $r0, s[0x0018], $r0;
9   add.half.u32 $r0, $r9, $r0;
10  add.u32 $r1, $r0, 0x000007f8;
11  ld.global.u32 $r2, [$r1];
12  Shl.u32 $r2, $r2, 0x00000100
13  Add.u32 $r4, $r2, 0x0000008f;
14  set.ne.s32.s32 $p0/$o127, $r3, $r1;
```

Figure 3.6: Code snippet from BTREE application illustrating bypassing operation in BOW.

3.4.2 BOW-WR: Compiler-guided writeback

BOW exploits read bypassing opportunities, but is not able to bypass any of the possible write operations as every computed value is written not only to the RF, but also to the BOC, following a write-through policy for simplicity. However, write bypassing opportunities are important: often a value is updated repeatedly within a single window. For example, consider `$r1` being updated by the instructions in lines 4, 5, and 6 of Figure 3.6; it only needs to be updated in the RF after the final write.

BOW-WR approaches bypassing using a write-back philosophy to enable write bypassing. In the simplest cast, it writes the computed results always to the BOC to provide opportunities for both read and write bypassing. When an updated operand slides out of the current active window, the forwarding logic checks if it has been updated again by a subsequent instruction within the active window. If so, the write operation will be bypassed, allowing the consolidation of multiple writes happening within the same window. In our prior example (Figure 3.6), when instructions 4 and 5 slide out of the active window, their updated `$r1` is discarded since in each case `$r1` is updated again within the window. When instruction 6 slides out, the value is written back (since neither instruction 7 nor 8 update `$r1`). The primary cost of BOW-WR (write-back instead of write-through) is that a new pathway needs to be established from BOCs to the RF.

Although using a write-back philosophy [69] significantly reduces the amount of redundant writes to the register file (Table 3.1), it is not able to bypass all such write operations; in many instances, as an operand slides out of an active window, it is written back from the BOC to the register file while it is not actually going to be used again by later instructions (the operand is no longer live). Another source of inefficiency arises since computed operands are always written back to the BOC; if these operands are not needed again in the active window, they could have been written directly to the RF, eliminating the write to the BOC.

In either of these situations, unfortunately, the microarchitecture does not have sufficient information to identify the optimal target of the writeback, since it depends on the future behavior of the program which is generally not visible at the point where the writeback decisions are made, leading to the redundant writes. Thus, to enable elimination of these redundant writes, we rely on the compiler to analyze the program and guide with the selection of the write back target. Specifically, the compiler performs liveness analysis and dependency checks to determine if the output data from an instruction should be written back only to the register file bank (when it will not be used again in the instruction window), only to the bypassing operand collector (for transient values that will be consumed completely in the window and no longer live after it), or both (which is the default behavior without the compiler hint). When we avoid writing values back to the RF, we reduce the pressure on the RF and avoid the cost of unnecessary writes for operands that are still in use. Similarly, when we write data to the BOC which is not going to be used, we pay the extra cost of this write only to later have to save the value again to the RF. An interesting opportunity also occurs in that transient values that are produced and consumed completely within a window, no longer need to be allocated a register in the RF. We discover that many operands are transient, leading to a substantial opportunity to reduce the effective RF size. Compiler-guided optimizations will allow us to avoid unnecessary writes and minimize energy usage. Table 3.1 shows the needed number of write accesses to the RF for the code in Figure 3.6 in the different versions of BOW(note that BOW write-through is identical to the unmodified GPU).

Destination	# of write accesses to the Register file in:		
Operand	BOW	BOW	BOW-WR
	(write-through)	(write-back)	(compiler Opt.)
$r0	3	1	0
$r1	4	2	1
$r2	2	1	0
$r3	1	1	1
Total	10	5	2

Table 3.1: Number of write operations to the register file for code snippet shown in Figure 3.6.

There are three possible actions which can be taken after an instruction's output value is generated, which we explain using a piece of code from BTREE kernel as shown in Figure 3.6. Please note that in the following explanation, we assume the *instruction window size* is 3 (each sliding window contains three consecutive instructions).

1) Reuse outside of instruction window: The first instruction in Figure 3.6 (`ld.global` in line 2) loads the data from the global memory into `$r3`. Reuse of `$r3` occurs in the `set.ne` instruction in line 14. Since the first use of `$r3` is outside of the instruction window (please recall that window size is 3), the compiler liveness analysis marks `$r3` to be written back directly to the register file as there is no bypassing opportunity within the window containing `ld.glonal` instruction.[1] In this case, where the first reuse distance is greater than the window size, there is no need to write this value back to the

[1]Further compiler optimizations to reorder instructions to increase bypassing opportunities are possible but we did not pursue this opportunity in the current version of our implementation.

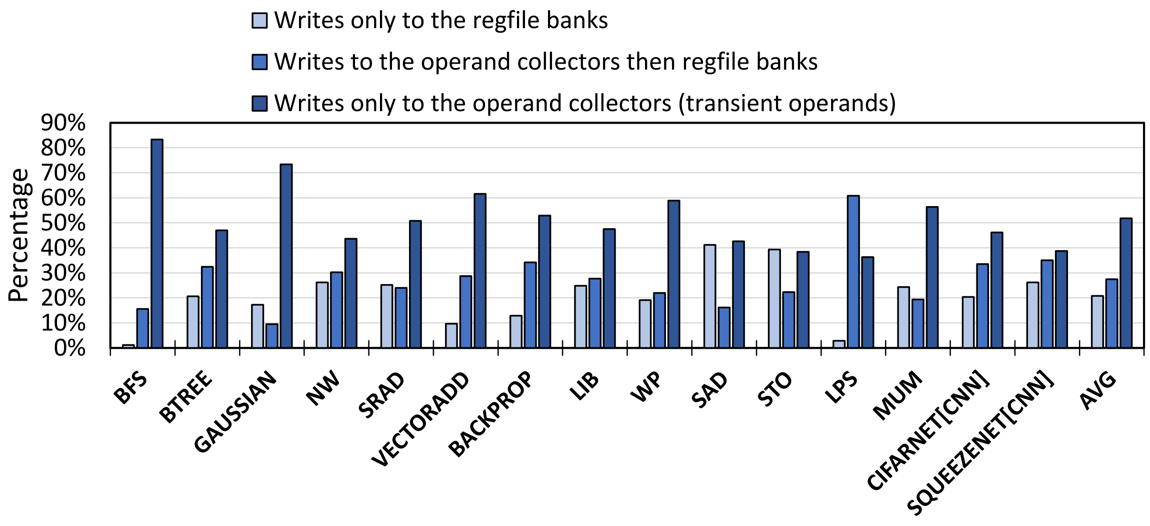


Figure 3.7: Distribution of write destinations in BOW-WR

bypassing operand collector. Figure 3.7 shows the breakdown of the instruction writes into the three categories that we are in the process of explaining. The leftmost bar represents this case where reuse is outside the instruction window, and occurs on average, in 21% of the computed operands. In this case, writing these operands to the BOC is unecessary.

2) Reuse inside of instruction window: Operand reuse occurs in this case as a produced value is consumed again in the window. For example, `mov` instruction in line 3 of Figure 3.6 writes into `$r2`, where the immediate reuse of `$r2` happens in the next instruction (`mul` instruction in line 4). `$r2` will be used one more time in this window by the third (last) instruction of this window (`mad` in line 5). In this scenario, by writing `$r2` into the bypassing operand collector, we can directly forward it to the next instructions. Later on, the `mad` instruction in line 7 will also read `$r2`, which can be forwarded from previous `mad` instruction (in line 5) as they fall within the same window. As another example, the `add` instruction in line 10 writes into `$r1`. Later on, this register will be accessed by the next instruction (`ld.global` in line 11) as a source operand. We can directly forward the value from `add` to `ld.global` by storing it in the bypassing operand collector. However, this is not the end of lifetime for `$r1` since it will be used later in `set.ne` instruction in line 14 (please note that the value could not be forwarded from `ld.global` to `set.ne` as they do not belong to the same instruction window).

In the case where there is reuse of the value in the active window, we obviously would like to write it to the BOC to enable this reuse. However, these reusable cases break into two categories based on whether we will eventually need to save the register back to the RF or not (avoiding the write altogether). We describe these two cases next.

(a) Reuse of a transient operand: In some instances, a value produced by an instruction will be reused only while it resides in the bypassing operand collector. As a result, the value's lifetime does not exceed the instruction window, meaning that there is no need for write backs to the register file bank once the instruction slides out of the instruction window. Lines 3, 4, 5, and 7 of Figure 3.6 show a case where an output value (register `$r2` in line 3) should be written back only to the BOC, because it is only used by future instructions *already within the same window* (or within the window of neighboring instructions). In this case, the write-back to the RF is bypassed since the `$r2` is a transient register value. In some cases, if this value came directly from memory or was produced by another instruction, we do not need to allocate a physical register in the register file for it. In this case, the immediate reuse distance across all the accesses is always less than IW. Figure 3.7 shows that these kinds of writes account for 52% of all operands. If indeed we can avoid allocating registers in many of these instances, we can further gain efficiency by reducing the effective size of the register file, allowing us for example to provision the GPU with smaller RF for the same performance, or gain performance by allowing additional Thread blocks for the same register file size.

(b) Reuse of a persistent operand: When a value is reused in the window, but continues to be live and will be reused later in the program, it must be saved back to the RF when it is evicted from the BOC. Lines 10, 11, and 14 of Figure 3.6 show such a case where the output value of the `add` (`$r1`) is going to be used within the same window (by the `ld.global` instruction), so it has to be written back to the corresponding BOC to take advantage of this bypassing opportunity. However, when the `add` operand is evicted from the BOC (to be replaced with the instructions in the next window), we need to write back `$r1` to the register file banks as well, as it is going to be used later by another instruction outside of the current window (which is `set.ne` in this example). More specifically, first read operation on register `$r1` will be simply bypassed within the BOC (as the value is being used immediately in the `ld.global` instruction). After this point, `$r1` is still alive but it is not going to be used within the same instruction window (distance is more than Instruction Window Size), so it has to be written back to the register file bank at the time of eviction for later use by the very bottom instruction. Figure 3.7 shows that 27% of all values fall in this category with window size of 3.

Identifying which of the three cases above to implement the most effective writeback option requires the ability to predict of how a register will be reused within and even outside the active register window. We elected to use compiler analysis to identify the type of each instruction writeback to provide hints to the architecture to identify the correct action for register writes. We use two additional flags in every instruction with a destination register, to indicate where the output data should be written. One bit is to enable writing to the BOCs, while the other bit is to enable writing back to the RF. Table 3.1 showcases the effectiveness of such compiler optimizations on the number of write operations on the register file. BOW with write-back policy is able to bypass a fraction of write operations. For example, the two consecutive writes into `$r1` in lines 4 and 5 of figure 3.6 could be bypassed as there `$r1` is being updated immediately by the next instruction. However, operands such as `$r2` in line 3 has to be written back to the register file as they slide out of the window, as there is no information on their future uses. Such redundant writes are avoided by the compiler annotations as `$r2` immediate reuse distance across all the accesses is always less than IW. Moreover, with the proposed compiler optimization, useless writes to the BOCs (about 21% of all write operations according to Figure 3.7) are avoided (for example, `$r3` in line 2 where it is not going to be used within the active window).

3.4.3 Reducing the Bypassing Storage Space

Thusfar, we have assumed that we provision each BOC conservatively with 4 registers for each instruction in an IW to account for the maximum possible storage required. The total size of a single bypassing operand collector will be $4 \times 128B \times 3$, or $1.5KB$ which is four times larger than an operand collector in our baseline architecture (which is $3 \times 128B = 384B$). However, we believe that these structures' occupancy likely to be low for the following three reasons.

- There are substantial bypassing opportunities within a window (nearly 60% with IW 3 as we saw in Figure 3.3). Only a single value of a reused register is stored and shared as it is forwarded to reusing insturctions.
- In the NVIDIA ISA, most instructions do not require three source registers. As shown in Figure 3.8, on average only 2% of the instructions need all three entries in the operand collector unit. For some applications such as BFS, BTREE, and LPS, no instructions with three register source operands are used. Please note that $OCUoccupancy = 0$ corresponds to those instructions that do not have any *register source operand* (such as `NOP` and `RET` with no source operand, or `SSY` and `BRA` with immediate operands).
- With the compiler optimizations, a considerable fraction of computed values (about 21% of total write operations) are not written back to the BOCs as they have no reuse within the window. In those situations, we do not use the entry for the destination register in the BOC.

To confirm our intuition, we analyze the occupancy of the conservatively sized operand collector (four entries per each instruction) for a window of three instructions. As Figure 3.9 shows, about half the benchmarks (for example, BFS, BTREE, and BACKPROP) never need more than half of the entries in each BOC. Even benchmarks with higher occupancy (like WP and SAD), do not use many of the available registers. On average, across all of our benchmarks, only 3% of the cycles require more than 50% of the available entries. There were no instances where the worst case scenario (all 12 entries occupied) arose in our experiments.

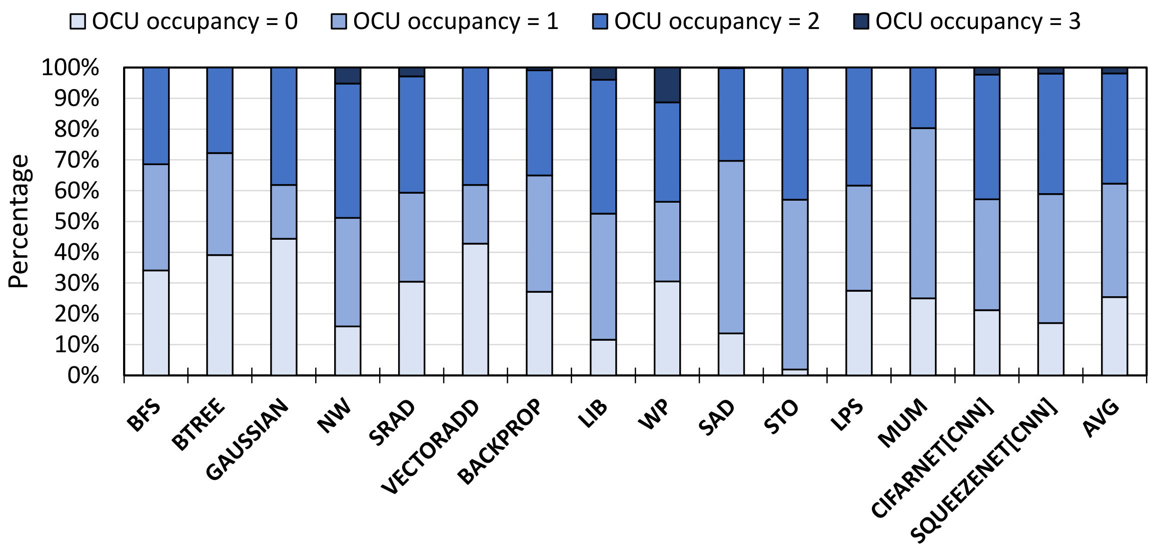


Figure 3.8: Operand collector units' occupancy.

Given this occupancy behavior, we reduce the buffer size in the BOC to half of the maximum possible size. As a result, there are situations where the occupancy is high and not all the operands within the window can be kept in the BOC, in which case we use a FIFO replacement policy to evict the oldest entry. We restrict the window size to the nominal window size (3 instructions in our example) and do not bypass instructions beyond the window size even if there is sufficient buffer space in the BOC. This allows us to simplify the compiler analysis and tag the writeback target in BOW-WR correctly in the compiler taking into account the available buffer size (without this simplifying assumption, an entry tagged for no writeback to the RF may need to be saved if it is evicted before all its reuse targets use it). In future work, we will consider enabling bypassing beyond the nominal window size limited only by the buffer space.

In worst case scenarios, sharing fewer number of entries in a BOC across multiple instructions may lead to an increase in the write-back traffic from BOC to the register file

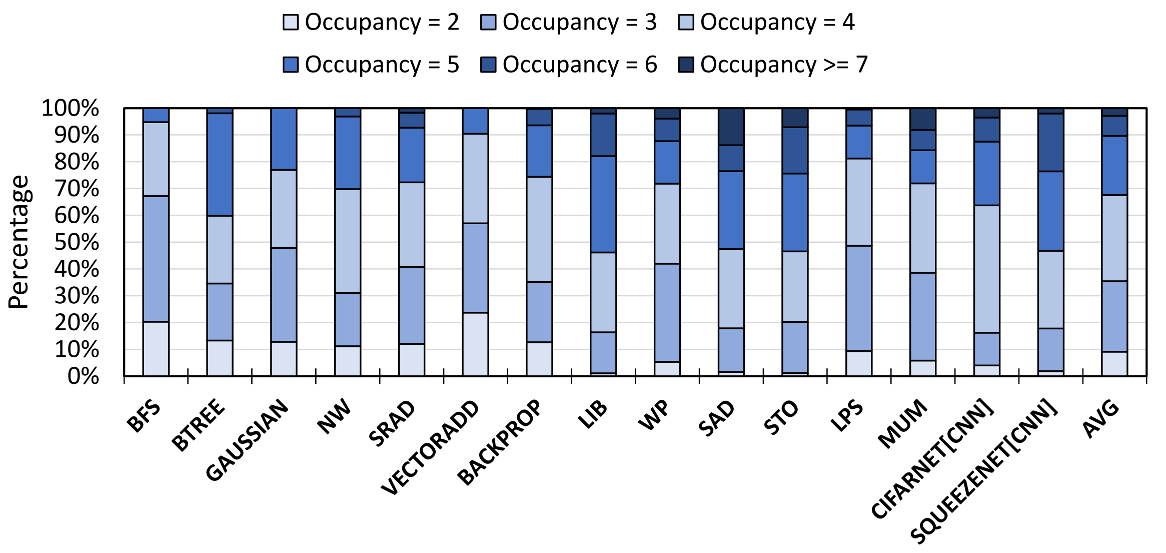


Figure 3.9: BOC occupancy with a window 3: half of the entries are unused.

due to space limitation, which in turn can potentially hurt performance and energy efficiency of BOW-WR. However, given that this happens only 3% of the time across our benchmarks (Figure 3.9), this effect is small. At the same time, reducing the size of the buffering from 12 to 6 entries per BOC means that the storage overhead in the BOC was reduced from 4x to only 2x that of the baseline GPUs.

3.5 Evaluation

We use GPGPU-Sim [17] which models an NVIDIA TITAN X Pascal (GP102) configuration [71] shown in Table 3.2. Benchmarks have been selected from the Rodinia [31], ISPASS [4], Parboil [125], and CUDA SDK [99] (see Table 3.3).

# of SMs	56
# of cores per SM	128
Max # of TBs/Warps/Threads per SM	16/32/1024
Register File size per SM	256KB
L1 Cache/Shared Memory Size per SM	48KB/96KB
L2 Cache Size	3MB
Warp Scheduling Policy	GTO

Table 3.2: Nvidia TITAN X (Pascal Arch.) Configuration

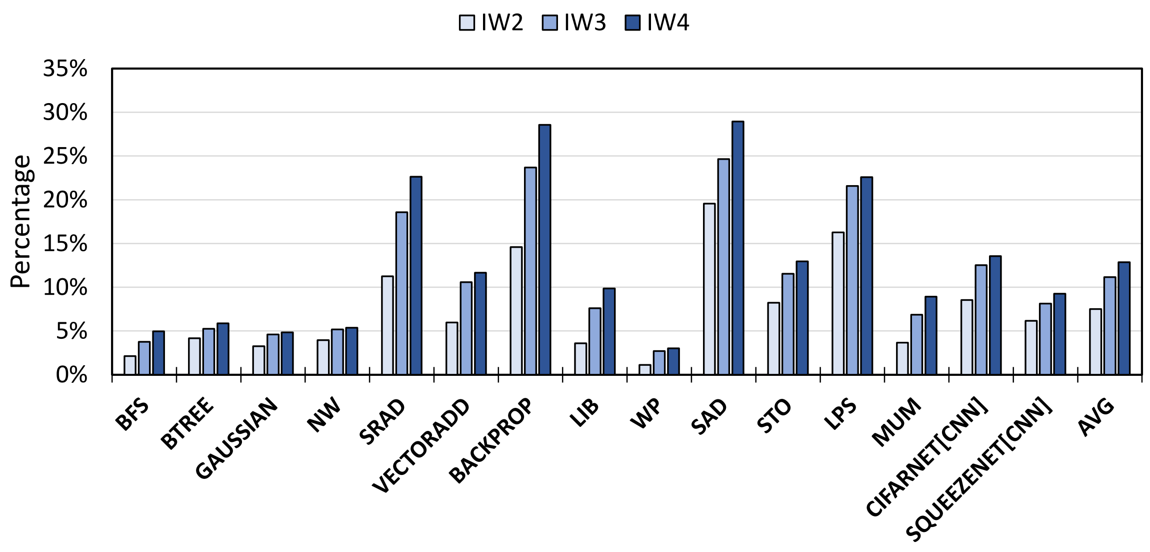
3.5.1 Performance/Energy Evaluation

Performance: Figure 3.10 displays the normalized IPC improvement achieved by BOW and BOW-WR compared to the baseline, using different instruction windows. As a result of bypassing substantial amount of read and write operations, port contention decreases (on both register file banks as well as BOCs), leading to better performance. Notably, we observe IPC improvement across all benchmarks. BOW-WR achieves marginally better performance due to its ability to reduce considerable amount of write operations, while BOW's improvement comes from bypassing the read operations. On average, with a window of three instructions, BOW and BOW-WR can improve the IPC by 11% and 13%, respectively. The small magnitude of advantage in IPC for BOW-WR over BOW is not surprising since writes are not on the critical path of instructions; however, as we will see later, the advantage of BOW-WR is higher in terms of energy savings. As *IW* grows beyond 3, we observe

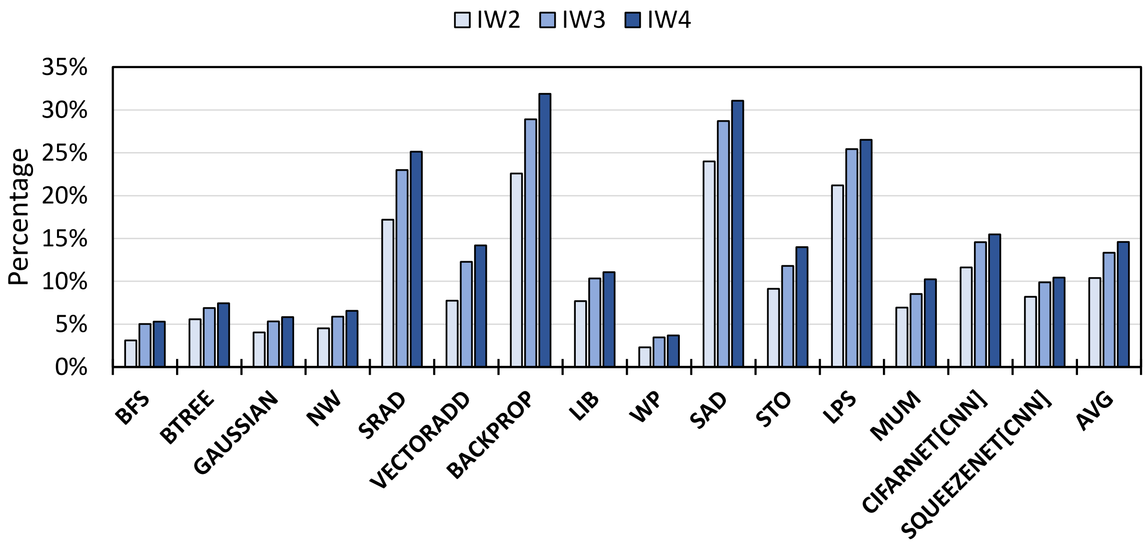
Suite	Bench. Name	Description
ISPASS [4]	LIB	LIBOR Monte Carlo
	LPS	3D Laplace solver
	STO	StoreGPU
	WP	Weather prediction
Rodinia [31]	BackProp	Back-propogation
	BFS	Breadth first search
	BTree	Braided B+ Tree
	Gaussian	Gaussian elimination
	MUM	MummerGPU (Sequencing)
	NW	Needleman-Wunsch
	SRAD	Speckle Reducing Anisotropic Diffusion
Tango [68]	CifarNet	CifarNet NN
	SqueezeNet	SqueezeNet NN
CUDA SDK [99]	VectorAdd	Vector-Vector Addition
Parboil [125]	SAD	Sum of Absolute Differences

Table 3.3: List of used benchmarks

diminishing returns in the IPC improvements. Operand bypassing is more effective on register-sensitive applications (such as SAD). In contrast, benchmarks such as WP with lower register usage and fewer operand reuse opportunities gain little performance.



(a) BOW



(b) BOW-WR

Figure 3.10: IPC improvement.

To evaluate the impact of reducing the storage size in the BOCs, we reduce the bypassing storage space by half (assuming window size of 3, each BOC has six entries instead of twelve, which are shared across every three consecutive instructions). Figure 3.11 shows

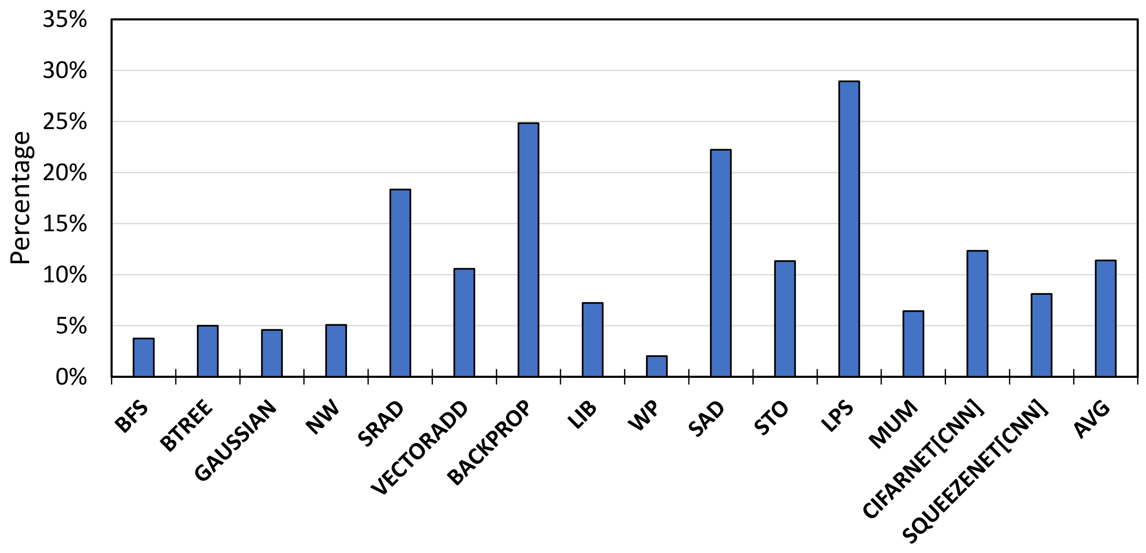


Figure 3.11: IPC increase with 6-entry BOC (half-size).

the performance effect of this space optimization. While most of the benchmarks with lower BOC occupancy sustain their performance improvement under this space constraint, the IPC improvement slightly degraded for benchmarks with higher BOC occupancy such as SAD. On average, we have observed a 2% performance loss with half bypassing storage; we still obtain nearly 11% IPC improvement even with half the storage size.

Figure 3.12 shows the normalized time that each application spends in the operand collection stage. The OC residence time is reduced significantly when $IW = 2$ and 3 (by about 60% in the latter case). However, we do not see substantial additional benefit with larger windows. This result demonstrates that BOW is able to successfully find most of the reuse opportunities and reduce the OC residency in a consistent way for all applications. In most cases, the more time the application spends in the OC stage in the baseline case, the more benefit it can get from operand bypassing. However, we do not see that this

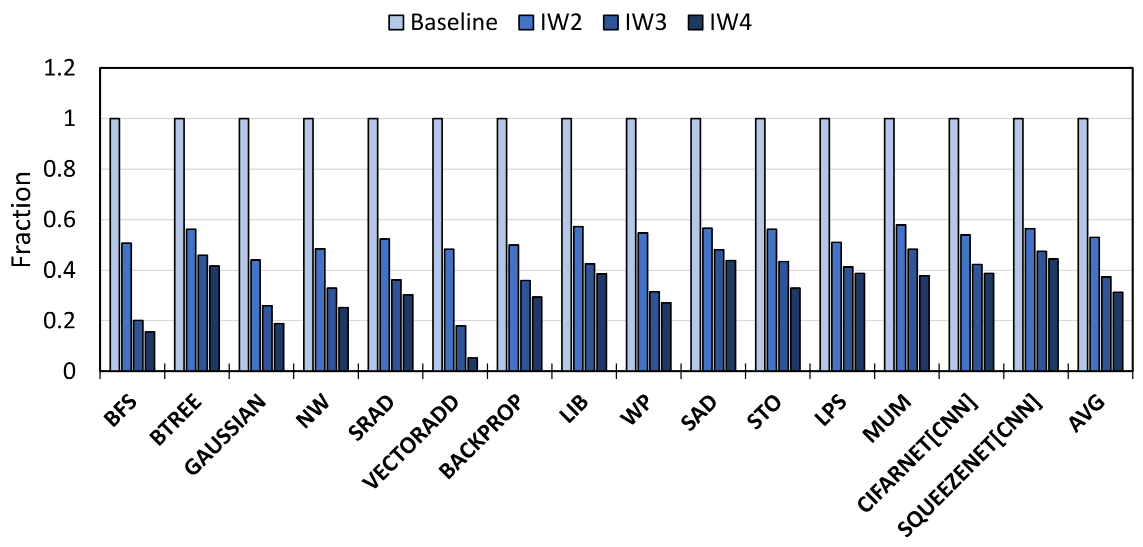
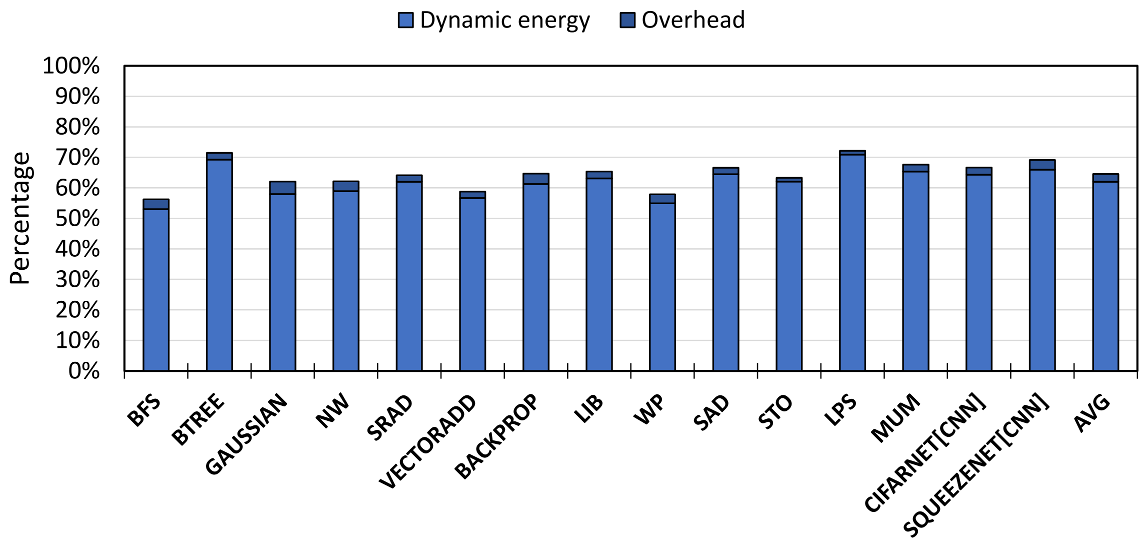


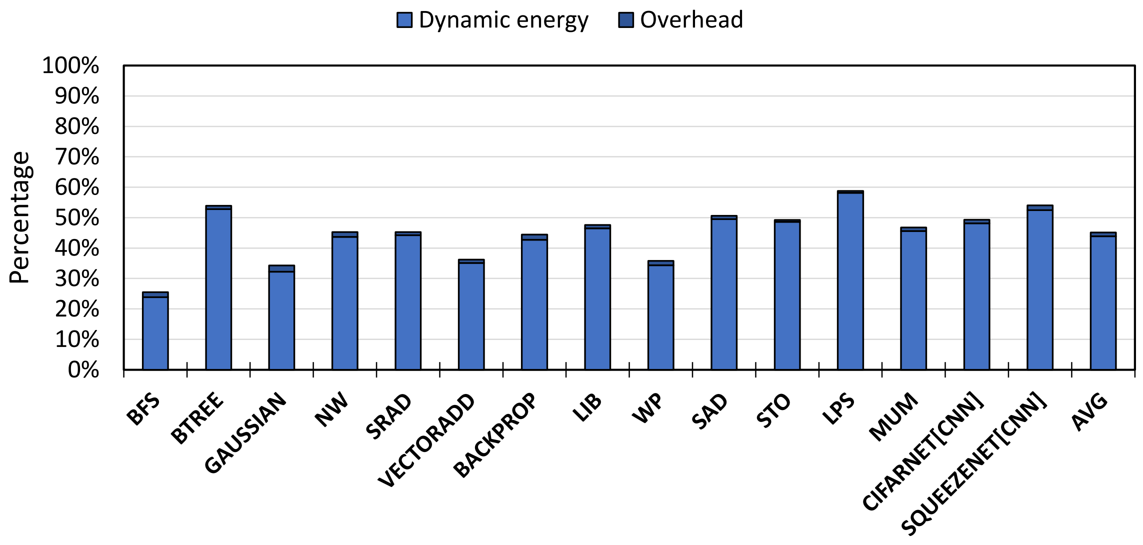
Figure 3.12: Cycles spent in OC stage for different window sizes (normalized to the baseline).

hypothesis hold across all applications since the impact on IPC varies because the application performance may be bound by other stages of the pipeline.

RF Energy: Figure 3.13 shows the dynamic energy normalized to the baseline GPU for BOW and BOW-WR respectively. The small segments on top of each bar represent the overheads of the structures added by BOW. Dynamic energy savings in Figure 3.13 are due to the reduced number of accesses to the register file as BOW and BOW-WR shield the RF from unnecessary read and write operations. Specifically, BOWwith a window size of 3 instructions reduces RF dynamic energy consumption by 36%, after considering the 3% increase in overheads. BOW-WR is even more successful in saving dynamic energy because it also avoids substantial amount of write operations to the RF. We observed 55% reduction to the overall dynamic energy in BOW-WR, after considering 1.8% increase in overhead. Note that even the overhead of BOW-WR is lower since the eliminated writes also consume



(a) BOW



(b) BOW-WR

Figure 3.13: Normalized RF dynamic energy

overhead through the additional BOW structures. The source of dynamic energy overhead include the accesses to the BOCs as well as the modified interconnect. It is interesting that although the IPC impact of BOWvaries across applications, the advantage in energy saving is relatively consistent: shielding the RF from operations reduces dynamic energy.

Comparison to Register File Cachine: Similar to BOW, there have been attempts to reduce the RF dynamic energy by caching the outputs in a smaller cache structure [45], a technique called register file cache (RFC). RFC was proposed in conjunction with liveness information provided by the compiler and used by a two-level warp scheduler to reduce the energy consumption by 36%. Comparing RFC to BOW, there are at least two major differences:

- Register File Cache (RFC) is organized like the original RF, but only smaller and closer to the operand collectors. Hence, it does not resolve the port contention issues. In contrast, BOW resolves this by distributing the cache across operand collectors.
- In RFC, all computed results are written back to the cache, regardless of whether or not they will be used in near future. This leads to redundant cache write backs which are avoided in BOW-WR through compiler hints.

We implemented RFC with 6 entries per each thread (with 32 threads in each warp and 32 warps per SM), and on average, observed less than 2% performance improvement. Note that RFC overhead in this configuration is 24KB, which is double than that of BOW-WR with space optimization. BOW-WR also saves substantially more power by consolidate writes.

Hardware Overhead: The largest additional structure in BOW is the widened bypassing operand collectors. In baseline BOW with an $IW = 3$ configuration, each BOC holds 12 entries, each 128B wide (1.5KB in total), while in our baseline architecture, each operand collector is 384B. This adds around 36KB of temporary storage across all BOCs, which is significant (but still only around 14% of the total register file). However, we have showed that this choice is highly conservative. BOW-WR is able to sustain most of the performance improvement with half-sized buffer storage, which adds 12KB of temporary storage across all BOCs (4% of the total register file size). We calculate the BOC (in baseline BOW) and register file power consumption using CACTI v7.0 [24] and report them in Table 3.4.

Parameter	BOC	Register bank	Percentage
Size	1.5KB	64KB	2%
Vdd	0.96V	0.96V	-
Access energy	2.72pJ	185.26pJ	1.4%
Leakage power	1.11mW	111.84mW	0.9%

Table 3.4: BOC overheads in 28nm technology

To analyze the hardware overhead, we modeled a network consisting of the 32x32 crossbar, BOCs, bus arbiters, and the bus in RTL using Verilog and synthesized it in 28nm technology using Synopsys Design Compiler. The power and area of the register banks were modeled in CACTI. The design comfortably meets the timing constraint for 1GHz clock. The total energy for the redesigned BOC network is 33.2mW assuming a 50% of the cycles write. This is small compared to the 2.5W power of the whole register bank. The overall area of the added circuitry is less than $0.04mm^2$ compared to the $1.72mm^2$ size of a register bank: the additional network area is less than 3% of the area of a register bank, and less than 0.1% the area of the full RF.

3.6 Related Work

Energy efficiency of GPUs has been an area of increasing importance [6,7,9–12,30, 44,70,83,84,88,91,114,116–118,131,140–142]. These prior works have explored improving the performance or energy efficiency of GPU register files in a number of ways.

GPU Register File Scalability: Since the number of live registers in proportion to the total number of registers is relatively small, there are also trends to compress, reduce, and even replace the register file altogether so as to save power used for registers that would never need be utilized in most applications. Jeon et al [62] introduced a method to virtualize the RF addressing, which would leverage the variations in the instructions being executed by different warps, and allow the dead registers from one warp to be renamed and reallocated to another. As a result, this approach effectively shares the same physical registers with multiple warps during the course of a kernel run, with the performance suffering little to no loss with even the RF cut down to 50% of its size. However, it incurs a significant overhead. On a similar note, motivated by the low ratio of live registers for a CTA to its allocated registers, Oh et al [105] proposed FineReg, a new GPU architecture which would enable running more concurrent CTAs, increasing the RF utilization and the overall performance. It featured a RF divided into active and pending regions, with running CTAs using the active region, and stalled CTAs being moved to the pending region, allowing a new CTA to be run in the active region. RegMutex [73] improved performance by sharing a subset of physical registers between warps during the GPU kernel execution. However, it substantially increases the dynamic energy due to higher warp occupancy on the SM. RegLess [78] replaces the register file with a smaller staging unit with the help of compiler annotations, leveraging the short-lived and long-lived behaviors of the register. RegLess achieves lower power and smaller register storage size while maintaining performance. The Latency-Tolerant Register File (LTRF) [115] uses compiler-analysis to identify registers to move into a register cache, which enables tolerance of large register files. However, this higher performance comes at the cost of a larger, more power-hungry register file. Compared to these solutions, BOW is the only solution that improves both the energy consumption *and* the performance of the RF.

Register File Compression: Some of the state-of-the-art have targeted narrow register values, i.e. values which use less than half of their allocated register. Not only do they cause unnecessary energy leakage due to the storage of unused bits, but also there will also be additional delay when accessing those elements, since they are treated as a wide register when accessed, regardless of the actual value. With compiler optimizations, if narrow values are identified, two of them can be squeezed into a physical register, which leads to more RF utilization and less overall energy consumption. Wang et al [139] proposed a method to pack narrow values in the GPU register file in a greedy fashion, achieving an average speedup of 18%. On top of register packing, Esfeden et al [15] also proposed to coalesce register file accesses in order to reduce the number of RF accesses, thereby reducing dynamic energy consumption by 17% as well as improving the performance by coalescing multiple register read operations into a single physical access. Voitsechov et al [137] aimed to increase GPU thread occupancy by identifying the last register usages within the code and releasing them at those points in the kernel for the usage of other threads. Combined with techniques for more efficient allocation for scalar and narrow values (de-duplication and packing, respectively), their method achieved an average of 12% speedup on register-bound workloads.

Related CPU register file optimizations: Register file efficiency has been an ongoing research topic in the CPU industry as well, having started long before the era of GPGPUs. Balkan et al [25] observed that a substantial fraction of computed values in a typical superscalar datapath are transient. They proposed a microarchitectural technique which predicts transient registers and avoids register allocations for predicted transient values. BOW also identifies and eliminates transient registers but does so using compiler hints. Park et al [106] predicted the bypassing opportunities to reduce the energy consumption, enabling register banks to be designed with lower number of ports. Swensen et al [126] addressed the issue that larger register sets have a longer access time than a smaller one, and based on this, proposed a hierarchical register set that contained close, middle and distant register sets, which would be used for different scenarios based on their time criticality, e.g. more critical tasks would be performed on the smaller, but closer, register set. There have also been other works that mainly target the access latency of CPU register sets [23,38,65,98,147]. The nature of RFs for superscalar CPUs is substantially different from those of GPUs, and therefore these techniques do not directly carry over to GPUs.

Chapter 4

RegMutex: Inter-Warp GPU Register Time-Sharing

4.1 Introduction

Registers are the fastest available memory to the threads in a machine executing a program. Register are being kept in-core closely coupled with the ALUs and usually are the most expensive form of memory (per-byte) in a machine. The set of registers for a processor are packed in a structure called *register file*. In GPUs, in order to enable concurrent residence of thousands of threads for massive Thread-Level Parallelism (TLP), the architecture employs a very large SRAM storage structure as the register file. A considerable fraction of die area and chip power has to be dedicated to this structure [81, 130].

Nonetheless the allocation of physical registers to architected registers in the kernel binary is static, i.e., the maximum number of live registers at any given point determines the kernel's physical register demand, and is exclusive, i.e., a warp's physical registers are solely its own for the lifetime of the thread-block containing the warp. This allocation scheme carves a portion of the physical registers for the warp regardless of the fluctuations in the register usage by the warp. In other words, even if all the requested registers are live only for a few instructions, the hardware reserves all the allocated physical registers for the warp, thereby making them inaccessible by other warps. This results in underutilization of a large portion of the register file during GPU kernel execution and hence ignoring the potential performance gain opportunity.

A number of solutions have been proposed to remedy this issue. Most notably, Jeon *et al.* [62] built on the Register Renaming Table (RRT) idea from the CPU realm to proactively map architected registers to physical registers on-demand. However, this solution, as well as other work that suggest fundamental modifications to the structure of the register file and its allocation mechanism [63, 132], impose significant hardware overheads. In addition, proposals such as the work of Jatala *et al.* [59] fail to address fluctuations in warp register demand during kernel execution, hence lack general applicability.

In this work, we present RegMutex (**Reg**ister **Mut**ual **Ex**clusion), a synergistic compiler-microarchitecture design that enables efficient register time-multiplexing between warps. In RegMutex, a subset of the architected registers are allocated on-demand as-a-whole and deallocated upon no demand. RegMutex utilizes the information gathered by compiler analysis to instruct the hardware for physical register allocation and deallocation. At compile time, RegMutex separates the group of kernel architected registers into a base register set and an extended register set. Using live-register analysis, the compiler determines the locations within the kernel where the number of live registers exceeds the size of the base register set and marks them as *acquire* points. Similarly, program points where the number of live registers falls equal to or below the size of the base register set are marked as *release* points. On the hardware side, the physical registers are allocated for the base register set whenever the warp is resident. The extended register set, on the other hand, is allocated physical registers only when the warp reaches an acquire point, and deallocated once the warp faces a release point. The extended register sets of all warps are allocated out of a communal pool of registers that is shared by all hardware-resident warps (the *shared register pool*).

RegMutex diminishes the pressure on the register file by eliminating the necessity of register file size accommodation with the maximum number of live registers at any point in the kernel. The warps proceed in the program as usual and will be blocked only when a large number of them have acquired the extended register set. A blocked warp will resume execution by acquiring the extended set as soon as one of other warps releases the shared resource. Essentially, the benefit of RegMutex can be viewed from two perspectives. First, RegMutex allows GPU programs to sustain approximately the same performance with a smaller hardware register file. Second, for programs that incur low SM occupancy due to excessive register usage, our technique enables higher number of concurrent warps to be resident in the hardware via sharing their register allocations with each other, leading to a superior performance. In other words, if a warp asks for a large number of architected registers, it can now co-reside with more warps on the SM. This chapter makes the following contributions:

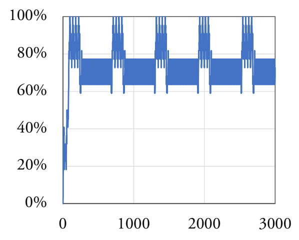
- We present RegMutex, a coordinated compiler-microar-chitecture technique as a remedy for GPU register file underutilization due to static and exclusive physical register allocation.
- We describe the RegMutex compiler and microarchitecture support schemes and show that this synergistic design introduces much lower (less than 2%) hardware storage overhead compared to existing solutions.
- We analyze the effectiveness of our solution by implementing it in the GPGPU-Sim simulation framework. We show that RegMutex enhances the performance of kernels for which the occupancy is limited by high register demand, and makes the application performance resilient on architectures that supply small register files.

The rest of this chapter is organized as follows. Section 4.2 expresses the motivation for a GPU register sharing approach. Section 4.3 describes our solution, elaborating RegMutex's compiler and micro-architectural design. Section 4.4 presents the experimental evaluations, and Sections 4.5 summarizes related work.

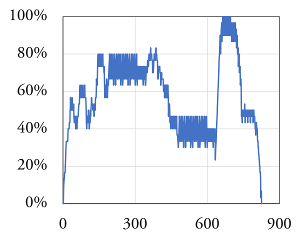
4.2 Motivation

A program, in its closest-to-machine language form, works with a set of registers. This set of registers are referred to as *architected registers*, and will be mapped to physical registers by the processor's hardware. To map architected registers to physical registers, CPUs utilize a mechanism called register renaming and a table called Register Rename Table (RRT). GPUs, on the other hand, use a simpler method for this purpose [60]. The mapping allows a simple $Y = X + B$ equation for each SIMD group (warp) to calculate its physical register indices where B is the base address of the block of registers assigned at run time to the specific warp, X is the architected register index (i.e., the offset into the block of registers), and Y gives the physical register index. This simple mapping avoids the overhead of performing register renaming for thousands of concurrently running threads. The set of physical registers is statically reserved for the warp's life-time (i.e., B is constant for the duration of the warp's execution), and becomes available for other threads only after the CTA (Cooperative Thread Array) to which the warp belongs retires [75].

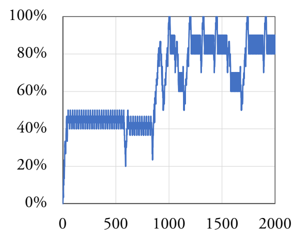
One important drawback of the above scheme, especially compared to RRT, is physical register underutilization during kernel execution. The static reservation is conservative in a sense that it requests for the *maximum* number of registers that are alive at any point in the GPU program. However, during a GPU program execution by the warp, not all the reserved physical registers are alive at all times. In fact, the time interval in which all the requested physical registers are utilized may only be a small fraction of the kernel execution time. This is particularly true for GPU applications containing nested loops in which register consumption increases within inner loops. Figure 4.1 illustrates this claim by showing the percentage of live registers with respect to the allocated registers during the program execution for a sample thread and six GPU kernels. Here we define a register *live* if its value is used in later instructions. It is evident from the plots that for the majority of the program execution only subsets of the requested registers are alive, and therefore, a large portion of the thread's allocated registers remain unutilized. Figure 4.1 also shows that register utilization may fluctuate constantly due to the GPU code shape.



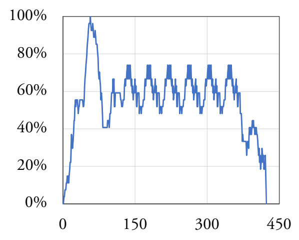
(a) CUTCP.



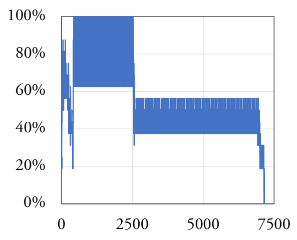
(b) DWT2D.



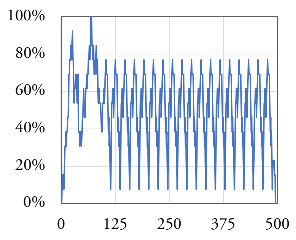
(c) Heartwall.



(d) HotSpot3D.



(e) ParticleFilter.



(f) SAD.

Figure 4.1: The utilization of a sample thread's allocated register set during kernel execution. X axis shows the number of instructions executed by the thread and Y axis shows the percentage of live registers with respect to allocated registers. Results are extracted using our extension to *GPGPU-Sim* [17]. Applications are from Rodinia [31] and Parboil [124].

Another drawback of the aforementioned scheme is limiting the occupancy for GPU programs (kernels) with threads that require a high number of architected registers. Occupancy is described as the ratio of the number of warps residing on the Streaming Multiprocessor (SM) over the maximum number of warps that warp schedulers in the SM allow for residency. For example, on Nvidia Volta GPUs, there can be up to 64 warps residing on an SM [3]. The higher the occupancy, the larger the number of candidate warps to be executed by the SM at any given time. This enables the GPU cores to cover memory access latency more effectively through having more concurrent warps (note that higher occupancy does not necessarily lead to better performance [138] due to possible side-effects such as cache pollution, but a low occupancy can cause resource underutilization). A warp that requires a high number of registers lowers the SIMD occupancy. It essentially disallows co-residency of other warps due to register file resource limitations, however the warp may need excessive registers only for a short period of the GPU program. In summary, these two drawbacks are two different faces of the same coin: registers are statically and exclusively reserved but not all of them are utilized at the same time.

Jatala *et al.* [59] propose a register sharing technique in which a few warps share the set of registers having higher-than-a-certain-threshold architected index. They propose a hardware lock for a pair of warps to acquire. The first warp that asks for a shared register acquires the lock and disallows the execution of its pair until it reaches the end of program. The main shortcoming of this solution is the one-time acquire with no in-kernel release. In other words, the warp that gets the ownership of the shared registers will not release it until the warp is finished. In addition, the solution requires hardware modification at the register file level for the accesses. For each register access, up to three conditions have to be verified (if the warp is shared, if the register is shared, if the lock is already acquired). The warp will have to loop inside these conditions if the answer to all of them is positive.

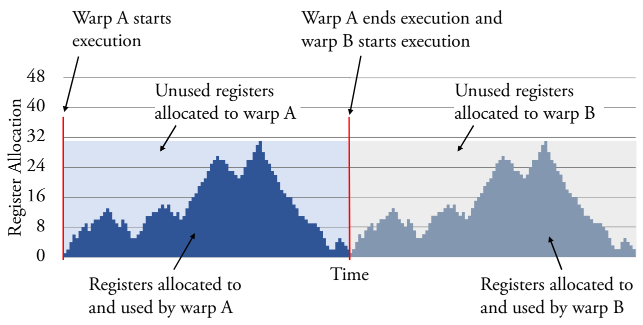
Jeon *et al.* [62] suggest virtualizing the register file to share the physical registers between the warps. They suggest embedding the dead or liveness information of the architected registers into the source code by the compiler, and using a Register Renaming Table (RRT) inside the hardware to proactively release dead registers from one warp and re-allocate them to a different warp. In other words, they borrow the idea of RRT from CPUs and implement it for the GPUs. The final outcome of this scheme is a smaller register file and reduced power consumption at the unignorable expense of higher hardware complexity. Jing *et al.* [63] take an even more drastic measure; they emulate the behavior of the register file using a cache by combining the register file and SM-private L1 cache. These solutions necessitate heavy hardware modifications such as RRT, Release Flag Cache as well as adding required support in the fetch stage of the GPU pipeline.

These drawbacks motivate the need for an inter-warp register sharing approach that introduces low hardware complexity while, simultaneously, being effective at reducing the underutilization of the register file.

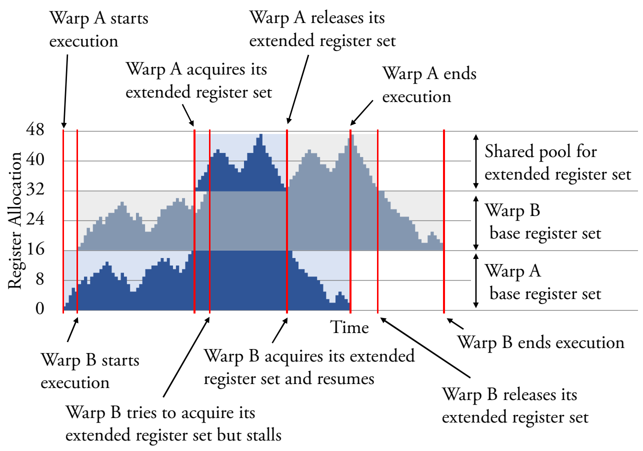
4.3 RegMutex: Inter-Warp Register Time-Sharing

In this section, we propose RegMutex, an effective approach to remedy GPU register file underutilization. RegMutex time-multiplexes the allocation of a subset of registers required by the kernel between multiple warps. During the execution of the kernel, when a warp is at a program point in which it does not work with any of the registers in this subset, its execution progresses normally. However, when the warp needs this subset, those registers need to be obtained from a shared register pool for the warp. More formally, RegMutex divides the architected register set into base register set B_s and extended register set E_s. B_s is assigned to physical registers in the register file as soon as the warp resides in the SM, similar to what we observe in existing hardware. On the other hand, E_s is allocated to register files only when the program requires more live registers than $|B_s|$; and also E_s is de-allocated right after the number of live registers in the kernel becomes equal to or less than $|B_s|$. The communality of the shared register pool enables on-demand register allocation for segments of the GPU kernel where the number of live registers increases.

When a warp is launched for execution on the hardware, while the B_s physical register assignment is instantaneous and lasts for the duration of warp execution, the allocation of E_s is controlled via compiler-generated instructions that enforce an *acquire-release* semantics. The compiler identifies the code segments in the program where the number of live registers exceeds $|B_s|$. Right before entering each such code segment, the compiler inserts an instruction that *acquires* E_s from the Shared Register Pool (SRP). And immediately after each such code segment, the compiler inserts a *release* instruction to release its E_s back to SRP. For an acquire, if the currently unused registers in SRP is insufficient to satisfy the acquire instruction for the extended registers, the warp has to wait for a release by another warp in SM to free up shared registers. In this case, the warp stalls and only becomes eligible for execution once sufficient shared registers have been freed up. In summary, the compiler drives the warps within SM to time-multiplex the pool of shared



(a) **Baseline execution without RegMutex.**



(b) **Execution using RegMutex.**

Figure 4.2: Example of two warps A and B executing identical code with and without RegMutex. Base register set size is 16 registers (per thread) as well as the Shared Register Pool (SRP) size. The architecture is assumed to have 48 hardware registers per thread.

hardware registers.

Figure 4.2 shows a simplified, illustrative example of a case akin to typical GPU execution, where multiple warps execute the same code. Each warp has a maximum register requirement of 31 registers per thread in the example. As shown in Figure 4.2(a), a baseline architecture without RegMutex reserves 31 registers per thread for the full duration of the execution of each warp, preventing any overlapping of execution of the two warps (as the combined register use of the two warps, at 62, exceeds the 48 available hardware registers per thread). Figure 4.2(b) shows an execution configuration using RegMutex where base register sets of 16 registers per thread each and extended register sets of 16 registers per thread each are utilized by the two warps. Here, the code regions that only require the base register sets can execute in parallel, serializing only the portions that require use of the extended register sets thus enhancing the execution time. Note that in this example, for simplicity, we assume the registers are the only hardware resource constraint.

RegMutex allows a warp to acquire and release its E_s as many times as needed during its lifetime. Here we enforce a fixed $|E_s|$ for the acquires within a kernel. Also, nested acquire-release instructions are not permitted. In other words, an acquire after another acquire without an intervening release or a release after another release without an intervening acquire should have no effect. Both these assumptions keep the hardware complexity of the design low and enable flexible use of acquire or release within conditional code. To facilitate the discussion on our solution in this chapter, we assume that concurrent warps on an SM execute identical programs, which is the case for the majority of GPU applications.

In the rest of this section, we elaborate upon the compiler and architecture support for our technique.

4.3.1 Compiler Support

The compiler performs a number of methodical steps to support RegMutex:

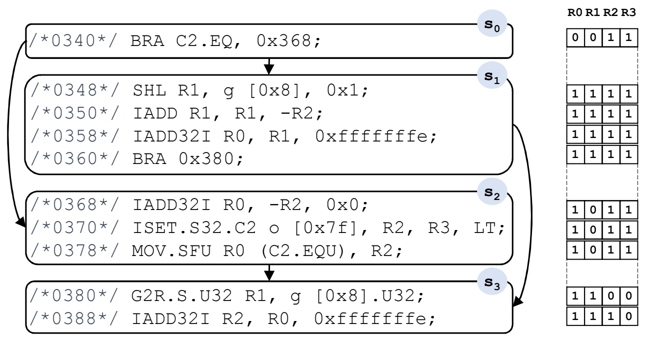
1. Register liveness analysis of the GPU assembly code to extract the register usage information.
2. Extended register set size determination.
3. Acquire/release primitive injection into the assembly code.
4. Architected register index compaction before and throughout the release state.

The first two steps analyze the kernel program and the latter two modify it. After these steps, the GPU kernel contains functionally the same code added only with extended register set acquire and release directives at proper locations. We now elaborate upon these steps.

Register Liveness Analysis

RegMutex relies on static (compile-time) register liveness information for setting the boundaries for extended register set use. Register liveness analysis helps our technique to recognize the program's register requirements at different instructions in order to instruct the executing microarchitecture for extended set acquire or release actions at appropriate locations.

As in [36], we define the static liveness for an architected register index as the set of (not necessarily consecutively placed) instructions at which the value previously written into the register has to be held intact since there is a non-zero probability that it will be read later. Figure 4.3 shows an example from a GPU program and the static liveness of registers. Within a basic block, if an architected register is written (*defined*) at an instruction and read (*used*) at some later instructions for the last time and without any intervening



```
/*0340*/ BRA C2.EQ, 0x368;

/*0348*/ SHL R1, g [0x8], 0x1;
/*0350*/ IADD R1, R1, -R2;
/*0358*/ IADD32I R0, R1, 0xfffffffe;
/*0360*/ BRA 0x380;

/*0368*/ IADD32I R0, -R2, 0x0;
/*0370*/ ISET.S32.C2 o [0x7f], R2, R3, LT;
/*0378*/ MOV.SFU R0 (C2.EQU), R2;

/*0380*/ G2R.S.U32 R1, g [0x8].U32;
/*0388*/ IADD32I R2, R0, 0xfffffffe;
```

Figure 4.3: A GPU code sample from DWT2D application and its static register liveness.

register definition, all the instructions between the definition point and the last use point are considered *live* for that particular register. Register R1 in basic block s_1 in Figure 4.3 is an example of this case.

However, in the presence of control flow divergence, liveness analysis is not straightforward because of the unavailability of path traversal information at compile time. If a register is defined before a branch and is used within at least one of the branched basic blocks, the register has to be considered alive within all the resulted branch basic blocks due to the uncertainty of the execution serialization by threads within the warp. In other words, the compiler has to be conservative in its assumptions. This makes the immediate post-dominator instruction of the branches the first candidate for considering the architected register *dead*. For instance, in Figure 4.3, although R3 is used within only s_2 it has to be considered alive throughout s_1 as well. Similarly, if a register is defined within a branch and is going to be used in a post-dominator basic block, it has to be assumed alive in other branches. The liveness status of Register R2 throughout basic block s_1 in Figure 4.3 is due to this observation.

RegMutex performs this analysis on architected registers in the GPU assembly program. The outcome of this step is a collection of Boolean vectors each representing the liveness of particular architected registers at particular instructions. We have visualized this in Figure 4.3 [1]. This information will be used in the next steps to determine the appropriate size for the extended register set as well as to inject the compiler-to-microarchitecture directives at appropriate program locations.

Extended Register Set Size Selection

Using static liveness information, the compiler now needs to determine the size of the extended register set, i.e., $|E_s|$. Note that $|E_s| + |B_s|$ is fixed and equals to the total number of registers the kernel asks for, hence, selecting either $|E_s|$ or $|B_s|$ enforces the value of the other. Also, note that increasing $|E_s|$ may have adversarial effects. On one hand, a large $|E_s|$ is expected to give higher occupancy by allowing more warps to be resident on the SM. On the other hand, the compiler has to mark larger sections of the program as being in an acquire state, thus warps execute more instructions while holding their extended set which may result in more contention over SRP sections during the run time.

As both the number of warps resident on SM and their scheduling freedom impact overall hardware utilization and performance, we use a simple yet methodical policy that achieves a desirable trade-off between improving physical register utilization and not curtailing scheduling freedom. After finding the baseline kernel's theoretical occupancy and the contribution of kernel register usage as a limiting factor, we select candidate values for $|E_s|$ from an empirically-derived set of $\{0.1, 0.15, 0.2, 0.25, 0.3, 0.35\}$ multiplied by the number

[1]A similar analysis and liveness representation are provided for CUDA application by the *nvdisasm* CUDA binary tool [2].

of registers used by the kernel. Then we keep the even numbers that result in the highest occupancy calculated only with the base set size. If multiple candidate elements for $|E_s|$ give the same theoretical occupancy, we go with the largest element that possibly results in concurrent progress of more than half the warps in the current occupancy in the acquire mode. Let us illustrate these steps using an example. Assume the kernel asks for 24 registers to run on our baseline (Nvidia Fermi architecture) which supports up to 20 registers per thread without limiting the occupancy. Let us also assume that register usage is the only factor limiting the theoretical occupancy here. Based on our approach, the candidate set for $|E_s|$ consists of the even numbers in $\lfloor 24 \odot \{0.1, 0.15, 0.2, 0.25, 0.3, 0.35\} \rfloor$ ($\odot$ is the element-wise product) which yields $\{2, 4, 6, 8\}$. From this set, 4, 6, and 8 result in $|B_s|$ equal to 20, 18, and 16 respectively which have full occupancy for only the base set. Assuming maximum number of warps per SM is 48 and the total number of registers in the register file is $32K$, these configurations leave 16, 26, and 32 sections for SRP which indicate the number of warps that can acquire E_s concurrently. RegMutex selects $|E_s| = 6$ since it is the largest candidate that allows more than half of the warps on the SM (in this example 26) in the calculated occupancy be in the acquire state.

After determining the number of registers in the extended set, the new theoretical occupancy of the kernel is obtained using calculated $|B_s|$. This occupancy gives the number of CTAs that the SM can host, and also determines the total size for the SRP.

Deadlock Avoidance To avoid deadlocks in our design, two additional rules govern $|E_s|$ selection. First, the distribution of $|B_s|$ and $|E_s|$ has to be such that there are enough registers in the shared pool for at least one warp's E_s. This ensures that warps do not stall indefinitely for an acquire. Second, $|B_s|$ has to be greater than or equal to the number of live registers at any point in the program that CTA-wide synchronization primitives such as `__syncthreads()` exist. This avoids any deadlock due to inter-dependency of warps. In other words, while a warp W_a is waiting for another warp W_b to arrive at the synchronization PC, warp W_b will not wait at an acquire instruction for warp W_a to release its extended register set.

Acquire/Release Primitive Injection

After recognizing the regions within the program that use the extended register set, the compiler injects acquire and release primitives respectively at the beginning and the end of such regions. For RegMutex, we create an instruction to convey acquire or release information to the hardware. Unlike [62] RegMutex need not rely on meta-instructions since the content of this instruction is either a release or an acquire command.

Architected Register Index Compaction

To preserve the simple $Y = X + B$ equation for the architected to physical register assignment (different B's for extended and base register sets), architected register indices have to stay within their boundaries during the release state. The compiler must therefore ensure that none of the extended register set members contain live values when the extended register set is released. We essentially need a mechanism to compact the architected register indices for the duration the extended set is not acquired, and also right before releasing the extended register set.

To achieve this goal, the compiler may have to move any live values in the extended register set to available registers in the base set during the release state and before releasing the extended register set. For example, let us assume a scenario where the base register set size is 6 and a warp has a live register set $\{2, 4, 5, 9\}$ right before the release. Before releasing the extended set, the compiler has to move architected register 9 to one of 0, 1, or 3 locations. Note that the movement of architected registers is instrumented by the compiler (usually with `MOV` operations). This is similar to what [59] suggests under the name of *register declaration reordering* but it is different in that index compaction may happen multiple times right before each release by moving the architected registers, whereas in *register declaration reordering* the index minimization is limited to happen only once by reordering register declarations. In addition, the compiler has to apply register location renaming for all the uses of that particular register until the end of its current live range. We use a similar analysis to that done in Section 4.3.1 for those registers exceeding the boundary at release states.

We emphasize that the above compiler analysis steps, when embedded within a compiler, need to be applied during the last stages of the compilation chain. This is because the technique needs to know the architected register assignment whereas compiler middle-ends such as LLVM work with virtual registers in SSA form.

4.3.2 Architecture Support

In this section, we explain the architectural requirements to enable RegMutex. We used the baseline design offered by *GPGPU-sim* [17], a simplified depiction of which is shown at the top of Figure 4.4. After decoding acquire/release primitive at the decode stage as a barrier operation, the acquire or releases command is given to the issue stage. At this stage, the warp acquires the extended set or waits for an extended set to be freed, or releases its extended set. Upon a successful acquire, the information for the acquired SRP section as the extended set is passed to the Operand Collector Unit for register mapping. Below we elaborate upon the hardware implementation of RegMutex at these two stages.

Warp Issue Management Organization

In RegMutex a warp has to ask for physical registers for its extended set upon reaching an *acquire* instruction. If no physical extended set is available, the warp has to wait for another warp to *release* their set. The warp needs to essentially imitate the behavior of already existing barrier synchronization and communication in GPUs. CUDA barrier synchronization instructions allow one warp W_a to signal its arrival to another warp W_b, and to warp W_b to wait for warp W_a to arrive at a particular barrier. These instructions have been used for warp specialization purposes and implementing producer-consumer models [26, 27] via PTX instructions `bar.sync` and `bar.arrive`. For RegMutex, we exploit a similar design where warps have to wait for the release of an extended physical set when faced with *acquire* or signal the release of their own extended physical set upon a *release*. Since barrier operations in SM are executed at the issue stage, we design RegMutex's allocation logic closely coupled with it.

The highlighted section of Figure 4.4 shows the RegMutex's modification interacting mostly with the issue stage of the microarchitecture model. Because we need to keep track of each warp's execution mode (*acquired* or *not-acquired*), we use a single bit per warp to indicate the warp status. In the baseline model, each SM can host up to 48 resident warps

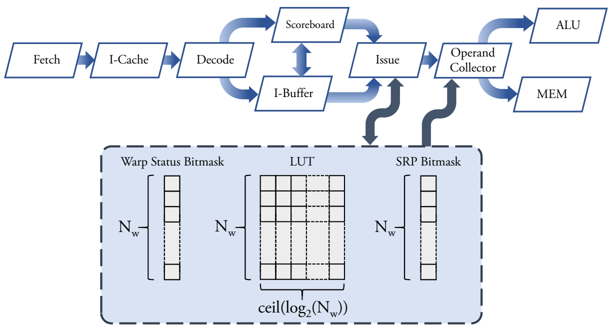
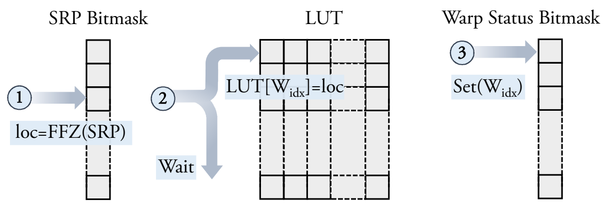


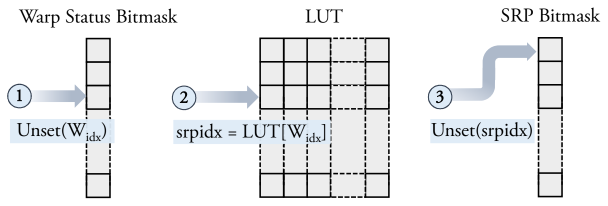
Figure 4.4: The baseline design from *GPGPU-sim* [17] (top) and RegMutex's added storage structures (bottom). Specified sizes are in bits.

($N_w = 48$) and therefore the warp status bitmask is 48 bits long. This bitmask is indexed by the warp index within the SM. In addition, another bitmask holds the status of sections of the Shared Register Pool (SRP). Since there can be up to N_w sections in the SRP, and since we disallow nested acquires and releases, the SRP bitmask is N_w bits long as well. Each bit in SRP bitmask indicates if a particular extended physical register set is acquired or not. The mapping between a warp and a bit in the SRP bitmask is performed via a lookup table (LUT in Figure 4.4). The table has one entry for each warp while each entry contains $\lceil \log_2 N_w \rceil$ bits indicating which one of the N_w SRP sections the warp has acquired (if warp's status bit is set). The total size for this table in our baseline is therefore $48 \times 6 = 288$ bits.

As depicted in Figure 4.5(a), when an acquire instruction reaches the issue stage, SRP bitmask is searched for an unset (zero) bit. This is equivalent of the Find First Zero (FFZ) operation on the SRP bitmask which returns the index of the least significant zero bit. If a valid index (for instance, $0 <= idx < 48$ in the baseline model) is returned, a section is available. Therefore, the index is written into the lookup table and the warp's status bit and the SRP availability bit are set. This index is then passed to the Operand



(a) Acquire.



(b) Release.

Figure 4.5: Acquire/release procedure implementation in RegMutex.

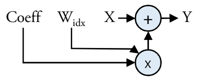
Collector Unit. However, if the returned index is invalid, the warp waits at the barrier and retries at later rounds when the warp gets scheduled again. Moreover, when a release instruction arrives at the issue stage, the warp status bit is unset, and the warp's acquired SRP section index is retrieved from the lookup table, as shown in Figure 4.5(b). This index determines the bit to unset in the SRP bitmask, specifying the release of the previously acquired extended physical register set. Also, note that in case the extended register set size does not allow having maximum number of SRP sections, those bits in SRP bitmask that do not correspond to any SRP section are set at the beginning of the kernel placement and stay intact throughout the execution.

Total number of bits introduced into the baseline by RegMutex is 384. Compared to register file virtualization approach [62], which requires $30,240$ bits for the renaming table and 1024 bits for register availability indication (excluding Release Flag Cache), RegMutex reduces the additional structure storage cost by more than 81x. Moreover, since the introduced acquire/release instruction is simple, RegMutex does not need to use meta-instructions, as opposed to [62], which necessitates partitioning the fetch stage into two separate stages.

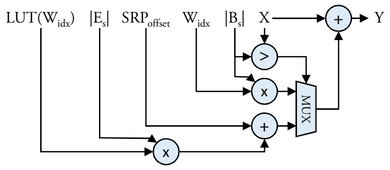
Architected-To-Physical Register Mapping

In GPUs registers are allocated per warp and indexed by the warp ID within SM. For instance, the baseline design from *GPGPU-sim* [17] is from Nvidia Fermi architecture containing 32K 32-bit physical registers. Given there are 32 threads within the warp, there are 1K of physical register *packs* to distribute among the warps. As we mentioned earlier, to map packs of architected registers to physical registers GPUs use a simple equation $Y = X + B$. In this equation, B is a warp specific base address assigned at run time and is resulted from multiplying the warp index within SM (W_{idx}) with a constant coefficient ($Coeff$) determined by the kernel's total register usage: $B = Coeff \times W_{idx}$. This baseline design can be viewed in Figure 4.6(a) and is implemented within GPU's Operand Collector unit before accessing the register file banks.

To support RegMutex, we augment the baseline design as shown in Figure 4.6(b). Since the base addresses for physical registers designated to hold B_s and E_s of a warp are disjoint, the warp compares the architected register index with $|B_s|$ to realize if the register belongs to the base set or the extended set. If the register belongs to the base set, in a fashion similar to baseline, the warp index within SM gets multiplied to $|B_s|$ to result the base address for the physical register. Otherwise, the SRP section assigned to the warp ($LUT(W_{idx})$) is multiplied by $|E_s|$ to get the base address within SRP. The result is added with SRP_{offset}, the offset of SRP within register file, to constitute the physical base address



(a) Baseline.



(b) Augmented for RegMutex.

Figure 4.6: Architected to physical register mapping design in the Operand Collector Unit. X is the architected register index and Y is the resulted physical register index.

for the register. In this design, the values of $|B_s|$ and $|E_s|$ are supplied by the compiler and alongside SRP_{offset} are given to the Operand Collector Unit at the kernel launch.

In summary, RegMutex enjoys a much simpler design compared to existing approaches such as [62] and [132] which micromanage the allocation of every register and necessitate additional structures such as Release Flag Cache.

4.3.3 Paired-Warps Specialization

In this part, we introduce a specialization of RegMutex that, rather than time-multiplexing the registers across all the SM's resident warps, shares the extended register set between specific pairs of warp. Although this approach reduces the register sharing opportunity, it lowers the amount of hardware modifications even further. In this specialization, the design sets aside $2 \times |B_s| + |E_s|$ physical registers for each pair of warps. Each warp's $|B_s|$ allocation is static and exclusive but E_s is time-multiplexed between the two. Therefore, while both warps can move forward in release state, only one of them may progress in the acquire state, disallowing the other warp to acquire E_s until the release point. Paired-warps specialization of RegMutex eliminates the need for the lookup table and the SRP bitmask, and only requires a bitmask with the length half the maximum number of warps in the SM, i.e., $N_w/2$, to specify the status of the extended set shared between pairs of warps.

4.4 Experimental Evaluation

To evaluate RegMutex's performance, we extended *GPGPU-Sim v 3.2.2* [17] simulator. We used the microarchitecture specifications for GeForce GTX480 GPU that comes with the simulation framework. It includes 15 SMs, 128 KB register file size per SM, 2 warp schedulers per SM, and the default greedy-then-oldest scheduling policy. In addition to the register file size in SM, we allow shared memory usage per SM and the maximum number of resident threads in SM to act as other constraints that affect the theoretical occupancy of the CUDA kernels. Note that although we carry out our simulations based on an Nvidia Fermi GPU, the principles behind register allocation on newer Nvidia CUDA-enabled architectures including Kepler, Maxwell, Pascal, and Volta have stayed the same: registers are still statically and exclusively reserved. Therefore, the resulting register file underutilization challenge does indeed still exist. Even though per-SM register file size has been doubled in newer architectures, the maximum number of resident warps on the SM in newer GPUs is also increased. As a result, in all post-Fermi Nvidia GPUs having more than 32 registers per thread definitely results in incomplete occupancy, which is troubling for applications with high register demand. Hence, our solution is applicable and generalizable to newer GPU architectures as well.

We utilized PTXPlus to extract basic block information as well as control flow analysis in order to implement RegMutex's compiler support. PTXPlus is a tool integrated with GPGPU-Sim that enables implementation of compiler optimizations when working with the simulator. It uses an augmented form of PTX intermediate representation that is extracted from the binary, and therefore is expected to fully preserve the optimizations applied at the PTX-to-SASS level. PTXPlus is the closest level to machine code that GPGPU-Sim allows for applying compiler optimizations.

We selected a total of 16 applications from Rodinia [31], Parboil Benchmark Suite [124], and Nvidia CUDA SDK [1] to verify the effectiveness of RegMutex under different workloads. These applications are shown in Table 4.1 and exhibit different SM resource requirements. Please note that these workloads suffer from high register usage and are selected to show the benefits of RegMutex in different scenarios. Applications that do not have such property are not affected by applying our technique since RegMutex evaluates all the registers as the members of the base register set, therefore, it does not insert any acquire or release instructions into the program. Moreover, none of the presented workloads incur simultaneously executing dissimilar kernels. Co-scheduling dissimilar kernels on an SM is not supported by our technique and results in falling back to the default execution mode (zero-sized extended set). Table 4.1 also specifies the number of registers per thread for each kernel. The numbers in the parenthesis show the number of registers rounded to the upper multiple of 4. The simulation framework uses this number for resource allocation calculations. We also showed the calculated base set size for RegMutex for each application in the table. All the applications are compiled with NVCC 4.0 and GCC 4.6 with -O3 compilation flag. Since PTXPlus is not compatible with CUDA Compute Capability 2.0 or higher, applications are compiled for Compute Capability 1.3.

Application	# Regs.	$\lvert B_s \rvert$	Application	# Regs.	$\lvert B_s \rvert$
BFS	21 (24)	18	Gaussian	12 (12)	8
CUTCP	25 (28)	20	HeatWall	28 (28)	20
DWT2D	44 (44)	38	LavaMD	37 (40)	28
HotSpot3D	32 (32)	24	MergeSort	15 (16)	12
MRI-Q	21 (24)	18	MonteCarlo	13 (16)	12
ParticleFilter	32 (32)	20	SPMV	16 (16)	12
RadixSort	33 (36)	30	SRAD	18 (20)	12
SAD	30 (32)	20	TPACF	28 (28)	20

Table 4.1: Workloads used in experiments. The number of registers per thread and RegMutex's base register set size are shown for each kernel.

4.4.1 Kernel Occupancy Boost Analysis

We first analyze the performance improvement enabled by RegMutex for 8 GPU kernels from Table 4.1 on the baseline architecture. The theoretical occupancy of this set of kernels are limited by the excessive register demand, hence, enhancing their occupancy by time-sharing a portion of the registers using RegMutex can be beneficial. For these applications, Figure 4.7 shows the percentage of execution cycle reduction with RegMutex calculated with respect to the number of baseline execution cycles. It also shows the influence of RegMutex on the theoretical occupancy of the kernel by plotting the initial occupancy of the kernels alongside the occupancy with RegMutex. On average, RegMutex has reduced the execution cycle of the kernels by 13% via enhancing the overall register file utilization.

In a case such as BFS, the boost in the occupancy resulted in 23% reduction in the execution cycles. On the other hand, SAD application does not enjoy such performance

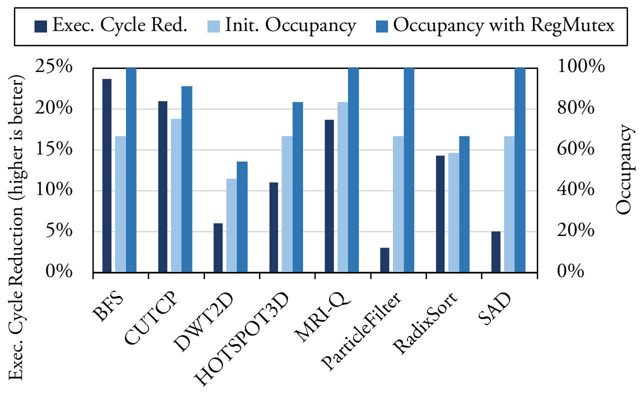


Figure 4.7: The performance improvement enabled by RegMutex over the baseline.

improvement with the same amount of occupancy enhancement. This tells us that theoretical occupancy cannot be directly indicative of the performance enabled by RegMutex, yet, is one of the contributing factors. In the case of BFS and SAD, extended set size and SRP section are other impactful parameters. SAD requires a considerably larger extended set compared to BFS (see Table 4.1) for the occupancy increase. This makes the number of SRP sections limited which increases the contention over acquiring the extended set between warps. DWT2D and ParticleFilter applications suffer from the same issue as well. Program nature is another contributing factor to the performance improvement provided by RegMutex. A kernel that holds an extended set more often and for longer instructions increases the chance of other warps having to wait at acquire points. The contribution extent for all these parameters depend on each other, and most importantly, for typical kernels that are data-driven, to the input of the kernel. Therefore, speculating suitable parameters using heuristics requires careful analysis of the program.

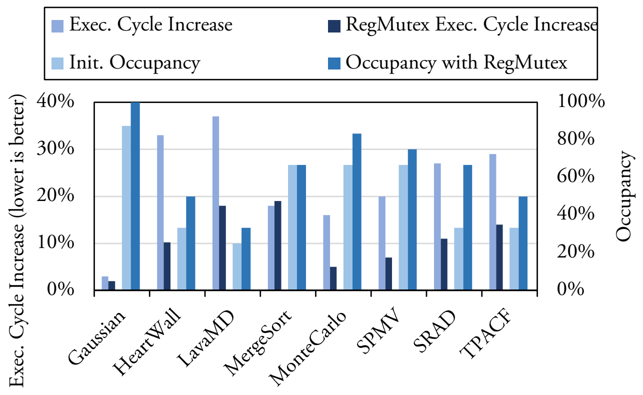


Figure 4.8: The performance of applications with and without RegMutex on an architecture with half the baseline's register file size.

4.4.2 Register File Size Reduction Analysis

In this part, we analyze the effect of RegMutex on 8 applications for which the register file size is not limiting the theoretical occupancy. For these applications, similar to GPU-Shrink [62], we halve the register file size of the baseline to 64 KB per SM and see the effect of this reduction with and without RegMutex and compare it with the baseline mode. Jeon *et al.* [62] argue that this design leads to significant power savings by reducing the register file dynamic and overall power consumption by 20% and 30% respectively. Note that unlike GPU-Shrink we did not enforce register spilling, but rather allowed GPU to determine the number of resident warps on SM under specified circumstances.

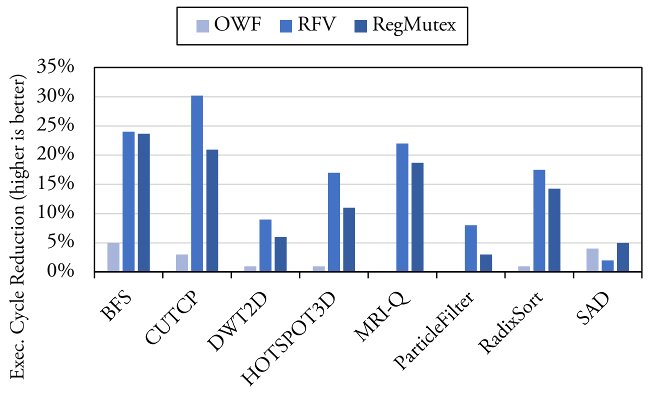
Figure 4.8 compares the execution cycle for scenarios where RegMutex is present and absent with the kernel's number of execution cycles for the architecture with full register file. It is evident that in presence of RegMutex, the kernel experiences much less increase in the number of execution cycles and allows the kernels to preserve the performance when an architecture with smaller number of physical registers is provided. For the applications in

Figure 4.8, while the design without RegMutex suffers from 23% increase in the number of execution cycles on average, with RegMutex we observe an average of 9% growth in the number of execution cycles.

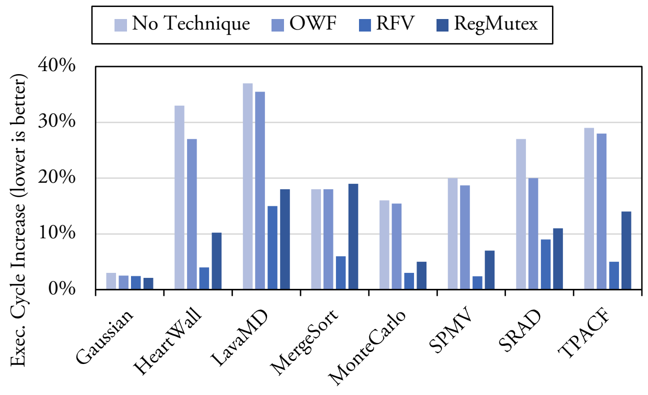
We also plotted the occupancy of the kernels before and after applying RegMutex on the architecture with half the baseline's physical registers in Figure 4.8. Similar to previous part, we observe that occupancy is a contributing factor to RegMutex's performance. In 7 out of these 8 applications RegMutex has successfully increased the register utilization by enhancing the occupancy of the kernel. It is only in MergeSort workload that our heuristic for extended register set size determination comes up with a size that does not increase the occupancy. Therefore, we observed no benefit, but a slight increase in execution cycle due to added RegMutex instructions. This is, in fact, the only workload among 16 applications for which the default RegMutex incurred slowdown.

4.4.3 Performance Comparison with Related Work

In this section, we compare performance improvement provided by RegMutex with the two most closely related approaches: i) Resource Sharing with the *Owner Warp First* (OWF) scheduling optimization [59], and ii) *Register File Virtualization* (RFV) [62]. To perform an apples-to-apples comparison, we used implementations of both approaches on GPGPU-Sim similar to the extensions used to implement RegMutex. Figure 4.9(a) presents the results of the comparison on the baseline architecture. The average *reduction* in kernel execution time in cycles is 1.9%, 16.2%, and 12.8% for OWF, RFV, and RegMutex respectively. We see that both RFV and RegMutex significantly out-perform OWF. While the improvement



(a) On the baseline architecture.



(b) With half the baseline architecture's registers.

Figure 4.9: RegMutex performance comparison with Register File Virtualization (RFV) [62] and the work of Jatala *et al.* [59], which we refer to it as OWF.

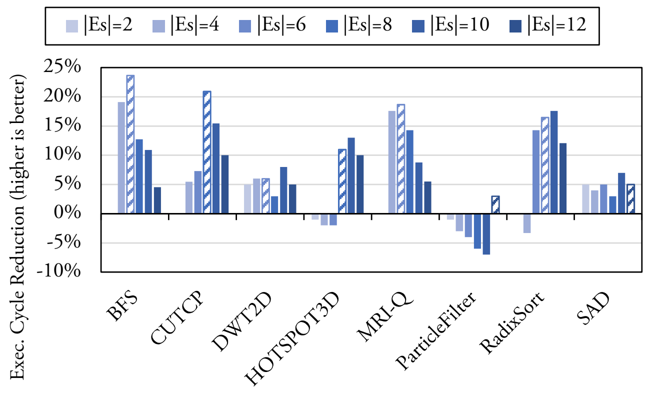


Figure 4.10: The sensitivity of kernel performance to variations in the extended set size with RegMutex. Columns with diagonal stripes are our heuristic's pick.

due to RFV is 3.4% higher than that of RegMutex on average, RegMutex has much lower hardware implementation complexity than RFV, as discussed in Section 4.3.2. RVF demands more than 31 kilobits for additional structure storage in the default architecture with 128 KB registers, whereas RegMutex only needs 384 bits, reducing the storage requirement by more than 81x.

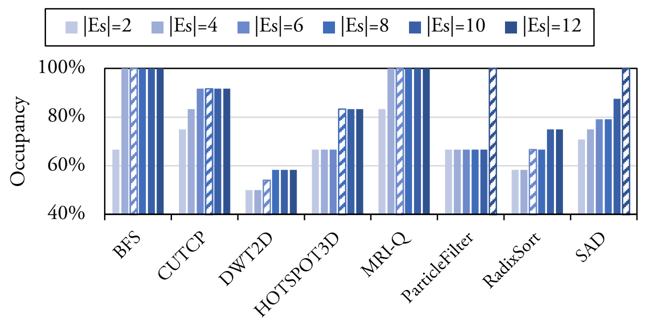
We also perform this comparison on the architecture with half the baseline's register file size. The results are shown in Figure 4.9(b). We observe an average of 22.9% *increase* in execution cycles that results from halving the register file size when no technique is applied. The average increase in kernel execution cycles is 20.6%, 5.9%, and 10.8% for OWF, RFV, and RegMutex respectively. Again, we see that both RFV and RegMutex significantly out-perform OWF in this case as well, and that RFV performs better than RegMutex, but does so with increased hardware implementation complexity (as discussed in Section 4.3.2).

4.4.4 Extended Set Size Sensitivity Analysis

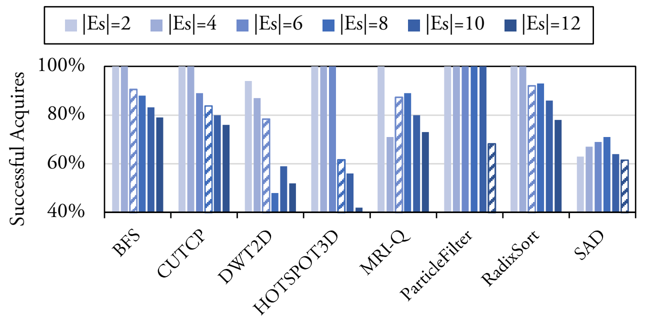
Here we analyze the performance sensitivity of our technique to the size of the extended set. In Section 4.3.1 we mentioned that the size of the extended set, i.e., $|E_s|$, which is chosen at compile time, affects the performance in two ways. Increasing $|E_s|$ results in $|B_s|$ decreasing which allows more concurrent warps to reside on the SM thus enhancing the occupancy. On the other hand, a higher $|E_s|$ means larger sections of a program are marked as acquire state therefore it is more probable for warps to be holding extended sets, hence, warps may have to wait more often and for longer times before they can acquire physical registers for their extended set.

To observe the influence of extended set size on the performance of kernels, we manually set $|E_s|$ to 2, 4, 6, 8, 10, and 12, and observed the execution cycle reductions. Figure 4.10 plots the results. We distinguished the extended set size determined by our heuristic (described in Section 4.3.1) using diagonal stripes. As you can see, although the best performing $|E_s|$ differs from one application to another and does not follow any particular trend, our method has been able to pick the best or one of the best extended set sizes for each application. This is due to prioritizing occupancy and then adjusting it based on the number of sections in SRP.

To further investigate the results, for different $|E_s|$'s, we measured the theoretical occupancy of each kernel and the percentage of successful acquire requests with respect to all acquire instructions executed and plotted them in Figure 4.11(a) and Figure 4.11(b). By comparing the results in these plots, it becomes clear that as $|E_s|$ gets larger, occupancy increases but the chance of a successful acquire usually reduces. Both of these two adver-



(a) Theoretical occupancy.



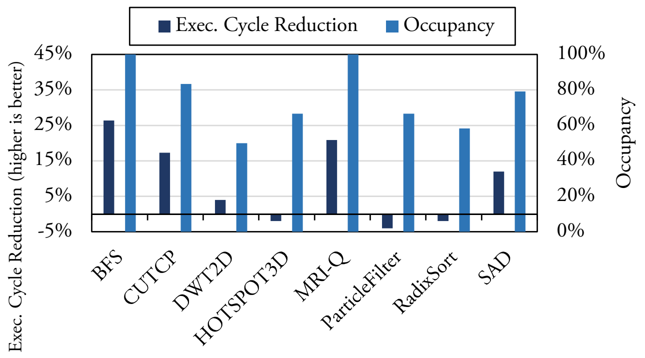
(b) Successful acquires among all acquire instructions.

Figure 4.11: The variations in the theoretical kernel occupancy and the ratio of successful acquires with respect to changes in the extended set size. Columns with diagonal stripes are our heuristic's pick.

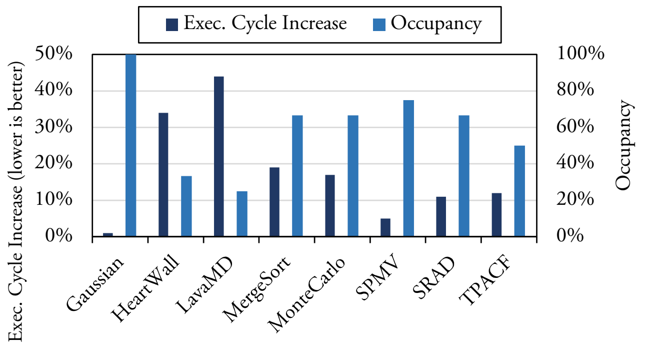
sarial effects contribute to RegMutex's performance. This makes suitable $|E_s|$ selection a challenging task that requires careful observation of the program behavior as well as static calculation of the kernel occupancy in the given architecture.

4.4.5 Paired-Warps Specialization Performance Analysis

As we mentioned earlier in Section 4.3.3, paired-warps specialization of RegMutex eliminates the need for the SRP bitmask and the lookup table by privatizing SRP sections



(a) Execution cycle reduction is measured against the baseline architecture.



(b) Execution cycle is measured for the architecture with half the baseline physical registers. To be consistent with previous plots, the increase is measured against the baseline kernel performance on the architecture with full register file.

Figure 4.12: The effect of RegMutex's paired-warps specialization on the execution cycle and the occupancy of kernels.

among pairs of warps. This reduces the hardware storage cost by more than 20x compared to the non-specialized RegMutex at the expense of lower generality. Figure 4.12(a) shows the execution cycle reduction and the resulted theoretical occupancy after applying paired-warps RegMutex to the baseline architecture. As can be derived by comparing this figure with Figure 4.7, this specialization is effective when the occupancy can be improved, as is the case for 5 of our 8 applications. For a few applications such as SAD, we observe even a higher reduction in execution cycles compared to the default RegMutex. We found that this is generally due to higher probability of acquires with this specialization, as shown in Figure 4.13. While paired-warps specialization guarantees the exclusive access to the extended set for a warp be shared with at most one other warp, default RegMutex may have to share a few SRP sections among many resident warps on SM. This can lead to an increased waiting time on acquire instructions for the default mode. However, when the occupancy stays intact, paired-warps specialization is unable to improve the performance. The inability to improve the occupancy in such cases stems from the guarantee this specialization has to provide for pairs of warps. This makes paired-warps specialization susceptible for such scenarios while the default mode exhibits flexibility and therefore resistance in these cases. In other words, exclusivity of $2 \times |B_s| + |E_s|$ registers for pairs of warps makes the specialization outperform or underperform the default mode depending on the application. On average, paired-warps specialization reduces the execution cycles for applications in Figure 4.12(a) by 8% which is 4% less compared to the default mode.

Figure 4.12(b) illustrates the increase in the execution cycles as well as the resulted occupancy when paired-warps specialization is used on the architecture with half the baseline physical registers. Here we observe the similar phenomenon as we described above as well. When the occupancy stays the same, as it is the case for 4 out of 8 applications, no performance improvement is provided. However, in other cases where the occupancy could be increased, paired-warps specialization becomes effective. For these applications, this specialization has increased the execution cycles over the baseline with full register file by

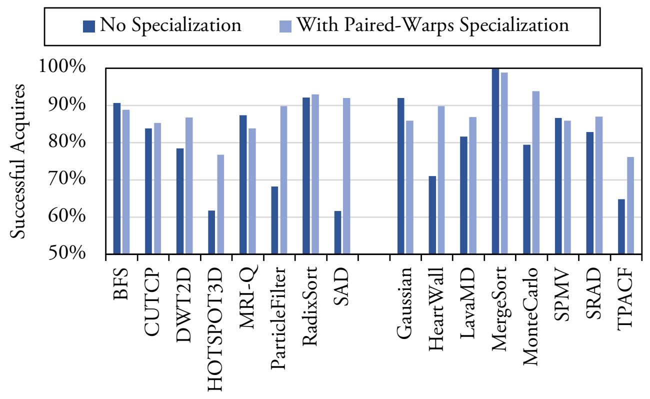


Figure 4.13: Acquire instruction success rate in RegMutex with and without paired-warps specialization. The results for the 8 leftmost applications are reported on the baseline architecture, and the rest, on the architecture with half of the baseline register file size.

17% which is 5% less than the baseline with half the register file but it is outperformed by the default RegMutex by 8% difference.

4.5 Related Work

As we explained earlier in Section 4.2, a number of directly related works propose solutions for static and exclusive GPU register assignment. Tarjan *et al.* [132] suggest virtualizing the registers and assigning them onto physical registers on-demand. However, this method is expected to incur hardware complexities even beyond what was proposed by Jeon *et al.* [62]. Gebhart *et al.* [45, 46] propose multi-level register file designs where long-lived registers and short-lived registers are placed in different register hierarchies for power efficiency purposes. While imposing high amount of modifications to the existing hardware, these solutions also lack generality due to the fixed sizes of the register file hierarchy levels. RegMutex, in contrast, does not disturb the performance of an application that does not utilize it. Tan and Fu [129] suggest another hierarchical approach where registers are classified into *fast* and *slow* categories to reduce susceptibility of GPU register file to process variation. However, RegMutex's aim is to offer approximately the same performance at a lower cost, or higher performance at the same cost, by reducing the number of required registers or by allowing residence of more warps on an SM. Also, as opposed to the work by Jatala *et al.* [59], RegMutex offloads the register ownership arbitration to the compiler and allows the set of shared registers be handed over between the warps multiple times. Zorua [135] is another work that utilizes a runtime-compiler-hardware synergistic approach for resource virtualization at runtime. While in Zorua, performance portability across multiple architectures is the goal and is achieved via virtualizing on-chip resources, RegMutex tackles the challenge of static resource assignment *during* the kernel execution. RegLess [78] replaces register file with a smaller actively-managed staging unit and LTRF [115] suggests a hierarchical RF design, both utilizing the compiler to provide hints to the hardware for the run-time use. Unlike these works, RegMutex does not fundamentally change or replace the RF structure, and can easily be disabled or enabled by the compiler. A patent application by Coon and Lindholm [35] also has the notion of grouping threads together based on resource sharing; however, they only allow one thread from each group to execute at a time while our goal is to maximize concurrent execution while sharing a limited resource.

Another body of papers, orthogonal to RegMutex, target economical use of GPU's available resources. Kim *et al.* [76] utilized unused registers for executing the warps in a special mode called pre-execution in order to cope with long stalls due to memory accesses. Warped-Compression [80] exploits the similarity of the register values between threads within a warp during the execution in order to eliminate redundant register file occupations. Compressing similar registers into one register essentially results in saving on the GPU register file power consumption. It also resembles the works of Jourdan *et al.* [66] for CPUs where logical registers are mapped into physical ones when sharing the same content. KernelMerge [50] aims to allow co-residency of two GPU kernels on one device to enables utilization of resources that are left unutilized when only one of the kernels is running. CCC [74] utilizes on-chip shared memory to collect tasks for future warp-efficient use. Also, Yoon *et al.* [146] propose an architecture to increase the on-chip resource utilization by improving the CTA scheduling policy. While sharing the same general goal with RegMutex, we observe no restriction on simultaneous application of these works with our proposed technique.

In the CPU realm, hardware-only approaches [90,93] as well as combined compiler-microarchitecture solutions [64,85,89] have been proposed for quick dead register identification and release. Ayala *et al.* [16] suggest a software-hardware technique that tags sections of the program which require only a small amount of registers for execution and allows disabling regions of the register file during the execution for energy saving purposes. As we mentioned before, such techniques in massively multi-threaded devices such as GPUs are often impractical due to the their heavy reliance on TLP between resident warps.

Chapter 5

LARF: Locality-Aware Register File for GPUs

5.1 Introduction

In big data era, one of the most critical computing problems is to speedup memory accesses. To deliver huge data requested by massively parallel threads, big data workloads such as deep learning are accelerated on many-core processors such as GPUs that are equipped with multiple high-bandwidth memory controllers as well as various on-chip memories. Though the on-chip memory size is almost doubling every new generations of GPU architectures, long memory latency and limited on-chip memory size are still the main performance bottlenecks of big data workloads. However, many big data workloads inherently have data sharing features. For example, large matrix operations such as matrix multiply and dot product are one of the most commonly used algorithms in big data workloads. In matrix

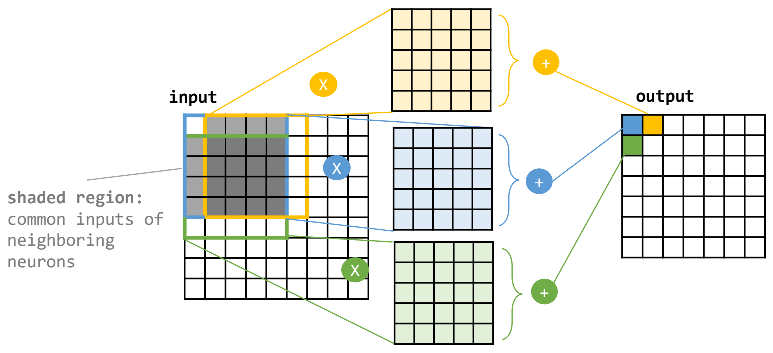


Figure 5.1: Input data sharing among three neurons that are vertically and horizontally neighboring in a conv layer

multiply, any two neighboring threads that are assigned to the adjacent output matrix entries in a row use the same row of the first input matrix. Also, the core algorithm of convolutional neural network (CNN), which is one of the most successful deep learning algorithms that has proved its efficiency in image recognition problems is dot-product operation. In a CNN, neighboring neurons calculate convolutions (dot-product) of sliding sub-windows of the input data where sub-windows overlap significantly.

Figure 5.1 shows an example data sharing across neighboring neurons in a convolution (conv) layer computation. The colored entries of the output matrix are the conv results of three neurons where blue and yellow neurons are horizontal neighbors and blue and green neurons are vertical neighbors. Each neuron uses a sub-set of data of the input matrix, which is highlighted with colored border lines in the input matrix. The blue-colored neuron takes the data in the first 5×5 matrix region, runs dot-product operations and stores the result to the output entry. Likely, the green and yellow neurons each takes a matrix from the one row below and one column after the blue neuron's input matrix. The data in the shaded regions that are overlapping between any two input regions are used by both neurons. In this example, out of 25 input data of each neuron, 20 are shared between any two neighboring neurons and 16 of them are used by all three neurons. In the typical CNN computing, as individual neurons executions are considered as independent, each neuron issues 25 memory reads even when the 20 and 16 shared data are somewhere in the on-chip memory already.

These redundant memory accesses lead to performance and energy overhead. Even when many of the data can be cached as they are accessed in the similar time windows, due to the limited size of L1 cache, L2 cache should be also excessively accessed. Given that interconnection between L1 and L2 is one of the critical performance bottlenecks of GPUs, it is not desirable that all the neurons independently access memory for their data. Shared memory may be used for reducing the memory access latency. However, due to the limited size, the workload code should be carefully designed to have each CTA to load only a small block of data over multiple iterations as in blocked matrix multiply. Also, using shared memory adds a burden of loading data from global memory to shared memory and then register file. Thus, allocating a data to the shared memory that is referenced only once by the code only elongates the data access latency. Therefore, the programmer should carefully determine access frequency of individual data at compile time to determine the placement of the data allocation, which is not easy. Also, these independent and redundant memory accesses lead to a significant on-chip memory waste. Because individual neurons (threads) store the loaded data to their private on-chip memories such as registers before computation, the on-chip memory ends up with maintaining multiple copies of identical data. For example, in Figure 5.1, each of the 16 data that are shared by all three neurons will be stored in a register of all three neurons. Therefore, there will be three redundant copies of the same data in the register file. If data sharing is done in the other on-chip memories such as shared memory, four copies of each data should be stored in the SM in this example: a copy in the shared memory and three copies in the register file for the three neurons. Given the massive parallelism of big data workloads, there will be a significant amount of redundant data in the on-chip memory. Therefore, there should be a mechanism that effectively exploits the inherent data sharing patterns of big data workloads to mitigate performance and on-chip memory shortage problems, instead of blindly increasing memory size and bandwidth which is costly.

In this chapter, we propose a locality-aware register file (LARF) for GPUs that reduces the redundant memory accesses by sharing data directly in the register file without any help from L1 or shared memory. Instead of having the three neurons in Figure 5.1 to store their 25 input data separately, our approach stores the overlapping 20 data to a common set of registers. A neuron accesses memory only when the data is not already in the register file. This approach not only helps reduce the redundant memory accesses but also decreases the register file usage because the three neurons share one physical copy of register for each of the overlapping data rather than having them in their private register separately. The registers that are saved by our approach and the other on-chip memories may be used for improving the performance further (e.g., by running more CTAs). Note that our approach can be orthogonally applied with the other optimizations as we do not require software modifications. The larger size and the lower typical utilization also makes the GPU register file more favorable for the data sharing over the other on-chip memories [15, 62]. Therefore, we leverage large register file space to support more threads by preserving common data longer without worrying about cache thrashing, insufficient shared memory space, or a sophisticatedly designed software. To maximize the coverage of data sharing, we also propose a locality-aware warp scheduler (LAS). To share a data with the other threads, the register should keep the data until when most of the shared parties (the other threads) consume the data. However, due to the scheduling timing disparity between threads especially in the advanced warp schedulers such as two level scheduler (TWL) and greedy scheduler (GTO), the register may be overwritten before the other threads are even scheduled to issue the load instruction. To resolve this timing issue, LAS checks the list of memory addresses of the data shared in register file and assigns higher priority to the warps whose next instruction is an off-chip memory load instruction and the target address is in the shared address list. Our evaluation results show that the LARF and LAS together improve performance by up to 3$\times$ and reduce the global load accesses by up to 80% for an assorted set of state-of-the-art deep learning and matrix multiply workloads.

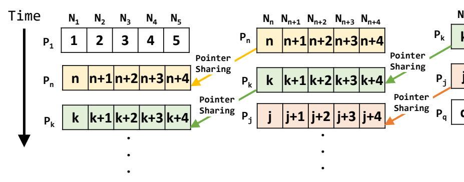
5.2 Background and Related Work

5.2.1 On-chip Inter-thread Data Sharing

On-chip inter-thread data sharing have been explored by several studies [14, 32, 53]. However, many of them used specialized data flow architectures where data sharing is easier via direct communication channels among processing elements. Diamos et al. and Khorasani et al. [40, 72] leveraged GPU register file to maintain RNN parameters. However, as they aim to share parameters across multiple time steps and hence do not need to consider concurrent accesses, the register mapping is relatively straightforward. WIR [77] leveraged physical register sharing to skip instructions. As WIR focused on reducing arithmetic operation executions for a better energy efficiency, WIR has a more complex design that consists of hash and instruction meta information tables. Also, the register reuse is allowed only when the warp-unit register has a perfectly identical data. Our approach focuses on memory access overhead and hence exploits common memory access patterns that enables warps to share data for both perfect- and partial-matching cases with a simpler address mapping table. Unlike their sophisticate design for also covering shared memory accesses, we simply focus on global memory accesses because skipping shared memory accesses does not provide notable performance advantage and unnecessarily idles the shared memory unless the application is re-written not to use shared memory. The state-of-the-art deep learning libraries use register file and shared memory to speedup the data access latency [33]. However, software-level data sharing leads to an excessive register file usage with redundant data copies because current GPUs do not allow inter-warp register sharing. In this chapter, we extend the architecture-to-physical register mapping in the architecture level to enable data sharing across warps.

5.3 Locality-Aware Register File

Warps may share either perfectly-matching data or partially-matching data. We explain how our approach supports both cases by using a conv layer example.


(a) **Perfect sharing among vertically neighboring neurons**

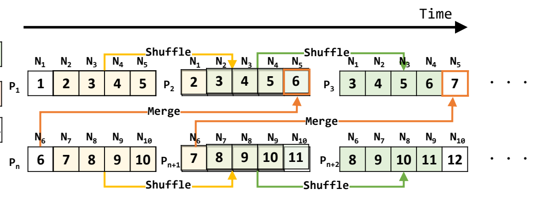

(b) **Partial sharing among horizontally neighboring neurons**

Figure 5.2: Proposed Data Sharing (an example of CNN)

5.3.1 Perfect Sharing

Any two vertically neighboring neurons have commonly used input data as illustrated as a shaded region between the input matrices of the blue and the green neurons in Figure 5.1. The entire contents of each row in the overlapping region are shared by the two neurons, which enables perfect sharing. As each neuron is assigned to a GPU thread in typical CNN implementations, vertically neighboring neurons are likely to be in different warps. For example, if the blue neuron is thread 0 of warp 0, the green neuron is thread 0 of warp N. Figure 5.2a shows the register updates of perfect sharing in a warp unit, where we assume that each warp has only five threads for simplicity. Each five-element array is a warp-unit register where each entry is a register of a thread (neuron) in the same warp. The three warp-unit registers in the same row are of the three groups of neurons that are vertically neighboring. In other words, the first warp having N_1 to N_5 is the warp that has the blue neuron of Figure 5.1, the N_{th} warp consisting of N_n to N_{n+4} is the one that has green neuron, and the K_{th} warp is another group of neurons that are mapped to one row below the N_{th} warp. As can be noticed, the second row of the first warp is identical with the first row of the N_{th} warp. Likely, the third row of the first warp is the second row of N_{th} warp as well as the first row of K_{th} warp.

These vertically overlapping rows can be directly shared by simply mapping the same physical register pointer to the architectural registers of the neighboring warps. For example, if the N_{th} warp loaded its first row to a physical register P_n, the first warp can get its second row values by simply mapping its destination register of the load instruction for the second row to P_n without needing to access memory or copy data from other registers. Likely, once one of the three warps loads the values in P_k, the other two warps can get these data by simply mapping their registers to P_k. With this pointer sharing, we can eliminate redundant loads and reduce the register file utilization by keeping only one copy of warp-unit register values across the neurons. To enable warps to check if their target data is in the register file already, we design an address mapping table which contains the target address information and the mapped physical register id. The details will be explained in Section 5.5.

The perfect sharing can be also found among horizontally neighboring warps in matrix multiply. For example, if two 128×128 matrices are multiplied, four horizontally neighboring warps that are assigned to the same row will use the same input row of the first matrix that can be shared by using physical register mapping. The perfect sharing is light weight because it only requires to map an architectural register to an existing physical register instead of a memory load. However, it is possible only when warps use exactly the same data. For partially-matching data, we use partial sharing, which is explained below.

5.3.2 Partial Sharing

The blue and yellow neurons in Figure 5.1 show the partial sharing between neighboring neurons. The horizontally neighboring neurons are likely to be processed by the neighboring threads in the same warp. For example, if the blue neuron is assigned to the thread 0 of warp 0, the yellow neuron is operated by the thread 1 of warp 0. For each load operation, these threads read data that are next to each other with a given stride distance. In this example, the stride is set to 1 and hence, thread 1 and 2 read (0,0) and (0,1) of the input matrix as their first data and (0,1) and (0,2) for their second data and so on. Figure 5.2b shows the register updates by these load patterns in a warp unit. The three arrays in each row show the first three data that each of the five neurons load from the memory. For example, N_1 loads 1, 2, and 3 (the first data in each array) for its first three data and stores them to the first 32-bit entry of the warp-unit registers P_1, P_2, and P_3, respectively. For each load operation, the values of the warp unit register (each array) are shuffled down by one of the previous load result. For example, N_1 uses 2 as the second data, which is the first data of N_2 and so on. This operation can be represented with NVIDIA CUDA Shuffle instruction, $P_2 = \_\_shfl\_down(P_1,\ 1,\ 5);$. If the destination register value of each load is kept until the second load, the majority of the data (four out of five each time) can be reused without issuing another load operation.

We still have one out of five data that should be newly loaded each time (6 and 7 in the second and the third loads, respectively in the example). These values can be fetched from neighboring warp's registers. The second row of Figure 5.2b shows the register updates of the neurons 6 to 10, which are grouped to the second warp. As warps are interleavingly scheduled in GPUs, after the first warp loads the data 1 to 5, the second warp loads the data 6 to 10. When the first warp is to load the second values, the data 6 is already in a register P_n. Therefore, the second warp-unit register values of the first warp can be constructed by merging the shuffled four values (2 to 5) and the first value of the second warp (6). By running two simple logical operations (SHIFT for shuffle and OR for merge) on the existing register values, individual data do not need to be loaded from the memory multiple times.

5.4 Locality-Aware Warp Scheduler

In LARF, warps share data opportunistically as far as the data is retained in the register file. A shared data will be retained in the register file until when 1) the mapped memory address is not overwritten and 2) the mapping information is not evicted from the system. The first condition will be enforced by checking the target address of store instructions. When an off-chip store instruction is issued, the target address is looked up in the address mapping table. Once the address is hit in the mapping table, the corresponding entry is evicted from the table because the contents of the memory address will be overwritten by the store instruction and hence the value in the mapped physical register will be stale. More details will be described in Section 5.5. Regarding the second condition, there are two timings that the mapping information is evicted from the address mapping table: 1) when the target address is updated, which is related to the first condition that we just explained and 2) when the address mapping table is full and new entry is to be entered. To minimize the space and energy overhead, we cannot hold infinite number of entries in the address mapping table. Thus, in our evaluation, we assumed to have up to 100 400 entries in the mapping table. To use the big data input, the mapping table is quickly filled, especially because the system cannot determine whether the data will be potentially shared by others at the load issuing time. Thus, the address mapping table is obliviously filled in first-in-first-out fashion. Therefore, sharing opportunity is depending on the warp scheduling timing. When the warps that share the same data are scheduled in a closer time proximity, the sharing chance will be higher.

The state-of-the-art warp schedulers follow a design that increases the timing disparity between warps to reduce long latency memory access overhead. For example, in two-level scheduler (TWL), warps in the ready queue are scheduled until when they encounter global memory load instructions and then move to the pending queue. The warps in the pending queue can be scheduled only when there is an empty spot in the ready queue, which will be the timing when a ready warp reaches a global load instruction. Therefore, memory access timing of groups of warps is highly diverse. The greedy algorithm (GTO) makes the timing gap even further. Though these state-of-the-art scheduling algorithms help individual warps to mitigate the memory access latency, due to the longer proximity of memory access timing among warps, the data sharing opportunity is lower.

Our proposed locality-aware warp scheduler (LAS) considers inter-warp load issuing proximity. To schedule warps that access the same memory address before the mapping information is evicted from the address mapping table, LAS prioritizes the warps whose next instruction is global memory load instruction and the target address is in the address mapping table. To reduce the timing gap between warps, the baseline scheduling algorithm is round robin (LRR). Like the baseline LRR, LAS schedules warps in first-in-first-out

```
IADD           R4, g [0x4], R4;
GLD.U32        R4, global14 [R4]
```

(a) SASS code of Tesla GPU

```
IADD           R8, R6, R3;
LD             R6, [R8+0x7f8];
```

(b) SASS code of Fermi GPU

```
add.s64              %rd52, %rd43, %rd53;
ld.global.nc.f64     %fd663, [%rd52];
```

(c) PTX code of Pascal GPU

Figure 5.3: Instruction Sequences of Global Memory Accesses in GPUs

fashion as far as the warps are executing non-global load instructions. When a warp's next instruction is global load instruction, it looks up the address mapping table and changes the scheduling order such that the warp is scheduled as early as possible if the target address is in the table. If otherwise, the warp is scheduled according to the baseline LRR policy.

To identify the warps to prioritize, LAS checks the opcode of the next instruction when a warp is finishing the write back pipeline stage. Note that in the GPU architecture that we used as a baseline [17], each warp has an instruction buffer that holds two following instructions. Whenever one of the instructions in the buffer is issued (evicted from the buffer), a new instruction (next instruction) is fetched from instruction cache, decoded, and filled in the buffer. Therefore, LAS can retrieve the next instruction opcode from the instruction buffer. Once the opcode indicates a global load instruction, the target address is checked. In GPUs, the memory address calculation is done in the memory execution logic. However, we observed that the global load target address calculation is done before the load instruction as can be seen in Figure 5.3. The three code snippets are acquired from AlexNet of Tango benchmark suite [68] that is compiled for NVIDIA Tesla GPU, B+Tree Rodinia benchmark suite [31] that is compiled for Fermi GPU, and cuDNN-based LeNet of GPGPU-Sim benchmark suite that is compiled for Pascal GPU. As can be seen in all codes, the target address of individual global load instructions is hold in a register, which is calculated apriori. In Pascal and Tesla GPUs, there is no offset in the address field. Therefore, the target address can be simply retrieved from the register `%rd52` and `R4`. Note that the prefix `global14` in Tesla GPU is an indicator for global memory accesses. In Fermi, an immediate value is added to the register value `R8`. Thus, the target address of global load instruction can be acquired simply from a register value without any calculation or in the worst case with a simple addition operation if there is an offset. Also as can be seen in Figure 5.3, we observed that the load instructions typically follow an `add` instruction that calculates the load target or base address. This means that the load target address can be checked without a register file lookup at the write-back stage of the warp. Note that to avoid data hazards between consecutive instructions, GPUs normally schedule different warps every cycle where a six-warp batch is known to be golden. Thus, when the `add` instruction is finishing the write-back stage, the following load instruction is not likely to be scheduled yet. Therefore, by simply checking the value that is about to be written back to the register file, LAS can check the target memory address and prioritize the load instruction, without an extra register file access.

In case newer GPU generation does not follow such a back-to-back target address calculation, we still check the operand register id of the load instruction. As the hardware knows the global load instruction format, the opcode and source register fields of the load instruction will be retrieved from the instruction buffer and the source register id is checked with the write-back stage instruction's destination register id. If they do not match, a register file will be referenced with the source register id. Note that as this checking happens when load instruction is the next instruction, the source register is guaranteed to be up-to-date in the register file already.

5.5 Architectural Modification

5.5.1 Dynamic Register Mapping

To share data in a register across multiple warps, it is essential to decouple the architectural registers used by the application code from the physical registers that actually contain the data. Due to the space and energy issues of large register file, there have been a few studies that explored the dynamic architectural-to-physical register mapping already [15, 62]. These studies used a compiler technique to detect register lifetime. Then, the hardware uses the compiler annotated register file lifetime information to map and release an architectural register to/from a physical register. With this method, the register file can maintain only the live registers and proactively reuse the physical register space for other architectural registers. The mapping information is managed by using a register renaming table. We used a similar mechanism to map multiple architectural registers to one physical register in LARF. As can be seen in Figure 5.4, a register renaming logic is implemented that checks the physical register availability and maps an available (unmapped) physical register to an architectural register whenever an architectural register lifetime begins (when the

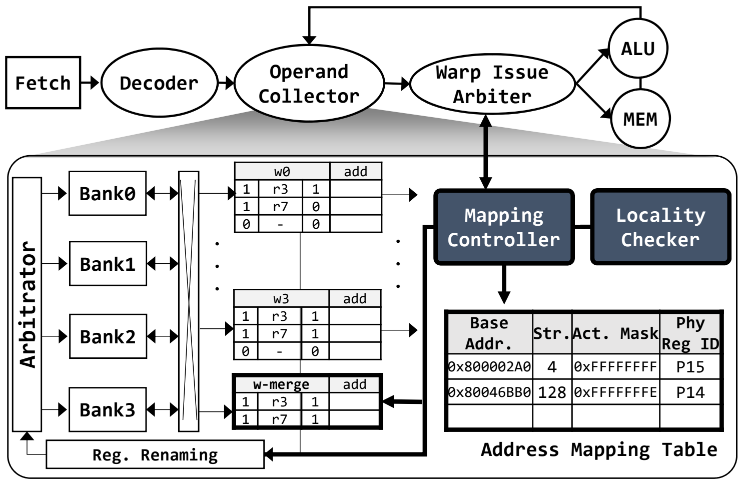


Figure 5.4: Architectural Modification

register is written by an instruction). The physical register availability is checked by using a bit vector that has as many bits as the number of physical warp-unit registers where the bit is set when the physical register is mapped to an architectural register and reset when the corresponding physical register is not mapped with an architectural register. The mapping between architectural and physical registers can be made by entering a new entry in the register renaming table. According to the compiler annotated lifetime information [62], once a register lifetime ends (when the register is last read in the code), the mapping is released by simply deleting the mapping information from the renaming table. The design of the renaming table is the same with the existing studies [62].

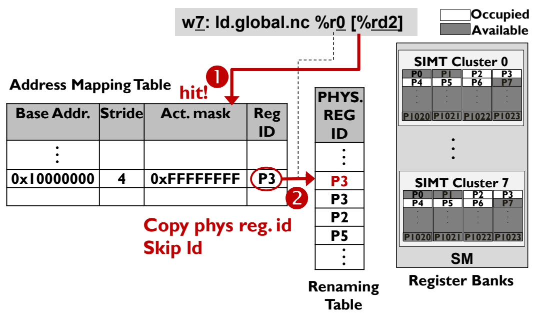
5.5.2 Address Mapping Table and Mapping Controller

Besides the dynamic register mapping logic that we followed existing designs, the new components that designed for LARF are highlighted with bold outline and dark color in

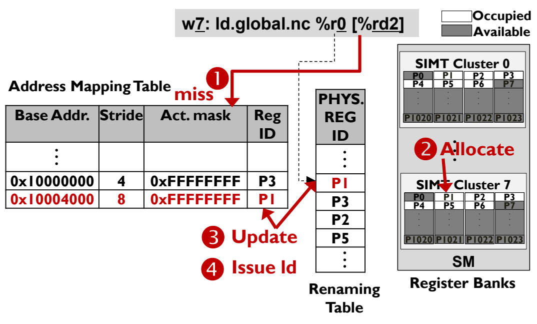
Figure 5.4. To indicate the physical register that holds the shared data, an *Address Mapping Table* is added. Address mapping table holds the load instruction target address information and the mapped physical register id. The data sharing in our proposed approach is done in warp level. Thus, each entry of this table contains the information of a warp-unit load accesses. As threads in a warp typically access consecutive addresses with a fixed stride, we record the base address (the address of the first thread) and the stride rather than storing addresses of all the threads. Though most of the input data are read by fully utilized warps (all 32 active threads), to also support various length input data, the active lane mask of the load instruction is also recorded in the table.

To identify global load instruction and target memory address as well as managing the address mapping table accesses, we design *Mapping Controller*. The mapping controller checks the opcode of instructions at operand collector pipeline stage. Once the opcode field of an instruction indicates a global load instruction, the target memory address is calculated. As explained in Section 5.4, the load target address can be either retrieved from the source register or by executing a simple addition operation over the source register value and an offset value. As instructions at operand collector stage are reading their operands from the register file, the source register value can be acquired without an extra register access. Once the source register value is ready, the target address is quickly calculated and the address mapping table is looked up with the address. If the address is not in the table already, the target memory address, stride, and active mask of the load instruction are recorded to the table and the load will be issued as normal. The destination register of the load instruction will be mapped to an available physical register by using dynamic register mapping mechanism and the physical register id is also filled in the address mapping table as shown in the figure. Note that the architectural-to-physical register mapping is still maintained in the register renaming table which is in the register renaming logic. Therefore, even when the entry in the address mapping table that is associated with the physical register id is evicted later due to a lot of memory accesses, as far as the architectural register lifetime is not ended, the physical register maintains the loaded value and the mapping between the architectural register and the physical register will be retained. In other words, the address mapping table holds the physical register id as a pointer such that the future load accesses can find the value from the mapped physical register as far as the mapping information is not evicted from the table, as explained in Figure 5.2a as *pointer sharing.* To calculate the stride of the memory access, a 32-bit subtractor logic is incorporated in mapping controller that calculates the address gab between adjacent threads. In our current design, we only support one stride per warp unit instruction because we observed that most of the memory loads fall in single-stride accesses. However, the mapping table can be extended to also support multi-stride accesses at the cost of space.

Figure 5.5 shows the interactions between address mapping table and register renaming logic when the load instruction is hit and miss in the address mapping table. Suppose that the address mapping table had one entry that is mapped to a global load instruction which accessed 32 data from 0x10000000 with 4-byte gap such as 0x10000000, 0x10000004, 0x10000008, and so on by a warp. Then, the address mapping table's initial status will look like what is shown in Figure 5.5a, which has base address as 0x10000000, stride as 4, and active mask as 32 1's. Assume that the loaded data are stored in a warp-unit



(a) Steps of Address Mapping Table Hit.



(b) Steps of Address Mapping Table Miss.

Figure 5.5: Example Scenarios with LARF

register, P3, which means that the 4-byte value of 0x10000000 is in lane 0 of P3, that of 0x10000004 is in lane 1 of P3, and so on. We assume that the baseline register file consists of clusters of four register banks such that there are eight clusters of four banks as shown in Figure 5.5a. In each cluster, a physical register such as P3 consists of 16 bytes that support four SIMT lanes, 4 bytes per lane. Thus, P3 of all eight clusters are accessed concurrently by a warp as a warp-unit register. As the mapping is retained in the address mapping table, P3 in the register file is marked as occupied in the figure. Given the initial status, assume that a fully utilized (32 active lanes) warp 7 is trying to issue a global load instruction and the source operand `%rd2` of thread 0 is 0x10000000 and that of the following threads are increasing 4 bytes each. The mapping controller looks up the address mapping table with this load information and finds that there is an entry already (❶). As the data can be found from the register file, the mapping controller simply copies the register id mapped in the address mapping table entry to register renaming logic where the renaming logic fills the the register renaming table with the physical register id (❷). Note that individual architectural register of each warp has a designated entry in the renaming table where the location is calculated as max. theoretical register count per thread × warp_id + destination arch. register id [62]. Once the register id is copied from the address mapping table to the register renaming table, the load instruction does not need to be executed and written back because the data that the instruction wanted to load is already in P3. If the register renaming logic receives a physical register id from the mapping controller, it skips allocating a new physical register, but simply copies the id to the renaming table.

Figure 5.5b shows another example when the load is not found from the address mapping table. Now, the same instruction is about to be issued but the value of `%rd2` is 0x10004000. Suppose that the initial status of the address mapping table was like the table in Figure 5.5a. As 0x10004000 is not in the table already (❶), the mapping controller inserts an entry with the load information, as highlighted. Once the register renaming logic maps the destination register of the instruction to an available physical register (P1 in this example) (❷), the id is sent to the mapping controller so that the new entry in the address mapping table is filled with the register id (❸). Then, the load instruction is issued as normal (❹) and the loaded values will be written back to P1. As can be seen in Figure 5.5b, P1 is newly activated in the register file.

An entry of the address mapping table is deleted when the corresponding memory address is newly written, when the address mapping table does not have a space to fill a new entry, or when there is no more mapped architectural registers. When a store instruction is issued, the address is checked from the address mapping table and the corresponding entry is deleted so that the following load instructions do not use outdated value. Also, when an architectural register becomes dead and the mapped physical register is released, the mapping controller searches the register renaming table by the physical register id to check if there is any more architectural register that is mapped to the same physical register. If there is no more entry in the renaming table that has the physical register id, the address mapping table entry associated with the physical register is deleted.

The address mapping table is sufficient for supporting perfect sharing. However, to support partial sharing that needs to merge two registers, we add one dedicated operand collector slot as marked with $w - merge$ in Figure 5.4. This new slot loads two registers that contain subsets of the requested warp-unit register data. These two registers can be found by checking the address mapping table. For each global memory load instruction, if mapping controller finds that no entry in the address mapping table can cover the requested data, it finds up to two entries that together cover the requested warp-unit data. For example, if a load needs to access addresses from 0x1000000C with stride 4 for 32 SIMT lanes, while the address mapping table has two entries where one entry covers 32 addresses from 0x10000000 with 4-byte stride and another has 32 data loaded from 0x10000080 with 4-byte stride, these two warp-unit registers can be used to create a warp-unit register for the requested load instruction, by shuffling the first warp register by 3 to make the 0x1000000C data to be placed in the SIMT lane 0 and merging with the lanes from 0 to 2 of the second register. If there are no two entries that can support all the requested data, the load is issued to the memory. Otherwise, the two registers are requested to the operand collector logic and the load execution is skipped. Once two register values are ready, the mapping controller uses them to construct the requested data. For this merging process, we design a small ALU that runs bit-level SHIFT operations for shuffling and an OR operation for merging. The merged value is written to the load instruction's destination register at the writeback stage.

The perfect sharing does not add any performance overhead because it only requires one address mapping table lookup and a register renaming table update. If the corresponding entry is found from the address mapping table, it does not even require register writeback because the destination register is mapped to a physical register that already has the requested value. When an entry is not found, the load is issued to the lower-level memory by following the normal load operation. The partial sharing takes longer time than the perfect sharing because two registers should be read from the register file and shuffle and merge operations should be done on them. However, as the destination register will be updated at write-back stage, which is multiple cycles later than the operand collector stage, it does not impose performance overhead.

Discussions: Is there any memory consistency or program correctness problem? No. One may wonder if the proposed load early completion at operand collector stage may cause a memory consistency problem by making load instructions to be completed before the previous store instruction. However, this never happens because 1) in GPU computing, all the warps are considered as independent unless there is a synchronization barriers such as __synchthreads() and hence the load and store of different warps are independent as far as the program is correctly implemented, 2) LARF follows the baseline hardware design that halts all active warps execution at barrier or synchronization point from proceeding to the following instructions until when all warps reach the barrier point, and 3) it is very rare that multiple warps interactively update and read to and from the same memory address in the target big data workloads such as deep learning and matrix multiply. To avoid unnecessary synchronization, most of these workloads assign one thread to one of the output matrix entries and have the one thread to update the entry exclusively. For example, in a convolution layer, the kernel is implemented by using as many threads as the number of entries in the output feature map and each thread calculates the designated entry value by reading multiple input data concurrently. No other threads update the same output entry. Also, in these workloads, the inputs are normally read-only. For example, in convolution layer, the input matrix and the convolution kernel are never updated but only read. Likely, in the matrix multiply, the two input matrices are never updated but only read. Therefore, the load early completion does not cause memory consistency or program correctness problem.

5.5.3 LAS Support

To prioritize a warp that is likely to have hit in the address mapping table, LAS checks the opcode of the next instruction of the warp that is executing the write-back stage and the value that is updated to the register file (which is the following load's target or base address) by the warp, as explained in Section 5.4. As at write-back stage, the computation results are returning back to the register file from the execution units and locality check needs address mapping table lookup, we added a *Locality Checker* near the mapping controller that has interfaces with both register file and address mapping table. The locality checker is an extension of the instruction buffer updation logic that clears dependencies of the decoded instructions in the instruction buffer whenever an instruction finishes the execution at write-back stage. At the write-back stage, the locality checker checks the opcode and offset field of the instructions in the instruction buffer of the corresponding warp. If the opcode is global load's, the offset value is sent to the mapping controller. The mapping controller than intercepts the values that are passed to the register file to be written back, sums the register value and the offset, and looks up the address mapping table with the calculated target memory address. If the address mapping table search hits, the mapping controller sends a signal to the warp issue arbiter so that the warp scheduler can decide the scheduling priority, as can be seen in Figure 5.4. The warp issue arbiter is the main logic of warp scheduler that determines the warp scheduling order. We assume that the baseline warp issue arbiter follows the LRR scheduling algorithm. Once the warp issue arbiter receives a signal from the mapping controller, it gives higher priority to the warp and then the warp will be scheduled at the earliest possible time when the instruction becomes ready, without waiting for its scheduling turn, which typically takes tens of cycles. The locality checker is implemented with a subtractor that checks if the opcode of the instruction matches the global load instruction's opcode, and an multi-bit AND logic that captures the offset field value and send it to the mapping controller. The target address calculation can reuse the existing address calculation logic in the mapping controller, which is explained in Section 5.5.2.

5.6 Evaluation

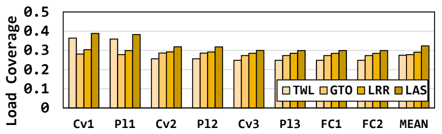
The idea is implemented in GPGPU-Sim v4.0.0 that is adapted to PyTorch integration to run DNN workloads. The simulator is configured as a Pascal GPU (GP102) that has 28 SMs, 128KB register file per SM, and 96KB shared memory per SM. CUDA version 8

Workload Type	Name	Benchmark Suite
CNN	AlexNet (AN)	Tango DNN Benchmark Suite [68]
CNN	CifarNet (CN)	Tango DNN Benchmark Suite [68]
CNN	MobileNet (MN)	Tango DNN Benchmark Suite [68]
CNN	LeNet (LN)	GPGPU-Sim cuDNN Benchmark
Conv Function	3DCONV (3DCV)	PolyBench/GPU [110]
MatrixMul	2MM	PolyBench/GPU [110]
MatrixMul	3MM	PolyBench/GPU [110]

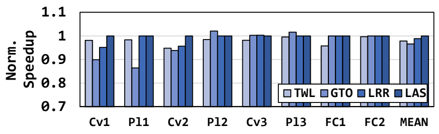
was used. Though ptxplus configuration option should be used to measure a more realistic register usage statistics, we could not use ptxplus because ptxplus is not supported by CUDA version 8. Thus, we ran all experiments based on ptx code. However, our measurements show a close-to-real register allocations because we implemented the lifetime-aware dynamic register mapping, which minimizes the register usage. For example, even when the ptx code keeps allocating a new architectural register by assuming that it has infinite registers, our code still uses as many registers as the number of live registers because whenever an architectural register's lifetime ends, the mapped physical register becomes available again and is reused by other architectural registers. To verify the realistic register utilization, we checked the register file usage statistics of a few workloads with lower version gpgpu-sim code with ptxplus option. We observed that the register utilization statistics of ptx code well synchronizes with ptxplus results. We evaluated our proposed design while running deep learning and matrix multiply workloads that include three CNNs (AlexNet, CifarNet, MobileNet) of Tango DNN benchmark suite [68], one CNN (LeNet) of GPGPU-Sim cuDNN benchmark suite, and two matrixMul (2MM and 3MM) and a 3 dimensional convolution workload (3DCONV) of PolyBench-GPU [110].

5.6.1 Performance

Individual CNN workloads consist of multiple layers where each layer is typically implemented in one or more CUDA kernel(s). Due to individual layers' algorithmic uniqueness, the performance characteristics of each layer type are quite different. For example, in CifarNet, there are three convolution layers, three pooling layers, and two fully-connected layers. In convolution layers, individual threads (neurons) run dot-product operations for a subset of the given large input matrix, while the pooling layer's operations are more like vector operations that extract the maximum or the average value of the given input. In fully-connected layers, each neuron run dot-product operations for all input entries and hence typically memory intensive. Therefore, the architectural characteristics of individual layers are quite different and hence it is necessary to investigate the performance in both per-layer level and end-to-end full network level. We show per-layer performance statistics of CifarNet and MobileNet. Due to the limited space we show end-to-end performance statistics for the remaining workloads.



(a) Normalized Load Sharing Coverage

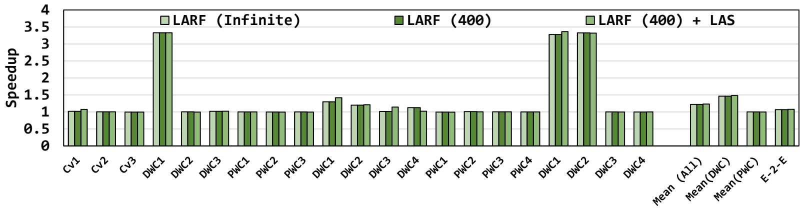


(b) Normalized Speedup

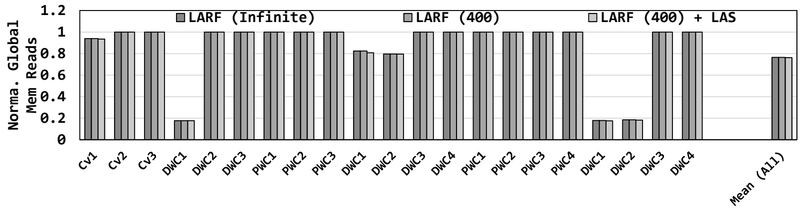
Figure 5.6: Scheduler Impact on CifarNet Execution: Cv: convolution layer, Pl: pooling layer, FC: fully-connected layer

Impact of Locality-Aware Scheduler:

Our proposed LAS is designed based on LRR to increase the data sharing coverage. This may be counter intuitive because TWL and GTO are known to be more advantageous for memory-intensive workloads as these schedulers effectively hide memory access latency by running two warp queues. To justify the better timing proximity of LRR, we measured the performance speedup and data sharing coverage of CifarNet while varying the warp scheduler between TWL, GTO, LRR, and our proposed LAS. By setting the address mapping table size as infinite, we checked the percentage of shared and skipped load instructions and the corresponding performance impact. Figure 5.6a shows the fraction of global load instructions that are shared by LARF out of total global loads when using different warp scheduler. Though TWL showed higher coverage than LRR in the first two layers, the remaining six layers consistently show better data sharing under LRR. This is because LRR minimizes the scheduling timing disparity among warps. As the warps in the same CTA run the same copy of code, if they are scheduled back to back, there is a higher chance to run load instructions that access neighboring memory addresses in the near time window, which helps increase



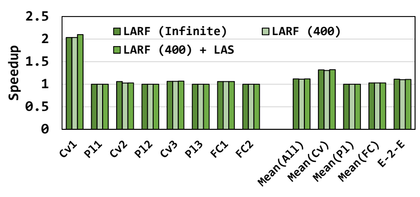
(a) Normalized Total Simulation Cycles



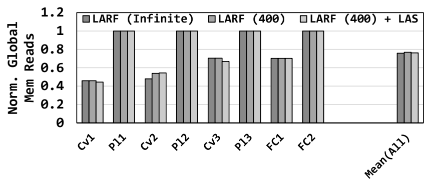
(b) Normalized Total Global Memory Reads

Figure 5.7: Per-Layer Speedup and Global Memory Accesses of MobileNet: Numbers in parenthesis are the address mapping table entry counts. DWC: depth-wise convolution layer, Cv: 3D convolution layer, PWC: point-wise convolution layer

the data sharing opportunity. Our proposed LAS is plotted as the darkest yellow bar charts in Figure 5.6a. As can be seen, it shows superior data sharing of all warp schedulers. The data sharing coverage is directly reflected in the performance. In Figure 5.6b, the speedup of individual layers over LAS is plotted (higher is better). The bar charts trend of individual layers and overall average are well aligned with the load coverage. The superior load coverage of LAS especially in the first two layers over TWL leads to higher performance.



(a) **Normalized Total Simulation Cycles**



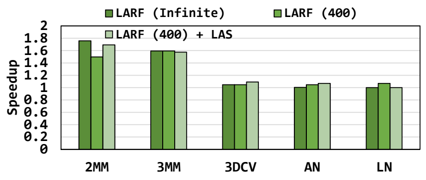
(b) **Normalized Total Global Memory Reads**

Figure 5.8: Per-Layer Speedup and Global Memory Accesses of CifarNet

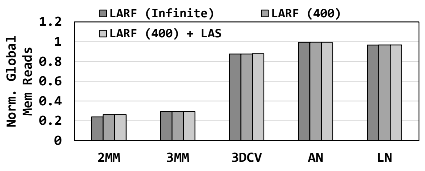
Per-Layer Performance:

Figure 5.7 and Figure 5.8 show per-layer speedup and global memory accesses of the proposed LARF and LAS, normalized by a vanilla Pascal architecture that does not use any of the proposed designs. We used the vanilla Pascal architecture as baseline of all experiments. To understand the theoretical maximum performance improvement and memory access reduction with LARF, we measured these metrics by setting the address mapping table to have infinite number of entries (*LARF (Infinite)* in the figure). Then, we set the address mapping table entry count to be realistic, which is 400 in this case that makes the total address mapping table size within 1 KB (*LARF (400)*). The bar charts entitled *LARF (400) + LAS* show the measurements when LAS is also activated.

In Figure 5.7, we can observe that regardless the order of layers, the layer type distinguishes the impact of LARF notably. For example, the depth-wise convolution layers show significantly better speedups than point-wise convolutions, where the first depth-wise convolution layer reaches almost 3.5× speedup. This is intuitive because depth-wise



(a) **Normalized Total Simulation Cycles**



(b) **Normalized Total Global Memory Reads**

Figure 5.9: End-to-End Speedup and Global Memory Accesses of Remaining Workloads

convolutions typically handle a larger 2D matrix inputs that are sliced from a given 3D input, while point-wise convolutions typically use 1×1×N small square inputs to merge the depth-wise convolution outputs. The sliced depth-wise convolution layer enables a more regular access patterns across neurons compared to 3D convolutions, which leads to a higher data sharing opportunity. As expected, the reduction of global memory reads turned out to be the main driver of the speedup as can be seen in Figure 5.7b.

Another notable thing that we can observe is that the LAS can escalate the impact of LARF even above the theoretical maximum impact that LRR scheduler can derive. According to our experimental results, in these layers, the idle stalls were reduced further, which means that the warp scheduling reordering can make warps to be ready faster by effectively reducing warp's waiting time to be scheduled.

Likely, in Figure 5.8, the two plots show that LARF is more effective to boost the performance of convolution layers. In CifarNet, the earlier layers showed higher improvement and global read reduction ratio, which is because of the larger input size in the earlier layers. With the same reason, in Figure 5.8b, we can observe a notable trend across convolution

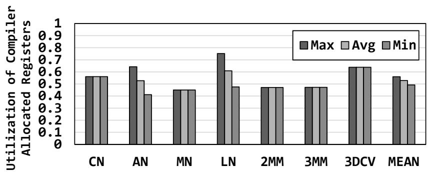


Figure 5.10: Utilization of Compiler Allocated Registers

layers that the global read reduction is reducing as going into the deeper layers. The highest impact of LAS is also found from the very first convolution layer that has a higher potential of data sharing opportunity. In summary, with LARF, the end-to-end performance of MobileNet and CifarNet was sped up to 7% and 10%, respectively both when LAS was used with LARF.

End-to-End Performance

Figure 5.9 shows end-to-end speedup and global memory reads reduction in the remaining five workloads. Each of the workloads shows up to 70%, 60%, 9%, 7%, and 6% speedup, respectively. In all workloads, LARF and a combination of LARF and LAS enable the workloads to reach the theoretical maximum speedup. Especially in 2MM, the LAS improves the performance over LARF-only solution by 40%.

5.6.2 Register File Utilization

As LARF inherently saves register usage by proactively share a copy of physical register across multiple architectural registers, we evaluated the impact of LARF towards the register utilization. Figure 5.10 shows the min, average, and max register utilization over compiler allocated architectural registers throughout the execution of individual workloads.

The utilization was measured per kernel unit. Across the workloads, out of the compiler allocated architectural registers, only 50% were utilized at any given point of time. This is different from register file utilization because the architectural register may not use up all the provided registers. As ptx code assumes to use infinite number of architectural registers, we measured the maximum number of architectural registers that are live at the same time in each warp execution. Out of them, LARF even saved almost 50% register usage because one copy of physical register can be shared across the architectural registers. Given that the parallelism of many big data workloads is limited by register usage, the register saving will help improve the performance of the big data workloads further. Note that, in LeNet of cuDNN benchmark suite, register was the limiting factor of CTA assignment for almost 80% of kernels.

5.6.3 Synergy With Existing Optimizations

As LARF can be orthogonally implemented with existing software-level optimization algorithms, we evaluated the impact of LARF over the existing optimizations and the performance improvement that is expected when applying LARF with the existing optimizations. To measure the impact with a more realistic code, by using ptxplus option, we evaluated only this experiment on Fermi architecture and used a 128×128 matrixMul code that is similar to NVIDIA CUDA SDK. We evaluated performance of matrixMul without any optimization and a blocked matrixMul using shared memory to understand the impact of the register sharing over the existing optimizations. The mapping was limited to 200 entries because the register file size is smaller than Pascal architecture. The register sharing without any optimization derived 83% speedup as plotted as the darkest yellow

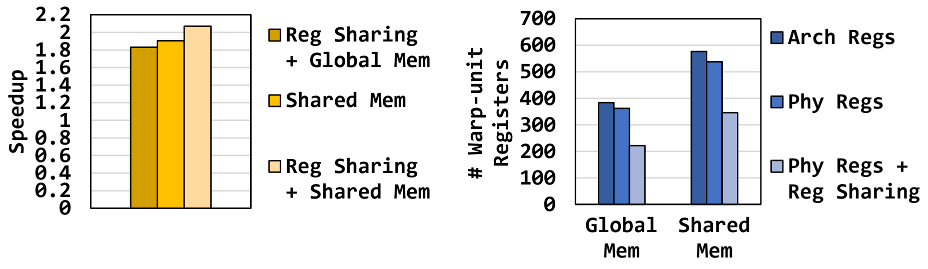


Figure 5.11: Speedup and Register Usage of 128×128 MatrixMul

bar in Figure 5.11 where software optimization (medium yellow bar) shows only 8% further speedup. Regarding the register usage, the optimized matrixMul uses 50% more registers than unoptimized one. With register sharing, both matrixMuls showed almost 40% register usage reduction, which spares 17KB and 24KB register file space each. With these spared registers, LARF allowed to run more CTAs and achieved 10% further speedup over the optimized code as shown in the lightest yellow bar of the left graph.

5.6.4 Area

Our design adds an operand collector slot, an address mapping table, a mapping controller, and a locality checker. An operand collector slot adds insignificant overhead as GPUs have 16 operand collector slots per SM already. Each address mapping table entry consists of a total of 83 bits where base address, stride, active count, and physical register id are 32 bits, 8 bits, 32 bits, and 11 bits, respectively, where the stride length is set based on the statistics of big data workloads which spans from zero to 512 and is typically multiple of four and the physical register id is set based on the Pascal GPU's per-SM register size which is 2048 warp-unit registers. The address mapping table size is limited to 1KB, which can have up to around 400 entries and the register renaming table in the baseline is 1KB. The mapping controller includes a small ALU that can run `add`, `sub`, `shift`, and `or` operations on two 1024-bit warp-unit registers. The locality checker consists of a `sub` logic for the opcode-field-length inputs, and an `and` logic that extracts the offset field value.

Chapter 6

Conclusions

Over the past decade, GPUs have continued to grow in terms of performance and size. The number of execution units has been steadily increasing, which in turn increases the number of concurrent thread contexts needed to keep these units utilized. In order to support fast context switching between large groups of active threads, GPUs invest in large register files to allow each thread to maintain its context. The Register File (RF) is a critical structure in GPUs responsible for a large portion of the area and power. In this dissertation, we sought to address some of main register allocation challenges by designing synergistic compiler/microarchitecture techniques to enable high-performance and energy efficient GPUs.

We first introduced the concept of register coalescing. We proposed CORF, a coalescing-aware register file design for GPUs that simultaneously reduces the leakage and dynamic access power, while improving the overall performance of the GPU. CORF achieves these properties by enabling the reads to multiple operands that are packed together to be coalesced, reducing the number of reads to the RF, and improving dynamic energy and performance. CORF combines compiler-assisted register allocation with a re-organized register file (CORF++) in order to maximize operand coalescing opportunities. Specifically, the new register file organization allows operands to be coalesced even if they reside in different physical registers, provided they reside in non-overlapping sub-banks.

In BOW, we observed that register values are reused repeatedly in close proximity in GPU workloads. We exploit this behavior to forward data directly among nearby instructions, thereby shielding the power-hungry and port-limited register file from many accesses (59% of accesses with an instruction window size of 3). Our best design (BOW-WR) can bypass both read and write operands, and leverages compiler hints to optimally select write-back operand target.

In RegMutex, we noted that static and exclusive register allocation on GPUs leads to register file under-utilization. We addressed this challenge by time-multiplexing the register use between warps. On the compiler side, RegMutex divides the architected register set into a base register set and an extended register set, and by analyzing the program, injects instructions in the kernel code where the extended register set activates and deactivates. On the microarchitectural side, while physical registers are allocated to the base architected registers for the lifetime of the kernel, RegMutex takes a communal approach on allocating physical registers to the extended architected register set. Using the information provided by the compiler, the warp acquires the physical registers for extended architected registers from a shared register pool when needed, and releases them to the shared pool upon deactivation of the extended register set. We showed that this approach enhances the performance of GPU kernels exhibiting a limited occupancy due to high register pressure, and allows application resilience when underlying microarchitecture employs a smaller register file.

Finally, in LARF, instead of modifying software or over-provisioning on-chip memory size, we proposed a data sharing mechanism among warps within the existing register file space. By exploiting the inherent data sharing feature of big data workloads, our proposed locality-aware register file and warp scheduler effectively reduce off-chip memory accesses. Our proposed design showed up to3.5×speedup and 80% global memory access reduction for the data-intensive kernels.

Bibliography

[1] Cuda computing sdk 4.2. `https://developer.nvidia.com/cuda-toolkit-42-archive`. accessed: 2017-08-11.

[2] *nvdisasm* cuda binary tool. `http://docs.nvidia.com/cuda/cuda-binary-utilities/#nvdisasm`. Accessed: 2017-08-11.

[3] Nvidia tesla v100 gpu architecture whitepaper. `http://www.nvidia.com/object/volta-architecture-whitepaper.html`. accessed: 2017-08-11.

[4] `https://github.com/gpgpu-sim/ispass2009-benchmarks`, 2009.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[6] 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)*, 2017.

[7] Mohammad Abdel-Majeed and Murali Annavaram. Warped register file: A power efficient register file for gpgpus. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 412–423. IEEE, 2013.

[8] Mohammad Abdel-Majeed, Hyeran Jeon, Alireza Shafaei, Massoud Pedram, and Murali Annavaram. Pilot Register File: Energy Efficient Partitioned Register File for GPUs. In *Proceedings of IEEE Symposium on High Performance Computer Architecture*. IEEE, 2017.

[9] Mohammad Abdel-Majeed, Daniel Wong, and Murali Annavaram. Warped gates: Gating aware scheduling and power gating for gpgpus. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, 2013.

[10] Mohammad Abdel-Majeed, Daniel Wong, Justin Kuang, and Murali Annavaram. Origami: Folding warps for energy efficient gpus. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, 2016.

[11] Amirali Abdolrashidi, Hodjat Asghari Esfeden, Ali Jahanshahi, Kaustubh Singh, Nael Abu-Ghazaleh, and Daniel Wong. Blockmaestro: Enabling programmer-transparent task-based execution in gpu systems. In *2021 48th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA)*. IEEE, 2021.

[12] AmirAli Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi N. Bhuyan, and Daniel Wong. WIREFRAME: Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs . In *MICRO '17: Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[13] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, 2004.

[14] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing. In *ISCA*, pages 1–13, 2016.

[15] Hodjat Asghari Esfeden, Farzad Khorasani, Hyeran Jeon, Daniel Wong, and Nael 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*, pages 701–714. ACM, 2019.

[16] José L. Ayala, Alexander Veidenbaum, and Marisa López-Vallejo. Power-aware compilation for register file energy reduction. *International Journal of Parallel Programming*, 31(6), 2003.

[17] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[18] Mohammad Bakhshalipour, Aydin Faraji, Seyed Armin Vakil Ghahani, Farid Samandi, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Reducing Writebacks Through In-Cache Displacement. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(2):16, 2019.

[19] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, Abbas Mazloumi, Farid Samandi, Mahmood Naderan, Mehdi Modarressi, and Hamid Sarbazi-Azad. Fast Data Delivery for Many-Core Processors. *IEEE Transactions on Computers (TC)*, 67(10):1416–1429, 2018.

[20] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Domino Temporal Data Prefetcher. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 131–142. IEEE, 2018.

[21] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo Spatial Data Prefetcher. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[22] Mohammad Bakhshalipour, Seyedali Tabaeiaghdaei, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Evaluation of hardware data prefetchers on server processors. *ACM Computing Surveys (CSUR)*, 52(3):1–29, 2019.

[23] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pages 237–248. IEEE, 2001.

[24] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017.

[25] Deniz Balkan, Joseph Sharkey, Dmitry Ponomarev, and Kanad Ghose. Spartan: speculative avoidance of register allocations to transient values for performance and energy efficiency. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 265–274, 2006.

[26] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *SC*, 2011.

[27] Michael Bauer, Sean Treichler, and Alex Aiken. Singe: Leveraging warp specialization for high performance on gpus. In *PPoPP*, 2014.

[28] Etienne Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1884–1896. Society for Industrial and Applied Mathematics, 2013.

[29] Preston Briggs. Register allocation via graph coloring. Technical report, 1992.

[30] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally. Architecting an energy-efficient dram system for gpus. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[31] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[32] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *ISCA*, June 2016.

[33] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. In *arXiv:1410.0759*, 2014.

[34] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. In *Conference on Neural Information Processing Systems (NIPS)*, Long Beach, CA, USA, Sep 2016.

[35] B. Coon and J. Lindholm. System and method for grouping execution threads, July 21 2007.

[36] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[37] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. In *arXiv:1602.02830*, 2016.

[38] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P Topham. Multiple-banked register file architectures. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 316–325. ACM, 2000.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Computer Vision and Pattern Recognition*, 2009.

[40] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent RNNs: Stashing Weights On-Chip. In *ICLR*, May 2016.

[41] Shi Dong and David Kaeli. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. In *ACM General Purpose GPUs (GPGPU-10)*, pages 63–72, New York, NY, USA, 2017.

[42] Sindhuja Gopalakrishnan Elango. Convolutional Neural Network Acceleration on GPU by Exploiting Data Reuse. In *Master's thesis at San Jose State University (SJSU)*, 2017.

[43] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 304–315. IEEE Computer Society, 2004.

[44] Hodjat Asghari Esfeden, Amirali Abdolrashidi, Shafiur Rahman, Daniel Wong, and Nael Abu-Ghazaleh. Bow: Breathing operand windows to exploit bypassing in gpus. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 996–1008. IEEE, 2020.

[45] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 235–246. ACM, 2011.

[46] Mark Gebhart, Stephen W Keckler, and William J Dally. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 465–476. ACM, 2011.

[47] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, May 1996.

[48] Felix A. Gers, Jurgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. In *Neural Computation*, pages 2451—-2471, 2000.

[49] Syed Zohaib Gilani, Nam Sung Kim, and Michael J Schulte. Power-efficient computing for compute-intensive gpgpu applications. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 330–341. IEEE, 2013.

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

[51] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. From model to FPGA: Software-hardware co-design for efficient neural network acceleration. In *Proceedings of IEEE Hot Chips Symposium.* IEEE, 2016.

[52] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented Approximation of Convolutional Neural Networks. In *Proceedings of International Conference on Learning Representations (ICLR)*, May 2016.

[53] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*, pages 243–254, 2016.

[54] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Proceedings of International Conference on Learning Representations*, 2016.

[55] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, June 2016.

[56] Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Chang-Hong Hsu, Michael A. Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2017.

[57] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. In *arXiv:1704.04861*, 2017.

[58] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5MB model size. In *arXiv:1602.07360*, 2016.

[59] Vishwesh Jatala, Jayvant Anantpur, and Amey Karkare. Improving gpu performance through resource sharing. In *HPDC*, 2016.

[60] Nuwan Jayasena, Mattan Erez, Jung Ho Ahn, and William J. Dally. Stream register files with indexed access. In *HPCA*, 2004.

[61] Hyeran Jeon, Hodjat Asghari Esfeden, Nael B Abu-Ghazaleh, Daniel Wong, and Sindhuja Elango. Locality-aware gpu register file. *IEEE Computer Architecture Letters*, 18(2):153–156, 2019.

[62] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. Gpu register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 420–432. ACM, 2015.

[63] N. Jing, J. Wang, F. Fan, W. Yu, L. Jiang, C. Li, and X. Liang. Cache-emulated register file: An integrated on-chip memory architecture for high performance gpgpus. In *MICRO*, 2016.

[64] T. M. Jones, M. F. R. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin. Compiler directed early register release. In *PACT*, 2005.

[65] Timothy M Jones, Michael FP O'Boyle, Jaume Abella, Antonio González, and Oğuz Ergin. Energy-efficient register caching with compiler assistance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(4):13, 2009.

[66] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *MICRO*, 1998.

[67] Deepak Kadetotad, Sairam Arunachalam, Chaitali Chakrabarti, and Jae sun Seo. Efficient Memory Compression in Deep Neural Networks Using Coarse-grain Sparsification for Speech Applications. In *ICCAD*, pages 78:1–78:8, 2016.

[68] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. Tango: A deep neural network benchmark suite for various accelerators. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Press, 2019.

[69] Mehmet Kayaalp, Khaled N Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[70] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. μc-states: Fine-grained gpu datapath power management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, 2016.

[71] Mahmoud Khairy, Jain Akshay, Tor Aamodt, and Timothy G Rogers. Exploring modern gpu memory system design challenges through accurate modeling. *arXiv preprint arXiv:1810.07269*, 2018.

[72] Farzad Khorasani, Hodjat Asghari Esfeden, Nael Abu-Ghazaleh, and Vivek Sarkar. In-register parameter caching for dynamic neural nets with virtual persistent processor specialization. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 377–389. IEEE, 2018.

[73] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. Regmutex: Inter-warp gpu register time-sharing. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 816–828. IEEE Press, 2018.

[74] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Efficient warp execution in presence of divergence with collaborative context collection. In *MICRO*, 2015.

[75] Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, and Wen-mei Hwu. Performance analysis and tuning for general purpose graphics processing units (gpgpu). *Synthesis Lectures on Computer Architecture*, 7(2):1–96, 2012.

[76] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram. Warped-preexecution: A gpu pre-execution approach for improving latency hiding. In *HPCA*, 2016.

[77] Keunsoo Kim and Won Woo Ro. WIR: Warp Instruction Reuse to Minimize Repeated Computations in GPUs. In *HPCA*, 2018.

[78] John Kloosterman, Jonathan Beaumont, D Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. Regless: just-in-time operand staging for gpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–164. ACM, 2017.

[79] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Conference on Neural Information Processing Systems (NIPS)*, Lake Tahoe, NV, USA, Dec 2012.

[80] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. Warped-compression: Enabling power efficient gpus through register compression. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 502–514. ACM, 2015.

[81] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. Gpuwattch: enabling energy optimizations in gpgpus. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 487–498. ACM, 2013.

[82] John Erik Lindholm, Ming Y Siu, Simon S Moy, Samuel Liu, and John R Nickolls. Simulating multiported memories using lower port count memories, March 4 2008. US Patent 7,339,592.

[83] 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)*, 2017.

[84] Zhenhong Liu, Daniel Wong, and Nam Sung Kim. Load-triggered warp approximation on gpu. In *Proceedings of the 2018 International Symposium on Low Power Electronics and Design*, ISLPED '18, 2018.

[85] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9), 1999.

[86] Jack L Lo, Sujay S Parekh, Susan J Eggers, Henry M Levy, and Dean M Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):922–933, 1999.

[87] Luis A Lozano and Guang R Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 292–302. IEEE Computer Society Press, 1995.

[88] A. Majumdar, L. Piga, I. Paul, J. L. Greathouse, W. Huang, and D. H. Albonesi. Dynamic gpgpu power management using adaptive model predictive control. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[89] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *MICRO*, 1997.

[90] J. F. Martinez, J. Renau, M. C. Huang, and M. Prvulovic. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO*, 2002.

[91] Sparsh Mittal and Jeffrey S. Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Comput. Surv.*, 47(2):19:1–19:23, August 2014.

[92] Sparsh Mittal and Jeffrey S. Vetter. A survey of architectural approaches for data compression in cache and main memory systems. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1524–1536, May 2016.

[93] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *MICRO*, 1993.

[94] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 29–, 2003.

[95] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-performance Microprocessor. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2003.

[96] D. Kaeli N. Farazmand, R. Ubal. Statistical Fault Injection-Based AVF Analysis of a GPU Architecure. In *The IEEE Workshop on Silicon Errors in Logic - System Effect (SELSE)*, March 2012.

[97] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, and Hamid Sarbazi-Azad. Neda: Supporting direct inter-core neighbor data exchange in gpus. *IEEE Computer Architecture Letters*, 17(2):225–229, 2018.

[98] Peter R Nuth and William J Dally. The named-state register file: Implementation and performance. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 4–13. IEEE, 1995.

[99] NVIDIA. Cuda toolkit. `https://developer.nvidia.com/cuda-toolkit`, 2007. Accessed: 2018-04-11.

[100] Nvidia. "Whitepaper: Nvidia's Next Generation CUDA Compute Architecture: Fermi", 2009.

[101] Nvidia. Nvidia cuda sdk 2.3. [Online]. Available: `http://developer.nvidia.com/cuda-toolkit-23-downloads`, 2009.

[102] Nvidia. Cuda programming guide, 2010.

[103] Nvidia. "Whitepaper: Nvidia's Next Generation CUDA Compute Architecture: KeplerGK110", 2012.

[104] NVIDIA. Nvidia tesla v100 gpu architecture. `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017. Accessed: 2018-11-26.

[105] Yunho Oh, Myung Kuk Yoon, William J Song, and Won Woo Ro. Finereg: Fine-grained register file management for augmenting gpu throughput. In *2018 51st Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 364–376. IEEE, 2018.

[106] Il Park, Michael D Powell, and TN Vijaykumar. Reducing register ports for higher speed and lower energy. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.*, pages 171–182. IEEE, 2002.

[107] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, July 2004.

[108] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. A case for toggle-aware compression for gpu systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[109] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd 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*, PACT '12, 2012.

[110] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[111] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel, Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *Proceedings of The International Symposium on Computer Architecture (ISCA)*, June 2016.

[112] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2016.

[113] H. Roth, L. Lu, J. Liu, J. Yao, A. Seff, K. M. Cherry, E. Turkbey, and R. Summers. Improving computer-aided detection using convolutional neural networks and random view aggregation. In *IEEE Trans. on Medical Imaging*, 2016.

[114] Mohammad Sadrosadati, Seyed Borna Ehsani, Hajar Falahati, Rachata Ausavarungnirun, Arash Tavakkol, Mojtaba Abaee, Lois Orosa, Yaohua Wang, Hamid Sarbazi-Azad, and Onur Mutlu. Itap: Idle-time-aware power management for gpu execution units. *ACM TACO*, 2018.

[115] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 489–502. ACM, 2018.

[116] M. H. Santriaji and H. Hoffmann. Grape: Minimizing energy for gpu applications with performance requirements. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[117] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[118] A. Sethia and S. Mahlke. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[119] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. Memzip: Exploring unconventional benefits from memory compression. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[120] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*, June 2018.

[121] Premkishore Shivakumar and Norman P Jouppi. Cacti 3.0: An integrated cache timing, power, and area model, 2001. Technical Report 2001/2, Compaq Computer Corporation.

[122] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks For Large-scale Image Recognition. In *International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015.

[123] Vilas Sridharan and David R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *IEEE International Symposium on High Performance Computer Architecture*, 2009.

[124] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[125] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[126] John A Swensen and Yale N Patt. Hierarchical registers for scientific computers. In *Proceedings of the 2nd international conference on Supercomputing*, pages 346–354. ACM, 1988.

[127] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA, June 2015.

[128] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast Implementation of DGEMM on Fermi GPU. In *SC*, pages 35:1–35:11, 2011.

[129] J. Tan and X. Fu. Mitigating the susceptibility of gpgpus register file to process variations. In *IPDPS*, 2015.

[130] J. Tan, S. L. Song, K. Yan, X. Fu, A. Marquez, and D. Kerbyson. Combating the reliability challenge of gpu register file at low supply voltage. In *PACT*, 2016.

[131] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 624–637. IEEE, 2018.

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

[133] Devashree Tripathy, Hadi Zamani, Debiprasanna Sahoo, Laxmi N Bhuyan, and Manoranjan Satpathy. Slumber: static-power management for gpgpu register files. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 109–114, 2020.

[134] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, 2017.

[135] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. Zorua: A holistic approach to resource virtualization in gpus. In *MICRO*, 2016.

[136] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. A case for core-assisted bottleneck acceleration in gpus: Enabling flexible data compression with assist warps. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[137] Dani Voitsechov, Arslan Zulfiqar, Mark Stephenson, Mark Gebhart, and Stephen W Keckler. Software-directed techniques for improved gpu register file utilization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):38, 2018.

[138] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, 2010.

[139] Xin Wang and Wei Zhang. Gpu register packing: Dynamically exploiting narrow-width operands to improve performance. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 745–752. IEEE, 2017.

[140] Daniel Wong, Nam S. Kim, and Murali Annavaram. Approximating warps with intra-warp operand value similarity. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[141] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. Gpgpu performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[142] Qiumin Xu and Murali Annavaram. Pattern aware scheduling and power gating for gpgpus. In *Parallel Architectures and Compilation Techniques (PACT), 2014 23nd International Conference on*, 2014.

[143] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture*. IEEE, 2016.

[144] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, 2000.

[145] Zahra Yarahmadi. *Study of the Bipartite Edge Frustration of Graphs*, pages 249–267. Springer International Publishing, Cham, 2016.

[146] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram. Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit. In *ISCA*, 2016.

[147] Hui Zeng and Kanad Ghose. Register file caching for energy efficiency. In *Low Power Electronics and Design, 2006. ISLPED'06. Proceedings of the 2006 International Symposium on*, pages 244–249. IEEE, 2006.

[148] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, 2000.

[149] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. TBD: Benchmarking and Analyzing Deep Neural Network Training. In *arXiv:1803.06905*, 2018.